DEBUGGER

Prepared by: Julio Arturo Benitez, Alexander Bonilla

TABLE OF CONTENTS

INTRODUCTION

The reason this document was created is to have proper documentation for our “Debugger” game, an educational MMO (massively multiplayer online [game]) that will help CSC210 (introduction to C++ programming) students from San Francisco State University understand the key concepts of computer programming, quizzing them throughout the game about key concepts of C++ programming as they experience the Debugger world. Students will be able to chat during the game progress, add friends, fight bugs in order to gain experience, scroll through their inventory, and wrap between worlds. Debugger was created during the multiplayer game development class (CSC 631) at San Francisco State University taught by professor Ilmi Yoon during the Fall semester of 2009. First and foremost, one has to describe the tools used and the steps taken in order to create the game.

Our class, composed of approximately 25 students, was separated into 7 different groups: server, client, art, database, launching, content, and concept design. Each group had specific responsibilities in order to make the game content come to life. During group distribution, I was placed inside the client team; the coding responsibilities of this team were heavy, as we had the responsibility to code the entire feel of the game, as well as camera control, character movement, chat implementation, register and log-in screens, friends and inventory list, among others. The team consisted of seven team members: Jason (group leader), Vivek, Gary, Nick, Yi, Alex, Tomi and Julio (myself), from here, subgroups were created in order to implement different parts of the game. It is important to explain to the reader that this project was not started from scratch, our class was fortunate enough to work with existing code from past classes from a previous game called NurseTown, a similar game that was created for nursing students in order to aid them in their nursing studies.

The client side of the game was implemented using the python programming language and the NetBeans (www.netbeans.org) IDE, although some class members chose to implement using the Eclispe IDE (www.eclipse.org). For the game engine, Panda3D was used, which can be downloaded for free at www.panda3d.org. If you choose to implement the game in NetBeans, please note that there are extra steps that will be necessary in order to get the game to run, these steps are the installation of the python programming language plug-in and setting up the right Panda3d paths in order for Panda3D to launch. This will be showed to the reader in the next section of this documentation. Also, a subversion client, (hosted @ thecity.sfsu.edu) was used in order to turn in all modifications of the code. Lastly, a log was kept in order to document our progress, thought, issues, next steps and problems using Google docs. This log was updated once or twice a week, depending on the work load that was assigned for that week.

When reading this document, please have in mind that everyone in the multiplayer game development class wrote the information for their own contribution to the game, in other words, the flow of the document might not be the best and some parts might seem a bit confusing, I have done my best effort to integrate all of the documentation together in the two weeks that were provided.

Any questions that the reader may have about the contents of this document or about the code that was written by me or any of my team members can be asked directly to me through email at jbenitez@sfsu.edu, I will try my best to answer any questions that reader may have, and if for some reason I am unable to answer your question, I will try my best to contact the author of the code that the reader might have a particular question about.

1A.GAME OVERVIEW

The "deBugger" program provides an interactive environment for users to learn basic to standard software development concepts. This program is a massively multiplayer online game with many similar game characteristics to popular MMORPGs (such as games like World of Warcraft, Guild Wars, and other popular MMO games) but with a smaller scope in mind. The “deBugger” game, like many popular MMO games, places its users in a persistent virtual world where they can interact with other players in real time. This game also offers the player the opportunity to explore environments such as the inside of a computer. These environments will give users the chance to take on and defeat computer “bugs.” The user will then be able to defeat these bugs, an act of which is called “debugging”, where this game explicably gets its name from.

This program uses a virtual online game interface where users will be able to answer verified, academic, programming related questions. With regular and consistent use, players of the "deBugger" game will be exposed to an ever increasing compilation of educational problems to help expand their knowledge and learn programming subject material related to their classes; a primary portion of which will be from San Francisco State University CSC 210‘s C++ programming material. Users of this software will be able to communicate with each other through the game‘s chatting module, providing a social environment where they may ask each other for guidance or be able to compete with one another (to be implemented in future versions). This social aspect adds an extra dimension to most educational games since players can seek support and it may add extra motivation as well as competition to continue playing and learn while advancing in the game. The MMO game model was chosen since it is very useful in keeping players interested in a game, as has been the trend with many popular MMORPGs such as World of Warcraft which is able to keep millions of subscribed users to regularly log in to its game and continue playing through its content. This is very essential to the games application as many educational games tend to fail to keep players interested over a long period of time due to lack of social environment and lack of new content.

This game will offer users the ability to increase the difficulty of questions they will be asked based on their progress. Their achievements will be based on the correct number of questions answered in the game world and through competitive play. With more achievements, more items and abilities will be unlocked to the player. The player can also continue playing to access more difficult questions and test their knowledge, as well as gain abilities and items they may not have had access to earlier in the game.

In order to provide all the different MMORPG aspects that “deBugger” offers, several different components were implemented. The user directly interacts with the client component, which is installed on the user’s computer. The client takes care of tasks such as providing the player with logging in and menus, displaying the virtual world, allowing the user to move around the world, allowing the user to send messages to other players, allowing the user to fight bugs, and other tasks. While the user is playing, the game client sends information to the game server, which is hosted and maintained remotely. The server takes care of managing the data the user provides through interaction, such as moving around the world and allowing other players to see those movements, receiving and sending chat information to and from other users, deploying bugs, spawning items, and other tasks. The server also acts as a mediator not only between different users but also between users and the database. The database component stores different user specific data such as passwords, items, friends, gold, etc. The questions used by the bugs are also stored online through a database. The bugs are maintained through a bug server dedicated to spawning and controlling the bugs. The questions are submitted through a website developed and maintained through the “deBugger” project as well. All these different components, and more, make up the complete “deBugger” experience.

"deBugger" was developed in a collaborative effort by the students enrolled in CSC 631 and CSC 831 at San Francisco State University during the Fall 2009 semester. This team was guided under the supervision of Professor Ilmi Yoon. "Nursetown", an educational online game developed by San Francisco State University students, was used as reference material during the development of "deBugger". This game was developed as an applied practice for students to be able to create software among varying different modules (such as a client side, server side, database side, etc) and collaborate together in order to create synergy amongst those modules to create a functional online game with content. To separate tasks, a collection of teams focused on different modules were created: game concept team, game launching team, testing team, database team, server team, content team, art development team, and client development team. This software was also developed with extensibility in mind, as the strength of many online games comes with the ability to add content in order to keep users interested.

Throughout the development and production processes, students were motivated to provide updates and keep in touch with other members of the class. Each individual student kept a weekly log of tasks, assignments and accomplishments which was hosted online. Through these logs, their respective team leads and their instructor were able to check in with any progress and issues individuals were having throughout the course of development and production. Milestones were also put into place throughout the course of progress. These milestones contained content agreed upon by the class and the instructor. Through these milestones, the class as a whole was able to check in with the overall progress of the game and could consequently increase or decrease the scope of the end product.

This document contains records of conceptual ideas and development details of the work put into the production of this game. These are the efforts of creating a software application that provides an enjoyable, challenging, social, and educational tool for students.

1B. INTROCTION TO GAME CONCEPT

As the Game Concept Team, we have created, having in mind the characteristics of online multiplayer games, a potential conceptual foundation for the game to implement with the help of the other teams from the 631 class.

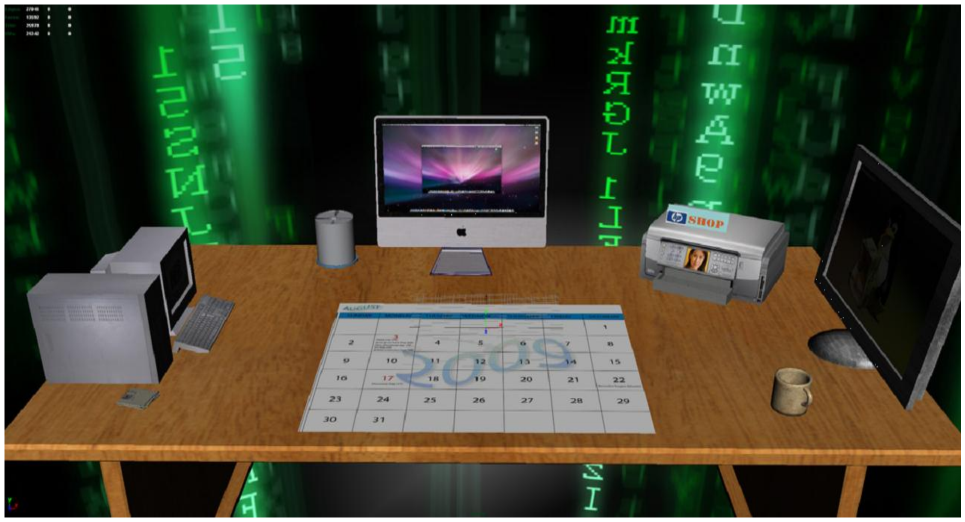
As a main foundational concept, it was very important to keep in mind that, just like in any other MMO games out there, the player who has played the game for a good set of hours, should always have the urge to come back and play more. To assure that, the game will consists of several solid and functional growth systems. For example, a new player initially does not even have access to 2/3rd of the game right from the start. The player would need to successfully complete the 'Easy' mode to be able to pass on to 'Moderate' and 'Hard'. That itself alone will give a player enough motivation to keep playing and at least try to get to Hard mode. Also the player might slowly become bored at always killing the enemy at the same pace and with the same effort. Thus, as the player plays, he/she will gain more boosts, be able to change appearance and attributes, be stronger and faster, and eventually begin to enjoy the game more and more.

Nevertheless, even though the game is an MMORPG game, with a foundation as similar to commercial games currently available today, unlike the usual theme, this game has an essence of educational purposes. Since this is a game created for computer science students, by computer science students, it is a top priority for players to use their knowledge to advance through the game, and in return learn more about computer science related topics and concepts while playing the game. It is important for not only the player with the most skills or highest level to be the best, but also the player with the most knowledge about computer science. This has always been a crucial concept to keep in mind while putting together the game.

Another factor of creating a legit MMO feeling is the social atmosphere and interactions. The game is chatting capable, so the players will be able to talk and ask each other questions, and socialize. Playing a game within a virtual society also causes the players to be more aware of their physical looks within the game. Everyone begins to want to look 'better' or 'different' then the other players, or contain an item or an article of clothing that is very rare or flashy. All these concepts have also been factored in, as you will read later on within the documentation.

Just like in any other MMORPG game, the player will need to create an account and have an avatar assigned to him/her to begin experiencing the game. Once the player installs and runs the game, he or she will be prompted with a login window, where the player would put the username and password information to log-in and start playing the game. If the player does not have an account already, then there will be an option to create one. Since this is not just a regular, but an educational game, it would be ideal to be able to keep track of student information. Then, the user will be able to choose a specific type of avatar to begin the game and evolve with.

Once the player logs into his/her account, they will spawn in a what is called a 'Global Map', which consists of three computers to choose from, of which each represents each difficulty of the game(Easy=Old looking, Moderate=Current technology, Hard=Futuristic). Each computer's map layouts are drastically different, but it will have the same basic theme.



(fig 1a.1 The 'Desktop' which contains the 3 different types of computers)

Once a computer is chosen, It will start the player out on the 'desktop', where there is an icon to begin/start a board game, and a 'my computer' icon to enter the computer and begin debugging(the hunting).

The motherboard architecture is very complex, so there will be more abstract rooms that follow easier concepts for the target players. The importance is that there is included the memory rooms, where players can move through room 'arrays' and continue to fight bugs for as long as they want.

Also in regards of the In-Game chatting system, it will be quite simple. When the player first logs in, as explained earlier, the avatar will appear in the Global Map ready to enter one of three terminals. The chatting in this Global room will be only to those in this room, which could be any player outside the computers and could be of any rank and level.

The Global Map is like global chat, and when players the a computer, only those who are also inside the computer can speak with each other. All of the three computers will have this capability, allowing chat everywhere with the people the player needs or wants to speak.

As for the avatar, here is the description for a general avatar growth and customization system. We will talk about it more in depth later on within the documentation.

There are two ways the user will be able to gain items to customize their looks, and gain certain attributes from. These options are either by defeating a bug or by purchasing the items from the shop using gold/money. When a certain enemy will be defeated, there is a chance that they might drop what is referred to as 'loot', one or a collection of items including money and accessories. There is also the shop, where the user will have access to purchase most of, if not all, the articles of clothing for a certain sum of money. However, players are not allowed to use these items unless they approach a certain given level.

As we all might know, avid MMO players can name one basic concept that is common to nearly all types of games. That is the Battle System. Any popular commercialized MMORPG game has a stable, vast, and exciting Battle System, therefore it was a very important concept to include within our game. Just like the other games, in this game the player will walk around an area that contains enemies and mobs, and the player will engage in a battle with them to gain certain bonuses and advance throughout the game.

Nevertheless, the game will not be only the Bug Hunting, but the board game will be a crucial part of the MMORPG game. The board game directly ties in with the 'Bug Hunting' concept of the game, which will be explained more in detail later in the documentation.

The board game is where a player can either 'join' or 'create' a game, in which several different players will be competing against each other to win by answering right on given questions regarding computer science, and moving through the board with a chance of luck of the dice. Players will compete against each other within a randomly generated map in which each player follows a unique path towards a general end point. The players will walk through the path, which is a collection of tiles, by a random die roll. Once the player reaches to the end, they will be the victor, and they will be awarded with a sum of gold, based upon the difficulty setting of the game.

Each tile will represent a question regarding computer science. With each correct answer given to a question, the player will receive a quantity of Experience Points, which are needed to level up the players character.

As mentioned before, Bug Hunting concept is tied directly with the board game. This will be the more common characteristic of an MMORPG game, which is 'monster' hunting. Since this will be a computer science related game, instead of having monsters, we will be having 'Bugs' that will need to be fixed.

Different events, as in quests and Bugs, will appear on different difficulty areas. Within this game concept, the player will be not gaining experience points, but gold from each defeated 'bug'.

A basic idea that will be implemented is that a certain MAIN quest, per difficulty area, must be completed, for the player to access the next area. Thus, this way for example, the player will not only need to be level 10 to access the Moderate area, but also finish the Easy area's MAIN quest.

1C.INTRODUCTION OF ART SUPPORT TEAM

The main tasks of the ART Support Team (AST) include the visualization of the game concept and the development of the content for the game (game levels, game characters, various static models, etc). When the game concept was finalized, our team had to come up with the visualization of the ideas in the designs of the game levels and the characters. Despite the fact that none of the members of the team was a professional artist, we tried approaching our tasks in the formal way. The very first stage of any of the team's work was brainstorming and after the completion of this step the implementation concept was chosen and only after the actual modeling work was under way.

Since there is no specific level editors for Panda3D game engine the team used Maya and 3DS Max modeling applications for all 3D work. The main reason why these applications were chosen is because the members of our team were familiar with these applications, but the work could've been done in any other 3D modeling application for which Panda3D has conversion utility (for example, Blender). Other tools we used included image editing applications (e.g. Paint, Gimp or Photoshop, etc) for texture editing in 3D models.

Autodesk's Maya application was used for 95% of the modeling work. Panda3D manual has a chapter devoted to the conversion of projects from this application into a .egg format which Panda3D uses. 3DS Max was used for just opening .3ds model files and conversion into the intermediary .fbx or .obj file formats which Maya application could use.

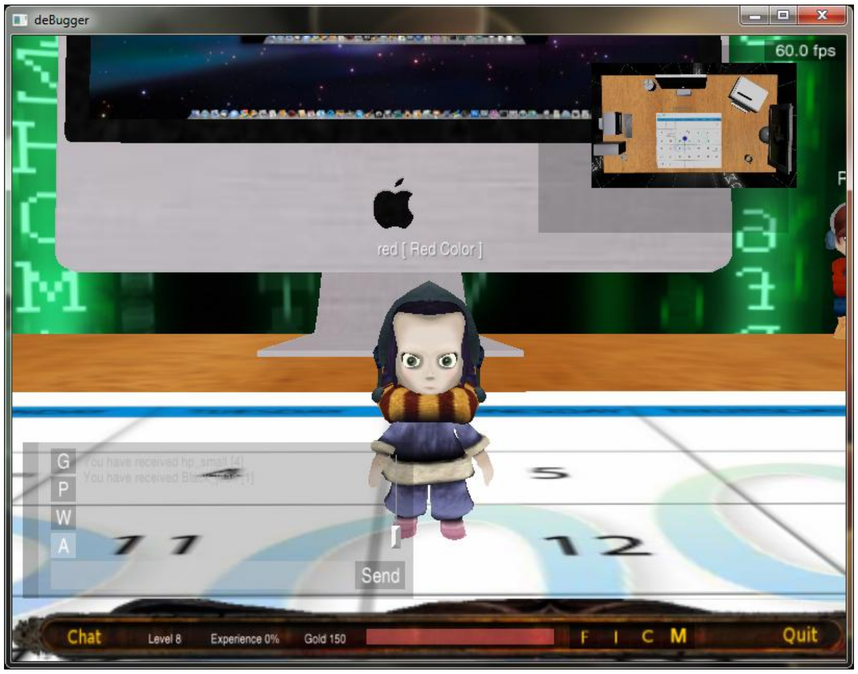
Once the class has settled on the concept for the game, the very first step was to find and collect as many suitable 3D models as we could in order to minimize our own work. For that task we used such sites as www.turbosquid.com, www.creativecrash.com. There are a number of exchange web portals for 3D artists where they sell their work or simply put it up for free; the above sites are just the ones of the more well-known places. Once we collected the models they had to be “cleaned” from the extraneous elements and the materials which Panda3D engine doesn't understand. With that being done the remaining work was modeling, animating of our own models followed by the conversion into .egg file format.

The details of this work are depicted in chapter 4 of this documentation.

1D.INTRODUCTION OF CLIENT TEAM

1.d. Introduction of Client Team

The client team comprises of 9 team members, namingly Alexander, Gary, Jason, Julio, Nicholas, Sara, Tomi, Vivekanand, and Yi. The client team has developed the client application for Debugger, which is programmed in Python, using Panda3D as game engine. This provides graphical interface, I/O operations, and collision detection; and facilitates rendering 3d images, which are important features of Massively multiplayer online role-playing game (MMORPG).



A set of protocols is defined in this application to communicate with the server and fetch live-data from database. When a client establishes connection with server, continuous exchange of messages takes place, which updates the client's graphical interface (GUI) accordingly. This application is built on Nurse Town project and the communication infrastructure is adopted from it.

In this application, the team has introduced several rooms based on "Inside computer architecture" theme. This application provides registration to new players, in which they can select avatar for themselves. It has also introduce some additional error checking when a player tries to login. It also notifies the server when player left the game. A special request will be send to the server 10 times per second to keep the connection alive. Once a player login, player can start at the very first scene where players can socialize with existing players using chatting. Currently, there are different kinds of chat a player can have, naming, global-chat (at global level), public-chat (at room level), and private-chat (1-to-1 chat).

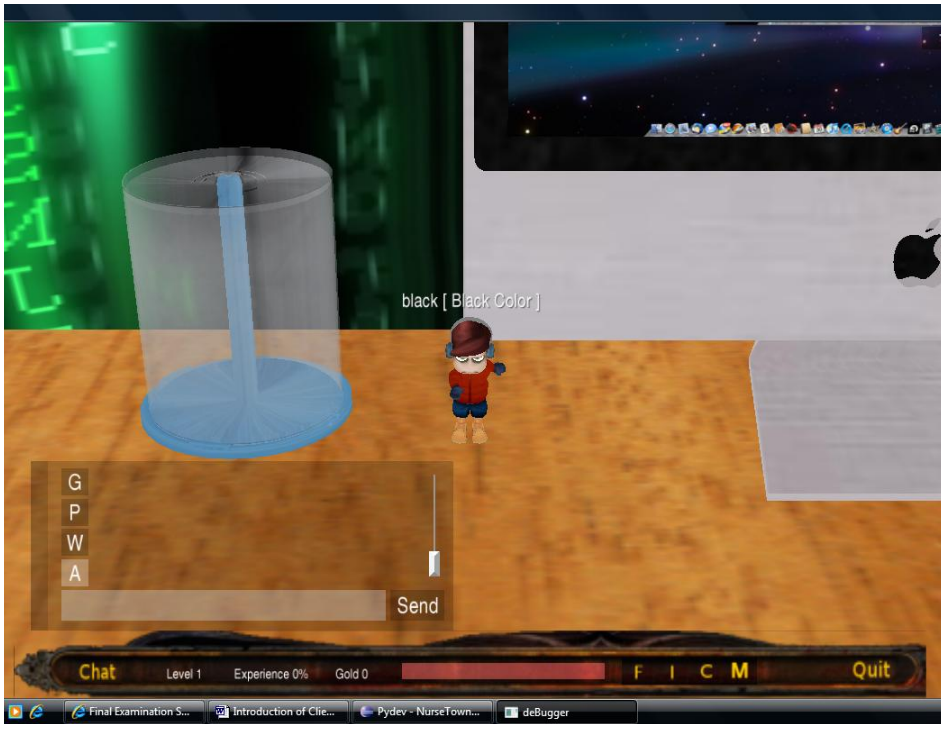


Figure: 1.d.2

Player can walk around the 3D scene and also start exploration into the world of different computer inside worlds. Beginner should start from a world, "easy computer". In each level, player will have a battle with bugs. Player has to defeat all the bugs in existing level to achieve that level and unlock the next level.

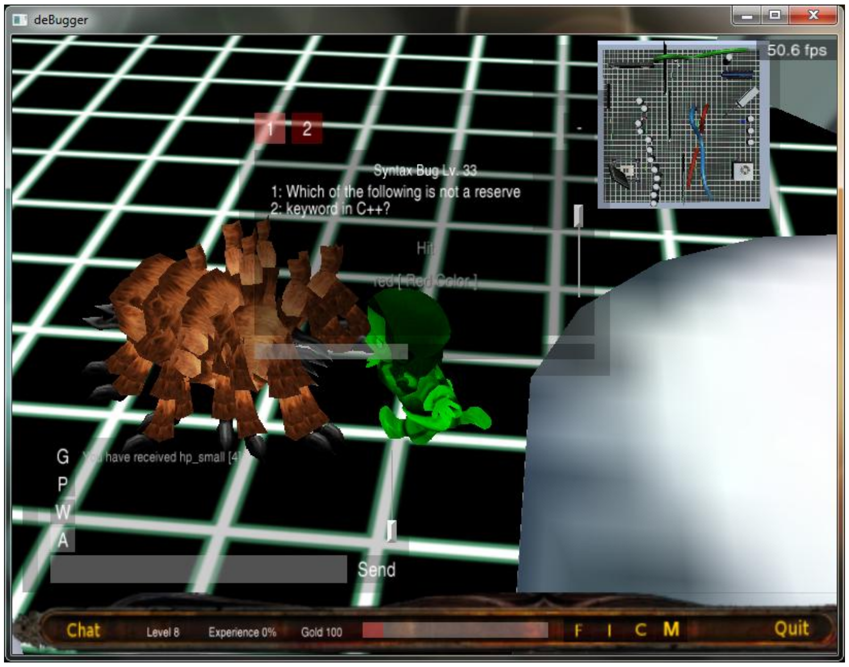


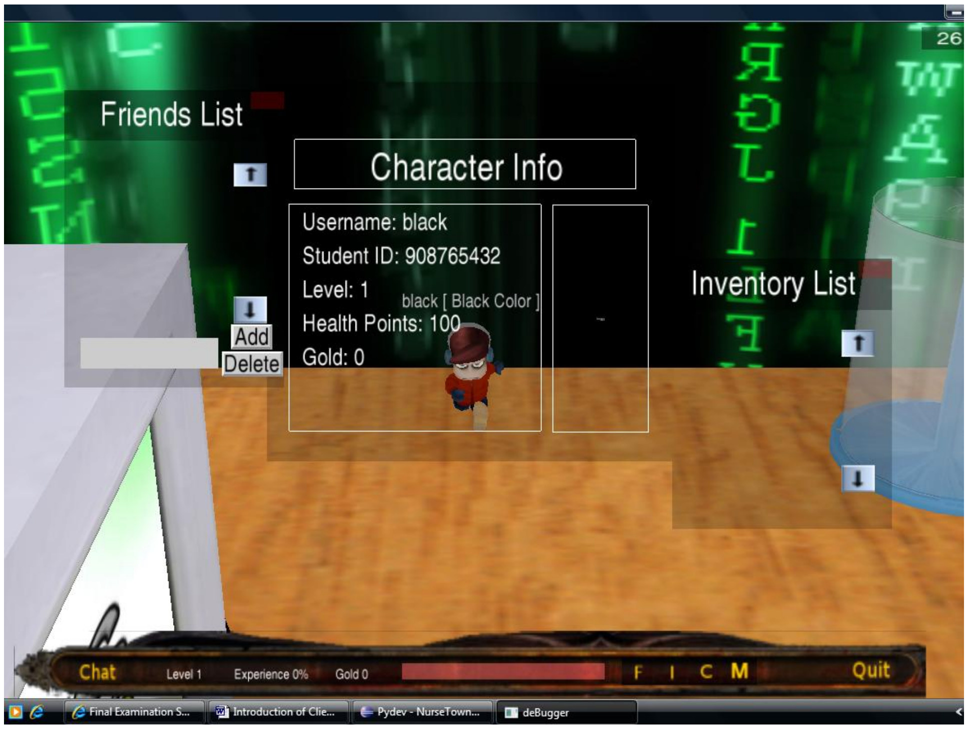
Figure: 1.d.3

There are different kinds of bug in a single level based on CSC-210 concepts, such as syntax, semantics, etc. When a bug encounters a player, it will send a request to the server and server will throw a question to the player based on bug id and level. Player has to answer that question in specified time. If player answer that question incorrectly, then he will lose some health points depending on the difficulty level. If player answer it correctly, then he will earn some gold. Occasionally when a bug is defeated, it will leave an item behind for the player, which increases the player's inventory.

Apart from mouse control, several hotkeys are provided for the player to make this game interactive. Those hotkeys include character movement and control for some addition panel; containing friends list panel, inventory panel, character info panel, chat panel and mini map. All these panels are functional and fetch live-data from the database.

A battle system has been introduced in this application.

A player can also maintain their friends list and add other players as their friends. When a player request friendship with another player, the other player has to accept or deny that friendship request. Player can have private-chat with their friends to discuss the question thrown by bug and defeat that bug. Later on both the players can delete the other player from their friends list. Furthermore, a player can view a list of inventory they own in this game. They can purchase different items from the shop spending gold. Players can spend these items to increase their health level.



Username: black
Student ID: 908765432
Level: 1
Health Points: 100
Gold: 0
black [Black Color]

Level 1 Experience 0% Gold 0

Figure: 1.d.4

A mini-map is provided on right-top corner to help the players see where they are. This min-map shows the entire room so that they can see the location of their opponents and other players as well. When a player moves around in the scene, the mini-map will be updated accordingly.

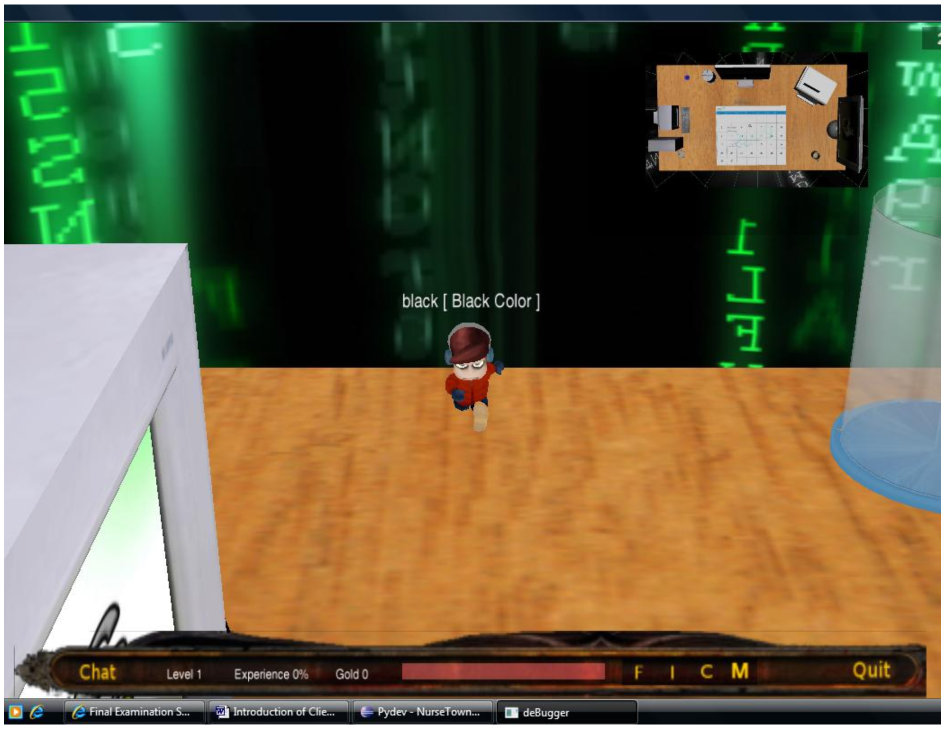


Figure: 1.d.5

1E.INTRO TO SERVER TEAM

The job of the server team was to implement any changes to the from the Nursetown server and extend and implement new features into the Debugger. Whatever was required to be done on server side, the server team had done. The team was small at first because the server code wasn’t being changed very much but over time as the game added new features, a lot of changes to the code needed to be so more people were added to get things done quicker. The first milestone required the server team change the login and registration protocol for the new game client to be able to login and also register new characters in the game. The second milestone is were a lot of changes happened. The server used to look more like Nursetown but after the second milestone, it looked more like our own. One of the biggest changes between the Nursetown server and the Debugger server was that the Debugger had non-player characters or NPC added into them. This was done with a modified game client that was called the bug server. These two separate programs worked together create NPCs for the players in the game to play against. The implementation and how it was achieved as including all the other milestones will be discussed.

1F.INTRODUCTION TO GAME CONTENT TEAM

The game contents are the notable backbone to this educational role-playing game. They should prove to be the reason any player wants to keep learning and playing this game. I wanted to make sure that our educational game could carry out its promise and turn out to entice learning the basics of programming. Common educational games have the playing appeal covering their learning goals. With our game, we wanted a simple board game to accompany a dungeon-esque bug killing game. This combo of slow and fast paced gaming really helps compliment the goal of teaching each player. Coming from a first hand experience, I knew how it felt being The Game's prospective audience. I learned the basics of programming from our current generation's prowess of technology. So having done that, I felt confident in what questions needed to be asked. Most of the questions our class thought out were: true/false, multiple choice, short answer, and 'write output' problems. We had to classify which questions were suitable for the game type a player was in. Questions that needed critical thinking, where several minutes would be allotted to finish, would be better for the board game mode, where movement is fixed. The more concise questions could be used when killing bugs. When a player kills bugs, he/she doesn't have lots of time to sit and answer the question because of their dwindling hit points. Once we established what kind of questions were going to be used, we needed to make a place for our game to access them. The previous game Nursetown had a question system that allowed potential questions to be submitted from a website from players and the game makers. Then, those questions could be profiled to make sure that they are grammatically correct and actually correct. The same thing applies for our game DeBugger. We set up our own question submission website that any student could use. This critical part of new content will shift towards the players, because they will receive gold for submitting new questions. The players / students of the class will make money in the game by contributing to it, and a higher level player or admin will check the question's validity. I made sure that I had the right idea and sat in on a few sessions of the 'intro to cs' classes. Most of what has been entered into the database covers the bases on C++ programming. Arrays, pointers, syntax, errors, and protocol are all reoccurring topics covered by the content. After getting a huge contribution from my Classmates, I have entered 100 or so questions into the website. That's a good start, but there should be hundreds more. There should be more content to fill the spectrum with plenty of detail so as nothing is missed. Currently, the board game is not implemented, and the only playable environment is the easy computer. The questions reflect that, and more complex questions are not in the database. Since this game stands on the questions that are asked, its vital that eventually thousands of questions are in the database. These questions would be divided into fields of difficulty, question type, and question worth. At the end of this experience, we have a made a great foundation to take off from.

1G.INTRODUCTION OF DATA BASE TEAM:

Do you want data? Come to us! Our home is stored in thecity.sfsu. But before we invite you in to our virtual container, we want to give you a brief overview of our purpose in this MMORPG game.

We are a team who keeps track most of the data stored in game. Some examples of what we carry are Username/Password, Inventory, and bugs' information.

The main teams we interact with are the Server Team, the Launching Team, and the Content Team/Web Team. When the Server Team is given a request from the Client, the Server Team will ask us for information. We then give them the requested information like Character's positions or inventory items. Similarly, Launching Team will also request information from us from Quiz Questions, so we will have to stay productive 24/7 to process their requests.

Other than retrieval tasks, other teams may also request to store information. If the Client wants to update the player's inventory, we will need to create another database slot for the user's data entry to store the Client-related items. In addition, when the Launching / Web/ Content teams request to store information like Account/Password and Quiz Questions for game content we will need to create new Quiz-related entries.

In other words, we are basically a passive virtual container who fulfills requests from other teams.

1H. INTRODUCTION OF GAME TESTING TEAM

The game testing team had two primary focuses. One was the implementation of the bug server, which is used to keep track of bug movement within the game world, as well as other features such as attacking and respawning. Secondly, we are to test the playability as well as compatibility of the game using the pre packaged installer that is created.

The bug servers goal is to primarily to offload the work from the server so that the server can work on maintaining information exchange between the various clients and bugs. The bug server does its job by acting like a client and giving some intelligence to otherwise completely random movement of bugs in the game.

Our first task is to program a standalone Bug Client for the Server to "spawn bugs." A "bug" is a monster in our game. Our Bug Client will behave very similarly to the User's Clients because we want to recycle certain programming characteristics, like collision detecting and certain protocols.

When the Server is initiated in thecity.sfsu.edu, the Bug Client will be sitting idle by the Server. When the Server is requested to "spawn bugs" from the Client, the Server will call the Bug Client to "spawn bugs." Then in the User's Client point-of-view, the users will see bugs spawned and running around, so they can see and engage with our bugs. In short, our Bug Client will work very closely with the Server to make the game interesting.

Our second task is to test the overall quality of the game, by testing various functionalities and its stability and detect bugs from the User's Client. We test the game's stability by first testing the number of logins before the game becomes laggy or crash. If the login test is reasonably working, we then test other in-game functionality like adding friend list, changing to different maps, loading bugs, engage with bugs etc. Basically we want to point out anything thing that can improve the overall quality of the game, be it improving efficiency or fixing bugs.

1I.INTRODUCTION TO LAUNCHING TEAM

Massively multiplayer online role-playing game (MMORPG) is a genre of computer role playing games in which players in large numbers interact with one another within a virtual game world. Debugger is a working example of an educational massively multiplayer online (MMO) game that will not only help CSC 210 students from San Francisco State University but would also help those who would like to test their knowledge of C++ programming language.

Debugger gaming project is a collaborative class project consisting of 19 students under the guidance of Dr. Ilmi Yoon and is part of the Fall 2009 curriculum of the CSC 631/831 – Multiplayer Game Development. For developing an MMORPG game requires various resources such as game concept and design, 3D models, game client and server, database, etc. Hence for that matter the whole class was divided into following teams:

- Artwork Support
- Client Development
- Database
- Game Concept Design
- Launching
- Server
- Testing

The launching team which comprises of Alan Nguyen and Vivekanand Rao whose responsibilities includes but not limited to:

- Creating Debugger's game client installation package.
- Developing a website for promoting our MMO game.

Debugger's game client installation package

Debugger makes use of Panda3D – a 3D game engine which includes a library of subroutines for 3D rendering, graphics, I/O, collision detection, and other abilities significant to the development of 3D games. For controlling the Panda3D library, we had the option of writing client application code in either C++ or Python. Panda3D's intended game development language is Python due to which the client team opted for it. However, in order for game players to install and start playing Debugger on a Windows and/or Macintosh machine would require a Python interpreter.

As there is no built in support for Python's interpreter on both Windows and Macintosh operating systems a game player will have to first install Python on his machine before he/she could actually start playing or running the game. To avoid a game player from undergoing the frustration of downloading and installing the correct version of Python's interpreter it is recommended to package the game as an installer. The Debugger's launch team would create a one step Windows self extracting package that would be available for download on Debugger's website thereby allowing players to run the game as an executable Windows program.

Panda3D comes with an included helper application called packpanda, which creates a one file executable that is able to install everything necessary to run our application. The specific command that was used in the creation of the setup file; dependant on the developing environment, was 'packpanda --dir src --name "deBugger - SFSU" --bam –pyc'. I will explain what each option does in order to make it a bit clearer.

--dir is the directory of the location of the Main.py file. This, like C++, is a requirement, so was unable to be changed. Src was the directory under Gaming (our 'project directory'). Inside of it, we had a Launcher.py file that was changed to a Main.py file to follow the packpanda program. We did not want our game to default to main, so I gave it a name of debugger – SFSU. –bam and –pyc are the compiled versions of the egg and py files, respectively.

Debugger's Launch website

Similar to all gaming sites, to have our presence felt on the web, Dr. Yoon recommended for having a website specifically catered towards Debugger. The website was built using HTML, CSS, Javascript and PHP and will be a one-stop information source for the gaming community. Apart from game download, the website will also provide an overview of Debugger, system requirements and installation of the game. In addition, the website will also provide screenshots of players and scenes, game interface along with brief explanation of each section. A list of both mouse and keyboard controls for activating the various functions/features in the game will be listed on the website. The website would also guide game players as to how to go about with the registration and selection of avatar process. For current and future SFSU students and anyone who would like to be part of Debugger team and interested in knowing as to how Debugger is developed can check out the Download's section which provides specifications of the same. Lastly, the website also provides link to the Question Creator application wherein one can enter C++ questions and its answers which will be used by the Debugger application.

Design:

To first design a website, the launching team tried to obtain an overall feel for what a MMO typically contains as a website in order to entice and retain players. A general pull of MMO's turned up with a variety of sites, and while not comprehensive, gives a good indication of the various features that are present on all MMO sites. This list of MMO sites is not comprehensive list, but is representative of the content.

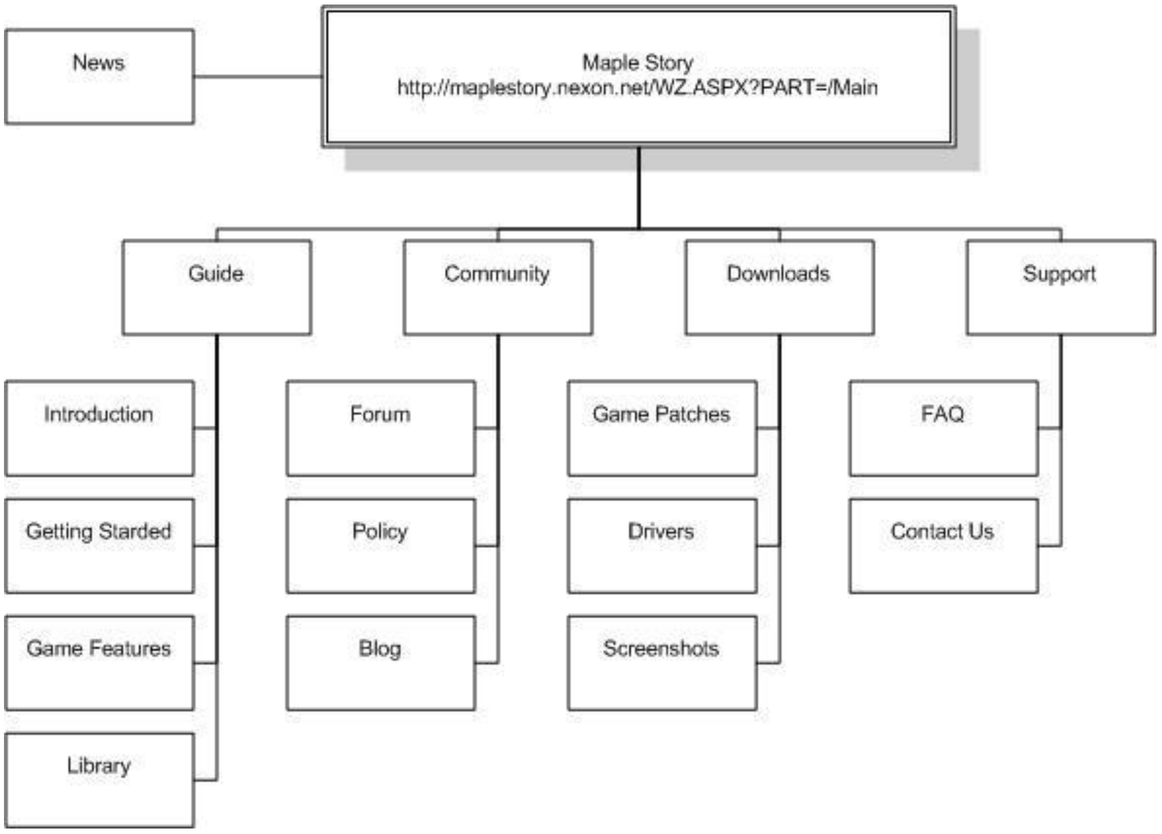
Maple Story

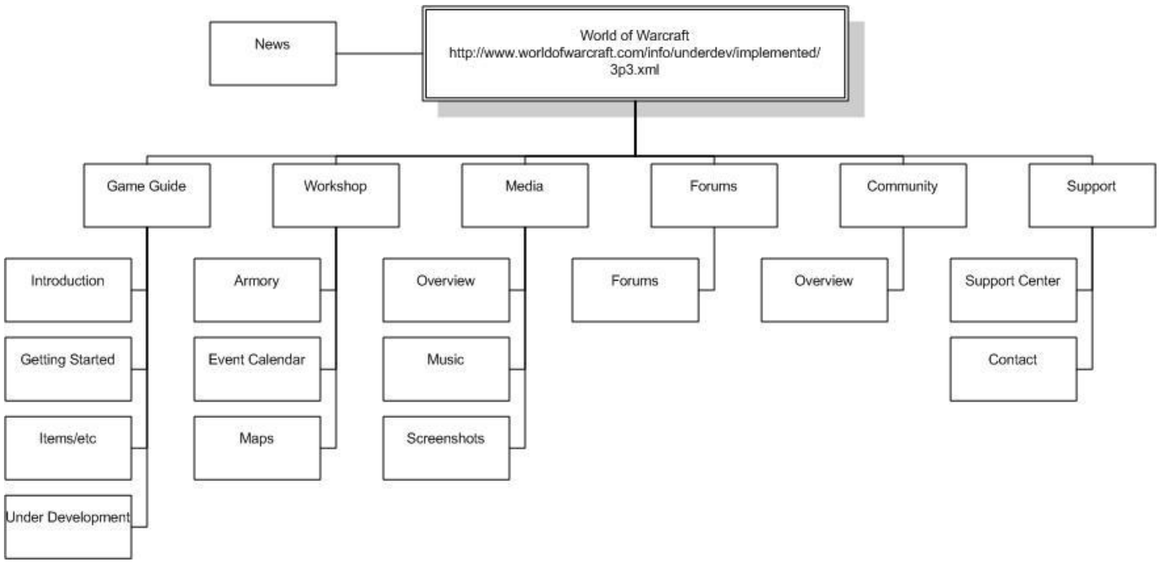
World of Warcraft

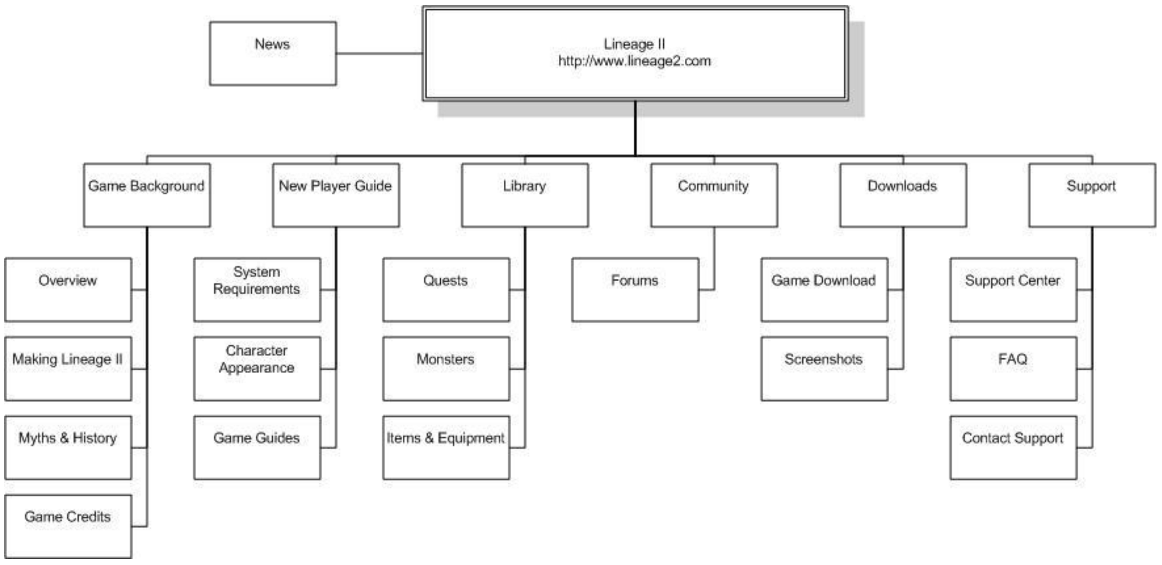
Lineage II

Luna Online

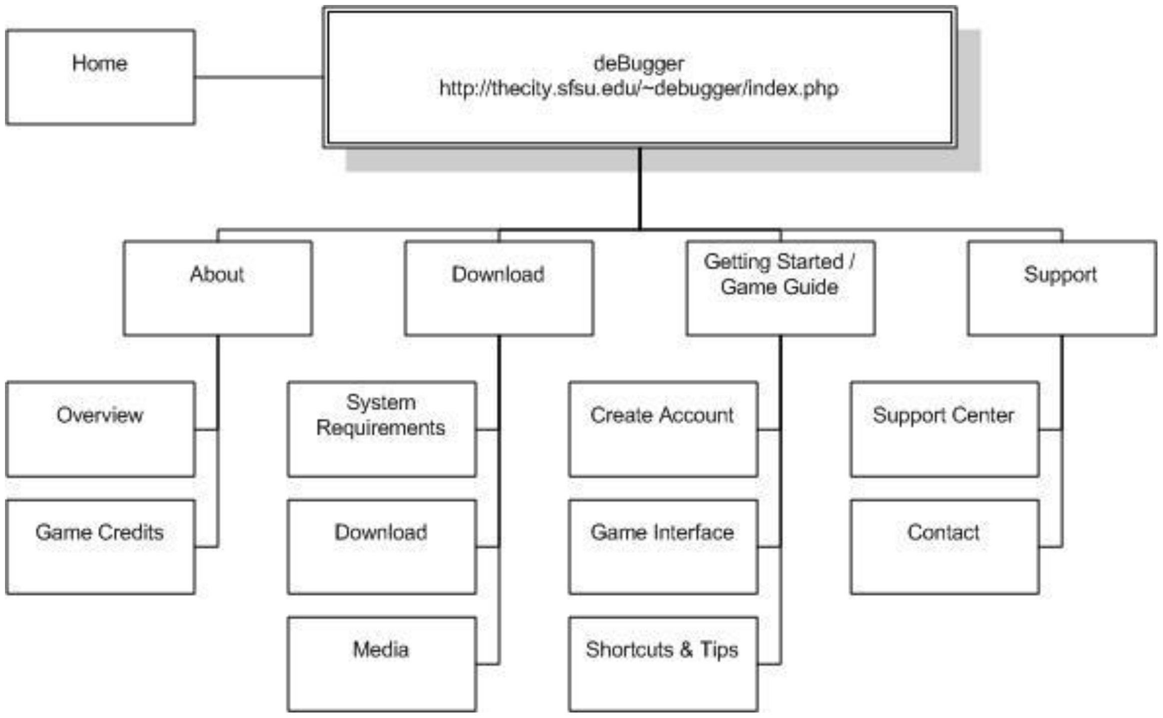
Below are the wire diagrams of the first three, as the fourth is made up of very little links.







Based off of these wire diagrams, we can reduce to the functionalities that is most representative of our game, as well as ridding it of redundancies such as the multiple community links as per the World of Warcraft.

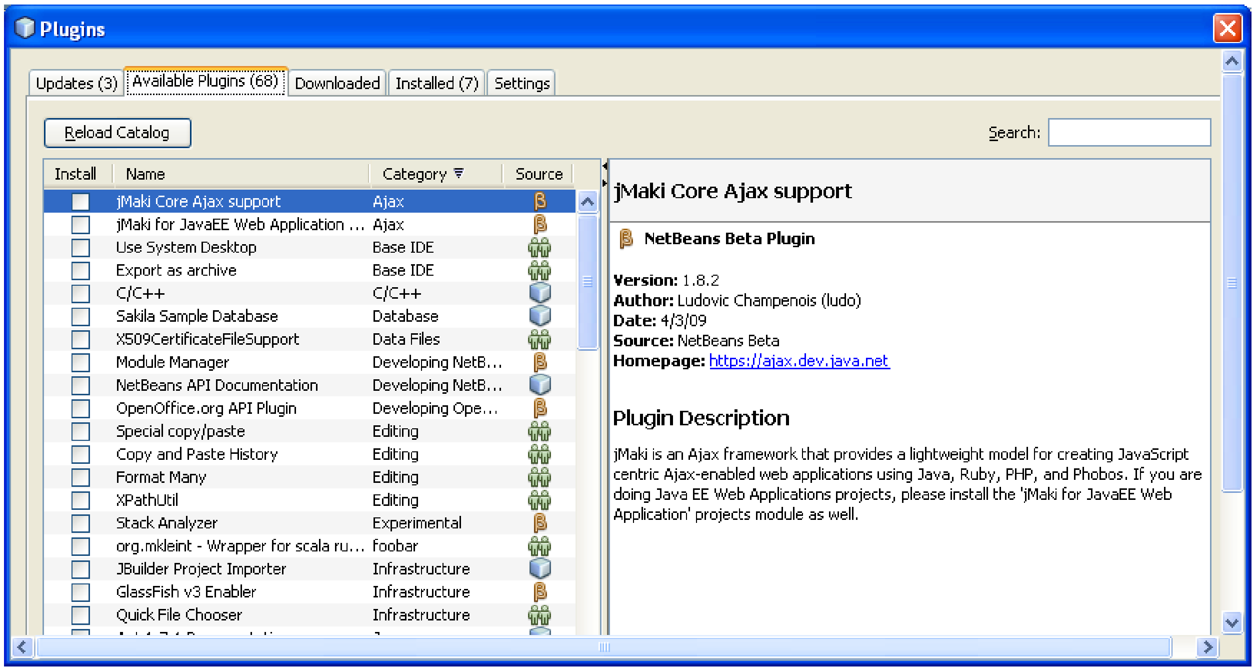


It contains all the functionalities that the other sites try to incorporate, in a format that is easily understandable, without any redundant information.

The website organizes the content within folders, which contains the individual php files for each of the webpages.

2. IDE SETUP (NETBEANS) CLIENT SIDE

As the reader may have already read, many of us chose NetBeans as our IDE in order to implement the game, we felt that the ease of use and documentation of NetBeans greatly surpasses eclipse in every way, well; some of us felt this way. The following tutorial assumes that the reader is using the windows operating system and NetBeans 6.7.1. After you have finished downloaded and installed Panda3D, the first thing that needs to be done is to install the Python plug-ins, this has to be done regardless if the reader downloaded every single package form the NetBeans website. First, navigate to the tools menu and click on plug-ins tab, a screen will pop-up, like the one below. Click in the "Available Plugins" tab, and on the search bar search for python, click on the radio box for python and jython, scroll down and click the install button. After the python plug-ins have been downloaded and installed, restart NetBeans.



Install	Name	Category	Source
☐	jMaki Core Ajax support	Ajax	β
☐	jMaki for JavaEE Web Application ...	Ajax	β
☐	Use System Desktop	Base IDE	
☐	Export as archive	Base IDE	
☐	C/C++	C/C++	
☐	Sakila Sample Database	Database	
☐	X509CertificateFileSupport	Data Files	
☐	Module Manager	Developing NetB...	β
☐	NetBeans API Documentation	Developing NetB...	
☐	OpenOffice.org API Plugin	Developing Ope...	β
☐	Special copy/paste	Editing	
☐	Copy and Paste History	Editing	
☐	Format Many	Editing	
☐	XPathUtil	Editing	
☐	Stack Analyzer	Experimental	β
☐	org.mkleint - Wrapper for scala ru...	foobar	
☐	JBuilder Project Importer	Infrastructure	
☐	GlassFish v3 Enabler	Infrastructure	β
☐	Quick File Chooser	Infrastructure	

jMaki Core Ajax support

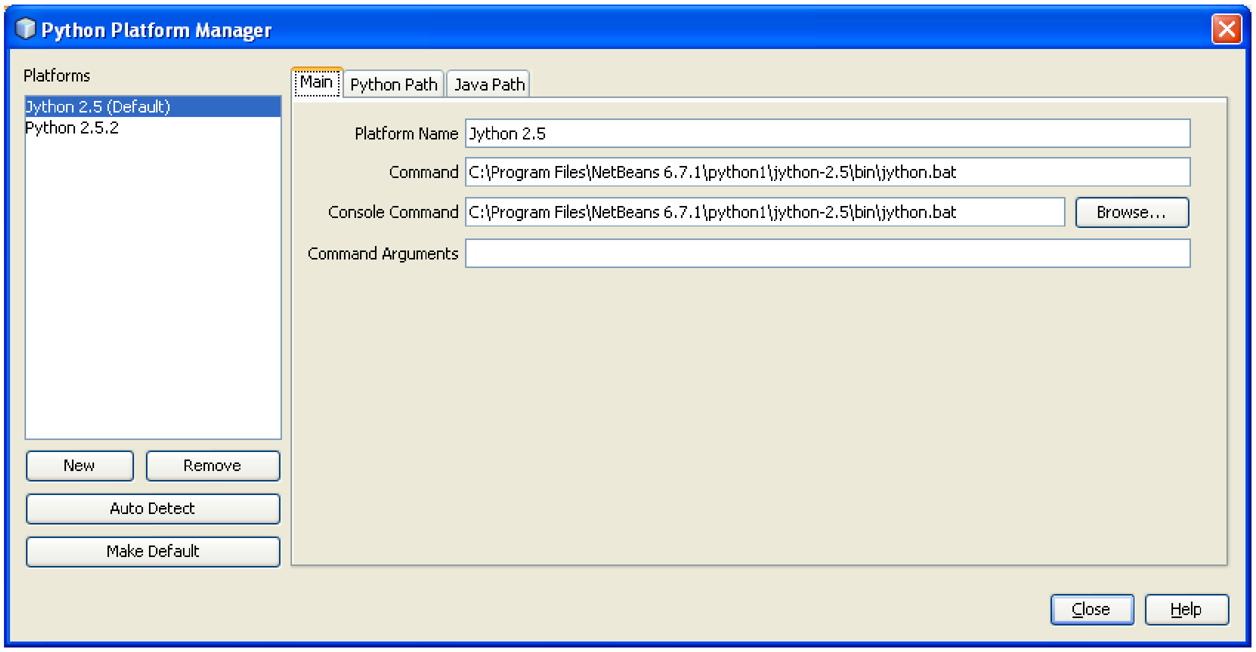
β NetBeans Beta Plugin

Version: 1.8.2
Author: Ludovic Champenois (ludo)
Date: 4/3/09
Source: NetBeans Beta
Homepage: https://ajax.dev.java.net

Plugin Description

jMaki is an Ajax framework that provides a lightweight model for creating JavaScript centric Ajax-enabled web applications using Java, Ruby, PHP, and Phobos. If you are doing Java EE Web Applications projects, please install the 'jMaki for JavaEE Web Application' projects module as well.

Now is time to set up the Panda3D libraries. Find you way to the tools menu and click on the "Python Platforms" tab, another screen will pop-up, like the one below. Make sure you have set Jython as your default platform, if no platforms show up on the screen, click on the "Auto-Detect" button and windows will find them for you automatically. When the platforms show up on the screen, then select Jython as your Platform



Platforms: Jython 2.5 (Default), Python 2.5.2

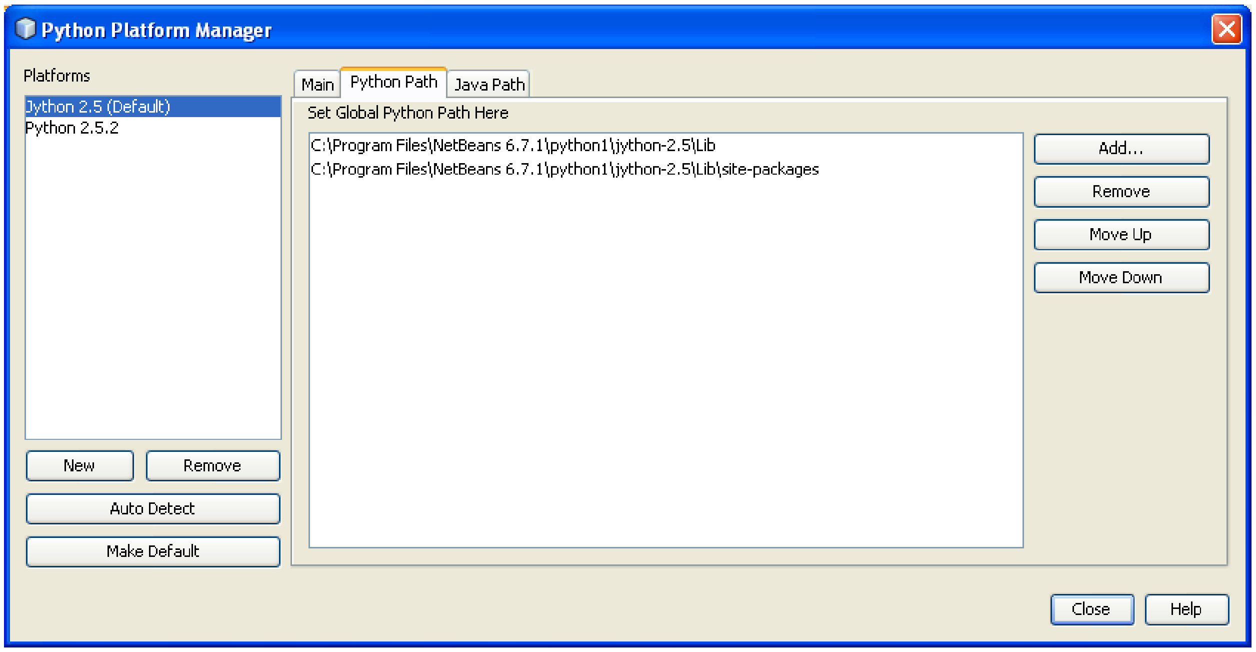
Platform Name: Jython 2.5

Command: C:\Program Files\NetBeans 6.7.1\python1\jython-2.5\bin\jython.bat

Console Command: C:\Program Files\NetBeans 6.7.1\python1\jython-2.5\bin\jython.bat

Command Arguments:

After Jython is set as your default platform, click on the “Python Path” tab and set your path. Once again if no path is shown, click on Auto Detect and windows will find them for you.



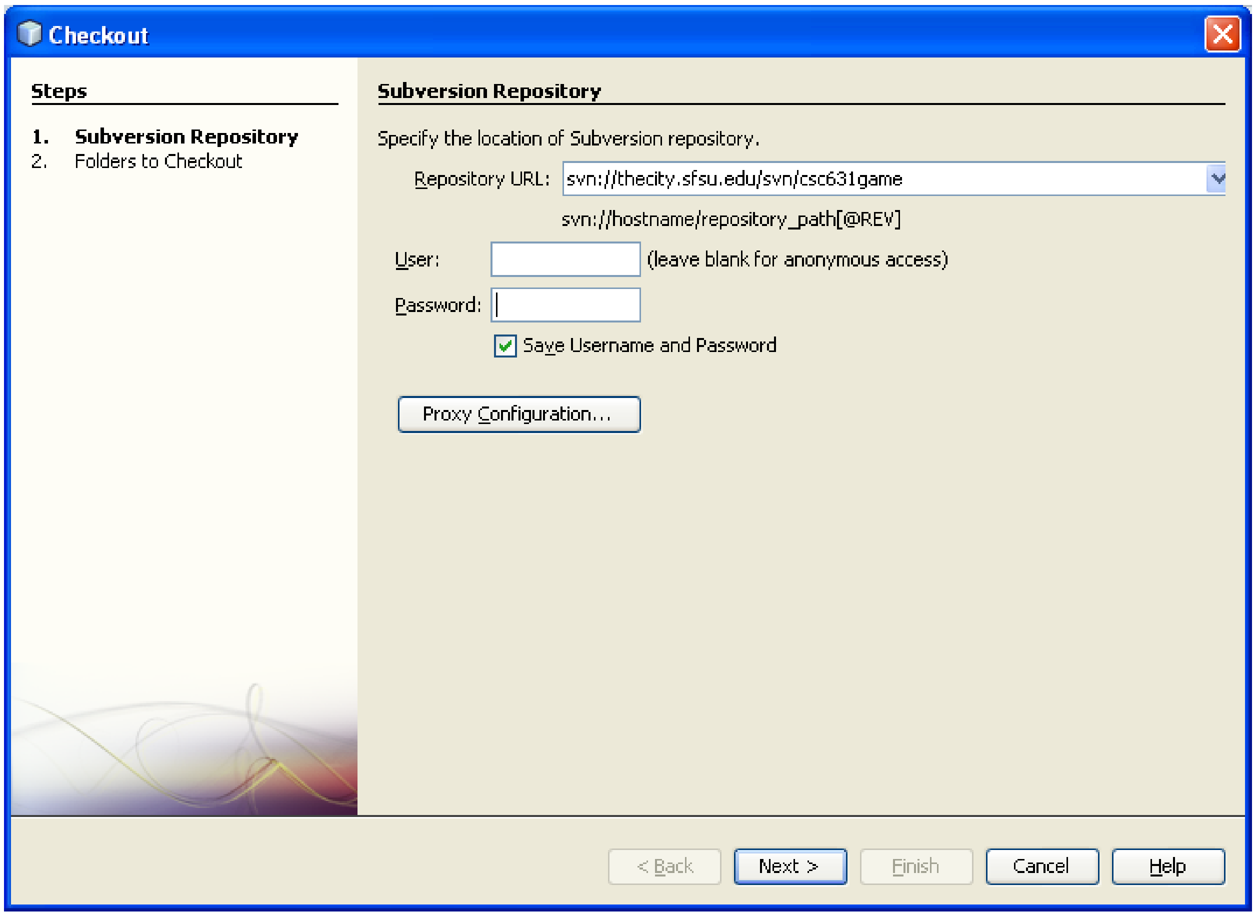
Platforms: Jython 2.5 (Default), Python 2.5.2

Set Global Python Path Here

```
C:\Program Files\NetBeans 6.7.1\python1\jython-2.5\Lib
C:\Program Files\NetBeans 6.7.1\python1\jython-2.5\Lib\site-packages
```

SUBVERSION

Congratulations, you have successfully set-up NetBeans. Next step would be to download the game from the repository, NetBeans makes this really easy, all one has to do is click on the "Team" menu and find your way to the subversion tab, hover your mouse on top of the subversion tab and click on the "Checkout" tab, yet again another pop-up screen will come up. Enter the subversion path, along with the User and Password fields, (when I was taking the class the user name and password were the same as your sfsu email account) click on next and the code will be downloaded. Create a new Project with existing sources; find the folder where you downloaded the game, and your done. Following is a snapshot of how the screen will look like. The repository URL that you see is the URL that we used when taken that class. I imagine it is not so different now.



Specify the location of Subversion repository.

Repository URL: svn://thecity.sfsu.edu/svn/csc631game

svn://hostname/repository_path[@REV]

User: (leave blank for anonymous access)

Password:

✓ Save Username and Password

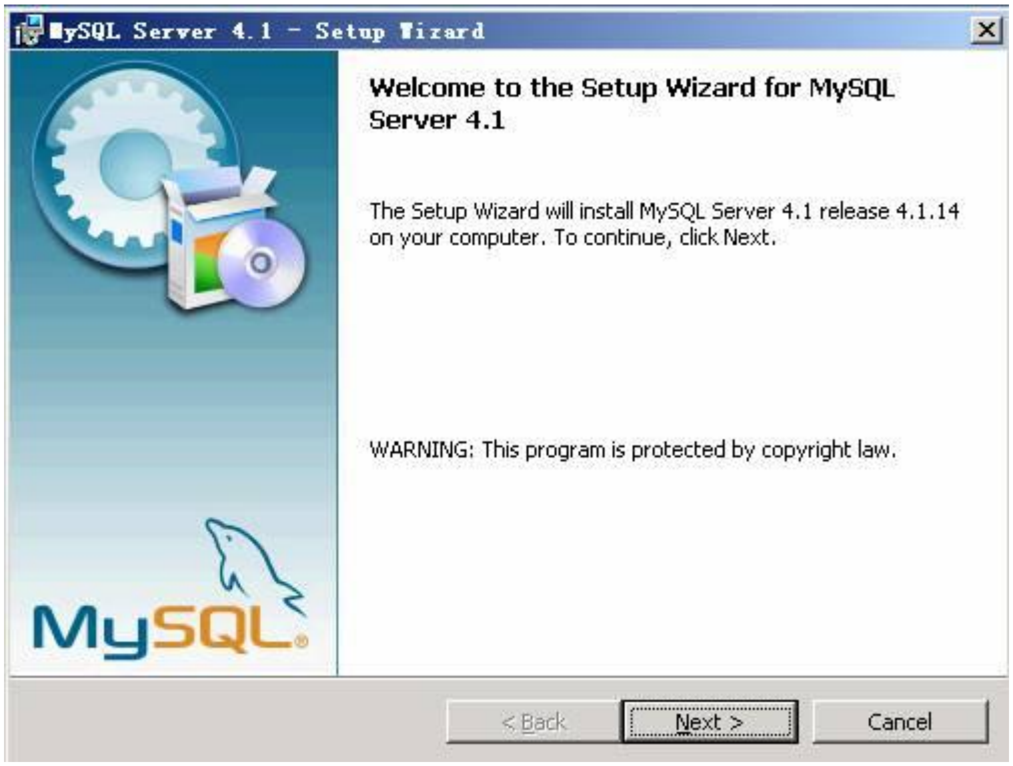
2. SETTING UP IDE (MINGHAOLI@SBCGLOBAL.NET) SERVER SIDE

(c.) Server side development IDE (needed for JAVA, mySQL, BugServer)

The following tutorial assumes that the reader is using the windows operating system to install the mySQL and mySQL GUI tool. Also, the setup of the NurseTownServer for JAVA(eclipse and netbean).

Here is the link for download the mySQL and mySQL gui tool: http://dev.mysql.com/downloads/ (suggest download the newest version)

1. Installation interface(the welcome page)


MySQL Server 4.1 - Setup Wizard

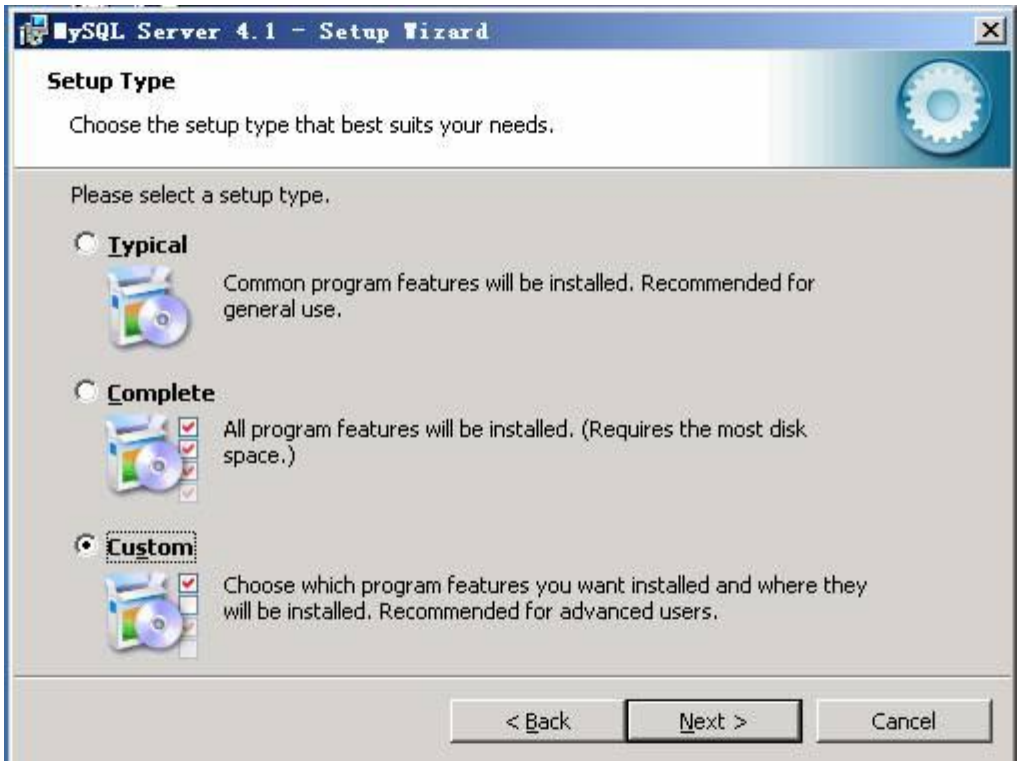
Welcome to the Setup Wizard for MySQL Server 4.1

The Setup Wizard will install MySQL Server 4.1 release 4.1.14 on your computer. To continue, click Next.

WARNING: This program is protected by copyright law.

Figure 2.c.1

2. Choose the installation type
   Choose the "Custom"


MySQL Server 4.1 - Setup Wizard

Setup Type

Choose the setup type that best suits your needs.

Please select a setup type.

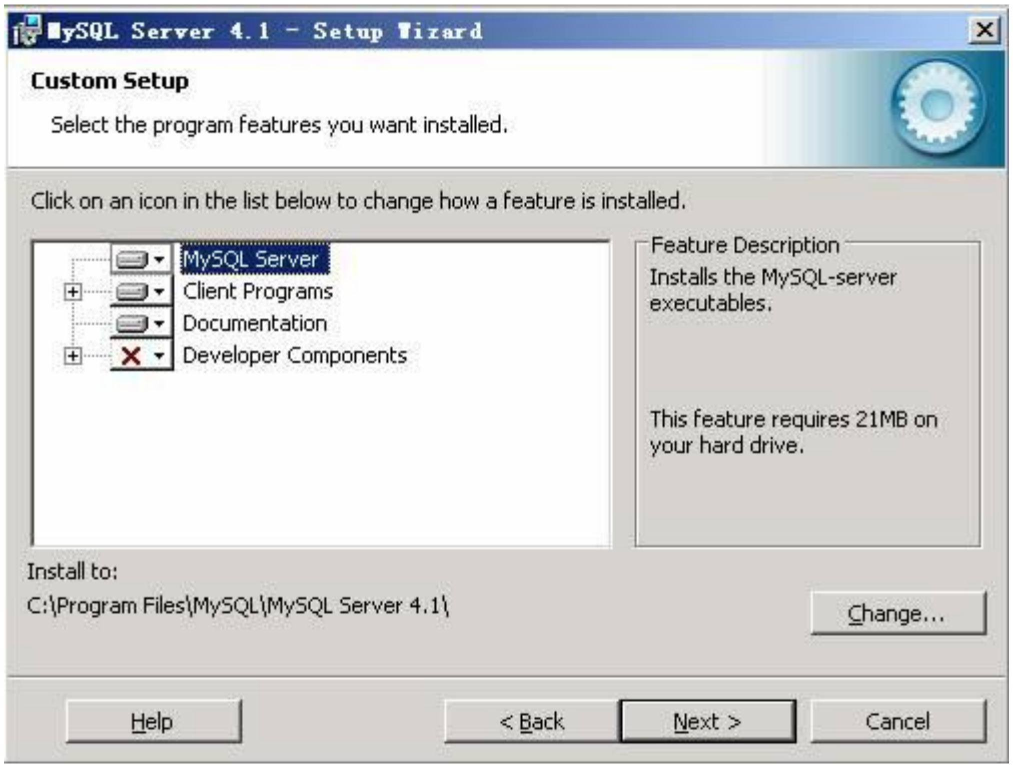
( ) **Typical**
Common program features will be installed. Recommended for general use.

( ) **Complete**
All program features will be installed. (Requires the most disk space.)

[✓ **Custom**]
Choose which program features you want installed and where they will be installed. Recommended for advanced users.

Figure 2.c.2

3. Custom installation
   In default case, MySQL will install to the C:\Program Files\MySQL\MySQL Server 4.1\



MySQL Server 4.1 - Setup Wizard

Custom Setup

Select the program features you want installed.

Click on an icon in the list below to change how a feature is installed.

- MySQL Server
- Client Programs
- Documentation
- Developer Components

Feature Description

Installs the MySQL-server executables.

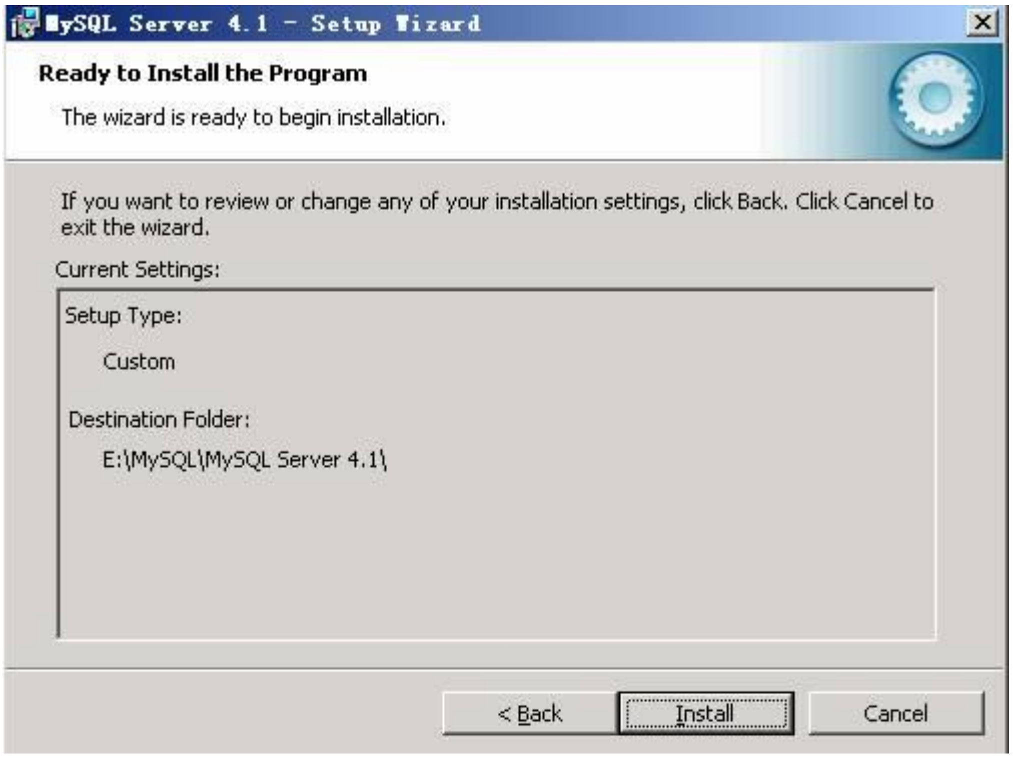
This feature requires 21MB on your hard drive.

Install to:

C:\Program Files\MySQL\MySQL Server 4.1\

Figure 2.c.3

4. Start to install



MySQL Server 4.1 - Setup Wizard

Ready to Install the Program

The wizard is ready to begin installation.

If you want to review or change any of your installation settings, click Back. Click Cancel to exit the wizard.

Current Settings:

Setup Type:

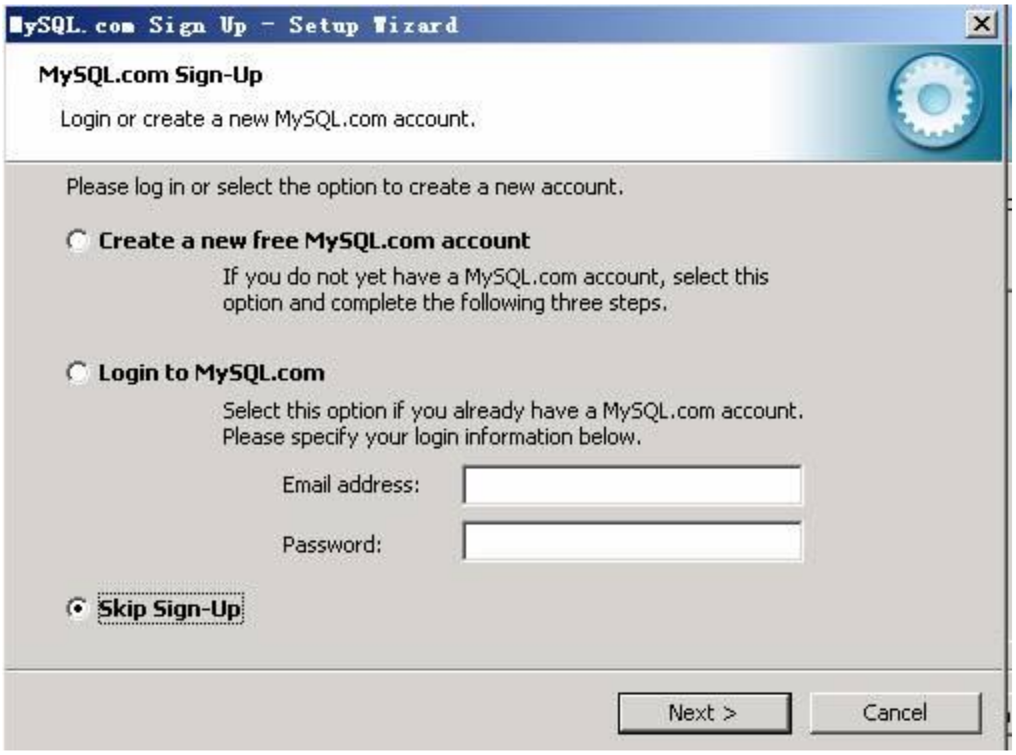
Custom

Destination Folder:

E:\MySQL\MySQL Server 4.1\

Figure 2.c.4

5. Create MySQL.com account interface, choose“Create anew free MySQL.com accout” 。 (suggest choose “skip sign-up”)


MySQL.com Sign Up - Setup Wizard

MySQL.com Sign-Up

Login or create a new MySQL.com account.

Please log in or select the option to create a new account.

○ Create a new free MySQL.com account

If you do not yet have a MySQL.com account, select this option and complete the following three steps.

○ Login to MySQL.com

Select this option if you already have a MySQL.com account. Please specify your login information below.

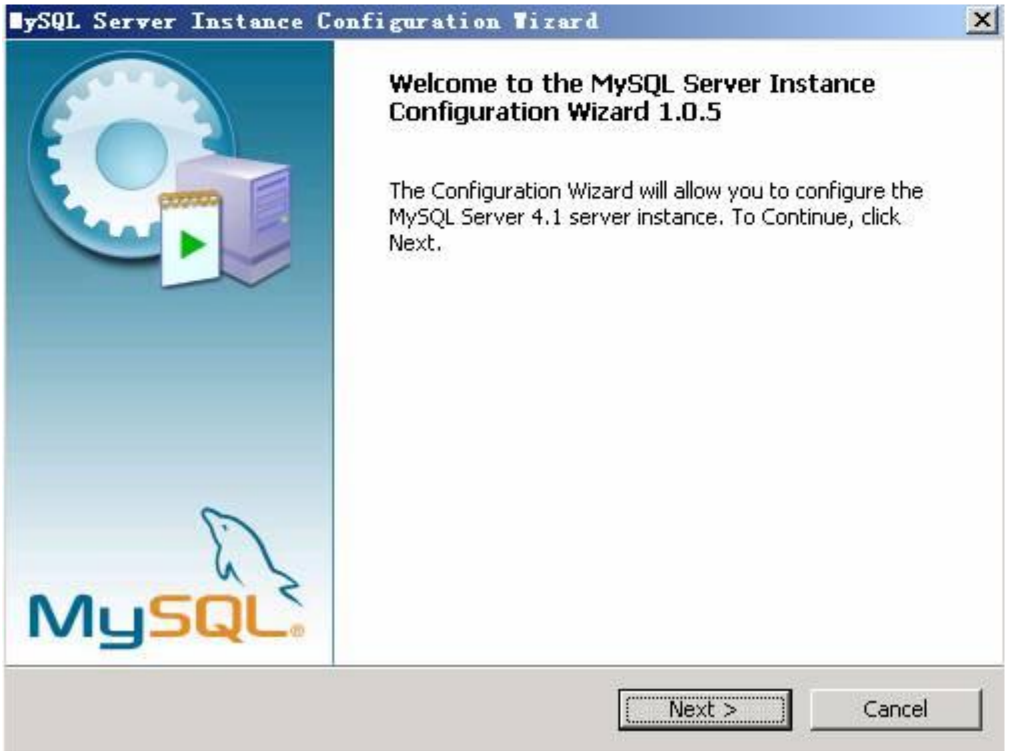
Email address:	
Password:	

[✓ Skip Sign-Up]

Figure 2.c.5

6. Finish installation

(p.s. there is a "Configure the MySQL Server now, suggest to click it)

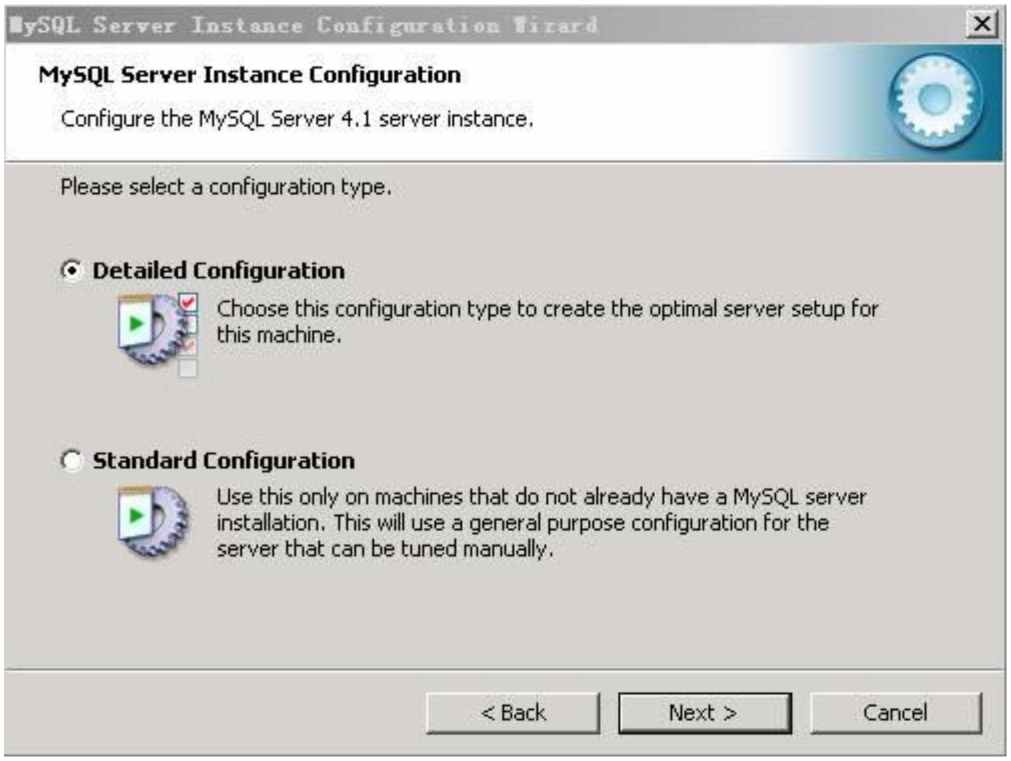

MySQL Server Instance Configuration Wizard

Welcome to the MySQL Server Instance Configuration Wizard 1.0.5

The Configuration Wizard will allow you to configure the MySQL Server 4.1 server instance. To Continue, click Next.

Figure 2.c.6

7. Configure (choose detailed configuration)


MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please select a configuration type.

(●) **Detailed Configuration**

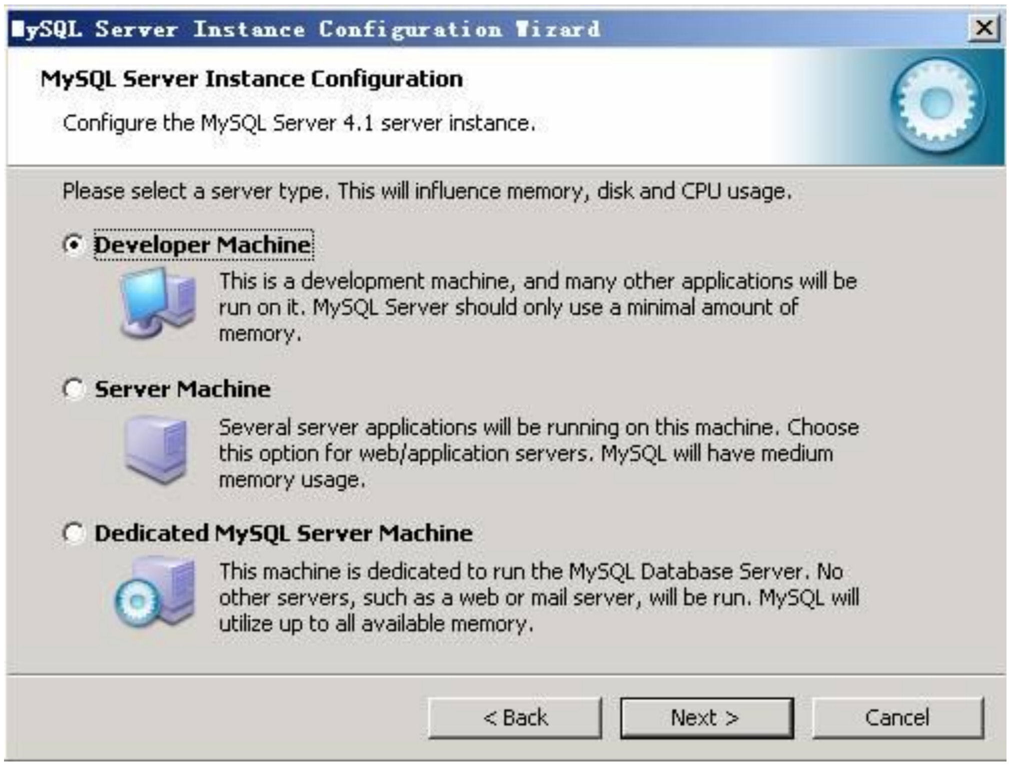
Choose this configuration type to create the optimal server setup for this machine.

( ) **Standard Configuration**

Use this only on machines that do not already have a MySQL server installation. This will use a general purpose configuration for the server that can be tuned manually.

Figure 2.c.7

8. Choose "developer machine"


MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please select a server type. This will influence memory, disk and CPU usage.

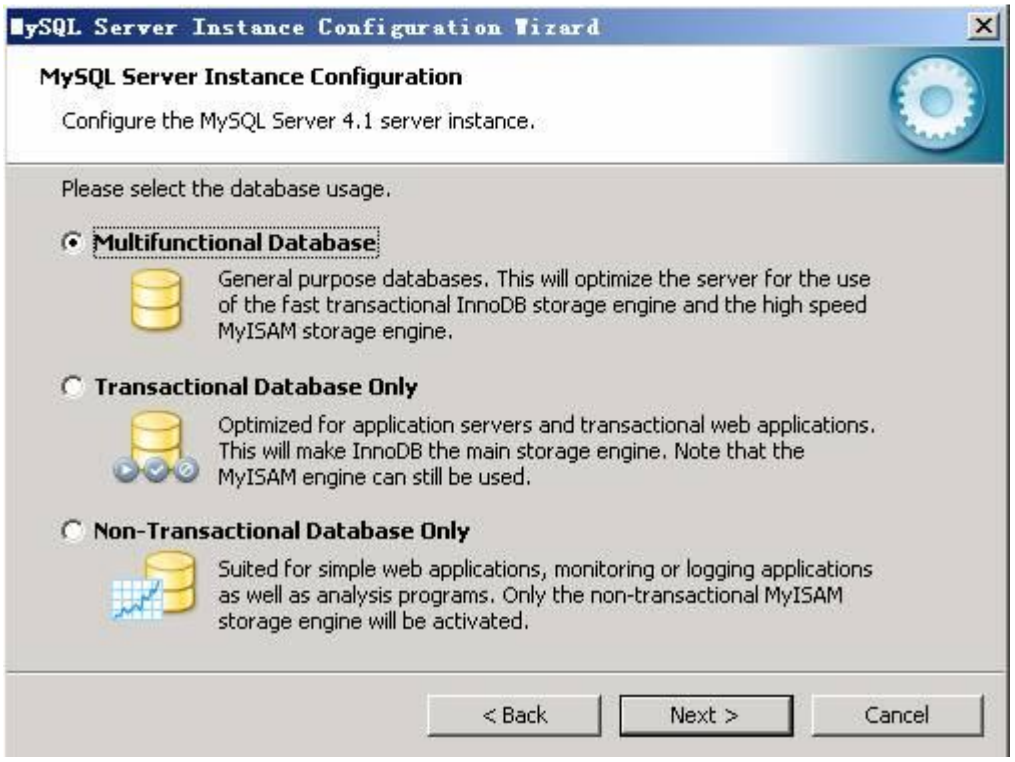
[✓ **Developer Machine**]
This is a development machine, and many other applications will be run on it. MySQL Server should only use a minimal amount of memory.

Server Machine
Several server applications will be running on this machine. Choose this option for web/application servers. MySQL will have medium memory usage.

Dedicated MySQL Server Machine
This machine is dedicated to run the MySQL Database Server. No other servers, such as a web or mail server, will be run. MySQL will utilize up to all available memory.

Figure 2.c.8

9. Choose database (choose multifunctional database)


MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please select the database usage.

(●) **Multifunctional Database**

General purpose databases. This will optimize the server for the use of the fast transactional InnoDB storage engine and the high speed MyISAM storage engine.

( ) **Transactional Database Only**

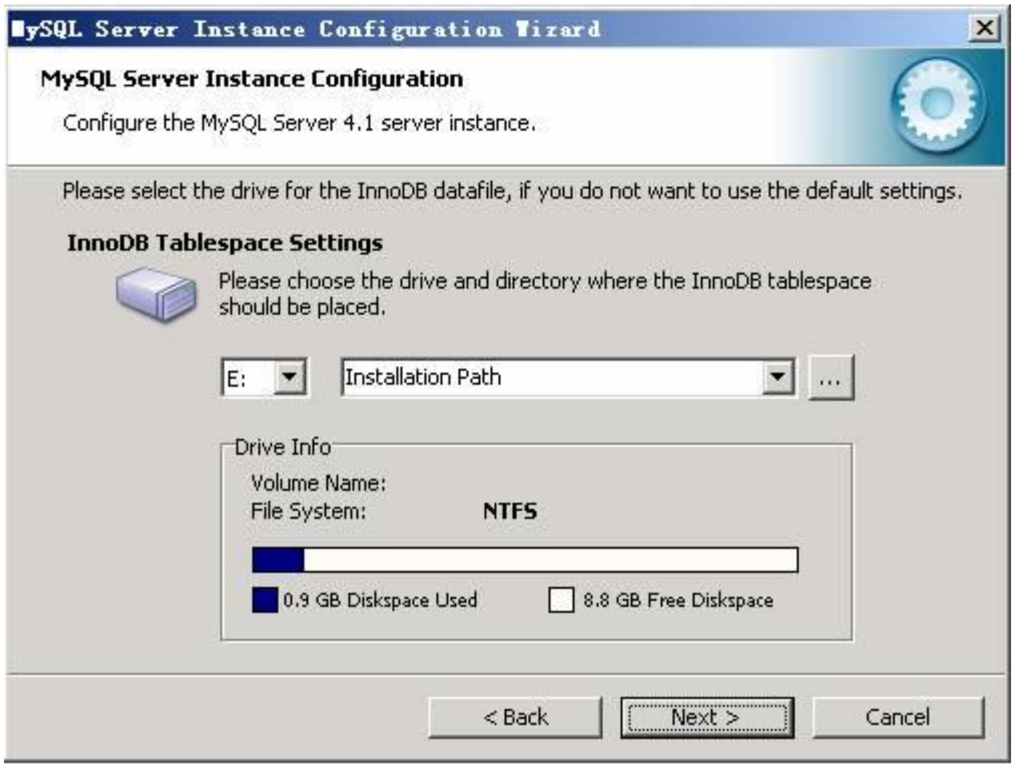
Optimized for application servers and transactional web applications. This will make InnoDB the main storage engine. Note that the MyISAM engine can still be used.

( ) **Non-Transactional Database Only**

Suited for simple web applications, monitoring or logging applications as well as analysis programs. Only the non-transactional MyISAM storage engine will be activated.

Figure 2.c.9

10. Choose innoDB



MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please select the drive for the InnoDB datafile, if you do not want to use the default settings.

InnoDB Tablespace Settings

Please choose the drive and directory where the InnoDB tablespace should be placed.

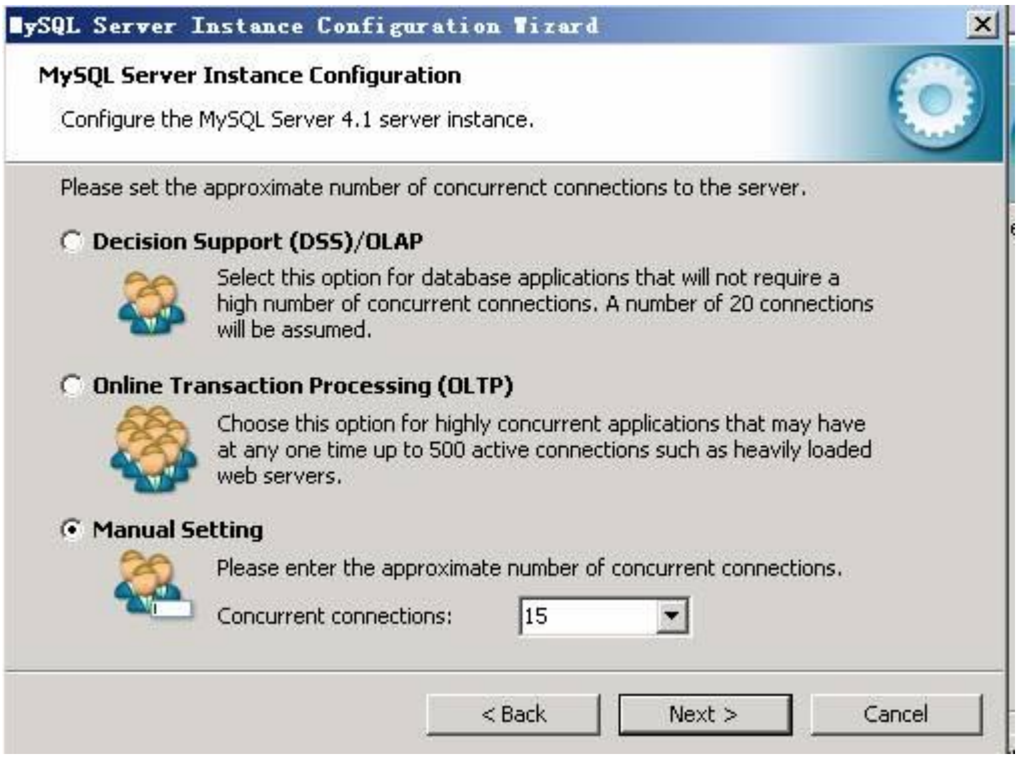
E: | Installation Path

Drive Info
Volume Name:
File System: **NTFS**

0.9 GB Diskspace Used
8.8 GB Free Diskspace

Figure 2.c.10

11. Choose the number of connection. (since that just for running and testing, not really need to make the number too large)


MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please set the approximate number of concurrenct connections to the server.

( ) Decision Support (DSS)/OLAP

Select this option for database applications that will not require a high number of concurrent connections. A number of 20 connections will be assumed.

( ) Online Transaction Processing (OLTP)

Choose this option for highly concurrent applications that may have at any one time up to 500 active connections such as heavily loaded web servers.

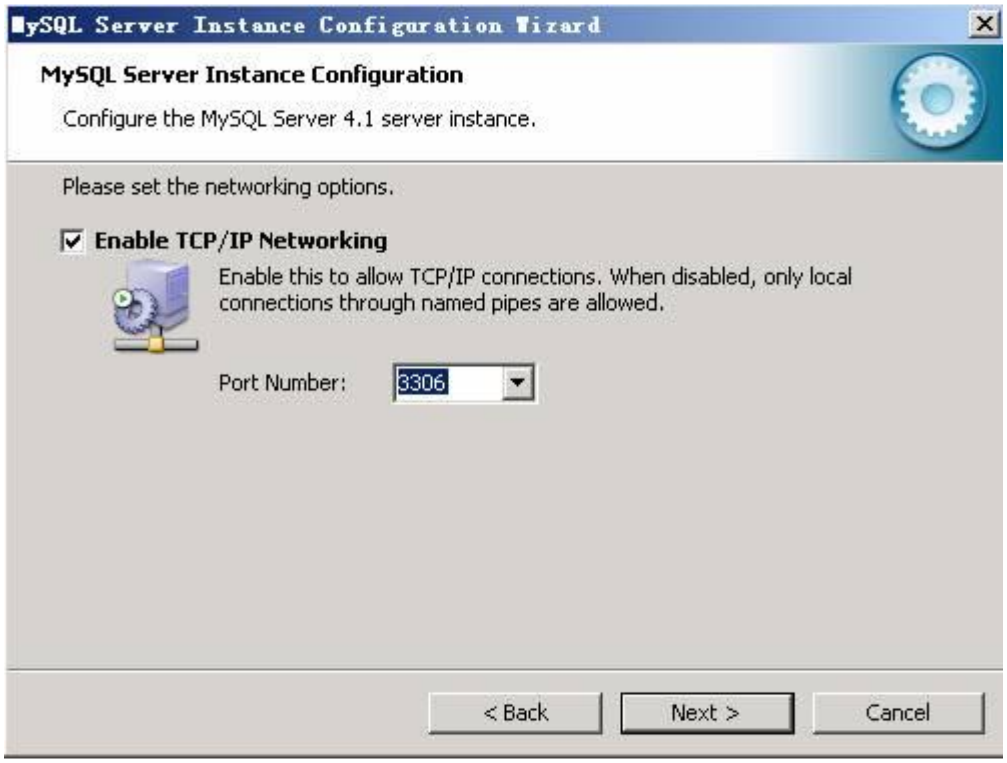
[✓ Manual Setting]

Please enter the approximate number of concurrent connections.

Concurrent connections: 15

Figure 2.c.11

12. setup MySQL 's TCP/IP. Just choose the default one is fine.


MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please set the networking options.

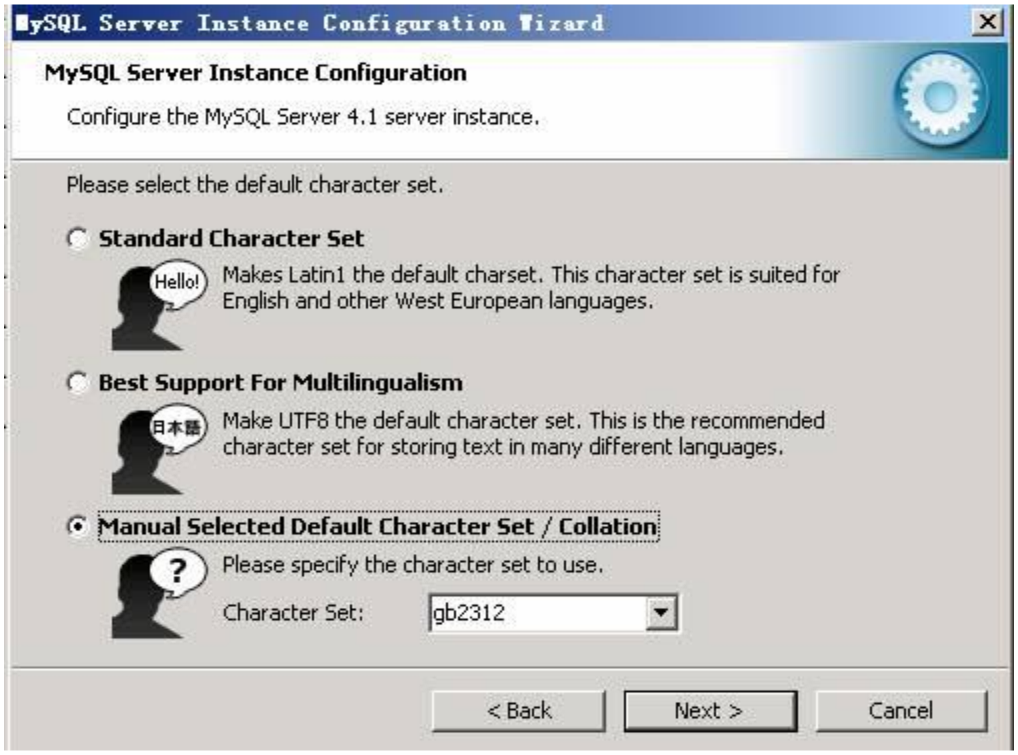
☑ Enable TCP/IP Networking

Enable this to allow TCP/IP connections. When disabled, only local connections through named pipes are allowed.

Port Number: 3306

Figure 2.c.11

13. MySQL character. In here will allow you to choose what language you want to use in the MySQL(p.s. for Chinese choose gb2312) for English, click the “Standard Character Set”.



MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please select the default character set.

( ) Standard Character Set

Makes Latin1 the default charset. This character set is suited for English and other West European languages.

( ) Best Support For Multilingualism

Make UTF8 the default character set. This is the recommended character set for storing text in many different languages.

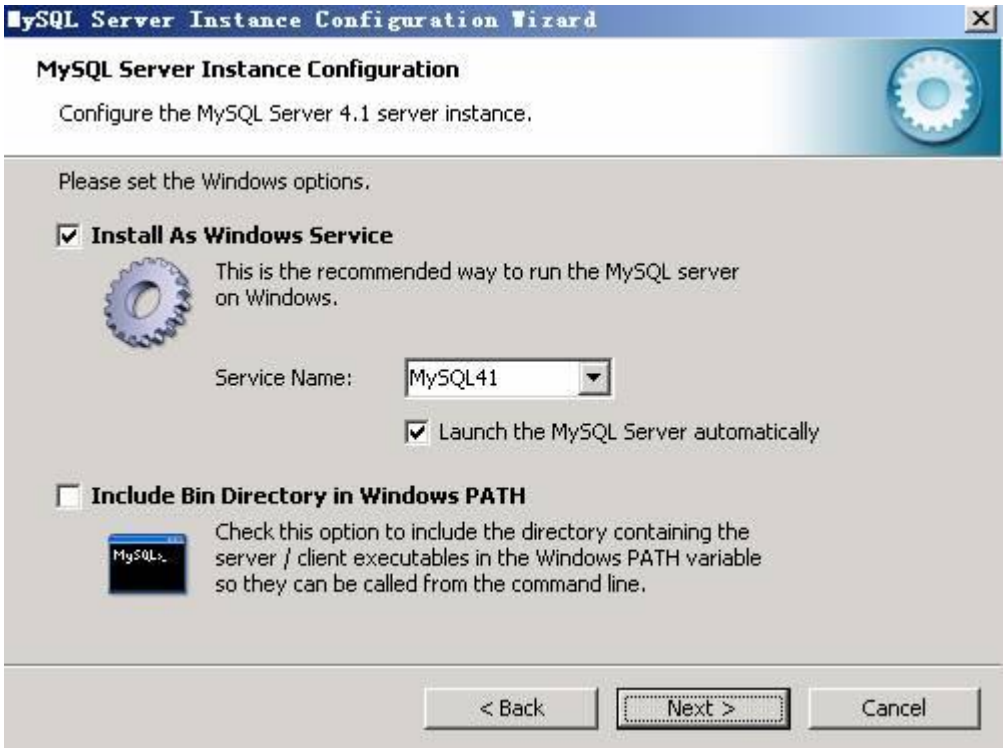
(•) Manual Selected Default Character Set / Collation

Please specify the character set to use.

Character Set: gb2312

Figure 2.c.12

14. The “install as Windows service” must click if you want to using window to run it. the service name just uses the default one, and the “launch the MySQL Server automatically” is optional. (suggest to click it, otherwise , if forget to start the server, the NurseTownServer will occur some error)



MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please set the Windows options.

[✓] **Install As Windows Service**

This is the recommended way to run the MySQL server on Windows.

Service Name: MySQL41

[✓] Launch the MySQL Server automatically

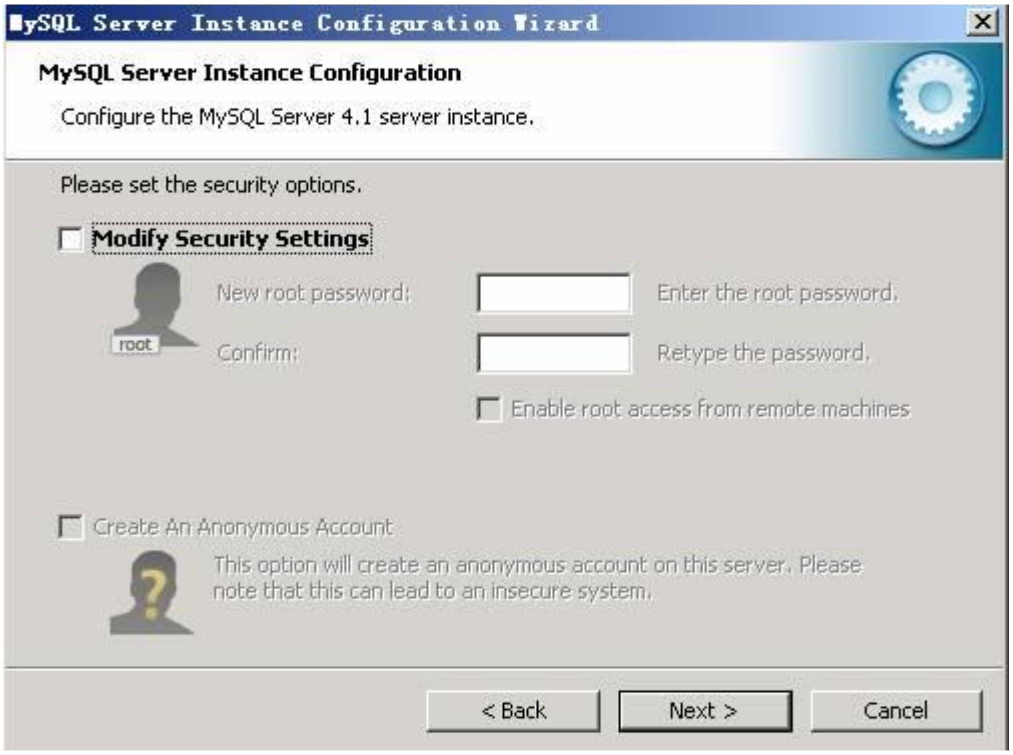
[ ] **Include Bin Directory in Windows PATH**

Check this option to include the directory containing the server / client executables in the Windows PATH variable so they can be called from the command line.

< Back | Next > | Cancel

Figure 2.c.13

15. Setup the account and the password for the MySQL server. (suggest to click "Modify Security Settings, but make sure remember the password.) (P.S. the default username is "root").



MySQL Server Instance Configuration Wizard

MySQL Server Instance Configuration

Configure the MySQL Server 4.1 server instance.

Please set the security options.

☐ **Modify Security Settings**

New root password: Enter the root password.

Confirm: Retype the password.

☐ Enable root access from remote machines

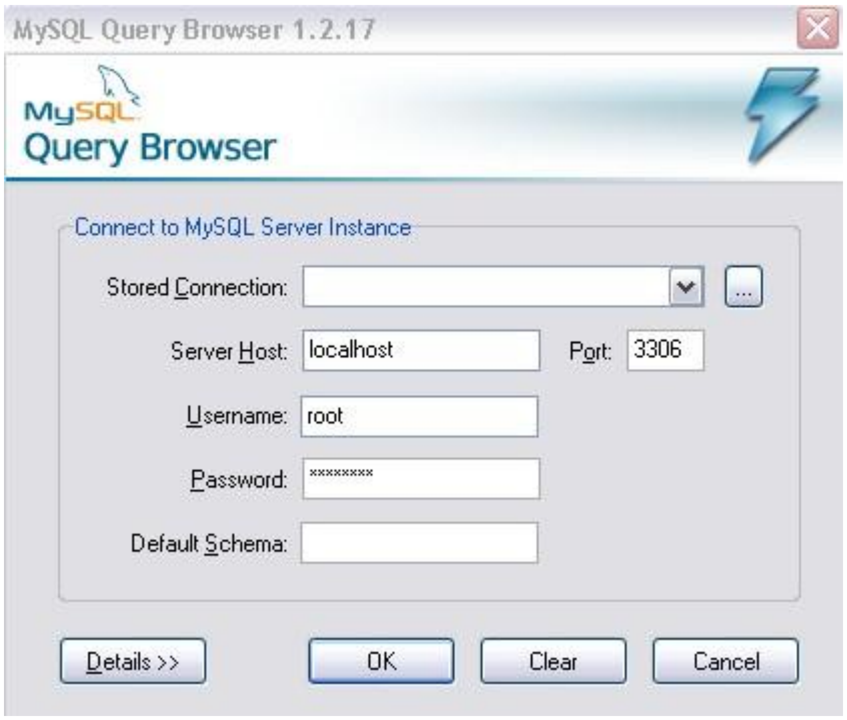
☐ Create An Anonymous Account

This option will create an anonymous account on this server. Please note that this can lead to an insecure system.

Figure 2.c.14

After all those steps, the installation is done. To check if it work. you can click the “start”, “all program”, “MYSQL”, “MYSQL system tray monitor”. After that, the right down conner will have the little gear icon. Right click it and choose “start instance”(if you not automatically start instance when install MySQL).

16. Then right click it again and choose “MySQL query browser”. Then enter the password


MySQL Query Browser 1.2.17

Connect to MySQL Server Instance

Stored Connection:

Server Host: localhost

Port: 3306

Username: root

Password: ********

Default Schema:

Figure 2.c.15

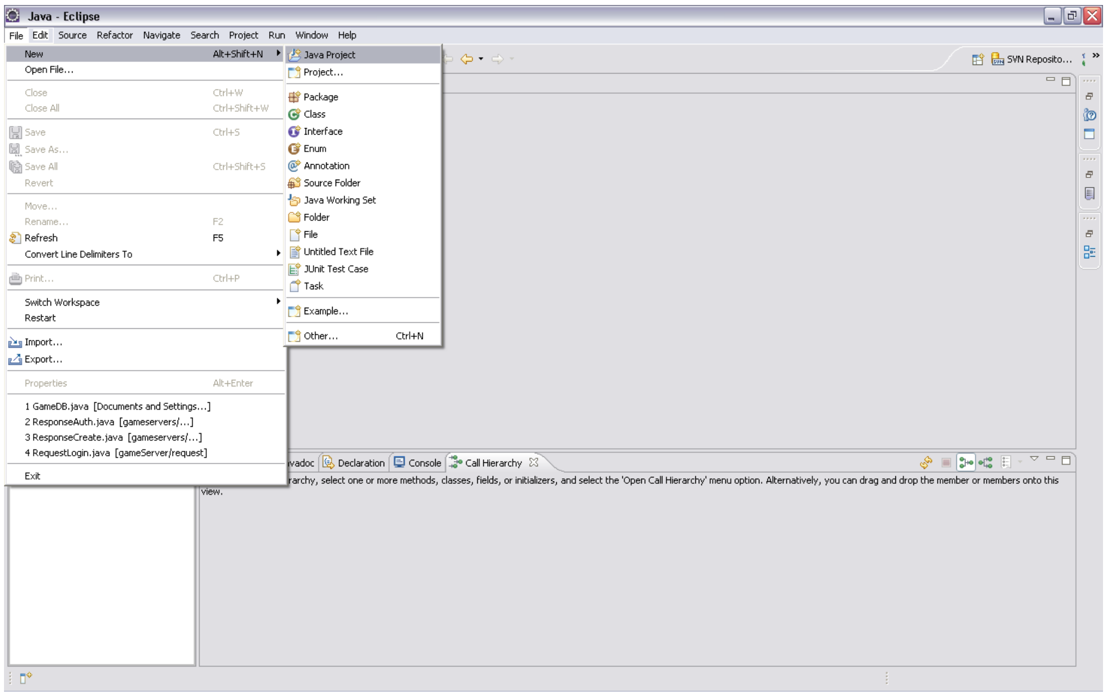
P.S if you installed the MySQL before, and delete it. now you reinstall, you have to delete the hidden folder before you reinstall it. otherwise the installation will occur some problem.

dump data to the DB.

1. Go to the link: http://thecity.sfsu.edu/phpmyadmin to get the SQL dump which is the data for the game.
2. Open the mySQL browser and click file→new script tab→copy the text from SQL dump to the script tab→run

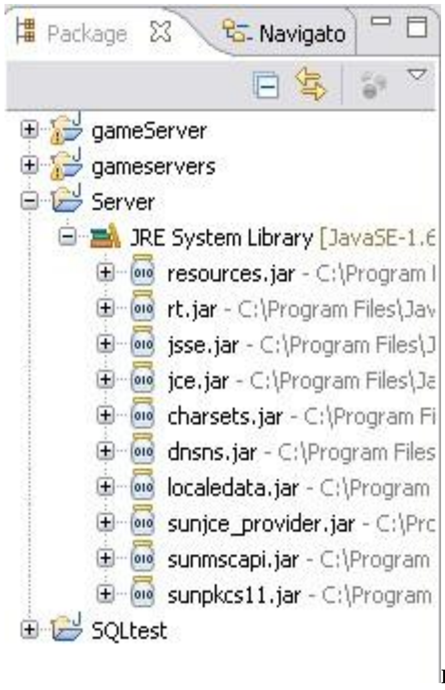
IDE setup (eclipse)

1. Create a new java project which not contain any of the package



rarchy, select one or more methods, classes, fields, or initializers, and select the 'Open Call Hierarchy' menu option. Alternatively, you can drag and drop the member or members onto this view.

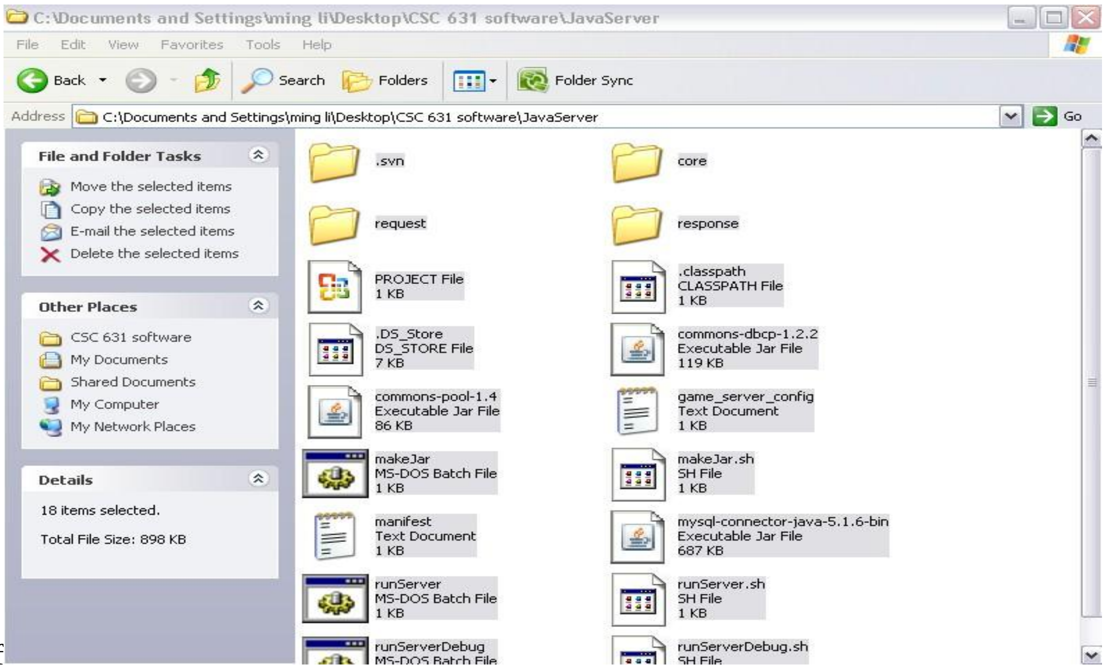
Figure 2.c.16


Package
Navigato
gameServer
gameservers
Server
JRE System Library [JavaSE-1.6
resources.jar - C:\Program
rt.jar - C:\Program Files\Jav
jsse.jar - C:\Program Files\J
jce.jar - C:\Program Files\Ja
charsets.jar - C:\Program Fi
dnsns.jar - C:\Program Files
localedata.jar - C:\Program
sunjce_provider.jar - C:\Prc
sunmscapi.jar - C:\Program
sunpkcs11.jar - C:\Program
SQLtest

no package in the project.

Figure 2.c.17

2. Select all the Server class (include the library) to the project



18 items selected.

Total File Size: 898 KB

.svn

core

request

response

PROJECT File
1 KB

.classpath
CLASSPATH File
1 KB

.DS_Store
DS_STORE File
7 KB

commons-dbcp-1.2.2
Executable Jar File
119 KB

commons-pool-1.4
Executable Jar File
86 KB

game_server_config
Text Document
1 KB

makeJar
MS-DOS Batch File
1 KB

makeJar.sh
SH File
1 KB

manifest
Text Document
1 KB

mysql-connector-java-5.1.6-bin
Executable Jar File
687 KB

runServer
MS-DOS Batch File
1 KB

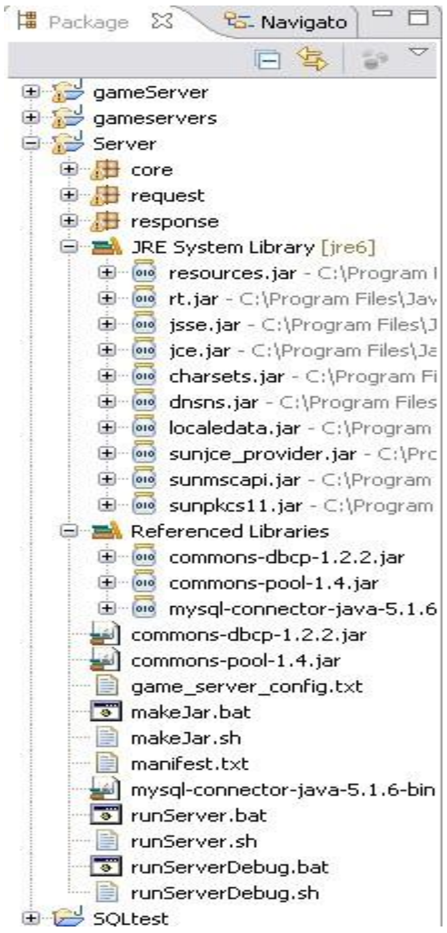
runServer.sh
SH File
1 KB

runServerDebug
MS-DOS Batch File

runServerDebug.sh
SH File

f

Figure 2.c.18

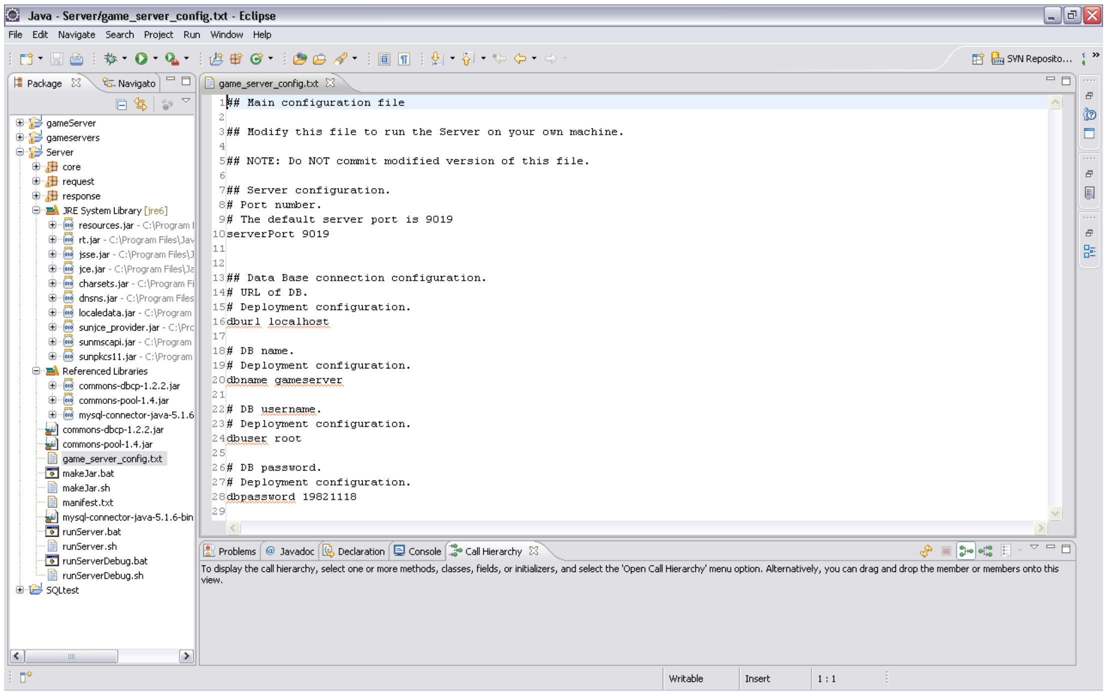


```
gameServer
gameservers
Server
    core
    request
    response
    JRE System Library [jre6]
        resources.jar - C:\Program I
        rt.jar - C:\Program Files\Jav
        jsse.jar - C:\Program Files\J
        jce.jar - C:\Program Files\Ja
        charsets.jar - C:\Program Fi
        dnsns.jar - C:\Program Files
        localedata.jar - C:\Program
        sunjce_provider.jar - C:\Prc
        sunmscapi.jar - C:\Program
        sunpkcs11.jar - C:\Program
    Referenced Libraries
        commons-dbcp-1.2.2.jar
        commons-pool-1.4.jar
        mysql-connector-java-5.1.6
    commons-dbcp-1.2.2.jar
    commons-pool-1.4.jar
    game_server_config.txt
    makeJar.bat
    makeJar.sh
    manifest.txt
    mysql-connector-java-5.1.6-bin
    runServer.bat
    runServer.sh
    runServerDebug.bat
    runServerDebug.sh
SQLtest
```

Figure 2.c.19

after all those file is copy to project, all the library should be contain as the figure.

3. Open game_server_config.txt file and change the “server port” number (depend on the client), change the “DB name”(depend on what database name you create), change the “DB username” (depend on what name you have set up during install the MySQL, and change the “DB password” to what you make when set up the MySQL.

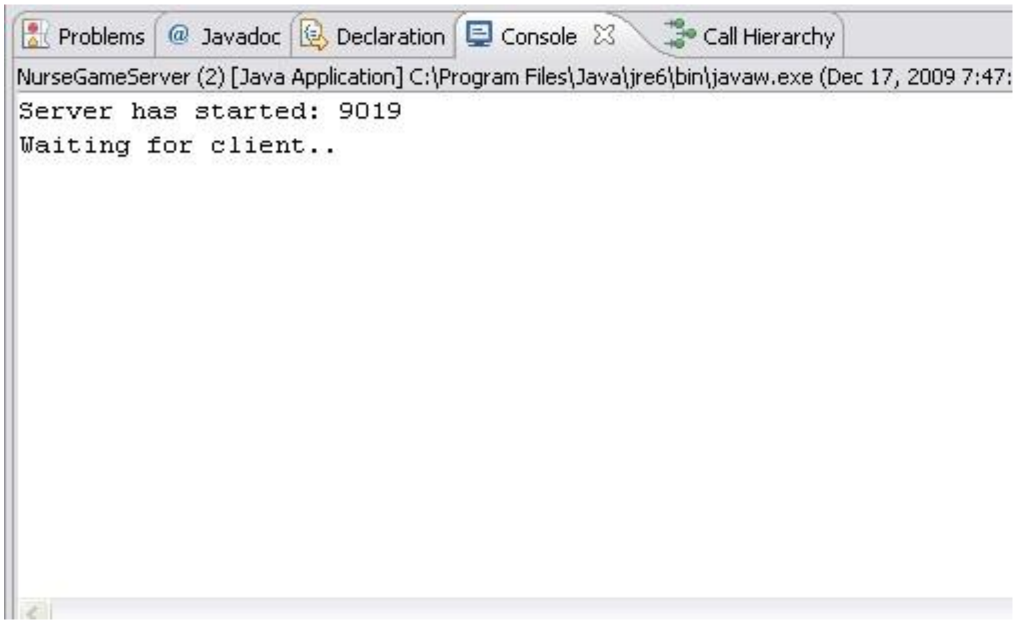


```
## Main configuration file

## Modify this file to run the Server on your own machine.

## NOTE: Do NOT commit modified version of this file.

## Server configuration.
# Port number.
# The default server port is 9019
serverPort 9019


## Data Base connection configuration.
# URL of DB.
# Deployment configuration.
dburl localhost

# DB name.
# Deployment configuration.
dbname gameserver

# DB username.
# Deployment configuration.
dbuser root

# DB password.
# Deployment configuration.
dbpassword 19821118

```

To display the call hierarchy, select one or more methods, classes, fields, or initializers, and select the 'Open Call Hierarchy' menu option. Alternatively, you can drag and drop the member or members onto this view.

Figure 2.c.20

4. Run the server.



```
NurseGameServer (2) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Dec 17, 2009 7:47:
Server has started: 9019
Waiting for client..
```

Figure 2.c.21

The figure is how the output looks like after run the server

3. ARCHITECTURE OVERVIEW

3.a Architecture Overview of Whole Game

debugger is a three-tiered client-server application utilizing multiple technologies to run. The game is organized into three tiers, the Presentation, Logic, and the Data Tier.

The presentation tier:

deBugger the game is written in Python using the Panda3D library. The game utilizes the Panda3D library to take care of various aspects of the game such as collision detection of objects, map changing, communication between players, and all of the events that occur while in the game such as attacking bugs, answering questions, using equipment and potions, and any other game logic that is required.

The Question Creator website utilizes PHP connecting to database through the built in functions. Its role is to provide a direct entry way for players as well as other members of the community to improve the quality of the game by directly allowing them to enter questions that may be used in the game as quiz questions. The questions must be verified by certain members of the game before being allowed answerable in game.

The logic tier:

Java written deBugger server utilizing JDBC for DB connectivity. The server is required in order to verify that players can log into the game, and perform tasks such as keeping track of player and bug movement, spawning, death, as well as sending the quiz questions, item drops, chat notices to players.

Bug Server is a modified client with its graphics window disabled. It manages the movement of bugs walking in a scene to facilitate the load from the server. The bug server keeps track of various pieces of information about a bug such as its health points and level, as well as initiating attacks with

The data tier:

mySQL database connecting to both the deBugger server as well as Question Creator through their respective functions. The data tier is important to the functionality of the game because it stores the persistent information such as inventory, player positions, quizzes, and level. Things like these we would like to store because if the player logs off, this information should be saved and later retrieved so that players would resume, not restart, their progress. Also, the questions that are saved should be accessible by various sources such as question creator and the game, which provides a centralized location for information.

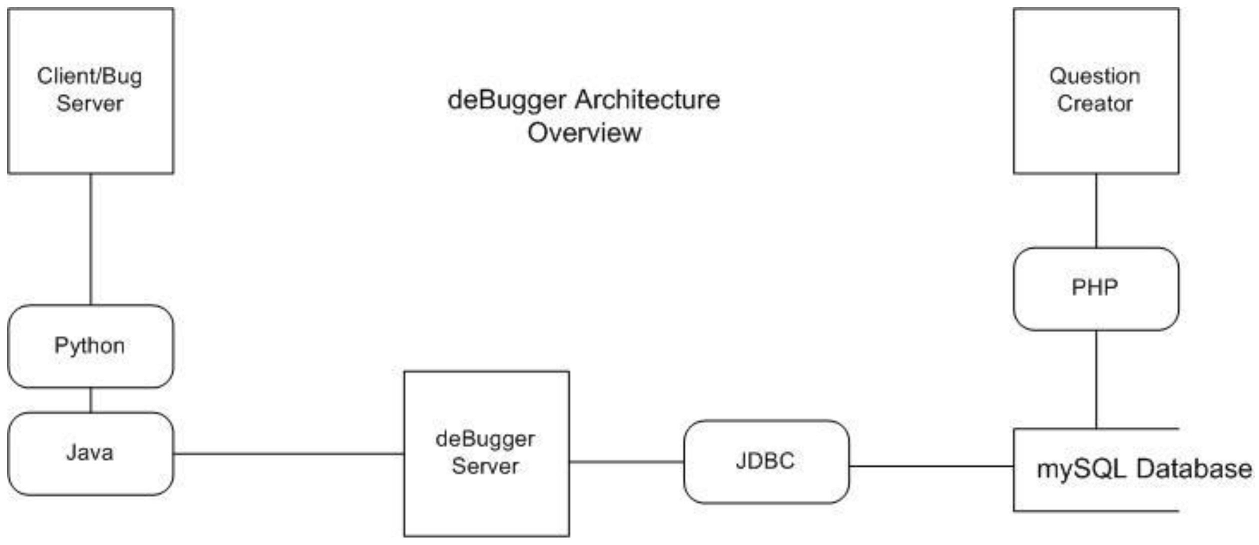


Fig. 3.a-1- debugger Architecture Overview showing the various components, as well as their interconnections of programming languages.

Terminology-

Heartbeat:

A heartbeat is an event that is called by client 10 times every second. Heartbeats are used to poll the server for newly updated information such as chatting, attacking, other players/bugs logging in and out, etc. Heartbeats are sent by both the client and the bug server, which is a modified client.

Client:

Client controls the logging and disconnection of players that want to play deBugger. Client controls movements and the various events that the game sends to the Server as packets of information. client polls the server for information on other players as well as bugs at regular intervals called a heartbeat. The client makes use of Panda3D to detect collisions so that the programmers do not have to calculate collisions themselves. Collisions trigger an event which will do various tasks, depending on what type of collision. For example, a collision between a bug and a player in the bug server will trigger a request that the bug attack aforementioned player. Likewise, a collision between a mouse pointer and the ground, tells the client to move the player to the new position, updating the server on the movement of the player at regular intervals.

Server:

The deBugger Java server handles the events that pertain to players and bugs. Connections from clients are tracked on the basis of movement, deaths, quizzes, item drops, and other tasks such as logging in/out. It stores information in a persistent database which it connects through using JDBC. It holds in queue information that has been sent to the server since the last update for each client so that when a client sends a heartbeat request, the server will send the queued requests back to the client. Each client has its own thread in which the server processes the events. The bug server acts like a client to the server, and the server updates the position of the bugs accordingly because it knows that a bug server is connected through the login that the bug server connected with.

Bug Server:

The bug server is a modified client which handles the movements of multiple bugs through a single connection to the server. Bugs are the various non controllable monsters throughout the game that players use to answer questions, gain gold, and obtain various item drops. The bug server makes it possible for the game to run without bugs, and off loads a lot of the load that the server can use to perform other duties. When bugs are moving, each bug acts as a player and will send an update on its position while it's moving, as well as polling the server through the heartbeat.

Database:

The mySQL database handles the data storage required by the server and question creator. It returns and stores data such as user information, equipment, questions, and bug information. A database has to be used because for certain types of information like quiz questions, we must be able to add and remove changes without taking down the game. This persistent information allows the question creator to update the game on newly approved questions dynamically, and lets the game immediately use those questions as they are approved.

Packet/Protocol:

Each packet of information, aptly named protocol, contains various bits of information as well as the payload used to describe an event or a response that has occurred. There are numerous events, but an example of the few important ones would be Login, SwitchMap, and Move.

Question Creator:

The Question Creator is a PHP website that connects to the *Database* in order to store and modify questions submitted by players. It gives players an easy way to add questions to the game while the game is running, and makes it possible for the game to always be continually updated with questions as they are made. It keeps track of the questions that needs to be approved, and allows the addition of questions from any user with an internet connection.

Logic-

The start of the deBugger game starts at the client, as shown below. From there, the client goes through multiple steps to work its way towards the main loop of the cycle. This cycle is after a player successfully logs on, and from then on, all future information being passed between the client and server follow the same cycle. The main loop of the cycle makes use of either an event dictionary (in regards to Python) or an event hash map (deBugger server) that will look up the events and handle them appropriately.

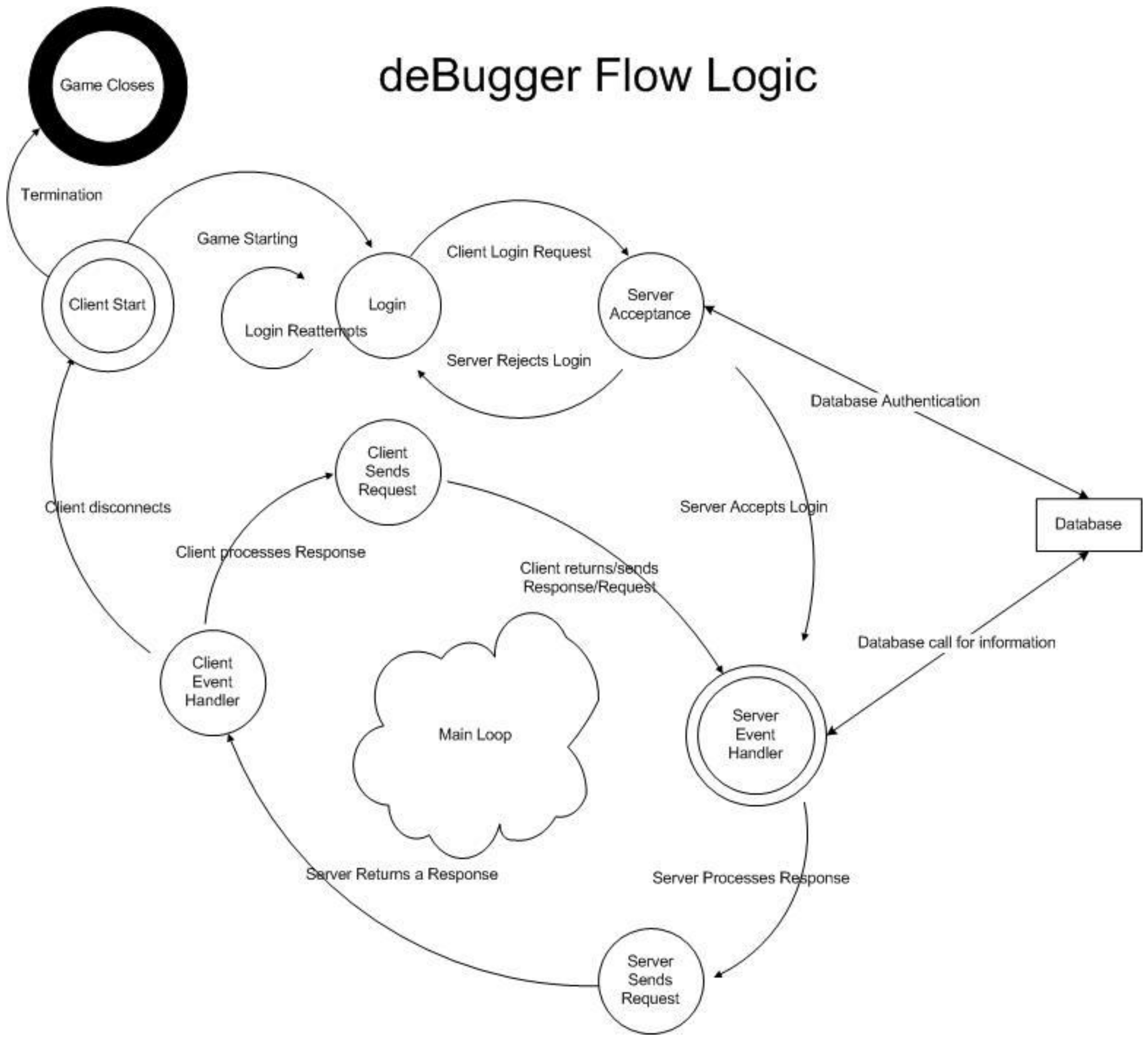


Fig. 3.a-2- deBugger Flow Logic used to show the cycle of information being passed between *Client* and *Server*
Note: As before mentioned, the Bug Server is a modified version of the client, thus follows the same cycle, the only difference is Client Event Handler, which in this case, will be the same type as the Server Event Handler; a multi state process (dealing with more than one entity)

Clients that log in authenticate with the deBugger Server before they are able to play the game. Logging in is sent as a request just as all other subsequent requests. Once a server responds with either an acceptance of the login, or a rejection, the client can proceed to playing the game.

Client / Bug Server:

Once a client or bug server connects with the deBugger server and is properly authenticated, events are raised when certain actions are performed. Clicking on the screen for example, raises an event on the client for it to begin moving the player. While the player is in motion, the player will update the server through an event on its current position. This is done through a task (loop) that occurs at regular intervals. Both the bug server and the client act the same in regards to events and how they are raised. The server knows from the login of the bug server, that it is not a player and should therefore load the respective class to handle multiple updates through a single connection.

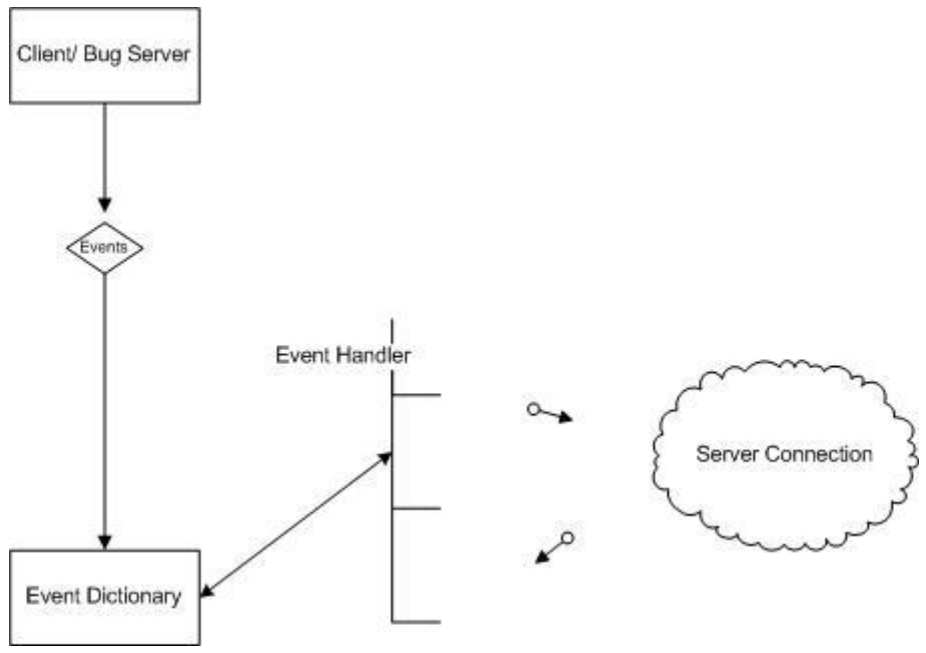


Fig. 3.a-3- Events are handled in both the Client and Bug Server in the same manner. To the event manager, there is no difference if there is one, or ten bugs; they are all handled the same.

Server:

The deBugger server has a thread for each connection. This allows the main deBugger server to concentrate on forming and removing connections, while allowing each of the tasks to deal with the packets sent by the client. Each of these connections listens to the specified connection that was created when the player or bug server logged in with, for updates from that client. Once this is done, the server will update every other connected client (depending on event), and put this information into the queue that each connection maintains for the next heartbeat. Also, at each heartbeat, the server shall send the client all of the queued updates since the heartbeat poll request.

Question Creator:

The question creator has access to the mySQL database and makes use of the tables included within to update the deBugger game on the various questions that are available to players. Since questions are loaded on request, this allows the game to retrieve a newly approved question without restarting. This dynamic interaction is possible through the use of persistent data from the database.

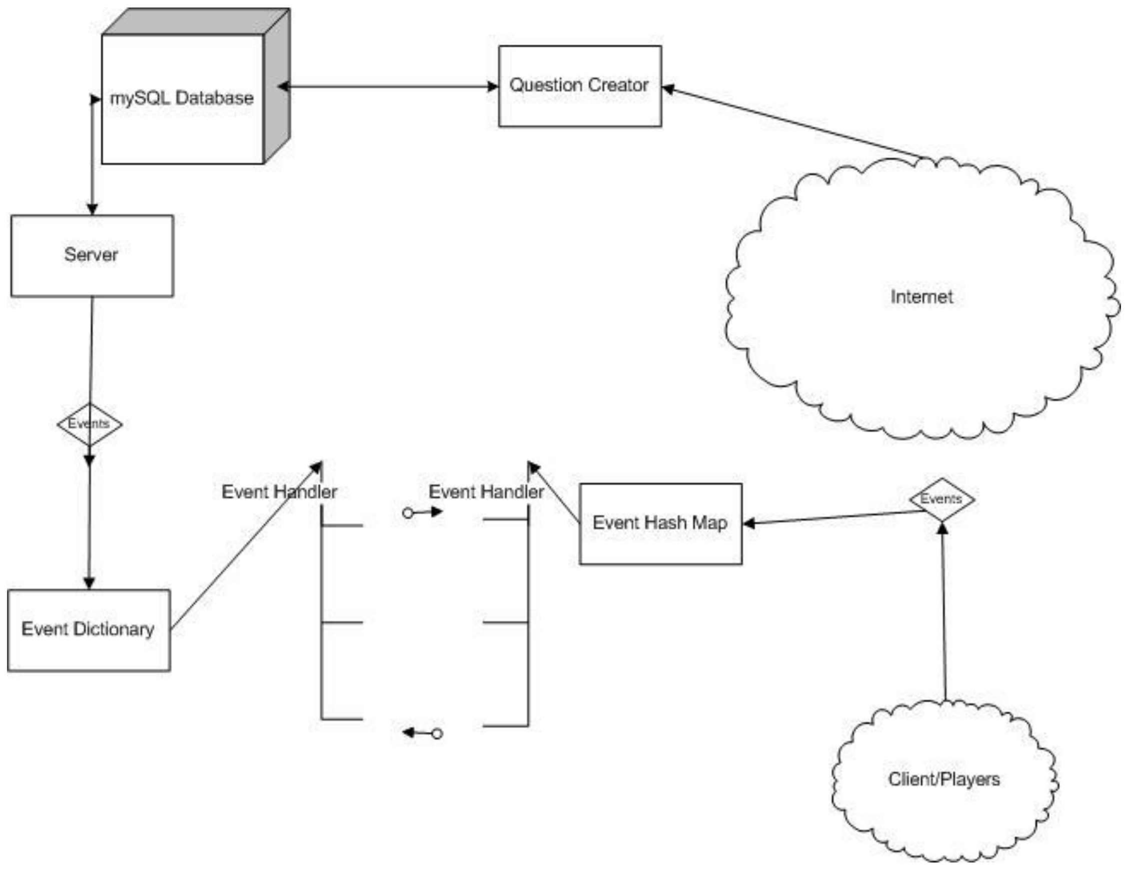


Fig. 3.a-4- Server client interaction. This is viewed from a single connection. Multiple clients will have the same view because of the threaded nature of the server.

Database:

Because the database is a packaged application, and the tables are better suited described in the respective section, this will only give an overview for the database. deBuggers database uses various tables to keep track of information in the game. Some of the tables include the user table, which keeps track of the players' information while they are both logged in, and offline. The equipment table, used to determine drops when a bug dies. Quiz questions for when a bug attacks a player. Experience table to determine how much experience is required for the next level up. Finally, bug drops which are used to determine what a bug drops when it is killed.

Communication:

Communication between the client and server is done either in Python, or in Java, depending on the server. A packet of data must contain: A protocol type, which will signify which event has been called, and thus, what function should be used to process such an event, and the information that it contains. Because the receiving end of the packet can retrieve the Event ID from the packet, which always comes first, the ordering of information in the packet can be obtained without it being stored in the packet itself. A typical packet will look something like the following:

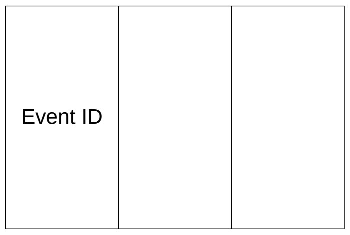


Fig. 3.a-5- Event ID holds the event for the receiver to look up using their lookup method, with the two extra boxes the information it contains.

Summary-

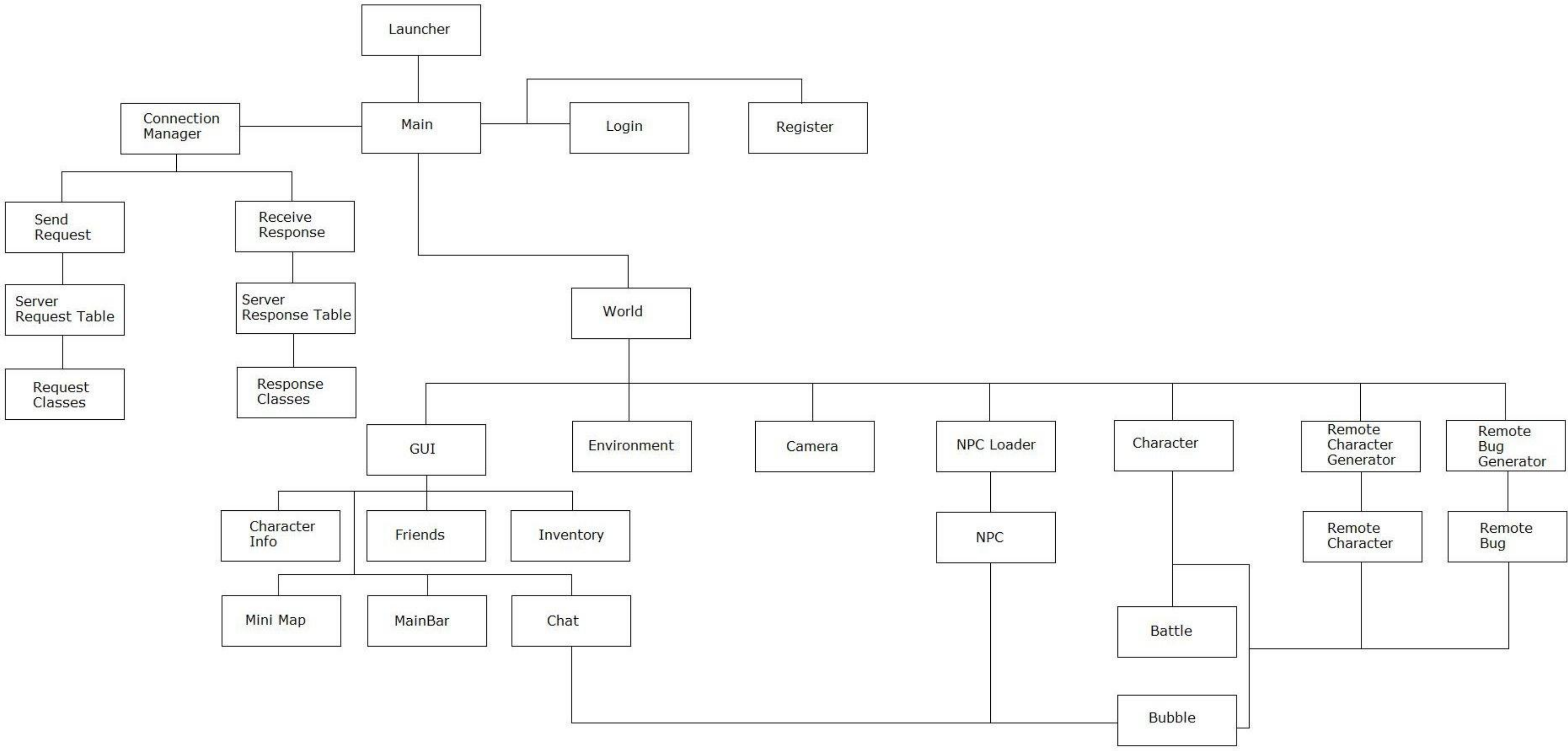
The client is written in Python with Panda3D assisting in various graphical tasks such as collision detection and scene display. The multi threaded deBugger server keeps track of all clients, as well as bugs, that are connected to the game and updates each of the clients as well as bug server when the clients or bug server sends a heartbeat to the server. The bug server is a modified client that keeps track of multiple bugs in the game, updating the deBugger server on the movement and status of each bug, just like a player. It uses one single networking connection to update the deBugger server, unlike the client with its single connection used for a single player. The database is a mySQL database that stores information in tables to be used by the server or question creator. The question creator website connects to the database to update the questions and approve questions that are already in the database so that they can be used by the game.

3.b UML Diagrams

3.c Architecture Design For Each Team

3.c.i Architecture Overview of Client Team

Almost all of the components send requests to the server. The reason for the hierarchichal structuring in the architecture diagram is that the parent component can be accessed from either of the components. Hence, as it can be seen from the diagram, connection manager exists in the main class. All other components exists under differnet packages under the main class. Since they are connected to the main class, they can access the connection manager via the "dot" dereferencing notation. Hence, if we want to access the connection manager from the Character class, all we have to do is "self.world.main.cManager".

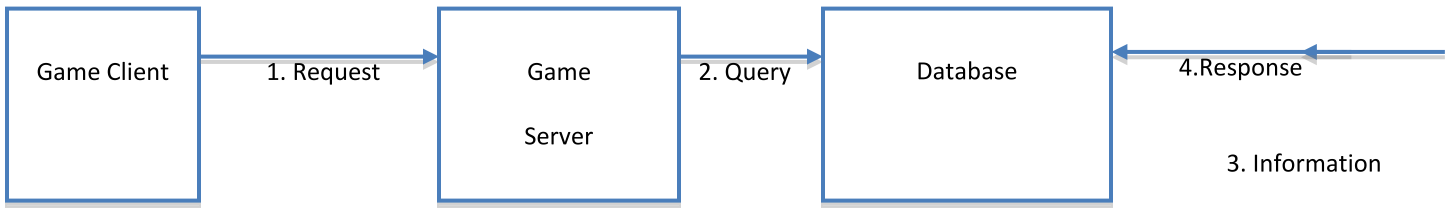

Launcher
Main
Connection Manager
Login
Register
Send Request
Receive Response
Server Request Table
Server Response Table
Request Classes
Response Classes
World
GUI
Environment
Camera
NPC Loader
Character
Remote Character Generator
Remote Bug Generator
Character Info
Friends
Inventory
NPC
Remote Character
Remote Bug
Mini Map
MainBar
Chat
Battle
Bubble

3.c.ii Architecture Overview of Server Team

3.3.2 Server Architecture

Our game works by communicating with three programs. The first is the game client, which the player uses on their computer to play the game. The second is the game server which will communicate with our game client and also get information from the database. Last is the database that stores our players information things such as experience, where they last log out from, health, experience.

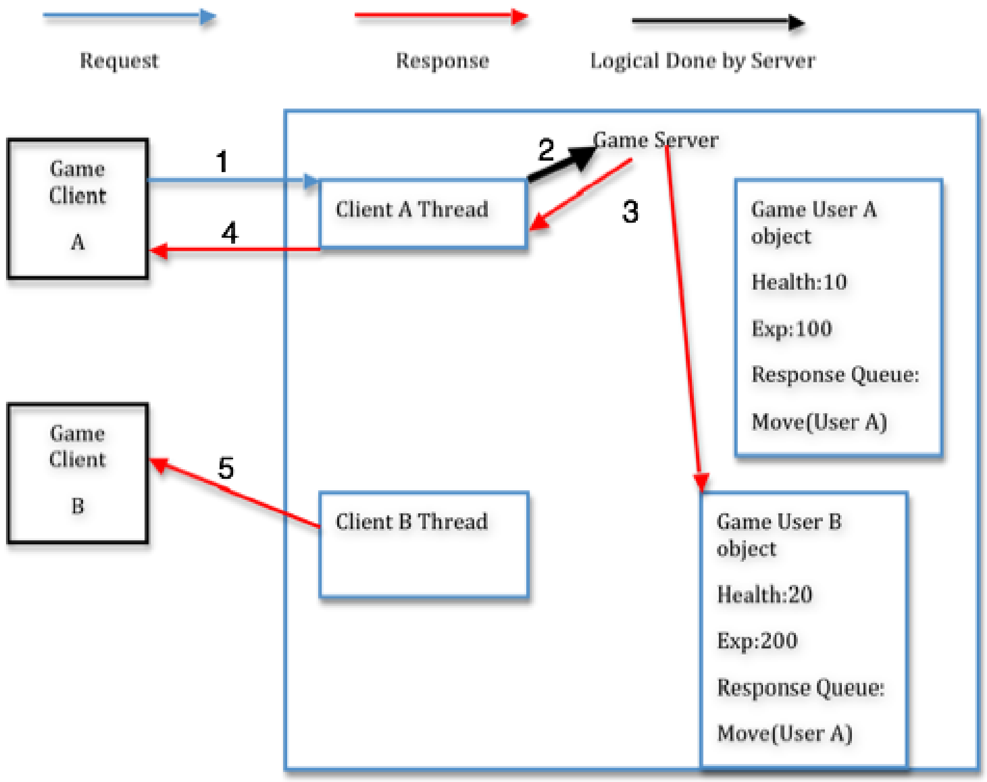
Overview of deBugger Game



The game server in the middle and does the communicating for both the game client and the Database. When the client starts up, the game server will create a thread for it. This thread will have the outputstream coming from the game client. The Game server at this point will wait till the client sends a request to the game server. The game server will then handle the request depending on what the kind of request was asked. At this point the game server will do one of two things, it will go into the database and retrieve information or store information to and from the database using the GameDB.java class methods or it will update the every game client response queue or do a both updating the game client response queue and getting information.

Next how the game server works actually works. When the game client first connects, the game server creates a thread from the GameClient.java for them but also a GameUser.java object. The GameClient.java just creates a thread that will listen on the port for incoming request. It also will continuity check to see if the game client is alive, if it's not it will force it off the game server. The GameUser.java object contains information about the player character. Things such as their user id, health, location on the map, map id, response queue and other such personal information. One of the most important things in the GameUser object is the response queue. This is the thing that lets other game clients know what's happening in the server and about the player.

Handling of Request and Response from Game Client to Game Server

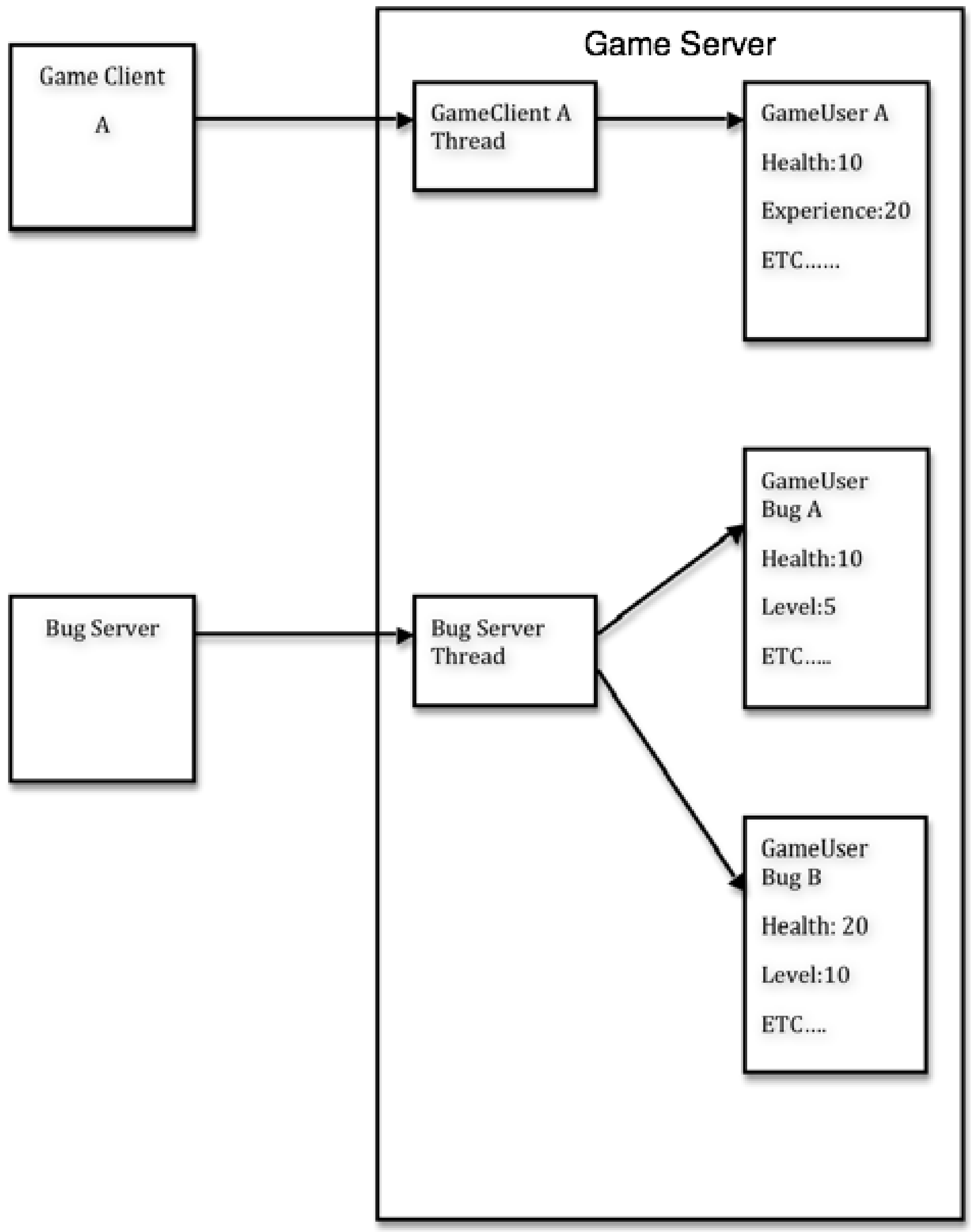


In the picture, starting at 1 the game client A sends the game server a request. The game server receives the request from its game client object. At 2 the game server does some logic and handles the request and creates the corresponding response. At part 3 two things happen, A response is sent back to the original requestor, which is client A, but also a response is created for game user B, which is owned by game client B object. This is to tell the other clients in the game that the client has moved to another location. At 4 the move response is sent back to the player. At 5 the game client B response queue is handled and tells game client B that game client A has moved to another location, this happens really fast so think of part 4 and 5 as one moment.

There are other classes in the game server library. Most of them don't require any kind of modification since they involve sending packets, or some kind of information handling of the DB, which don't really need to be handled modified at all. The constants.java is basically all the constants request and response shared by the client and server. So if a new protocol request and response needs to be made it should be added here. The GamePacket.java and GamePacketsSream.java are classes that help send back information to the client. They have to do with big endian and small endian and the way they PyDatagrams are sent. The GameDB does the communicating with the database, like sending queries.

Some new features that were added to the server architecture are bugs, booting up the bug server, and rebooting the bug server when it launches. First we'll start with the bugs on the server. Bugs are NPCs in our game. In our game we have a bug server that controls these bugs. The bug server is nothing more then a modified game client that runs on some kind of A.I. The game server handles these bugs are GameUser objects except for the fact that these bugs are only controlled by one game client which is the bug server. Here a diagram to show the difference between the game client and the bug server.

Difference between Player and Bug Server overview

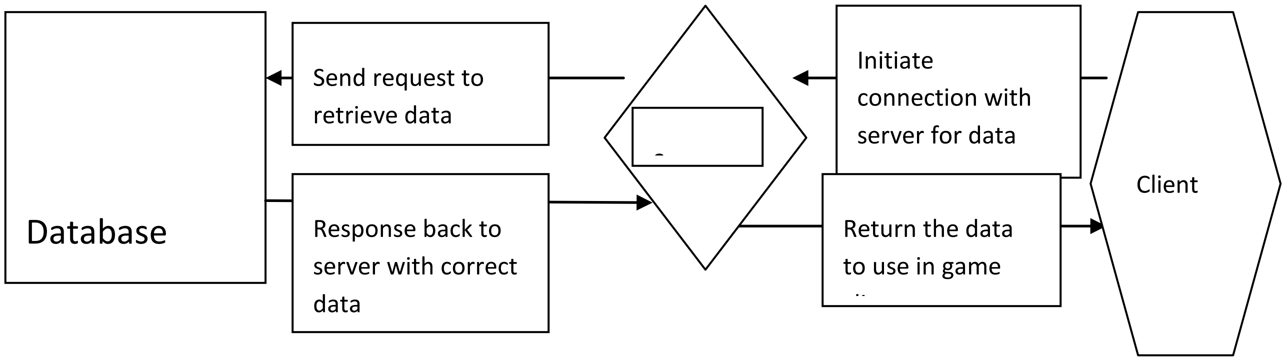


The arrows just point which client owns which GameUser objects. As you can see the player only can control one GameUser object in the game server. As for the Bug server, it sends information to different bugs in the game server. How it does this by request a create bug request? The game server will create a GameUser object with the following bug information given from the Bug Server. All bugs have a user id of 1000 or greater so they can be easily detected as bugs.

When the game server is first launched, it will launch a sub-process with the help of BugServerProcess.java class. On launch the server will execute a thread to run the subprocess. The server will look in the game config file for the bug server path. If it's not found it will say where it can't find the server. If it does find the server it will launch bug server. It takes about 30 seconds or so for the bug server to completely run and create bugs on the server but it will output information when it does. The bug server logins into the game with the user name "green". This is useful to detect if the active thread is alive. If the bug server ever dies it will be re-launched by the game server. In the GameClient.java class if the user id "green" is not detected in the active thread list, the server will remove all the bugs, and then kill the old process and restart create a new thread and start it again.

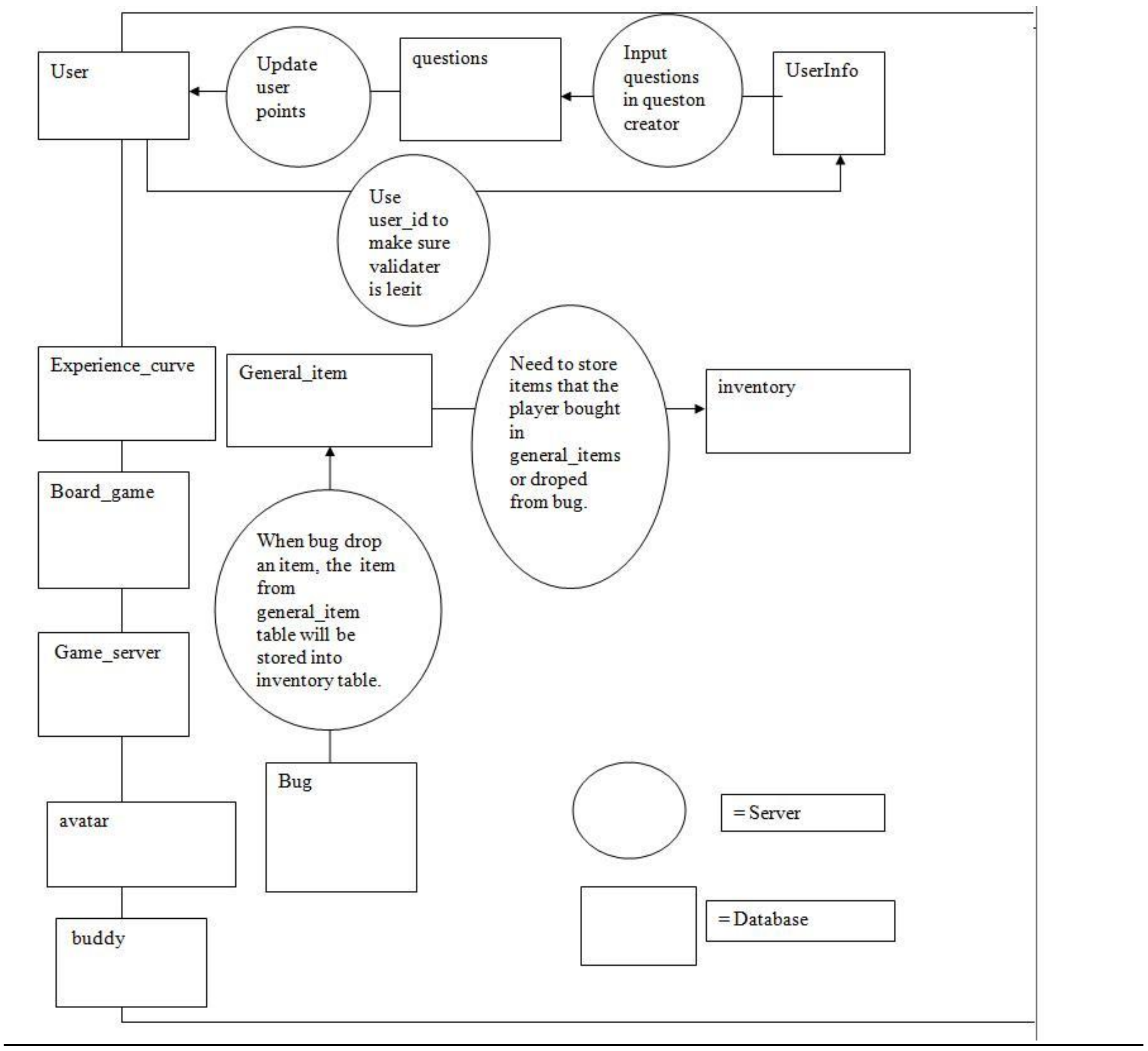
3.c.iii Architecture Overview of Database Team

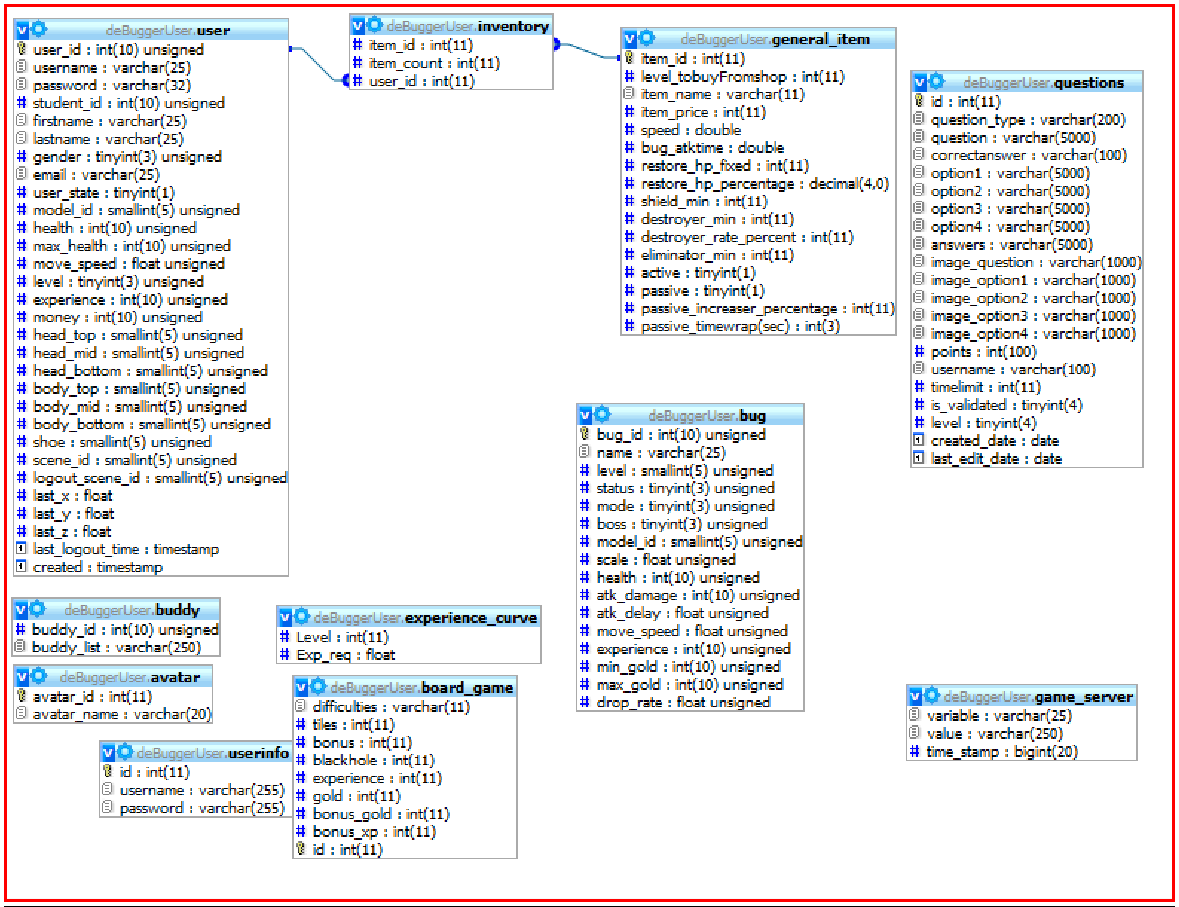
Fig.1 Showing the basic pattern how database is connected to Client via Server.


Database
Send request to retrieve data
Response back to server with correct data
Initiate connection with server for data
Return the data to use in game
Client

As we can see in Fig.1, game client will initiate the connection with server to retrieve the data. We can say that server is just a carrier that will send the data which is retrieved from database to game client. This basic pattern will be similar to all of our data tables because database is a place where we store all data which will be using in our game client. In deBugger serever code, GameDB.java is the file which will be senting the request and receive the information from the database. In addtion, most of the update will also be done in this file. For example, if your character is dead or out of health point, this server code will update the character's health point to zero.

So, I will go into details about the particular table which is connected to server and question creator website since Fig.1 is just showing introduction to our architecture of database. The following figure will show us how many tables are in deBugger database and the connection between tables.


User
Update user points
questions
Input questions in queston creator
UserInfo
Use user_id to make sure validater is legit
Experience_curve
General_item
Need to store items that the player bought in general_items or droped from bug.
inventory
Board_game
When bug drop an item, the item from general_item table will be stored into inventory table.
Game_server
Bug
avatar
= Server
buddy
= Database



deBuggerUser.user
- user_id : int(10) unsigned
- username : varchar(25)
- password : varchar(32)
- student_id : int(10) unsigned
- firstname : varchar(25)
- lastname : varchar(25)
- gender : tinyint(3) unsigned
- email : varchar(25)
- user_state : tinyint(1)
- model_id : smallint(5) unsigned
- health : int(10) unsigned
- max_health : int(10) unsigned
- move_speed : float unsigned
- level : tinyint(3) unsigned
- experience : int(10) unsigned
- money : int(10) unsigned
- head_top : smallint(5) unsigned
- head_mid : smallint(5) unsigned
- head_bottom : smallint(5) unsigned
- body_top : smallint(5) unsigned
- body_mid : smallint(5) unsigned
- body_bottom : smallint(5) unsigned
- shoe : smallint(5) unsigned
- scene_id : smallint(5) unsigned
- logout_scene_id : smallint(5) unsigned
- last_x : float
- last_y : float
- last_z : float
- last_logout_time : timestamp
- created : timestamp

deBuggerUser.inventory
- item_id : int(11)
- item_count : int(11)
- user_id : int(11)

deBuggerUser.general_item
- item_id : int(11)
- level_tobuyFromshop : int(11)
- item_name : varchar(11)
- item_price : int(11)
- speed : double
- bug_atktime : double
- restore_hp_fixed : int(11)
- restore_hp_percentage : decimal(4,0)
- shield_min : int(11)
- destroyer_min : int(11)
- destroyer_rate_percent : int(11)
- eliminator_min : int(11)
- active : tinyint(1)
- passive : tinyint(1)
- passive_increaser_percentage : int(11)
- passive_timewrap(sec) : int(3)

deBuggerUser.questions
- id : int(11)
- question_type : varchar(200)
- question : varchar(5000)
- correctanswer : varchar(100)
- option1 : varchar(5000)
- option2 : varchar(5000)
- option3 : varchar(5000)
- option4 : varchar(5000)
- answers : varchar(5000)
- image_question : varchar(1000)
- image_option1 : varchar(1000)
- image_option2 : varchar(1000)
- image_option3 : varchar(1000)
- image_option4 : varchar(1000)
- points : int(100)
- username : varchar(100)
- timelimit : int(11)
- is_validated : tinyint(4)
- level : tinyint(4)
- created_date : date
- last_edit_date : date

deBuggerUser.bug
- bug_id : int(10) unsigned
- name : varchar(25)
- level : smallint(5) unsigned
- status : tinyint(3) unsigned
- mode : tinyint(3) unsigned
- boss : tinyint(3) unsigned
- model_id : smallint(5) unsigned
- scale : float unsigned
- health : int(10) unsigned
- atk_damage : int(10) unsigned
- atk_delay : float unsigned
- move_speed : float unsigned
- experience : int(10) unsigned
- min_gold : int(10) unsigned
- max_gold : int(10) unsigned
- drop_rate : float unsigned

deBuggerUser.buddy
- buddy_id : int(10) unsigned
- buddy_list : varchar(250)

deBuggerUser.experience_curve
- Level : int(11)
- Exp_req : float

deBuggerUser.avatar
- avatar_id : int(11)
- avatar_name : varchar(20)

deBuggerUser.userinfo
- id : int(11)
- username : varchar(255)
- password : varchar(255)

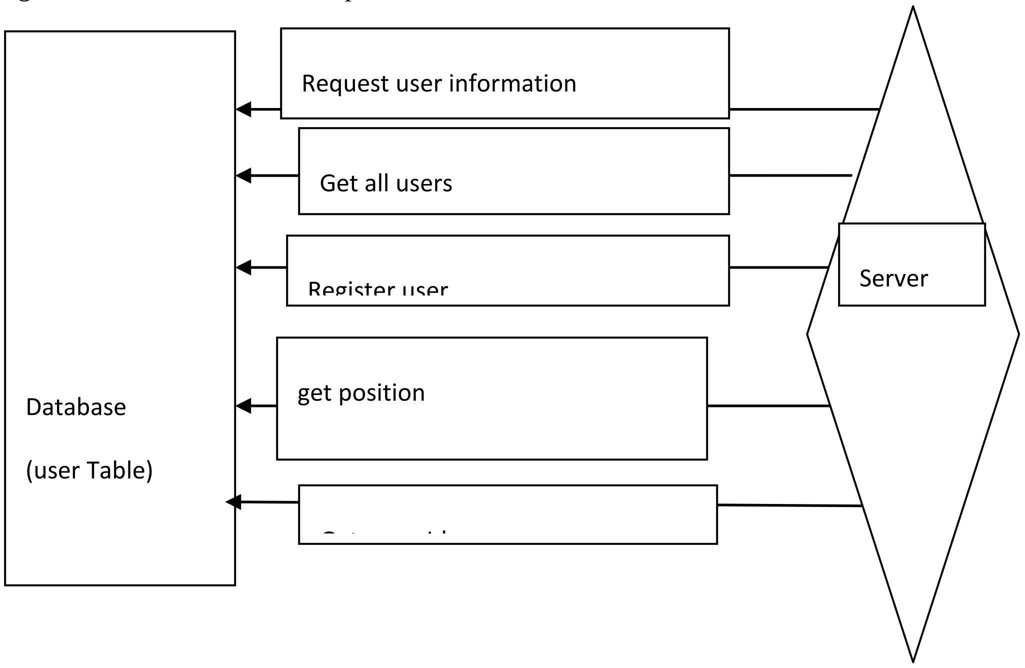
deBuggerUser.board_game
- difficulties : varchar(11)
- tiles : int(11)
- bonus : int(11)
- blackhole : int(11)
- experience : int(11)
- gold : int(11)
- bonus_gold : int(11)
- bonus_xp : int(11)
- id : int(11)

deBuggerUser.game_server
- variable : varchar(25)
- value : varchar(250)
- time_stamp : bigint(20)

<u>Showing the big picture of deBugger database</u>

<u>Architecture of "user" Table</u>

Fig.2 Zooming into user table from server's point of views

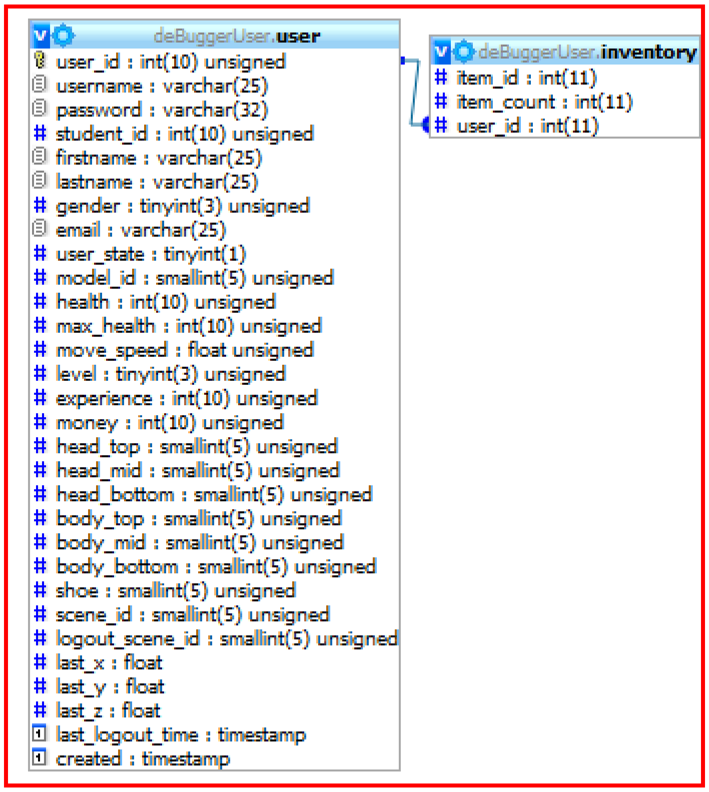


User table from the database take a huge role in this game and a lot of tasks which are done in this table has administrative purposes as u can see in Fig.2. When the player

MMORPG game. We need to distinguish with the player with other player who are also using the game. Afte the player type in username and password, server team will validate username and password. If username and password does not exist or does not match on the database, the player have to retype them again.

If the player successfully log into the game, all of user information will be retrieved from this table. User information will include model_id, last coordinate position x of character, the last logout scene_id, money and health points. Since the player might not want to go back to room #1 if he is already in room #10. So, we need to get scene_id before he or she log out at the last time. That is why, we are going to keep track of scene_id information.

Another cool feature we can get from this table is searching all users with same scene_id. With this feature, we can include buddy list and we can notify the player with list of players who are in the same room. The last and most important task will be registering a new user. When server creates a new user, this table automatically increment user_id for a new user which means each new user will get an unique and new user_id. Then, information for new user will be set by default values.



```
deBuggerUser.user
user_id : int(10) unsigned
username : varchar(25)
password : varchar(32)
# student_id : int(10) unsigned
firstname : varchar(25)
lastname : varchar(25)
# gender : tinyint(3) unsigned
email : varchar(25)
# user_state : tinyint(1)
# model_id : smallint(5) unsigned
# health : int(10) unsigned
# max_health : int(10) unsigned
# move_speed : float unsigned
# level : tinyint(3) unsigned
# experience : int(10) unsigned
# money : int(10) unsigned
# head_top : smallint(5) unsigned
# head_mid : smallint(5) unsigned
# head_bottom : smallint(5) unsigned
# body_top : smallint(5) unsigned
# body_mid : smallint(5) unsigned
# body_bottom : smallint(5) unsigned
# shoe : smallint(5) unsigned
# scene_id : smallint(5) unsigned
# logout_scene_id : smallint(5) unsigned
# last_x : float
# last_y : float
# last_z : float
last_logout_time : timestamp
created : timestamp

deBuggerUser.inventory
# item_id : int(11)
# item_count : int(11)
# user_id : int(11)
```

Picture:1. Showing foreign keys between user and inventory table.

The last thing that I need to mention for this table is foreign keys. Foreign keys are used to reference two columns in two separated table. The foreign key will make sure you have only one user_id for certain items as you can see in Picture.1. If we don't make it foreign keys, we can have different user_id in user table and inventory table. Then, we might give an item to wrong user_id or gvie twice to both user_id.

Architecture of "inventory" Table

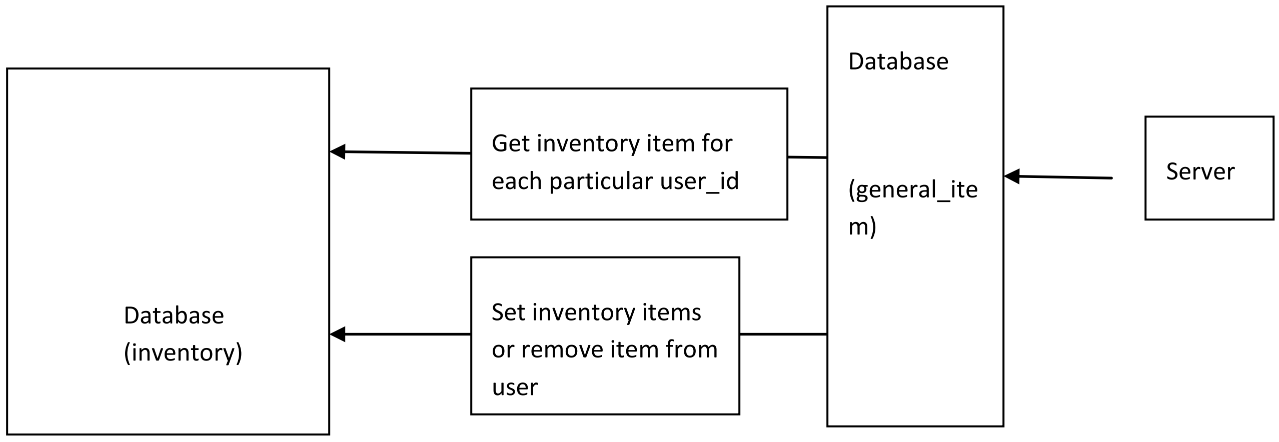
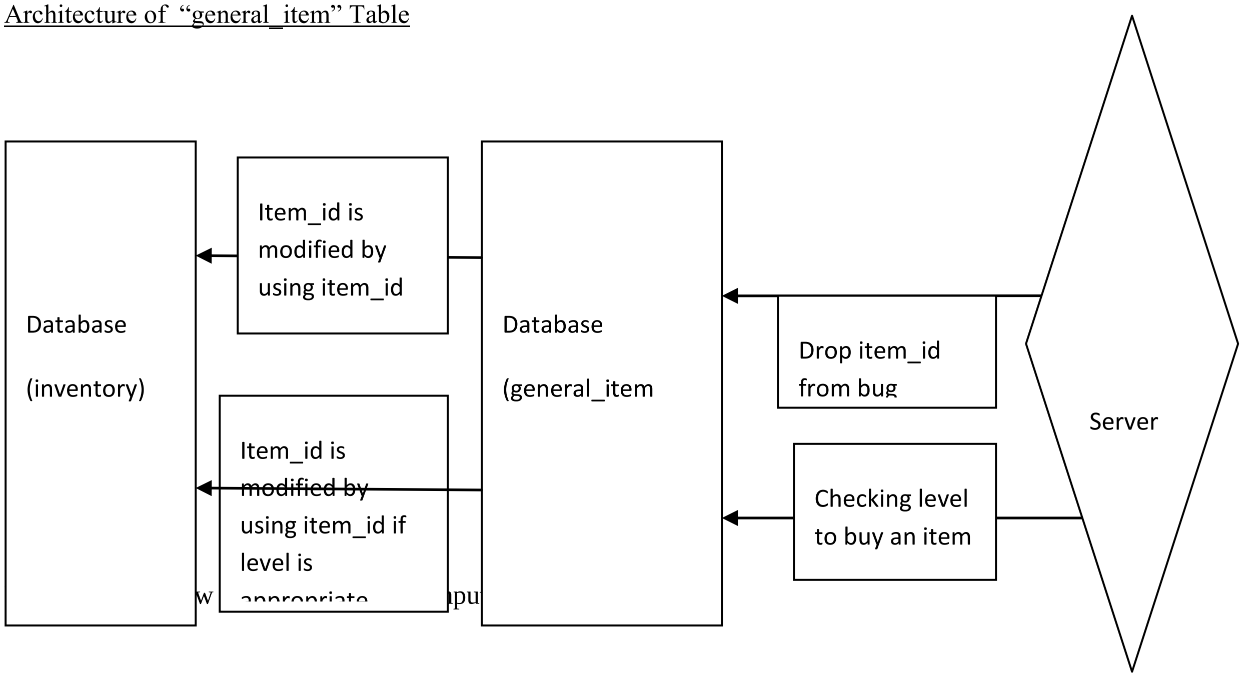

Database (inventory)
Get inventory item for each particular user_id
Set inventory items or remove item from user
Database (general_item)
Server

Fig.3 Showing how inventory is accessed and mutated by server.

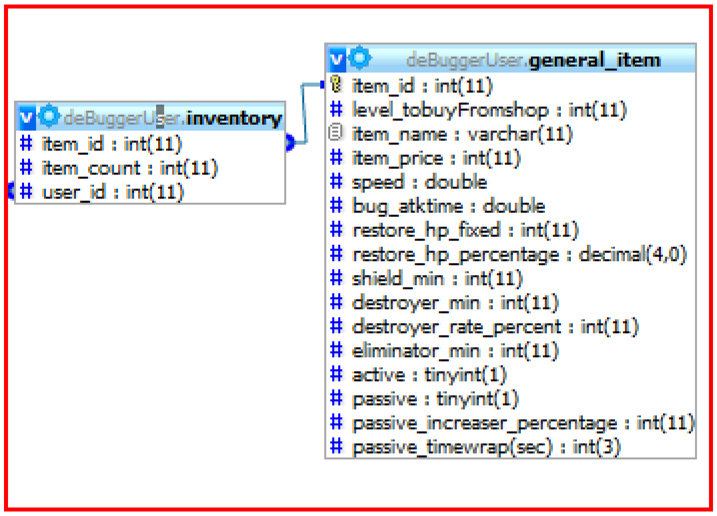
This inventory will be used to store the items which are bought from general_item by the particular user_id. This inventory will be similar to a class in Java/C++ programming language because we will accessing and mutating this inventory table. When we are loading the game, we will be accessing all the items which are bought by the player in the past. If the user buy some item from general_item or the user just used the item such as hp_fixed, we will be adding or deleting items from invenetory table.

<u>Architecture of "general_item" Table</u>


Database (inventory)
Item_id is modified by using item_id
Item_id is modified by using item_id if level is appropriate
Database (general_item
Drop item_id from bug
Checking level to buy an item
Server

In order to get item or accessories, the player need to buy it with money from general_item. But, general_item table has a field which is called level_tobuyfromshop. In addition, this field hold the level limit and it will check the player is appropriate to buy a particular item. That means every item has different level to buy from general_item table.

Another way to get an item from general_item table will be fighting the bug. If the character can answer the question whichever the bug asked, it will get an random present from general_item table. Anyway, both ways to get an item will modify inventory by adding item_count and item_id for the particular user_id. We also make foreign keys for item_id in both general_item and inventory as you can see in Picture.2. I also mentioned in details about foreign key in Architecture of "user" table.



```
deBuggerUser.inventory
# item_id : int(11)
# item_count : int(11)
# user_id : int(11)

deBuggerUser.general_item
  item_id : int(11)
# level_tobuyFromshop : int(11)
  item_name : varchar(11)
# item_price : int(11)
# speed : double
# bug_atktime : double
# restore_hp_fixed : int(11)
# restore_hp_percentage : decimal(4,0)
# shield_min : int(11)
# destroyer_min : int(11)
# destroyer_rate_percent : int(11)
# eliminator_min : int(11)
# active : tinyint(1)
# passive : tinyint(1)
# passive_increaser_percentage : int(11)
# passive_timewrap(sec) : int(3)
```

Picture.2 Showing foreign keys in both inventory and general_item

Architecture of "questions" Table

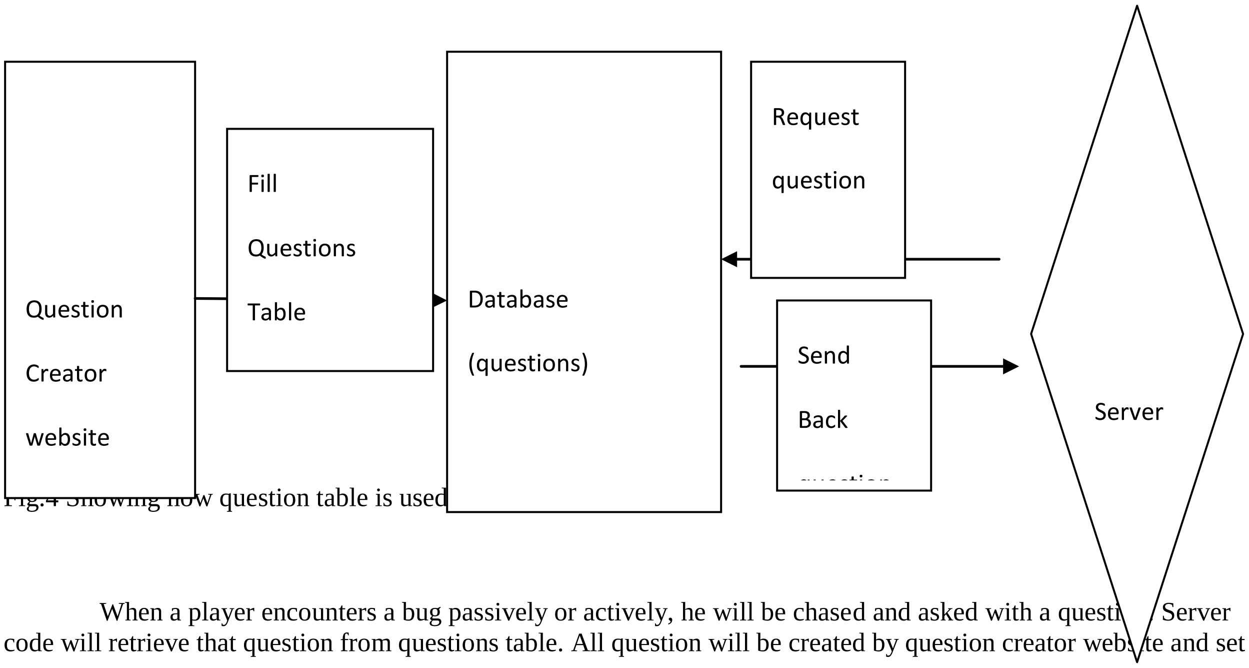


Fig.4 Showing how question table is used

When a player encounters a bug passively or actively, he will be chased and asked with a questi Server code will retrieve that question from questions table. All question will be created by question creator webs te and set it to be validated. Then, the question is ready to be used by bug and the question table is used to retrieved the data as usual.

Architecture of "game_server" Table

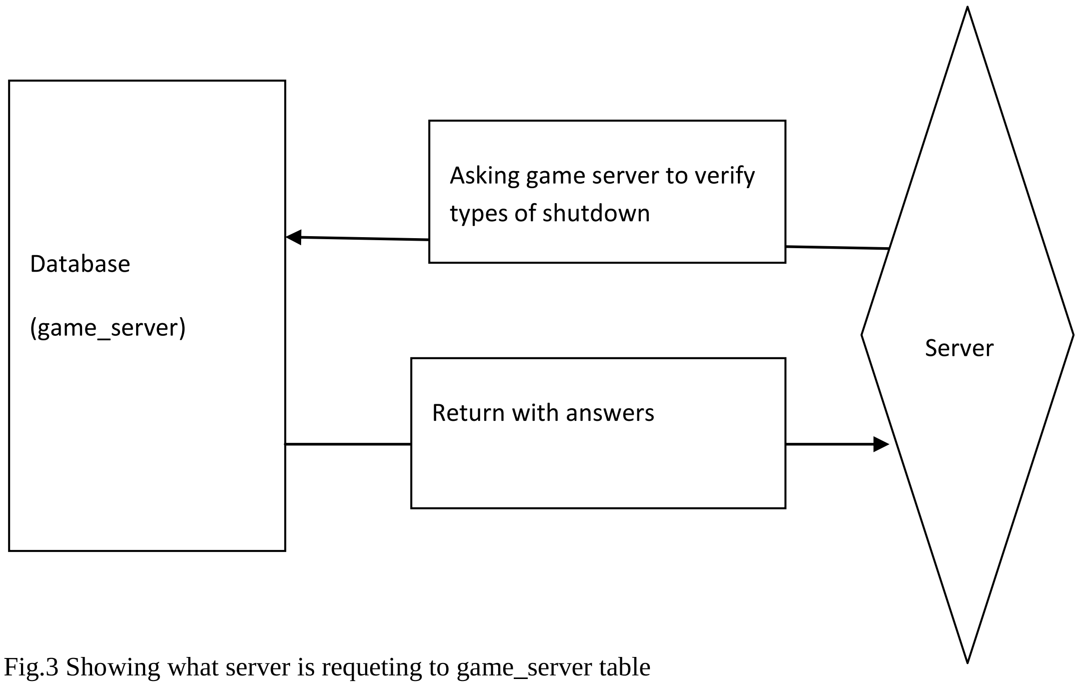
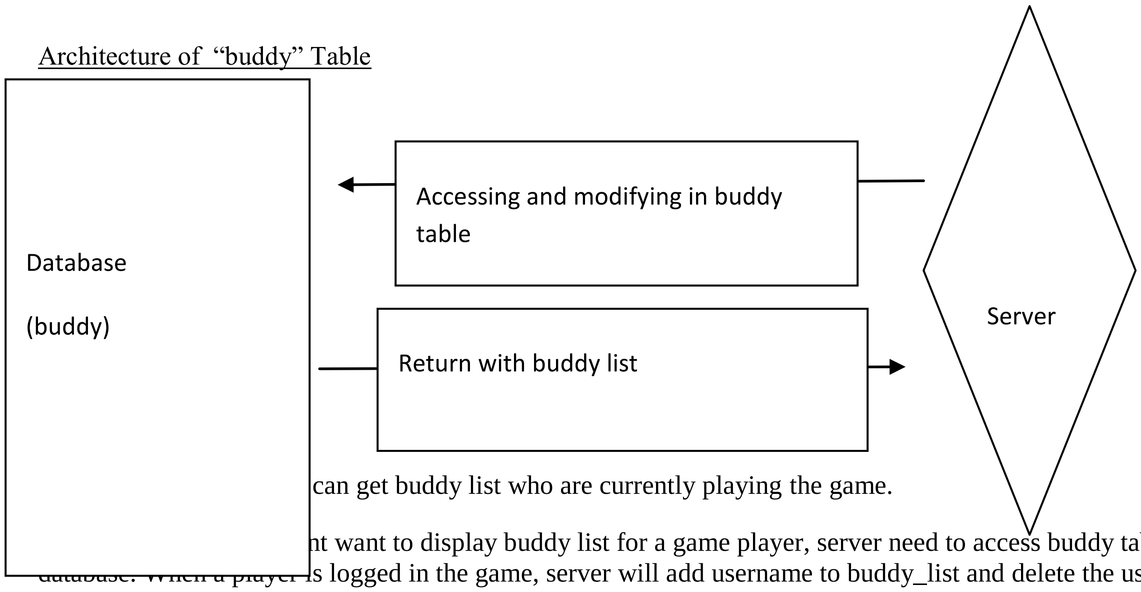


Fig.3 Showing what server is requeting to game_server table

Server is just checking shutdown time by storing variable and values in game_server table. In addition, server can also verify what types of shutdown type was happened because there are emergency shutdown type and maintenance shutdown type in this table.

Architecture of "buddy" Table



can get buddy list who are currently playing the game.

nt want to display buddy list for a game player, server need to access buddy table from the database. When a player is logged in the game, server will add username to buddy_list and delete the user list if the user is not playing the game at all. So, they can retrieve the data whenever they need it.

Architecture of "bug" Table

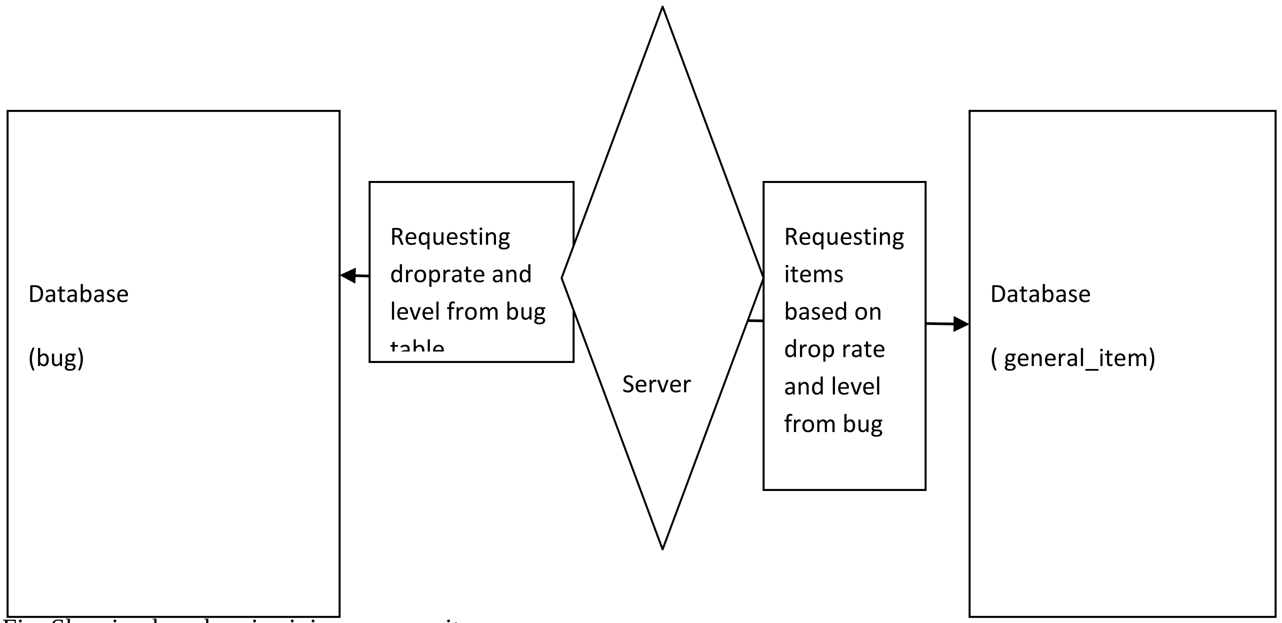


Fig. Showing how bug is giving away an item.

When a player successfully kill a bug, he will gain random items from general_item. As usual, server codes will be checking player's level to determine appropriately. In the server codes, the way we differentiate between bug types is to check drop rate sicne low-level bug will have very low drop rate. Dependiong on drop rate, bug table will be inserting items to inventory table.

Architectur of board_game, avatar, userinfo and experience_curve

These tables are there for future use and we implemented from game concept design's draft. Since these table are not implemented in server code, we just leave them blank for expansion and modification in next level of game. Question creator website is using userinfo table for importing questions. Avatar table will be used to store avatar name and avatar id for future use. Also, board_game is implemented according to game concepte design's draft. The last one, experience_curve will be used as a reference when a user needs to upgrade to his or her level because a level is deteremined by experience.

4.c.iv Architecture Overview of Testing Team

4.c.v Architecture Overview of Launching Team

The technologies used to implement Debugger's launch website consisted of HTML, CSS, Javascript and PHP. Netbeans, an open-source integrated development environment tool was used for developing the site.

Website Architecture

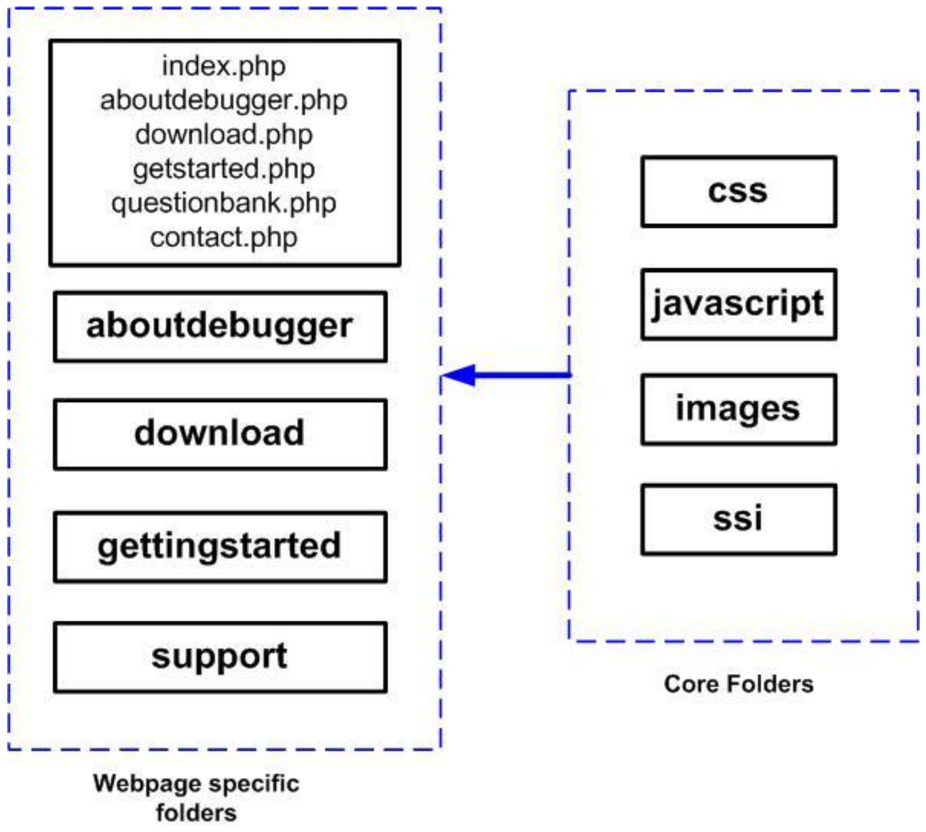


Figure 3.c.v.i – Debugger's launch website architecture diagram

The above figure shows the architecture diagram of the launch website which consists of the following core folders which are shared by all web pages:

- css
- images
- javascript and
- ssi

In addition to above core folders, the website also makes use of the following folders which is used to store web pages specific to them and consist of:

- adoutdebugger
- download
- gettingstarted and
- support

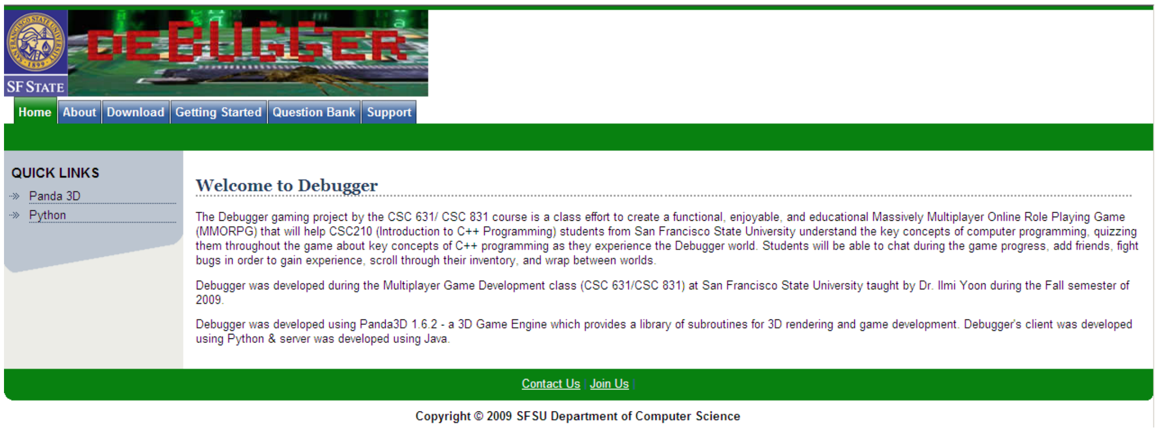
Following is a brief explanation of the purpose of core folders:

css – As the name indicates, this folder is used to store all the cascading style sheet files which define the display of HTML elements. Currently, all the web pages make use of 2-column display format for the main content of the page. Lastly, all the web pages make use of external style sheets.

images – The images folder is used to hold images pertaining to Debugger's website which include banner image, SFSU logo and other design images. Images specific to debugger game are stored in a folder named debuggerui placed under the images folder.

javascript – This folder as the name indicates is used to store javascript files. As of now the website is making use of only one javascript file i.e. navigation.js. By making use of this file the necessary hover effect for the navigation tabs is obtained. In addition to this, the currently clicked navigation tab is indicated by making that tab look larger compared to other tabs. Lastly, all the web pages on the website refer externally to navigation.js file in the <head></head> tag of .php files.

Below is the snapshot of launch website's home page which is indicated by the raised Home tab. Also the clicked tab appears in green color whereas the other tabs appear in blue color.



Welcome to Debugger

The Debugger gaming project by the CSC 631/ CSC 831 course is a class effort to create a functional, enjoyable, and educational Massively Multiplayer Online Role Playing Game (MMORPG) that will help CSC210 (Introduction to C++ Programming) students from San Francisco State University understand the key concepts of computer programming, quizzing them throughout the game about key concepts of C++ programming as they experience the Debugger world. Students will be able to chat during the game progress, add friends, fight bugs in order to gain experience, scroll through their inventory, and wrap between worlds.

Debugger was developed during the Multiplayer Game Development class (CSC 631/CSC 831) at San Francisco State University taught by Dr. Ilmi Yoon during the Fall semester of 2009.

Debugger was developed using Panda3D 1.6.2 - a 3D Game Engine which provides a library of subroutines for 3D rendering and game development. Debugger's client was developed using Python & server was developed using Java.

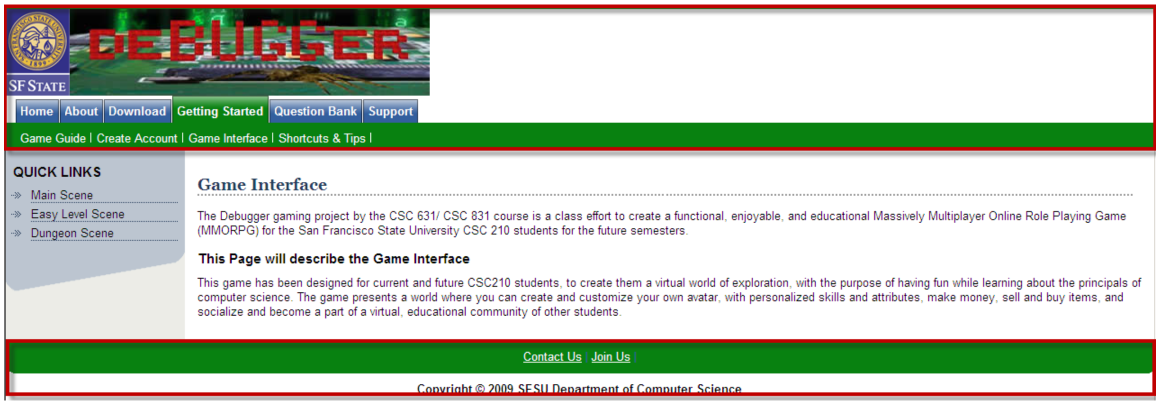
Contact Us | Join Us |

Copyright © 2009 SFSU Department of Computer Science

Figure 3.c.v.ii – Home page of Debugger’s Launch website

ssi – For easy maintenance, debugger’s launch website makes use of server side include files which consist of html code that can be shared by all or some web pages. Generally, the header and footer always remain the same for all the pages on a website due to which the launch website makes use of heading.html to represent the header contents and footer.html to represent the footer content.

Below is the snapshot of Game Interface page under Getting Started tab. The highlighted section in red indicates the header and footer section of the webpage which is common for all the pages on the website.


SF STATE
DEBUGGER
Home | About | Download | Getting Started | Question Bank | Support
Game Guide | Create Account | Game Interface | Shortcuts & Tips |
QUICK LINKS
Main Scene
Easy Level Scene
Dungeon Scene
Contact Us | Join Us

Game Interface

The Debugger gaming project by the CSC 631/ CSC 831 course is a class effort to create a functional, enjoyable, and educational Massively Multiplayer Online Role Playing Game (MMORPG) for the San Francisco State University CSC 210 students for the future semesters.

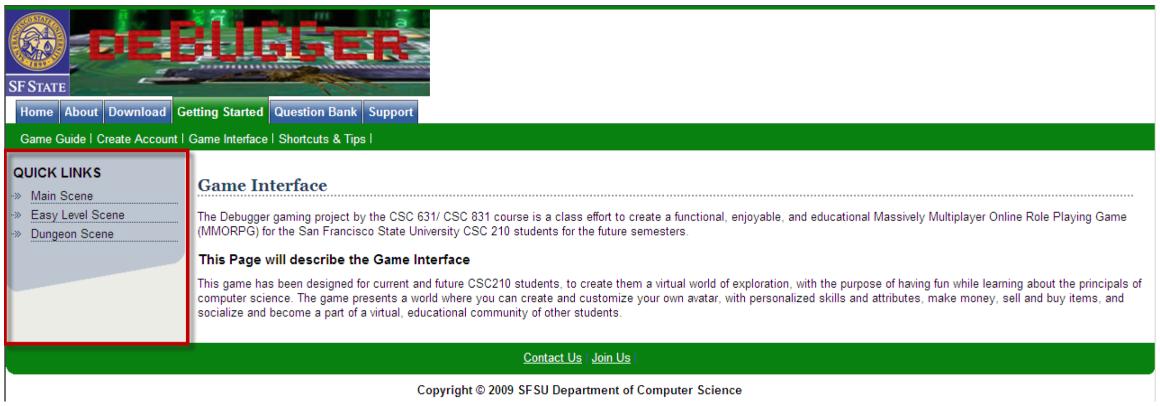
This Page will describe the Game Interface

This game has been designed for current and future CSC210 students, to create them a virtual world of exploration, with the purpose of having fun while learning about the principals of computer science. The game presents a world where you can create and customize your own avatar, with personalized skills and attributes, make money, sell and buy items, and socialize and become a part of a virtual, educational community of other students.

Copyright © 2009 SFSU Department of Computer Science

Figure 3.c.v.iii – Game Interface page of Debugger’s Launch website indicating header and footer section.

Debugger’s Game Interface webpage also makes use of gameinterface.html as a server side include file to represent the left column navigation which is shared with links titled Main Scene, Easy Level Scene and Dungeon Scene. Below is the snapshot in which the highlighted section in red shows the left column navigation.


SF STATE
DEBUGGER
Home | About | Download | Getting Started | Question Bank | Support
Game Guide | Create Account | Game Interface | Shortcuts & Tips |
QUICK LINKS
Main Scene
Easy Level Scene
Dungeon Scene
Contact Us | Join Us

Game Interface

The Debugger gaming project by the CSC 631/ CSC 831 course is a class effort to create a functional, enjoyable, and educational Massively Multiplayer Online Role Playing Game (MMORPG) for the San Francisco State University CSC 210 students for the future semesters.

This Page will describe the Game Interface

This game has been designed for current and future CSC210 students, to create them a virtual world of exploration, with the purpose of having fun while learning about the principals of computer science. The game presents a world where you can create and customize your own avatar, with personalized skills and attributes, make money, sell and buy items, and socialize and become a part of a virtual, educational community of other students.

Copyright © 2009 SFSU Department of Computer Science

Figure 3.c.v.iv – Game Interface page of Debugger’s Launch website indicating the left side column navigation

Chapter 4 - Scope, Tasks and Milestone

4.a - Implementation of Game Concept Team

Ali Ugur

Jordan Mangini

Sara Tily

The gaming project by the CSC 631 course is a class effort to create a functional, enjoyable, and educational MMORPG game for the San Francisco State University CSC 210 students for the future semesters.

As the Game Concept Team, we have created, having in mind the characteristics of online multiplayer games, a potential conceptual foundation for the game to implement with the help of the other teams from the 631 class.

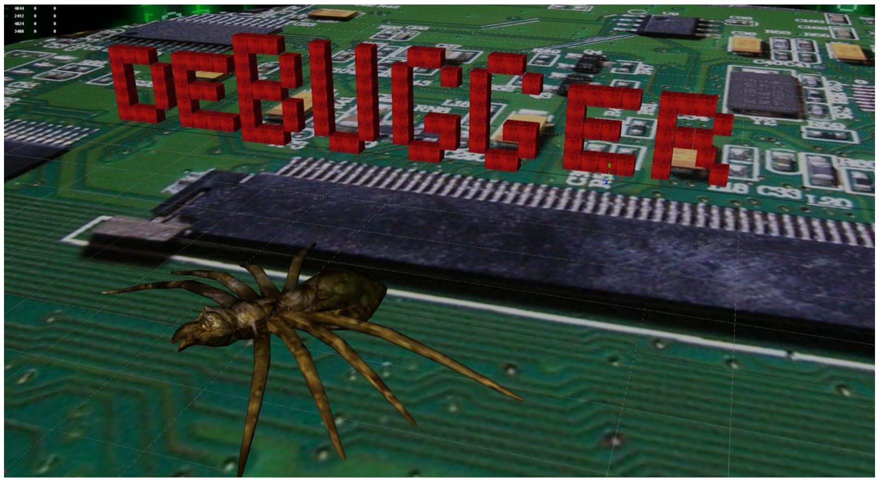
Abstract

This game has been designed for current and future CSC210 students, to create them a virtual world of exploration, with the purpose of having fun while learning about the principals of computer science. The game presents a world where you can create and customize your own avatar, with personalized skills and attributes, make money, sell and buy items, and socialize and become a part of a virtual, educational community of other students.

The Beginning:

(Logging in and creating a character/avatar)

(Implementers: Ali Ugur)



The player will need to create an account and have an avatar assigned to him/her to begin experiencing the game. Once the player installs and runs the game, he or she will be prompted with a log-in window, where the player would put the username and password information to log-in and start playing the game. If the player does not have an account already, then there will be an option to create one. Since this is not just a regular, but an educational game, it would be ideal to be able to keep track of student information, thus the necessary registration information will be as followed;

* First Name
* Last Name
* Student Id
* Username
* Password
* Reenter password
* Email
* Semester taken/taking CSC210

(Should be expendable for future additions...)

Then, the user will be able to choose a specific type of avatar to begin the game and evolve with.

There will be only a handful of avatar to choose from, but the options of customization within the game will be very extended, thus nearly every player will have a unique look.

The Avatar:

(Items and customizations)

(Implementers: Sara Tily and Ali Ugur)

Here is the description for a general avatar growth and customization system.

There are two ways the user will be able to gain items to customize their looks, and gain certain attributes from. These options are;

1. By defeating a bug.

Similarly to other RPG games, when a certain 'enemy' (Bugs in this particular game) are defeated, there is a chance that they might drop what is referred to as 'loot', one or a collection of items including money and accessories. Therefore, when a player defeats a bug, there is a certain percent of chance to get an item(and article of clothing) in return. If the player killed that bug at a lower level, then the chances to get some accessories will be low and the chances to get an accessories will increase as the player defeats higher level bugs.(Explained more in detail later)

2. By purchasing the items from the shop using gold/money.

Certain areas of the map will have shops, where the user will have access to purchase most of, if not all, the articles of clothing for a certain sum of money. Of course, some items will cost more than others.

However, players are not allowed to use these items unless they approach a certain given level, in which they will be allowed to wear the items. This will utilize the users to push their characters to play more and improve and get to certain level, so they can wear these articles of clothing. Here is the level distribution of items;

Level 1- given t-shirt and jean Level 8- will be able to wear customized shirt and pants

Level 14- will be able to wear glasses

Level 20- will be able to wear a belt

Level 28- will be able to wear shoes

Level 35- will be able to wear a hat (normal or sports cap)

Level 50- MAX LEVEL. Character will have aura around him/her.

The characteristics and attributes of these items will be explained later on under the Shop section of this document.

<u>World Map:</u>

(Spawn Room and different difficulties)

(Implementers: Jordan Mangini with Minor help from Ali Ugur)

Once the player logs into his/her account, they will spawn in a what is called a 'Global Map', which consists of three computers to choose from, of which each represents each difficulty of the game(Easy=Old looking, Moderate=Current technology, Hard=Futuristic).

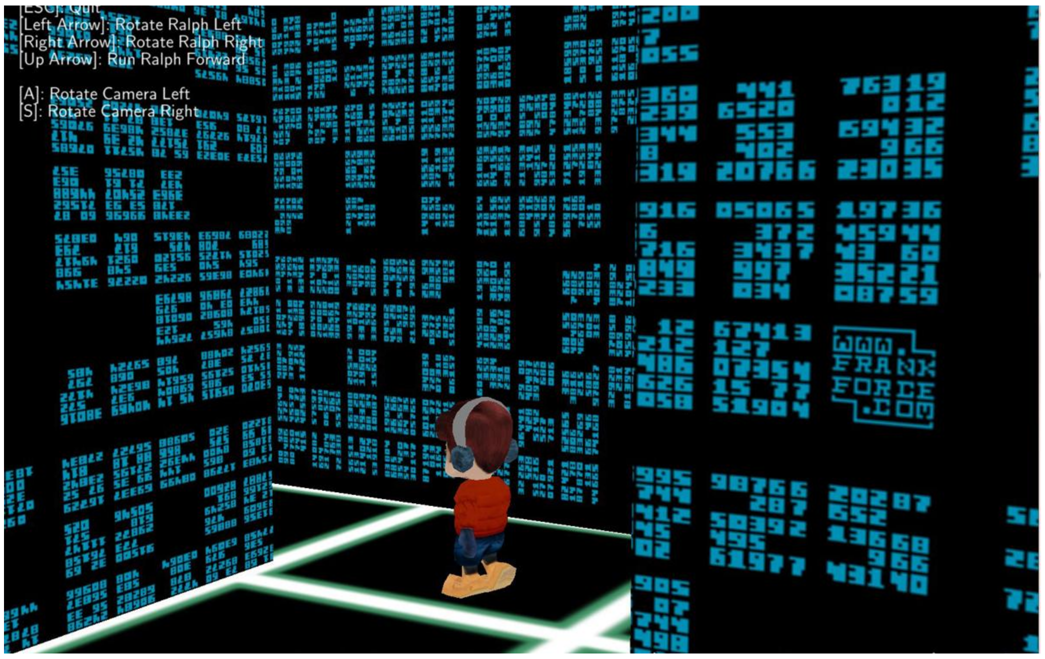
Each computer's map layouts are drastically different, but it will have the same basic theme.

Once a computer is chosen, It will start the player out on the 'desktop', where there is an icon to begin/start a board game, and a 'my computer' icon to enter the computer and begin a journey of the interior of computer and you will have to do some debugging(the hunting).

The motherboard architecture is very complex, so there will be more abstract rooms that follow easier concepts for the target players. a keyboard room, disk drive room, RAM room are all possibilities. The importance is that there will be included the memory rooms, where players can move through room 'arrays' and continue to fight bugs for as long as they want.

Also, there will be qualifiers to determine if we wish the player to move through N number of rooms, on one dimension, and the path home.

The 'dungeon' sections on a map have one exit, requiring the player to clear out every room, and move through all those rooms to get back out.


[ESC]: Quit
[Left Arrow]: Rotate Ralph Left
[Right Arrow]: Rotate Ralph Right
[Up Arrow]: Run Ralph Forward
[A]: Rotate Camera Left
[S]: Rotate Camera Right
WWW.FRANKFORCE.COM

(Inside the 'Easy' Dungeon)

Tied in with these concepts, with more complex computers(harder difficulties), there will be larger memory banks that need exploration, perhaps with more bugs in these rooms.

<u>The Board Game:</u>

(How and Why?)

(Implementers: Sara Tily and Jordan Mangini with Major help from Ali Ugur)

YET TO BE IMPLEMENTED

The board game will be a crucial part of the MMORPG game, but it will not be a stand-alone portion. The board game directly ties in with the 'Bug Hunting' concept of the game, which will be explained more in detail later.

The board game is where a player can either 'join' or 'create' a game, in which several different players will be competing against each other to win by answering right on given questions regarding computer science, and moving through the board with a chance of luck of the dice.

The basic concept of the board game is this;

Players will compete against each other within a randomly generated map in which each player follows a unique path towards a general end point. The players will walk through the path, which is a collection of tiles, by a random die roll. Once the player reaches to the end, they will be the victor, and they will be awarded with a sum of gold, based upon the difficulty setting of the game.

The route will be randomly generated at the beginning of each quiz for each player. However the number of tiles in a route will depend on the level of the difficulty.

Here is the breakdown;

* Easy - 20 tiles, 1 bonus, 1 blackhole
* Moderate - 35 tiles, 3 bonus, 1 blackhole
* Hard - 50 tiles, 5 bonus, 1 blackhole

Each tile will represent a question regarding computer science. With each correct answer given to a question, the player will receive a quantity of Experience Points, which are needed to level up the players character.

There will also be Bonus tiles, which will reward the player with a randomly generated amount of Experience Points or Gold.

Also each unique path will have a blackhole. The blackholes are a trap in each route which if landed on, sends the player back to starting point. The blackholes will always be generated within the second half of each path.

The breakdown of experience and gold points dependent on difficulty is as shown;

Xp:

easy; 1-3 xp per right answer, or 4-10 xp on Bonus.

mod; 3-67 xp per right answer, or 16-150 xp on Bonus.

hard; 75-560 xp per right answer, or 200-1000 xp on Bonus.

gold:

easy; 5-10 gold if the victor, or 3-7 gold on Bonus.

mod; 25-40 gold if the victor, or 20-30 gold on Bonus.

hard; 60-100 gold if the victor, or 50-70 gold on Bonus.

Where the scale from one level to another level will increase based upon this formula:

NewCap = (PreCap)+(PreCap*1/4)

Example:

Level 1-2 : 20xp

Level 2-3: 25xp (20)+(5)...

Since the routes/paths for each player will be created randomly, it would be ideal to have a crtpath function, which is called whenever a player enters the game, thus a path for that player is created randomly, with the given parameter(difficulty).

Finally, there will also be three different styles of maps for these routes to be formed upon. This way players will not always have to play on the same looking board each and every time. The Quiz Creator can pick one of them at the time of creating the quiz.

Player level- Skills

Level 1- None

Level 4- Single reference card

Level 8- Second reference card OR first TA help

Level 14- Penalty opponent 1 block

Level 20- Second TA help OR Penalty opponent 2 blocks

Level 28- Third reference card OR Experience Multiplier (Exp*1.5)

Level 35- Third TA help OR Penalty opponent 3 blocks

Level 45- Experience Multiplier (Exp*2) and the users name is displayed GOLD or RED.

Level 50- MAX LEVEL. Character will have aura around him/her.

The OR implies the character can choose one or the other, never both, that way each character in the end will end up different pros and cons.

<u>The Shops:</u>

(What to buy?)

(Implementers: Ali Ugur)

Here is a list of items that can be bought at the shops within the game;

HP Potions

Small Potion - Restores 20 HP. - 50 Golds.

Medium Potion - Restores 50 HP. - 120 Golds.

Big Potion - Restores 60% of HP. - 250 Golds.

Mega Potion - Restores full HP. - 500 Golds.

BG Skills

Single reference card - 1000 Golds.

Second reference card OR first TA help - 1500 Golds.

Penalty opponent 1 block - 2000 Golds.

Second TA help OR Penalty opponent 2 blocks - 3000 Golds.

Third reference card OR Experience Multiplier (Exp*1.50) - 4000 Golds.

Third TA help OR Penalty opponent 3 blocks - 6000 Golds.

Experience Multiplier (Exp*2) and the users name is displayed GOLD or RED.

- 10000 Golds.

Character will have aura around him/her. - 20000 Golds.

BH Skills

ACTIVE:Will be user triggered to use.

Shield 1 - Escape a question. Once every 15 minutes. - 1000 Golds.

Shield 2 - Escape a question. Once every 10 minutes. - 2000 Golds.

Shield 3 - Escape a question. Once every 5 minutes. - 4000 Golds.

Destroyer1 - automatically solves a bug. Once every 45 minutes. Success rate of 50%.

- 2000 Golds.

Destroyer2 - automatically solves a bug. Once every 30 minutes. Success rate of 75%.

- 4000 Golds.

Destroyer3 - automatically solves a bug. Once every 15 minutes. Success rate of 90%.

- 6000 Golds.

Eliminator1 - one of the choices of the answers will be eliminated. Once every 20 minutes.

- 1000 Golds.

Eliminator2 - one of the choices of the answers will be eliminated. Once every 15 minutes.

- 2000 Golds.

Eliminator3 - one of the choices of the answers will be eliminated. Once every 10 minutes.

- 4000 Golds.

PASSIVE:Will always be enabled.

Increaser1 - possibility of a drop increases. +5% - 4000 Golds.

Increaser2 - possibility of a drop increases. +10% - 6000 Golds.

Time-Warp1 - bug attack time increases. +1 second. - 2000 Golds.

Time-Warp2 - bug attack time increases. +2 seconds. - 5000 Golds.

shirts/jeans

Easy Drops:

Black shirt/jeans - 800 Golds.

Red shirt/jeans - 800 Golds.

White shirt/jeans - 800 Golds.

Blue shirt/jeans - 800 Golds.

Green shirt/jeans - 800 Golds.

Pink shirt/jeans - 800 Golds.

Purple shirt/jeans - 800 Golds.

Brown shirt/jeans - 800 Golds.

Moderate Drops:

Haste shirt - player speed increased by .05 - 2000 Golds.

Haste jeans - player speed increased by .1 - 2000 Golds.

Durable shirt - player HP increased by 2% - 2000 Golds.

Durable jeans - player HP increased by 2% - 2000 Golds.

Time shirt - duration of attack time increased by .5 second - 2000 Golds.

Time jeans - duration of attack time increased by 5 second - 2000 Golds.

Hard Drops:

leather jeans - hp increase by 10%, player speed decreased by .15 - 5000 Golds.

shorts - player speed increased by .2, hp decreased by 12% - 5000 Golds.

210 shirt - speed increase by .2, hp increase by 10% - 6000 Golds.

213 shirt - duration of attack time by 2 seconds - 8000 Golds.

413 shirt - hp increase by 15%, duration of attack time by 1 seconds - 10000 Golds.

<u>BOSS drops:</u>

1337 shirt - hp increase by 25%, duration of attack time by 4 seconds speed increase by .3 - 20000 Golds

glasses

<u>Easy Drops:</u>

Reading glasses - duration of attack time increased by .5 second - 1500 Golds.

<u>Moderate Drops:</u>

Frameless glasses - player speed increased by .1 - 2000 Golds.

sunglasses - player HP increased by 2% - 3000 Golds.

Hard Drops:

Aviators - hp increase by 3%, speed increase by .15 - 4000 Golds.

party glasses - speed increase by .2 - 4000 Golds.

Big glasses - duration of attack time increased by 1 seconds - 4000 Golds.

BOSS Drops:

Future glasses - hp increase by 10%, speed increase by .3, duration of attack time increased by 2 seconds.
- 6000 Golds.

belts

<u>Moderate Drops:</u>

Sturdy belt - player hp increased by 3% - 3000 Golds.

Light belt - player speed increased by .2 - 4000 Golds.

New belt - duration of attack time increased by 1 seconds - 4000 Golds.

leather belt - hp increased by 10%, duration of attack time decreased by 2 seconds. - 4000 Golds.

Hard Drops:

Magical belt - hp increased by 8%, speed increased by .1, attack time by 1 seconds.

- 5000 Golds.

shoes

<u>Moderate Drops:</u>

White shoes - player hp increased by 3%, duration of attack time increased by 1.5 seconds

- 4000 Golds.

Black shoes - player speed increased by .2, duration of attack time increased by 1 seconds

- 4000 Golds.

Hard Drops:

Dragon shoes - hp increased by 15%, player speed increased by .25 - 6000 Golds.

Magical shoes - hp increased by 20%, duration increased by 1.5 second. - 6000 Golds.

210 shoes - hp increased by 20%, player speed increased by .15, duration increased by 2 seconds. - 8000 Golds.

213 shoes - speed increased by .50. - 8000 Golds.

413 shoes - hp increased by 25%, player speed increased by .2, duration increased by 2.5 seconds. - 10000 Golds.

BOSS Drops:

1337 shoes - hp increased by 30%, player speed increased by .25, duration increased by 4 seconds. - 20000 Golds.

hats

<u>Moderate drops:</u>

baseball cap - duration of attack time increased by 1.5 seconds. - 4000 Golds.

gentleman's hat - hp increased by 10%. - 4000 Golds.

cowboy hat - speed increased by .15. - 4000 Golds.

Hard Drops:

magicians hat - hp increased by 15%, player speed increased by .10 - 6000 Golds.

210 hat - - hp increased by 15%, player speed increased by .10, duration increased by 1.5 seconds. - 8000 Golds.

213 hat - hp increased by %30. - 8000 Golds.

413 hat - increased by 20%, player speed increased by .15, duration increased by 2 seconds.

- 10000 Golds.

BOSS Drops:

1337 hat - hp increased by 25%, player speed increased by .40, duration increased by 2.5 seconds. - 20000 Golds.

The player can sell back any of these items to the shop for 25% of the item's price.

<u>The Bug Hunting and Quests:</u>

(Again...How and Why?)

(Implementers: Ali Ugur)

As mentioned before, there is another concept within the game that will be tied directly with the board game. This will be the more common characteristic of an MMORPG game, which is 'monster' hunting. Since this will be a computer science related game, instead of having monsters, we will be having 'Bugs' that will need to be fixed.

Different events, as in quests and Bugs, will appear on different difficulty areas. Within this game concept, the player will be not gaining experience points, but gold from each defeated 'bug'.

A basic idea that will be implemented is that a certain MAIN quest, per difficulty area, must be completed, for the player to access the next area.

Thus, this way for example, the player will not only need to be level 10 to access the Moderate area, but also finish the Easy area's MAIN quest.

Easy Mode:

Lets talk about the Bugs...Easy mode will contain;

*Syntax Lvl 1 - lvl 9: The most common of all bugs. Will be all over the map.

Asks typical Syntax questions...

*Semantic lvl 4 - 11: Second most common of the bugs. Will be in majority of the map.

Asks typical Semantic questions...

*Unknown lvl 8 - 12: These bugs are uncommon, will be in the minority of the map.

Harder, unknown problems that need to be diagnosed...

*BOSS: A single, RARE bug, that needs to be fixed at all costs!

Player needs to be in the corresponding quest to do it.

You fight a certain number of Syntax/Semantics to get to it.

(10 syntax and 10 semantics)

Moderate Mode:

Can be accessed once Easy Boss is beaten, and player is at a level of at least 10.

*Syntax Lvl 12 - lvl 34: The most common of all bugs. Will be all over the map.

Asks typical Syntax questions...

*Semantic lvl 18 - 38: Second most common of the bugs. Will be in majority of the map.

Asks typical Semantic questions...

*Unknown lvl 32 - 40: These bugs are uncommon, will be in the minority of the map.

Harder, unknown problems that need to be diagnosed...

*Trouble lvl 30 - 42: These bugs are very uncommon, will be in the minority of the map.

A lot harder, complex computing/programming problems...

*BOSS 1: A single, RARE bug, that needs to be fixed to fix the BIGGER bug!

Player needs to be in the corresponding quest to do it.

You fight a certain number of Syntax/Semantics/Unknowns to get to it.

(15 syntax and 15 semantics and 5 unknowns)

*BOSS 2: A single, RARE bug, that is the biggest bug in the known area!

Player needs to be in the corresponding quest to do it.

You fight a certain number of Unknowns/Troubles to get to it.

(10 Unknowns and 10 Troubles)

Hard Mode:

Can be accessed once Moderate Boss is beaten, and player is at a level of at least 35.

*Syntax Lvl 36 - lvl 47: The most common of all bugs. Will be all over the map.

Asks typical Syntax questions...

*Semantic lvl 40 - 52: Second most common of the bugs. Will be in majority of the map.

Asks typical Semantic questions...

*Unknown lvl 42 - 54: These bugs are uncommon, will be in the minority of the map.

Harder, unknown problems that need to be diagnosed...

*Trouble lvl 48 - 60: These bugs are very uncommon, will be in the minority of the map.

A lot harder, complex computing/programming problems...

*BOSS 1: A single, RARE bug, that needs to be fixed to fix the BIGGER bug!

Player needs to be in the corresponding quest to do it.

You fight a certain number of Syntax/Semantics to get to it.

(20 syntax and 20 semantics)

*BOSS 2: A single, RARE bug, that needs to be fixed to fix the BIGGER bug!

Player needs to be in the corresponding quest to do it.

You fight a certain number of Semantics/Unknowns to get to it.

(25 semantics and 10 unknowns)

*BOSS 3: A single, RARE bug, that is the biggest bug in the known area!

Player needs to be in the corresponding quest to do it.

You fight a certain number of Unknowns/Troubles to get to it.

(25 unknowns and 20 troubles)

Here is how the 'Bug Fixing' will take place. Similar to other MMORPG games, certain bugs will be aggro(aggressive, thus follows and runs after the player) or they will be passive(calmly sitting and walking around). Once the player comes near the bugs, the 'battle system' will initiate, in which a question will be asked to the player and the player will have to answer the question before the bug deals enough damage to him/her to bring their HP to 0. If answered correctly, the bug will be destroyed and the player will gain a certain amount of Gold. As mentioned before, there will also be a chance of the bug dropping items, where the percentage of the chance will depend on the bug. This will be gone over more in detail below.

If the player gives a wrong answer, the bug will deal triple the amount of damage it deals over time. Once the player has no more HP left, he will be respawned to the 'common area' of the map, with half of their original HP.

The shops that sell accessories as mentioned before, also sell Health Potions, which when bought and taken, recover a certain amount of HP. The players HP increases with his/her level, and with certain attributes.

Continuing on about the drop rates, players will be able to gain random drops from the Bugs they fix. These drops can be anything from random amount of gold to clothes and accessories for the player's avatar. Of course, only the harder Bugs will contain the more expensive items, and the easier Bugs will have a lower chance of a successful drop.

Something like;

Lvl 1 - 8 Bugs = Drop rate of 2%

Lvl - 15 Bugs = Drop rate of 5%

Lvl 16 - 22 Bugs = Drop rate of 15%

Lvl 23 - 30 Bugs = Drop rate of 30%

Lvl 31 - 40 Bugs = Drop rate of 50%

Lvl 41 - 50 Bugs = Drop rate of 75%

Lvl 51 and Above Bugs = Drop rate of 100%

The player's core HP amount, (discluding the bonuses they might gain from other affects) will

be the level of the player multiplied by 10. In comparison to this, how much damage in regard of HP each Bug will be doing is spread like this:

Easy Mode:

*Syntax Lvl 1 - lvl 9: Bug's level * 1

*Semantic lvl 4 - 11: Bug's level * 1.5

*Unknown lvl 8 - 12: Bug's level * 2

Moderate Mode:

*Syntax Lvl 12 - lvl 34: Bug's level * 1

*Semantic lvl 18 - 38: Bug's level * 1.5

*Unknown lvl 32 - 40: Bug's level * 2

*Trouble lvl 30 - 42: Bug's level * 2.5

Hard Mode:

*Syntax Lvl 36 - lvl 47: Bug's level * 1

*Semantic lvl 40 - 52: Bug's level * 1.5

*Unknown lvl 42 - 54: Bug's level * 2.5

*Trouble lvl 48 - 60: Bug's level * 3

and here is time per attack;

Easy Mode:

*Syntax Lvl 1 - lvl 9: 2 seconds

*Semantic lvl 4 - 11: 2 seconds

*Unknown lvl 8 - 12: 3 seconds, 2 questions

Moderate Mode:

*Syntax Lvl 12 - lvl 34: 3 seconds, 2 questions

*Semantic lvl 18 - 38: 3, 2 questions

*Unknown lvl 32 - 40: 2.5 seconds, 2 questions

*Trouble lvl 30 - 42: 3 seconds, 3 questions

Hard Mode:

*Syntax Lvl 36 - lvl 47: 3 seconds, 3 questions

*Semantic lvl 40 - 52: 2 seconds, 3 questions

*Unknown lvl 42 - 54: 1.5 seconds, 4 questions

*Trouble lvl 48 - 60: 2 seconds, 5 questions

And last but not least, how much gold they are prone to drop;

Easy Mode:

*Syntax Lvl 1 - lvl 9: 5-15 golds

*Semantic lvl 4 - 11: 10-20 golds

*Unknown lvl 8 - 12: 20-30 golds

Moderate Mode:

*Syntax Lvl 12 - lvl 34: 15-40 golds

*Semantic lvl 18 - 38: 22-50 golds

*Unknown lvl 32 - 40: 36-55 golds

*Trouble lvl 30 - 42: 40-70 golds

Hard Mode:

*Syntax Lvl 36 - lvl 47: 50-75 golds

*Semantic lvl 40 - 52: 60-90 golds

*Unknown lvl 42 - 54: 80-120 golds

*Trouble lvl 48 - 60: 100-200 golds

This will create a competitive atmosphere within the game, at the same time will not be too hard for the player to progress upon and improve himself/herself.

Each boss will also have a randomly generated drop. Since the bosses have to be beaten with a party, it will drop multiple items, that is given to each participating player randomly.

The basics of Quests are explained below;

There will be different quests on different areas. Players will be urged to do quests because they will result in a reward to be gained when beaten. The awards will be usually either a tidy sum of

gold, or a basic item the Avatar can wear.

Here are certain quests that will be available;

EASY:

Rookie: Talk to the NPC that gives this quest. You need to kill 20 syntax bugs and report back to him. Reward is 1 red shirt and 1 blue jeans.

Rookie Part 2: Need to have completed the first part of Rookie. Talk to the same NPC to gain this quest. You need to kill 20 semantic bugs and report back to him. Reward is 1 blue shirt and 1 red jeans.

Not a Noob: Talk to the specific NPC that gives you this quest. No prereqs needed. Need to collect 50 shards of correction. They can be obtained from syntax/semantics. The drop rate is 75%. Report back to him when you have all 50 shards of correction. Reward is Reading Glasses and 500 Golds.

MODERATE:

A New Challenge: Talk to the NPC that gives this quest. You need to collect 100 syntax shards. Drop rate is 75%. Return to the NPC when done. Reward is Haste Shirt.

A New Challenge2: Talk to the same NPC that gives the previous quest. You need to collect 100 semantic shards. Drop rate is 75%. Return to the NPC when done. Reward is Durable Jeans and 1000 Golds.

Exploring the Unknown: Talk to the NPC that gives this quest. You need to defeat 50 Unknowns. Return to the NPC when done. Reward is Frameless Glasses.

A new Trouble: Talk to the NPC that gives this quest. You need to defeat 50 Troubles. Return to the NPC when done. Reward is White Jacket.

HARD:

No More Joke: Talk to the NPC that gives this quest. You need to collect 100 Unknown shards. Drop rate is 50%. Return to the NPC when done. Reward is 2500 Golds.

No More Joke2: Talk to the same NPC that gives the previous quest. You need to collect 100 Trouble shards. Drop rate is 50%. Return to the NPC when done. Reward is 4000 Golds.

Of course, there is also the MAIN quest per difficulty area, to fight the boss to be able to go to the next area.

To fight a boss, towards the end of each 'bug hunting' area of each difficulty map, there will be the BOSS spawn point. Whoever wishes to fight the BOSS first would need to talk to the NPC that will be placed next to the BOSS' spawn point. The player can talk to the NPC and accept the quest to fight the BOSS. Once the quest is accepted, the player can walk towards the spawn point and will be prompted to create a party. Once a party is created by 'one' character, the others like him can 'join' the part. Once the party creator is content with the size of the party, he can initiate the battle.

The BOSS will give the party members a set amount of questions to answer, and the players will need to finish these questions before all the team members die.

For example, the Easy BOSS will need 50 questions to be solved to be beaten, and will do 40 damage to all players every 5 seconds. If a question is answered correctly, that party member will recover 10 HP. All the 'increase bug attack time' attributes of all the players will be added to the BOSS, divided by 2.

For example if a player has a total of +2 seconds and another player has +1 second, the BOSS now will attack every 6.5 seconds.

If a player dies, they will be spawned to a spawn point a little further away from the BOSS, with a spawn time of 45 seconds. The player can run towards and the BOSS and continue trying to beat the BOSS. Remember that players respawn with half full HP. If all the players are dead, at a given time, the BOSS challenge will be voided, and the players will spawn back outside of the BOSS area.

Here is the amount of BOSS questions, attack times, and damage;

Easy:

50 questions, every 5 seconds, 40 damage

Moderate:

125 questions, every 4 seconds, 100 damage

Hard:

200 questions, every 3 seconds, 220 damage

There are of course skills tied in with the Bug Hunting portion of the game, but unlike the board game skills/attributes, the Bug fixing will have its own set of skills, which are NOT unlocked by gaining a certain skill.

As explained before, within the board game, the player must gain a certain level, thus in return unlock a skill, then must purchase it. For the Bugging, they do not have to be unlocked but solely be purchased from the shops. The skill shop can be accessed in the same area as the Bug hunting area.

Here is a list of skills for the Bugging;

*A Shield skill will be implemented, where the player can bypass the question. Thus player can escape from the bug without losing HP.

*A Destroyer will be implemented, where the Bug will be automatically solved.

*An Eliminator will be implemented, where one of the choices of the answers will be eliminated.

*An Increaser will be implemented, where the possibility of a drop will be increased.

*Time-Warp will be implemented, where the attack time of the bugs will be increased.

This system will make it so that there will be a balance of necessity to participate in both portions of the game. Remember, Hunting does NOT give experience points, people will still need to play the board game and gain levels. This way, both games can live in harmony.

<u>The Chat System:</u>

(Chatting with others and party members)

(Implementers: Jordan Mangini)

The In-Game chatting system will be quite simple.

when the player first logs in, as explained earlier, the avatar will appear in the computer lab(Global Map) ready to enter one of three terminals. The chatting in this Global room will be only to those in this room, which could be any player outside the computers and could be of any rank and level.

The computer lab is like global chat, and when players the a computer, only those who are also inside the computer can speak with each other. All of the three computers will have this capability, allowing chat everywhere with the people the player needs or wants to speak.

Regarding Party Chat system, the concept of players only being able to talk with the others who are in the same 'world' aka computer as them simplifies the problem. A player can ONLY talk to another player in the same computer. A party member MUST be in the same computer as the other party member. Thus, there will not be any necessities for bypassing the chatting hierarchy explained above. There will only be needed to have a filter that will filter out everyone but the party member to chat with.

And here are the implementations of the level, item, attribute, and growth point systems;

EXPERIENCE CURVE

Level	Exp. Needed	Min. Correct	Max. Exp
1	5	5	1
2	6.25		
3	7.81		
4	9.77		
5	12.21		
6	15.26		
7	19.07		2
8	23.84		
9	29.8		
10	37.25	13	3
11	46.57	15	3
12	58.21		
13	72.76		

14	90.95		
15	113.69		
16	142.11		
17	177.64		
18	222.04		
19	277.56		
20	346.94		
21	433.68		
22	542.1		
23	677.63	40	17
24	847.03		
25	1058.79		
26	1323.49		
27	1654.36		
28	2067.95		
29	2584.94		
30	3231.17		
31	4038.97		

32	5048.71		
33	6310.89		
34	7888.61		
35	9860.76	150	67
36	12325.95		75
37	15407.44		
38	19259.3		
39	24074.12		
40	30092.66		
41	37615.82		
42	47019.77		
43	58774.72		
44	73468.4		
45	91835.5	250	367
46	114794.37		
47	143492.96		
48	179366.2		
49	224207.75		

50	280259.69	500	560

CLOTHINGS

Name	Price	HP	Speed	Bug Atk Time
SHIRTS				
Black	800			
Red	800			
White	800			
Blue	800			
Green	800			
Pink	800			
Purple	800			
Brown	800			
Haste	2000		0.05	
Durable	2000	2.00%		
Time	2000			.5 sec

210	6000	10.00%	0.2	
213	8000			2 secs
413	10000	15.00%		1 secs
1337	20000	25.00%	0.3	4 secs
JEANS				
Black	800			
Red	800			
White	800			
Blue	800			
Green	800			
Pink	800			
Purple	800			
Brown	800			
Haste	2000		0.1	
Durable	2000	2.00%		
Time	2000			.5 sec
leather	5000	10.00%	-0.15	
shorts	5000	-12.00%	0.2	

GLASSES				
Reading	1500			.5 sec
Frameless	2000		0.1	
Sun	3000	2.00%		
Aviator	4000	3.00%	0.15	
party	4000		0.2	
Big	4000			1 secs
Future	6000	10.00%	0.3	2 secs
BELTS				
Sturdy	3000	3.00%		
Light	4000		0.2	
New	4000			1 secs
leather	4000	10.00%		-2 secs
Magical	5000	8.00%	0.1	1 secs
SHOES				
White	4000	3.00%		1.5 secs
Black	4000		0.2	1 secs
Dragon	6000	15.00%	0.25	

Magical	6000	20.00%		1.5 secs
210	8000	20.00%	0.15	2 secs
213	8000		0.5	
413	10000	25.00%	0.2	2.5 secs
1337	20000	30.00%	0.25	4 secs
HATS				
Baseball	4000			1.5 secs
Gentleman	4000	10.00%		
Cowboy	4000		0.15	
Magician's	6000	15.00%	0.1	
210	8000	15.00%	0.1	1.5 secs
213	8000	30.00%		
413	10000	20.00%	0.15	2 secs
1337	20000	25.00%	0.4	2.5 secs

BUGS

BUGS:SYNTAX

Level	Drop Rate	HP Damage	Atk Time	Gold
1	2.00%	1	2 secs	5
2	2.00%	2		
3	2.00%	3		
4	2.00%	4		
5	2.00%	5		
6	2.00%	6		
7	2.00%	7		
8	2.00%	8		
9	5.00%	9		15
12	5.00%	12	3 secs at 2	15
13	5.00%	13		
14	5.00%	14		
15	5.00%	15		
16	15.00%	16		
17	15.00%	17		
18	15.00%	18		

19	15.00%	19		
20	15.00%	20		
21	15.00%	21		
22	15.00%	22		
23	30.00%	23		
24	30.00%	24		
25	30.00%	25		
26	30.00%	26		
27	30.00%	27		
28	30.00%	28		
29	30.00%	29		
30	30.00%	30		
31	50.00%	31		
32	50.00%	32		
33	50.00%	33		
34	50.00%	34		40
36	50.00%	36	3 secs at 3	50
37	50.00%	37		

38	50.00%	38		
39	50.00%	39		
40	50.00%	40		
41	75.00%	41		
42	75.00%	42		
43	75.00%	43		
44	75.00%	44		
45	75.00%	45		
46	75.00%	46		
47	75.00%	47		75

BUGS:SEMANTIC

Level	Drop Rate	HP Damage	Atk Time	Gold
4	2.00%	6	2 secs	10
5	2.00%	7.5		
6	2.00%	9		
7	2.00%	10.5		
8	2.00%	12		

9	5.00%	13.5		
10	5.00%	15		
11	5.00%	16.5		20
18	15.00%	27	3 secs at 2	22
19	15.00%	28.5		
20	15.00%	30		
21	15.00%	31.5		
22	15.00%	33		
23	30.00%	34.5		
24	30.00%	36		
25	30.00%	37.5		
26	30.00%	39		
27	30.00%	40.5		
28	30.00%	42		
29	30.00%	43.5		
30	30.00%	45		
31	50.00%	46.5		
32	50.00%	48		

33	50.00%	49.5		
34	50.00%	51		
35	50.00%	52.5		
36	50.00%	54		
37	50.00%	55.5		
38	50.00%	57		50
40	50.00%	60	2 secs at 3	60
41	75.00%	61.5		
42	75.00%	63		
43	75.00%	64.5		
44	75.00%	66		
45	75.00%	67.5		
46	75.00%	69		
47	75.00%	70.5		
48	75.00%	72		
49	75.00%	73.5		
50	75.00%	75		
51	100.00%	76.5		

52	100.00%	78		90

BUGS:UNKNOWN

Level	Drop Rate	HP Damage	Atk Time	Gold
8	2.00%	16	3 secs at 2	20
9	5.00%	18		
10	5.00%	20		
11	5.00%	22		
12	5.00%	24		30
32	50.00%	64	2.5 sec at 2	36
33	50.00%	66		
34	50.00%	68		
35	50.00%	70		
36	50.00%	72		
37	50.00%	74		
38	50.00%	76		
39	50.00%	78		
40	50.00%	80		55

42	75.00%	105	1.5 secs at 4	80
43	75.00%	107.5		
44	75.00%	110		
45	75.00%	112.5		
46	75.00%	115		
47	75.00%	117.5		
48	75.00%	120		
49	75.00%	122.5		
50	75.00%	125		
51	100.00%	127.5		
52	100.00%	130		
53	100.00%	132.5		
54	100.00%	135		120

BUGS:TROUBLE

Level	Drop Rate	HP Damage	Atk Time	Gold
30	30.00%	75	3 sec at 3	40
31	50.00%	77.5		

32	50.00%	80		
33	50.00%	82.5		
34	50.00%	85		
35	50.00%	87.5		
36	50.00%	90		
37	50.00%	92.5		
38	50.00%	95		
39	50.00%	97.5		
40	50.00%	100		
41	75.00%	102.5		
42	75.00%	105		70
48	75.00%	135	2 secs at 5	100
49	75.00%	147		
50	75.00%	150		
51	100.00%	153		
52	100.00%	156		
53	100.00%	159		
54	100.00%	162		

55	100.00%	165	
56	100.00%	168	
57	100.00%	171	
58	100.00%	174	
59	100.00%	177	
60	100.00%	180	200

<u>What is Not Implemented?</u>

Majority of the Game Concept has not yet been implemented.

The Board Game portion of the game is entirely missing, and will later be implemented to keep the foundation of the game and the growth system together.

The Shop has not been fully implemented, as well as all the clothing items, excluding the HP potions.

All the skills are also missing at the moment, but will be later implemented.

Nevertheless, the basis of the bugs are functioning, as well as a few working, animated avatars. The chatting system also functional.

Even though not all the types and levels of Bugs are implemented, Bugs do exists, and they do function accordingly, dropping 'loots' and dealing damage.

As of right now, only "Easy" mode has been implemented.

<u>4.b Art Support Team</u>

4.b (Art Support Team) Scope, Tasks and Milestones

The main landing scene of the game is represented by a computer table with three computers which symbolize various difficulty levels. For our game level implementations the team has chosen the “insides” of the computers (e.g. motherboard with all the computer parts). The easiest level is represented with an oldest model of a computer which runs MS-DOS, the medium level is represented by the current model of IMAC computer and the hardest (e.g. expert) level is represented by a futuristic looking computer which runs LINUX. Each of these computers has a NPC object by approaching which a player will be transported into an appropriate scene.

The characters start as very general looking characters who wear jeans and shirts. By completing the levels they will be accumulating points which they can use to buy various clothes for themselves. The character customization presents a substantial amount of work. Not only the team has to create characters and animate them, but these characters has to be modified to accommodate all the customization options. The AS team can't produce all the variations of each character with all possible customizations; and, in essence, this shouldn't be even done because this should be accomplished from the client code programatically. The goal of the art team is to produce characters with well defined bone structure and a set of different textures for different parts of the models (e.g. pants, shirts) along with a number of customization items (e.g. boots, hats, glasses). The textures for different parts of the models can be substituted from the code and various items can be attached to the bones of the models. There are examples of the code on the Panda3D manual and forums.

As it is a very challenging task to implement the whole game in just one semester, the game was divided into multiple milestones. Each milestone is a step forward in terms of functionality and the implemented content. The first semester of development included two milestones and the elements of these milestones (e.g. only the elements specific to AST tasks) are listed below:

MILESTONE 1

1. Login Scene
2. Main Landing Scene of the game
3. General characters (male and female)

MILESTONE 2

1. Easy Level Motherboard Scene
2. Easy Level Dungeon Scene

To implement any of the above points, a lot of work has to be done, but in general it comes down to the following steps.

For game levels:

- Brainstorming the implementation ideas
- Conceptualization through sketching
- Search of suitable 3D models
- Work on individual models which are part of the game levels
- Putting them all together into one large scene
- Conversion into .egg format

For game characters:

- Search of suitable 3D characters.
- Modification of the found models (e.g. poly count reducing, material conversion, deletion of extraneous elements, etc)
- Animation cycles creation (e.g. walk, run, attack, death, etc)
- Conversion into .egg

Along a way we've ran into a number of problems and here are a few words about them and the above steps:

1. Panda3D only understands a limited number of shaders (materials). At the time of this writing Panda3D engine understood 2 types of shaders: Blinn and Phong. That's why all models were either modeled with these materials or all of the existing materials were converted into the above materials.

2. When modeling each separate items for the game levels, it is a good idea to keep the poly count as low as possible. In general, for the current generation of computers it is advised to have poly count for game characters under 4K poly's. For game levels the restrictions are less known; we used the example of the Roaming Ralph level which weights about 16Mb in .egg format. So, if after conversion into .egg our game level weighted less than 16Mb then it was a good indication that the game engine won't be bogged down.

3. When modeling each separate items for the game level it is a good idea to group the item's parts into separate groups. This is later simplifies the insertion of proper xml snippets into .egg file necessary for collision detection. Insertion of these xml properties into .egg can be done automatically and the instructions on how to do that can be found on Panda3D forums/manual. Our team used the command line tool for conversion and there is no argument for this tool which would do this automatically. In any case this is a very simple process. Here is the description: lets say we have a model of a hard drive in a separate Maya project. It is finished and we are ready to import it into the main project file for the game level, before doing that we highlight all of the mesh objects for the model and group them; we give a group a name and only after this we import this project into the main project file. It will also be imported as a separate group. When we will be done with the modeling of the game level project, we'll run the conversion tool and will get a .egg file. Then .egg file can be opened in any text editor. Once it is opened, we search for the name of the hard drive group, once we found the XML element for that group, right after the first xml tag for it we insert just one line of code. For example:

```
<Group> hardDrive {

        <Collide> { Polyset keep descend }            // this line makes collision detection possible

        etc...
```

After that a character won't be able to go through a hard drive object in the scene.

4. It seems that Panda3D engine doesn't like models which are modeled with other than simple polygon objects. We encountered problems with conversion of animated NURBS and SubDiv's models. So, it is recommended to use polygons as the main modeling object types.

5. In order to use a command line conversion tool, several environment variables has to be created on the execution machine. Please, refer to Panda3D manual for exact instructions, but the following is just a brief description of what was required to be done on our machine.

The following environment variables are very specific to our environment (Windows, Maya, Panda1.6.2). You can easily understand where these environment variables should point to from their values.

MAYA_LOCATION=C:\Program Files\Autodesk\Maya2009

PYTHONHOME=C:\Panda3D-1.6.2\python

PYTHONPATH=C:\Panda3D-1.6.2\python\bin;C:\Program Files\Autodesk\Maya2009\Python\lib\site-packages

PATH=C:\Program Files\Autodesk\Maya2009\bin;..... // precede PATH variable with a path to MAYA bin directory

What is needed to convert a static model into .egg:

1. cd into the directory of Panda3D bin folder (it contains all the conversion utilities)

2. maya2egg2009.exe “path/to/original/maya/project/file” -o “path/to/output/egg/file” -ps rel

e.g. for the description of all options, run maya2egg2009.exe -help

To convert an animated model into .egg:

The approach used here relies on having separate files for 3D models and their animation cycles; therefore, in order to convert a model along with its animation, two commands need to be run.

1. cd to the directory of Panda3D bin folder

2. maya2egg2009.exe “path/to/maya/project/file” -o “path/to/panda3d/character.egg” -a model -cn modelName // converts the model itself without animations

3. maya2egg2009.exe “path/to/maya/project/file” -o “path/to/panda3d/character-{animationName}.egg” -a chan -cn modelName -sf startFrame -ef endFrame // create animation file for the model converted by the previous command.

NOTE Any of the existing model files (.egg) can be opened and the exact commands which were used to create them are listed at the very top of the files.

4.c – Client Team:

This section covers the scope and tasks for each module taken care of by the client team.

4.c.i -REGISTER

Before playing the "deBugger" a user must be able to create an account. With a personal account, a user will be log in to the game and have their personal statistics, properties, achievements and items stored. In order to enter the "deBugger" game, a username and password for an account will be checked against an online database before a user is granted access and logged in. In order to create an account with a username and password, one can simply click on the "Register" button available at the login screen.

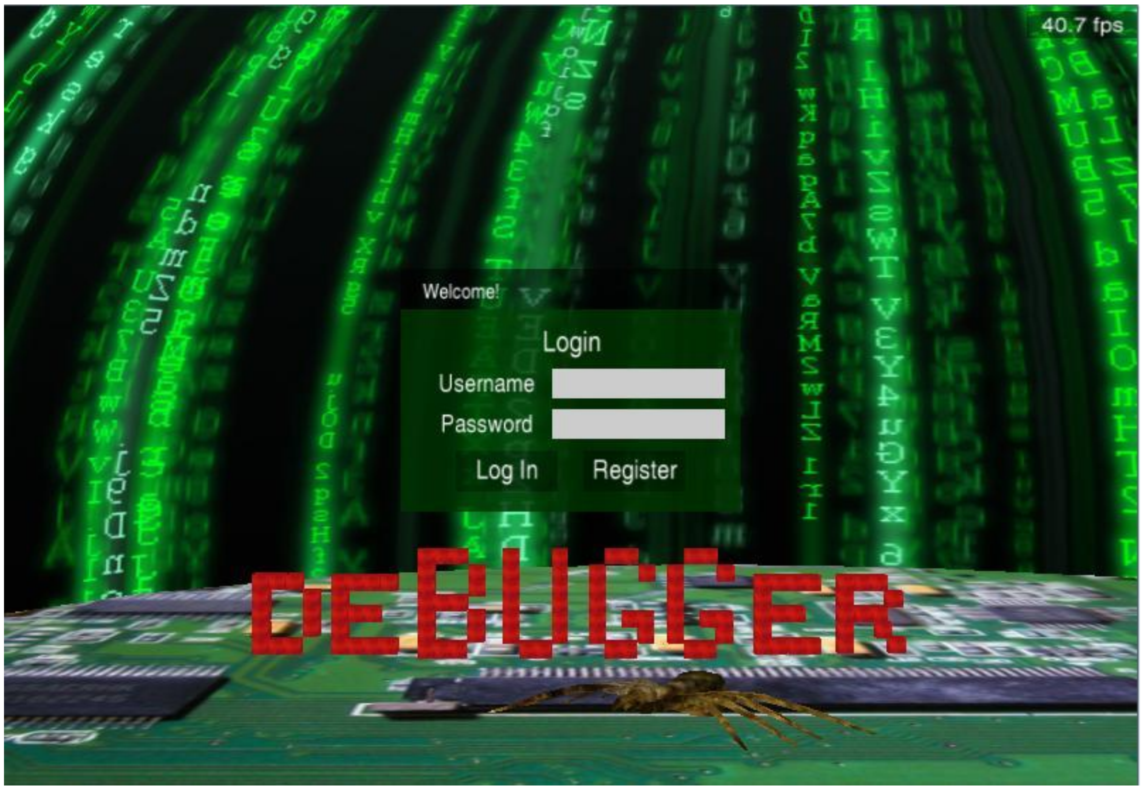
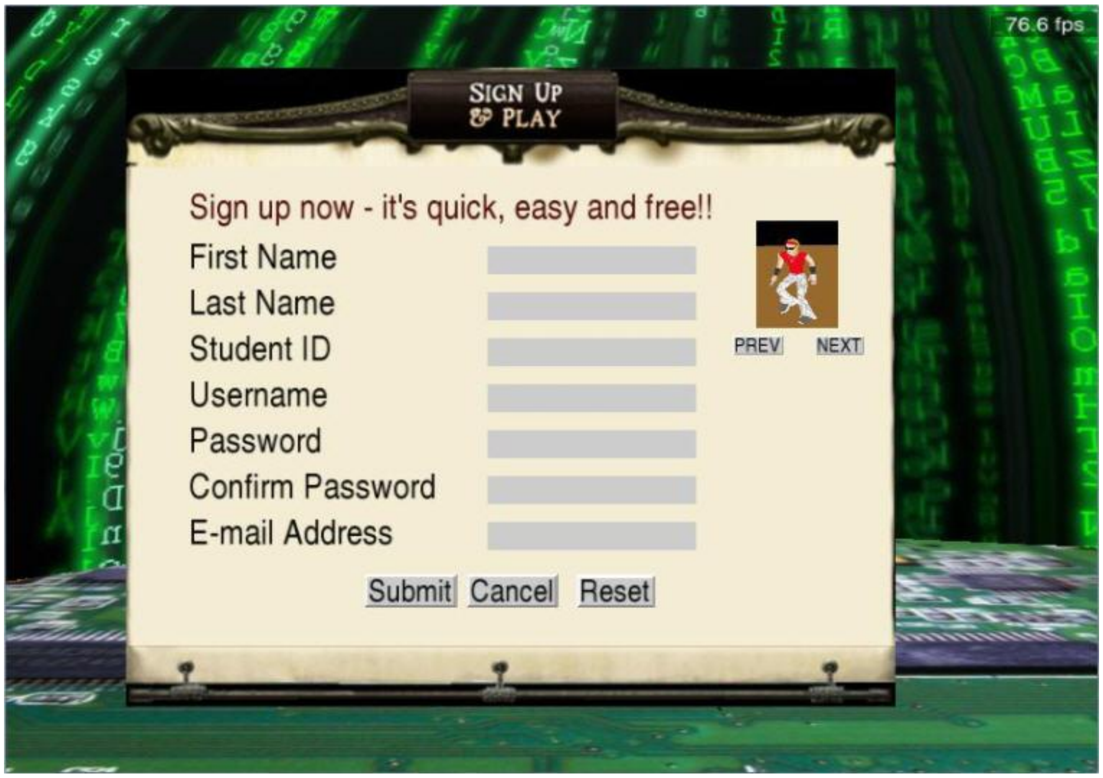


Figure 4.c.i.1 - The Login Screen has a button to direct users to the Registration Screen.

After clicking on the “Register” button from the login screen, the user will be prompted with the registration screen. The registration screen will be where a user will be able to create a new game account that they can use in order to log in and play the “deBugger” game. This screen includes field descriptors alongside text entry field. The user will be asked to enter unique information such their name, student ID and e-mail address. The user will also be asked to come up with a username and password that they can use in order to log in to their account and play the “deBugger” game. The registration screen will also feature a password confirmation field. This is normally implemented to make sure that the user hasn't accidentally mistyped their own password. It also prevents them the frustration from trying to guess what their mistyped password may have been.



Sign up now - it's quick, easy and free!!

First Name	
Last Name	
Student ID	
Username	
Password	
Confirm Password	
E-mail Address	

Figure 4.c.i.2 - Registration Screen

Also featured on the registration screen is an avatar selector. In the upper right hand portion of the registration menu, there is a character image along with buttons designated “prev” and “next”. These buttons allow a user to scroll through a list of images of in game playable characters. The user can leave the image on the character that they have found they would like to in the game as. This adds some individuality to a player's in game likeness and doesn't restrict the players to look exactly the same when first starting out. Depending on a player's progress, they may able to add more characteristics to the base character that they choose at the registration screen.

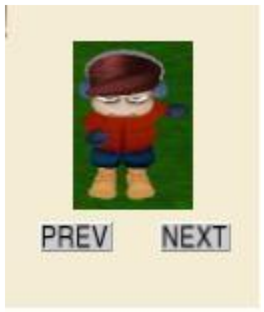


Figure 4.c.i.3 - Users can use the Prev and Next buttons to scroll through playable characters.

The bottom of this screen features buttons that offer functionality over control of the screen. The first button, labeled “Submit”, will take all the information gathered and try to send it to the server to create an account. If a user has entered invalid information then they will be prompted to correct the error(s). The second button, labeled “Cancel”, allows the user to exit this screen and go back to the log in screen. The third button, labeled “Reset”, resets all the text entry fields and makes them blank. This makes it convenient for a user to enter in new information without having to manually clear every field themselves.

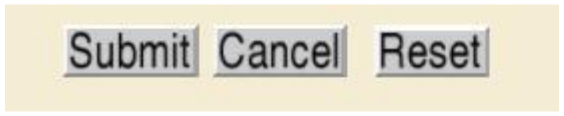


Figure 4.c.i.4 - Action Buttons for user to control the registration screen.

After successfully entering and submitting their data, the user will then be able to log in to the “deBugger” game.

4.c.ii – Login Process

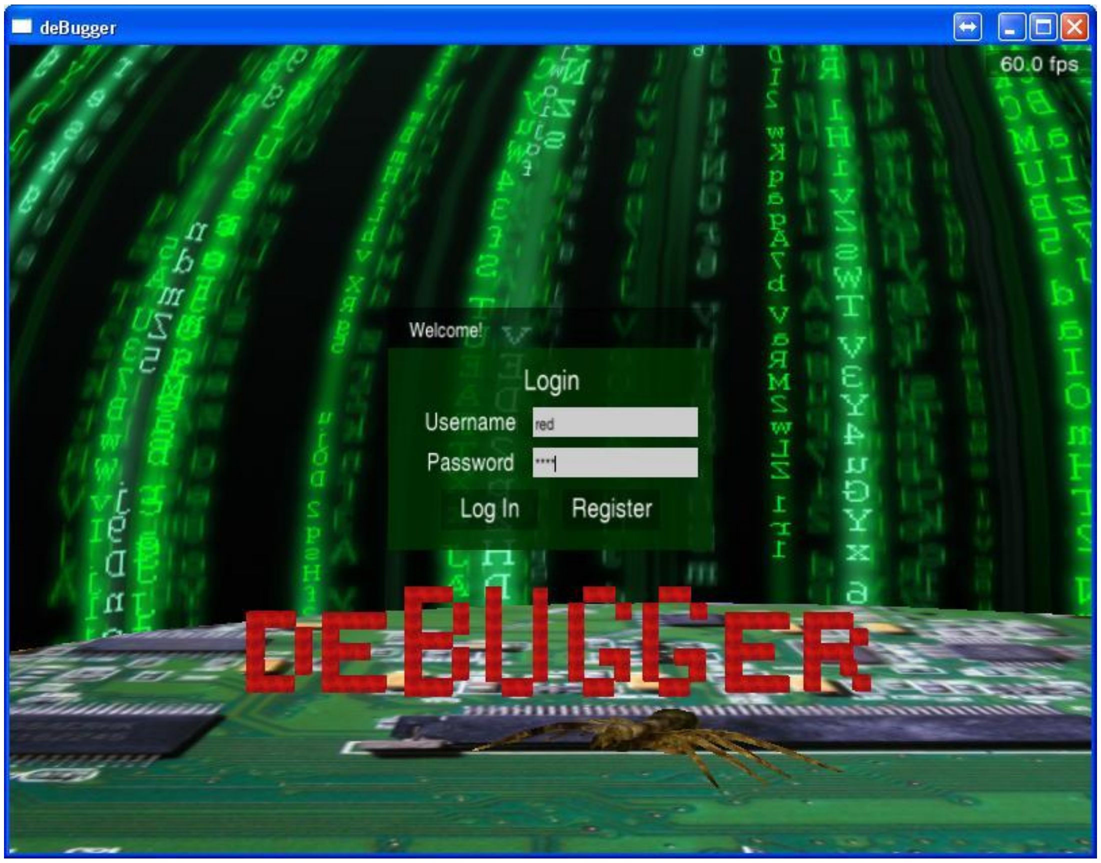


Figure 4.c.ii.1 Login Screen

Login screen is the first screen that user will see when the user open the debugger game. The screen consists of a background screen and a login box. Inside the login box, there are username and password input box. The username input box is just a regular text input field where it takes the user input and displays it there. The password input is a little bit different in a way that it takes the user input and displays the encryption version of the text input. Below these two input fields there are two input buttons: log in button and register button. The register button will take the user to register screen and the login screen will authenticate the inputs that user entered. When user click the log in button, the system will first try to do a quick validation on the inputs that user entered. If any of the inputs invalid, the error notification will pop up telling the user the invalid input. Once the quick validation succeeded, the inputs together with the protocol constant will be sent to the server. The server will then authenticate the inputs against the database. The server will send a respond back to the client based on the authentication process.

Once the client gets back a respond from the server, the client will either display an error notification when the authentication fails or go to the world when the authentication succeeds.

4.c.iii – Chat Module

Chat is a required feature in most online games which adds a social networking side to the game. Games provided on social networking sites such as Facebook also provide a chat feature. Debugger is a student community game wherein students can make friends, carry on discussions, ask for help and create a network within the gaming community due to which the chat feature was developed.

Debugger consists of 4 types of chat i.e. global chat, party chat, whisper chat and public chat. However due to time constraints we have only implemented only public and whisper chat whereas global and party chat are kept as future work. While making use of whisper chat, a game player needs to make use of the following format - '/nametowhisperto' where nametowhisperto indicates the name of the game player to whom the message needs to be communicated. At any point of time a game player can switch into any chat mode by pressing its equivalent button either by making use of mouse or by using keyboard. The Chat module has the following four buttons with text 'G', 'P', 'W' and 'A' to indicate global chat, party chat, whisper chat and public chat. To distinguish whether the game player is in global, party, public or whisper chat mode the chat module makes use of ***'!'***, ***'%'***, *''* and *'/'* operators. Also the text of each chat type is attributed with a different color which is also acts as an indicator to distinguish between the chat types.

The Chat feature in Debugger has a show/hide function using which a game player can hide the chat box when he is not chatting. The Chat box also has a drag and drop functionality using which the chat box can be dragged and placed anywhere on the screen. In addition to mouse, the chat box is also accessible via keyboard controls. A game player can toggle through the various chat modes by making use of 'Tab' key or 'Shift+Tab' key. Also by making use of the 'Enter' key a game player can switch between chat modes or into play mode. Lastly, during the course of the game all the chat messages are stored in a chat log which is deleted once the player exits from the game.

4.c.iv – Bubbles

Chat Bubble

One of the key reasons why MMO games are so successful is the reliability of the chat modules and functionalities. This game has two primary methods to display messages from users. The first of which is the chat box, which can keep track of several different types of chat from many different sources. The second method that the game displays messages between users is through the chat bubbles. The chat bubbles appear graphically in game, not requiring a user to divert their eyes towards the chat box and away from the main environment.

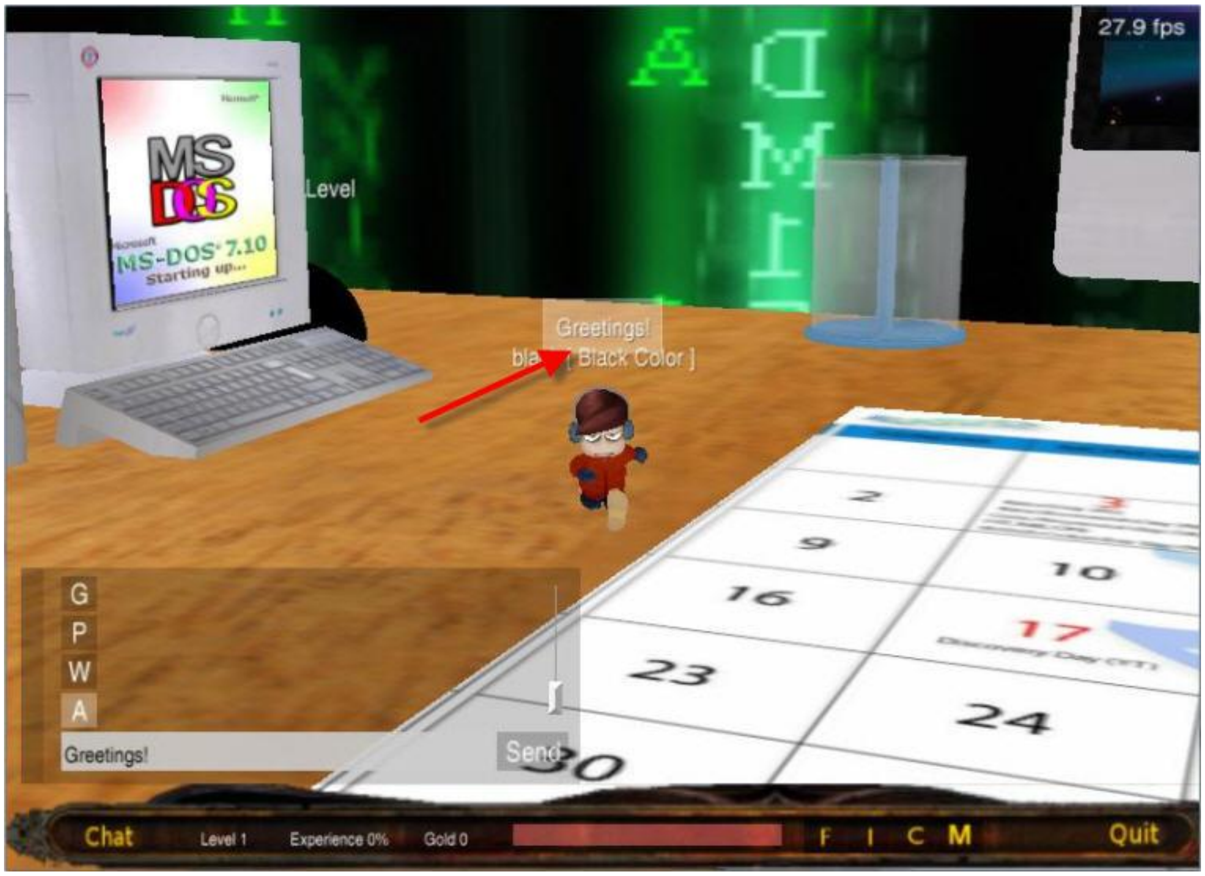


Figure 4.c.iv.1 - A player's message will appear over their head locally.

One of the characteristics about the chat bubble is that it displays over the source character's head. This allows players to easily find the source of the message without having to match names via the chat box. This prevents the pace of the game from being disrupted in the circumstance that there are critical events going on in the virtual world environment. Another characteristic about the chat bubble is that it displays messages from characters that are immediately visible to the player. This is different from the chat box, which has a wider range of access to messages from many characters.

In order to display messages in a chat bubble a user can simply input a message to send through the chat box. This will show the message in both the chat box and in the chat bubble. While the message is being created, the bubble will be filled with the text “Typing..”. This alerts other players that a character may be busy typing a message to notice something on screen or perform any immediate actions.

After a short amount of time or when a new message is being sent, the chat bubble and its contents will disappear from over the character.

Name Bubble

What makes online games unique from other types of games is the large amount of different players that can be playing simultaneously. For online games to help users identify each other quickly is through the use of visual tags which display a user's name. This is not unlike when a person goes to a convention where they are more than likely to meet a large group people they don't know. In a situation such as at a convention, several people are encouraged to wear name tags to help identify one another and make socializing an easier and sometimes   more fun experience.

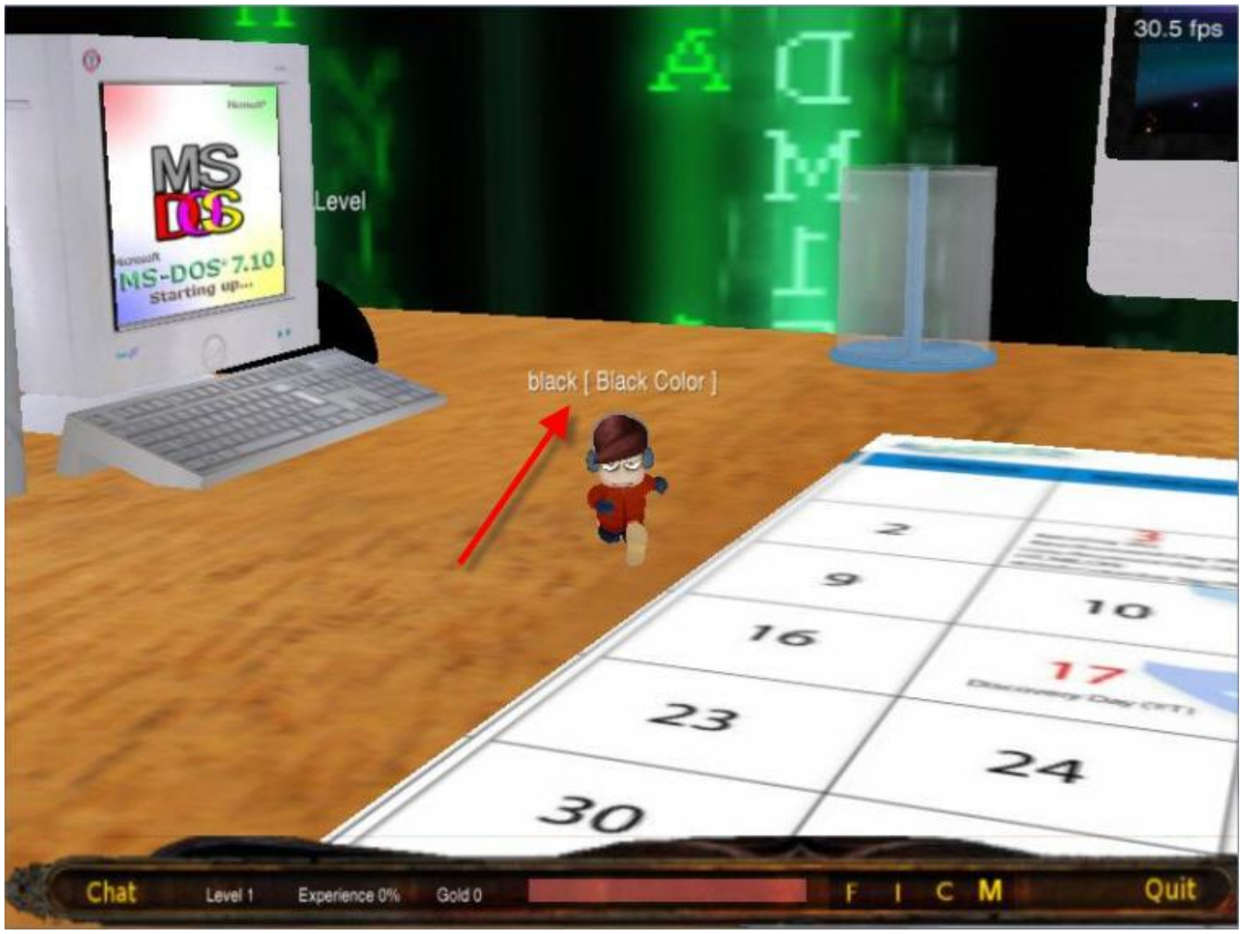


Figure 4.c.iv.2 - A character can be identified through the name displayed over their avatar.

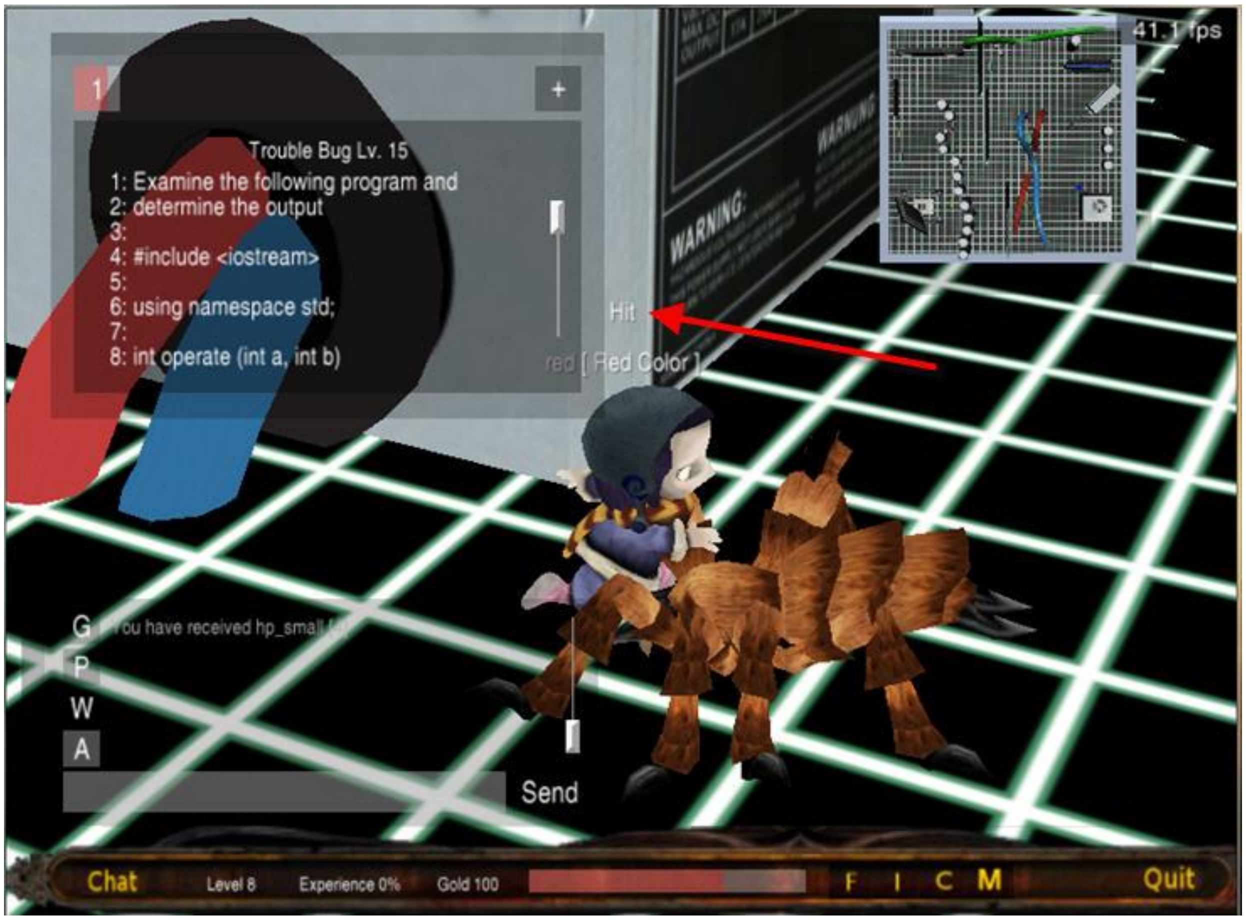
On a similar note to that of name tags at a convention, a character's name is displayed over the avatar's head in the game. The feature implemented to allow this functionality is called the name bubble. The name bubble is used for players to identify characters and other players while in the virtual world environment. What makes this unique from most other bubbles is that the name bubble is persistently displayed.

The name bubble is also helpful for distinguishing characters that may look similar or even identical. This may be the case in a situation such as being a low level character with limited customization ability. This may also be the case when some characters find that they have similar customized options.

Through the name bubble identification and socializing becomes easier. An easy and friendly social environment is fundamental for keeping a player interested in MMO games. With a good social environment, players may be willing to spend more time in “deBugger” learning and having fun simultaneously.

DamageBubble

In many MMORPGs, when a player is in combat the user will have to be alert to several different events happening at once. This becomes escalated in a game such as “deBugger” when the user has to think about how to solve an academic problem meanwhile keeping track of personal statistics. An important visual aide to help the player keep on top of everything that is going is the implementation of the damage bubble. With the damage bubble, text is displayed over the user's avatar to show how their character is being affected by enemy attacks. This helps during combat as it provides an extra cue and gives the player either a sense of urgency or alert the player to the status of a battle. With having a damage bubble in close proximity of a character's avatar, the player does not need to constantly check their health bar. This prevents the user's eyes from jumping across the screen and lessens eye strain.


```
Trouble Bug Lv. 15
1: Examine the following program and
2: determine the output
3:
4: #include <iostream>
5:
6: using namespace std;
7:
8: int operate (int a, int b)
```
red [Red Color]

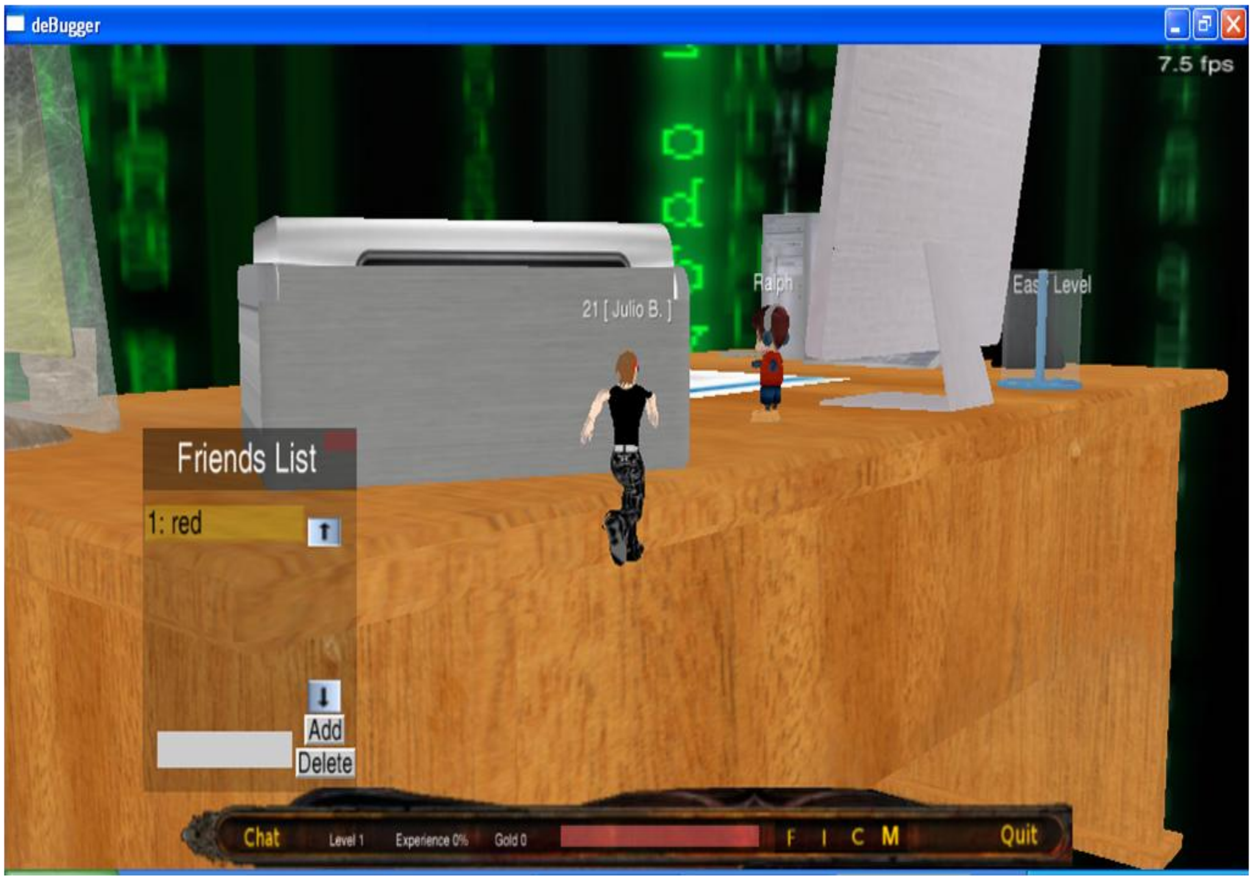
You have received hp_small

Figure 4.c.iv.2 - When a character takes damage, a "Hit" message will appear and alert the player.

The damage bubble is normally displayed over the character's name. After a small amount of the time, the damage bubble will disappear. This will allow a new damage to appear should the character experience damage once again.

4.c.v – Friends

This class has the responsibility o f adding and deleting a friend to the user friends list both locally and to the server. The rules to add a friend are simple; the user will just enter the username of the player that he/she wants to add, note that the user to be added needs to be logged-in in order for the request to be sent, upon depressing the add button, the user adding the friend will get a pop up screen asking if he or she is sure that a friend is to be added, upon depressing the yes button, a request will be sent first to the server, after the server has received the request, then a request will be sent to the user that is to be added, up receiving the response of the user to be added, then it will be determined whether or not that friend is to be appended to the friends list. The friends list can be brought up by depressing the “f” button on the users menu, or by just pressing the ctrl-f button sequence. Note that the friends list cannot have duplicate names; this is the reason why when a user enters the name into the list, the names are turned into lower case, making it easier to check if that particular name exists in the list already, if it does, then a pop up screen will come up and inform the user that the friend already exists in his/her list. Below you can see an example of how the friends list looks like, in this case, the user has already a added a friend (red) into his list



The code for Friends.py can be found inside the src.main.World.Gui.Friends package in the Friends.py module. Hopefully, you have already read how the rules in order to add a friend, here is an explanation of how everything works behind the scenes.

The first piece of code explained is how the code handles the strings entered by the user. We were told at the beginning of the class that duplicate name were not to be allowed, therefore, we were faced to make a decision, and at the end, we decided that the best thing to do was to turn all of the user input to lower case before the request is sent to the server and added to the list. Here some of the code

```
    def sendEntry(self):
      if(str.lstrip(str.rstrip(self.addFriendTextEntry.get())) == ""):                    #1
        self.errorDialog = OkDialog(dialogName="errorDialog", text="Friend Name required", command=self.okSel)
      elif(self.addFriendTextEntry.get().lower() in self.friendsListArray):               #2
        self.errorDialog = OkDialog(dialogName="errorDialog", text="Friend already in list", command=self.okSel)
      else:
        self.verifyAdd = YesNoDialog(dialogName="verifyAdd",text="Add "+self.addFriendTextEntry.get().lower()+" as a
friend?",command=self.verifySelection)                                                    #3


    def verifySelection(self, args):
      if(args):
        rContents = {"buddy" : self.addFriendTextEntry.get().lower()}
        self.main.main.cManager.sendRequest(Constants.CMSG_INVITE_BUDDY, rContents)  ← SERVER REQUEST
        self.addFriendTextEntry.set('')
        self.verifyAdd.cleanup()
      else:
        self.verifyAdd.cleanup()
```

As one can see, the method sendEntry(self) is in charge of capturing the user string as he/she enter a name to be added in the text box, then it checks to see if

1) The field is empty, if it is it gives an error message saying that a name is required.

2) If a name does exist in the textfield, the code checks to see if the name entered already exists in the FriendsListArray, if the name does exist an error message is given, if it does not exist it goes to the else stamen where the verifySelection function is called

3) verifySelection then checks to see if the user does indeed want to add a friend, the variable *args* acts as a boolean, if it's true, then the request is sent to the server, the server code will then be in charge of asking the user if it ok to be added added to the friends list.

The next piece of code to be examined is to see what happens after the server receives the request

```
def buddyAnswerAcc(self, accepted, buddy):
    if(accepted):
        self.errorDialog = OkDialog(dialogName = "errorDialog",
                              text = buddy + "has accepted your friend request",
                              command = self.okSel)
        if self.friendsListArray is not None:
            self.friendsListArray.append(buddy)
        else:
            self.friendsListArray = [buddy]
        self.updateList()
        self.addFriendTextEntry.set('')
    else:
        self.errorDialog = OkDialog(dialogName = "errorDialog",
                              text = buddy + "Does not like and does not want to add you",
                              command = self.okSel)

def buddyAnswer(self, accepted):
    self.verifyBuddyAccept.cleanup()
    if(accepted):
        rContents = {"accepted" : True, "requestorBuddy" : self.requesteeBuddy}
        self.main.main.cManager.sendRequest(Constants.CMSG_ACCEPT_BUDDY_INVITE, rContents)
        if self.friendsListArray is not None:
            self.friendsListArray.append(self.requesteeBuddy)
            else:
                self.friendsListArray = [self.requesteeBuddy]
            self.updateList()
            self.addFriendTextEntry.set('')
        else:
            rContents = {"accepted" : False, "requestorBuddy" : self.requesteeBuddy}
            self.main.main.cManager.sendRequest(Constants.CMSG_ACCEPT_BUDDY_INVITE, rContents)

    def getString(self, buddy):
        self.requesteeBuddy = buddy
        self.verifyBuddyAccept = YesNoDialog(dialogName="verifyBuddyAccept",text=buddy+" wants to add you
 as a friend? ",command=self.buddyAnswer)
```

All of this code will be explained in more detail in the understanding the understanding the code section, each function is clearly explained in order for the user to understand the how the process works.

4.c.vi – Inventory

In many MMO games, a player will be able to add, store and use items at their own discretion. Many of these items will normally benefit the player through aiding statuses, adding revenue or other general game statistics. When a user wants to access these items they normally do it through an inventory list. This list is a GUI menu that appears and disappears at the user's discretion, allowing them to hide it when they don't want the menu hampering their field of view or field of depth in the game or allowing them to view it when they want to use an item.

The inventory list in the "deBugger" has simple properties that are characteristic to many inventory menus. The inventory menu in "deBugger" has a title bar that can be dragged around the screen. This allows the player to place the inventory menu in a less obstructive position at any given time. This can be helpful if there are other activities going on in the virtual environment that the user would like to keep an eye on. In this situation, the player can simply left click and hold the mouse button on the title bar. While continuing to hold the left click button, they can drag the menu around the screen. When they have moved the menu to a position on the screen that they prefer, they can let go of the left click on the mouse. From here, the inventory menu will stay in this position on the screen until they decide to move it again.

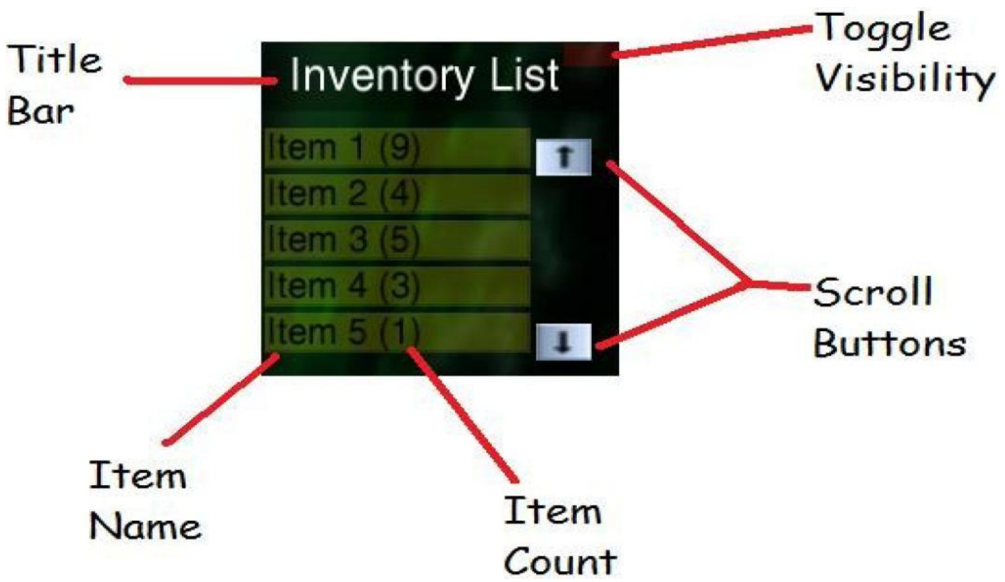


Figure 4.c.vi.1 - Inventory Menu Features

The inventory menu also has toggling functionality. The inventory menu can be toggled by either clicking the red box in the upper right hand corner (hiding only), pressing "I" on the main bar or by pressing a combination of “CTRL+I” once in the virtual environment. Having the menu toggle allows it to be hidden when the user doesn't need to view the list, giving the player more view access of the environment. When they need to view the menu, they can toggle it back on and the menu will be in the same position where it was toggled off.

When a user has more than the maximum number of items that can be displayed at one time (currently 5 items at the time of this writing), the user will be able to use the scroll the buttons. The scroll buttons are on the right edge of the inventory menu. The button with an arrow pointing up signifies scrolling up and the button with the arrow scrolling down signifies scrolling down. The scroll buttons will allow the user to view more items than those currently displayable on screen by going the list of items sequentially and removing older entries from the display. When the user has hit the end or beginning of a list, the corresponding scroll button will then stop scrolling the list.

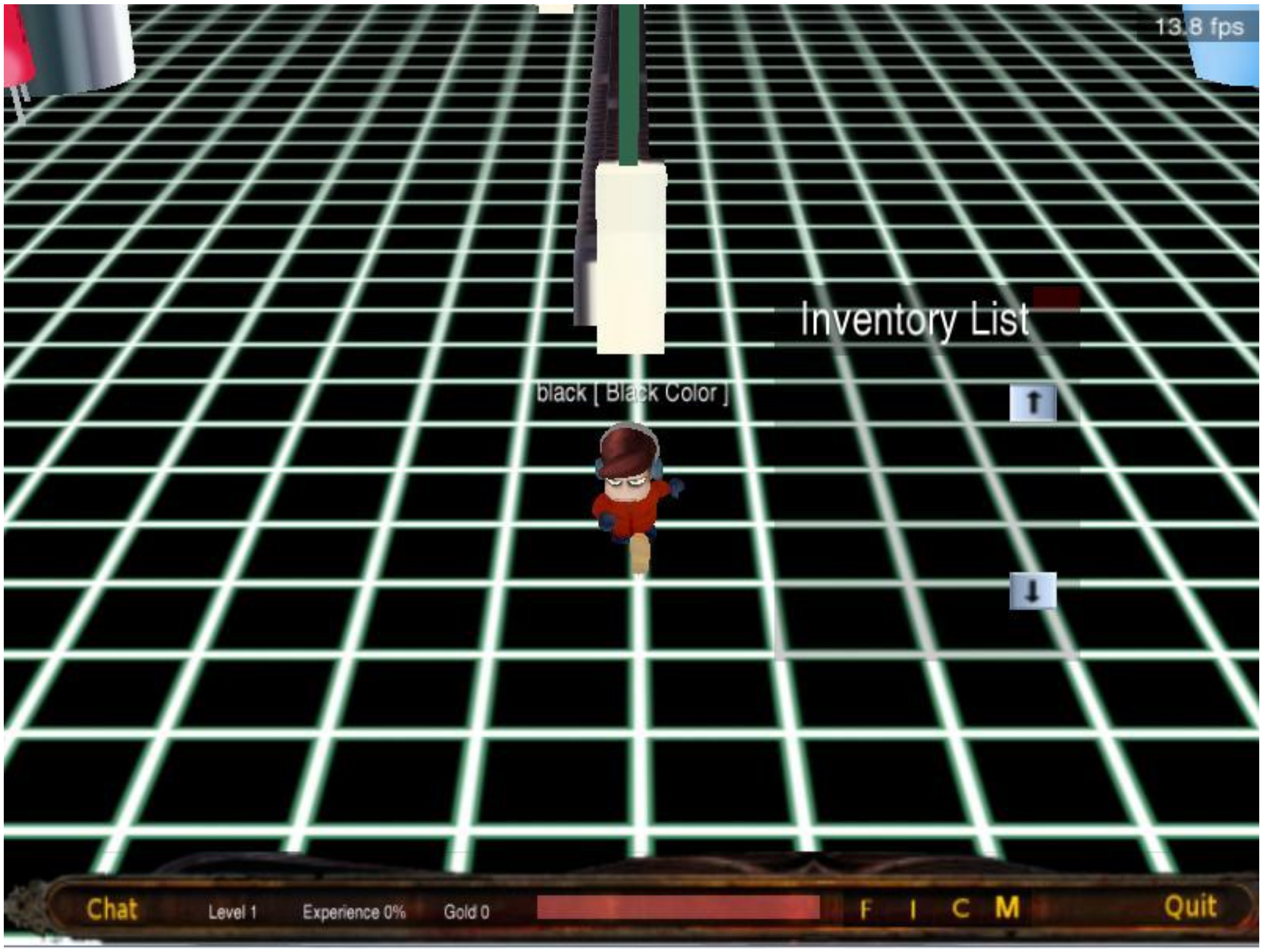


Figure 4.c.vi.2 - When a user has no items, they will see an empty inventory menu.

The items are displayed on the inventory menu as buttons. The buttons contain information for the user such as the item's name and the item's count value, which displays the quantity of the item which the user currently has. When the user wants to use an item, they can click on the item button. After clicking on the button, a prompt will display to the user for whether or not they want to use the item. After verification, the item will one less value in the item count field. If there are no more items, then the item will disappear from the inventory list. At the time of this writing, the only usable items are health restoration items.

Being able to easily access the items the player has collected allows the game to feel more mainstreamed. With quick access, the player doesn't have to return back to a neutral point or navigate several menus to view their items.

4.c.vii – Character Info

Character info panel displays information about the logged in player. A player can toggle the display of this panel by clicking ‘C’ button from the bottom menu or using ctrl-c / ctrl-c-up hot key. This panel contains:

- Username
- Student ID
- Level
- Health Point
- Gold
- Avatar image

Following methods are used to implement this panel

Method Name	**Parameters**	**Description**
init	self world	Create Character info frame
startdragmode	self param	Allow Draggable Window (Using Title Bar). Bind Main Frame (Parent Frame) To Mouse
stopdragmode	self param	Allow Draggable Window (Using Title Bar). Bind Main Frame (Parent Frame) To Mouse
getinfo	self	Retrieve Character info
setinfo	self	Update Character Info

togglevisibility	self	Toggles the visibility of character info panel
setupTextNode	self	Setup the components of character information Box Frame
unload	self	Unloads the character info

4.c.viii – Hot Keys

Debugger has used Panda3D built-in keyboard and mouse support. Following is the list of the Hot-keys implemented in it.

Keys/Mouse click	**Function**
Enter	Toggle between the chat window and play area
Enter-up	Toggle between the chat window and play area
Shift-tab	Switch between chat room in backward direction
Shift-tab-up	Switch between chat room in backward direction

Tab	Switch between chat room in forward direction
Tab-up	Switch between chat room in forward direction
W	Moves the character in up direction
w-up	Moves the character in up direction
A	Moves the character in left direction
a-up	Moves the character in left direction
s	Moves the character in down direction
s-up	Moves the character in down direction
d	Moves the character in right direction
d-up	Moves the character in right direction
arrow_up	Moves the character in up direction
arrow_left	Moves the character in left direction

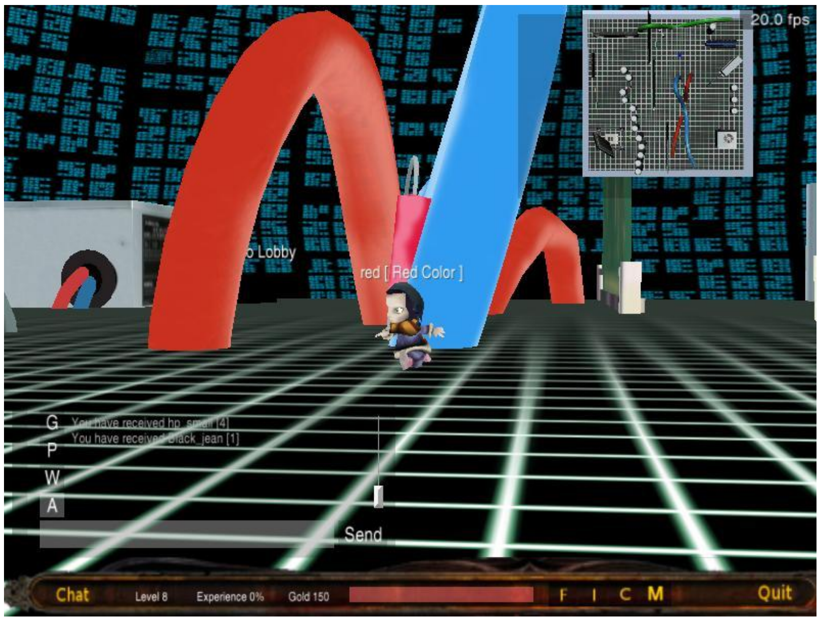
arrow_right	Moves the character in right direction
arrow_down	Moves the character in down direction
arrow_up-up	Moves the character in up direction
arrow_left-up	Moves the character in left direction
arrow_right-up	Moves the character in right direction
arrow_down-up	Moves the character in down direction
ctrl-f	Toggles the display of Friend List of the player
ctrl-f-up	Toggles the display of Friend List of the player
ctrl-I	Toggles the display of Inventory List of the player
ctrl-i-up	Toggles the display of Inventory List of the player
ctrl-c	Toggles the display of Character Info of the player
ctrl-c-up	Toggles the display of Character Info of the player

ctrl-m	Toggles the display of the mini map on the screen
ctrl-m-up	Toggles the display of the mini map on the screen
mouse1	Spots a mark on play area and character will run to reach that point
mouse1-up	Spots a mark on play area and character will run to reach that point
mouse3	Rotates the camera around
mouse3-up	Rotates the camera around
wheel_up	Zoom_in
wheel_down	Zoom_out

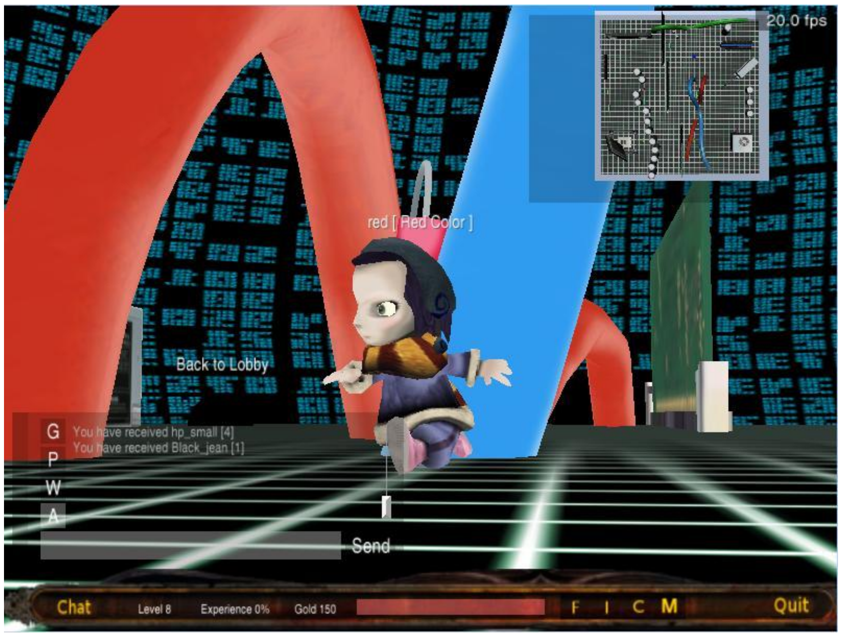
4.c.ix – Camera Control

For Debugger, we the Client Team decided to model camera control around other MMOs' third-person, mouse-steering design. The camera is constantly floating around the player's avatar, centered on him/her as the player navigates the world with the keyboard. The player has control over the camera's orientation with respect to the avatar via the mouse wheel, which is used to zoom the camera in and out, and the right mouse button, with which the player may drag the camera around the invisible semi-sphere surrounding the avatar defined by the camera's distance from the avatar. We feel that these two functions give the user simple and very comfortable camera control that will allow him/her to navigate the game's environments with ease.

Orientation is everything: the camera needs to constantly follow the player's avatar as it moves around the scene. When the player presses the W or S keys to move forward or backwards (respectively), the camera moves as well. The camera moves through the scene: for every unit that the character moves forward, the camera will move forward that many units.


You have received hp_small [4]
You have received Black_jean [1]

The camera floats away from the avatar. Players may control the camera by rotating it up, down, and around the avatar, but it wouldn't feel natural if the camera were allowed to drop below the floor! If the user brings the camera down too low...



… The collision handler will detect it and push the camera up. Here, the camera is "resting" on the ground, unable to cross through.

4.C.x Character (General Overview)

Just like almost every MMORPG, a player would always start with a specific character of choice that they will use to help accomplish a particular set of goals that they will have to face either alone or with a partner. In *deBugger*, every player has the choice of choosing a preferred character. That character is given a set of attributes—name, level, gold, health points, move speed, attack speed, etc. Once created, the player will have the ability of performing several basic tasks such as the ability to move around the environment, travel to other environments and meeting other existing players. Each player will also be given the opportunity to defeat many bugs using this specific character. As shown in Figure 4.C.x-1, this may be one of several characters that a player may be choosing in *deBugger*.

This section will cover the general overview of the important components that make up a character, which will be movements using different interfaces and a brief explanation of how the use of collision can enforce some limitations to these movements such as preventing the character from walking into an obstacle.

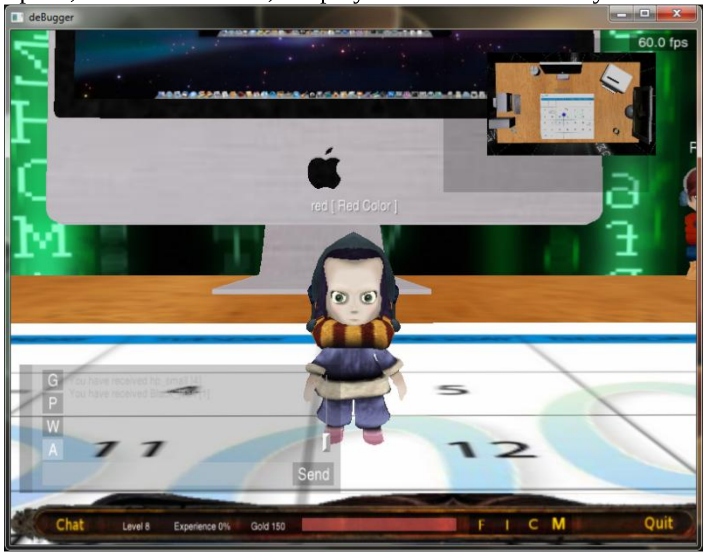


Figure 4.C.x-1: Character (Avatar)

4.C.x.1 Character Movements (Mouse Interface)

In *deBugger,* the player will be given the choice of two usable interfaces, in which, one of these will require the use of a mouse. Moving a character is quite simple. All you have to do is grab the mouse and left-click onto any parts of the environment (Figure 4.C.x-2) and the character will simply and automatically walk to that destination that you've clicked (Figure 4.C.x-3). Do notice that whenever you click anywhere in the environment, a small sphere object will reveal itself, signifying that it is the location you have clicked on. Within a few seconds, this sphere object should visibly disappear.

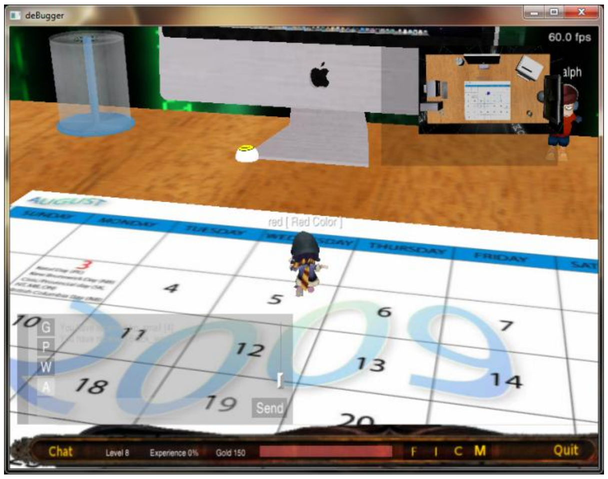


Figure 4.C.x-2: Clicked on Environment

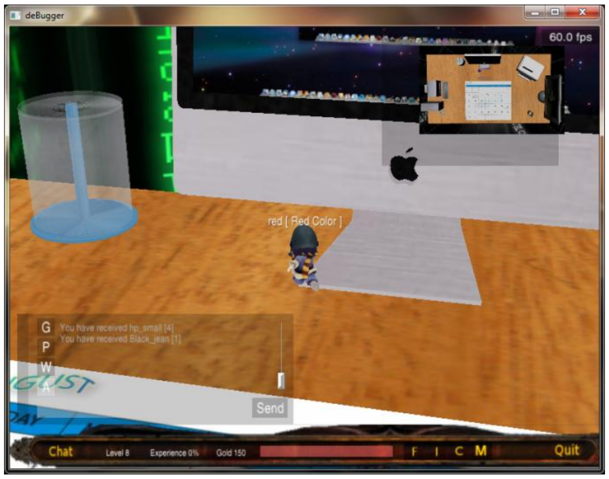


Figure 4.C.x-3: Moved to Destination Point

4.C.x.2 Character Movements (Keyboard Interface)

As you've probably seen above that you can use the mouse to move the character quite easily just with a single click. Now you will be introduced with the other interface, which is requires the use of the keyboard. There are two sets of directional keys that you can use to move the character. The first set consists of the keys—*W, A, S, D*—which are located on the left side of your keyboard. The second set of keys consists of the keys—*Up Arrow, Left Arrow, Down Arrow, Right Arrow*—which are most likely located on towards your right side of the keyboard. Using either of these sets of keys is entirely up to the player whether you prefer your left hand, right hand or alternate if you wish to do so. From now on, we'll just be referring these keys by their direction.

You might notice that there are only four keys, four directions, that you can physically push, but you can actually use a combination of two keys to perform a new direction. For example, if you want to move up, you would simply just push the key corresponding to up. If you were to choose to move at an angular direction such as upper-left, you can hold down both the up and left directional keys to move at a north-west-wise direction. This applies to all directions making total of 8 possible directions a character can move on the screen. Do note that pushing opposite directions at once, for whatever reasons, is not advised. As shown above in Figure 4.C.x-4, you can see a player pushing a combination of up and left which forces to character to look at an upper-left direction.

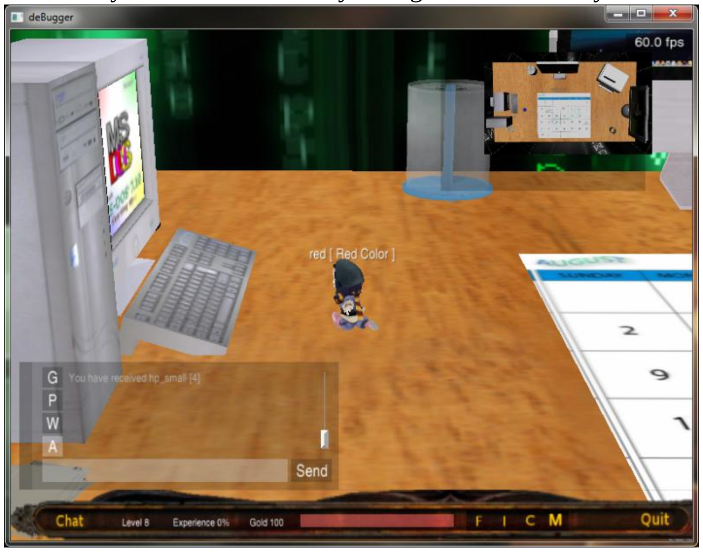


Figure 4.C.x-4: Using the Keyboard

4.C.x.3 Collision Support

As with every game, there is some type of collision support for all objects in the game to prevent the object from going into any geometry that it is not supposed to. Collision is also used to keep an object above an environment surface to make it appear as if they are walking on the environment. Collision can also be used for many other things such as generating events, but for this situation, the character will not be allowed to walk off a table or into a computer component.

Although you may not be able to see this below in Figure 4.C.x-5, whenever a character moves off a surface, the game will know that it is no longer colliding with the environment. the character will not be able to move any further. For the computer keyboard as shown below in Figure 4.C.x-6, the game will know that you are trying to walk into an object that you are not allowed to. And exactly like the first situation, trying to walk off a table, you just simply will not be able to advance any further due to this limitation. Collision is also used throughout many parts of the game other than just for the character. Although this might've been explained in other sections, whenever you use the mouse to click on the environment does, in fact, make use of collision events to determine the exact location that you may have clicked on. This is just one of the few others that take advantage of collision.

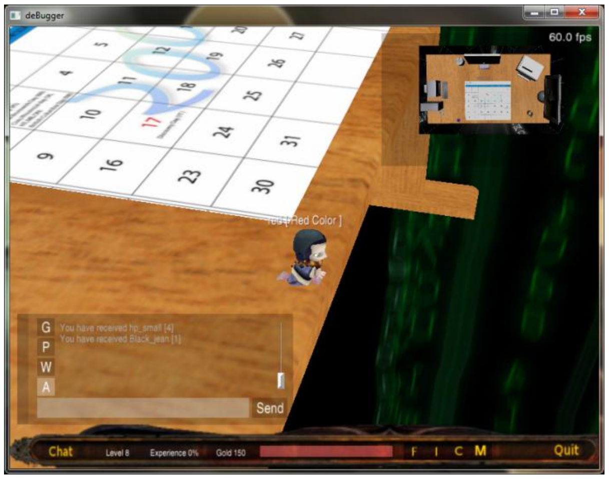


Figure 4.C.x-5: Moving off the Table

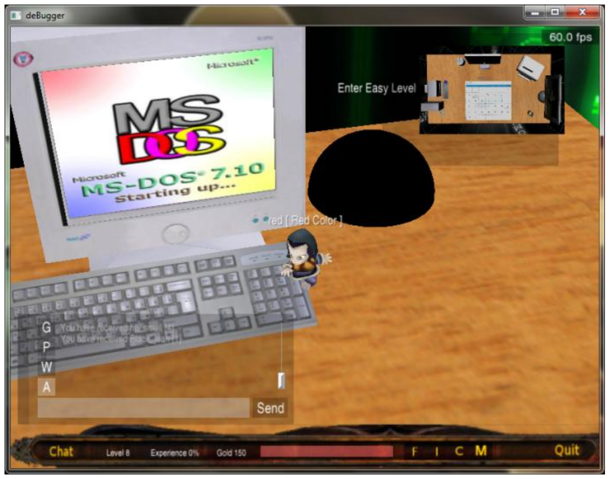


Figure 4.C.x-6: Moving into the Keyboard

For further information and details, look for the section called *Character (Understanding the Code)* in Chapter 5.

4.C.xi NPC (General Overview)

What are NPCs? NPCs are just objects within the game that represents many different things to perform certain basic tasks. In *deBugger*, there are exactly three types of NPCs. The most widely used one represents a portal that transport a player from one map to the other. The portal object is simply just a black sphere that transports the player instantaneous whenever it is within range. The black portal can be seen in Figure 4.C.xi-1.

Another type of NPC would be one where a player can simply interact with it through a single mouse click. Upon doing so, a menu would appear that presents a brief dialog as if it is talking to the player. The player would also be given a few responses to continue the conversation or as another means of warping by clicking

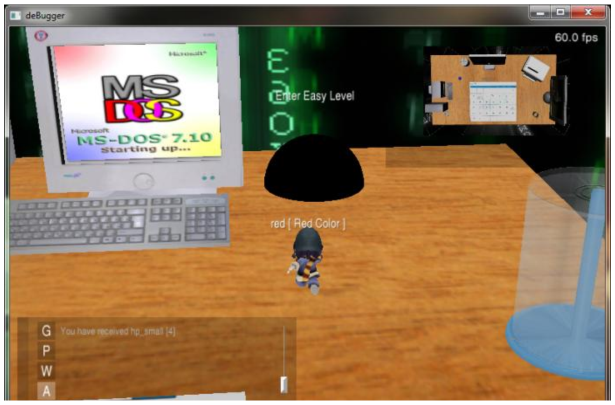
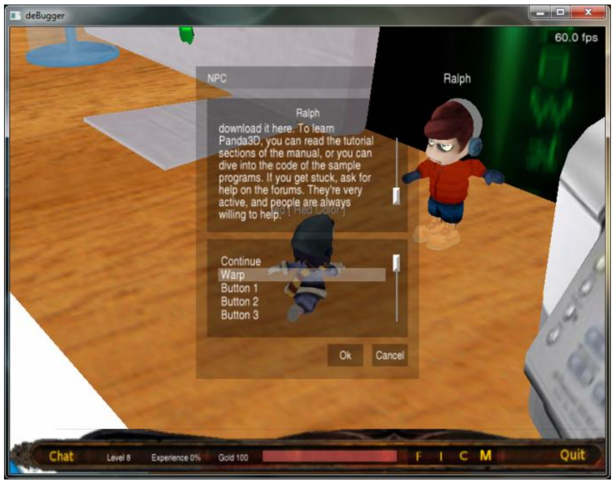


Figure 4.C.xi-1: Black Portal NPC

in the menu. This is similar to the portal, however, unlike the portal, you are required to click on the menu or else you will not be transported.

The last type of NPC represents a shop. Similar to the previous type, whenever a player clicks on the NPC, a shop interface would appear instead. Any given player would be able to buy an item from this shop NPC as long as if they have the amount of gold required.

All of these NPC can and may look like a character or just a random object like the portal, but isn't controlled by anyone. If the intention was to create a character-like NPC, it can be AI controlled, which is similar to a bug, in the sense, that it can roam around within a fixed radius. A NPC is really just something for the player to interact with other than just bugs and other players. The other two NPCs with different interfaces can be seen below.



Ralph
download it here. To learn Panda3D, you can read the tutorial sections of the manual, or you can dive into the code of the sample programs. If you get stuck, ask for help on the forums. They're very active, and people are always willing to help.

Figure 4.C.xi-2: Conversational NPC

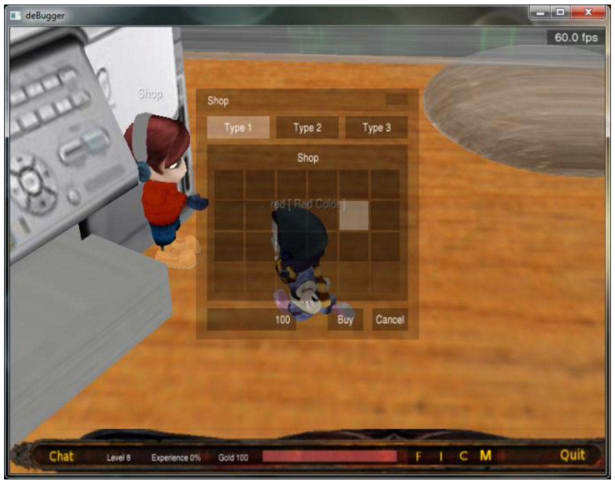
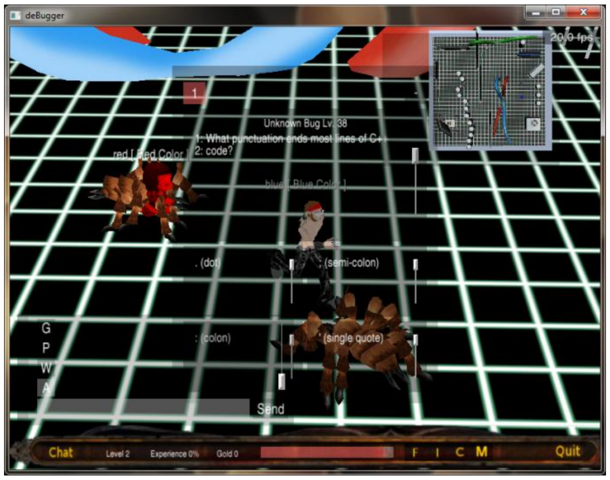


Figure 4.C.xi-3: Shop NPC

For further information and details, look for the section called *NPC (Understanding the Code)* in Chapter 5.

4.C.xii Battle System (General Overview)

In *deBugger*, the *Battle System* is what brings everything together because this is what the game is all about, *Bug Hunting*. The *Battle System* enables the interaction between players and bugs through a series of questions. All players must face this challenge to defeat every single bug that gets in their way. And the one that defeats a bug will be rewarded with unique items and gold that can be exchanged for better items. This is essentially the foundation of the entire game. The *Battle System* consists of many components such as the battle interface, player interaction, bugs, rewards, questions and many more. As shown in the figures below, you can get a glimpse as to what the *Battle System* will look like.



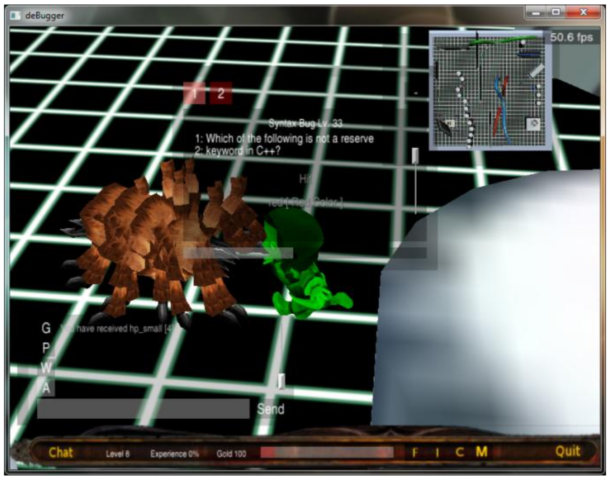


Figure 4.C.xii-1: Multiple Players in Different Battles

Figure 4.C.xii-2: A Player Against Multiple Bugs

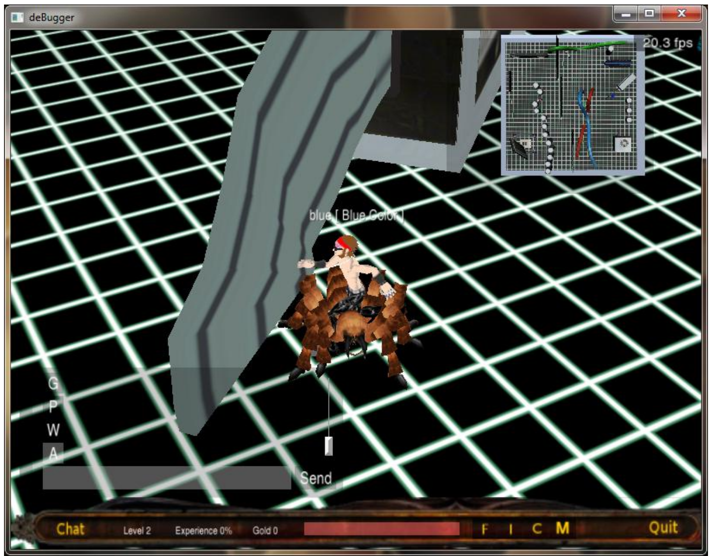
This section will present a general overview of the important components that will make up the entire *Battle System* as mentioned previously.

4.C.xii.1 Battle Interface (Battle GUI)

The battle interface consists of a graphical menu that will simply display the question and the available choices where only one will be correct. Whenever a player responds to the question, that particular choice will be eliminated. The question box and all response buttons will consist of a scrollbar so that if the question or response is longer than the viewing space, the player will be able to scroll through the remaining parts of the question and responses to read the rest of what was given. On top of the battle interface consists of a tabular view of a list of bugs that the player has been encountered with. Each of these tabs will contain the remaining amount of health points, questions that each bug has left. Whenever the tab no longer appears to be red, the bug is defeated and the player will be properly rewarded. You can see all of these graphical features in the above two figures.

4.C.xii.2 Bugs

Because this is a *Battle System* that deals with battle situations, there must be a player and an enemy target, which in this case is a bug. Without any of these, the *Battle System* would simply be never used. The bug acts almost exactly as a character except the fact that a bug is controlled by AI, whereas the character is controlled by a human player. Each bug will spawn with a certain amount of health points, which is also equivalent to the amount of questions that each bug may contain. If you may look above again, the players are interacting with the enemy bugs. The bugs can visibly come in many forms and sizes. Each bug will also vary in attributes resulting in different difficulty for the player. The difficulty of each bug is simply determined by its level that will generate the corresponding question. Whenever a bug takes damage, the bug is simply losing questions. There are also two types of bugs. There are aggressive and passive bugs. Aggressive bugs will simply look for a nearby target and initiate the attack. Passive bugs will not attack you unless you provoke it by clicking on it. The figure below shows that any player can simply walk up to a passive bug without being harmed and no battle will be initiated.



4.C.xii.2 Rewards

Figure 4.C.xii-3: Passive Bug

The *Battle System* also consists of a rewards system that whenever a bug is defeated, the player will receive a random item by chance and that all players will also receive a small amount of gold. The quality of the items will also increase as the player defeats harder and more difficult bugs. The same goes for the amount of gold that each player receives. The types of items that any given player can receive ranges from potions to a wide variety of equipment that will have a chance of boosting a player's attributes. Whenever you receive a reward after defeating a bug, you can simply look at the item by going into your inventory or as displayed in the chat box as shown to the right.

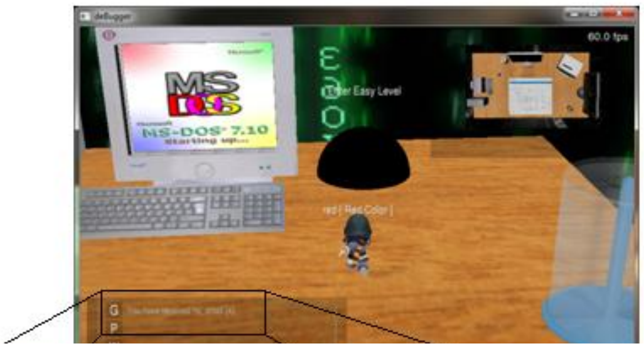


Figure 4.C.xii-4: Item Given to Player

For further information and details, look for the section called *Battle System (Understanding the Code)* in Chapter 5.

4.c.xiii – Protocols

This section describes the "*MMORPG Client/Server Protocol*" used in the debugger game program at San Francisco State University. The protocol is the game's core as it is used for client/server communication without which the game will not be able to function as a true MMORPG.

A protocol is a set of rules which is used by a computer to communicate with other computers across a network. It controls the connection, communication. And data transfer between two or more computers across a network. Most protocols have one or more of the following properties:

- Detection of a physical connection
- Handshaking
- How to start/end a message
- Formatting a message

The protocols defined in this section have been decided upon by the Client as well as the Java Server. Hence, we have established a means of communication between the Java Server and the "deBugger" client. Deviation from these protocols will result in either the Java Server crashing or the Game Client crashing. Appropriate modifications should be made to the protocols on both ends.

The protocols are situated in the "net" or networking package on the Game Client. For further granularity the net package is further sub-divided to a "request" package and a "response" package. All requests that the client can send are located in the request package. All the responses that the client is liable to receive from the server are located in the response package.

Specifications

Game Engine

Panda3D is the game engine for the development of deBugger. This game engine provides fast prototyping and useful APIs for rendering characters, game world and creating game objects in 3D environment. Its programming language is Python, which is a popular script language for game development.

Endianness

Game Client: Panda3D – Little endian

Game Server: Java – Big endian

General package format

[Total length of package][Data1 encoded][Data2 encoded]……

----> 2 bytes Short <---

(fixed)

debugger Client/Server Package format:

[Total length of package][Request/Response ID][Data1 encoded][Data2 encoded]……

----> 2 bytes Short <--- ---> 2 bytes Short <--

(fixed)

Package format for certain primitive types:

1. ***Integer (Size: 4 bytes)***

   Pydatagram allows sending integers in 32 bits and 64 bits. Here, we use 32 bits Integer, which is a 4 bytes number on the wire. Endianness will be converted when sending from Java to Python.

2. ***Short (Size: 2 bytes)***

Pydatagram sends out data package starting with a two bytes short number indicating the size of this data package. In Java Server side, it always assumes the received data package started with total length in short number format. Again, when it sends out data package, total length will be added at the very beginning of the package.

3. ***Boolean (Size: 1 byte)***

Pydatagram sends out Boolean value in one byte, 0 means false, and 1 means true.

4. ***String (Size: max of 100 bytes)***

Pydatagram sends out String by as a pair of values: String length followed by String content.

Preceding each string, a two byte short specifies the length of the string in bytes, which followed by the string content each byte representing one character encoded in Hex number.

During the course we determined that when a string goes beyond 100 bytes, an over flow occurs and the client cannot extract the data from the packet sent by the Java server. The Java server however is able to handle Strings of length greater than 100.

Hence, whenever we encountered a string of length greater than 100 which was being sent from the Java server to the Python client, we divided the string into n parts (n = orig. string length % 100), added n to the packet followed by n strings. Hence, the Python client knows that there are n strings in the packet.

5. ***Float (Size: 4 bytes)***

Similar to the String type, Pydatagram sends out the float value in paired values: Float length followed by the Float value.

Preceding each float, a two byte short specifies the length of the float in bytes, which followed by the float content each byte representing one digit encoded in Hex number.

Following are the requests used on the Game Client as well as the Bug Server i.e. Python to Java Server in PyDatagram format,

Legend:

Gray – Implemented on the client as well as the server

Orange – Not being used in the current implementation

Blue – TODO

Type	Description	Format
RequestAttack	This request is sent by the bug to attack a particular user. The user to be attacked is specified as the target_id. The id of the bug that is attacking is also sent along with the type of the bug and the damage caused by the bug. This request is only sent by the bug server.	Short Constants.CMSG_REQ_ATTACK Int attacker_id Int target_id Int Damage Short type
RequestBattleAdd	This request is sent by the bug to a particular user. A bug cannot attack a user until it has been added to the battle list. If the user's battle list is full, the bug cannot attack the user. The id of the user to be attacked is sent as the target_id and the id of the attacking bug is sent as attacker_id	Short Constants.CMSG_REQ_BATTLE_ADD Int target_id Int attacker_id
RequestBattleAcc	This is sent by the client. When the client receives a ResponseBattleAdd, the client will check to see if the bug can be added into the battle list. If the bug can be added to the battle list, the client responds with accept set to 1 or else set to 0. The id of the user that has accepted the request is sent as target_id and the id of the attacking bug is sent as attacker_id	Short Constants.CMSG_REQ_BATTLE_ACC Short accept Int target_id Int attacker_id
RequestBattleRemove	This is sent by the the bug to the client if the bug has been killed or if the client has run too far away. The bug id is	Short Constants.CMSG_REQ_BATTLE_REMOVE Int attacker_id

	sent as attacker_id and the user id is sent as target_id	Int target_id
RequestBuddies	This is sent when the client wants to see his/her buddy list. It is an almost blank packet consisting of just the short that identifies the request as a buddy request	Short Constants.CMSG_SEND_BUDDIES
RequestBuddyAdd	This is sent when the client wants to add a buddy to his/her list. The client will have to specify the username of the buddy that he/she wants to add.	Short Constants.CMSG_INVITE_BUDDY String buddy
RequestBuddyAcc	This is sent when the requestee either accepts/denies a buddy request. The requestor's username and whether the requestee has accepted the request or not is specified in the packet	Short Constants.CMSG_ACCEPT_BUDDY_INVITE String requestorBuddy Boolean accepted
RequestBuddyRemove	This is sent when the client wants to remove a buddy from his/her list. The username of the buddy to be removes is specified in the packet	Short Constants.CMSG_REMOVE_BUDDY String username
RequestDead	This is sent by the client when either the client is dead or the client kills a bug. The id of the dead user or bug is specified in the packet as object_id	Short Constants.CMSG_REQ_DEAD Int object_id
RequestGlobalChat	This is sent when the user is chatting in global mode. The	Short Constants.CMSG_GLOBAL_CHAT String msg

	message is sent in the packet	
RequestHeartbeat	This is sent to the server 10 times per second. This protocol is to let the user know that the client is still connected to the server. Without this protocol, the servers will timeout the client. Client's state has not changed, but enough time has passed that the client would like an update from the server. Each client now is a thread and associated with a specific username, when request comes, server knows which client is sending the heartbeat, so no user id is required. The server will be able to check the client's queued response and send all of them out to the client socket.	Short Constants.CMSG_HEARTBEAT
RequestInventory	This is sent to the server when the user wants to see all the inventory items. Just the short that identifies this request as an inventory request is specified in this packet	Short Constants.CMSG_REQ_INVENTORY
RequestInventoryAdd	CURRENTLY NOT BEING USED. This can be used whenever the user wants to buy something from the shop. The itemId and the count is specified so the user's inventory gets updated with the specified number of items	Short Constants.CMSG_REQ_INVENTORY_ADD Int itemId Int count
RequestInventoryR	This is used whenever the user	Short

emove	uses an item. The id of the item used and the number of items used is specified.	Constants.CMSG_REQ_INVENTORY_REMOVE Int itemId Int count
RequestItem	This is used whenever a bug dies. A random item of the level of the bug is returned to the use. NOT USED CURRENTLY	Short Constants.CMSG_REQ_ITEM Int bugLevel Int bugType
RequestLogin	This is used on the login page. The username and password that the user has entered is authenticated against the database. Client requests to login with a username and password. The server validates this and responds with ResponseAuth	Short Constants.CMSG_AUTH String username String password
RequestLogout	This is used when the user wants to disconnect from the server. Just the short that identifies this request as a disconnect request is passed	Short Constants.CMSG_DISCONNECT
RequestMap	This is used whenever the user is switching maps. The new map id is specified in the packet	Short Constants.CMSG_SEND_SCENEID Int map_id
RequestMove	This is used whenever a bug or a user moves. The id of the bug/user, x y z coordinates, facing direction i.e. h, speed and whether the bug/user is moving or not is specified in the packet. Client issues a change to their	Short Constants.CMSG_MOVE Int user_id Float x Float y Float z

	location, and isMoving flag. This is used when a client wishes to move or stop moving. It is followed by creating a number of ResponseMove responses which are sent to other logged in users. Server will update other users with these responses.	Float h Float speed Boolean isMoving
RequestPrivateChat	This is used when one user wants to chat privately with another user. Any other users logged in at that moment will not be able to see the message typed. The user to whom you want to whisper to is specified as targetName	Short Constants.CMSG_PRIVATE_CHAT String targetName String msg
RequestPublicChat	This is used when a user wants to chat publicly with whoever is in the same map as him/her.	Short Constants.CMSG_PUBLIC_CHAT String msg
RequestQuestion	This is used by RemoteBug to request a question after the bug has been created. The bug id and bug level is passed.	Short Constants.CMSG_REQ_QUESTION Int bug_id Int level
RequestRegister	This is used by the client when a user tries to register himself/herself. The specified parameters are extracted from the text boxes provided to the user in the Registration Page	Short Constants.CMSG_REGISTER String username String password String firstName String lastName Int student_id String email

		Short avatar_id
RequestRegisterBug	This is used by the bug server to register the bug with the java server. The bug details are specified in the packet.	Short Constants.CMSG_REQ_REGISTER_BUG Int bugId String bugType Short bugLevel Float bugScale Short modelId Short sceneId Float x Float y Float z Float h Short isBoss Int maxHealth Int healthPoints Int money

Following are the responses sent by the Java Server to the Game Client as well as the Bug Server.

Type	Description	Format
ResponseAttack	The server sends this response to the user specified in the packet. Via this packet, the appropriate damage is caused to the user receiving this packet	Short Constants.SMSG_REQ_ATTACK Int attacker_id Int target_id Int damage Short type
ResponseBattleAdd	This is sent by the server to the user specified in the packet. It is used so that the user knows that a bug is trying to attack	Short Constants.SMSG_REQ_BATTLE_ADD Int attacker_id Int target_id
ResponseBattleAcc	This is sent by the server to the bug server. This protocol is to let the bug know whether the user specified in the RequestBattleAdd has accepted/declined the battle request.	Short Constants.SMSG_REQ_BATTLE_ACC Int bug_id
ResponseBattleRemove	This is sent by the server to the bug server to stop the bug from chasing the targeted user infinitely	Short Constants.SMSG_REQ_BATTLE_REMOVE Int attacker_id Int target_id
ResponseBuddies	This is sent by the server to the user who requested for a list of his/her buddies	Short Constants.SMSG_SEND_BUDDIES String buddies

ResponseBuddyAdd	This is sent by the server to the user specified in the RequestBuddyAdd. This is to let the requestee know that someone wants to add them in their friends list	Short Constants.SMSG_INVITE_BUDDY String requestorBuddy
ResponseBuddyRemove	This is sent by the server to the user specified in RequestBuddyRemove to let the user know that he/she has been removed from the friends list. The buddy specified in the packet needs to be removed from the friends list	Short Constants.SMSG_REMOVE_BUDDY String removedBuddy
ResponseBuddyAcc	This is sent by the server to the requestor of a buddy add request. This lets the requestor know if his/her buddy request has been accepted or not. Upon receiving this response, the requestor will then add the requestee to his/her friends list	Short Constants.SMSG_ACCEPT_BUDDY_INVITE String buddy Boolean accepted
ResponseChangeMap	This is sent to all users to let them know that the user (id specified in packet) has changed scene	Short Constants.SMSG_SEND_CHG_SCENE Int user_id
ResponseCreate	This is sent to all users so that they create a user with the details specified in the packet. This will	Short Constants.SMSG_CREATE Int user_id

	instantiate a RemoteCharacter instance on the clients that receive this response. The RemoteCharacter instance represents the logged in user on the other clients.	String username String firstName String lastName Short gender Short userState Short avatar_id Short level Short map_id Float last_x Float last_y Float last_z Float last_h
ResponseCreateBug	This is sent by the server to all clients so that they can render the bug with the details specified in the packet	Short Constants.SMSG_CREATE_BUG Int bug_id String name Short model_id Float scale Short level Short map_id Float last_x Float last_y Float last_z Float last_h Boolean mode Short boss

		Boolean status Int max_health Int health Int money
ResponseDead	This is sent to all users so that they know that the user/bug (id specified in packet) has died. Upon receiving this packet the client will either remove the bug from scene or if the id specified belongs to a character, it will make the character lie down i.e. the character will look like its dead	Short Constants.SMSG_REQ_DEAD Int object_id
ResponseGlobalChat	This is sent to all users to let them know that the user (id specified in the packet) has sent a message	Short Constants.SMSG_GLOBAL_CHAT Int user_id String username String msg
ResponseInventoryAdd	This is sent to the user who had sent the RequestInventoryAdd to let them know whether the item(s) have been successfully/unsuccessfully to his/her inventory. NOT BEING USED CURRENTLY	Short Constants.SMSG_REQ_INVENTORY_ADD Boolean isValid
ResponseInventoryRemove	This is sent to the user who had sent the RequestInventoryRemove so that they know that the item has been removed	Short Constants.SMSG_REQ_INVENTORY_REMOVE

	successfully/unsuccessfully from his/her inventory	Boolean isValid
ResponseItem	This is sent to the client so that the client can add the item with the details specified in the packet to the inventory list	Short Constants.SMSG_REQ_ITEM Short itemId Int itemLevel Int itemCount String itemName String itemPrice Float speed Int hpRestore Int shield
ResponseLogin	This is sent to the user who sent the RequestLogin so that the details can be stored in the Character class (This class is ResponseAuth on the server, I named it ResponseLogin so there was uniformity in the naming convention). The server also sends a ResponseCreate to all other logged in users to notify them that a new user has logged in.	Short Constants.SMSG_AUTH_RESPONSE Int flag Int user_id String username Int student_id String firstName String lastName Short gender String email Short userState Short avatar_id Int health Int maxHealth Float moveSpeed

		Short charLevel Int experience Int gold Short map_id Float lastX Float lastY Float lastZ
ResponseLogout	Client sends out request to disconnect. Server replies and removes user from current list also inform other clients to remove the user. This is done by sending ResponseRemoveUser to other logged in users’. All ResponseRemoveUser will be queued up in all OTHER users’ update queue.	Short Constants.SMSG_DISCONNECT
ResponseMap	This is sent to the client to let the client know that that the scene has been changed successfully/unsuccessfully NOT BEING USED CURRENTLY	
ResponseMove	This is sent to all clients so that they can update the user (id specified in packet) with the details specified in the packet	Short Constants.SMSG_MOVE Int user_id Float x Float y Float z

		Float h Float s Float m
ResponsePartyChat	This is sent to all users belonging to the same party. TODO	Short Constants.SMSG_PARTY_CHAT Int user_id String username String msg
ResponsePrivateChat	This is sent to the user (id specified in the packet). The receiver will know that the sender (username in packet) has sent a message	Short Constants.SMSG_PRIVATE_CHAT Int user_id String username String msg
ResponsePublicChat	This is sent to all users logged in. The sender is specified in the packet as username.	Short Constants.SMSG_PUBLIC_CHAT Int user_id String username String msg
ResponseQuestion	This is sent to the client that has requested a question. A random question is generated depending on the bug level specified. The question level is dependent on the bug level. If the length of the questions or options exceed 100 bytes, they are broken down into parts and sent in the packet. The number of parts are specified in the	Short Constants.SMSG_REQ_QUESTION Int bug_id Int question_id Int type Int questionParts String question (the same number of strings as questionParts) Int optionParts String options (the same number of strings as optionParts)

	packet.	Int answer
ResponseRegister	This is sent to the user that had tried to register. The user will receive a flag that will indicate if the registration was a success or not	Short Constants.SMSG_REGISTER Int flag
ResponseRegisterBug	This is sent to the bug server after the RequestRegisterBug has been processed by the client.	Short Constants.SMSG_REQ_REGISTER_BUG Boolean success
ResponseRemoveUser	This is sent to all clients so that they know that the user (specified by user_id in the packet) has been removed from the current scene	Short Constants.SMSG_REMOVE_USER Int user_id
ResponseUpdate	The user's global status needs to be updated. This is always the response of heartbeat requests. One or more "update responses" will be packaged in this response. The response sends out all queued Responses and cleans the queue for this client.	*The format is any kind of Response format described in this table. Since this Response is just sending out all queued responses belong to one client, there is no specific format.*

4.c.xiv – Minimaps

1. **Mini-map definition**

Mini-map is a miniature map of the whole or part of the world map. Often, it is place on the corner of game screen to display the location of the player, allies, enemies, items, and terrain. Usually, there are 2 types of design, move the point or move the map.

1.1. Move the point: the small version of the whole map will be display, and it will not move. The point (icon) represent the location of each unit inside the map will be moving.

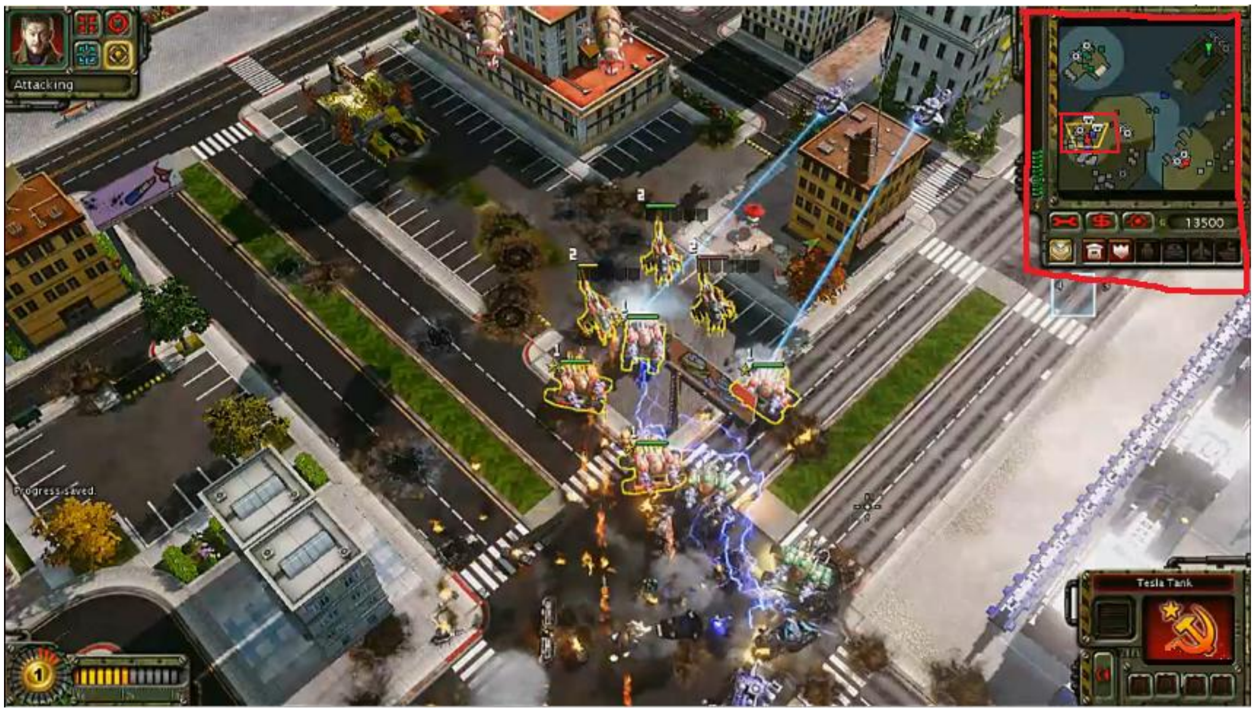


Figure 4.c.xiv.1

This is the screen shoot from Red Alert 3. In the big red square are the mini-map, and the small red square is the area represent the main screen. The 2D mini-map is the bird eye view of the whole 3D world. Each unit in the 3D world is represented by points. The mini-map just sit still, only thing that move are the points. I think it is the easiest way to program, because we only need to care about the movement of the points; we don't need to deal with way to display the mini-map. I think if there are not too many "moving" things to display, it should take much memory; otherwise, it may cause lag.

1.2. Move the map: the point that represents the main character will not move. Only display points that represent other units which near the main character will move with the map. It usually displays only part of the map around the main character, which moves in the opposite direction as the main character to simulate the main character's movement.

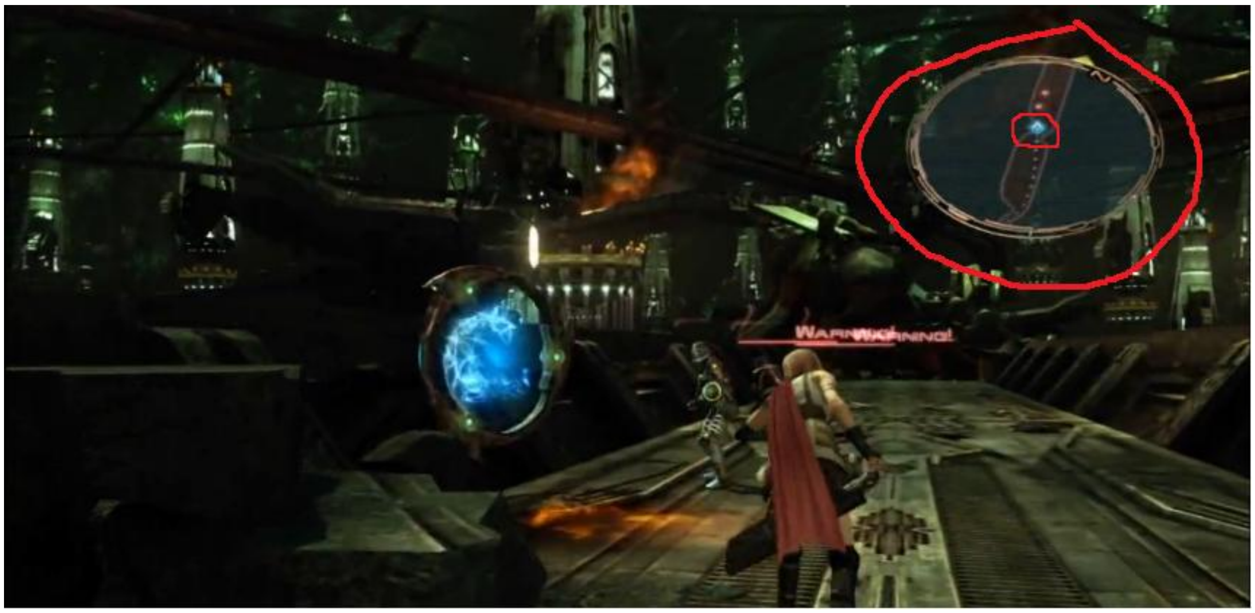


Figure 4.c.xiv.2

This is the screen shoot from Final Fantasy XIII. In the big red circle is the mini-map, and the small red circle is the point represent the main character. The path looking thing in the center of the mini-map represents the current terrain of the 3D world. The point in the mini-map just sit still, only thing that move is the mini-map, imagine the map is moving the opposite direction / rotation of the character. If programming, first, we need to move the mini-map everything opposite to the character. Second, we need to have a way to cover up the rest of the map that don't need to display, in a transparent way, so it doesn't block the player's view. I think this way will have a constant consumption of the memory, since it takes the same amount of memory to move the map and only display a few moving thing.

2. **<u>Usage of mini-map in "Debugger" game</u>**

Tell character location in the world by the mini-map

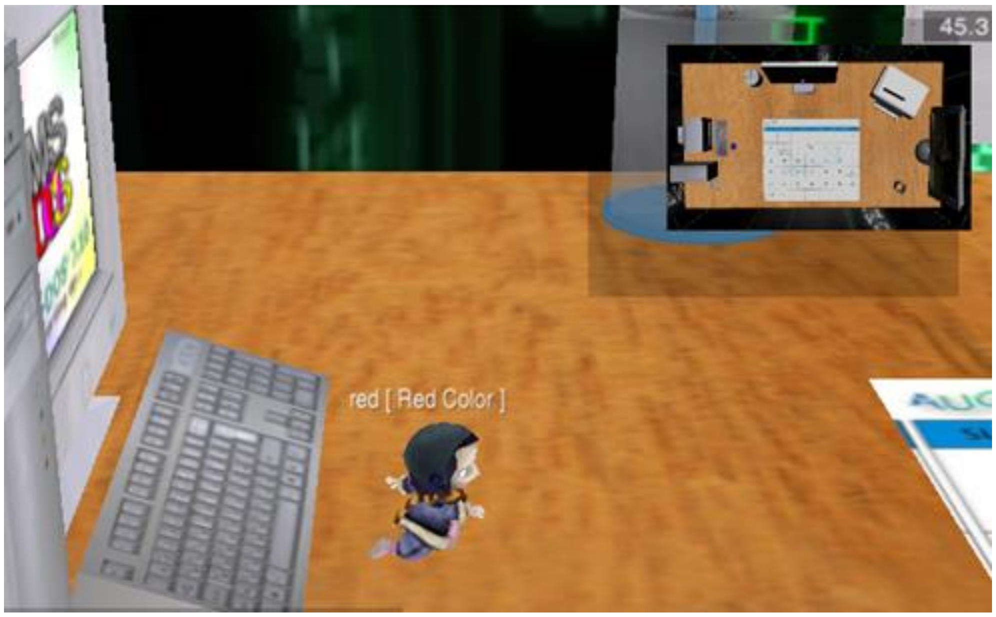


Figure 4.c.xiv.3

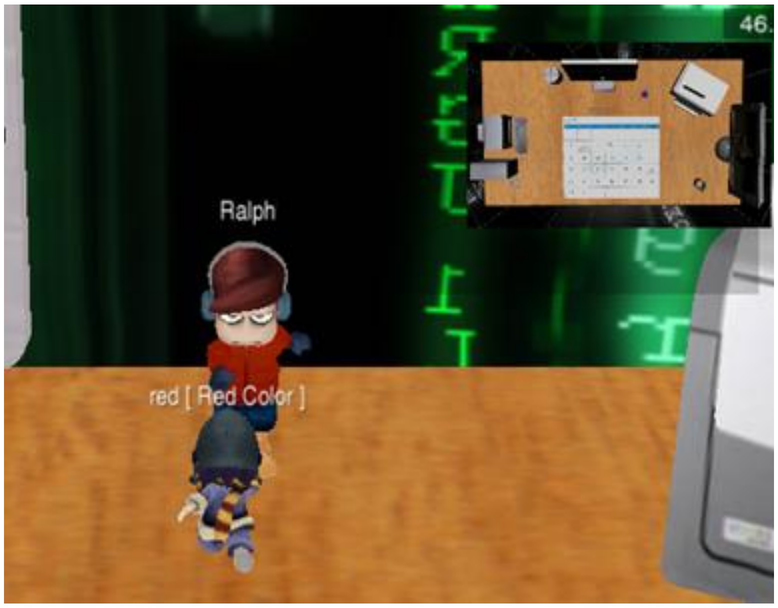


Figure 4.c.xiv.4

The 2 image above show the character “Red Color” walks from a PC to NPC player Ralph. The blue dot on the mini-map shows the position.

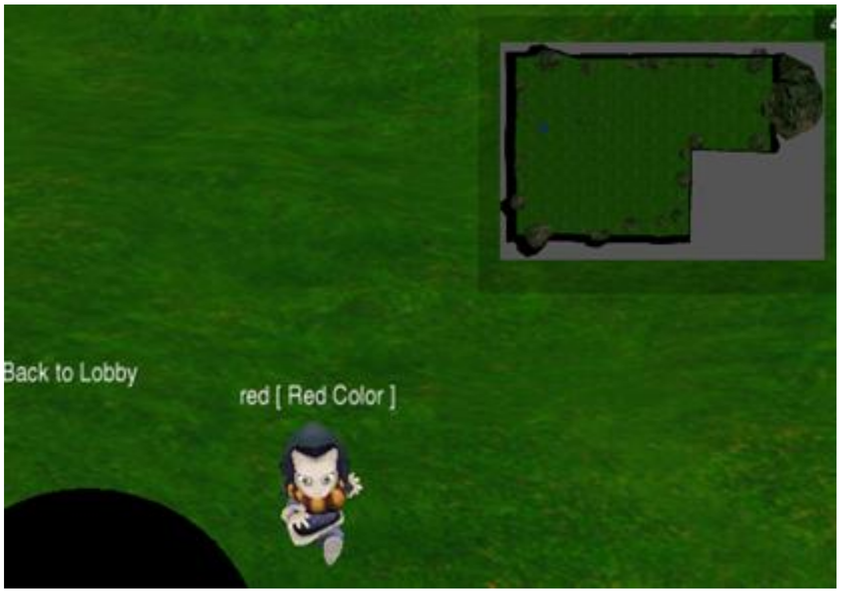


Figure 4.c.xiv.5

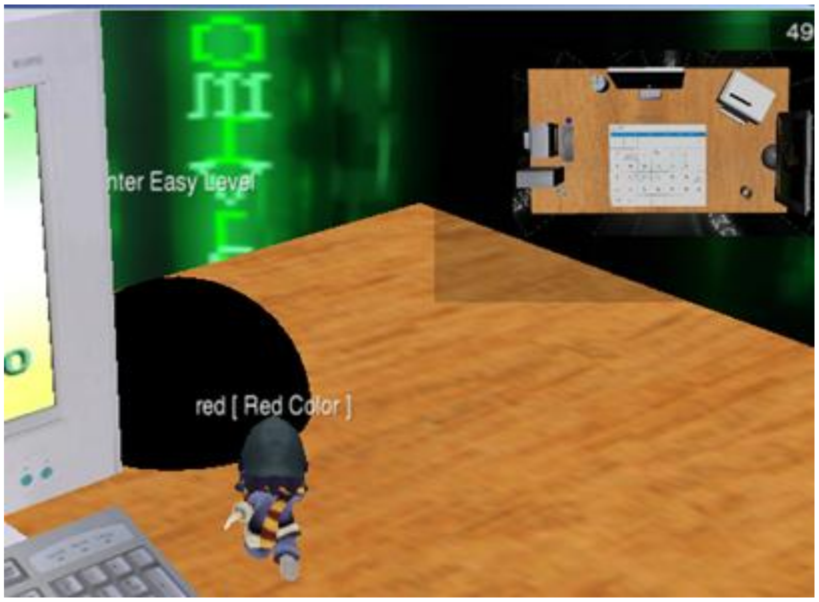


Figure 4.c.xiv.6

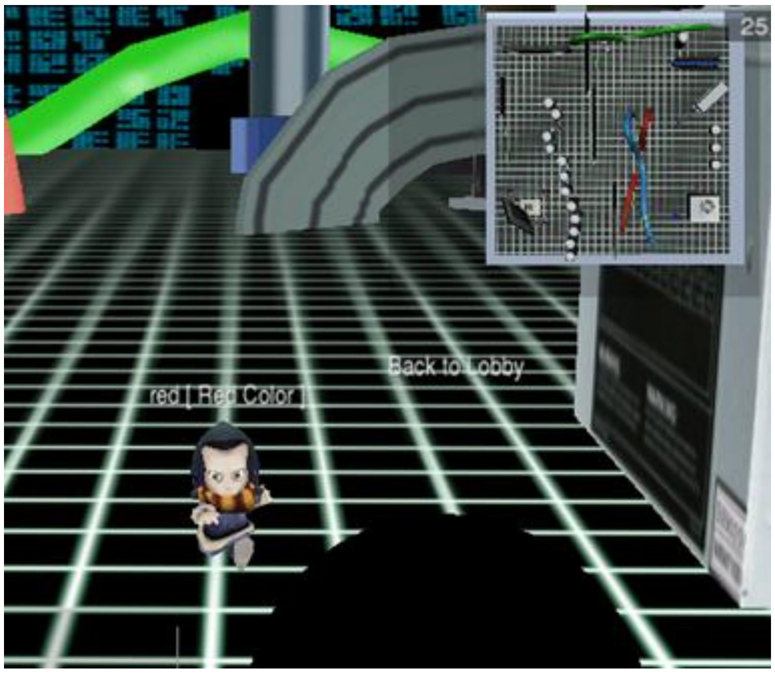


Figure 4.c.xiv.7

NPC player Ralph and those black half spheres are gateways to another map. World.py has the current position and current map ID. Minimap.py has task function "*def* ***updatePosition****(self, task):*" it always running until it was remove by user or exit the game; it will keep on checking the current position and current map ID. When it found map changed it will remove the old mini-map object and point object, then recreate new map and point. The 3 images above show map switched to Panda map, and back to Lobby, then to Motherboard map.

3. **<u>File Minimap.py</u>**

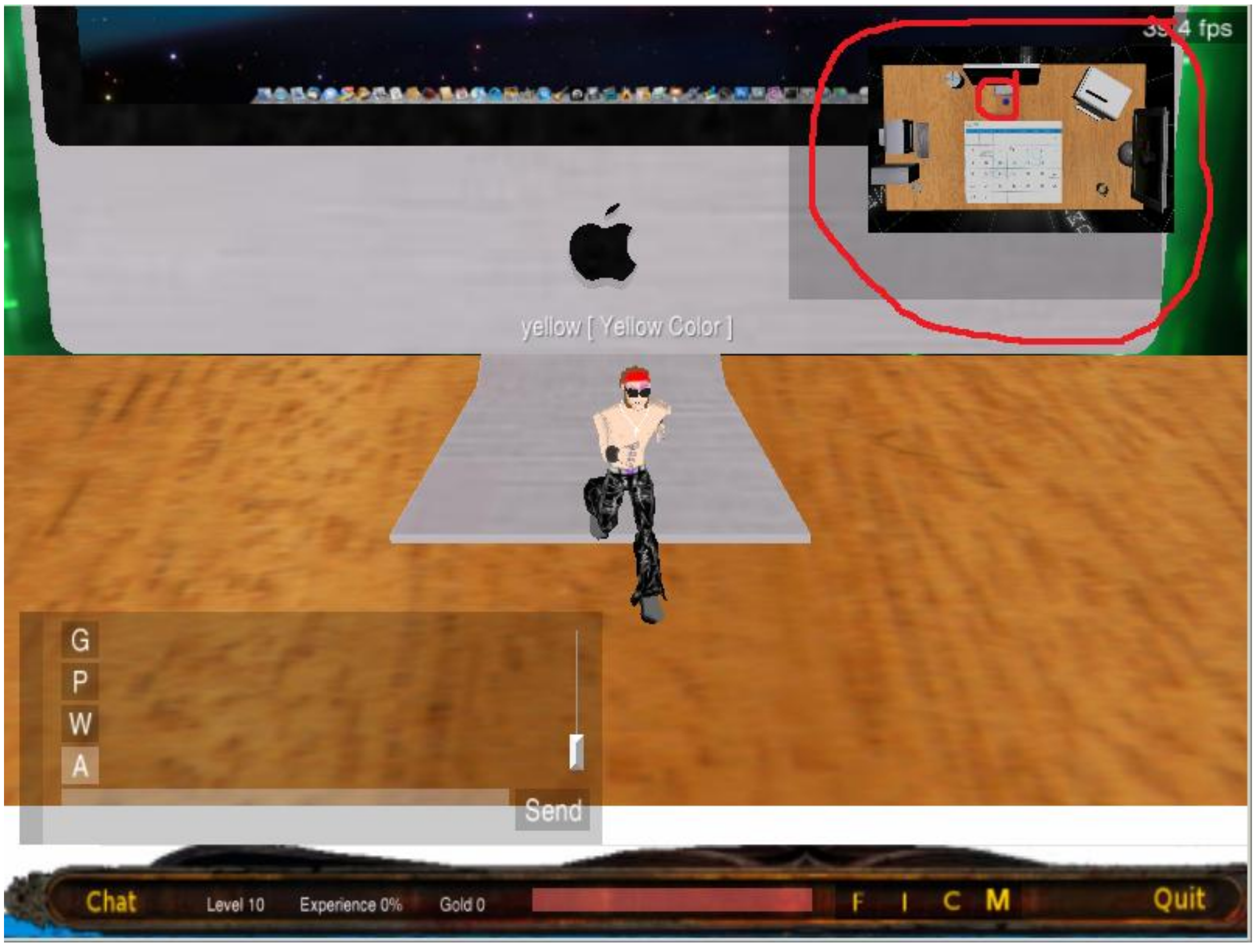


Figure 4.c.xiv.8

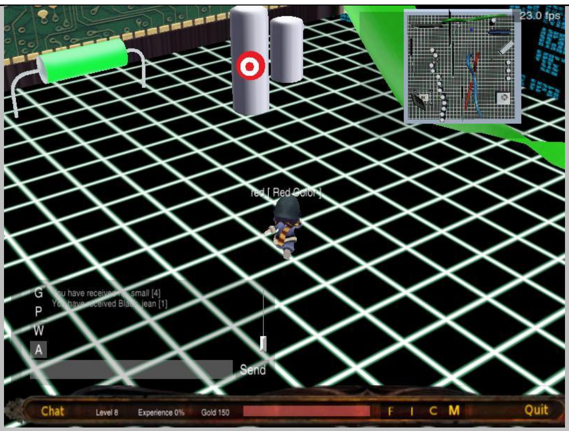
This is the first mini-map display after first time log in.

4.c.xv – Mouse Picker

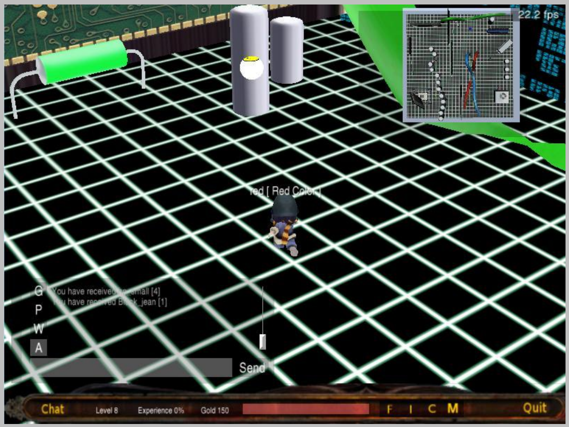
Point-and-click actions are key when it comes to making game's controls clear and comfortable. People constantly interface with their GUI operating systems and applications through the mouse, pointing and clicking their way through filesystem directories, internet browsers, and video games. For a 3D MMORPG like Debugger, clicking is a must. Most of the game's point-and-click logic can be found in main.World.MousePicker.MousePicker.py.

When looking at the game's world, there are quite a few things that the user can click on: the ground, the sky, NPCs and bugs, his/her own avatar, and other players' avatars to name a few. For the user, the action should be easy and come naturally: over the mouse over something in the game's window and click away. Implementing this in Panda3D isn't quite as simple, but the concepts remain the same.

When the user clicks the screen, he/she is clicking at pixels on a 2-dimensional plane. The game's scenes, however, are 3D rather than 2D, and they lack *de facto* pixels. What we have to work with are the camera—the in-scene device that sends data from the game to the graphics pipeline—and three-dimensional geometric shapes. So, when the user clicks a part of the image seen through the camera's eye we need to figure out exactly what in the scene the user has picked.



Clicking the capacitor at the bullseye sends a ray from the camera through the scene, colliding with the capacitor, the ground, and the scene's background. However...



Since the capacitor is closer to the camera than everything else in the ray's path, it is the primary case that the client handles, as seen by the movement target object.

4.c.xvi – Other Panda 3D Issues

We found out that in Python we cannot do a datagram.getString() for a string of length greater than 100 bytes. This causes an overflow and causes the client to crash. The only time we encounter this problem is when the Java Server sends a ResponseQuestion to the client containing the question and its options. The solution to this problem was to break down a string into parts of length 100 each and transmit these strings preceded by the count of the number of strings the original string was broken down into. Refer to ResponseQuestion protocol in the Protocol section for packet format details.

One other minor issue encountered was lag between the bug server and java server. However this is subject to network traffic and also depends on the server on which the Java Server is hosted i.e. thecity. We found that if there are many programs running on thecity, it introduces considerable lag.

Another issue involving jerkiness of movement of bugs was solved after introducing a 1 second delay while sending move requests from the bug server. The reasoning behind the delay was that the bug server was sending out many move requests too fast for the Java Server to process within the heartbeat interval. As a result the Java Server was updating the bug positions only after the heartbeat is received thus introducing jerkiness.

4.d – Server Team

The job of the server team was to implement any changes from the Nursetown server and extend and implement new features into the Debugger. Whatever was required to be done on server side, the server team had done it. The team was small at first because the server code wasn't being changed very much but over time as the game added new features, a lot of changes to the code were needed so more people were added to get things done quicker. The first milestone required the server team change the login and registration protocol for the new game client to be able to login and also register new characters in the game. The second milestone is were a lot of changes happened. The server used to look more like Nursetown but after the second milestone, it looked more like our own. One of the biggest changes between the Nursetown server and the Debugger server was that the Debugger had non-player characters or NPC added into them. This was done with a modified game client that was called the bug server. The implementation and how it was achieved as including all the other milestones will be discussed.

First milestone was to get the login and registration to work with our new client. It was a simple change but it required communication with the protocol team in order to make sure if things were running ok. The registration and login were already in the created from the last team we just had to modify them slightly. Registration required more changes then login because at first the registration, would only create a new person in the database, but didn't add any of the new data that was needed in our database and for the game to run correctly. Modifications were done to the database and we were able to get registration to run correctly after that. Jason and Kaven did this. Jason doing the protocol on the client and Kaven doing the implementation and modification of the database and game server to register new players and login them in.

4.4.2 Creating Bugs and Launching Bugs

In the second milestone required the server to deal with bugs and work in conjunction with the bug server. A new protocol was created to tell the game server that a bug needs to be created and added into the game. When the game server gets the request, it adds a bug into the active list of bugs. We first had to figure out a way to store the bugs in the server. The game server only had one active list of users in the beginning. Bugs were added into this list with other players. The problem with this was that the game server was also added responses in the queue for bugs. Bugs don't really need to know chat messages or if person died. So a separate list was created for them so there would be less stress on the game server to send out all the responses for bugs since bugs didn't need to be updated with certain responses. This lets the bugs move around and attack but not have their queues updated which causes less stress on the game server. Implementation was by Kaven and Gary. Kaven did the coding to add bugs into the server and Gary made modifications to help make the game server more efficient.

Another part of the second milestone was to launch the bug server on start. This was done by creating a process thread and launching the python. A new class BugServerProcess.java was created to handle this executing the bug server on launch. At the time of the running the bug server if the server tries to launch the bugserver from any location other then the bug server directory it will cause errors. Which is why server needs the location of the bug server. When launching the bug server the process will launch from inside the bug server directory and things will run fine. Also the bug server must be launched with the option “python -u”, if the not launched this way, the process will hang and nothing will work. In the case of if the bug server dies for some reason, the server will try to boot it up again. It does this by checking to see if the active thread from the bug client is removed. If it is, all the bugs are removed and it will try to launch bug server again. Kaven did this.

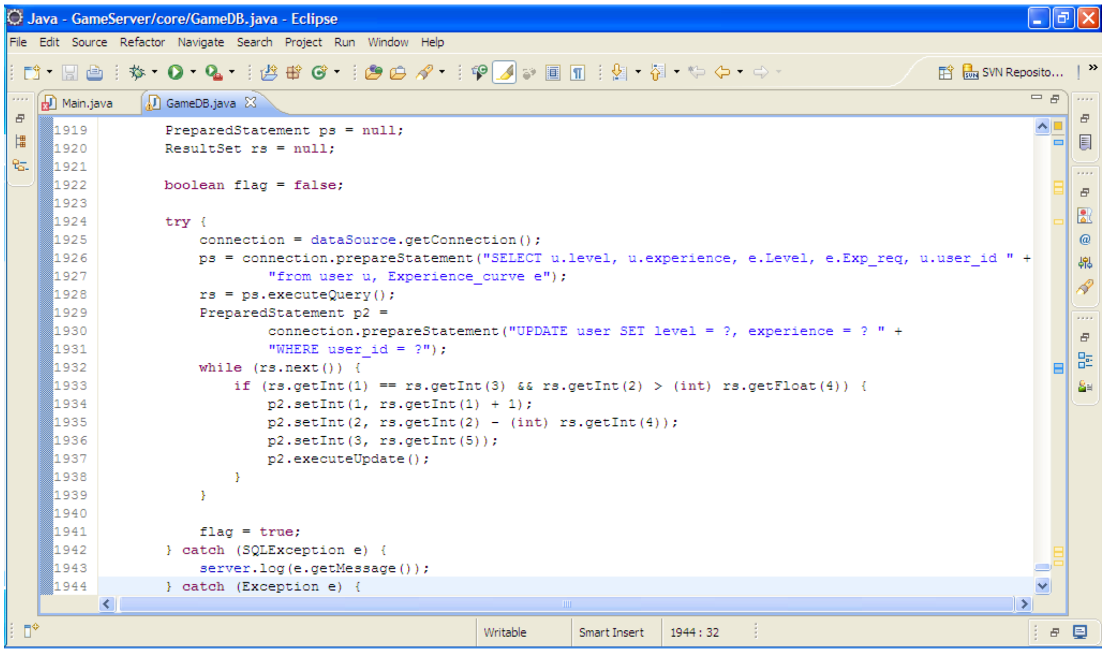
4.e –Connection to the DataBase

Please refer to the understanding the code section for this part,

4.e.1 –Connection to the DataBase

4.e Connecting to DB (minghaoli@sbcglobal.net) by Ming Hao Li

The part I was working is the Connecting DB (GameDB.java), which is used to connect to database and manipulate the data in database. In this part, I need to modify and add some more queries to the original code, which was written by Zoran. Also, there are some people help me to finish up. Since the timing problem, some of the queries is built but haven't test in the client side. The explanation of this code is very straight forward and easy to understand. Basically, the query is the language we use to talk to the database and manipulate the data in database, such as using the "select" to fetch data, using "update" to update the data, using "insert" to add more data, and "delete" to delete the data. After we finish up each of the query, we have to execute them by calling the method "executeUpdate" for update, delete, and insert, "executeQuery()" for select. The following piece of code, is the pretty much of the code that using those basis query keyword.


Java - GameServer/core/GameDB.java - Eclipse

```
        PreparedStatement ps = null;
        ResultSet rs = null;

        boolean flag = false;

        try {
            connection = dataSource.getConnection();
            ps = connection.prepareStatement("SELECT u.level, u.experience, e.Level, e.Exp_req, u.user_id " +
                    "from user u, Experience_curve e");
            rs = ps.executeQuery();
            PreparedStatement p2 =
                    connection.prepareStatement("UPDATE user SET level = ?, experience = ? " +
                    "WHERE user_id = ?");
            while (rs.next()) {
                if (rs.getInt(1) == rs.getInt(3) && rs.getInt(2) > (int) rs.getFloat(4)) {
                    p2.setInt(1, rs.getInt(1) + 1);
                    p2.setInt(2, rs.getInt(2) - (int) rs.getInt(4));
                    p2.setInt(3, rs.getInt(5));
                    p2.executeUpdate();
                }
            }

            flag = true;
        } catch (SQLException e) {
            server.log(e.getMessage());
        } catch (Exception e) {
```

File Edit Source Refactor Navigate Search Project Run Window Help
SVN Reposito...
Main.java
GameDB.java
Writable
Smart Insert
1944 : 32

It is important to notify that since some of the tuples have the same name but in different table. Thus, that very important to make the object of each table. For example, the user table and experience_curve talbe both has tuple level and experience. In case we want to separate those tuples, we need to create object of both two table like: user u, Experience_curve e. and then in the select part we can use u.level and e.level that the compiler can distinquish those two tuples. Also, in the query, the "?" that is used to indicate the value we want to use to compare. So, as long as we use it, we have to set the value in by using "setInt(index, value)" which the setInt is for integer, and the index is indicate the which number of the "?" be the value we set to, and the value is the value we use to set into the "?". Also, the setInt is indicating the be set value is integer type. Thus, if the type we want to set is String, we need to use setString instead. For the getInt(n), that is the value we get from the table according the query. The n is indicating the index of tuple. For example, select u.level, u.experience. in this case, we use the getInt(1) that is get the value of tuple u.leve. if we use getInt(2) that is get the value of tupelo u.experience.

The following methods is the methods I and some of the classmates had been added and modified

modified methods

1. Public gameuser creategameuser(Resutlset rs)
   Method used to get the user information and populate with the result for user login
   This method had been modified by several of people, which set more information for the game user object.
2. Public Vector<GameUser> getallgameuser()
   Major contributor: unknow.
   Methods used to get all the game user information and store in the vector which will be return to the method call.
   This method had been modified on the tuples name.
3. Public short getsceneid(int id)
   Major contributor: unknow.
   Methods used to get the scene's id and populate with the result for user login scene
   This method had been modified on the tuples name.
4. Public Boolean registeruser(String username, String password, String firstName, String lastname, int student_id, String email, short avatar_id)
   Major contributor: (tan)???
   Methods used to insert user information to the DB when user registers
   This method had been modified for get more game user information when the user register.

New methods

1. Public String droprandomitemfrombug()
   Major contributor: Ming
   Method used to get the random item name from bug which will populate with the result for showing what item is showed when the bug is killed.
2. Public float dropratedrombug()
   Major contributor: Ming
   Method used to get drop item rate from bug when the bug is killed. The return value will be used to calculate whether the buy will drop the item.
3. Public Vector<GameItem> getInventory(int user_id)

Major contributor: unknow
Method used to get the item list from user’s inventory, and populate with the result for when user check their items.

4. Public void getitemfrombugorstore(int itemid, long userid)
   Major contributor: unknow
   Method used to insert more item(s) to game user’s inventory.
5. Public GameItem getitems(int level, int user_id)
   Major contributor: unknow
   Method used to get the drop rate from specific level of bug, and calculate if the bug will drop the item or not. If bug will drop item, item info will be insert into inventory’s table. If bug drop gold, game user’s gold will be update.
6. Public GameQuestion getQuestion(int id, int level)
   Major contributor: unknow
   Method used to get the question and the question type from specific bug which determine by the bug’s id and level. The populate result will be return for getting the question.
7. Public HashMap<String, String> getuserinfo(int user_id)
   Major contributor: unknow
   Method used to get all the information from the user table basing on the user_id.
8. Public Vector<Float> getuserlogoutposition(int user_id)
   Major contributor: Ming
   Method used to get the game user’s logout position and update to the DB’s user table. That is used for the user login in the position where the user last time logout.
9. Public void incrementgold(int user_id, int gold)
   Major contributor:
   This method is used to update game user’s gold.
10. Public Boolean removefrominventory(int item_id, int item_count, int user_id)
    Major contributor: unknow
    The method is used to delete the user item from his inventory.
11. Public void resetallsceneid()
    Major contributor:
    This method is used to reset the scene id, which is used for the player go to some other scene.
12. Public Boolean setlastposition(int user_id, Vector<float> position)
    Major contributor: Ming
    This method is used to update the user position, where the position is the position the user logout. It also will update the user logout time.
13. Public void getupdateuserexp(int user_id, int exp)
    Major contributor: Ming
    This method is used to update the user’s experience in user table, when the experience is change.
14. Public Boolean validatedlevelup(int user_id)
    Major contributor: Ming
    This method is used to check if the game user’s experience will be sufficienct to levelup. If true, update the level and the experience in the user table.

4.f – Database Team

Database: **deBuggerUser**

Major contributor: Tun

Minor contributors: Calvin and Alvin

Documentation: Calvin

The *deBugger* database is primarily used by the game server and question creator. It have modified a few times through the whole semester. The final version of *deBugger* database has following tables: *avatar, board_game, buddy, bug, experience_curve, game_server, general_item, inventory, questions, user,* and *userinfo.* For the documentation, Tun does the *user, general_item,* and *inventory* tables. Alvin does the *avatar, board_game,* and *buddy* tables. Calvin does the *game_server, experience_curve,* and *bug* tables.

The *game_server* table is reuse from the Nursetown database. It was created to facilitate features such as emergency shutdown, maintenance shutdown, and starting-up game server remotely. For further information about the *game_server* table, look for the database section in chapter 5 Understanding the Code.

The *experience_curve* table is use to store the definition of levels in the game. It has all the levels and the experience points needed for each level. In order to go from one level to another level, the player needs to have certain amount of experience points. The level is very important for each player in the game because the players are not allowed to buy certain items or go the certain places unless they get to certain level. For further information about the *experience_curve* table, look for the database section in chapter 5 Understanding the Code.

The *bug* table is use to store all the bugs such as syntax buy, semantic buy, unknown bug, and trouble bug. The *bug* table also stores many different characteristic about each bug, such as level to play, attack damage, moving speed, and drop rate. In the game, player will gain a certain amount of gold by destroyed a bug or lost some health points if the bug is not killed. For further information about the *bug* table, look for the database section in chapter 5 Understanding the Code.

Our goal in database is to build the database which connects to mainly server. Then, server will respond to game client. Like other databases, we have to build all tables with columns. And these tables need to meet the specific requirements which is given and needed by game client. Most of the time, we follow game concept's design team and communicate game client team to discuss to specify their requests.

I have listed all tables on debugger database in table.1. The left side of table.1 is the name of table from database and the right side describes the overall function of the database. There are total of 11 tables in our debugger game database.

TABLE NUMBER	TABLE NAME	TABLE OVERALL FUNCTION
1.	AVATAR	IT WILL BE USED TO STORE AVATAR NAME AND AVATAR_ID
2.	BOARD_GAME	BASIC INFORMATION ABOUT BOARD GAME WILL BE HERE SUCH AS DIFFICULTIES, GOLD WHICH CAN GET FROM BOSS, XP.ETC.
3.	BUDDY	THIS TABLE WILL HAVE BUDDY_LIST TO ADD
4.	BUG	THIS TABLE WILL BE STORED TO GET BUG TYPE AND INFORMATION ABOUT GAINING XP, MONEY FROM BUGS.
5.	EXPERIENCE_CURVE	THIS TABLE IS A REFERENCE TO LEVEL UP THE USER.
6.	GAME_SERVER	THIS TABLE IS USED TO CLARIFY WHY SHUTDOWN WAS

		HAPPENED
7.	GENERAL_ITEM	THIS TABLE INCLUDES ALL ITEMS THE PLAYER CAN BUY FROM THIS GAME.
8.	INVENTORY	THIS TABLE WILL HAVE INPUTS FROM GENERAL_ITEM BECAUSE THE PLAYER WILL GET HIS ITEM FROM HERE AFTER HE BOUGHT IT IN GENERAL_ITEM TABLE.
9.	USER	THIS TABLE IS USED FOR REGISTERING A USER, LOGGING INTO THE GAME AND STORING THE CHARACTER'S BODY PARTS.
10.	USERINFO	THIS TABLE IS USED TO INPUT USERNAME TO CONTRIBUTE QUESTION ON THE QUESTION CREATOR WEBSITE
11.	QUESTIONS	THIS TABLE IS USED TO STORE ALL QUESTIONS , QUESTION TYPE,POINTS AND VALIDATOR NAME.

Table.1 Showing all tables with descriptions.

4.g – Testing Team

As members of the Game Testing Team, we have two tasks at hands.

1) Bug Client - a python based program that is called by the Server Team. Its function is to spawn "bugs," the monsters, in the game.

2) Testing the game's functionality, stability, and bugs - test how stable the game is and what bugs that causes the game to crash might exist.

Bug Client (Major Contribution: Alan and Game Client Team

Minor Contribution: Alvin and Tun)

The main strategy we used was to program the bug client, so that it inherited most of the functionalities in the user's client. Doing so provided us with many benefits:

- reduced the total amount of coding
- increased the code/logic consistency between the client/bug clients
- can reuse the protocols from the user's client for the bug client.

The main challenge was to take what the client had done and to convert it to a simplified version for the bug client. This consisted of several tasks.

- took out the graphic parts
- changed the routine/task from the user's point-of-view to bug's point-of-view,

so that the client will use the bug's tasks instead.

- update the server so that it can tell the difference between a bug's client vs a user's client.

4.g.i Bug Protocol-

Before talking about the Bug Protocol, we shall briefly talk about the design of the bug client. Because it is a modified client, many of the features are taken from the client, and shall not be discussed in detail here. What this will focus on is the main differences, and the changes that were done to the client, that allows the bug client to act as such.

The bug protocol is nothing more than modified request and responses. So for example in the bug move protocol, instead of "user_id", we have "object_id" Changes like these will be noted, but will not be gone into deeply for it is better explained from where it was derived from, the client protocols. The main protocols that we will be talking about is the following:

1. Bug Register
2. Battle Add
3. Battle Accept
4. Attack
5. Question
6. Battle Remove
7. Dead
8. Create Bug
9. Bug Move, Map, and the other protocols.

Bug Register:

The Register Bug (RequestRegisterBug.py) Protocol is the main method of a bug spawning on a map. Using this method, the server creates a new bug on the server which it will send out the next time a player logs in or switches into the map. The work however is not done within the RequestRegisterBug.py file however, but rather the Bug.py file, which is where most of the game logic for the bug takes place.

Register bug is called when a bug is first initialized. This is important for one main reason. When a player successfully kills a bug, that bug is not deleted from the game. Instead, it is simply reused after a set timer set by it's respawnSpeed. Because we have register bug within the initialization function, once the respawn task is complete, it calls the initialization function which will go through the entire process of setting up the bug, and effectively recreating it. Once this is done, the information that the server obtains will be for the new bug.

Comparatively, the register bug protocol is very similar to the player register protocol. The only difference is in the various attributes that bugs have which players do not have, such as level, money, boss, and bug type. Likewise, bugs do not have emails, and other various attributes.

Battle Add:

Arguably one of the most important protocols in the bug server is the battle add protocol. This protocol tells the server to tell the specific client that they are being attacked by the bug. Without this, bugs will simply follow a player indefinitely, without effect. The battle add is again called within Bug.py.

Here a bug finds the closest target given by the Remote Characters List, and if within sight radius, moves towards that target. Once that bug is within attacking radius, it will send a request to the server which then sends again, a request to the client to request addition to the players attack list. If it is successful, it will receive a response as such. However, it is completely possible that the bug will be rejected if the player is in one of other states, in which case a battle remove response will be generated instead.

On the likely chance that such a bug has been added, the bugs target will be set to the client it has attacked.

Battle Accept:

This protocol simply sets the target of the intended attacker to the client that it was able to attack. Once this protocol is raised, the bug would be able to begin attacking the player using the next protocol, which is the attack protocol. When a bug has a target, it will actively pursue that target until out of range, it dies, the player dies.

Attack:

Once a bug is in attacking mode, signified by a target, it will begin doing damage. This damage is done routinely, as the function that calls the bugAttack function is the bugRoutine function, which is called by a task. Attacking will be done based on certain conditions such as distance and attackability. Dead bugs cannot attack, and so the logic lies within the attacking function.

Question:

This protocol tells the server to send a question to the player. By doing so, we can set the time limit of the question by the bug. This protocol is also called in a task, so we can simply change the time limit by changing the bugs regeneration speed.

Battle Remove:

At the conclusion of battling, either with the death of the bug, player, or fleeing, the bugs target is simply removed, and thus the bug will resume its usual roam sequence until the next attackable target comes into range.

Dead:

Dead is not a protocol that directly relates to a bug, but there is two instances of death. Either a bugs death or the players death. With a player death, the bug will determine that since that player is no longer attackable, its target is removed. However, when a bug dies due to no health points, two things happen. First is a task that is called to respawn the bug after a set amount of time. The second is to unload the bug, removing it from the active game environment and resetting its current motions to a stop. We do not want a bug to roam while it's dead, and so instead we simply pause all such sequences.

Once a bug is dead, the task that is run will begin to count down. Once that time elapses, the bug is reloaded onto the map, and because the register bug protocol is inside of the initialization function, it is called and therefore respawns on the players map.

Create Bug:

Create bug was a protocol that was used to allow the server to determine when to create bugs. This would have been a trivial matter to change. Instead of having BugGenerator create bugs, it would have been offloaded into the protocol, where the protocol would generate a bug. Here we can define and specify various attributes, so this is a placemarker for future reference.

Bug Move, Map, and the other protocols:

By utilizing the various protocols afforded by the client, we can cause various actions such as bugs chasing after players, even through portals. The homogeneity of the move protocol allows bugs and players to move in the same fashion, without future change in architecture, and this was done by design.

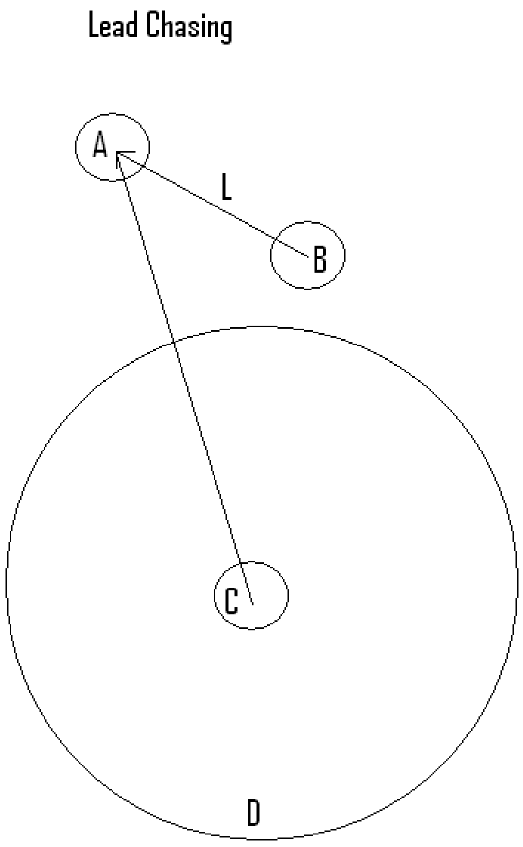
This section included other protocols because, due to the way the bug server was designed on top of the client, and not a completely separate program, we could include functionality that we could not have done in other games with two separate systems. Things such as chatting to bugs, adding them to the buddy list, trading, and auctioning are all possible with the reuse of the functionality from the client. Because the bug client receives all communication from players, all that would have to be done is the simple implementation of the proper protocol, with the specified action. Because a bug closely mimics a player, these additions are possible without architectural reworking.

4.g.ii Bug AI

The biggest outlook on bug AI for this game was the collisions. Because bugs acted like players, they could not navigate around corners without trouble. The single biggest solution which is a naïve one, was to ignore walls and terrain features. deBugger was not a tile based game, and thus using pathfinding over an infinite number of points would be troublesome. Along with the performance issues, this would have been difficult to accomplish without redesigning how the game ran.

Using the ideas from other MMO games, bugs that ignored walls were able to attack players and avoid glitches such as kiting where one player is able to attack monsters without taking damage. Not all implementations come without their compromises, and this one also raises problems. Because bugs can go through walls, they would be able to attack a player when the player themselves cannot see the bug, raising the problems of luring a bug out of it's lair, to being attacked without knowledge. There are a few solutions to that, one of which is to use a ray to see if a bug can actually "see" a player, and if so, then proceed to ignore the walls.

One human aspect of the bug AI that was implemented however, was the use of forward thinking. Because a bug was able to tell a player by their heading, a bug can preemptively go to a players destination, where they will also collide. This is useful because in this case, a bug is taking the shortest route to the player, a straight line to the players destination.



4.g.ii – Lead Chasing. A is the destination of Player B. C is the bug attempting to intercept B. Instead of C chasing after B, C heads towards A, thereby moving in a straight line and hopefully getting to the target sooner.

This works best when players heading is perpendicular to the bug.

4.g.iii Library/Component to run Bug Process

Because the bug server is run as a client all that is really needed is for the host machine of the bug server to have the same set up as the client. Using various utilities, it would be possible to pre package the bug server into one that is an executable without any external libraries.

Game Testing (Major Contribution Alvin and Tun

Minor Contribution Alan)

Our second task is to test the overall quality of the game, by testing various functionalities and its stability and detect bugs from the User's Client. Implicitly we are also testing how well all the teams are working together.

The following are all the tests that we conducted to test every part of the game's components.

i - Test: Login

ii - Test: Character Movement

iii - Test: Camera Movement

iv - Test: Chat Function

v - Test: Chat Modes

vi - Test: General Hotkeys

vii - Test: NPC (Non Playable Characters)'s functionalities

viii - Test: Map Changing

ix - Test: Quit Window

x - Test: Battle System - Bug's AI and Quiz Window

xi - Test: Game Performance

xii - Test: Collision

xiii - Test: User's positions after Relog.

Because we wanted to test how each component work using a general test case, it is important to note that we want to play from a User's perspective.

i - Test: Login

- We tested the login by registering an account and attempted to use it to log into the game. This test passed showing that the connection between the client to the server, the server to the database, and vice versa, were connected successfully.

ii - Test: Character Movement

- Because the client team needed to code the character's movements from scratch, it is essential to test their correctness. We tested the character's movement by using the keyboard ASDW and mouse clicks. Both worked, so the client listened to and registered the movement inputs correctly.

iii - Test: Camera Movement

- Due to the important of being able to "see" things, it is essential to test the camera movement. We tested whether the camera was too sensitive or not. The final result was that the camera's sensitivity was just right.

iv - Test: Chat Function

- It can be frustrating if you have the viewing part work but not the chatting part. This is why we needed to test the chat functions as well. This was a simple test of whether hitting toggles on the chatting window. It works show one can type a message and display it to others.

v - Test: Chat Modes

- Since there are multiple chat modes, we tested each individually. There are a total of four chat modes.

G – General (Works)

P – Party (In progress)

W – Whisper (Works)

A – All (Works)

All of the chat modes work except for party chat because the party concept is still needed to be implemented.

vi - Test: General Hotkeys

- General Hotkeys are made available to enhance the user's interaction with the game. The current available hotkeys are…

ctrl + i: Toggle inventory list

ctrl + c: Toggle character info window

ctrl + f: Toggle friend list

ctrl + m: Toggle mini map

All tests were passed that they toggle their respective windows correctly.

vii - Test: NPC (Non Playable Characters)'s functionalities

- NPC are what make the game interesting. Currently, there are two type of NPC available in game. One is "Ralph," a general NPC with plenty of information and can warp, and two "Shop," a NPC who sells items. The tests were conducted to see what functionalities they have.

- Ralph – There are still some blank choice selection, acting as place holder for future implementations.

- Shop – Currently, the shop is empty since the items are mostly still under development.

viii - Test: Map Changing

- Changing map is what allow the user to explore the game content. This was tested by entering map changing spots in game, known as portals. All tests were passed, so the portals are stable.

ix - Test: Quit Window

- Having a Quit Window is important because it allows a way for the program to deallocate unused memory. The tests were mainly focused to see whether the Quit Window's functionalities worked correctly as intended. The Quit Window passed the tests and that it worked as it supposed to.

x - Test: Battle System - Bug's AI and Quiz Window

- The battle system consists of Bug's AI and Quiz Window which are the main interaction between the user and the bugs. If the user is within a certain radius, one will be chased after by an aggressive bug. Once engaged, a window will be popped up, prompting the user a quiz question and a set of multiple choices. This window passed the general test. However, it can still be improved like reducing the transparency between the quiz window and its background.

xi - Test: Game Performance

- Game Performance is the number one issue in a game. Nothing can be more frustrated when your program runs slower than your reaction. In our tests, sometimes the game ran smoothly for 8 – 10 players online at once, while having 4-6 bugs loaded. But other times, the game can lag while only 4 players were on. These are some reasons that we thought it might have caused the lag:

  - Server is lagging from time to time.
  - The nature of Panda 3D (doesn't support multithreaded – we're just imitating one)
  - Collision complications

xii - Test: Collision

- Collision is also important, so users can feel the solid objects in the game. Without collision, users will be running through walls and other players. We have tested running the character into various objects. Although most objects were solid, the user's character experienced "half in, half out" situation, which half of the avatar is inside the object.

xiii - Test: User's positions after Relog.

- Being able to continue from where one stopped is very important to keep one's progress, which is why testing the relog function is needed. This tested was conducted to see whether the character's position was saved after logged off. In result, the character's position was saved, so after being relogged, the user can continue on where one stopped.

Game Testing Checklist:

i - Test: Login

ii - Test: Character Movement

iii - Test: Camera Movement

iv - Test: Chat Function

v - Test: Chat Modes

vi - Test: General Hotkeys

vii - Test: NPC (Non Playable Characters)'s functionalities

viii - Test: Map Changing

ix - Test: Quit Window

x - Test: Battle System - Bug's AI and Quiz Window

xi - Test: Game Performance

xii - Test: Collision

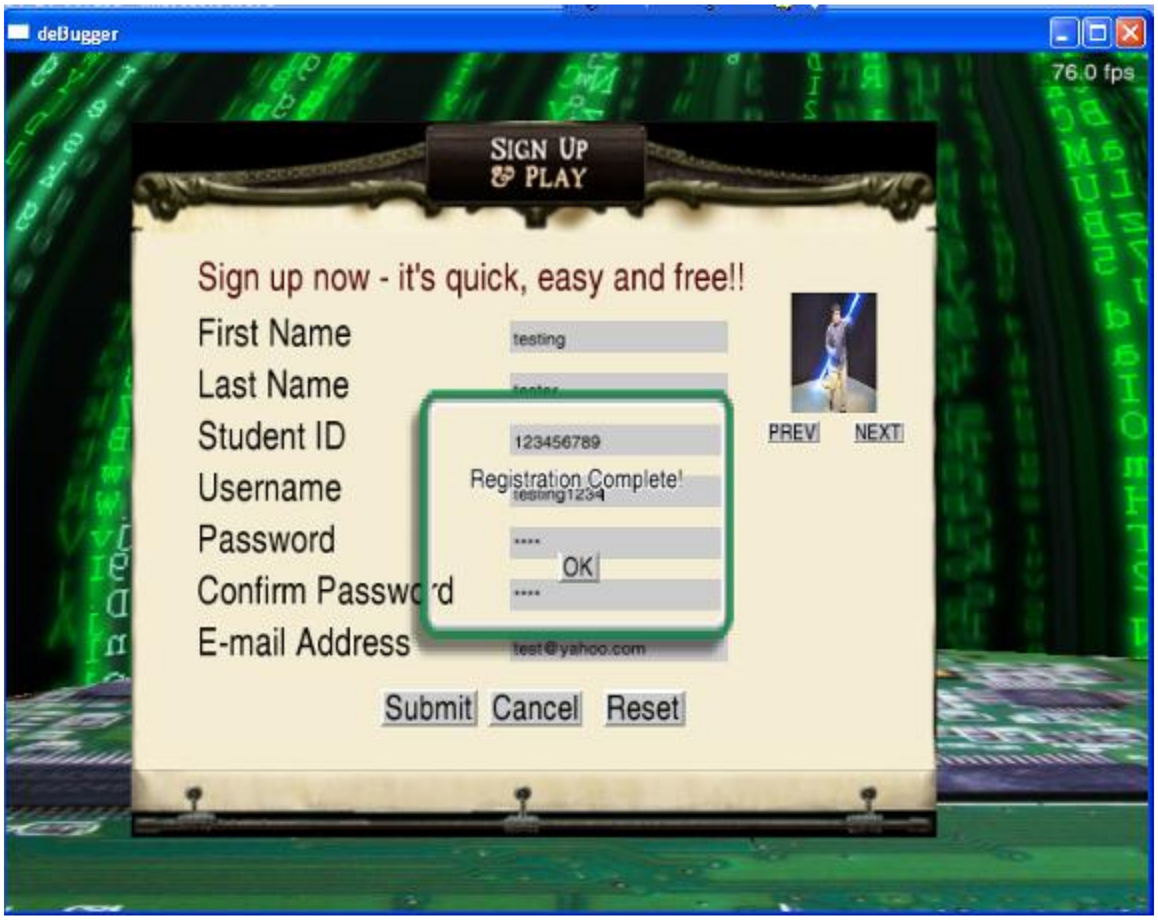
xiii - Test: User's positions after Relog.

i - Test: Login

REGISTER ACCOUNT? TESTING CONNECTION WITH DATABASE.	WORKS.
LOGIN? REQUEST/RESPONSE CONNECTION WITH SERVER?	WORKS.

Analysis:

A series of tests were conducted to analyze the Client's connection with other teams, like the Server and the Database. The first test was conducted to test the connection between the Client and the Database. The test plan was to fill out the application and send them to the Database.



Sign up now - it's quick, easy and free!!

First Name	testing
Last Name	tester
Student ID	123456789
Username	testing1234
Password	****
Confirm Password	****
E-mail Address	test@yahoo.com

Registration Complete!

Figure 1: Show registration.

Then the second test was conducted to confirm whether the previous information was sent correctly or not. The result returns positive signifying the functionalities work.

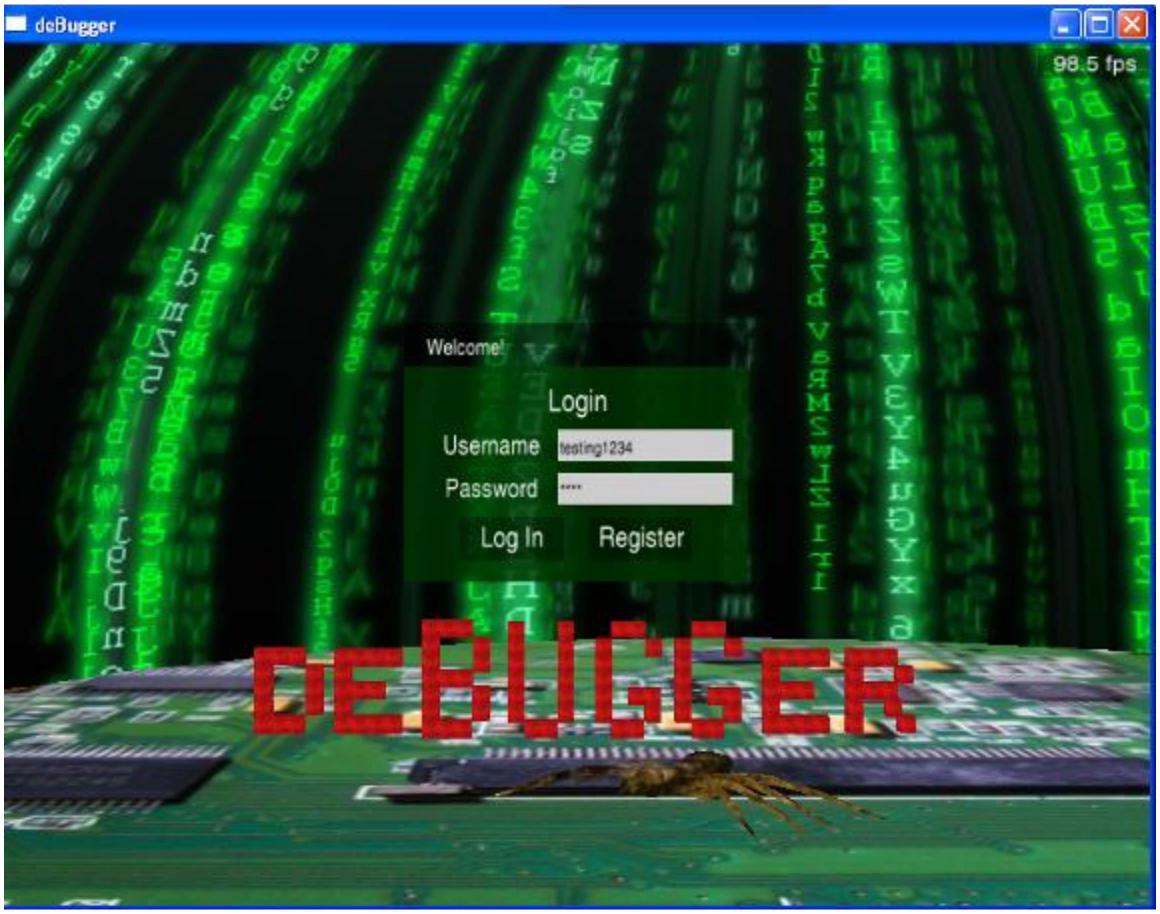


Figure 2: Show login with an actual account.

ii - Test: Character Movement

KEYBOARD: CHARACTER MOVEMENT BY ASDW	WORKS.
MOUSE: CHARACTER MOVEMENT BY POINT CLICK	WORKS.

Analysis:

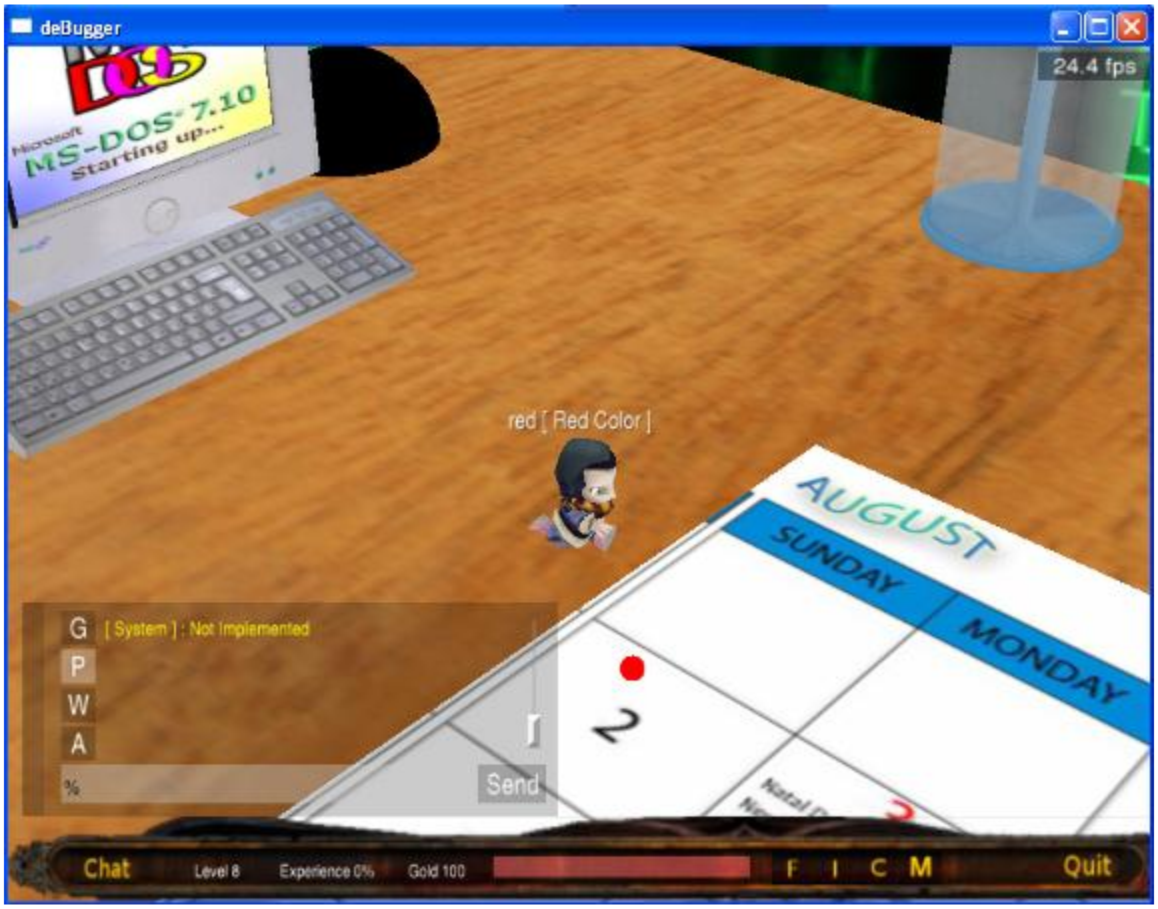
A series of tests were conducted to analyze the user's character movement. Since there were two types of movement control, a separate test was conducted for each one. First, the keyboard keys were tested to see whether they support the character's movement. Second, the mouse left click was tested to see whether the character will respond to this user's command. Both tests passed.

iii - Test: Camera Movement

MOUSE: TURN CAMERA BY HOLDING ONTO RIGHT CLICK.	WORKS.
MOUSE: ZOOM CAMERA BY SCROLLING MOUSE WHEEL.	WORKS.

Analysis:

A series of tests were conducted to analyze the user's camera movement. There were two types of camera movements: 1) turn camera in angle; 2) zoom camera in and out. First, a test was conducted to test whether holding onto the right mouse key and drag left/right changed the user's angles. Second, a test was conducted to test whether scrolling the mouse wheel zoomed in/out of user's camera. Both tests passed.



[System] : Not Implemented

Figure 3: Show mouse starting position.

Held right click…. and….

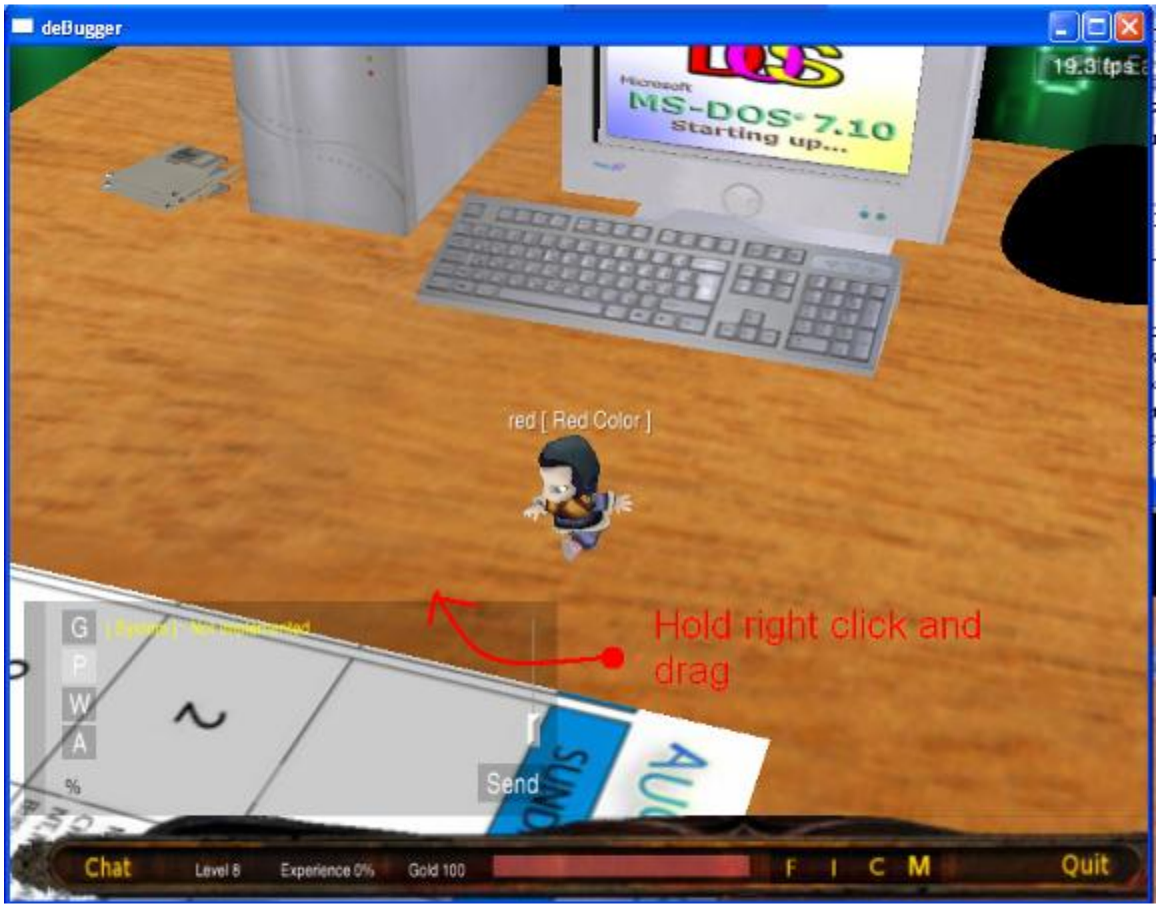


Figure 4: Show mouse ending position.

… dragged left/right to change angles.

iv - Test: Chat Function

HOTKEY: < TAB > SWITCHES TO DIFFERENT CHAT MODE WHILE CHAT BOX IS FOCUSED.	WORKS.
HOTKEY: < ENTER > TOGGLE CHAT FOCUS/UNFOCUS	WORKS.

Analysis:

A series of tests were conducted to analyze some of the hotkeys available for chat. A test was conducted to test the < Enter > hotkey to see whether it will toggle the chat focus or not – a chat focus is like turning a flag on, so the client starts to read/store the user's input string in the chatBox. A second test was conducted to test the < Tab > hotkey to see whether it will switch between the different chat modes available (shown below). Both tests passed.

v - Test: Chat Modes

G: GENERAL MODE (RED COLOR) CHATBOX: ! *FOLLOWEDBYSOMEMSG*	WORKS.
P: PARTY MODE CHATBOX: % *FOLLOWEDBYSOMEMSG*	NOT IMPLEMENTED.
W: WHISPER MODE (BLUE COLOR) CHATBOX: /*TARGETEDPLAYERSNAME* *FOLLOWEDBYSOMEMSG*	WORKS.
A: ALL MODE (WHITE COLOR) CHATBOX: *SOMEMSG*	WORKS.

Analysis:

A series of tests were conducted to analyze the different available chat modes. There are currently four chat modes in game.

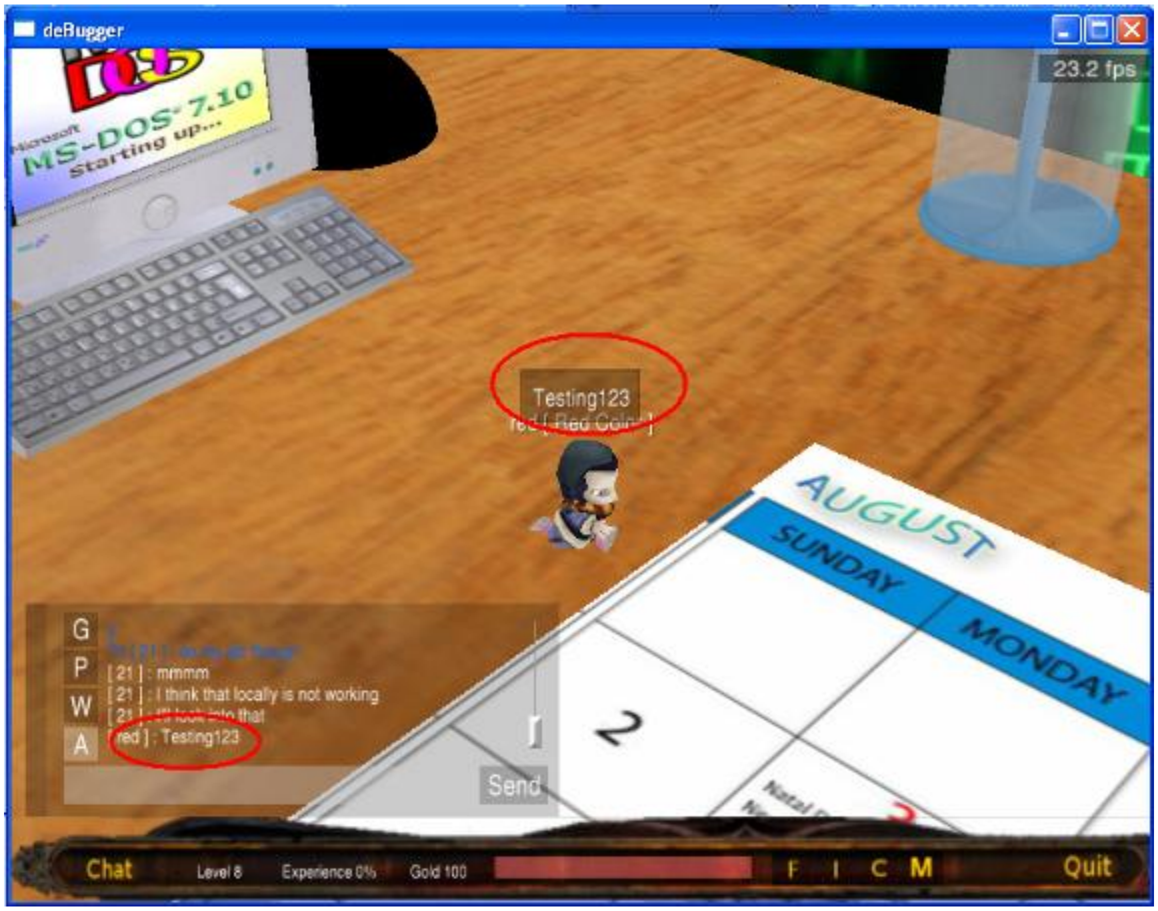
First, a test was conducted to verify whether the General chat mode worked or not. In a test case for General chat mode, chatBox: "! Testing 123," all users read the message "Testing 123." At the same time, the user's chat box changed color from default White to Red indicating the other users that the user is sending a general chat.

Second, a test was conducted to verify whether the Party chat mode worked or not. Since the Party mode is still under development, this Party chat fails the test.

Third, a test was conducted to verify whether the Whisper chat mode worked or not. A test string was given, "/Yellow Testing123." The user named "Yellow" read the string "Testing123." Whispered text turned blue color.

Fourth, a test was conducted to verify whether the default chat mode, All chat, worked or not. A test string was given, "Testing123." Other users that are in the same room saw "Testing123."

All tests, except Party chat mode, passed.



[21] : mmmm
[21] : I think that locally is not working
[21] : I'll look into that
[red] : Testing123

Figure 5: Show text being displayed correctly.

Showed the string “Testing123” was typed and sent to all users locally.

vi - Test: General Hotkeys

< CTRL I >: TOGGLE INVENTORY LIST.	WORKS.
< CTRL C >: TOGGLE CHARACTER INFO WINDOW.	WORKS.
< CTRL F >: TOGGLE FRIEND LIST.	WORKS.
< CTRL M >: TOGGLE MINI MAP.	WORKS.

Analysis:

A series of tests were conducted to analyze hotkeys which toggle certain GUI in the User’s Client.

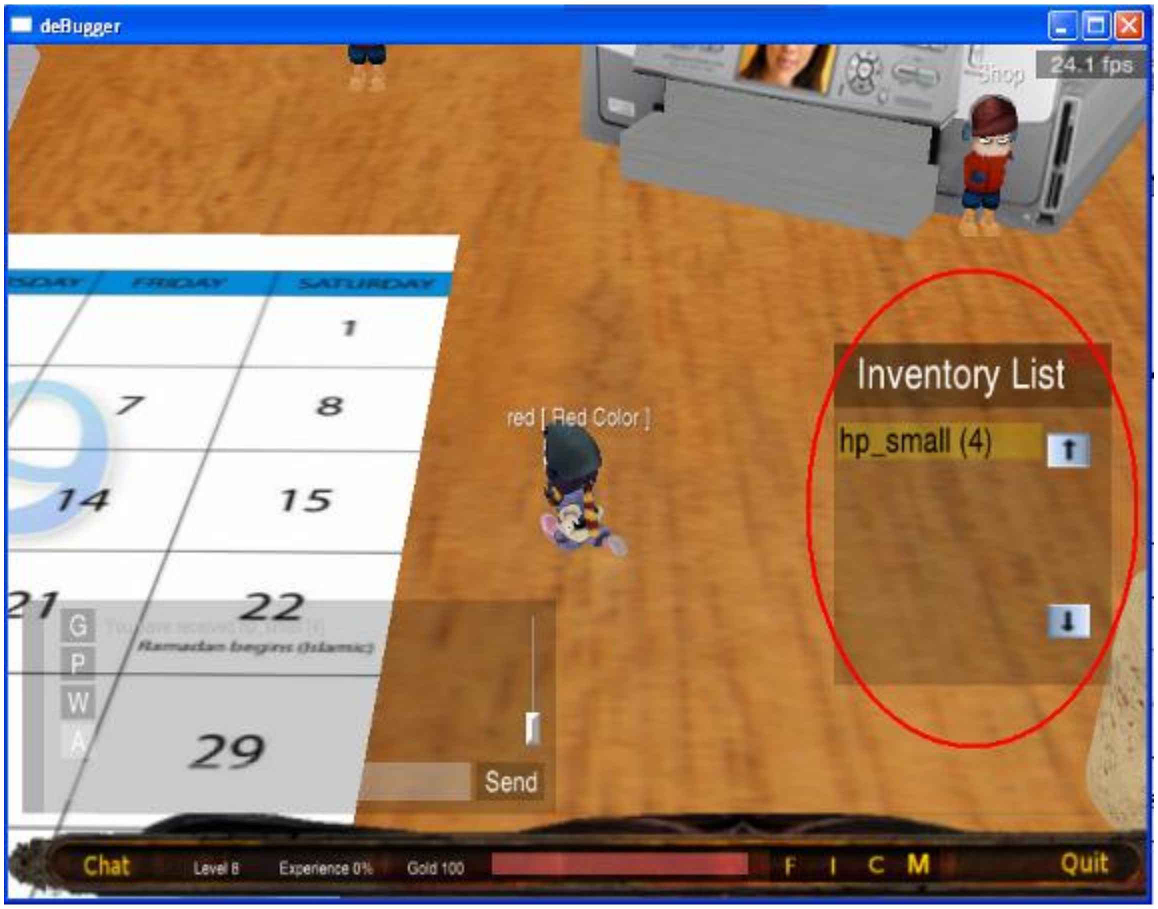
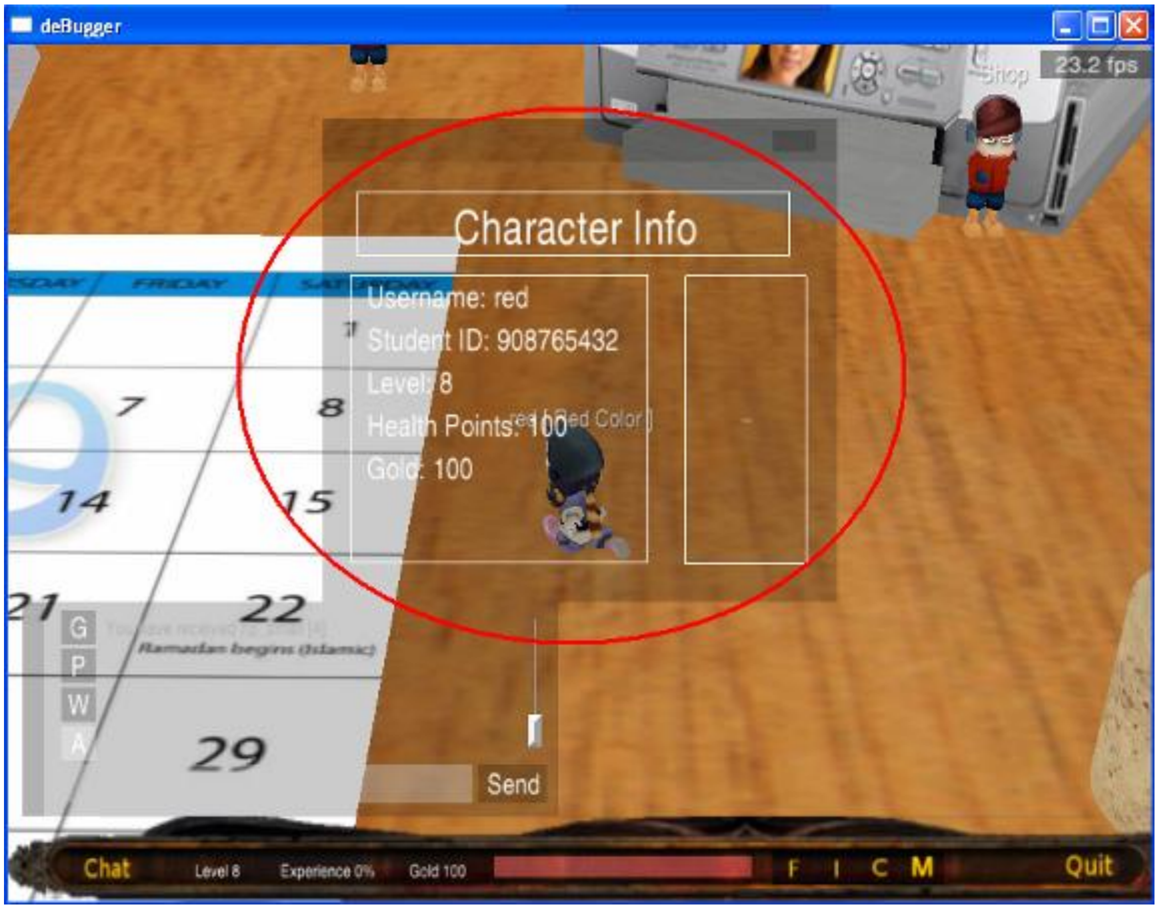


Figure 6: Show the inventory menu after pressing ctrl + i keys.

Holding the < ctrl and i > keys, the user's inventory list was toggled (shown above).


Character Info

Username: red

Student ID: 908765432

Level: 8

Health Points: 100

Gold: 100

Figure 7: Show the character menu after pressing ctrl + c keys.

Holding the < ctrl and c > keys, the user’s character info was toggled (shown above).

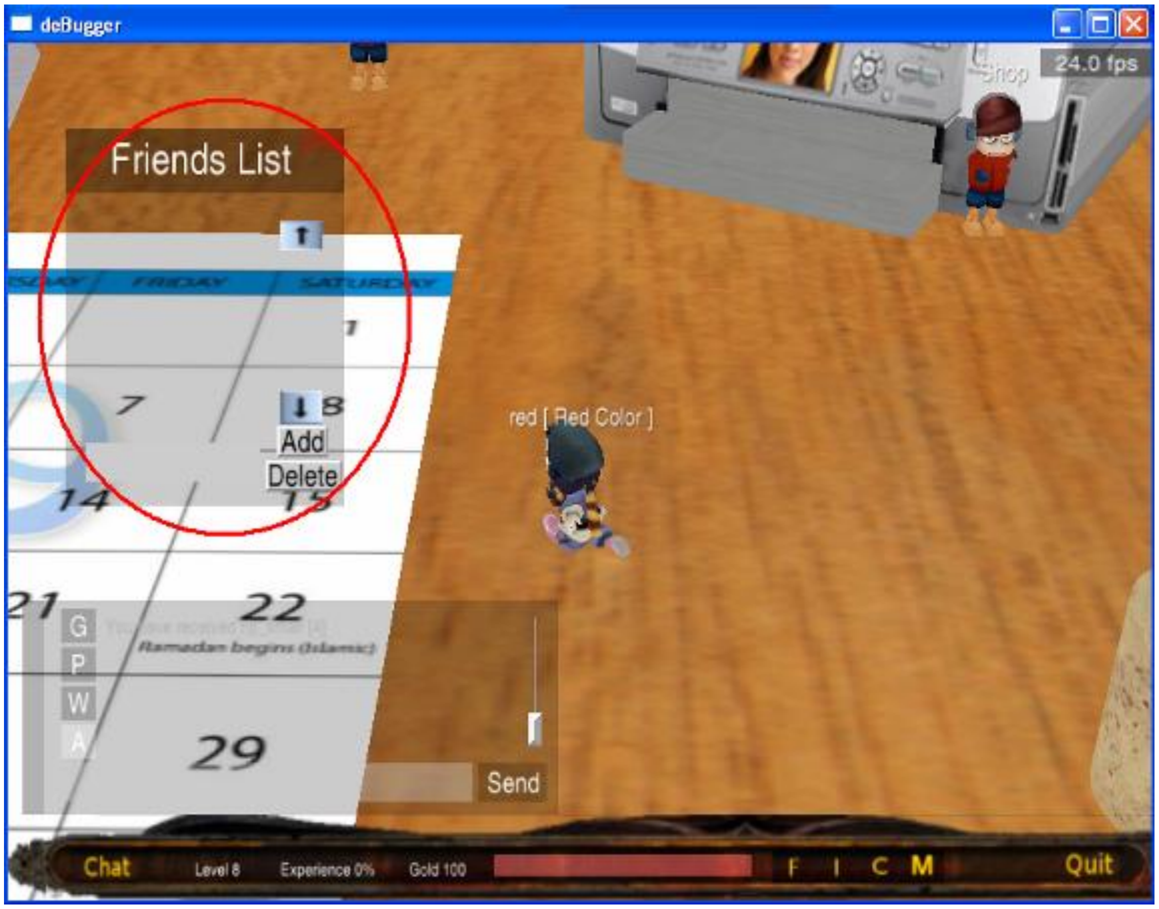


Figure 8: Show the friends list after pressing ctrl + f keys.

Holding the < ctrl and f > keys, the user's friend list was toggled (shown above).

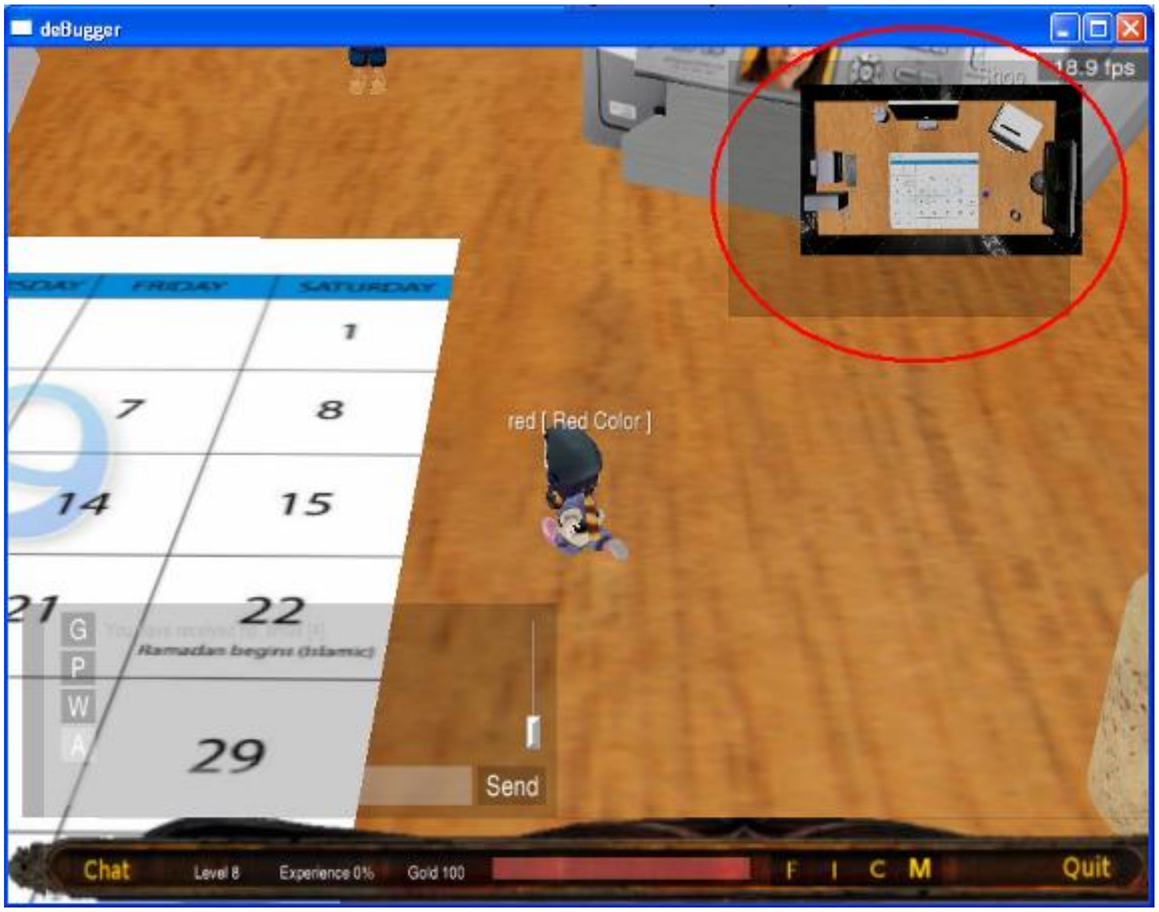


Figure 9: Show the mini map after pressing ctrl + m keys.

Holding the < ctrl and m > keys, the minimap was toggled (shown above).

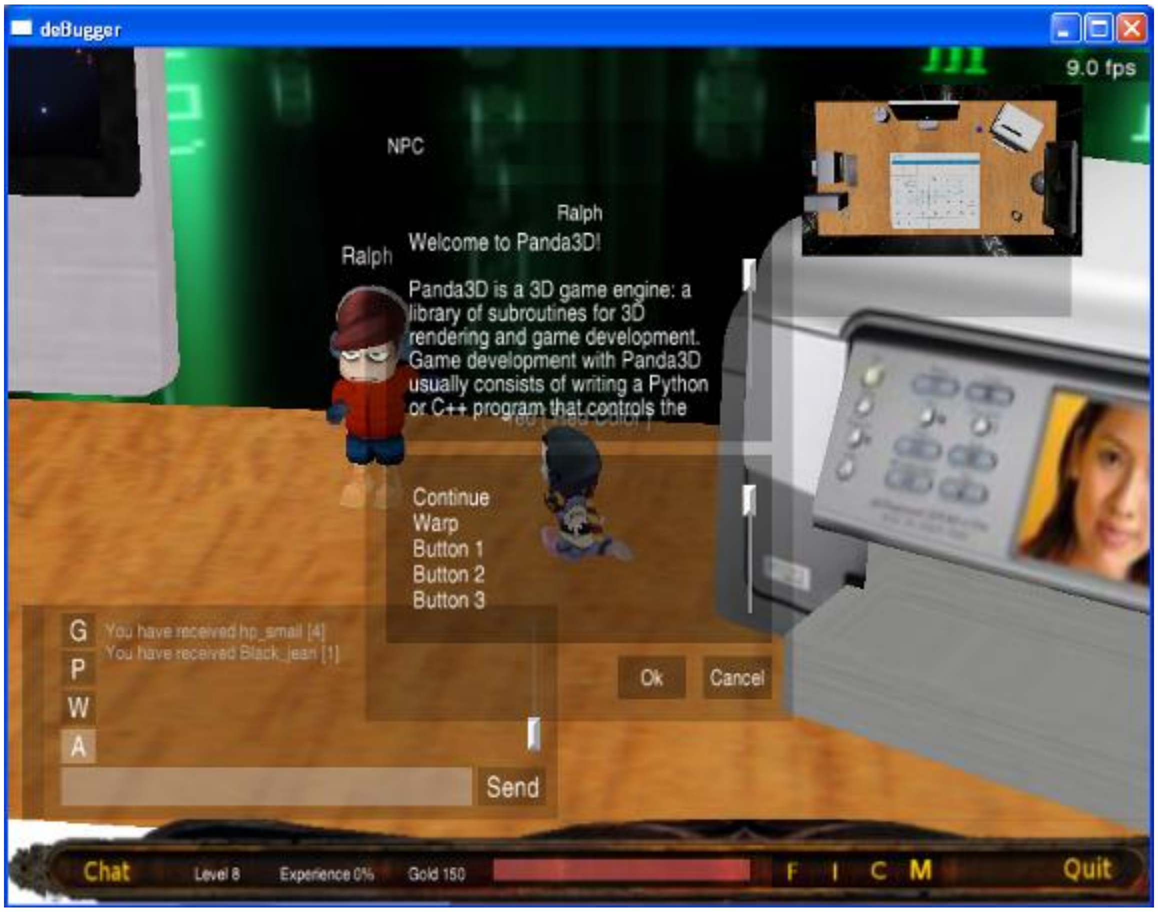
vii - Test: NPC (Non Playable Characters)'s functionalities

RALPH (LOBBY): AN NPC WHO GIVES VARIOUS INFORMATION ABOUT THE GAME AND ALLOW WARPING TO A DIFFERENT MAP.	IN DEVELOPMENT.
SHOP (LOBBY): AN NPC WHO SELLS	IN DEVELOPMENT.

ITEMS.	

Analysis:

A series of tests were conducted to analyze the available NPCs and their functionalities. We had two NPCs, one was Ralph, a general NPC, and another was Shop, a NPC who sells items.



Ralph

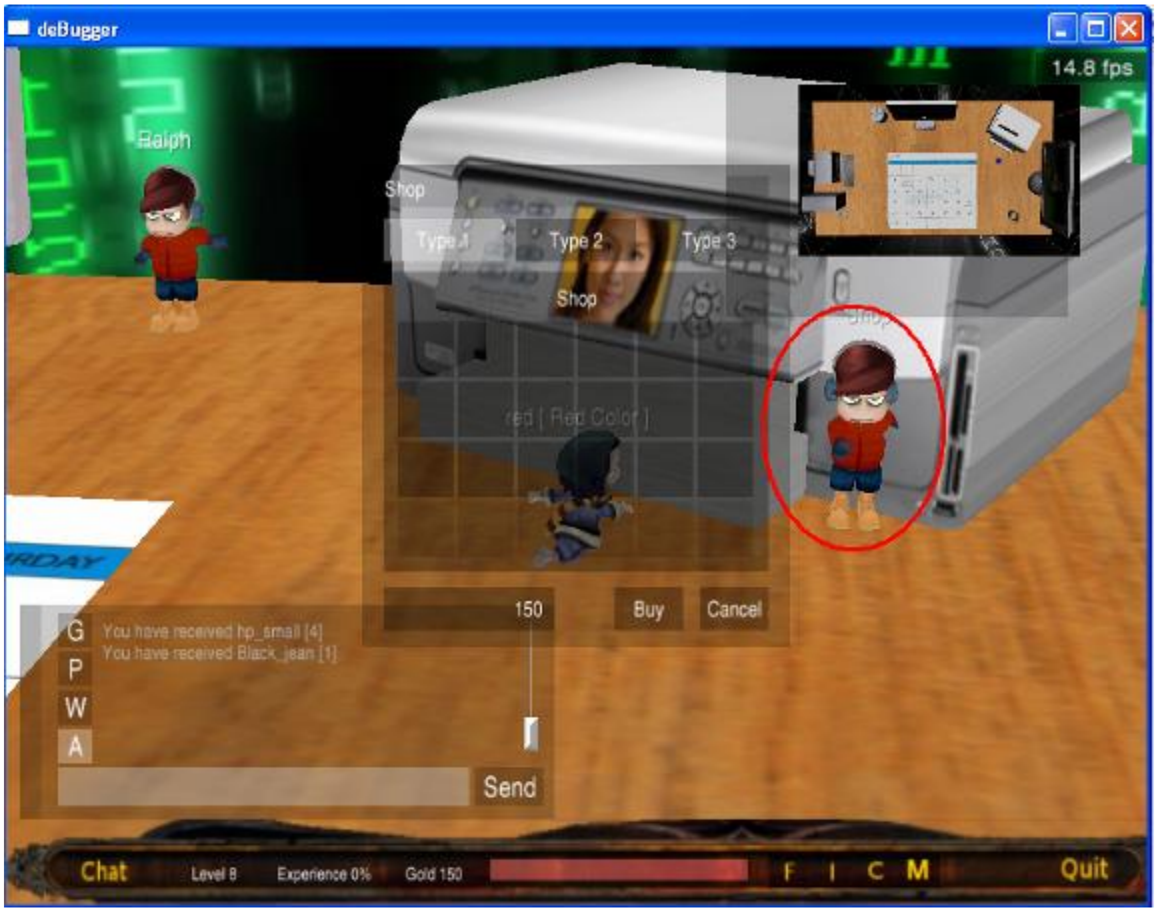
Welcome to Panda3D!

Panda3D is a 3D game engine: a library of subroutines for 3D rendering and game development. Game development with Panda3D usually consists of writing a Python or C++ program that controls the

You have received hp_small [4]
You have received Black_jean [1]

Figure 10: Show NPC Ralph interaction.

First, a test was conducted to see what commands were working for Ralph. Some commands worked, while others acted as placeholders for future implementations. Some working commands included: Continue, Warp, and End. Commands that were in development included Button 1, Button 2,…, Button 4.



You have received hp_small [4]
You have received Black_jean [1]

Figure 11: Show NPC Shop interaction.

Second, a test was conducted to see what functionalities were working for Shop NPC. The tabs which separate the level of available items, Type1 … Type3, worked. The lower left corner correctly showed the amount of gold the User currently had. However, items were not yet available for further testing, like drag and click to buy items, or double click items to buy, or selling items, etc. The Shop NPC was still in development.

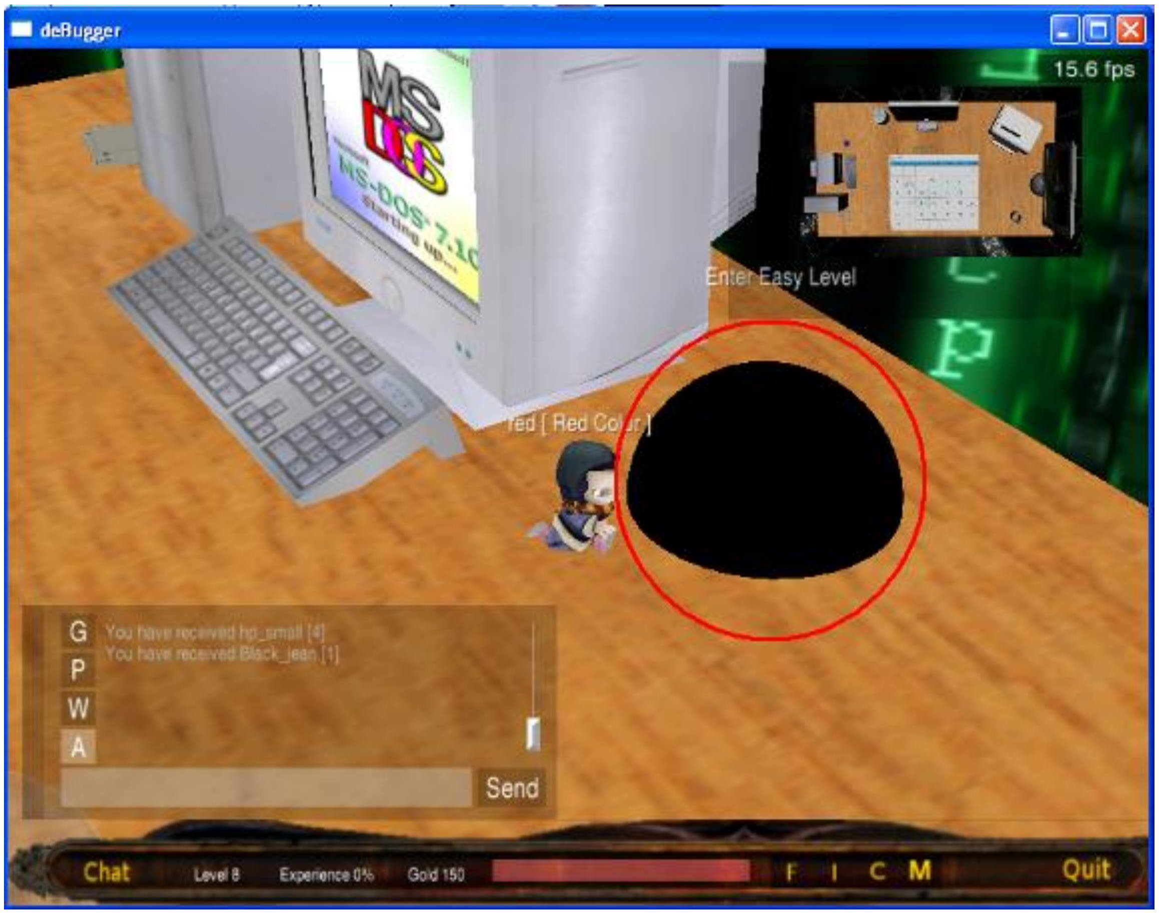
viii - Test: Map Changing

PORTAL (LOBBY): ENTER EASY LEVEL	WORKS.

PORTAL (EASY LEVEL): BACK TO LOBBY	WORKS.
NPC COMMAND (LOBBY): WARP TO BATTLEGROUND.	WORKS.
PORTAL (BATTLEGROUND): BACK TO LOBBY.	WORKS.

Analysis:

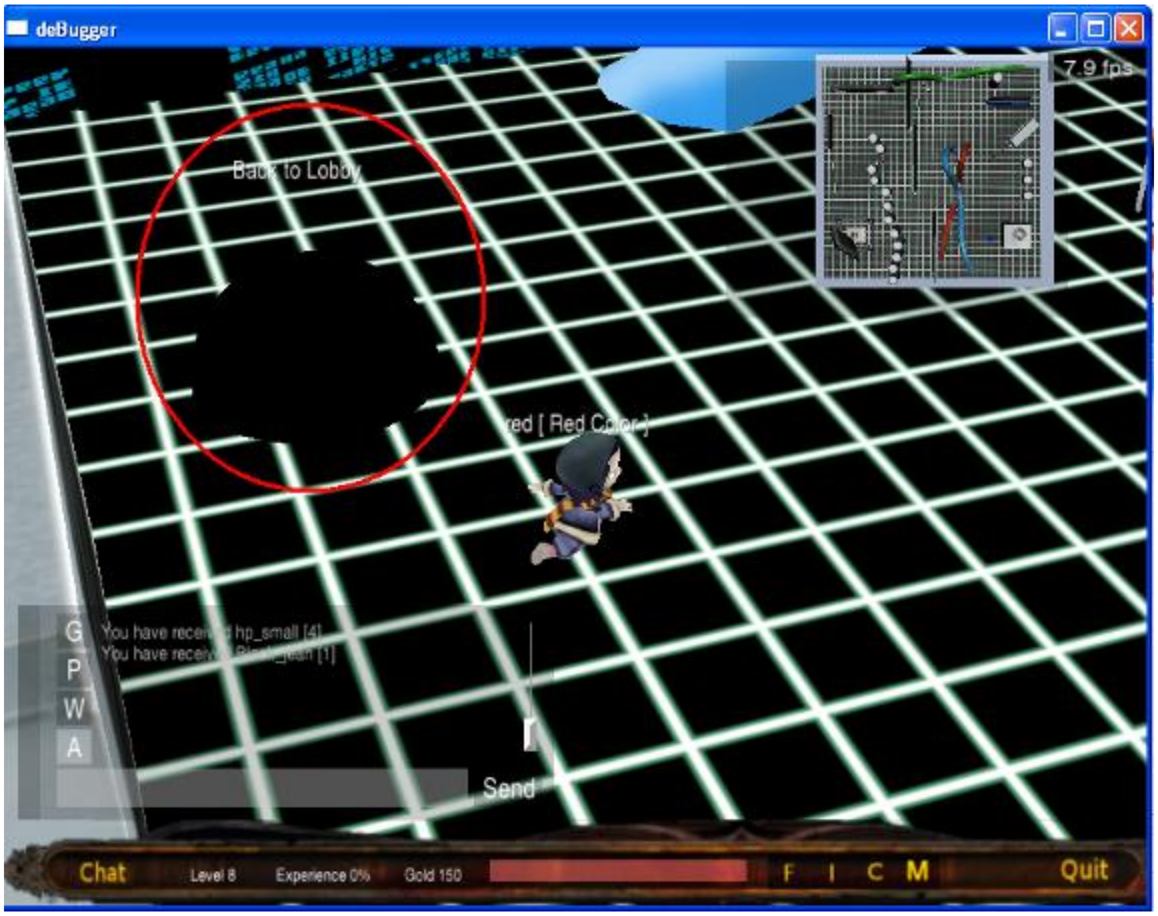
A series of tests were conducted to analyze the available warps in game from one map to another. This was done by moving the User's character into these black looking spheres or by using the NPC's warp command, shown below.



You have received hp_small [4]
You have received Black_jean [1]

Figure 12: Show changing map with portal entrance.

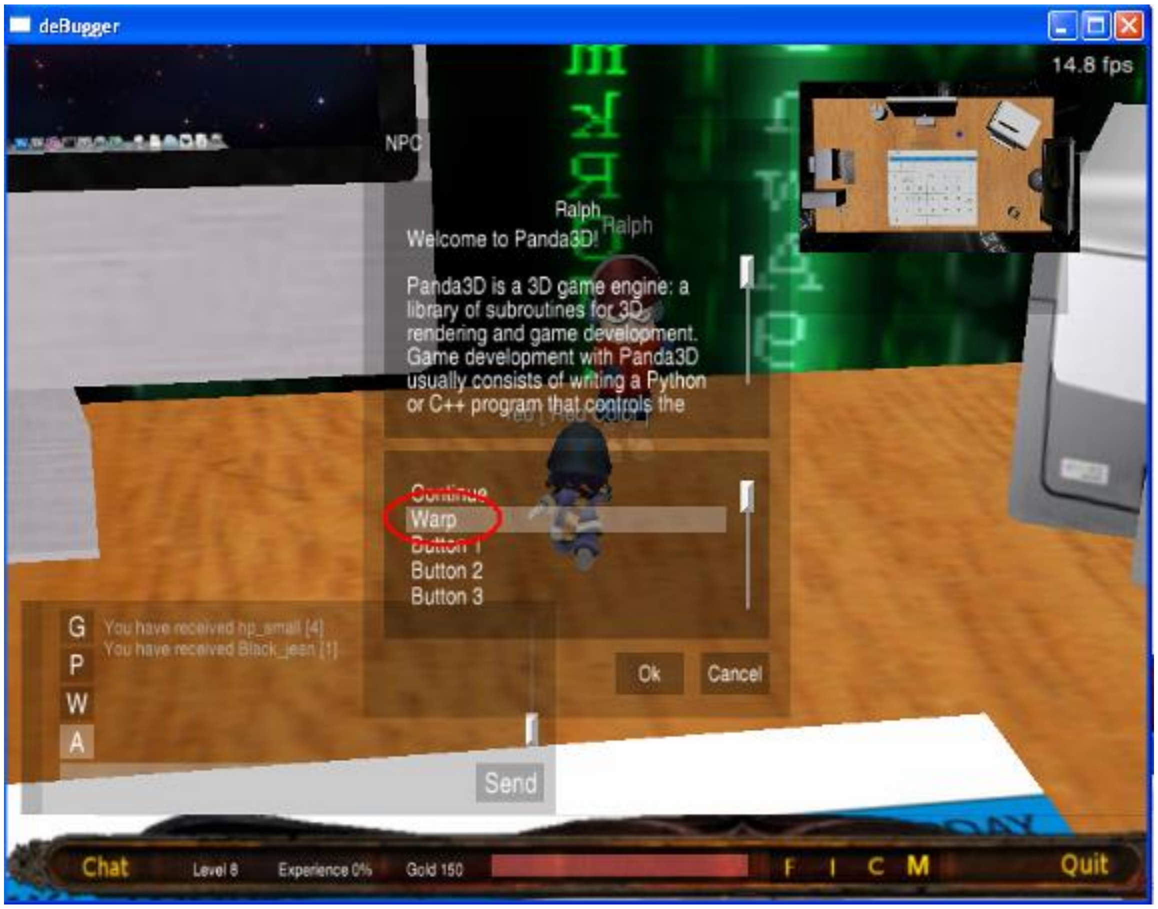
First, a test was conducted to change map from the Lobby --> Easy Level (shown above).



You have received hp_small [4]
You have received Blast_gean [1]

Figure 13: Show changing map with portal exit.

Second, a test was conducted to change map from Easy Level --> Lobby.



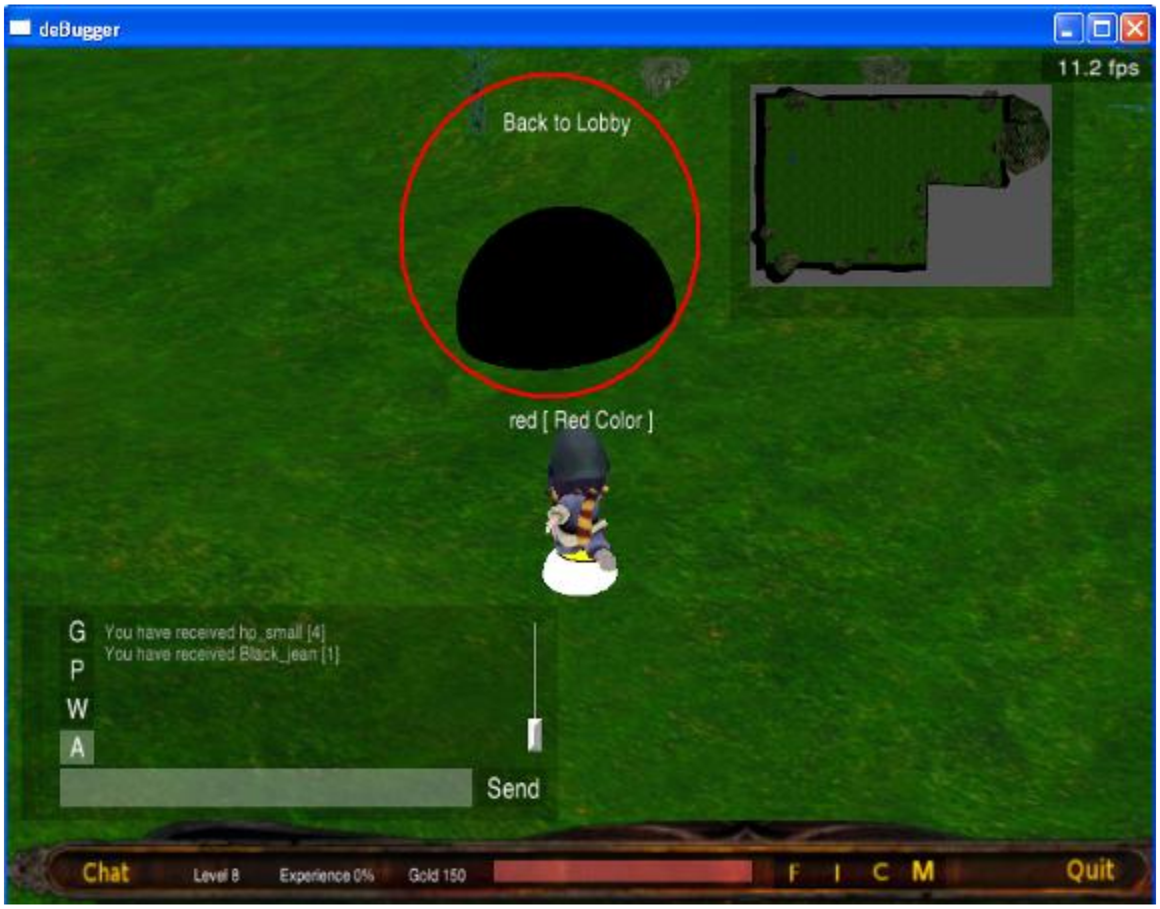
Welcome to Panda3D!

Panda3D is a 3D game engine: a library of subroutines for 3D rendering and game development. Game development with Panda3D usually consists of writing a Python or C++ program that controls the

You have received hp_small [4]
You have received Black_jean [1]

Figure 14: Show changing map with NPC's Warp command.

Third, a test was conducted to change map from NPC warp (Lobby) --> Battleground.



You have received hp_small [4]
You have received Black_jean [1]

Figure 15: Show changing map with Back to Lobby portal.

Fourth, a test was conducted to change map from Battlegound --> Lobby.

ix - Test: Quit Window

QUIT COMMAND (RESUME): GO BACK INTO THE GAME AND CONTINUE PLAYING.	WORKS.

QUIT COMMAND (RECONNECT): GO BACK INTO LOGIN PAGE. THE USER CAN THEN CHOOSE TO RELOG, CHANGE ACCOUNT, OR REGISTER FOR NEW ACCOUNT.	WORKS.
QUIT COMMAND (QUIT GAME): CLOSE THE GAME CLIENT AND EXIT GRACEFULLY.	WORKS.

Analysis:

A series of tests were conducted to analyze the available commands in the Quit Window.

First, a test was conducted to see whether Resume command worked or not. It worked like “cancel Quit Window” and returned back into the game.

Second, a test was conducted to see whether the Reconnect command worked or not. It disconnected the user from the current game session and return back to the login screen.

Third, a test was conducted to see whether the Quit Game command worked or not. It disconnected the user from the current game session and closed the client.

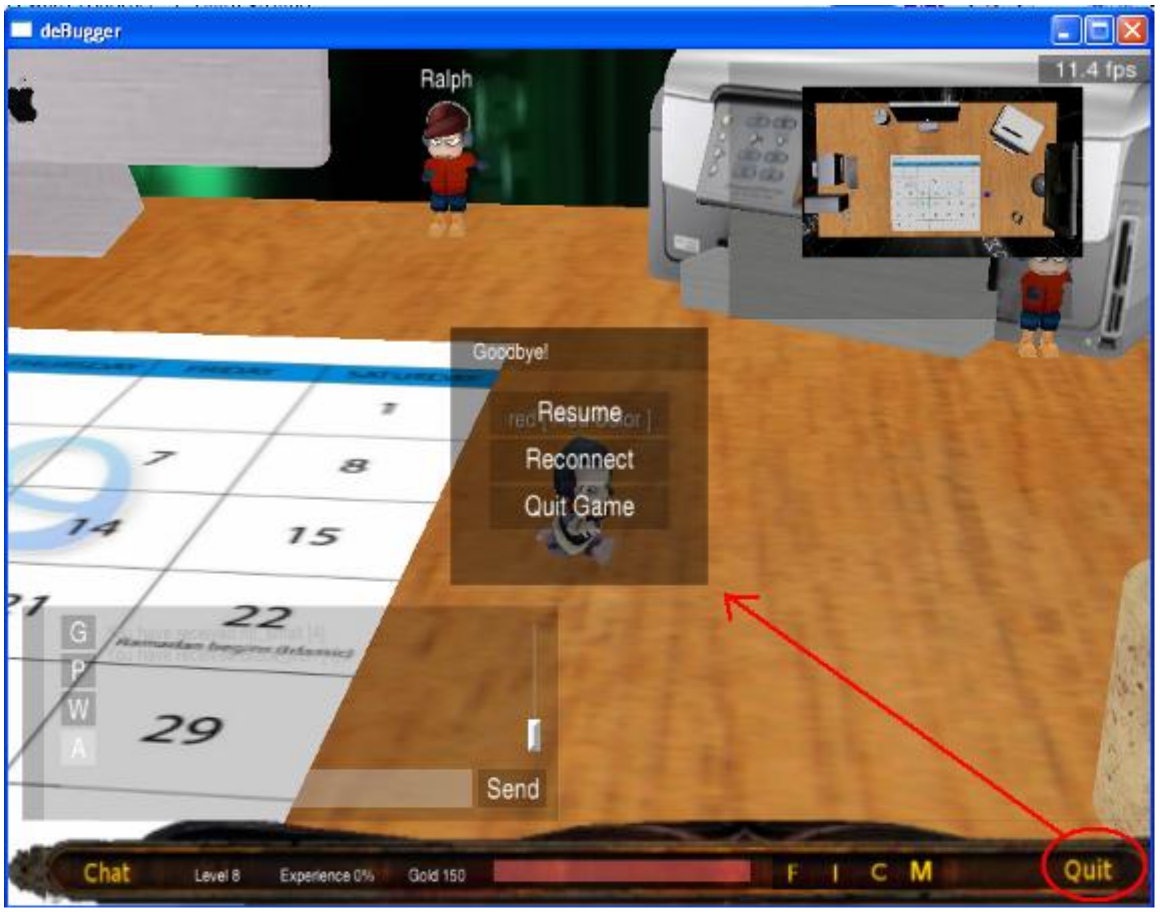


Figure 16: Show Quit Window and its choices.

x - Test: Battle System - Bug's AI and Quiz Window

BUGS AI (RANDOM MOVEMENT): BUG SHOULD MOVE RANDOMLY EVERY SO OFTEN.	WORKS.
BUGS AI (ENGAGE PLAYERS WITHIN RADIUS) – BUG SHOULD CHASE AFTER USER IF THE USER IS WITHIN A DEFINED RADIUS	WORKS.

BUGS FUNCTIONALITY (POP QUIZ WINDOW): IT SHOULD DISPLAY A QUIZ WINDOW.	WORKS.
QUIZ WINDOW (FORMAT): IT SHOULD DISPLAY A PROGRAMMING RELATED QUESTION AND MULTIPLE CHOICES.	WORKS.
QUIZ WINDOW (FUNCTIONALITY): ANSWERING: USER WILL TAKE DAMAGE IF SELECTED WRONG ANSWER. BUG WILL TAKE DAMAGE IF SELECTED CORRECT ANSWER.	WORKS.
STATUS (HP DEPLETION): WHENEVER THE USER CHOOSES A WRONG CHOICE, THE USER SHOULD TAKE DAMAGE, WHICH SHOULD BE INDICATED WITH A RED BAR. THIS SHOULD BE THE SAME FOR BUG'S HP AS WELL.	IN DEVELOPMENT: WORKS ONLY BEFORE 1ST DEATH. HP DOES NOT RECOVERY AFTER THAT, SO USER REVIVES WITH 0 HP.
STATUS (HP REACHES 0): IF THE USER'S HP REACHES 0, THE USER'S CHARACTER SHOULD DIE. THIS SHOULD BE THE SAME WHEN BUG'S HP REACHED 0.	WORKS.

Analysis:

A series of tests were conducted to analyze the Bug's AI and the Quiz Window's functionalities.

To study the Bug's AI, a series of tests were conducted to test Bug's aggressiveness. The character was moved to a radius, and the bug engaged the user. Then a Quiz Window displayed indicating that the User and the Bug were engaged in a battle.

In the Quiz Window, a programming related question was displayed. As the User answer incorrectly or passed a certain time frame without answering, the user took damage. Once the user lost all the character's health points (Hp), they user's character died. All these conducted tests are shown below.

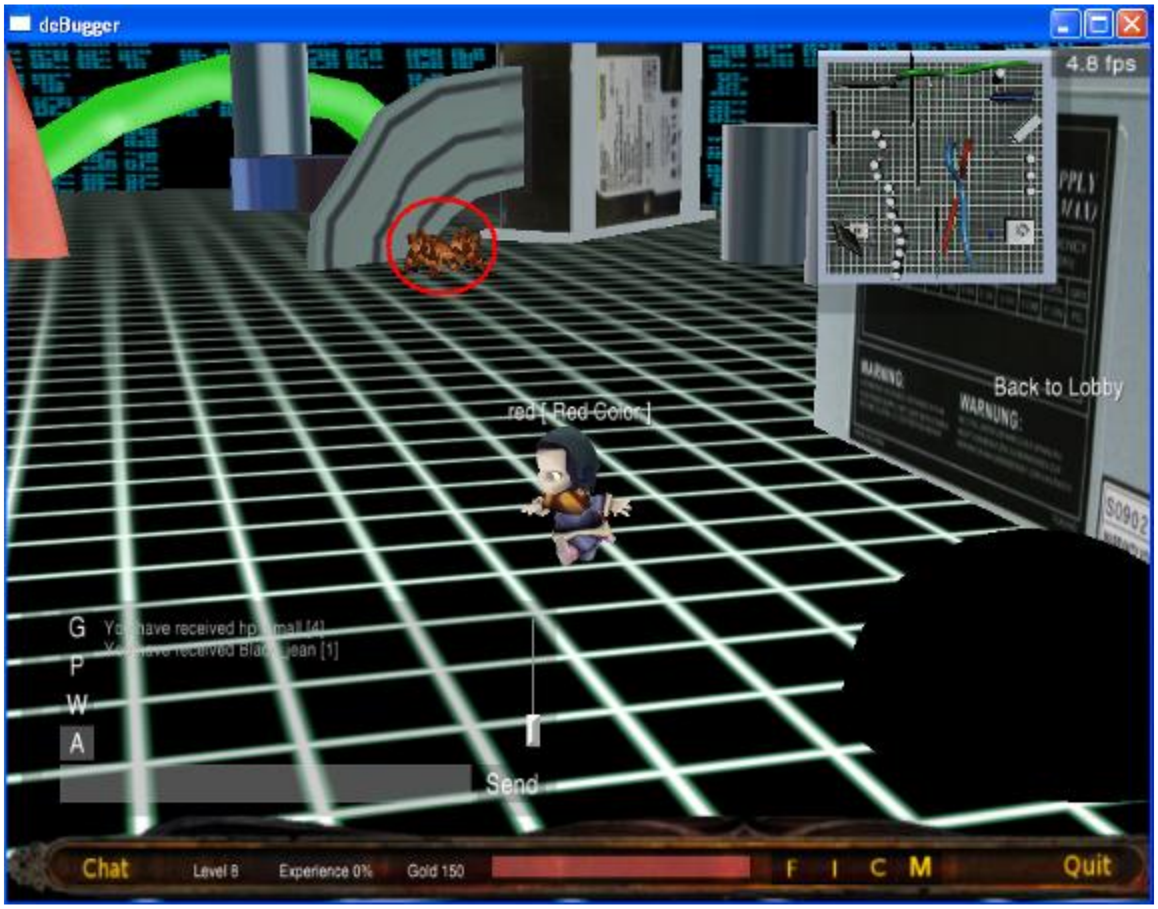
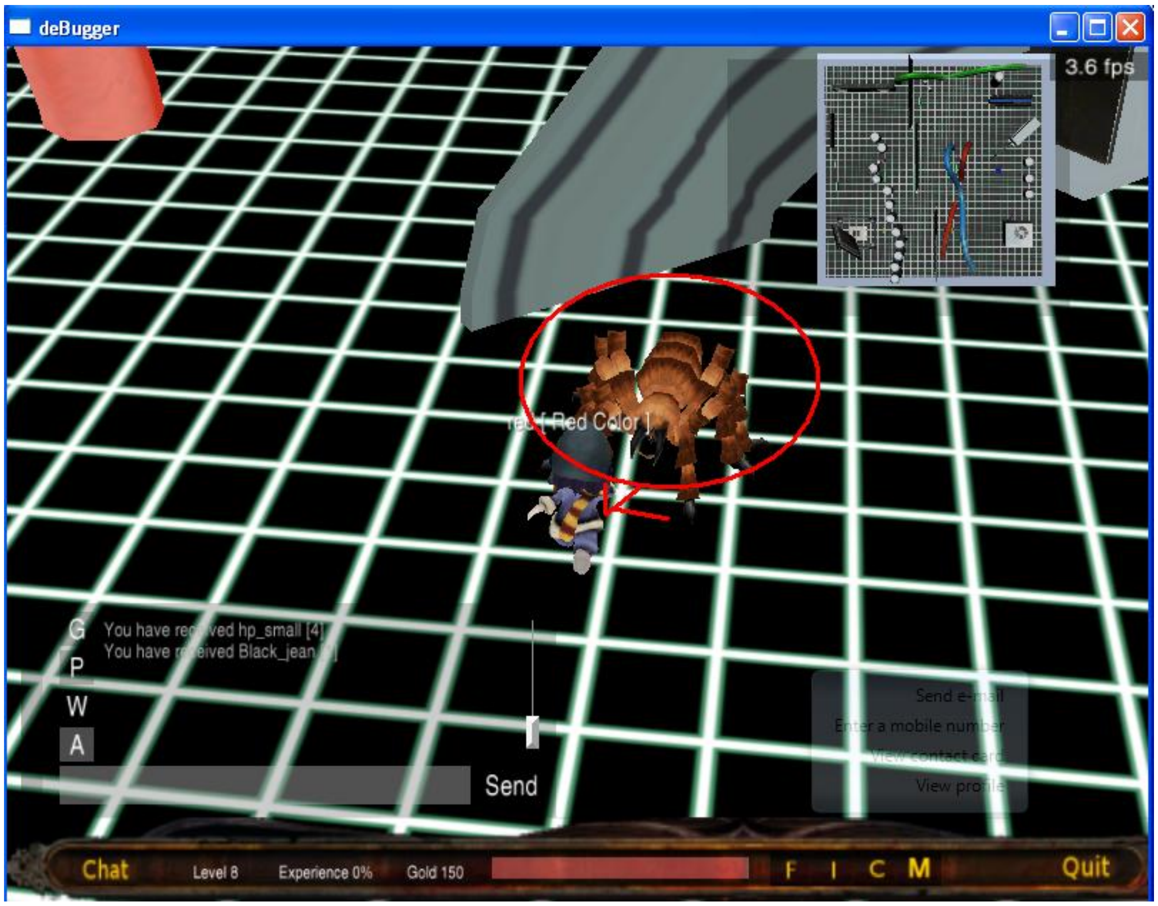


Figure 17: Show bug's random AI.

First, a test was conducted to analyze the Bug's random movement. It wondered around in a random motion.



You have received hp_small [4]
You have received Black_jean [1]

Figure 18: Show bug's aggressiveness.

Second, a test was conducted to analyze the Bug's aggressiveness. Once the user entered the Bug's radius, it chased the user's character.

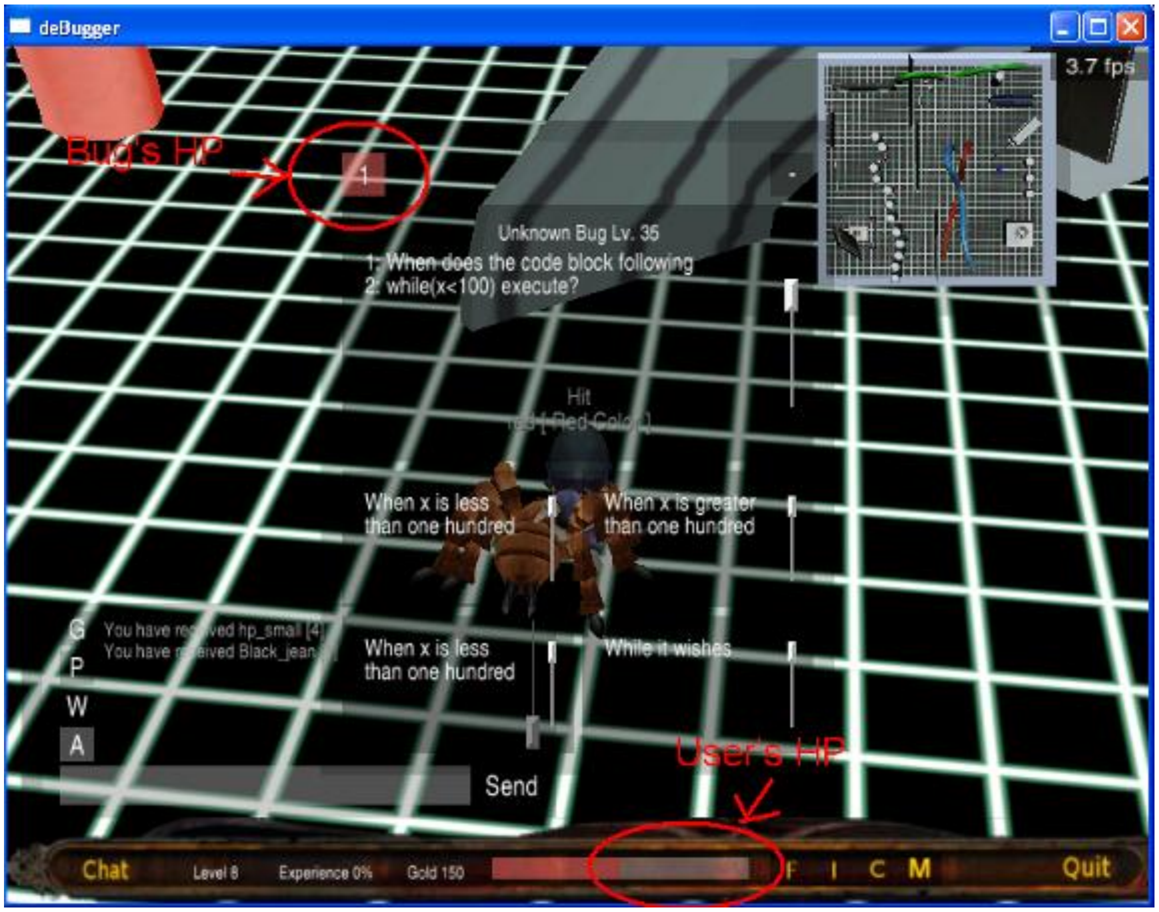
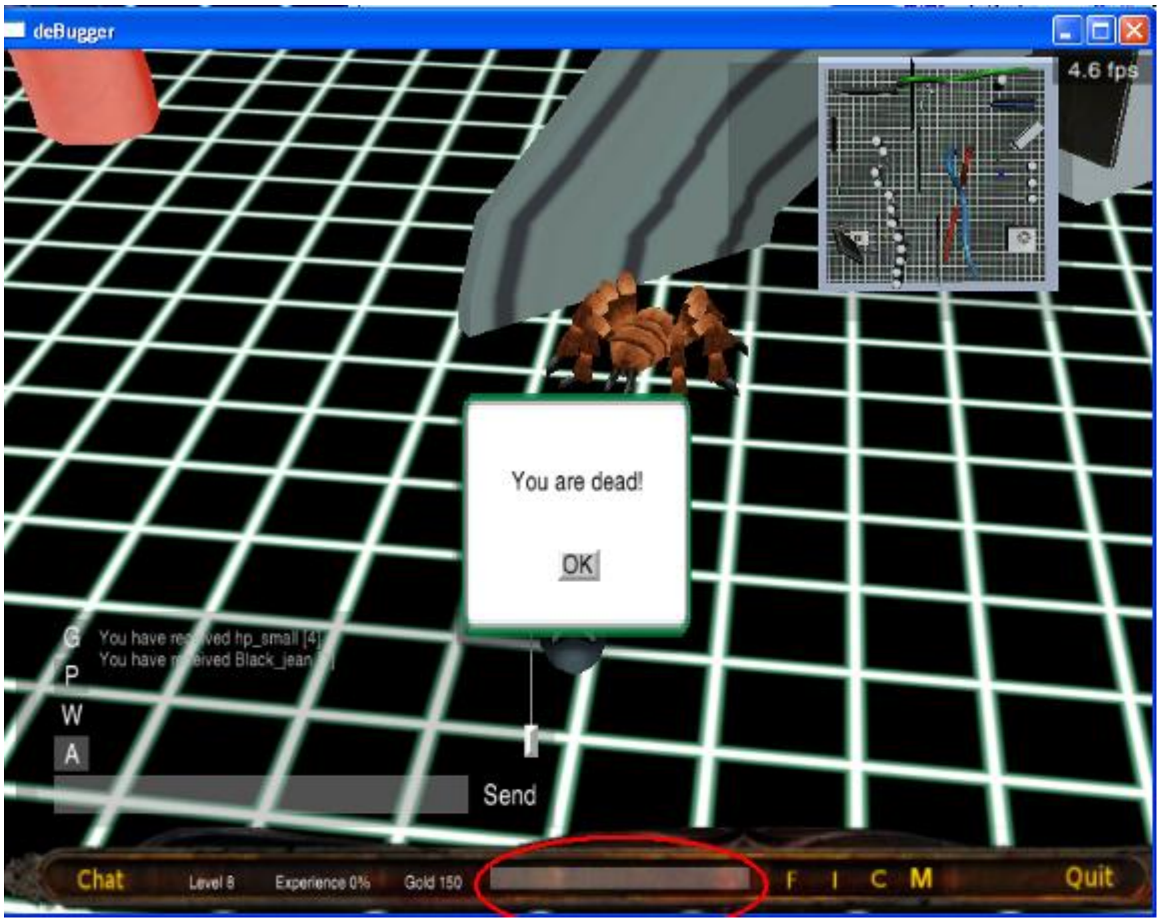


Figure 19: Show bug's and User's hp.

Third, a test was conducted to check whether the Quiz window displayed correctly or not. A programming related question was displayed. Multiple choices were displayed. Bug's HP was displayed.

Fourth, a test was conducted to analyze the Bug's battle. As the user selected a wrong answer or simply waited and passed a certain available time frame, the user's character received damage (shown above).



You are dead!

```
You have re[illegible]ved hp_small [4]
You have r[illegible]eived Black_jean [illegible]
```

Figure 20: Show User's death after depleting all HP.

Fifth, a test was conducted to analyze what happened after the user's character depleted all its HP. The user ended up dead with a message window popped up saying "You are dead!"

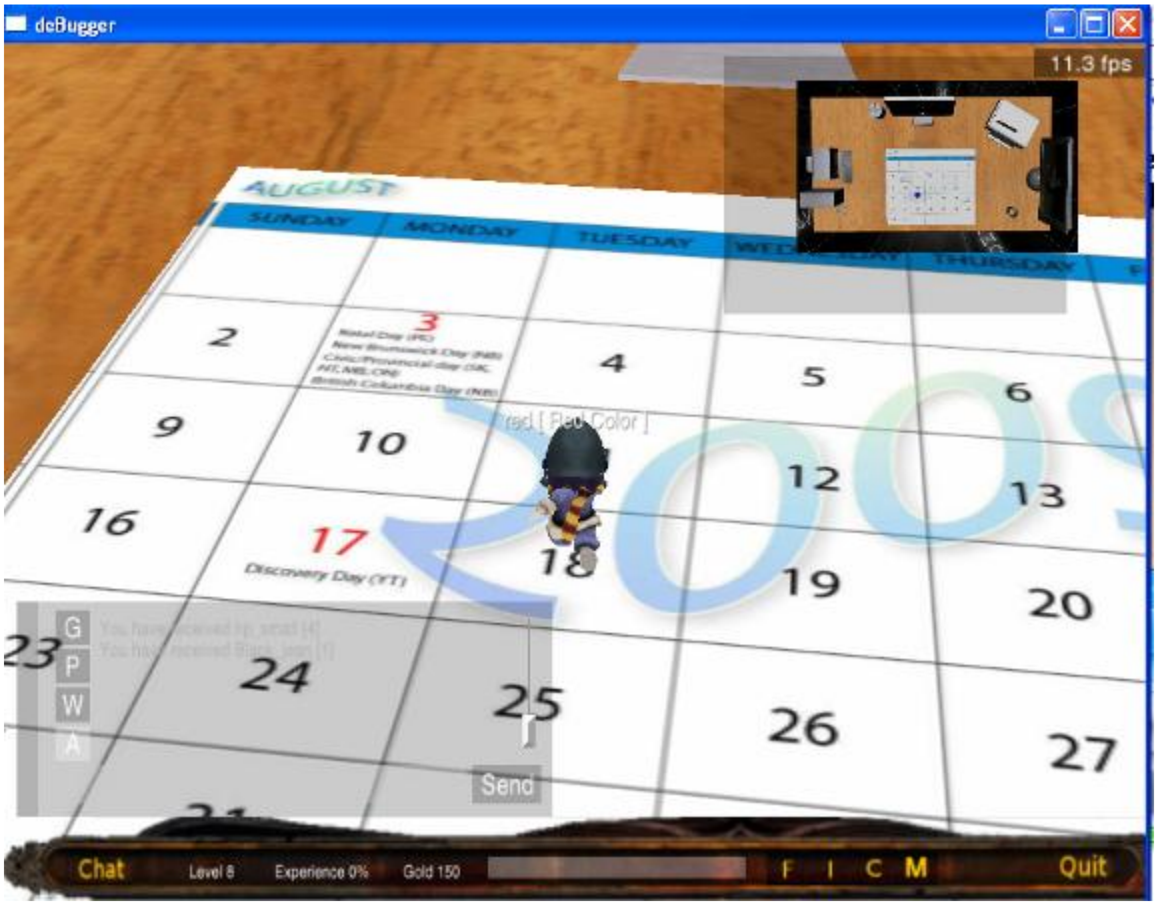


Figure 21: Show User's re-spawn point.

Sixth, a test was conducted to note what happened after the "OK" button was clicked upon death. The user's character spawned back into the lobby, but the HP did not recover. Therefore, the character's HP is still in development.

xi - Test: Game Performance

LOBBY (LAG): MODERATELY	NEEDS IMPROVEMENT.
EASY LEVEL (LAG): HEAVILY	NEEDS IMPROVEMENT.

BATTLE GROUND (LAG): MODERATELY	NEEDS IMPROVEMENT.

Analysis:

A series of tests were conducted to analyze the game's performance, aka "Lag issues."

First, the lobby was tested and the lag was moderate. The user can move the character freely except for occasional lags.

Second, the Easy Level, however, lagged quite a bit due to the heavy graphic in the environment. The user experienced occasional lags. Also, even though there were two bugs, each of them experienced extreme lag. The bugs acted very choppy, and it took ~10 seconds for the bug to detect a user's character was nearby and engaged. The Quiz Window took a good few seconds to pop up after being engaged.

Third, the Battle Ground was plain so it just suffered from the general moderate lag in game.

The game performance definitely needed improvement.

xii - Test: Collision

USER'S CHARACTER WITH BUGS: SHOULD POP A QUIZ WINDOW	WORKS.
USER'S CHARACTER WITH PORTALS: SHOULD WARP TO NEXT MAP	WORKS.
USER'S CHARACTER WITH ENVIRONMENT: SHOULD STOP	SEMI-WORKS. CHARACTERS CAN BE "HALF IN, HALF OUT."

BEFORE ENVIRONMENT, AS IF THE ENVIRONMENT IS SOLID	

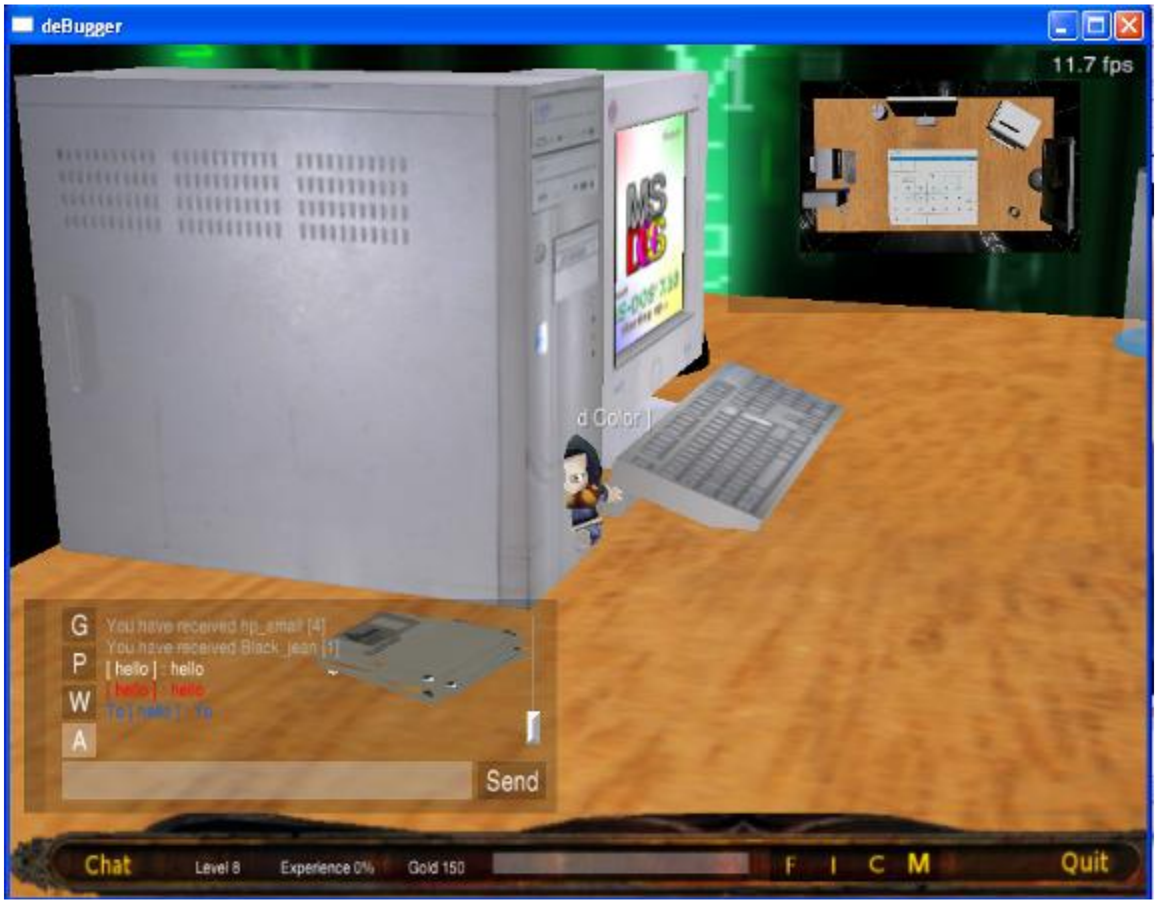
Analysis:

A series of tests were conducted to analyze the collision.

First, a series of tests were conducted to note the collision between user's character and the bugs. Once engaged, Quiz Window popped up. This was shown in the "Test: Battle System - Bug's AI and Quiz Window."

Second, a series of tests were conducted to analyze the collision between the user's character and the portals. Once the user entered a portal, the user will be teleported to another map. This was shown in the "Test: Map Changing."

Third, a series of tests were conducted to see how "solid" the environment was. A character was moved to various objects to test the solid-ness of objects. In general, objects in the surrounding were solid enough for characters to not go through. However, a character was still experience "half in, half out" situations. So objects can be described to be semi-solid.



You have received hp_small [4]
You have received Black_jean [1]
[hello] : hello
[hello] : hello
To [hello] : Yo

Figure 22: Show character experience semi-collision, being "half in, half out."

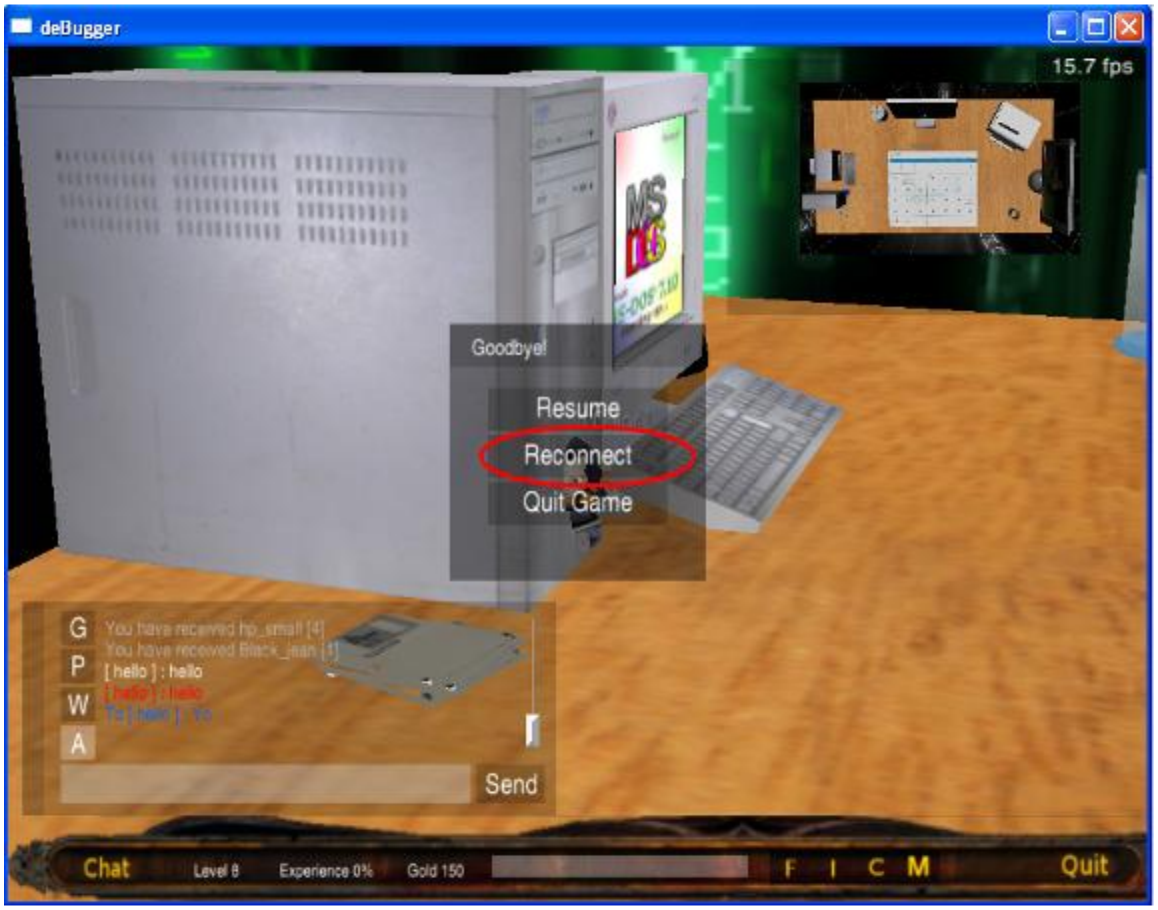
The image above showed a character suffering "half in, half out" with an object computer.

xiii - Test: User's positions after Relog

POSITION SAVED: SHOULD SAVE THE USER'S CHARACTER'S POSITION AND RELOG AT THE SAME SPOT.	WORKS.

Analysis:

A series of tests were conducted to analyze the relog game function. First, the user found a nice spot to relog. The user then selected Reconnect command from the Quit Window. The user relogged, and the character relogged at the same location. So the character's position was saved correctly.



You have received hp_small [4]
You have received Black_jean [1]
[hello] : hello

Figure 23: Show the character's position (before logout).

First, the user founded a nice spot to logout – here the user's character is "half in, half out." The user then selected Reconnect to disconnect from the game.

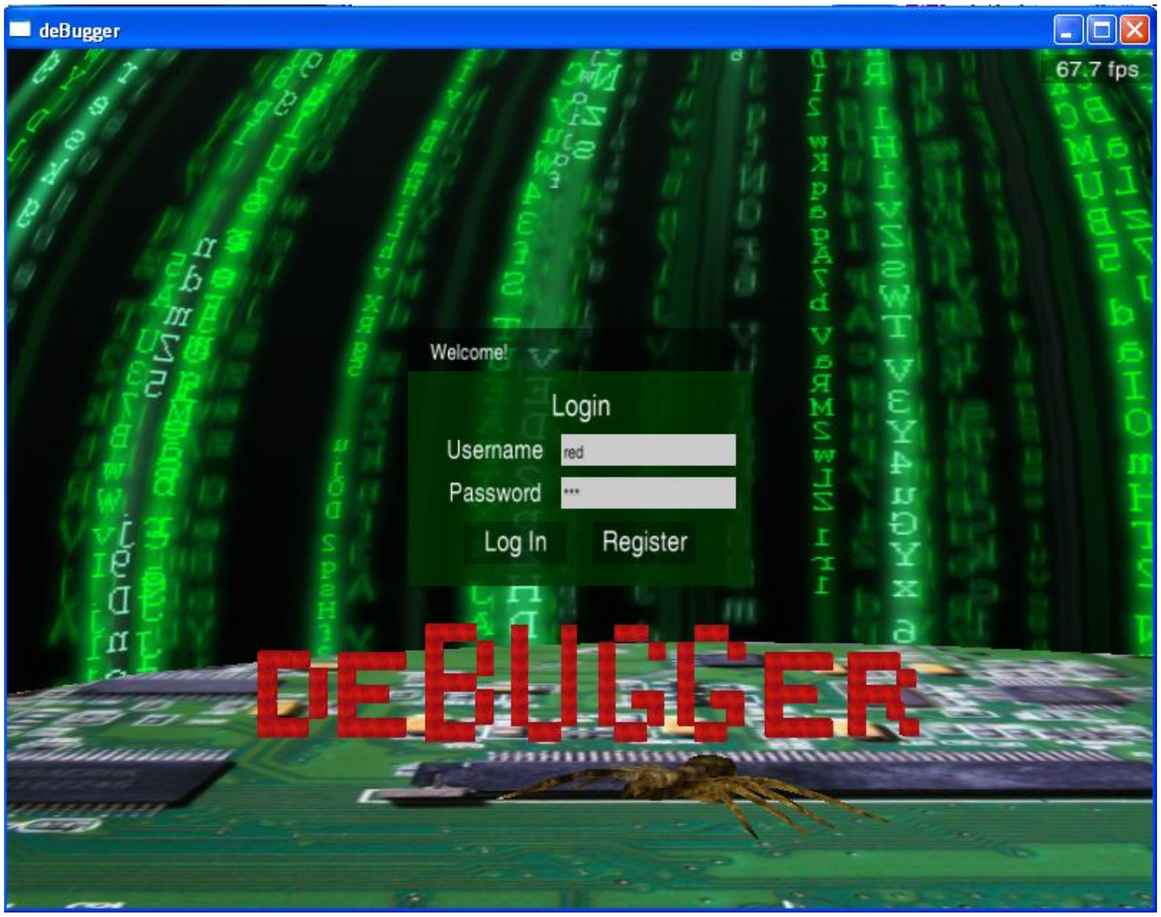
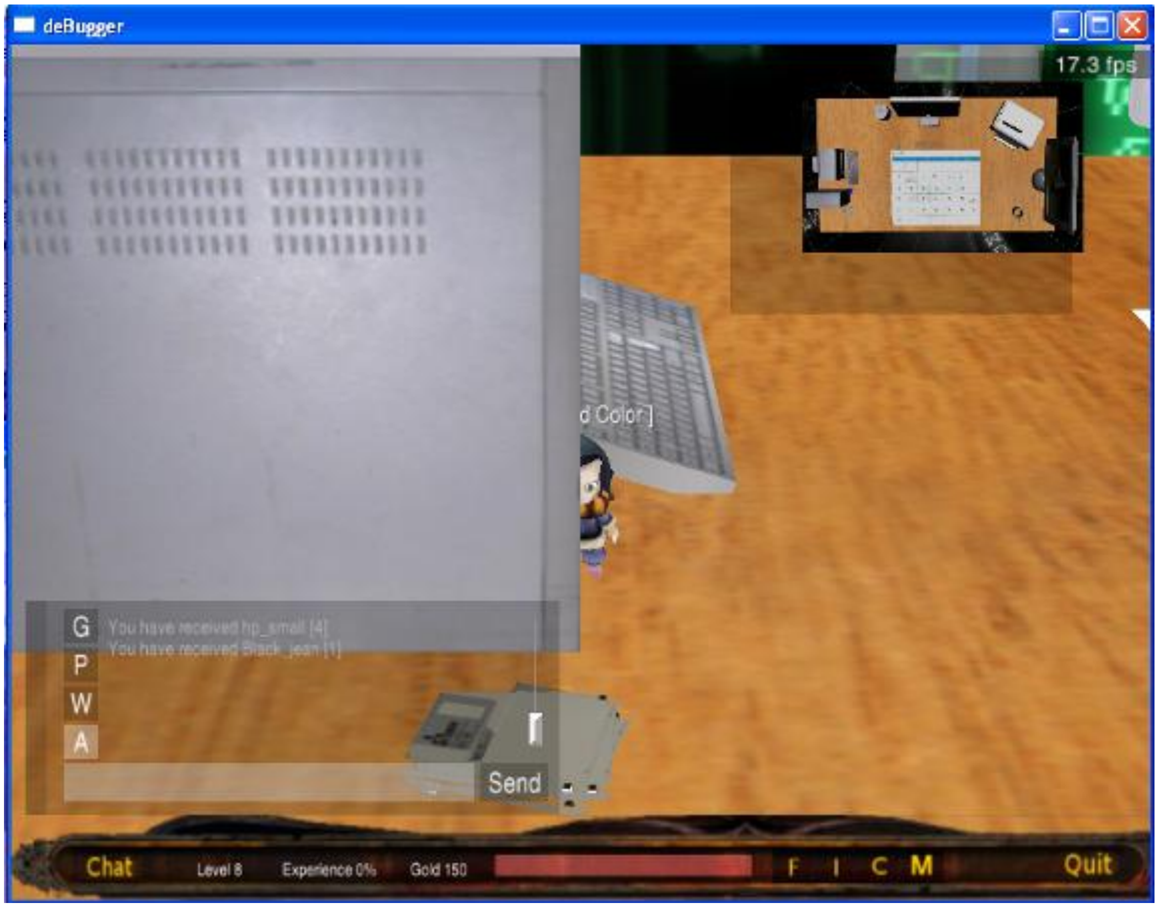


Figure 24: Show login screen in between logout and relog.

Second, the user relogged with the same account.



```
You have received hp_small [4]
You have received Black_jean [1]
```

Figure 25: Show the character restores it position after relog.

Third, the user's character was still "half in, half out" after relogged.

4.h – Launching Team

The launch team was set with the task of making a website that contained enough information in an organized manner that allows interested students to come and play the game. One of the major benefits for such a site is that it completes the package of media that a player is typically interested in seeing before delving into a game.

The launch team had four main goals:

- Site to download deBugger.
- Site easy to navigate.
- Comprehensive site.
- Easy to extend additional functionality.

The goals were accomplished mainly by mimicking the successes of major MMOs that are currently in production.

Debugger's game client installation package [Major Contributor; Alan. Minor; Vivek]

Debugger makes use of Panda3D – a 3D game engine which includes a library of subroutines for 3D rendering, graphics, I/O, collision detection, and other abilities significant to the development of 3D games. For controlling the Panda3D library, we had the option of writing client application code in either C++ or Python. Panda3D's intended game development language is Python due to which the client team opted for it. However, in order for game players to install and start playing Debugger on a Windows and/or Macintosh machine would require a Python interpreter.

As there is no built in support for Python's interpreter on both Windows and Macintosh operating systems a game player will have to first install Python on his machine before he/she could actually start playing or running the game. To avoid a game player from undergoing the frustration of downloading and installing the correct version of Python's interpreter it is recommended to package the game as an installer. The

Debugger’s launch team would create a one step Windows self extracting package that would be available for download on Debugger’s website thereby allowing players to run the game as an executable Windows program.

Panda3D comes with an included helper application called packpanda, which creates a one file executable that is able to install everything necessary to run our application. The specific command that was used in the creation of the setup file; dependant on the developing environment, was ‘packpanda --dir src

--name "deBugger - SFSU" --bam --rmext egg --pyc --rmext py --rmdir .svn’.

We will explain what each option does in order to make it a bit clearer.

--dir is the directory of the location of the Main.py file. This, like C++, is a requirement, so was unable to be changed. Src was the directory under Gaming (our ‘project directory’). Inside of it, we had a Launcher.py file that was changed to a Main.py file to follow the packpanda program. We did not want our game to default to main, so we gave it a name of debugger – SFSU.

–bam and –pyc are the compiled versions of the egg and py files, respectively.

--rmext/dir removes the files with the extensions after the –rmext/dir command, so in our command, we’re removing .egg, .py, and .svn files to lighten the package as well as remove source code and svn files.

Compiling the installer takes about 15 minutes, and works on windows (there is a RPM version, but this seems to be installed on a linux machine, so we cannot test). Size is roughly 81MB without the source code, egg, and py files.

Debugger’s Launch website [Major Contributor; Vivek, Alan. Minor: Ali]

Similar to all gaming sites, to have our presence felt on the web, Dr. Yoon recommended for having a website specifically catered towards Debugger. The website was built using HTML, CSS, Javascript and PHP and will be a one-stop information source for the gaming community. Apart from game download, the website will also provide an overview of Debugger, system requirements and installation of the game. In addition, the website will also provide screenshots of players and scenes, game interface along with brief explanation of each section. A list of both mouse and keyboard controls for activating the various functions/features in the game will be listed on the website. The website would also guide game players as to how to go about with the registration and selection of avatar process. For current and future SFSU students and anyone who would like to be part of Debugger team and interested in knowing as to how Debugger is developed can check out the Download’s section which provides specifications of the same. Lastly, the website also provides link to the Question Creator application wherein one can enter C++ questions and its answers which will be used by the Debugger application.

Design [Major Contributor: Vivek, Alan.]:

To first design a website, the launching team tried to obtain an overall feel for what a MMO typically contains as a website in order to entice and retain players. A general pull of MMO's turned up with a variety of sites, and while not comprehensive, gives a good indication of the various features that are present on all MMO sites. This list of MMO sites is not comprehensive list, but is representative of the content.

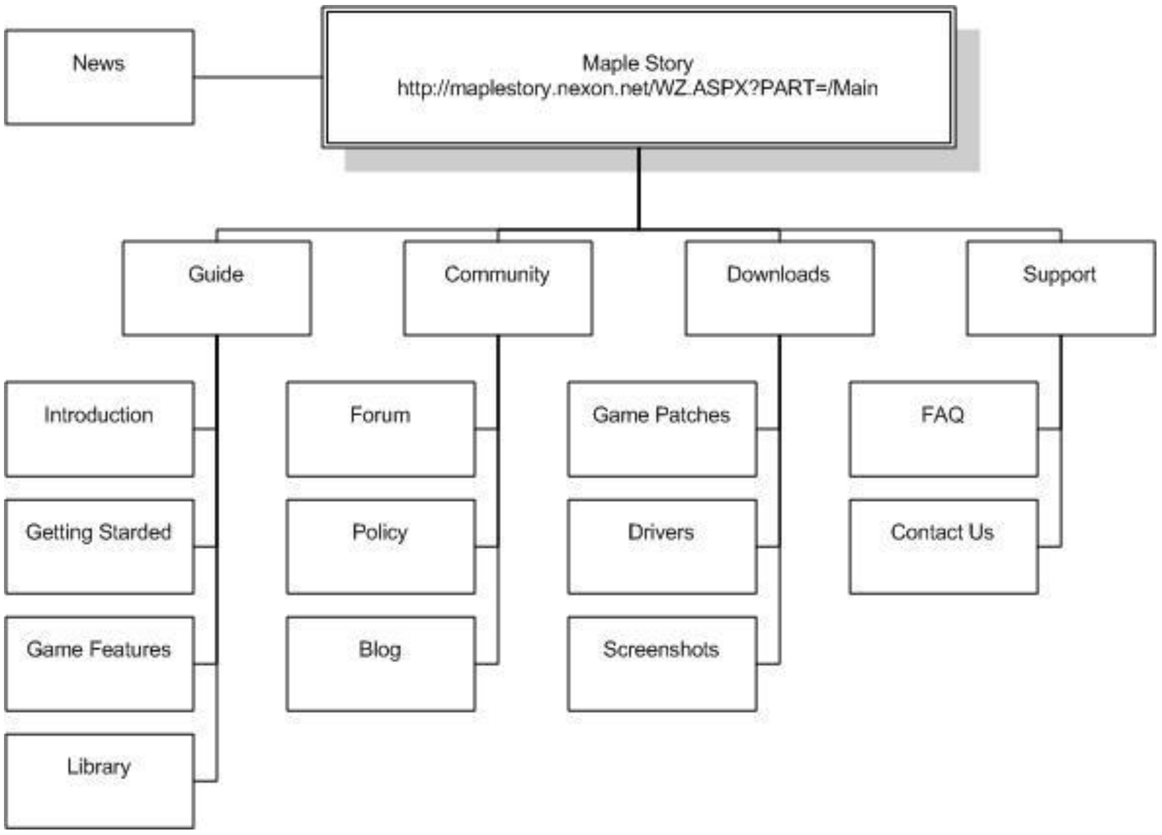
Maple Story

World of Warcraft

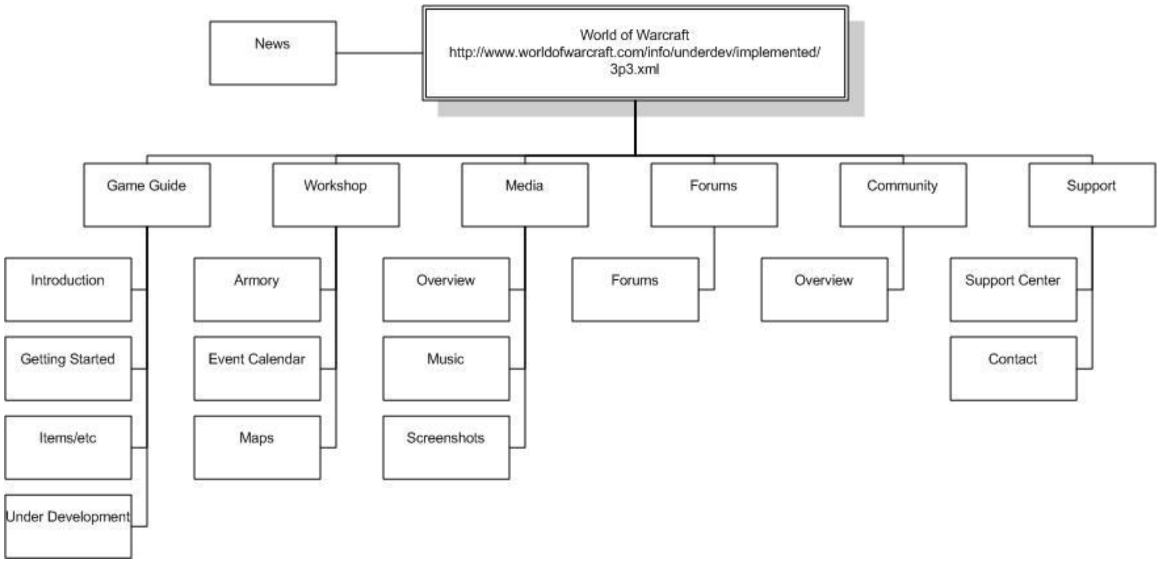
Lineage II

Luna Online

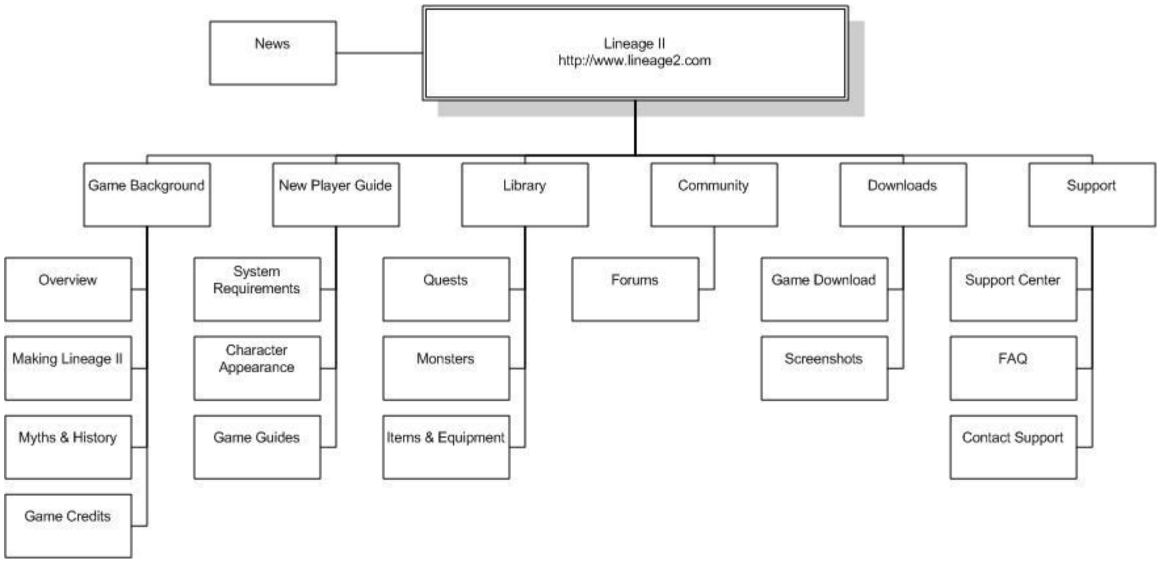
Below are the wire diagrams of the first three, as the fourth is made up of very little links.



4.h-1 – Maple Story and their website layout

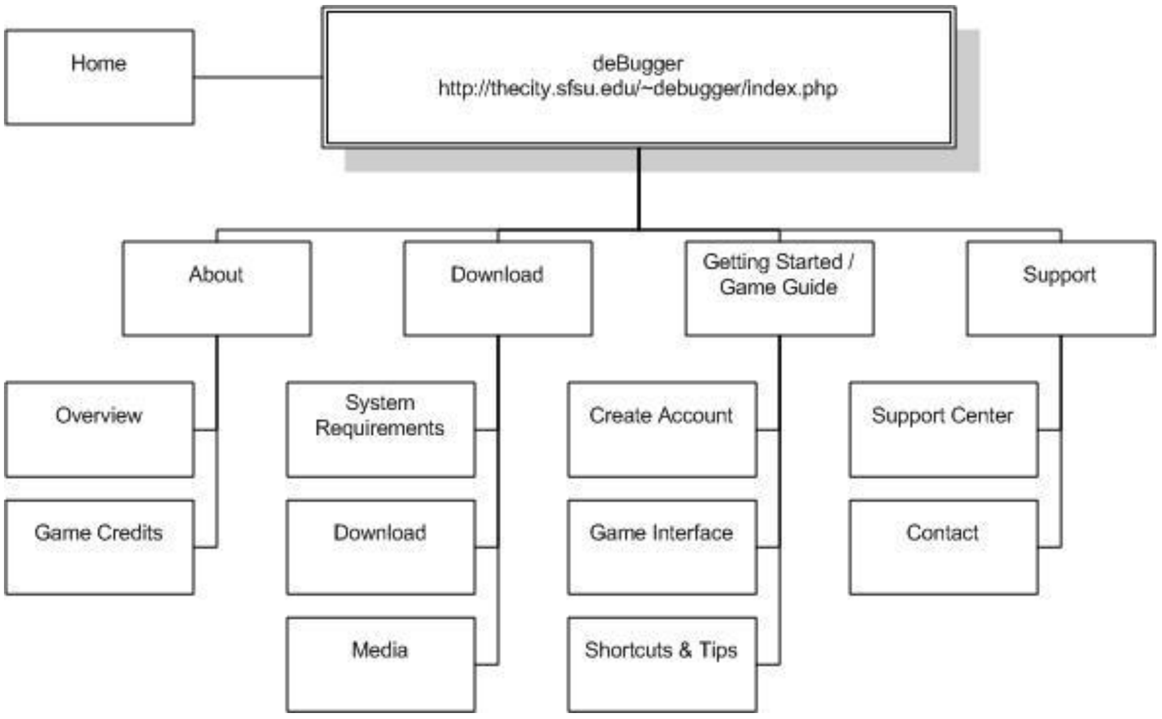


4.h-2 – World of Warcraft and their website layout



4.h-3 – Lineage II and their website layout

Based off of these wire diagrams, we can reduce to the functionalities that is most representative of our game, as well as ridding it of redundancies such as the multiple community links as per the World of Warcraft.



4.h-4 – deBugger and our website layout

It contains all the functionalities that the other sites try to incorporate, in a format that is easily understandable, without any redundant information.

The website organizes the content within folders, which contains the individual php files for each of the webpages.

Chapter 5-Understanding the code

Now comes the documentation of the code that I worked on. I was assigned to work on three classes of the game, Registration.py, Friends.py and Items.py. Each one of these classes had a specific task in the game that will be explained as these classes are introduced to the reader.

REGISTRATION.PY

REGISTER

Before playing the "deBugger" a user must be able to create an account. With a personal account, a user will be log in to the game and have their personal statistics, properties, achievements and items stored. In order to enter the "deBugger" game, a username and password for an account will be checked against an online database before a user is granted access and logged in. In order to create an account with a username and password, one can simply click on the "Register" button available at the login screen.

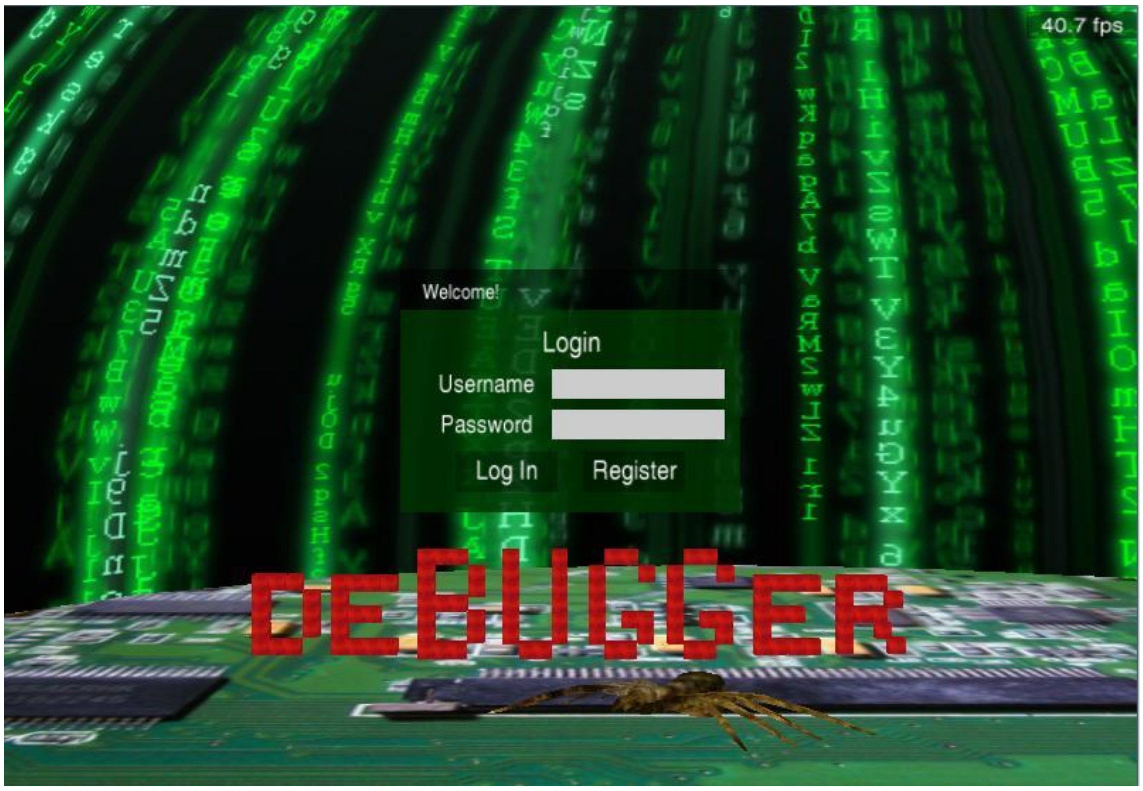
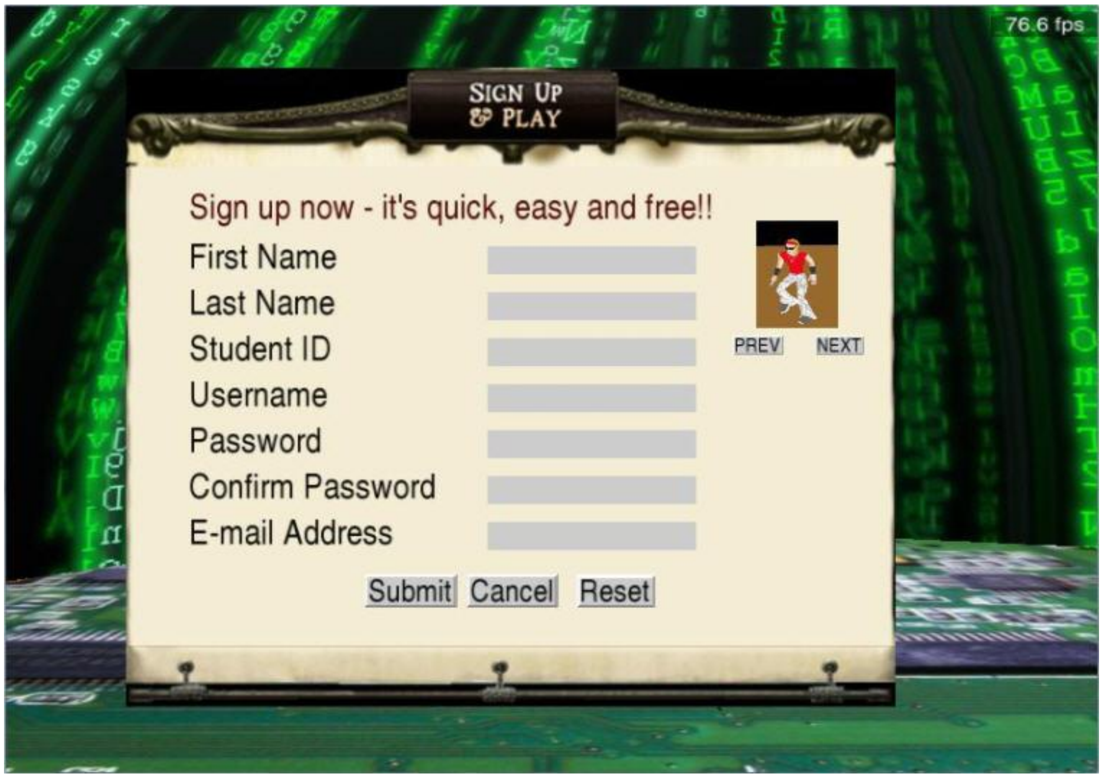


Figure 4.c.i.1 - The Login Screen has a button to direct users to the Registration Screen.

After clicking on the “Register” button from the login screen, the user will be prompted with the registration screen. The registration screen will be where a user will be able to create a new game account that they can use in order to log in and play the “deBugger” game. This screen includes field descriptors alongside text entry field. The user will be asked to enter unique information such their name, student ID and e-mail address. The user will also be asked to come up with a username and password that they can use in order to log in to their account and play the “deBugger” game. The registration screen will also feature a password confirmation field. This is normally implemented to make sure that the user hasn't accidentally mistyped their own password. It also prevents them the frustration from trying to guess what their mistyped password may have been.



Sign up now - it's quick, easy and free!!

First Name

Last Name

Student ID

Username

Password

Confirm Password

E-mail Address

Figure 4.c.i.2 - Registration Screen

Also featured on the registration screen is an avatar selector. In the upper right hand portion of the registration menu, there is a character image along with buttons designated “prev” and “next”. These buttons allow a user to scroll through a list of images of in game playable characters. The user can leave the image on the character that they have found they would like to in the game as. This adds some individuality to a player's in game likeness and doesn't restrict the players to look exactly the same when first starting out. Depending on a player's progress, they may able to add more characteristics to the base character that they choose at the registration screen.

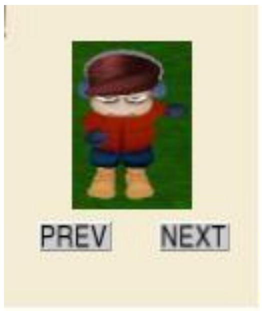


Figure 4.c.i.3 - Users can use the Prev and Next buttons to scroll through playable characters.

The bottom of this screen features buttons that offer functionality over control of the screen. The first button, labeled “Submit”, will take all the information gathered and try to send it to the server to create an account. If a user has entered invalid information then they will be prompted to correct the error(s). The second button, labeled “Cancel”, allows the user to exit this screen and go back to the log in screen. The third button, labeled “Reset”, resets all the text entry fields and makes them blank. This makes it convenient for a user to enter in new information without having to manually clear every field themselves.

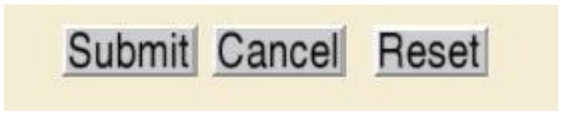


Figure 4.c.i.4 - Action Buttons for user to control the registration screen.

After successfully entering and submitting their data, the user will then be able to log in to the “deBugger” game.

REGISTER – Understanding The Code

The Register.py file is in the main\Register package. This file is used for creating a registration screen. When this screen comes up, the user will be shown text entry fields with descriptions for each field. The user will then be able to enter their information into these fields and submit their information. Their information will then be sent the server to initiate the process for create an game account for the user.

Register Class

This class creates all the informational and text entry fields. It also provides the user with the ability to scroll through a list of images to choose a character model. When the user is done, this class will be able to submit a request to the server with all the user's entered information.

DataMembers

main	Provides access to main's classes.
eFocus	Used for switching focus between text entry fields.
registerEntry	Holds all user inputted values for registration.
envModel	Holds background model for registration screen.
environ	Loads and displays environment for registration screen.
menuTop	Image border for top of registration screen.
menuEdges	Image border for edges of registration screen.
menuBottom	Image border for bottom of registration screen.
path	Holds system path for image selection directory.

charImageList	Holds a list of all files within image selection directory.
characterImageArray	Holds a list of all appropriate files within image selection directory.
pictureIndex	Holds index of current image within characterImageArray being displayed.
menu	Displays pictureIndex's corresponding image to the registration screen.
avatarID	Holds the image file name of the current image.
osNP	TextNode displaying onscreen instructions to the user.
fnameNP	TextNode describing the first name text entry field to the user.
lnameNP	TextNode describing the last name text entry field to the user.
studentIDNP	TextNode describing the student ID text entry field to the user.
userIDNP	TextNode describing the username text entry field to the user.
pwdNP	TextNode describing the password text entry field to the user.
confpwdNP	TextNode describing the confirm password text entry field to the user.
emailNP	TextNode describing the email text entry field to the user.
fnameTextEntry	Text entry field for first name.
lnameTextEntry.	Text entry field for last name.

studentIDTextEntry	Text entry field for student ID.
userIDTextEntry	Text entry field for user name.
pwdTextEntry	Text entry field for password.
confpwdTextEntry	Text entry field for confirming password.
emailTextEntry	Text entry field for email address.
submit	Panda 3D button object. Calls submitText function when clicked.
cancel	Panda 3D button object. Calls login function when clicked.
reset	Panda 3D button object. Calls resetText function when clicked.
prevButton	Panda 3D button object. Calls prevImage function when clicked.
nextButton	Panda 3D button object. Calls nextImage function when clicked.

<u>Member Functions</u>

__init__(self, main)

```
    def __init__(self, main):

        self.main = main

        self.eFocus = -1
        self.registerEntry = []
        self.setupBackground()
        self.setupTextNodes()
        self.setupTextEntries()
        self.setupButtons()

        self.accept('enter', self.submitText)
        self.accept('tab', self.toggleEntry, [1])
        self.accept('shift-tab', self.toggleEntry, [-1])
        self.accept('window-close', self.main.closeClient)
```

This function takes in main as an argument. This function initializes eFocus and registerEntry. It then calls methods to set up the background, text entry fields and buttons respectively. It also creates key commands for keyboard navigation through the menu.

setupBackground(self)

This function loads the background and the environment for the registration screen. It then loads the menu borders on the screen.

Afterwards, it loads and initializes all the images into characterImageArray.

```
        MYDIR = os.path.abspath(sys.path[0])
        self.path = MYDIR + '\\models\\avatarimages\\'
        self.charImageList = os.listdir(self.path)
        self.characterImageArray = []
        for fname in self.charImageList:
            if(fname != '.svn'):
                self.characterImageArray.append(fname)

        currentImage = Constants.MYDIR + '/models/avatarimages/' + self.characterImageArray[0]

        self.pictureIndex = 0
```

It follows with setting up the default values for the array index and avatar image.

```
        self.menu = OnscreenImage(image=currentImage, pos=(0.60, 0, 0.30), scale=(0.1, 1, 0.14))
        self.setAvatarId(currentImage)
```

setAvatarId(self, currentImage)

This function takes a String current image as its argument. It then sets avatarID to the currentImage file name.

getAvatarId(self)

This function returns the value stored in avatarID.

setupTextNodes(self)

This function creates and pins all the descriptive text nodes onto the registration screen.

setupTextEntries(self)

This function creates and pins all the text entry fields onto the registration screen. It also appends all the values in each text entry field to registerEntry.

setupButtons(self)

This function creates and pins all the button objects onto the registration screen.

submitText(self)

This function checks to make sure each text entry field has been filled in. It also checks the syntax on certain fields to ensure no illegal characters have been entered. In the case of an illegal entry, an error message will be called through createErrorBox function.

If all entries are legitimate then the values will be sent to the server for registration processing.

```
elif (self.main.startConnection() is True):
        rContents = {'username': self.userIDTextEntry.get(),
                'password': self.pwdTextEntry.get(),
                'firstName': self.fnameTextEntry.get(),
                'lastName': self.lnameTextEntry.get(),
                'student_id': int(self.studentIDTextEntry.get()),
                'email': self.emailTextEntry.get(),
                'avatar_id': self.getAvatarId() }
#                         'avatar_id': int('3') }
        self.main.cManager.sendRequest(Constants.CMSG_REGISTER, rContents)
```

resetText(self)

This function sets each text entry field within registerEntry to be cleared. It then sets the text cursor to the first entry.

login(self)

This function switches the environment to the login screen environment.

```
    def login(self):
        self.main.switchEnvironment('Login')
```

toggleEntry(self, direction)

This function allows the text cursor to navigate through the text entry fields in a sequential order.

setFocus(self, eNum)

This function takes in int eNum as a parameter. It sets eFocus to eNum, recording the currently focused entry.

selectEntry(self, eNum)

This function takes in int eNum as a parameter. It takes focus off all other text entry fields and sets focus on the eNum'th entry.

showAlert(self)

This function displays failed register message.

showConfirm(self)

This function creates a confirmation dialog.

```
self.confirmDialog = OkDialog( dialogName = 'confirmDialog',
                               text = 'Registration Complete!',
                               command = confirm,
                               state = DGG.NORMAL )
```

When this dialog is clicked, a nested confirm function is called. The confirm function cleans up the dialog and starts the login function.

```
def confirm(arg):
    self.confirmDialog.cleanup()
    self.login()
```

prevImage(self)

This function clears the current displayed image. It then decrements the pictureIndex and checks its boundaries in order to cycle through characterImage array.

```
self.pictureIndex -= 1
self.pictureIndex = self.pictureIndex + len(self.characterImageArray)
self.pictureIndex = self.pictureIndex % len(self.characterImageArray)
```

It then displays the new picture.

nextImage(self)

This function clears the current displayed image. It then increments the pictureIndex and checks its boundaries in order to cycle through characterImage array.

```
self.pictureIndex +=1
self.pictureIndex = self.pictureIndex % len(self.characterImageArray)
```

It then displays the new picture.

createErrorBox(self, msg)

This function taking is String msg as a parameter. It creates an error message dialog using Panda 3D OkDialog.

```
    self.errorDialog = OkDialog( dialogName = 'errorDialog',
                                 text = msg,
                                 command = cleanUp,
                                 state = DGG.NORMAL )
```

When the dialog is clicked, the button calls the nested cleanUp function. The cleanUp function clears the dialog and calls selectEntry.

```
    def cleanUp(arg):
        self.errorDialog.cleanup()
        self.selectEntry(self.eFocus)
```

unload(self)

This function destroys all class variables.

FRIENDS.PY

FRIENDS OBJECT

def __init__(self, name, index, friendColumnPosition, friendTopRowPosition)

The init method inside FriendsObject class, is responsible for setting up the position of the name boxes inside the friends list, whenever a name needs to be added, this constructor is called in order to specify position of the name fields, as well as size and color.

def unload(self):

The unload method is in charge of hiding any images that might be on the screen, (like the list itself) when the user chosses to do so. Mainly when the 'f' button from the navigation bar is pressed.

FRIENDS

def __init__(self, main):

This init method inside of the friends class does all of the initialization, It "builds" the friends list array where all the friends name will be stored, builds the title bar, the frame of the list, as well as sets up the image path so it knows where the arrow images are stored.

def acceptFriendList(self, serverFriendList):

The accept friends list method receives the names from the server, the server constantly keeps sending the list to the class, that is why we first check if the list has been accepted, if the list has not been processed, then we will enter the if statement and start separating the names. The server sends the name in the following fashion Tom:Jerry:John:Josh, notice that the names are separated by a colon, This is why inside the list we “split” the array, so we know when a new name starts.

def scrollListUp(self):

The scroll list up method does exactly that, scrolls the list up when the user wants to scroll the friends list. This was perhaps one of the more challenging implementations for us because the specs kept changing all the time, and because it would behave differently when testing the list locally and when receiving the names from the server, hence, there are some many checks going on in here, but the basic flow is as follows. The method first checks to see if names exist inside the list, if name exists and its less than 5 names, just keep writing the names, there is no need to scroll (if we were to scroll in this case, the game would crash because of an out of index array exception). Then we would continue to the next statement, where more than five names exist. Here we have to check again if the list is empty, again we have to do this check to avoid crashing, we keep unlading the names and keep subtracting one, meaning that one names has been processed, so we have n-1 names to iterate, this brings the next check, if the array index reaches a negative value, then we just reset it to zero, then write the names to the screen.

def scrollListUp(self):

The scroll list down is very similar to the scroll list up method, the checks are the same but logic is a bit different, where we check the names that are left to iterate by subtracting 5 to the entire list (only five names are visible at all times) then we check to see if the array index has gone out of boundaries by comparing it with the names that are left to iterate, if it is bigger, we just set it equal to the names left to iterate variable.

def writeFriendsToScreen(self):

The write friends to screen writes the friends names to the list, this is also a large function that underwent lots of modifications when the names where actually being received by the server. Basically, we “separate” this method in two parts, the first one is when the list is empty and the names that are already written is less than five, if this is the case, then we need to make sure that the index is not negative, and so we then place the names inside the list, the variables friends list row and column keep changing as a name is added, so they do not overlap other names that have already been written into the list. Almost the same thing is done when the list already contains five names or more, only that now we iterate from 0 to five, because only five names can be printed at most, also we check to see if the name is too long, a name with more than ten characters will not fit inside the names box, so it has to be split up.

def sendEntry(self):

The send entry method receives the name to be added to the list turns it into lower case, and checks to see if that name already exists, if it does, an error message is displayed. If it doesn’t a popup window will ask the user if it ok to add a certain user, the answer is sent to the verify selection method

def verifySelection(self, args):

Upon receiving a ‘yes’ answer by the user, this method will sent the request to the server using the protocols written by other team members.

def buddyAnswer(self, accepted):

Buddy answer receives the answer of the user that is to be added, if the other user responds yes to the request, then the name of that user is added locally to the list, and then this action is sent to the server.

def deleteEntry(self, accepted):

The deleteEntry method deletes a friend from the list, in this case, we do not need to inform any user that he/she is to be deleted from a users list, deleteEntry uses deleteBuddy method to delete locally, deleteEntry sends the request to the server to delete the name from the server list.

def updateList(self, accepted):

The updateList method is in charge of refreshing the list as soon as a name is deleted or added to the list. The same algorithm from scroll list up is used, as soon as the name is added or deleted, the list is just scrolled up in order for the changes to be visible immediately..

CODE STRUCTURE OF THE MODELS FOLDER (ART SUPPORT TEAM)

```
/models
        /bugs
                /bug1
        /characters
                /males
                        /male1, male2, male3, male4, male5, ralph
                /females
                        /ralphSister
        /minimaps
        /scenes
        /screenshots
```

Here are a few details about the above folders:

/bug1 - contains a model of a tarantula bug. It is a very low resolution model (less than 1000 polygons). It has 6 different animations (e.g walk, run, attack, death) and several textures. The textures are of different colors; they can be applied to a bug model from the code and this will simulate various types of bugs.

/male1, male2, male3, male4, male5 are the folders of the same male character. The first semester didn’t include character customization as an item into the scheduled milestones, therefore, the idea behind multiple folders with different textures for the same character is to create a visible diversity until the proper character customization will be implemented. The only thing which differentiates the characters in the above folders are the textures. Each folder contains a screenshot of the character for visual representation and for inclusion into the registration window.

/ralphSister – the only working version of a female character (as of Fall 2009). All other attempts to animate and convert female characters were unsuccessful due to various problems which couldn't be resolved. The model comes with a number of proper animations (walk, run, etc).

/minimaps – folder contains the screenshots of the level maps. The screenshots were taken from Maya application by a simple Screen Capturing program and modified in Photoshop (resized and cropped). These images can also be captured from pview tool (one of the tools in /bin folder of Panda3D game engine. Refer to Panda3D manual).

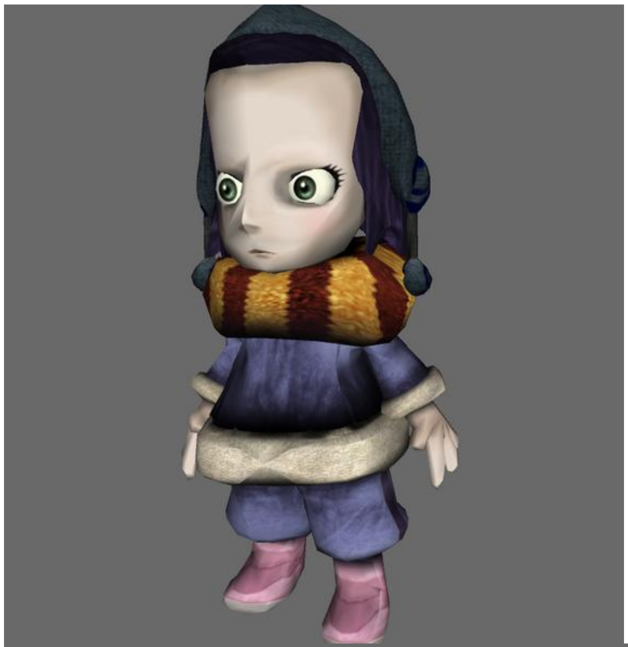
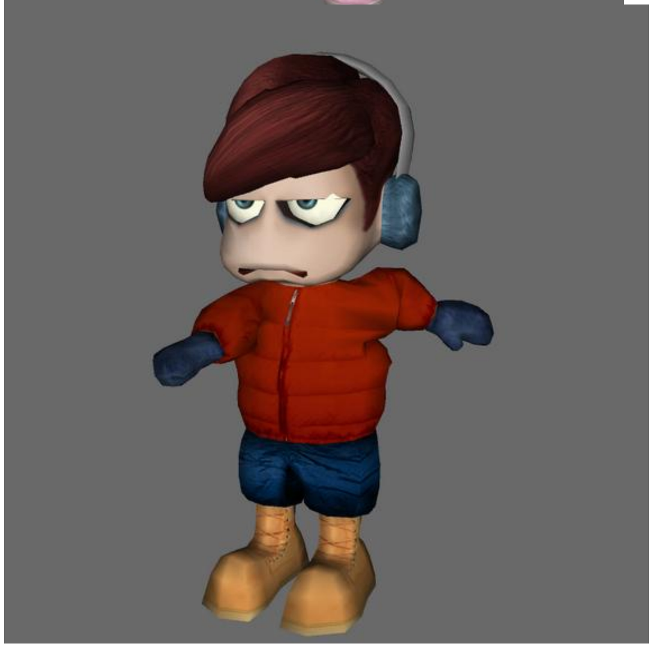
/screenshots – as the name implies, the folder contains various screenshots of the game levels. These images can be used on the accompanying web site for the game.

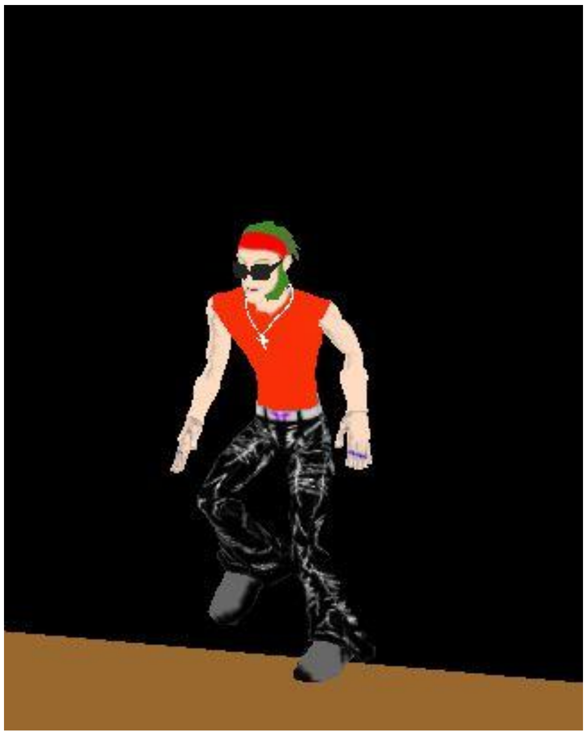
Currently the root of /models folder also contains various images used for the UI of the game windows (buttons, panels, etc). If the next group of students decide to improve them, it will be a good idea to gather them under one folder and call it /ui, for example.

SCREENSHOOTS

Characters

Ralph and Eve

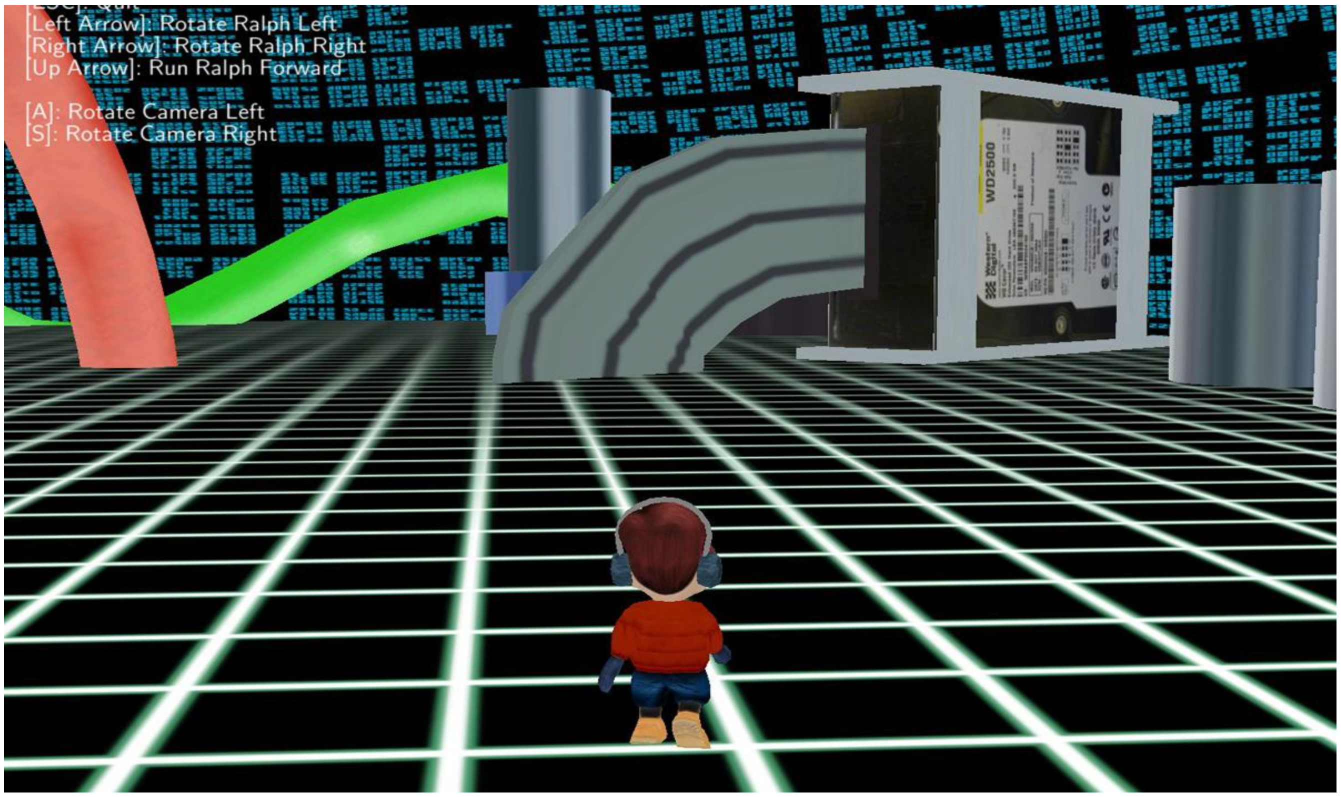


Male Character (with multiple reincarnations)

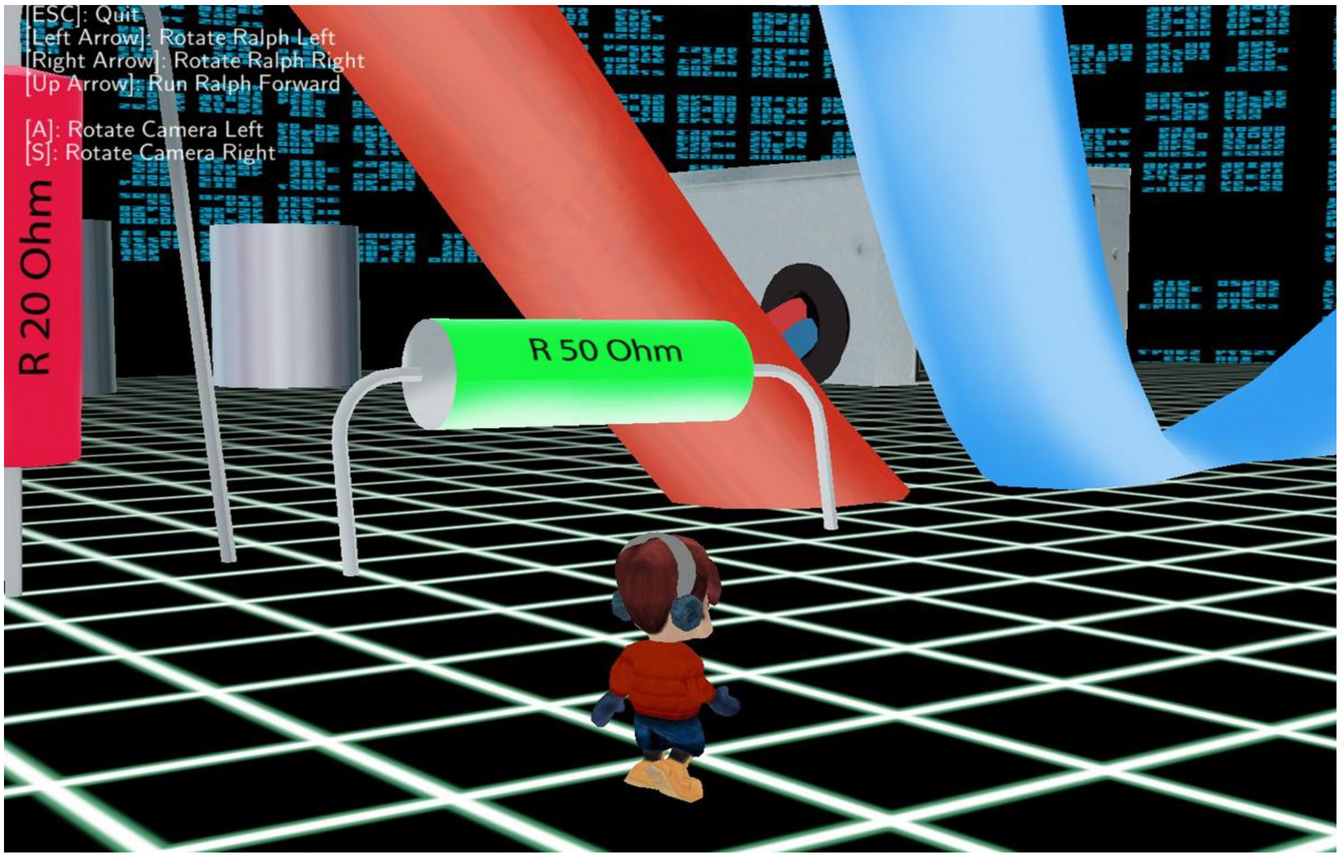


GAME LEVELS

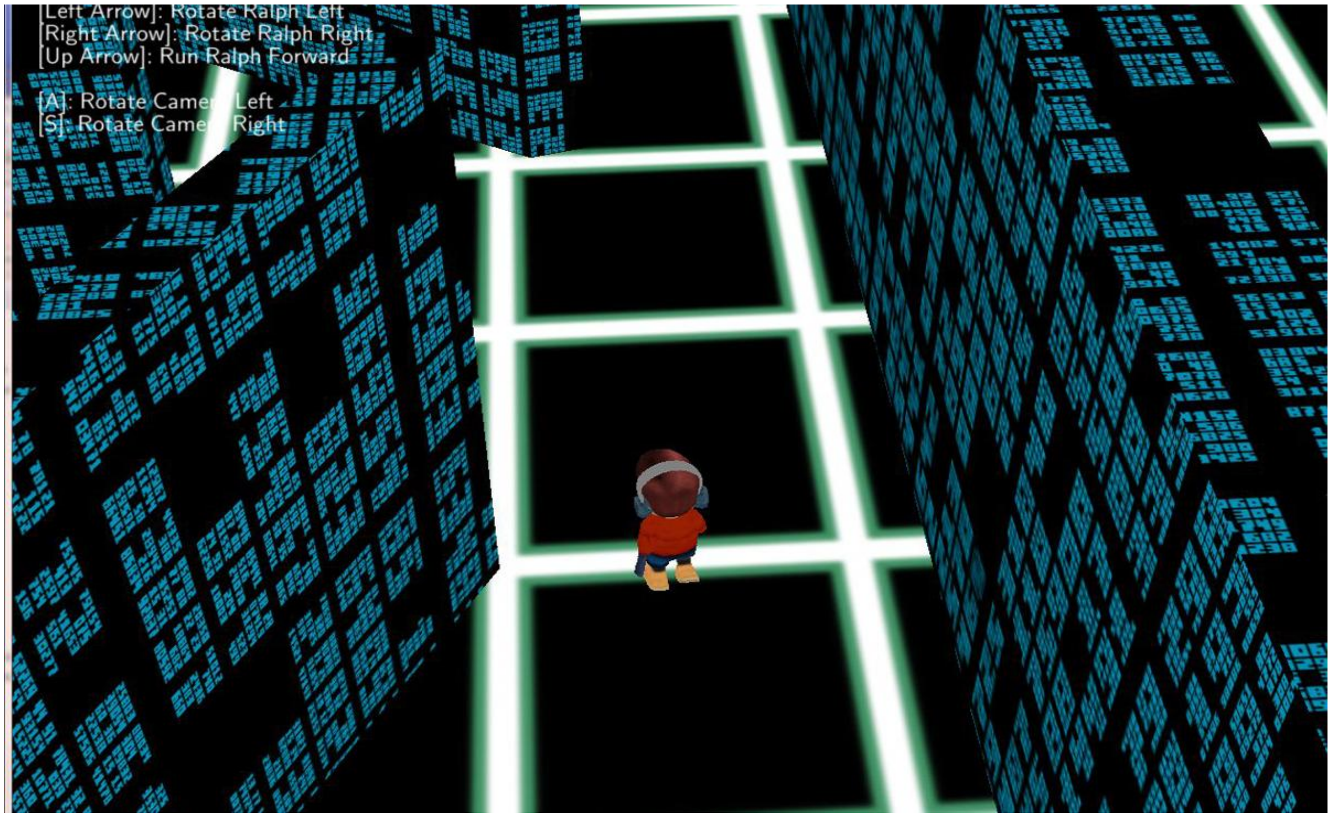
MAIN GAME LEVEL

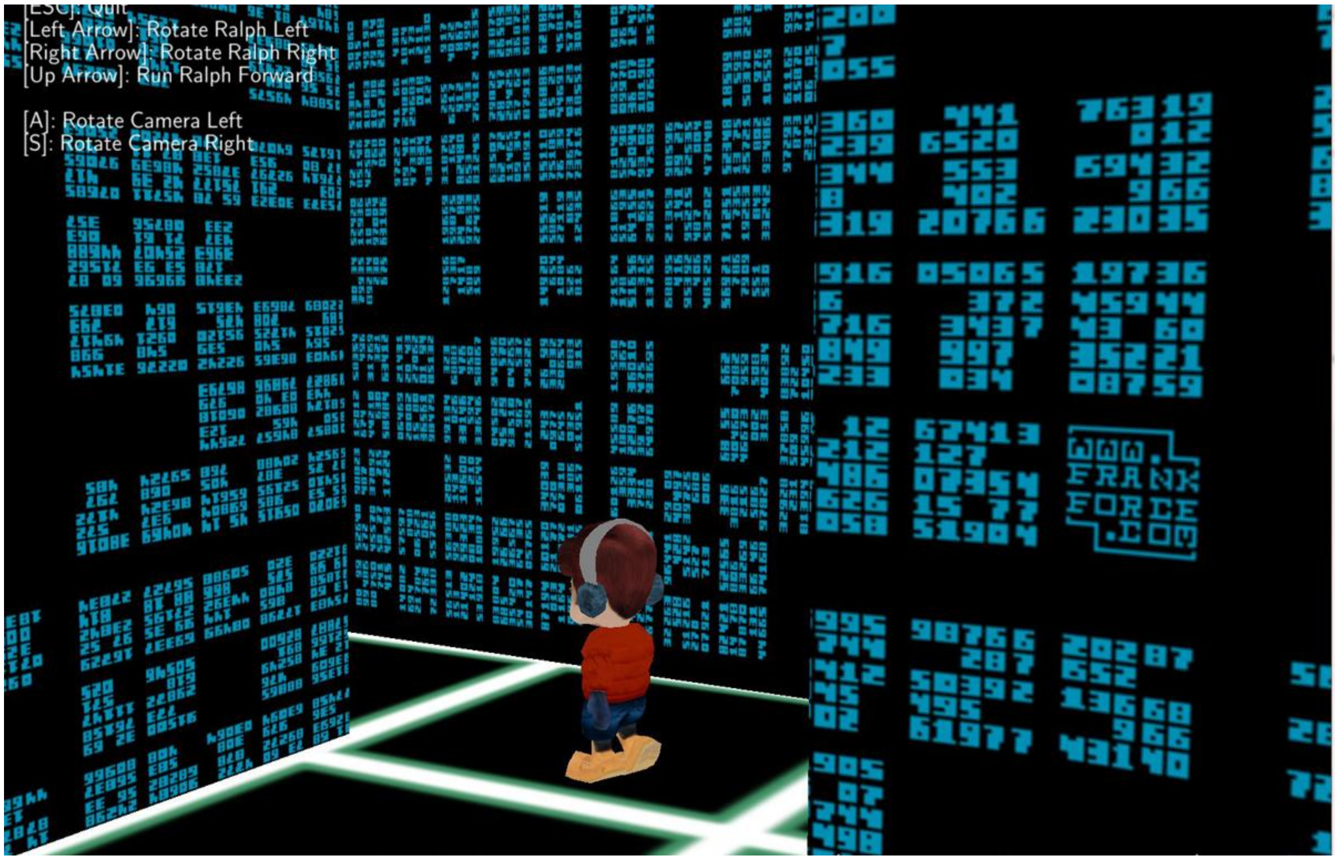
EASY MOTHERBOARD LEVEL


[ESC]: Quit
[Left Arrow]: Rotate Ralph Left
[Right Arrow]: Rotate Ralph Right
[Up Arrow]: Run Ralph Forward
[A]: Rotate Camera Left
[S]: Rotate Camera Right
WD2500


[ESC]: Quit
[Left Arrow]: Rotate Ralph Left
[Right Arrow]: Rotate Ralph Right
[Up Arrow]: Run Ralph Forward
[A]: Rotate Camera Left
[S]: Rotate Camera Right
R 20 Ohm
R 50 Ohm

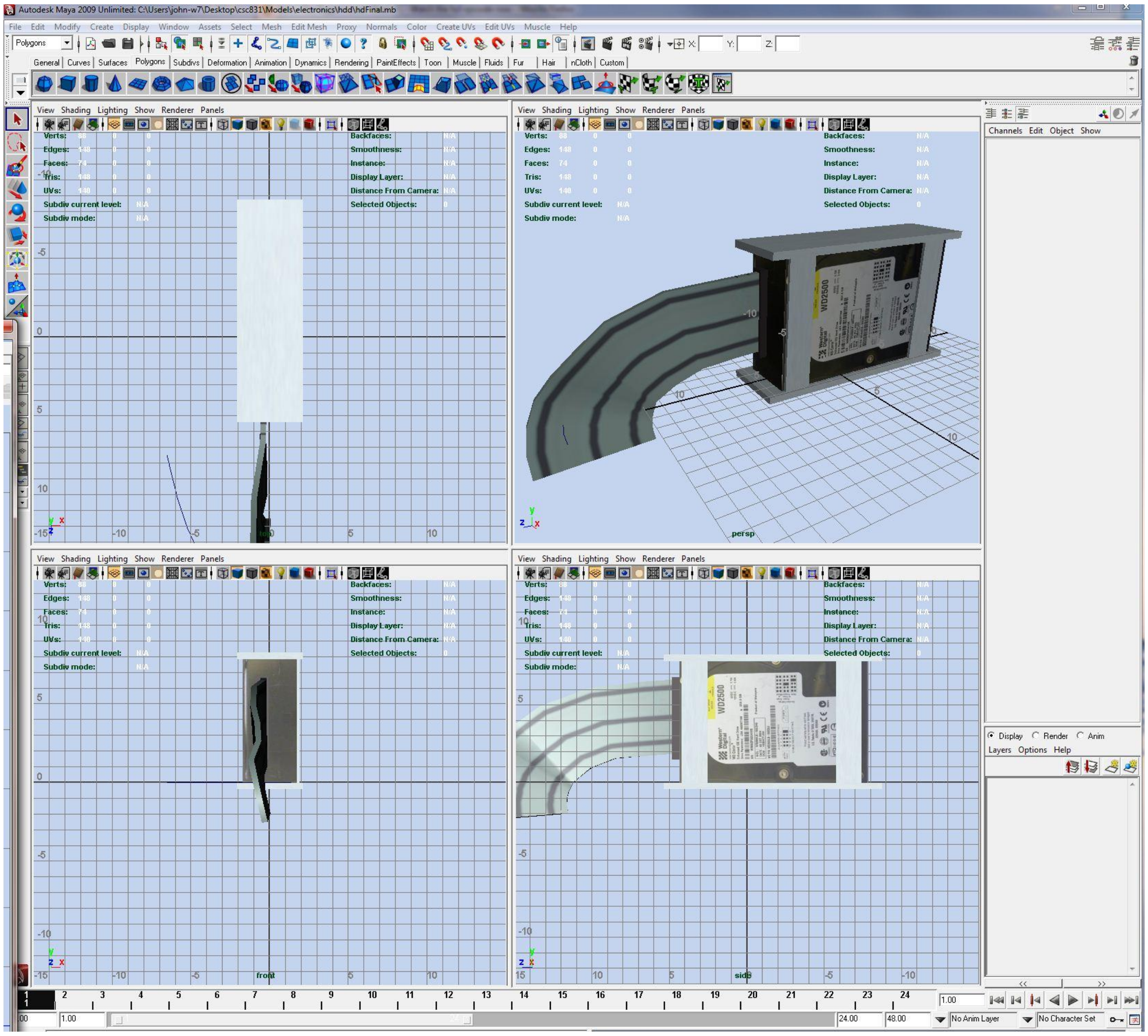
EASY DUNGEON LEVEL


[Left Arrow]: Rotate Ralph Left
[Right Arrow]: Rotate Ralph Right
[Up Arrow]: Run Ralph Forward
[A]: Rotate Camer Left
[S]: Rotate Came Right


[ESC]: Quit
[Left Arrow]: Rotate Ralph Left
[Right Arrow]: Rotate Ralph Right
[Up Arrow]: Run Ralph Forward
[A]: Rotate Camera Left
[S]: Rotate Camera Right
WWW.FRANKFORCE.COM

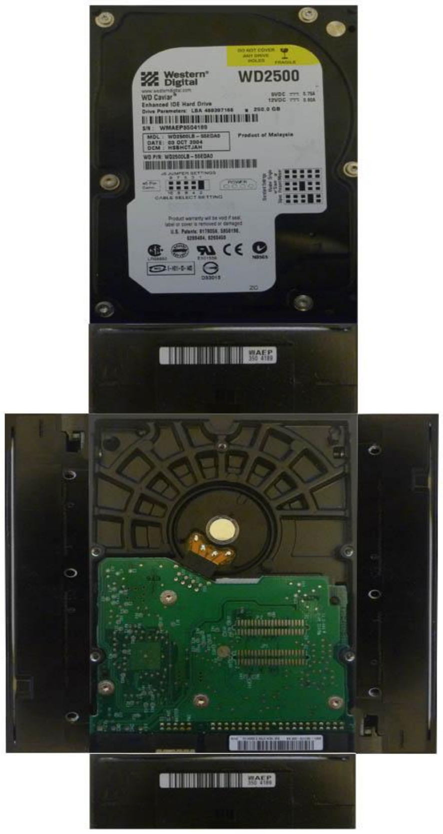
Overview of the steps to create and convert a simple 3D model into .egg format.

We will use hard drive model as an example. The following picture shows the final result in Maya application.


Autodesk Maya 2009 Unlimited: C:\Users\john-w7\Desktop\csc831\Models\electronics\hdd\hdFinal.mb
File Edit Modify Create Display Window Assets Select Mesh Edit Mesh Proxy Normals Color Create UVs Edit UVs Muscle Help
General Curves Surfaces Polygons Subdivs Deformation Animation Dynamics Rendering PaintEffects Toon Muscle Fluids Fur Hair nCloth Custom
View Shading Lighting Show Renderer Panels
persp
front
side
Channels Edit Object Show
Display Render Anim
Layers Options Help
No Anim Layer
No Character Set

The above model is very simple: it consist of just seven boxes. All of the elements of this model are basic polygon cubes. Polygon cube is a primitive building object in Maya and you will be able to create it with a simple click of a button. Also, all primitive building polygonal objects in Maya come with default set of UV coordinates, which greatly simplifies the work of a modeler.

Since the coordinates are there, the only thing needed to be done is to create a shader (e.g.material) and assign texture to it which later will be applied to a cube. A texture can be created in pretty much any image editing program: Gimp, Photoshop, FireWorks, etc. First, you need to get pictures of a hard drive. We took a camera and made six pictures of the hard drive from a very close distance in order to eliminate distortions due to the prospective nature of the camera lens. Then we imported these six pictures into a single project of the image editing software (our case Photoshop). You can "stitch" the sides of the hard drive together as we did in the picture below, but this is not necessary and you can lay them out in the way which makes the most sense to you. The only important requirement is the size of the resulting image should be divisible by 2. The most widely used sizes are 512*512, 1024*1024, 2048*2048 and etc.



When the texture is ready, we go back to Maya to create a new shader (Phong, Blinn) and assign the texture created above to that material and the last step is to assign the cube to the created material. There are good chances that the texture won't be properly aligned with the sides of the cube, but this can be easily fixed. Adjustments to UV coordinates are done in UV Texture Editor in Maya. We won't go into the details of how to do that, just go to www.youtube.com and search for "uv coordinate layout maya" and you will find a number of tutorials which will show you how to do that. In the end if all went well, you will have a box which will look like a hard drive.

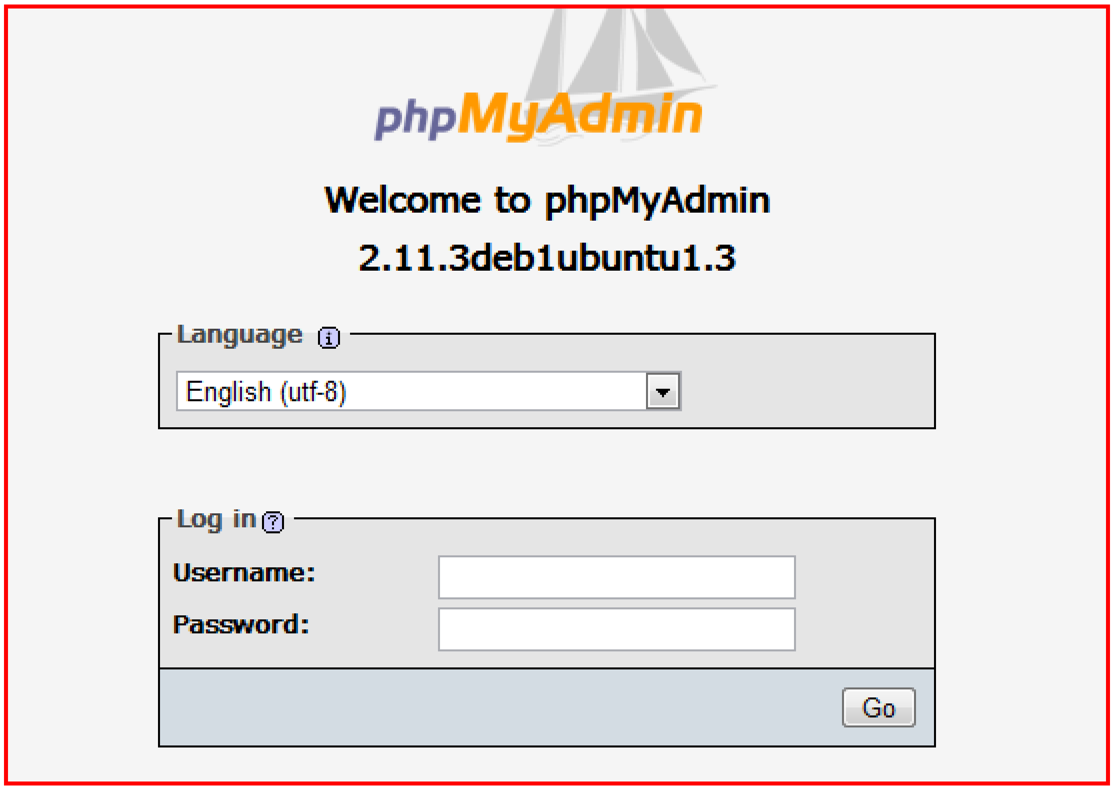
The above model also has some extra elements (hard drive caddy and IDA cable). All these elements are modeled from simple polygon cubes as well and the process is identical to the described above.

When the model is finished it is a good idea to group all elements of the model (in our case, hard drive, hard drive caddy, IDE cable) into one group. In Maya → Windows-> Outline, choose the above elements and press: Control+G and give it a name. This is important for keeping things under control as projects (especially game levels) grow in size and complexity.

CREATING OR SETTING UP AN ACCOUNT ON THECITY.SFSU.EDU

Major Contributor – Tun Win

Firstly, I want to explain what thecity.sfsu.edu does. It is just a server for computer science undergraduate level and it can be used for many purposes for SFSU computer science department's researches. In our debugger game, we use this server to communicate between server team and database team. Now, I will start giving you a link to create a database http://thecity.sfsu.edu/~khanhman/mysql/add_mysql.html. With this link, I create the whole new database. Creating an account is simple that you just give your new database username and password. After you create an account, you can start actually using by going to this link http://thecity.sfsu.edu/phpmyadmin/. Then, Fig.1 screenshot will ask you username and password. It will be same as you filled on this page, http://thecity.sfsu.edu/~khanhman/mysql/add_mysql.html.



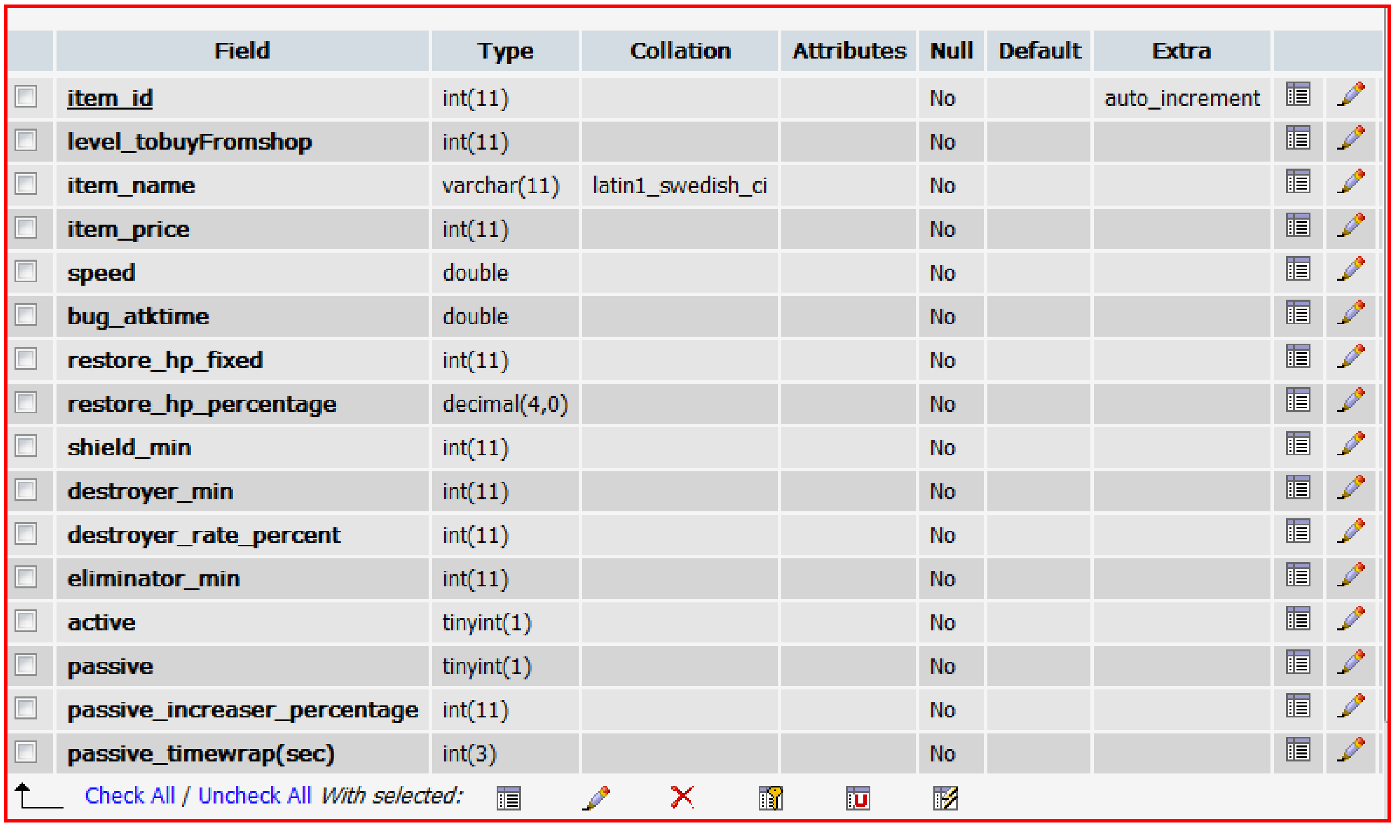
Showing the sample of thecity.sfsu.edu/phpmyadmin webpage

Most of the time, you will be accessing this page to change, modify and design your database. Server team will also use the username and password to retrieve the data you will have in this database. After you successfully log into this page, you will have nice GUI tools (phpMyadmin) to aid your database design. If you will be reusing debugger database from this semester, username is "deBuggerUser" and password is just "deBugger". These username and password are case sensitive. So, I suggest you to copy and paste it into thecity server.

Table- general_item

Major Contributor – Tun Win

Minor Helper – Alvin Wan and Calvin Kuang



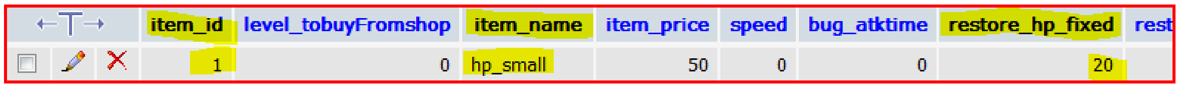
	Field	Type	Collation	Attributes	Null	Default	Extra	
☐	item_id	int(11)			No		auto_increment	
☐	level_tobuyFromshop	int(11)			No			
☐	item_name	varchar(11)	latin1_swedish_ci		No			
☐	item_price	int(11)			No			
☐	speed	double			No			
☐	bug_atktime	double			No			
☐	restore_hp_fixed	int(11)			No			
☐	restore_hp_percentage	decimal(4,0)			No			
☐	shield_min	int(11)			No			
☐	destroyer_min	int(11)			No			
☐	destroyer_rate_percent	int(11)			No			
☐	eliminator_min	int(11)			No			
☐	active	tinyint(1)			No			
☐	passive	tinyint(1)			No			
☐	passive_increaser_percentage	int(11)			No			
☐	passive_timewrap(sec)	int(3)			No			

Check All / Uncheck All *With selected:*

Fig.2 The whole structure and types of general_item table

In general_item table, we store all items we can buy from stores such as shirts, jeans, health_points and TA help. Usually, most of the items for decoration can be bought without level limit. Other items that can help character's performance will be checked by level_tobuyFromshop. The details which describe how each particular item work for characters will be available in Game Concept Design Team's implementation section. But, I will explain how this table should be used and retrieved properly from Server team. Client team will give lists of item to player to choose graphically in the game. Then, client team will communicate our database using server team.

Once client team reach to our database's general_item table, their request will be integer and this integer will indicate item_id. So, client team will properly give out item_id whatever a player has chosen in the game to buy for his or her character. Meanwhile, the user_id which is trying to buy this item_id will be checked in user table. In user table, we will have user_id and level to check that this use is eligible to buy this item or not. But, this checking will not effective for simple decorative item like shirt, jean and hat because those items can be bought from this table without restrictions. There are multiple columns which will be turned on if user buys that particular item. For example, if user buys hp_small which is item_id 1 as you can see in Fig.3 picture, we will retrieve user_id and health from user table. Then, health will be increment to 20 for that user_id. The items from the whole table will be working as this procedure since we design it this way.



item_id	level_tobuyFromshop	item_name	item_price	speed	bug_atktime	restore_hp_fixed	rest
1	0	hp_small	50	0	0	20	

Fig.3 Showing how the whole table should work.

Columns from general_item table	Descriptions of a column
item_id	Type – int Extra feature – auto-increment(auto increment means that it will automatically increment next row as soon as user fill in some data in other columns.) This data is used to retrieve and check item

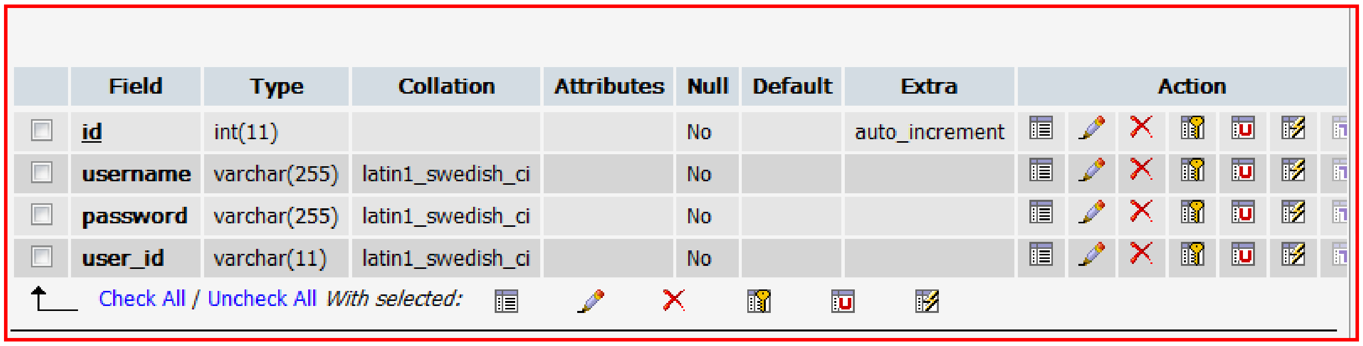
	properties.
level_tobuyFromshop	Type –int Extra feature – NONE This data is used to check whether user have enough level to buy particular item.
item_name	Type – varchar Extra feature – NONE This data can be used for making sure that server team retrieves the right data.
Item_price	Type – int Extra feature – NONE This data is used to subtract coin or money from user's table.
speed	Type – double Extra feature – NONE This amount of speed will multiply to user's speed
Bug_atktime	Type – double Extra feature – NONE

	When character attack the bug, the hits from character will be multiplied by this data.
Restore_hp_fixed	Type- int Extra feature – NONE When character is hit by the bug, the character will lose some hp points. Buying this restore_hp_fixed will add data to its hp.
Restore_hp_percentage	Type – decimal Extra feature – NONE This is different version of restoring the health points for player. Instead of int, we will go by percentage when we adding the health points.
Shield_min	Type – int Extra feature – NONE Shield_min means the minute of character can hold the shield to protect from bug.
Destroyer_min	Type – int Extra feature –NONE Bug will be destroyed until the amount of data when the player buys this item.
Destroyer_rate_percent	Type – int Extra feature –NONE This is just another version Destroyer_min.

Eliminator_min	Type –int Extra feature – NONE A question that is asked by bug will be eliminated for the amount of data which is from this column.
active	Type – tinyint Extra feature – NONE When active is on, user will control to use destroyer, eliminator and shield.
Passive	Type – tinyInt Extra feature – NONE Passive will be on by default
Passive_increaser_percentage	Type – int Extra feature –NONE This data will increase drop rate of bug.

Table- userInfo

Major Contributor – Parijat



	Field	Type	Collation	Attributes	Null	Default	Extra	Action
☐	**id**	int(11)			No		auto_increment	
☐	**username**	varchar(255)	latin1_swedish_ci		No			
☐	**password**	varchar(255)	latin1_swedish_ci		No			
☐	**user_id**	varchar(11)	latin1_swedish_ci		No			

Check All / Uncheck All *With selected:*

I think Parijat is using this table to access his website which will be creating the questions. The way we are going to link this table with user table will be placing a user_id from user table. So, we can check which player is contributing our question table.

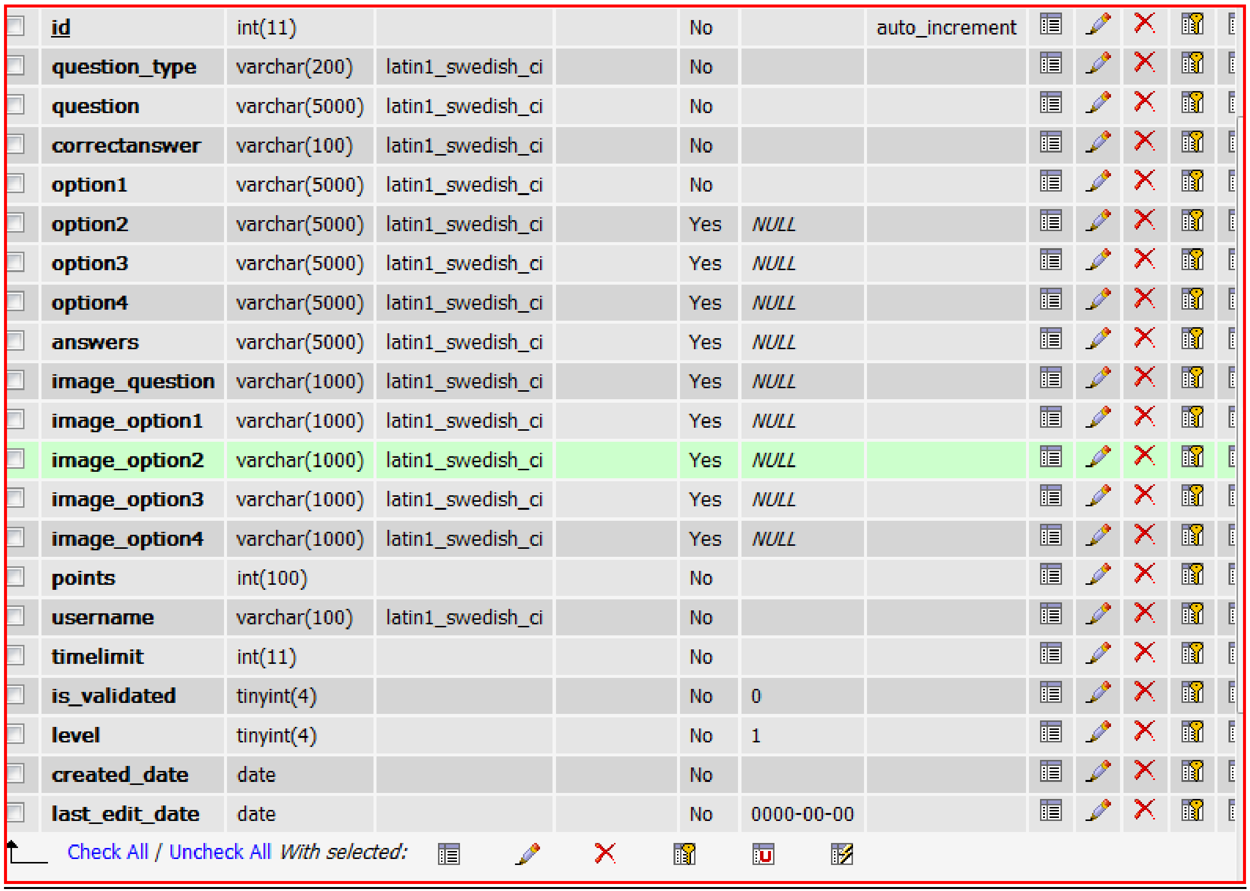
Columns from userInfo table	Descriptions
Id	Type –int Extra feature – auto_increment It is similar to user id since every table should have one to distinguish within rows.
Username	Type –varchar Obviously, this is username to access question creator website.
Password	Type – varchar This is the password to access question creator website.

User_id	Type – int This user_id is from user table. This data is used to link with user table and userInfo table.

Table- questions

Major Contributor – Parijat

Minor Helper – Tun Win (I initiated this table format and Parijat modified as this table needed to fit for question creator website.)



id	int(11)			No		auto_increment
question_type	varchar(200)	latin1_swedish_ci		No		
question	varchar(5000)	latin1_swedish_ci		No		
correctanswer	varchar(100)	latin1_swedish_ci		No		
option1	varchar(5000)	latin1_swedish_ci		No		
option2	varchar(5000)	latin1_swedish_ci		Yes	*NULL*	
option3	varchar(5000)	latin1_swedish_ci		Yes	*NULL*	
option4	varchar(5000)	latin1_swedish_ci		Yes	*NULL*	
answers	varchar(5000)	latin1_swedish_ci		Yes	*NULL*	
image_question	varchar(1000)	latin1_swedish_ci		Yes	*NULL*	
image_option1	varchar(1000)	latin1_swedish_ci		Yes	*NULL*	
image_option2	varchar(1000)	latin1_swedish_ci		Yes	*NULL*	
image_option3	varchar(1000)	latin1_swedish_ci		Yes	*NULL*	
image_option4	varchar(1000)	latin1_swedish_ci		Yes	*NULL*	
points	int(100)			No		
username	varchar(100)	latin1_swedish_ci		No		
timelimit	int(11)			No		
is_validated	tinyint(4)			No	0	
level	tinyint(4)			No	1	
created_date	date			No		
last_edit_date	date			No	0000-00-00	

Check All / Uncheck All *With selected:*

When I started to create this table, I did not include image_option because I did not know that we need to store this information. The way this table works is as usual as other tables. This table will be checking the answer which is given by player with the correct answer. After checking, this table will be filled with points for each player. In addition, player can only answer to the question which is validated and some of the questions will be advance level. In order to take an advanced level question, player need to be at least in level which is specified in this table. There will be only two question types in this game which are true-false type question and multiple-choice questions.

When a player validates a question in this table, we need to record username, validation time, and money which a player will get for contributing. So, the player cannot argue that he didn't get money or credit for giving a question. Since we have to deal with a lot of question from same user, we will need to record current timestamp as a validation time.

Id	Type – int

	Extra Feature – auto_increment This is just question id for each question.
Question_type	Type –varchar This data will be used to distinguish true-false question or multiple choice questions.
Question	Type – varchar This data will have actual question that is displayed to ask player in the game.
Correct_answer	Type – varchar This data will have the correct answer to the question.
Option1,2,3,4 (combined to save space)	Type – varchar If the question is true false type, only 2 options will be given to the player. And there will be 4 options for multiple choice questions.
Answers	Type – varchar This data is the answer that comes from the player. We will be checking this answer with an actual answer.
Image_option1,2,3,4	Type – varchar We will put some images to help visually the player.
Points	Type – int

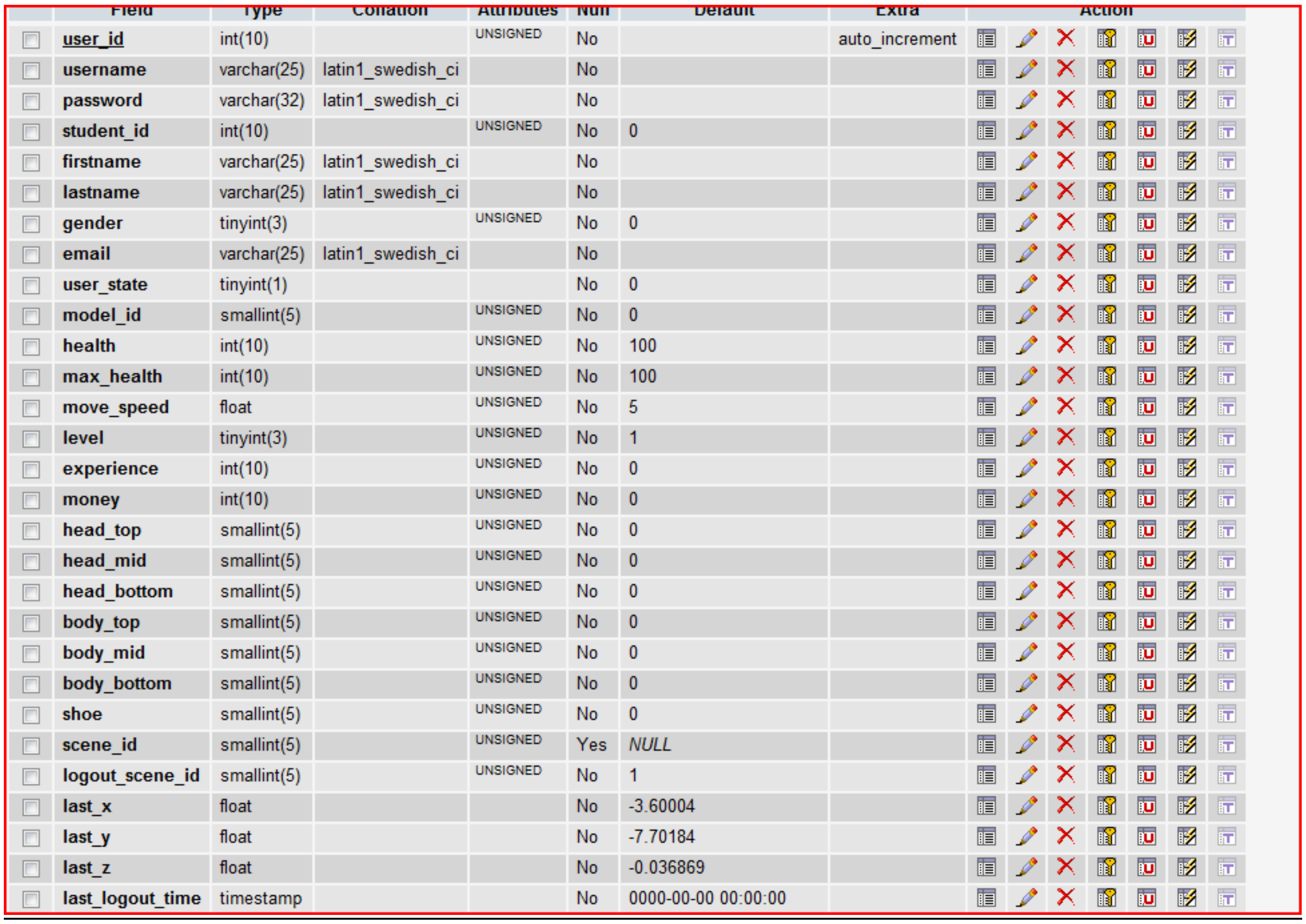
	This data is the score the player will get for answering the questions.
Username	Type – varchar This data is used to determine where to insert the score or points.
Timelimit	Type –int There will be time limit for each question. Usually, the player will only get very small amount of time.
Is_validated	Type –tinyint This type is like Boolean type because it will be true if the question is validated. On the other hand, the data will be false.
Level	Type –tinyint The data is for level the player before he or she can answer the questions.
Created_data	Type –date This data is used to specify the date when the question is created.
Last_edit_data	Type- date So, we can easily notice the time we modify the question.
Time_validated	Type – Timestamp (current_timestamp) When a player validates a question, this data

	will be stored current time.
Money_validated	Type- int When a player validates a question, he will gain money. That money will be in this data.

Table- user

Major Contributor – Tun Win

Minor Helpers – Alvin Wan, Calvin Kuang



	Field	Type	Collation	Attributes	Null	Default	Extra	Action
	user_id	int(10)		UNSIGNED	No		auto_increment	
	username	varchar(25)	latin1_swedish_ci		No			
	password	varchar(32)	latin1_swedish_ci		No			
	student_id	int(10)		UNSIGNED	No	0		
	firstname	varchar(25)	latin1_swedish_ci		No			
	lastname	varchar(25)	latin1_swedish_ci		No			
	gender	tinyint(3)		UNSIGNED	No	0		
	email	varchar(25)	latin1_swedish_ci		No			
	user_state	tinyint(1)			No	0		
	model_id	smallint(5)		UNSIGNED	No	0		
	health	int(10)		UNSIGNED	No	100		
	max_health	int(10)		UNSIGNED	No	100		
	move_speed	float		UNSIGNED	No	5		
	level	tinyint(3)		UNSIGNED	No	1		
	experience	int(10)		UNSIGNED	No	0		
	money	int(10)		UNSIGNED	No	0		
	head_top	smallint(5)		UNSIGNED	No	0		
	head_mid	smallint(5)		UNSIGNED	No	0		
	head_bottom	smallint(5)		UNSIGNED	No	0		
	body_top	smallint(5)		UNSIGNED	No	0		
	body_mid	smallint(5)		UNSIGNED	No	0		
	body_bottom	smallint(5)		UNSIGNED	No	0		
	shoe	smallint(5)		UNSIGNED	No	0		
	scene_id	smallint(5)		UNSIGNED	Yes	*NULL*		
	logout_scene_id	smallint(5)		UNSIGNED	No	1		
	last_x	float			No	-3.60004		
	last_y	float			No	-7.70184		
	last_z	float			No	-0.036869		
	last_logout_time	timestamp			No	0000-00-00 00:00:00		

In my opinion, this table is the most important table for the whole database. When a user starts loading to play debugger game, the user will be typing his username and password. Meanwhile, server team will be retrieving the data from our database by using information which is provided by the player or user. Then, the information from a player should be matched to our user table to start playing a game. Otherwise, error message will be printed out to retype the username and password.

In addition, this table also contains user information such as first name, last name, gender and email. These data can be used to retrieve user password if a player forget his or her password for this game. The rest of fields other than actual user information such as username and password will use to hold values for game user's information. For example, player might not want to go back to scene#1 if he ended up scene#3 at the last time he played. So, we need a field to retrieve to put player's character to the last scene he played and that field will be last_logout_scene. After the player is successfully log into the last logout scene, the character will have same position x, y and z, health points, money and experience.

Columns from user table	Descriptions
User_id	Type- int Extra feature – auto_increment This data is used to connect with other tables as needed. If foreign keys are needed, user_id will be referenced.
Username	Type-varchar This data is used to store username which will be entered in login screen when player started loading the game.
Password	Type-varchar This data is used to store password which will be entered in login screen when player started loading the game.
Student_id	Type-int This data is used to retrieve password or username and checked to make sure username is valid.
Firstname	Type-varchar This data is used to retrieve password or username, and checked to make sure username is valid.
Lastname	Type-varchar This data is used to retrieve password or username and checked to make sure username is valid.

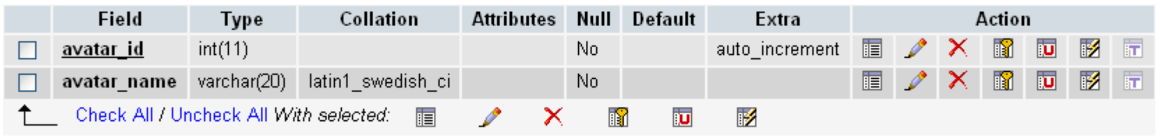
Gender	Type-tinyint This data is additional information to check username.
Email	Type-varchar This data is used to retrieve password or username and checked to make sure username is valid.
User_state	Type-tinyint
Model_id	Type-smallint The data is to make sure client team pull up a right character.
Health	Type-int This data is used to store the player health since the last time he or she played.
Max_health	Type-int This data will set the limit for the player health. Max health will be 100.
Move_speed	Type-float This data will be increased by the percentage only if the player buy an item from general_item table.
Level	Type-tinyint User's level will be stored in this data

Experience	Type-int User’s experience will be stored in this data
Money	Type-int User’s money will be stored in this data.
Head_top	Type-smallint When we load the character from the last time that the player plays, we need to store the top part of the character’s head from this data. So, we can display it properly when user plays our game again.
Head_mid	Type-smallint When we load the character from the last time that the player plays, we need to store the middle part of the character’s head from this data. So, we can display it properly when user plays our game again.
Head_bottom	Type-smallint When we load the character from the last time that the player plays, we need to retrieve the bottom part of the character’s head from this data. So, we can display it properly when user plays our game again.
Body_top	Type-smallint When we load the character from the last time that the player plays, we need to retrieve the top part of the character’s body from this data. So, we can display it properly when user plays our game again.

Body_mid	Type-smallint When we load the character from the last time that the player plays, we need to retrieve the mid part of the character's body from this data. So, we can display it properly when user plays our game again.
Body_bottom	Type-smallint When we load the character from the last time that the player plays, we need to retrieve the bottom part of the character's body from this data. So, we can display it properly when user plays our game again.
Shoe	Type-smallint When we load the character from the last time that the player plays, we need to retrieve the shoe of the character's body from this data. So, we can display it properly when user plays our game again.
Scene_id	Type-smallint This data is used to store all scene_id.
Logout_scene_id	Type-smallint When we load the character from the last time that the player plays, we need to retrieve logout_scene_id from this data. So, we can display it properly when user plays our game again.

Last_x	Type-float When we load the character from the last time that the player plays, we need to retrieve coordinate of X of the character from this data. So, we can display it properly when user plays our game again.
Last_y	Type-float When we load the character from the last time that the player plays, we need to retrieve coordinate of Y of the character from this data. So, we can display it properly when user plays our game again.
Last_z	Type-float When we load the character from the last time that the player plays, we need to retrieve coordinate of Z of the character from this data. So, we can display it properly when user plays our game again.
Last_logout_time	Type-timestamp Display the last time that a user plays our game.
Created	Type-timestamp When a new username is created, the timestamp will be stored into this data.

Table: avatar (Major Contributor: Alvin, Minor Contributors Tun and Calvin)



	Field	Type	Collation	Attributes	Null	Default	Extra	Action
☐	avatar_id	int(11)			No		auto_increment	
☐	avatar_name	varchar(20)	latin1_swedish_ci		No			

Check All / Uncheck All *With selected:*

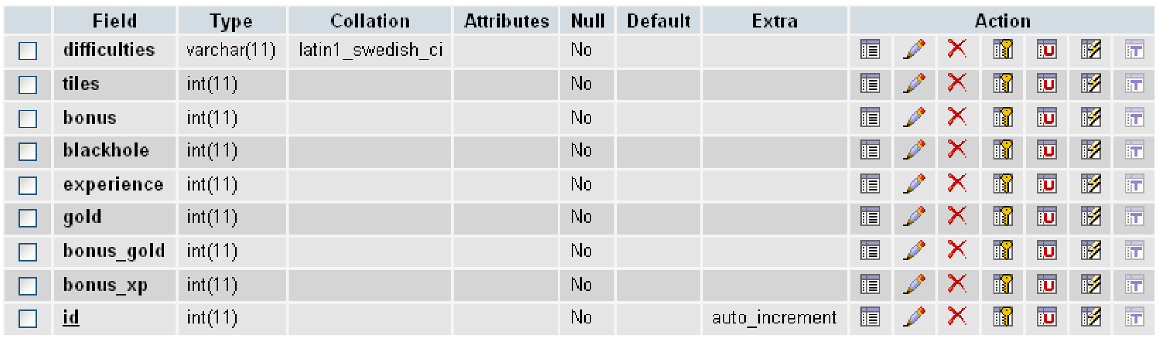
The purpose of having an avatar table is to be able to store the avatar's information. Avatar can be very complex, and as more and more customization is presented in the game, this table will become increasingly more important.

So far, we only store the avatar's name. But later, we can expand this table by introducing cloths, faces, hair style, color, and many other customizations, allowing Users to play their own personal and unique avatar!

The following is the field label on the left side, while its description is on the right.

Fields	Description
Avatar_id	avatar's ID int: - holds an ID to make each entry unique.
Avatar_name	the name of the avatar. varchar: - holds avatar's name as a string.

Table: board_game (Major Contributor: Alvin, Minor Contributors Tun and Calvin)



	Field	Type	Collation	Attributes	Null	Default	Extra	Action
☐	difficulties	varchar(11)	latin1_swedish_ci		No			
☐	tiles	int(11)			No			
☐	bonus	int(11)			No			
☐	blackhole	int(11)			No			
☐	experience	int(11)			No			
☐	gold	int(11)			No			
☐	bonus_gold	int(11)			No			
☐	bonus_xp	int(11)			No			
☐	id	int(11)			No		auto_increment	

Note: this board game table is mostly recycled from the previous game (Nurse Town by Zoran). However, we made changes to suit our needs like blackhole, gold and xp fields for our new version of the board game.

The main purpose of this board_game table is used to store the information from the board game component.

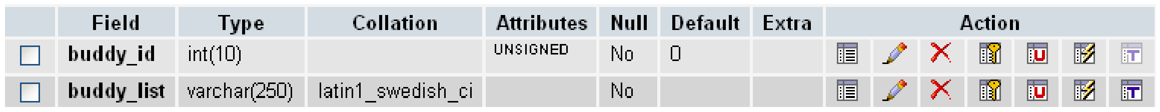
The following is the field label on the left side, while its description is on the right.

Fields	Description
Difficulties	records the different modes of the game (Easy, Medium, Hard). varchar:

	- holds a string, “easy,” “medium,” or “hard.”
Tiles	field records the number of tiles in the map because each map can vary in the number of tiles. int: - holds a number of tiles in the map.
Bonus	stores the bonus values that a user can achieve. int: - holds the bonus value of each entry.
Blackhole	keeps tracks of blackhole(s). int: - holds the number of blackhole(s) generated on each level of difficulty
Experience	an experience chart that can be obtained in a given level. int: - holds the experience reward of each entry.

Gold	a gold chart that can be obtained in a given level. int: - holds the gold reward of each entry.
bonus_gold	a chart listing the bonus gold for each level int: - holds the gold bonus of each entry.
bonus_xp	a chart listing the bonus xp for each level int: - holds the xp bonus of each entry.
Id	provides an id for each entry making each unique. int: - holds ID to make each entry unique.

Table: buddy (Major Contributor: Alvin, Minor Contributors Tun and Calvin)

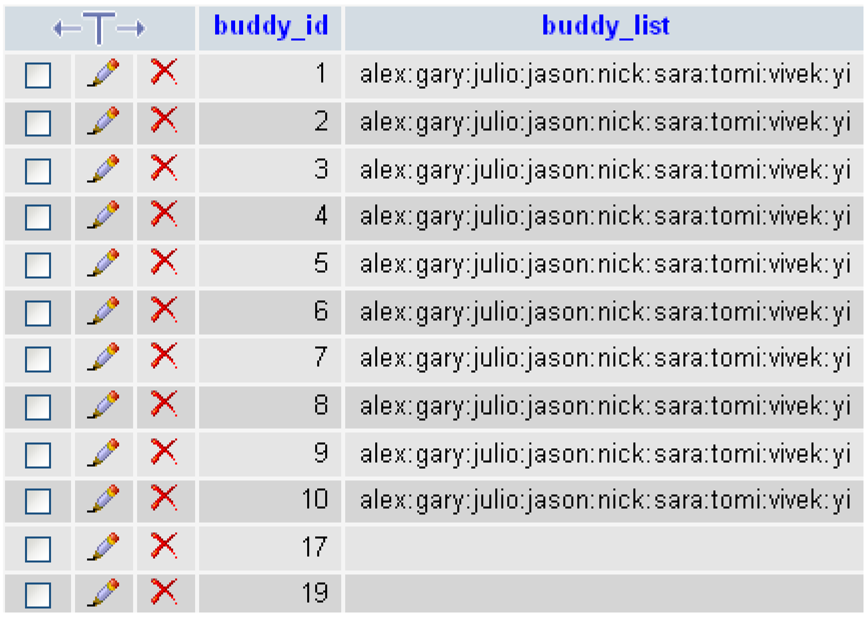


	Field	Type	Collation	Attributes	Null	Default	Extra	Action
☐	buddy_id	int(10)		UNSIGNED	No	0		
☐	buddy_list	varchar(250)	latin1_swedish_ci		No			

When a user adds a friend, there is actually a lot of underneath work. First, the User's Client requests the Server to store friend's name into the User's buddy_list in the database. Then, the Server communicates with the Database and stores the given friend's name into the User's buddy_list. Lastly, the Server responses back to the Client, so the User sees something like "Friend added successfully."

The following is the field label on the left side, while its description is on the right.

Fields	Description
Buddy_id	user's ID int: - holds Buddy_ID to make each entry unique.
Buddy_list	a list of User's friends, separated by a delimiter (in this case it's colon ":") varchar: - holds the entry's friend list as string.



←T→			buddy_id	buddy_list
☐	✎	✗	1	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	2	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	3	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	4	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	5	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	6	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	7	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	8	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	9	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	10	alex:gary:julio:jason:nick:sara:tomi:vivek:yi
☐	✎	✗	17	
☐	✎	✗	19	

Above is a sample of what a friend's list looks like in the Database. As we can see, each buddy_id has its own list. The list itself is a list of friends' names and is separated by a delimiter, a colon ":" in this case.

DATA BASE TEAM

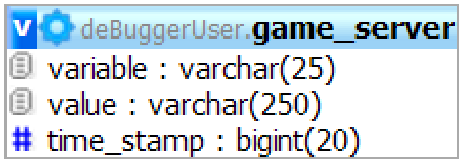
Database: **deBuggerUser**

Major contributor: Tun

Minor contributors: Calvin and Alvin

Table: **game_server**

Documentation: Calvin



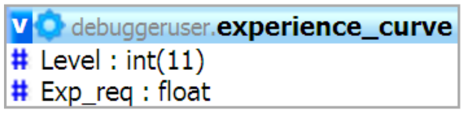
deBuggerUser.**game_server**
variable : varchar(25)
value : varchar(250)
time_stamp : bigint(20)

Figure 5f.1 Data type description show in Figure 5f.4.

The *game_server* table is reuse from the Nursetown database. It was created to facilitate features such as emergency shutdown, maintenance shutdown, and starting-up game server remotely. The *game_server* table has three fields such as variable, value, and time_stamp. The variable field is use to store the shutdown types and process id. The value field stores the shutdown flag such as 0 for no shutdown, 1 for emergency shutdown, and 2 for maintenance shutdown. The time_stamp field stores the shutdown time.

Table: **experience_curve**

Documentation: Calvin



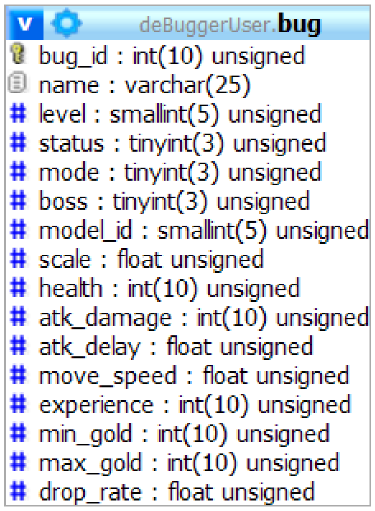
debuggeruser.**experience_curve**
Level : int(11)
Exp_req : float

Figure 5f.2 Data type description show in Figure 5f.4.

The *experience_curve* table is use to store the definition of levels in the game. The scale from one level to another level will increase based on the formula NewCap = (PreCap)+(PreCap*1/4). For example, if level 1 needs 5 experience points, then level 2 needs 6.25 experience points. The *experience_curve* table has all the levels and the experience points needed for each level. In order to go from one level to another level, the player needs to have certain amount of experience points. The level is very important for each player in the game because the players are not allowed to buy certain items or go the certain places unless they get to certain level. The player's level and experience points are store in the user table. When the player receives a quantity of experience points, it will check the *experience_curve* table to see if the player can go to the next level.

Table: **bug**

Documentation: Calvin


deBuggerUser.**bug**

```
bug_id : int(10) unsigned
name : varchar(25)
level : smallint(5) unsigned
status : tinyint(3) unsigned
mode : tinyint(3) unsigned
boss : tinyint(3) unsigned
model_id : smallint(5) unsigned
scale : float unsigned
health : int(10) unsigned
atk_damage : int(10) unsigned
atk_delay : float unsigned
move_speed : float unsigned
experience : int(10) unsigned
min_gold : int(10) unsigned
max_gold : int(10) unsigned
drop_rate : float unsigned
```

Figure 5f.3 Data type description show in Figure 5f.4.

The *bug* table is use to store all the bugs such as syntax bug, semantic bug, unknown bug, and trouble bug. The *bug* table also stores many different characteristic about each bug. This table has fields such as bug_id, name, level, status, mode, boss, model_id, scale, health, atk_damage, atk_delay, move_speed, experience, min_gold, max_gold, and drop_rate. The name field stores the type of bugs such as syntax bug, semantic bug, unknown bug, and trouble bug. The level field stores the require level of the player to play with that bug. The atk_damage field holds the amount of damage for each bug. In the game, player will gain a certain amount of gold by destroyed a bug or lost some health points if the bug is not killed. These are the information that gets from the bug table for all the bugs in the game. It will update the user table information for every bug the player plays.

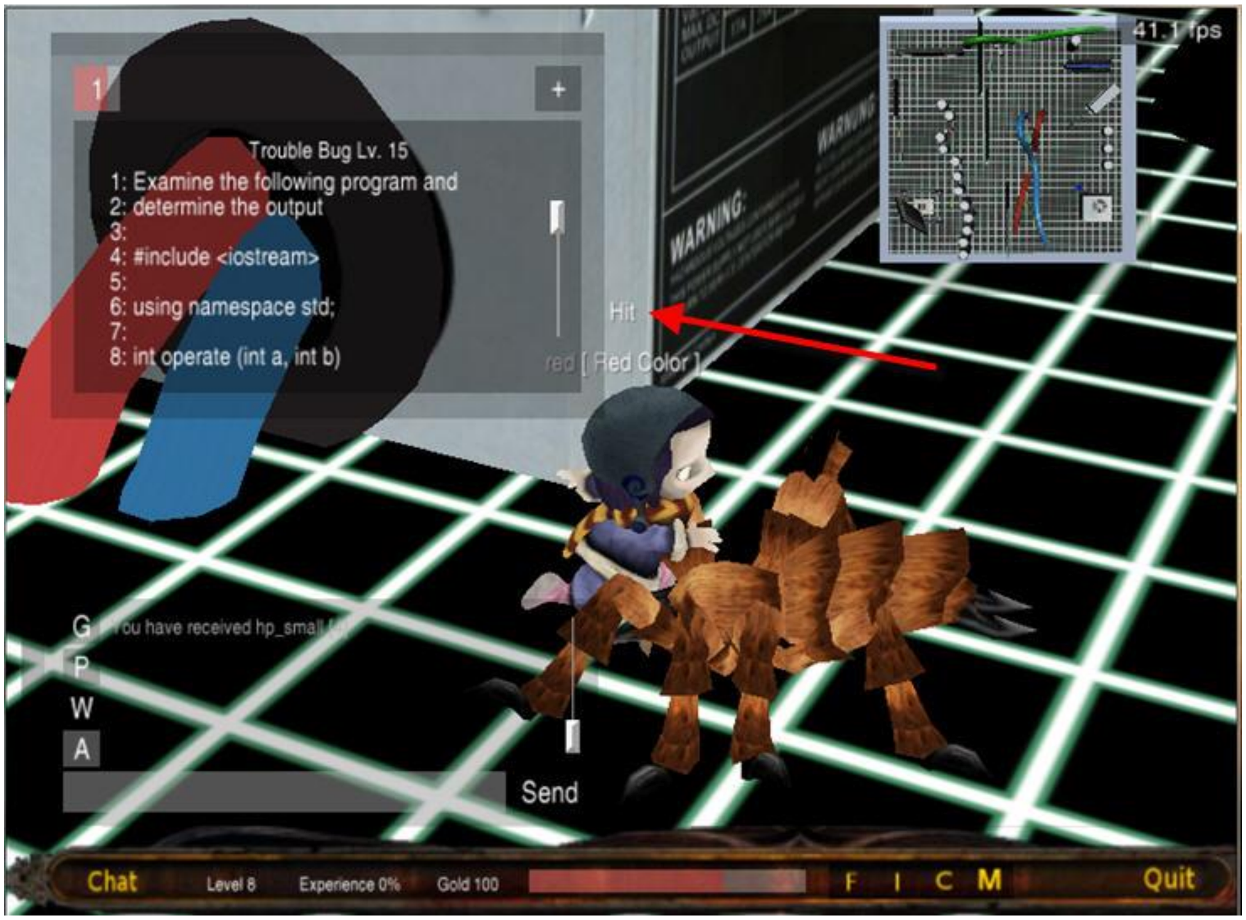
Data Type	**Description**
varchar	A variable-length string between 1 and 255 characters in length.
bigint	A 64-bit integer represented as 8 bytes.
int	A 32-bit integer represented as 4 bytes.
float	A floating- point number; 8-digit precision represented as 4 bytes.

smallint	A 16-bit integer represented as 2 bytes.
tinyint	A 8-bit integer represented as 1 byte.

Figure 5f.4

DAMAGEBUBBLE

In many MMORPGs, when a player is in combat the user will have to be alert to several different events happening at once. This becomes escalated in a game such as “deBugger” when the user has to think about how to solve an academic problem meanwhile keeping track of personal statistics. An important visual aide to help the player keep on top of everything that is going is the implementation of the damage bubble. With the damage bubble, text is displayed over the user's avatar to show how their character is being affected by enemy attacks. This helps during combat as it provides an extra cue and give the player either a sense of urgency or alert the player to the status of a battle. With having a damage bubble in close proximity of a character's avatar, the player does not need to constantly check their health bar. This prevents the user's eyes from jumping across the screen and lessens eye strain.



```
Trouble Bug Lv. 15
1: Examine the following program and
2: determine the output
3:
4: #include <iostream>
5:
6: using namespace std;
7:
8: int operate (int a, int b)
```

You have received hp_small

Figure 4.c.iv.2 - When a character takes damage, a "Hit" message will appear and alert the player.

The damage bubble is normally displayed over the character's name. After a small amount of the time, the damage bubble will disappear. This will allow a new damage to appear should the character experience damage once again.

DamageBubble – Understanding The Code

DamageBubble.py is located within the main\World\Bubble package. This file is in charge of creating and displaying a visual indicator of whether or not a player has taken damage.

DamageBubble Class

This creates bubbles displaying any damage a user is experiencing or evading during combat. This class also controls bubble features such where to display the bubble in the virtual world environment,what message to display, and how/when to make it appear.

Data Member

xObject	This object provides access to world variables for the current user.
damageBubbleSequence	Panda 3D Sequence Object. This provides a sequence of events for the bubble to perform.
lastDistance	This float variable holds the last distance.
lastPosition	This 3 point variable holds the last position.
textColor	This 4 point variable holds the color and alpha values for the text in the bubble.
dBubble	Panda 3D text node. This bubble holds the appropriate shadow.
dBubbleNodePath	This holds the path for dBubble.

Member Function

__init__(self, xObject)

This function initializes xObject and the default values for the sequence, colors, dBubble, and dBubbleNodePath. It then adds dBubbleRoutine to taskMgr.

setText(self, msg)

This function checks to see if damageBubbleSequence needs to be paused. Afterwards, it sets a new damageBubbleSequence and starts it.

dBubbleRoutine(self, task)

This function takes a task as an argument. It starts by getting new drawing positions if they are different from lastPosition.

```
        if (self.xObject.getPos() != self.lastPosition):
            center = self.xObject.getObject().getChild(0).getBounds().getCenter()
            radius  = self.xObject.getObject().getChild(0).getBounds().getRadius()
```

It then sets these positions to dBubbleNodePath and updates lastPosition to reflect the new values.

```
            self.dBubbleNodePath.setPos(self.xObject.getPos())
            self.dBubbleNodePath.setZ(self.xObject.getObject(), center.getZ() + radius * 1.3)

            self.lastPosition = self.xObject.getPos()
```

It then sets a new scale to dBubbleNodePath and updates lastDistance if these have changed.

unload(self)

This function removes the dBubbleRoutine from taskMgr. It then clears dBubbleNodePath. Afterwards, if there's a damageBubbleSequence the functions pauses the sequence.

INVENTORY

In many MMO games, a player will be able to add, store and use items at their own discretion. Many of these items will normally benefit the player through aiding statuses, adding revenue or other general game statistics. When a user wants to access these items they normally do it through an inventory list. This list is a GUI menu that appears and disappears at the user's discretion, allowing them to hide it when they don't want the menu hampering their field of view or field of depth in the game or allowing them to view it when they want to use an item.

The inventory list in the “deBugger” has simple properties that are characteristic to many inventory menus. The inventory menu in “deBugger” has a title bar that can be dragged around the screen. This allows the player to place the inventory menu in a less obstructive position at any given time. This can be helpful if there are other activities going on in the virtual environment that the user would like to keep an eye on. In this situation, the player can simply left click and hold the mouse button on the title bar. While continuing to hold the left click button, they can drag the menu around the screen. When they have moved the menu to a position on the screen that they prefer, they can let go of the left click on the mouse. From here, the inventory menu will stay in this position on the screen until they decide to move it again.

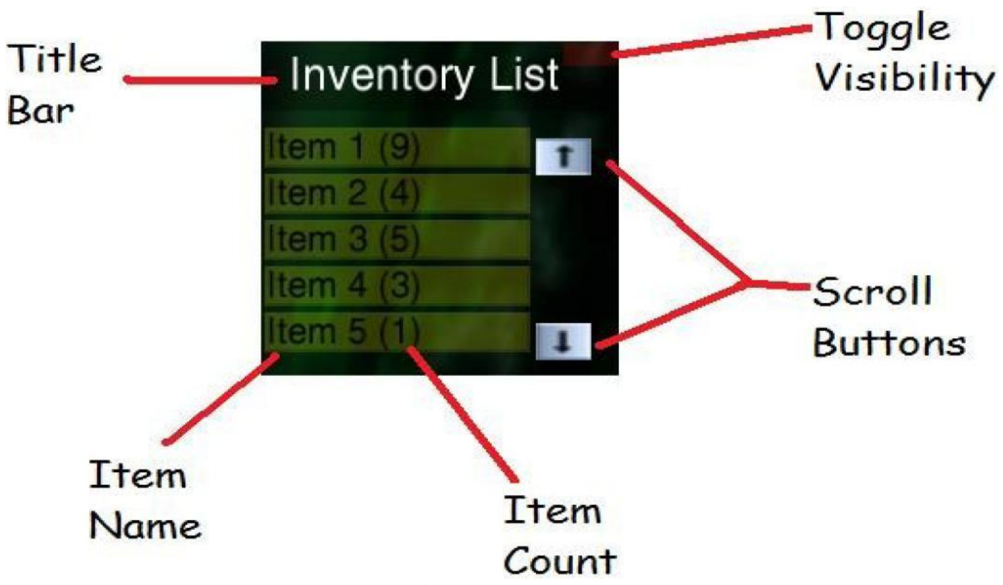


Figure 4.c.vi.1 - Inventory Menu Features

The inventory menu also has toggling functionality. The inventory menu can be toggled by either clicking the red box in the upper right hand corner (hiding only), pressing “I” on the main bar or by pressing a combination of “CTRL+I” once in the virtual environment. Having the menu toggle allows it to be hidden when the user doesn't need to view the list, giving the player more view access of the environment. When they need to view the menu, they can toggle it back on and the menu will be in the same position where it was toggled off.

When a user has more than the maximum number of items that can be displayed at one time (currently 5 items at the time of this writing), the user will be able to use the scroll the buttons. The scroll buttons are on the right edge of the inventory menu. The button with an arrow pointing up signifies scrolling up and the button with the arrow scrolling down signifies scrolling down. The scroll buttons will allow the user to view more items than those currently displayable on screen by going the list of items sequentially and removing older entries from the display. When the user has hit the end or beginning of a list, the corresponding scroll button will then stop scrolling the list.

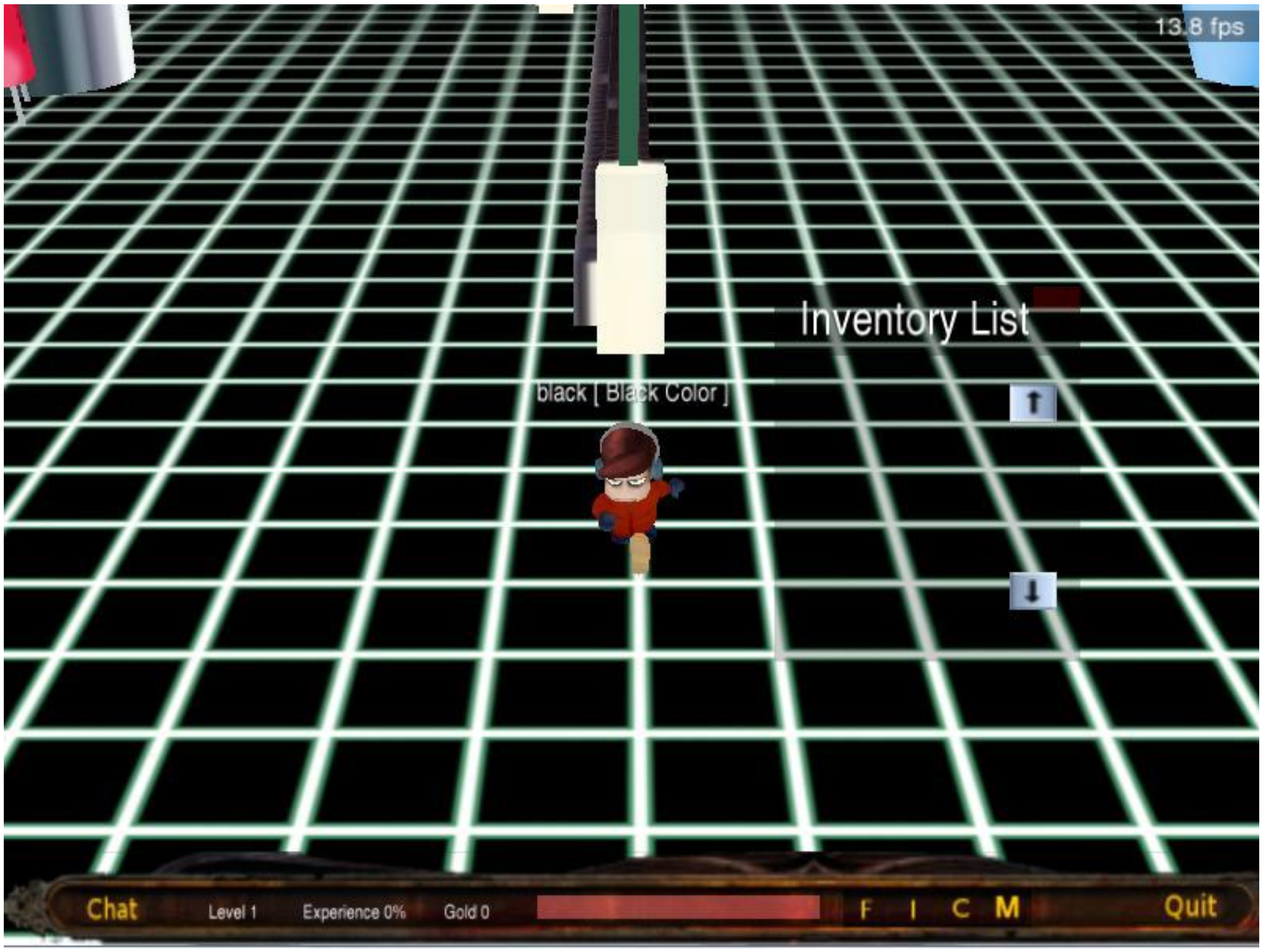


Figure 4.c.vi.2 - When a user has no items, they will see an empty inventory menu.

The items are displayed on the inventory menu as buttons. The buttons contain information for the user such as the item's name and the item's count value, which displays the quantity of the item which the user currently has. When the user wants to use an item, they can click on the item button. After clicking on the button, a prompt will display to the user for whether or not they want to use the item. After verification, the item will one less value in the item count field. If there are no more items, then the item will disappear from the inventory list. At the time of this writing, the only usable items are health restoration items.

Being able to easily access the items the player has collected allows the game to feel more mainstreamed. With quick access, the player doesn't have to return back to a neutral point or navigate several menus to view their items.

Inventory – Understanding The Code

The Inventory.py file contains code used for displaying the user's item catalog. Through the classes and methods provided in this file, the user is able to interact with a GUI menu which makes all of the user's items available. Other functionality in this file includes changing the current view of the items displayed, updating items, item usage, and more. Through it's different classes, Inventory.py is able to contain item information and update it to the user.

ItemObject Class

The ItemObject class is a container class. An ItemObject holds all the item properties for a specific item. These properties are specific for the user currently holding the item.

Data Members

itemId	Stores an item's unique ID.
itemLevel	Stores an item's level property.
itemCount	Stores the amount of the item that the user currently has.
itemName	Stores the item's name.
itemPrice	Stores the price of the item.
speed	Stores the speed value of the item.
hpRestore	Stores the amount of HP that the item restores.

shield	Stores the shield value of the item.

Member Functions

__init__(self, id, level, count, name, price, speed, hpRestore, shield)

Class constructor. This function takes in item properties: id, level, count, name, price, speed, hpRestore, shield. The values of these arguments are assigned to local data members.

unload(self)

Resets item id to 0. Currently a place holder function for possible extensibility.

InventoryObject Class

This class handles button functionality for each individual item. This class allows an item's details to be displayed as a button. An InventoryObject is then used by the Inventory class to be displayed as part of the entire Inventory menu. This class also deals with how the user interacts with each InventoryObject (button).

Data Members

itemObject	Stores an ItemObject for the class to use. Contains information that the class can display to the user.
main	Provides access to other classes from main.
inventoryButton	Panda3D button object. Holds button size as well as position. Displays item

	name and count. Calls verifyItemUsage function when clicked on.
verifyDialog	Panda3D dialog object. Displays a verification dialog, asking the user whether or not they want to use the item that was just clicked on. Possible answers are "Yes" or "No". Calls verifySelect function once a choice is made.

__Member Functions__

__init__(self, main, itemObject, inventoryColumnPosition, inventoryTopRowPosition)

Class constructor. This function takes in main and itemObject and sets their values to local data members. The inventoryColumnPosition parameter assigns a two dimensional x-position (horizontal) for the button to be written onto the screen. The inventoryTopRowPosition parameter assigns a two dimensional y-position (vertical) for the button to be written onto the screen.

verifyItemUsage(self)

Creates a YesNoDialog which displays a prompt to the user whether or not they want to the class' item. This dialog is assigned to the verifyDialog data member.

verifySelect(self, arg)

This function takes in arg as argument. If arg is true (in which case "Yes" was selected), then the item will be used and information will be sent to the server to reflect the item amount reduction. This is done by creating a message with 'itemId' as a key and the item's ID as the value. Another message is created where 'count' is the key and '1' (the amount of item that has been used) is the value. These two messages are concatenated and assigned to local variable rContents. These messages are sent along with a constant to cManager's sendRequest, where a notification will be sent to server. Afterwards, a check is made to see if the item can restore health. If this is the case, then a call is made to use the health item. The function ends with a call to cleanup and remove the dialog from the screen.

Inventory Class

The inventory class handles most of the functions in charge of drawing the menu interface. This includes holding all the inventory buttons, title bar, scroll buttons, as well as other items. The inventory class also manages most of the functionality. This includes conducting how the buttons are drawn, the scrolling functionality, the menu toggling, how an item is used, updating the menu when an item is obtained or used, as well as other functionality.

Data Members

inventoryObjectArray	Keeps a list of InventoryObjects to display on to the menu.
inventoryListArray	Keeps a local list of itemObjects that the user has as an inventory catalogue.
hasAcceptedInventoryList	Flag that is set to true when inventoryListArray is initially populated from the server. This flag prevents inventoryListArray to be populated from the server more than once, prohibiting the list the potential to have duplicates.
hasSetUpInventoryTextNodes	Flag that is set to true when inventoryObjectArray has been set and initialized with inventoryObjects based on the inventoryListArray that is set from the server.
mainFrame	Complete 2D frame printed onto the screen. This frame is where most of the inventory menu writes onto and holds most of the inventory menu content.
titleBar	Complete 2D frame representing dark border near top of main frame.
closeButton	Complete 2D button colored red and position near top right corner of main frame. Offers control for toggling menu visibility.

downButton	Complete 2D button with image of down arrow attached over. This button is positioned in the lower right of the main frame. Clicking this button will call the scrollListDown function.
upButton	Complete 2D button with image of up arrow attached over. This button is positioned on the right side of the main frame. Clicking this button will call the scrollListUp function.
main	Provides access to other classes from main.
headingNP	Writes the menu title text to the main frame.
headingNP2	Writes a loading message text to the main frame.
arrayIndex	Keeps track of which ItemObject in inventoryListArray will be displayed at the beginning of inventoryObjectArray. This is used for scrolling the menu items up and down.
inventoryTopRowPosition	Holds the y-value for where the first InventoryObject should begin being written onto the menu.
inventoryColumnPosition	Holds the x-value for where InventoryObjects will be written on the menu.

Member Functions

startDragMode(self, param)

Allows mainFrame to be moved around the screen on mouse press.

stopDragMode(self, param)

Places mainFrame on last moved position on mouse release.

toggleVisibility(self)

Switches mainFrame from visible to hidden or back. This done based on the state of mainFrame's current visibility.

```
        if (self.mainFrame.isHidden()):
            self.mainFrame.show()
        else:
            self.mainFrame.hide()
```

acceptInventoryList(self, serverInventoryList)

Makes a request to the server to accept the user's inventory catalog. This request is only made based on checking a flag to see if the request has already been made.

```
        if(self.hasAcceptedInventoryList == 0):
            rContents = {}
            self.main.main.cManager.sendRequest(Constants.CMSG_REQ_INVENTORY,rContents)
            self.hasAcceptedInventoryList = 1
```

addItemToInventoryList(self, id, level, count, name, price, speed, hpRestore, shield)

Takes item attributes and assigns them to a local itemObject. From here, it scrolls through an existing list of itemObjects checking to see if this object is already in the list.

```
            for index in range(0,len(self.inventoryListArray)):
                if(self.inventoryListArray[index].itemId == itemToAdd.itemId):
```

If the item is in the list, then the item's amount is incremented. If the item is not in the list, then the is added to the list.

```
                    self.inventoryListArray[index].itemCount += itemToAdd.itemCount
```

```
                hasAppended = 1
                break
        if(hasAppended != 1):
            self.inventoryListArray.append(itemToAdd)
```

If no list exists, then a list is created with the item in the list. After the item has been added then the screen will refresh if inventoryObjectArray has been set up and initialized.

useHealthItemFromInventoryList(self, healthItem)

Takes in ItemObject healthItem as a parameter. This function goes inventoryListArray, comparing the ID of healthItem with that of the IDs of the user's itemObjects.

```
    for index in range(0,len(self.inventoryListArray)):
        if(healthItem.itemId == self.inventoryListArray[index].itemId):
```

Once a match is found, the user's Hp get appropriately incremented. This function will then decrement the item's stock by one.

```
            self.main.charHero.healthPoints += self.inventoryListArray[index].hpRestore
            if(self.main.charHero.healthPoints > self.main.charHero.maxHealthPoints):
                self.main.charHero.healthPoints = self.main.charHero.maxHealthPoints
            if(self.inventoryListArray[index].itemCount > 1):
                self.inventoryListArray[index].itemCount -= 1
                self.refreshButtonsOnScreen(0)
```

If this causes the item's stock amount to go to 0, the item will be deleted from the list. After this, the list of InventoryObjects will be refreshed on the menu.

scrollListUp(self)

This function handles scrolling the list up. Graphically, each item will be replaced with the item one position closer to the beginning of the inventoryListArray. If there are less than the total number of InventoryObjects that can be written to menu, then the menu is updated through writeInventoryToScreen. Otherwise, the menu is cleared of all existing InventoryObjects, arrayIndex is decremented by one (keeping a non-negative value) and the menu is updated through writeInventoryToScreen.

scrollListDown(self)

This function handles scrolling the list up. Graphically, each item will be replaces with the item one position closer to the end of the inventoryListArray. If there are less than the total number of the InventoryObjects that can be written to menu, then the menu is updated through writeInventoryToScreen. Otherwise, the menu is cleared of all existing InventoryObjects, arrayIndex is incremented by one (while keeping below the max inventoryListArray size minus the max total number of inventory objects that can be displayed on screen) and the menu is updated through a call to writeInventoryToScreen.

refreshButtonsOnScreen(self, deletedItem)

This function takes in a '0' or '1' int value in deletedItem as a parameter. This function refreshes the current InventoryObjects written to the menu after an itemObject has been added or used. It starts by checking if the total number of InventoryObjects in inventoryObjectArray is less than the max number that can be written to screen. If this is the case, then all the InventoryObjects will be cleared and the menu will be updated through a call to writeInventoryToScreen. Otherwise, all existing InventoryObjects will be cleared followed by a check to see if an ItemObject was deleted. If it was, then arrayIndex will be moved up a position. Afterwards, the menu will be updated through writeInventoryToScreen.

setupInventoryTextNodes(self)

This function makes a check to see if hasSetupInventoryTextNodes is 0. Upon a successful check, this function sets arrayIndex to the starting inventoryListArray position of 0, along with giving starting default values to inventoryTopRowPosition and inventoryColumnPosition. It then initializes inventoryObjectArray to an empty list before updating the menu through a call to writeInventoryToScreen. After this it sets

hasSetupInventoryTextNodes to 1 and removes headingNP2 from the menu.

writeInventoryToScreen(self)

This function goes through the inventoryListArray beginning at arrayIndex. Based on the size of inventoryListArray, this function will traverse the inventoryListArray until either at the end of inventoryListArray or when it has reached the max number of displayable InventoryObjects. Whilte traversing, this function takes ItemObjects from inventoryListArray and uses them to create InventoryObjects to be written to the menu.

```
      self.inventoryObjectArray.insert(inventoryObjectIndex,                                    InventoryObject(
self.main,self.inventoryListArray[inventoryListIndex],
           self.inventoryColumnPosition,inventoryRowPosition))
      self.inventoryObjectArray[inventoryObjectIndex].inventoryButton.reparentTo(self.mainFrame)
```

It then inserts these InventoryObjects into inventoryObjectArray. In the scenario that an item's name is too long, it calls splitName.

splitName(self, name)

This function takes a String name as a parameter. It cuts the name down to the first third and returns the new, shrunken name.

unload(self)

This function clears all InventoryObjects and destroys the mainFrame. This is used when clearing out the inventory class instance.

NAME BUBBLE

What makes online games unique from other types of games is the large amount of different players that can be playing simultaneously. For online games to help users identify each other quickly is through the use of visual tags which display a user's name. This is not unlike when a person goes to a convention where they are more than likely to meet a large group people they don't know. In a situation such as at a convention, several people are encouraged to wear name tags to help identify one another and make socializing an easier and sometimes   more fun experience.

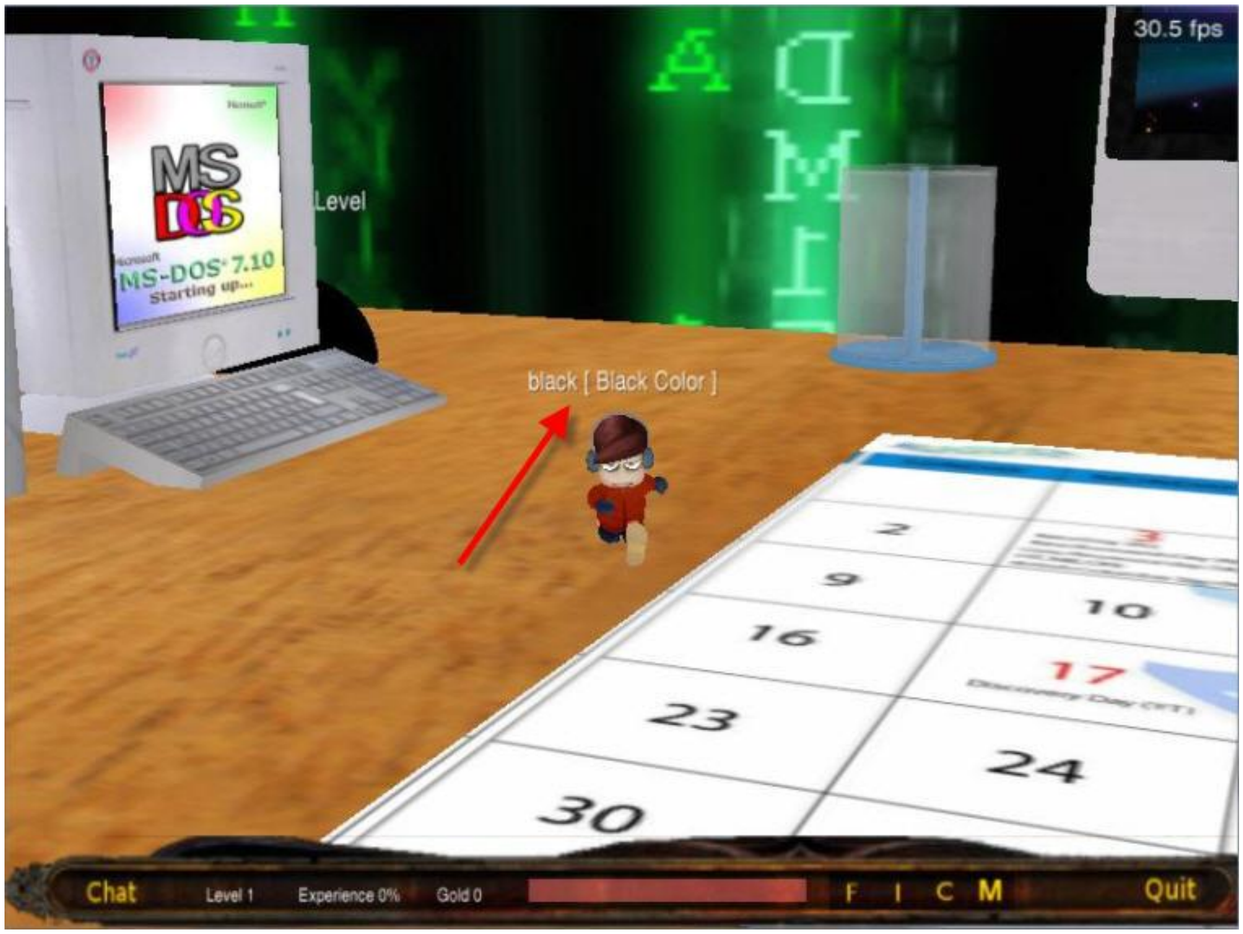


Figure 4.c.iv.3 - A character can be identified through the name displayed over their avatar.

On a similar note to that of name tags at a convention, a character's name is displayed over the avatar's head in the game. The feature implemented to allow this functionality is called the name bubble. The name bubble is used for players to identify characters and other players while in the virtual world environment. What makes this unique from most other bubbles is that the name bubble is persistently displayed.

The name bubble is also helpful for distinguishing characters that may look similar or even identical. This may be the case in a situation such as being a low level character with limited customization ability. This may also be the case when some characters find that they have similar customized options.

Through the name bubble identification and socializing becomes easier. An easy and friendly social environment is fundamental for keeping a player interested in MMO games. With a good social environment, players may be willing to spend more time in “deBugger” learning and having fun simultaneously.

Name Bubble – Understanding The Code

NameBubble.py is located within the main\World\Bubble package. This file is in charge of displaying user names above the character's avatar.

NameBubble Class

This class creates a name bubble to display the character's name graphically. This is done primarily by getting the character's coordinates and writing their display name into the environment for others to see.

Data Members

xObject	This object provides access to world variables for the current user.
lastDistance	This float variable holds the last distance.
lastPosition	This 3 point variable holds the last position.

textColor	This 4 point variable holds the color and alpha values for the text.
nBubble	TextNode object that holds user's name.
nBubbleNodePath	Holds path for nBubble.

Member Functions

__init__(self, xObject, displayName = "")

This function takes in an xObject and String displayName as parameters. It sets xObject locally and initializes lastDistance and lastPosition. It then sets textColor and creates nBubble. Afterwards it will assign text to nBubble based on displayName. Following this it sets more nBubble display properties that will give the bubble any graphical details, if necessary. It then creates nBubbleNodePath and adds nBubbleRoutine to taskMgr.

nBubbleRoutine(self, task)

This function takes a task as an argument. It starts by getting new drawing positions if they are different from lastPosition.

```
if (self.xObject.getPos() != self.lastPosition):
    center = self.xObject.getObject().getChild(0).getBounds().getCenter()
    radius  = self.xObject.getObject().getChild(0).getBounds().getRadius()
```

It then sets these positions to nBubbleNodePath and updates lastPosition to reflect the new values.

```
    self.nBubbleNodePath.setPos(self.xObject.getPos())
    self.nBubbleNodePath.setZ(self.xObject.getObject(), center.getZ() + radius)

    self.lastPosition = self.xObject.getPos()
```

It then sets a new scale to nBubbleNodePath and updates lastDistance if these have changed.

unload(self)

This function removes the nBubbleRoutine from taskMgr. It then clears nBubbleNodePath.

MINIMAP.PY (WRITTEN BY YI LU)

1. **Class**

class MiniMap(DirectObject):

This class object is creating in Gui.py constructor. It creates a mini-map and a point that represent the actor on the mini-map. Both mini-map and point are objects also know as model. Each time actor login or go to another map, mini-map and point will be recreated. It require access of current actor position and map ID through World.py which is the parent class to find out which map is the actor on and its current position by use task which run all the time. Actor's 3D environment position will convert to point position on the 2D mini-map. Map ID use to switch map.

2. **Function**

def isSwitchedMap(self):

It called inside updatePosition() task to check if need to switch map by matching the saved map ID and the current map ID. If they are the same return false, else save current map ID, call switchMap() function, and return true.

def switchMap(self):

It will remove both mini-map and point by removeNode() and recreate them by addMinimap() and addPoint(), since object has layering so biggest object first smallest object last. It also needs to check which map and pass the correct arguments.

def updatePosition(self, task):

This task function set position for point, simulating movement of the point. It checks the map before getting the current actor's position. It then call the convertStartPoint() function to get the 2D position from 3D position.

def addPoint(self, (actorX, actorZ, actorY)):

It creates a point of given actor's position on mini-map. The point is a plane object with a blue dot texture. It calls the convert point function to get the 2D position for the mini-map.

def addMiniMap(self, mapName = None, mapPos = (0,0,0), mapHeight = 0, mapWidth = 0):

It creates a mini-map of given map name position and display size. The min-map is a plane object with a given map image texture.

def convertStartPoint(self, actorX, actorZ, actorY):

It converts given actor's 3D environment position (x, y, z) to point position on 2D mini-map. When call this function, arguments are given in the order of (x, y, z), in 3D x and y is on the ground and z is point to the sky, in

2D x and z is on the mini-map and y is point to the user, so when convert 3D to 2D, we have 3D_x to 2D_x, 3D_y to 2D_z, 3D_z to 2D_y, that way 3D_z and 2D_y can be ignore.

3. Position the point on the map at the correct location and speed

To position the point at the correct location is hard. [I don't have a good way to do it, if you have time please find a better way to do this part, thank you] There are some variable affect the position of the mini-map and the point.

***self*.mapPos1 = (12.5, 55, 10.5)**

(x is 12.5 left right, y is 55 point to you, z is 10.5 up down) it is at upper right corner

***self*.mapHeight1 = 5.9**

***self*.mapWidth1 = 9.7**

They are height and width of the mini-map

***self*.map1PositionReduceRatio = 7.0**

Since in 3D environment character movement will cover more distance, so we need to reduce the movement distance for 2D map, bigger world will need to reduce more, this affect the movement speed

convertedPoint = ((13.4 + (actorX / *self*.map1PositionReduceRatio)), 60, (11.5 + (actorZ / *self*.map1PositionReduceRatio)))

((13.4 + …), (60), (11.5 + …))

This is the position of the point

In order the set the correct location and movement speed, I will first find a location on the world and on the map and move the map or the point to match the location. Then I will match the speed, depends on the height and width, speed can be match only height or width. After that I will change the height or width to match the speed.

4. For the future semester

There is still some hard coded stuff in Minimap.py. For example, I didn't make the mini-map's point position to a variable, I leave it hard coded because we only need to change position to either the map or the point on the map. If you like make them a variable ((13.4 + …), (60), (11.5 + …)) is the hard coded position of the point. There will be more map later on so all variable related to positioning should put them into list or hash map. For example, ***self*.mapHeight1, *self*.mapHeight2, *self*.mapHeight3,… into *self*.mapHeight[0], *self*.mapHeight[1], *self*.mapHeight[2].** It will be nice to have auto positioning, I am out of time but I was thinking it can be done. First make auto resize of all map that come with different size. Next auto adjust the speed to each map. If these two function work, we don't need to position the character to any landmark just put it to the corner of the map. For future idea, I would like to have mouse control interacting with mini-map. [mousepicker] It means when player click on a location of the mini-map the character will go to that location on the 3D world. Next is display. How much do you want to display?

Other player? Bug? NPC? That might need to work with other teammate who does server and database, be careful of tightly couple.

CHARACTER.PY (GARY NG)

Character.py

As mentioned in Chapter 4, every player will begin with a character that they will use throughout the entire game to help accomplish various goals and tasks. Each character will be unique, in a sense, that they will be having different items, avatar, attributes and many more.

In order for this to happen, a character instance must be created within the game, both frontend and backend. This is where the *Character* module comes in, which will be further explained in greater details below.

Getting Started

In *Python* and like every other programming language, you would first have to import a few modules to be able to use existing and user defined external methods. The following module as shown below imports the *randint* method from the module called *random*.

This simple method is mainly used to randomize a set of designated values. Later on, you will see it being used to calculate the chance at which a player can successfully dodge an attack.

As you've probably noticed from previous sections, the following modules below are being imported, not from *Python*, but from *Panda3D* to be used within *Python*.

```
from direct.gui.DirectGui import *
```

The purpose of the *DirectGui* module in this *Character* module is used to allow us to create and display pop up messages, but at the same time, this module brings in other functionalities to create simple and even more sophisticated menus and buttons. The *Actor* module is one of most essential modules to allow the creation and loading of models such as a character to be rendered on the screen. And finally, the *Interval* module, which is used to create timers, sequences, and, of course, intervals for situations where timing is needed to perform some particular tasks.

Other modules bundled with Panda3D are called *PandaModules*, which contain a wide variety of helper functions that are used to perform other specialized tasks.

```
from pandac.PandaModules import BitMask32

from pandac.PandaModules import
CollisionHandlerQueue
```

The *PandaModules* shown above are mainly used for adding collision support into the game. The *BitMask32* module is used for adding and modifying a particular model's bitmask, which determines what a particular collidable object can collide with depending on the object's bitmask value. All the modules beginning with *Collision* are required to add collision into the game, which will be explained further later on. The *Point3* module is mainly used to allow the use of methods for calculations between two points. Although a position can be stored as a tuple in *Python*, you will not be able to use various methods to extract a particular component or being able to calculate the distance between the two points using simple mathematical operators.

The remaining modules below are written entirely by us, which are not part of the *Python* and *Panda3D* module suite.

```
from common.Constants import Constants

from common.DatabaseHelper import *

from common.Entity import Entity
```

These modules are mainly created and used to help extend the functionality of this *Character* module, which will be explained in greater details when we get there. And moving on, we will now get started on the important functionality of the remaining of this *Character* module.

Character Class

As with every class in *Python*, this is how you would start off creating a class. The *Character* class isn't just a regular, but an inherited class from the *Entity* class.

The *Entity* module has been imported from above, which basically acts as an abstract class that contains several accessor and mutator methods to retrieve important information such as the position, level, health points, heading direction and other character information. Any other classes that want to retrieve necessary information can easily do so with the use of these methods.

Initialization

As with every other class, there is a constructor that will be used to initialize certain variables that will be used within this class to serve as member variables. As shown below, you might notice that the constructor looks very different than other programming languages.

This constructor is always called whenever a *Character* object is instantiated. The *self* parameter is used to show that this is a class method. The *world* parameter is used to accept the object *World*, which is the location at which a *Character* object is created. While doing so, it passes a reference of *World* into the *Character* class, so that it can access any other variables and methods that may be located in the *World* class as such can be seen below.

```
self.model_id = self.world.charData['avatar_id']

self.level = self.world.charData['charLevel']
```

Other important variables in the *Character* class that are used to distinguish between this character and another are shown below.

```
self.object_id = self.world.charData['user_id']
```

As another reminder, the keyword *self* is used to show that this variable is a class variable. As you can see from the code snippet, *self.object_id* is a variable that stores a unique identifier to distinguish between different objects, which, in this case, the character. The variable called *self.unitType* is used as a secondary identifier to show that this is a character type object based on the value 0.

Every character, whether it's you or your fellow friend's character has a *Battle System* instance object stored within each *Character* class, which will be explained in greater details later on. The way how the *Battle System* is declared is shown below, which passes a reference from *World* and the current character into it. To learn more about the *Battle System*, please look further into the section called *Battle System*.

Other variables to note are these so-called factor variables as shown below.

```
self.animSpeedFactor = 0.25

...
```

You may be wondering what exactly are these values and where are they derived from. These values are arbitrary values that are used as constants to make fine adjustments for certain tasks. The *self.turnFactor* variable is set to 100 because this particular variable is used as part of some formula when rotating a character being rendered on the screen. The character movements in this game are so-called frame-rate independent. This means that even a player with a slow machine can move at the same speed as another player with a faster machine. This does not guarantee smooth movements but to ensure that if a character was to move to a certain position, no matter how slow your machine is, the character will eventually get to that destination at the same duration of time as if it was moving smoothly. To accomplish this, the character would look like it's jumping to these destination points instead of walking. This same concept also applies for the variable *self.moveFactor*.

Character Loading

The loading process of a character model is very simple and doesn't require a lot of effort. In *deBugger*, there is a module that has been imported called *DatabaseHelper*. This module contains a user defined method called *dbSelectRowByID* as shown below.

```
result = dbSelectRowByID('avatar', 'avatar_id'
```

The *dbSelectRowByID* method takes in exactly 3 parameters—table name, attribute and value. Through a database query execution, it will return a set of data that will be stored in the variable *result*. The variable *result* is actually a dictionary data type, which is similar to a hash map that allows us to access any values generated from the query execution. By retrieving the *avatar_file*, we will be presented with the relative path of the location at which the character model is being stored. In order to load and render the character on the screen, the following has to be executed.

```
self.actorObject = Actor('models/characters/' +
```

Using the *Actor* constructor, you can load a specified character model and the actor animations that go along with that particular model. The string '*run'* is more of an arbitrary tag that will be used to distinguish between different animations whenever needed, but in our case, there will just be one animation, which is the running animation whenever a character is moving on the screen. But, in order for any models to render on the screen, one must be parented to the *render* node that represents the root of the scene graph, which is in charge of all things needed to be rendered in the 3-D world.

After loading the character model, a position must be set for the character to spawn at in the environment or else you will not see anything despite it being already rendered. Another important step is to tag the character model with a specific name and value as shown below.

For now, the purpose of this tag is to identify the object whenever the game is needed to search for this character model, for example, whenever someone clicks on this model, it will return what has been clicked on and the value that it contains. As shown above, this value stores the identifier for this particular character so that it can be easily referenced later on. For further explanation, look up *Mouse Picker* because that is where this is being used.

Collision Detection for Movements (Part I)

This section will only explain how to make an object collidable as in what steps are required to get started. The usage of these collision features will be further explained later on. To get started, the *Character* class contains something called a *Collision Traverser*, which goes through a list to check all objects that are being collided. And declaring one is simple.

Once the traverser is instantiated, a collision solid is needed to act against another collidable object, which in this case, a collision solid of a ray is required to collide against the environment that the character is walking on. As shown below, this is how to attach a collision ray onto the character model.

```
self.charGndColliderNode =
CollisionNode('charGndNode')

self.charGndColliderNode.addSolid(CollisionRay(0
```

The collision ray is set exactly 500 units above the ground onto the character model to ensure that even the character model itself, not only to just stay above the ground, but to also not walk into a wall. The -1 value is used as the direction the ray will be pointing at, which is towards the ground.

There are two collision masks you have to pay attention to, which is a *from* collision mask and an *into* collision mask. The *from* collision mask is set with a bitmask value of 0x4. This value again is arbitrary, but must conform to other bitmask being used throughout the game. The reason for the value 0x4 in this particular game is to only allow the collision ray to collide with the ground, which has its own bitmask value. For example, an object with a bitmask value of 0x2 can only collide into another object with a bitmask value of 0x2 or else no event is generated, which then cannot be processed. The *into* collision mask is set off, which is equivalent to the bitmask value of 0x0, because no collidable objects are allowed to collide into this ray as this is not needed.

Then finally, this collision ray is attached and parented to the collision model using *attachNewNode*.

In order to process these collision events, the events must be stored into some container called a *Collision Handler* as shown below.

```
self.charGndHandler = CollisionHandlerQueue()
```

A collision handler queue is required so that you can have access to all of the collision events that were generated and use them selectively. Then you would simply attach it to the collision traverser so that these collision events from the collision ray can be captured. To continue on as to how this is going to be used for keeping the character above the ground, look a little further into movements.

Starting Tasks (Subroutines)

Since we're still in the constructor, at the very end the initialization part of this *Character* class, we are going to start up something, in Panda3D, which is called a *Task*. A *Task* is just simply a subroutine that gets executed every frame to perform various tasks for the character to move, collide and other essential features. The code below shows exactly how two tasks are started.

```
taskMgr.add(self.updatePosition, 'updatePosition-'
```

The *add* method must be called from Panda3D's task manager, which just simply takes in a method such as *self.updatePosition* as the first parameter and a unique task identifier as the second so that this specific task can be removed later on. The easiest way to keep the identifier unique is just simply by attaching the character's identifier to it as shown above. And the task will be executed as soon as when the game runs. The purpose of these subroutine methods will be explained in greater details when we get to these particular methods.

charAttack Method

This method is used to send an attack to a *Bug* object in the game. The *Bug* class will be explained further in greater details later on. As shown below, the code snippet only shows the important aspects of this method.

```
self.actorObject.lookAt(self.objectTarget.getPos()
```

The way this game is set, is that whenever an attack is executed, the character model will face the direction at which the bug is located. But due to some weird bug, whenever a character model uses the *lookAt* method, the character model will turn towards the bug, but looks at the opposite direction. The solution to this problem is to add exactly 180 degrees from the angle the character is facing so that it will actually be looking at the bug instead. Then it will simply send an attack request as shown below.

```
rContents = {'attacker_id'  : self.object_id,

             'target_id'    :
```

But because this is an online game, it requires networking and protocols. To learn more about networking and protocols, please look for the section on *Protocols*.

charChase Method

This is the chasing method, which automates the character's movements to follow a designated target, which can be either a bug or NPC. In order for the character to chase a designated target, the *self.chaseTarget* variable must be set *True*. Whenever the variable is enabled, the character is then allowed to chase that designated target. Depending on the type of target, the character will either stop within attack range or within an action range as shown below.

```
self.setDestPoint(self.objectTarget.getPos())

destDistance = (self.destPoint -
self.actorObject.getPos()).length()
```

It must calculate the distance from the character's current position and the target's position. If *self.objectTargetType* is 1, then it is a bug, so it must check against the character's attack range. If *self.objectTargetType* is 2, then it is a NPC, so it must check against the allowable action range, so that the player can talk to the NPC. If the character is not within any of these ranges, the character would have to walk a little bit further as shown below.

```
timeElapsed = globalClock.getDt()
```

As mentioned earlier, the character movements are frame-rate independent, therefore it must check the amount of time that has elapsed between the last and current frame. Depending on the value, if the value is too great, meaning there is lag, the character would need to walk a much greater distance within the same time interval to keep up with faster machines that aren't lagging. If for some reason, your machine is running too fast, the time between frames would be very close to zero, therefore, you would be walking as you should be as there is no such thing as negative time.

charEngage Method

This method is always being executed right before an attack. The purpose of this method is to engage a bug that is in passive mode as a means to provoke it by sending a certain request to the bug as shown below.

```
if (self.canSeek):

    self.setCanSeekDelay()
```

The character must always be within attack range for this to successfully occur. As shown above, there is a delay called the seek delay, which is to only send one request at a time, meaning you can only engage the same or different bug after this specified delay.

charCollide Method [Collision Detection for Movements (Part II)]

This method plays a major role in keeping the character above the ground and outside of obstacles at all times. Similar to every other collision process, the traverser must be executed at all times to check for collision events as shown below.

Now moving on, if any collision events took place within this frame, it will check for it in the collision handler queue to see if any exists. If exists, the following is performed.

```
if (self.charGndHandler.getNumEntries() > 0):
```

It will first sort out all the collision entries so that it will get the latest collision event that occurred, if any. Then will be processed as shown below.

```
if (collisionEntry.getIntoNode().getName() ==
'terrain'):

    self.position = self.actorObject.getPos()
```

Whenever the character is above walk-able ground, that current position will be stored as the last valid position and while it is doing that, it will adjust the character's Z position if necessary for elevated terrains. If anything goes wrong or that the character has walked into the wall or obstacle, the character will be relocated back to its last valid position as recorded from the previous statement. This will ensure that a character can never walk off a cliff, table, etc. or into the wall of a dungeon.

Here are a few screenshots demonstrating collision in action. If you look closely, you can see a wireframe pattern on the collided object showing that a collision is occurring.

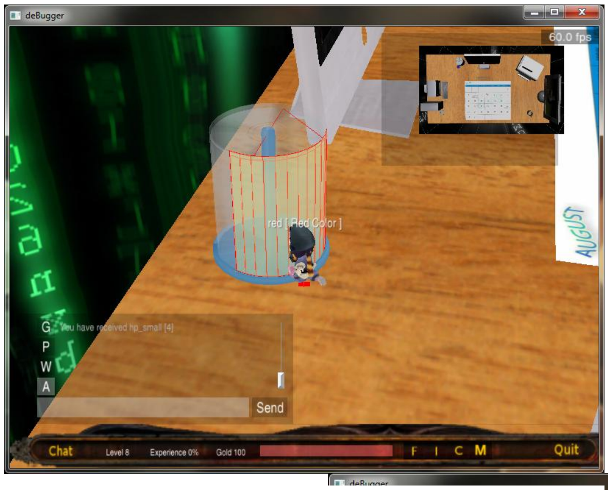

You have received hp_small [4]

Figure 5.C.x-1: Colliding with Disc Spindle

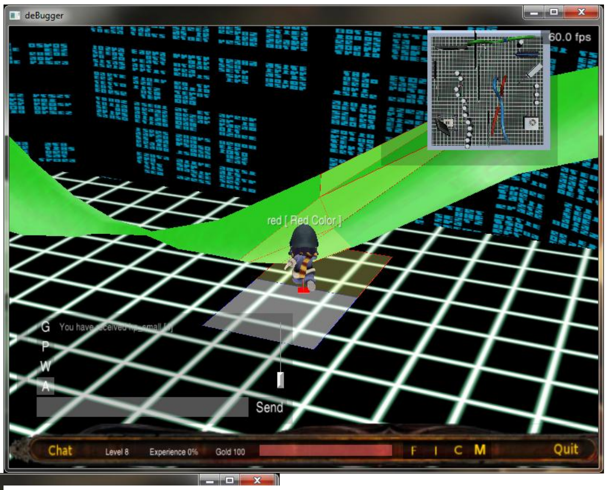

You have received hp_small [4]

Figure 5.C.x-2: Colliding with Obstacle

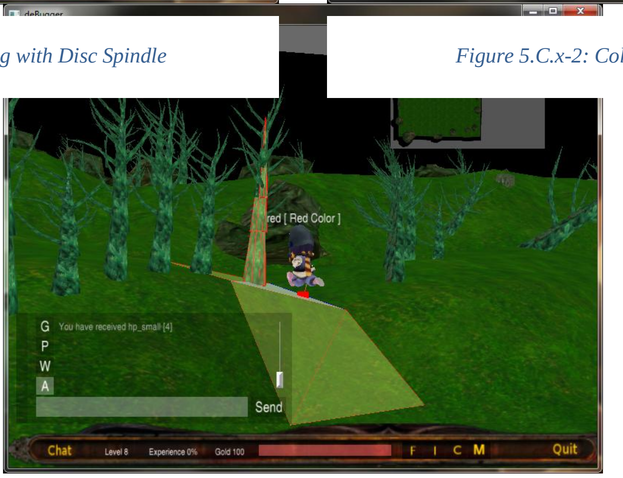
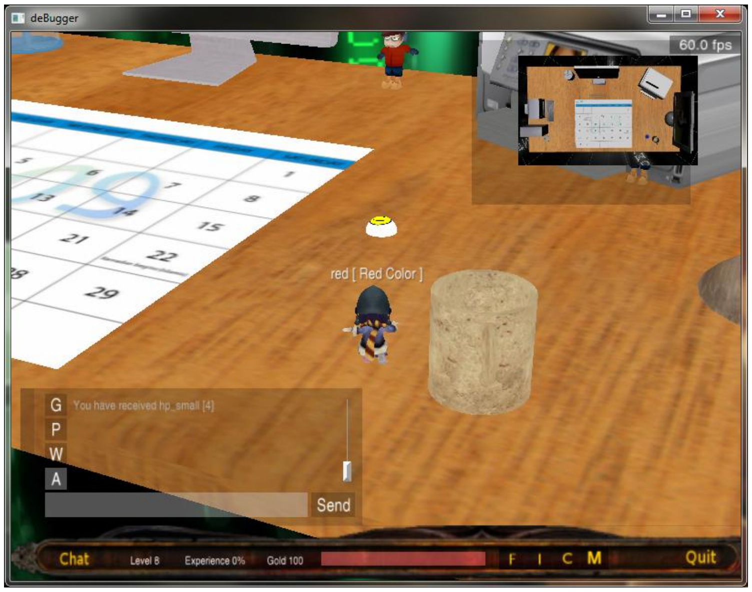

You have received hp_small [4]

Figure 5.C.x-3: Colliding with Tree and Ground

Do note that the collision wireframe only appears when a specific method for the traverser is called. This method comes from Panda3D, which is called *showCollisions*. In order to do this, look below.

charMove Method

This method is used to help automate the character moving process whenever the player tries to move by clicking with the mouse. Whenever a player uses the mouse to click at a destination, the character would automatically move to that destination at some certain speed and distance. Here’s a screenshot.



You have received hp_small [4]

Figure 5.C.x-4: Clicked on Environment

As shown above, the sphere on the ground is the chosen destination and the character will then be instructed to walk towards that destination at a certain speed.

Since this is character movements, this method will also make use of the frame-rate independent concept, which is based on the time elapsed between two frames. As shown below, the character will be positioned to face towards the destination.

```
self.actorObject.lookAt(self.destPoint)
```

As mentioned earlier in the *charChase* method, the reason for the 180 degrees adjustment is because whenever the character uses the *lookAt* function, the character would always be looking at the opposite direction at which it is supposed to be looking at. The way how a character is being moved forward is shown below.

```
timeElapsed = globalClock.getDt()

self.actorObject.setY(self.actorObject,

                       -(timeElapsed *
```

It takes the time elapsed between two frames and adjust its position accordingly depending on how fast or slow the machine is,

```
self.actorObject.setHpr(self.heading, 0.0, 0.0)
```

so that everyone would be moving at the same rate. The remaining distance towards the destination will be then calculated. Since every step a character makes is not constant, there are chances that the character may have moved beyond the destination point. In order to prevent this, the character will be stopped a varying distance of less than 0.25 or whenever the character instantly changes direction, meaning it has passed it since the character is programmatically forced to look at its destination. If one of the above conditions is true, the following below will be executed.

The character's position will automatically be fixed onto that destination point and the direction at which the character is supposed to be facing will be fixed back to forward as before.

charMoveK Method

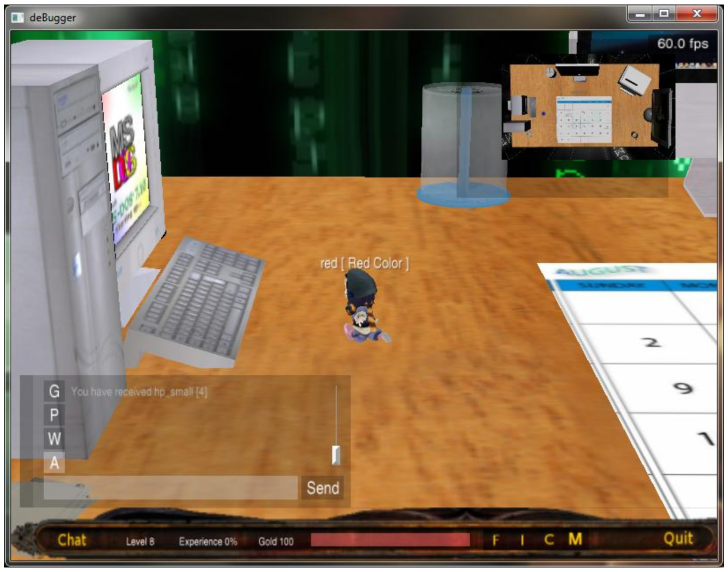
This method is similar to the *charMove* method but instead of using the mouse, it uses the keyboard. There is another class that this method uses which is the *ControlScheme* class that monitors and tracks all keyboard events related to movements. This includes the *WASD* and *Arrow Keys*. As shown below, it must check whether or not the player is pushing the corresponding keys.

```
if (self.world.controlScheme.keyMap['walk-up'] !=
0 or
```

Whenever any of these events are triggered, the character will face towards that particular direction and move forward as shown below.

```
if (self.world.controlScheme.keyMap['walk-up'] !=
0):

    if (self.world.controlScheme.keyMap['walk-
left'] != 0):
```

The following above only shows three of the eight possible directions at which a player can move with the keyboard. The code snippet above is showing movements that are related to only the up direction. If a player is pushing both the up and left directions, it will turn the character to the specified degrees towards that direction, which is upper-left as shown below.



You have received hp_small [4]

Figure 5.C.x-5: Moving Upper-Left

If the player is pushing both the up and right directions, it will again turn to that corresponding direction, which will create an upper-right direction as shown below.

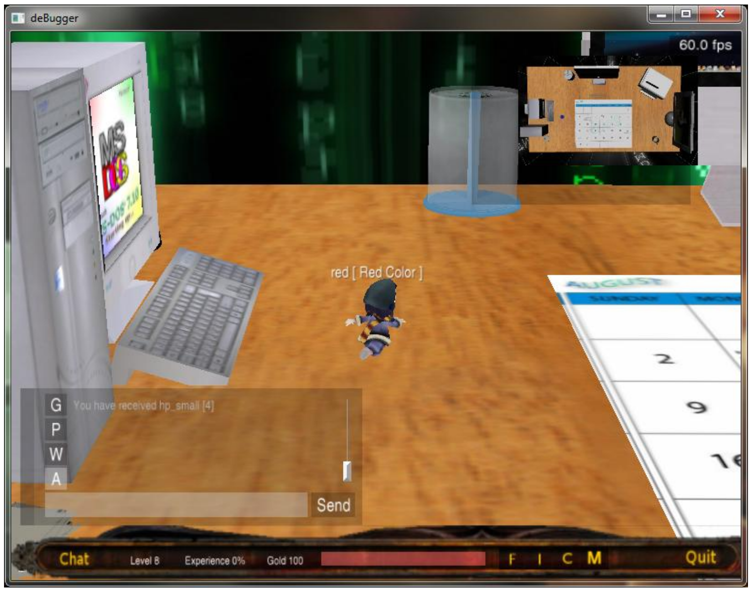


Figure 5.C.x-6: Moving Upper-Right

If the player is simply just pushing up it will simply just face up as shown below.

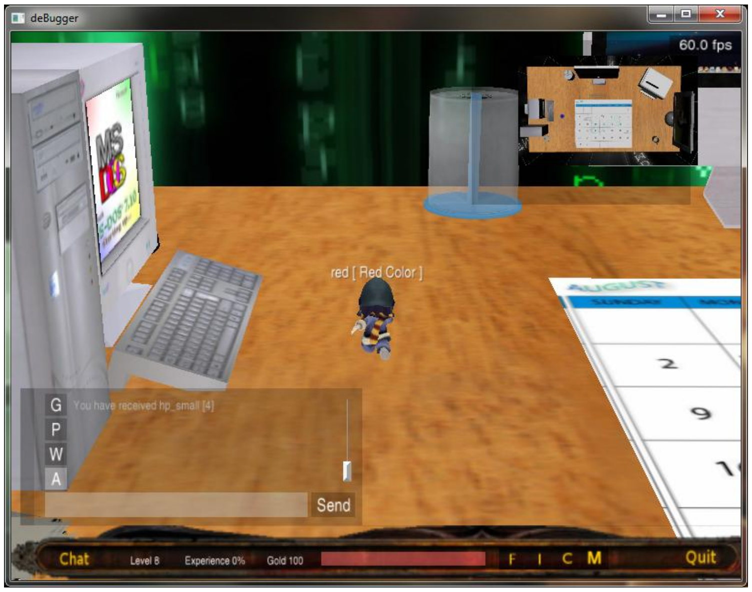


Figure 5.C.x-7: Moving Straight Up

But notice how the camera that's being used as shown below.

This is to ensure that the character will be facing that direction, so that if moving straight up is exactly what the player wants, the character will always move up even if the camera turns. And lastly, notice the screenshots, there are no sphere on the ground for a destination point because you are not using a mouse to automate the walking process. You are actually walking the character yourself manually by pressing a directional key.

checkWarp Method

This particular method is used to check all objects within the surrounding area to see if the object is a portal. If the character is within a particular distance of the portal, the player will be warped as shown below.

```
if (distance <= oTarget.portalRadius):
```

The character would be transported instantly to another environment wherever the portal leads to. This is the only way of travelling between multiple environments. Here's an example how a portal looks in the game.

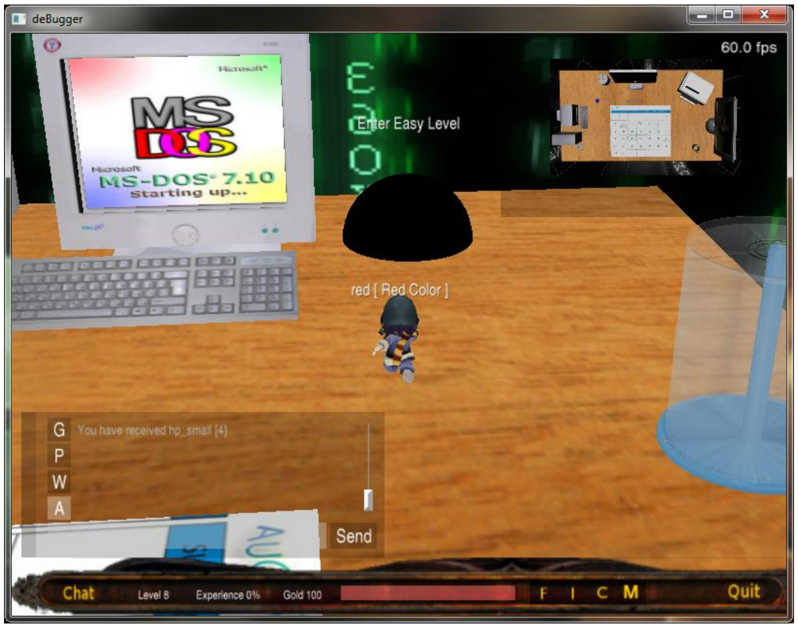


Figure 5.C.x-8: Moving Towards Portal

setDestPoint Method

This method is mainly used in conjunction with the *charMove* method. Whenever the player uses the mouse to click at a certain destination, it will call this method, which is used to set a particular destination target to automate the character to walk to that specific spot. The way how it works is very simple as shown below.

```
self.destPoint = destPoint
```

The specified destination point will be chosen and stored. It will then ensure that the character will prepare itself by telling the character to look at the destination point to be moved to. And then it simply just stores the current direction the character is facing.

```
if (not self.isMoving):

    self.isMoving = True
```

One more step to prepare the character for moving is to check whether or not the character is already moving. If the character is not moving, it will flag it so that the character will know to move and while calling the *'run'* animation to play at a certain speed to simulate a walking experience being rendered on the screen. Just with these few steps, the character will then successfully walk towards that destination point that was set.

takeDamage Method

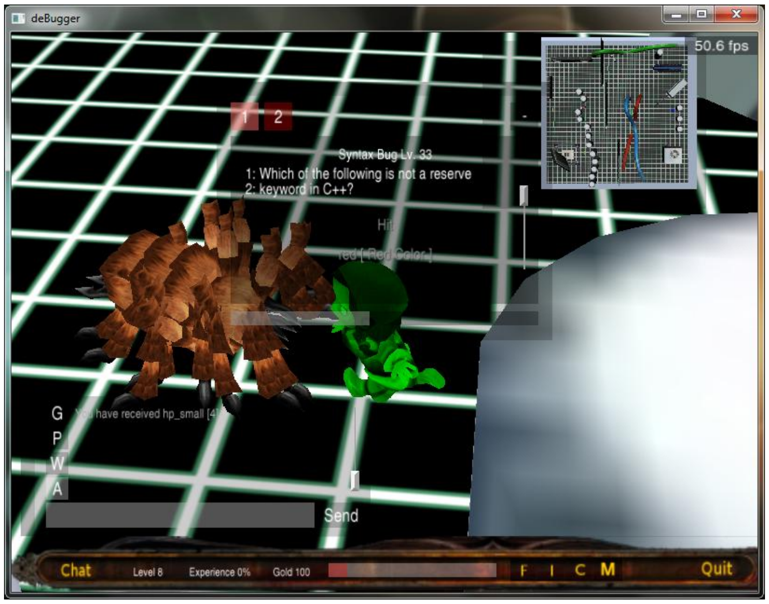
This method is mainly used with the *Battle System*. Whenever the character is being attacked by a bug, this method will be called to adjust your health points and check to see whether your character has taken enough damage to be defeated as shown below.

```
if (dType == 1 or randint(0, 99) >
int(round(self.dodgeRate))):

    # Damage Amount Taken
```

If a player has been defeated in battle, the character will be flagged as dead so that the game and bugs will know that this particular character is unable to do anything because it has been defeated. As you may recall, this is where the *randint* method is being used to calculate the change at which the character can dodge an attack causing the bug to miss, which in effect, you take no damage at all. One more thing is that when a character is defeated, the game will send a request to the server to let everyone know that you have been defeated as shown below.

```
rContents = {'object_id'    : self.object_id}
```

There are also some simple special animation effects whenever a character takes damage such as the text *'Hit'* is shown above the character and a flickering color on the character to signify that you are taking damage as shown below.



Syntax Bug Lv. 33
1: Which of the following is not a reserve
2: keyword in C++?

You have received hp_small [4]

Figure 5.C.x-9: Character Taking Damage

charRoutine Method

This is the main routine task that drives everything for a character from moving, chasing and attacking. As shown below, the method header looks a bit different than other methods mentioned above which is the *'task'* parameter.

The *'task'* parameter signifies that this method will be treated as a subroutine that will continue to run forever as needed. But in order for this method to run forever, it must have the return statement as shown below.

But the most important part of this method is what's between the above statements as shown below. But we'll first start out with character movements.

```
if (not self.isDead):

    self.charMoveK()
```

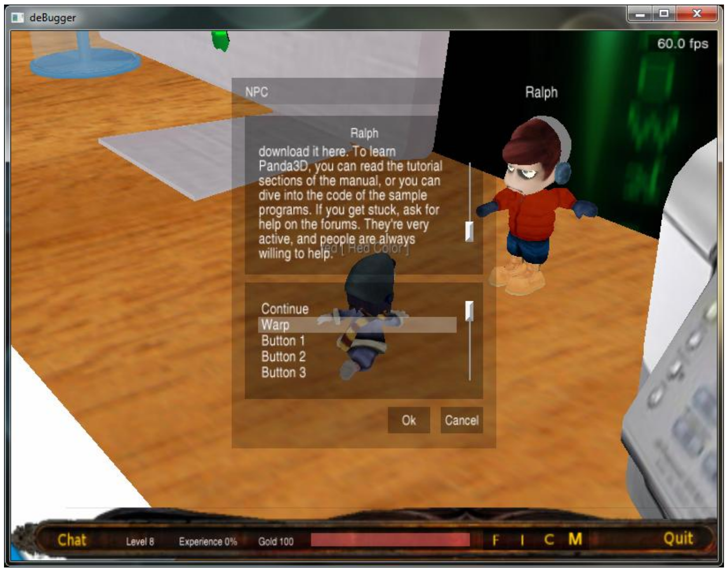
This is exactly where those *charMove* and *charMoveK* methods are being executed. These methods are being executed ever frame making the character movements possible. Because this subroutine is accessed every frame, the character is taking approximately one step at a time as if you were walking. Now moving to the bug related tasks of this subroutine as shown below.

```
elif (self.objectTarget != None):

    self.targetList.append(self.objectTarget)

    self.objectTargetType =
self.objectTarget.getType()
```

Whenever you've selected a target with your mouse, the target will be set and stored in a list of other targets that you had already clicked on. By checking the object's type, where the value 1 represents a bug, it will perform a sequential set of methods, which calls both the *charChase* and *charEngage* methods that will be used to tell the character to follow and walk towards that specific target, which in this case is a bug. If certain circumstances and conditions are met, the following methods will perform their own specific function. Similar to this bug type object, whenever a player clicks on a NPC, the same thing would happen as shown below.

Although this may look similar to the bug, the character does not have to engage the target since NPCs cannot be harmed, therefore, you will only chase. If your character is within a specific range, it will call the *getAction* method that will execute whatever purpose that the NPC serves, such as a dialog menu or a shop interface as shown below.

```
elif (self.objectTargetType == 2):

    self.charChase()
```



NPC

Ralph

download it here. To learn Panda3D, you can read the tutorial sections of the manual, or you can dive into the code of the sample programs. If you get stuck, ask for help on the forums. They're very active, and people are always willing to help.

Figure 5.C.x-10: NPC Dialog Interface

And lastly, there are a few more methods that will be executed in this subroutine as shown below.

```
self.charCollide()
```

The *charCollide* method is executed here in every frame so that right when a character moves, it will quickly check for where the character's at and ensure that it is above the ground and out of obstacles. The *checkWarp* method is also executed here so that if a player decides to travel into another environment, it will check all nearby portals for this to occur.

unload Method

As with most of the other components in this game, there is an *unload* method at the very end of it. This method is only called whenever this character is required to be unloaded from the game. The process is very simple. It basically just undoes everything that was instantiated, created, etc. Remember how you should be creating a unique identifier for each task in the very beginning?

```
taskMgr.add(self.updatePosition, 'updatePosition-'
```

As shown below, in order to stop this task from executing, you will have to call the *remove* method from Panda3D's task manager.

```
taskMgr.remove('updatePosition-' +
```

If you compare the two code snippets from above, you will notice that the identifiers look exactly the same, which makes it a whole lot easier to stop to this task from continuing on. And moving on, this is how you will unload the model from game itself.

```
self.actorObject.unloadAnims({'run':
'models/characters/' + self.charModel
```

Just simply destroying the *Character* instance is not sufficient enough to remove the character from the game. If you were to just destroy the *Character* instance, the character model would still exist running inside the Panda3D engine. So you have to perform these necessary steps to completely remove the character from the game. As shown above, the first step is to unload any animations that it was loaded prior to this statement such as the *'run'* animation that was loaded when the character model was loaded. Then you have to unload the model from memory using the path of where the model is located. And finally, it will just call these two Panda3D methods called *cleanup* and *removeNode* so that it can dereference the node of the character and destroys it.

Panda3D makes use of timers, sequences and intervals to perform tasks that requires time. The *Character* class actually uses a few sequences to perform some timed tasks. Because this instance is no longer needed, these sequences must be stopped since the sequences are also no longer needed. If these sequences are not stopped at the removal of this instance, you may encounter certain problems that may crash the game. To stop one of these sequences, check one out below.

```
if (self.canSeekSequence != None):
```

And that's it! Just remember to undo every Panda3D related instances and sequences and you'll be fine. It will also ensure that the memory that was used will free itself to prevent the game from taking up all of the system resources and potential memory leak.

updatePosition Method

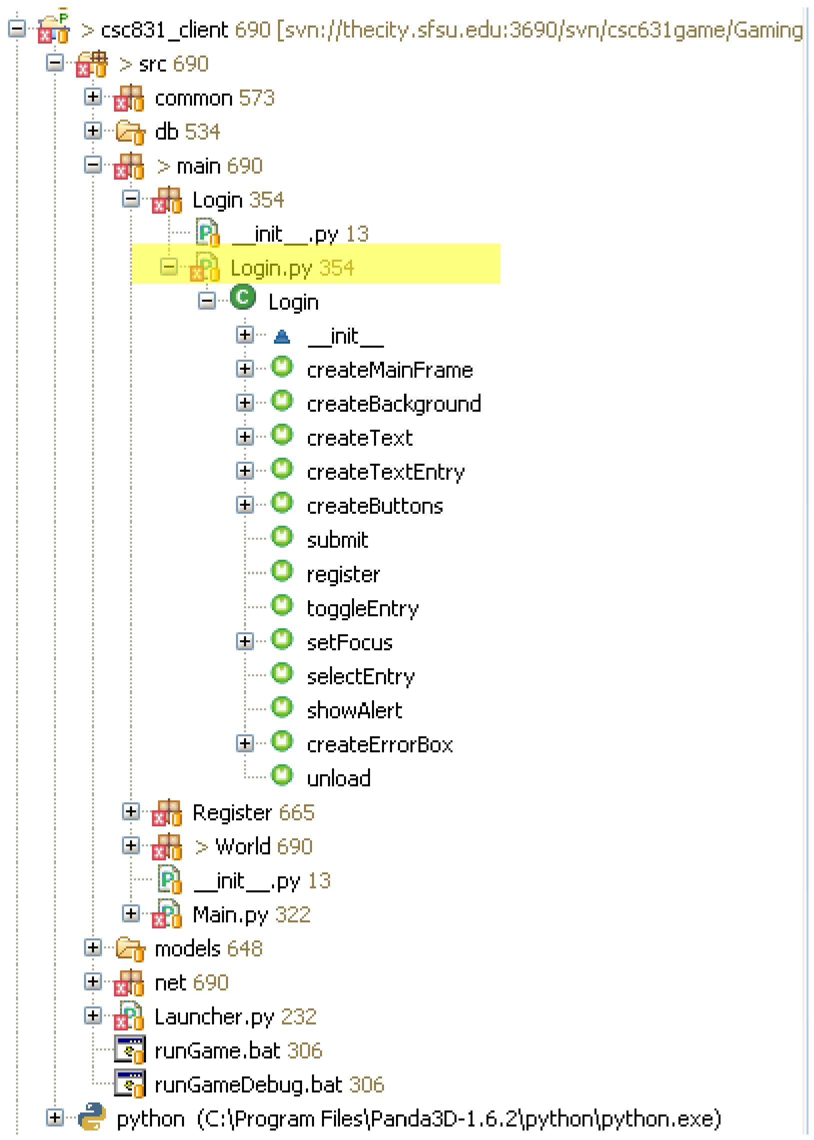
This particular method located in the *Character* class allows your character to send the current position so that others in the game will know exactly where you are and whether if you're moving or stopped. As shown below, it simply just grabs the current position and the direction your character is facing and then sends it off to the server.

```
rContents = {'user_id'  : self.object_id,

             'x'        : currentPos.getX(),

             'y'        : currentPos.getY(),
```

You may also notice that it is also sending the move speed because a character at any point in the game can have a variable speed so that others in the game will also know that your move speed has changed. And lastly, this is also one of those tasks that are executed every frame so that it can constantly be sending your latest position to the server.

LOGIN PROCESS

The file that contains the login process is called Login.py and it is located at main/Login folder or package. The Login class consists of 14 functions which are used to display the login screen user interface and to capture the event that the user performs.
__init__ function is the entry point of the Login class. It contains a couple of function calls which set up the login screen.
createMainFrame function is used to display the center login box. This function will set the position of the login box and display the 'Welcome' title. 3



```
> csc831_client 690 [svn://thecity.sfsu.edu:3690/svn/csc631game/Gaming
  > src 690
    common 573
    db 534
    > main 690
      Login 354
        __init__.py 13
        Login.py 354
          Login
            __init__
            createMainFrame
            createBackground
            createText
            createTextEntry
            createButtons
            submit
            register
            toggleEntry
            setFocus
            selectEntry
            showAlert
            createErrorBox
            unload
      Register 665
      > World 690
      __init__.py 13
      Main.py 322
    models 648
    net 690
    Launcher.py 232
    runGame.bat 306
    runGameDebug.bat 306
  python  (C:\Program Files\Panda3D-1.6.2\python\python.exe)
```

createBackground function is used to display the background of the login screen. This function will query the database to get the image background.
createText function is used to display all the input labels, such as Username and Password. This function also displays the login box header.
createTextEntry function is used to display the text input fields for the username and password. While the input field for the username is a clear text, the password's input field is set to be obscured by adding 'obscured = 1' option when calling the DirectEntry function.
createButtons function is used to display the two buttons at the bottom of the login box. The left button, 'Log In', is set to call submit function when it is clicked. The right button, 'Register', is set to call register function.
submit function is used to capture the user inputs. This function will also do a quick validation on the input by checking if they are not empty. Once it passed this validation, the function will send the user inputs, username and password, together with the protocol constant to the server.
register function is used to switch screen to register screen.
toggleEntry function is used to capture user input when the user is using 'tab'. This function will move the focus from one input field to another based of the 'direction' variable that is passed as the second argument of this function.
setFocus function is triggered by toggleEntry function. This function is used to set the focus to the specific field that is passed as the second argument of the function.
selectEntry function is also triggered by the toggleEntry function. This function is used to set the focus to only 'eNum' variable field that is passed as the second argument of the function.
showAlert function is used to triggered a notification message when the authentication failed. This function is calling createErrorBox function with a sring argument of 'Login Failed!' to display the notification message.
createErrorBox function is used to display the notification error. It takes a second argument of 'msg' variable that contains the notification message and displays it in a box.

unload function is used to unload the Login class when the screen is about to switch to another screen.

BattleSystem.py

As mentioned before in Chapter 4, the *Battle System* is one of the central parts of the entire game. The *Battle System* consists of several components such as the battle interface, the interaction between the player and bugs, rewards and many more. This section will help further explain the logic and functionality of the entire *Battle System*.

Getting Started

As usual with other classes, a few Panda3D modules will have to be imported. One particular module that is required for the *Battle System* is the *DirectObject* module as shown below.

```
from direct.showbase.DirectObject import DirectObject
```

The *DirectObject* module enables us to generate and capture events triggered either by the game, input devices or user defined conditions. More details will be explained further on.

Battle System Class

Similar to the *Character* class, the *Battle System* will be also make use of a class for inheritance, but unlike the *Character* class, the *Battle System* will, instead, be inheriting all methods of the *DirectObject* module, so that we can directly access those methods provided from this module.

```
class BattleSystem(DirectObject):
```

The method that the *DirectObject* provides is the *accept* method. The *accept* method captures all events with a specified tag. For more details, look further into *Initialization*.

Initialization

The constructor for the *Battle System* is as follows.

```
def __init__(self, world, charObject):
```

The *Battle System* class will be accepting a reference of the *World* instance and the *Character* instance so that this specific *Battle System* instance can access other variables and methods that are located within the *World* or *Character* classes.

This class will also make use of some specific containers that will store various objects for easy access, so that these objects can be referenced by a particular index. The following containers that are listed in the constructor are as follows.

```
self.bugHealthBar = []

self.bugTab = []

self.responseButton = []
```

Although these containers may not mean anything just yet, you will find out later in greater details. All of these containers, which are of list data type, will be storing all objects related to the *Battle Interface*. The list called *self.bugHealthBar* will simply store *DirectWaitBar* instances that are simply health bars that will display the amount of health a particular bug has left. Do note that the amount of health is equivalent to the amount of questions a single bug can have.

The next on the list is another list container called *self.bugTab*, which will be storing *DirectButton* objects that will later be used as quick access to a certain bug and its question. The *DirectButton* module allows the creation of clickable buttons that will trigger an event to call a particular method for processing.

The *self.responseButton* and *self.responseScrollBar* again are just another set of list containers used to store *DirectButton* and *DirectSlider* objects respectively. The *DirectSlider* object is simply just a scrollbar that has a varying range of values.

These last two containers called *self.vQuestion* and *self.vResponse* are used to store *DirectLabel* objects, which simply are just labels for displaying text on the screen. Each *DirectLabel* object within these containers will serve as a line of text. If the text is too long, there will be a method that will break up the string of text into multiple parts to be stored on the lines afterwards.

And again, these containers are only meant to keep every *Direct* object organized for easy access using an index. And that index will most likely correspond with the order at which it is displayed from top to bottom. The use of these containers will be further explained shortly.

Another important container as shown below will keep track of all *Battle* instances that are created. Again, these are *Battle* instances and not *Battle System* instances. Do not mix the two. A *Battle* instance is a subset of the *Battle System*, which is used to represent an individual active battle between the player and the bug.

Now there's only one more major container left, which is the *self.bugList* container. This will store all bugs

```
self.battleList = []
```

that you, as the player, are engaged with. This container will have a limit, which is currently set to 4 as shown

```
self.bugList = []

self.currentBugNum = 0
```

below.

The reason for the *self.maxBugCount* limit is to maintain a fixed amount of bugs that can be attacking you at all times. It would also give everyone a chance to fight a bug because no one player should have the right to take up all the bugs on the map.

As mentioned earlier, the *DirectObject* module allows us to capture triggered events from anywhere within the game. The event that we want to capture for the *Battle System* is the event labeled as *'endBattle'*, which calls the method, *self.checkEndedBattles*, as shown below.

```
self.accept('endBattle', self.checkEndedBattles)
```

This specific event is generated from within a *Battle* instance whenever the *Battle* instance detects that if the bug is out of questions or health, then it is signifying that the bug has been defeated. Once this event is generated, this *Battle System* instance will capture that event and remove that specific *Battle* instance from the *self.battleList* container to show that the battle has ended.

Starting Tasks (Subroutines)

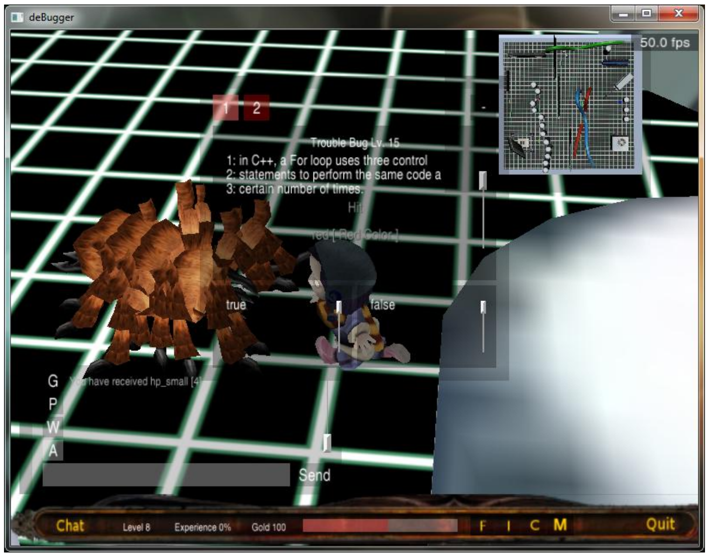
Similar to the *Character* class, the *Battle System* has its own set of subroutines that are required for the *Battle System* to perform its job. There are exactly two subroutines that are being used in this class, which are as follows.

```
taskMgr.add(self.selectBugK, 'selectBugK')
```

The *self.selectBugK* method is used to monitor the *'Tab'* keyboard input key so that a player can easily tab through the list of available bugs in the *Battle Interface*. The second subroutine is going to call the *self.battleSystemRoutine* method, which, of course, will be looping over and over until it is no longer needed. The *self.battleSystemRoutine* method is considered as the main driver for all *Battle System* instances.

Creating the Battle Interface (Battle GUI)

The *Battle Interface* is the most important component of the *Battle System* because it provides the player a graphical user interface that displays a list of battles, a specific question and the responses that goes with that question. Without this interface, the player would not be able to interact with bugs found around the environment. The entire concept of *Bug Hunting* lies within this *Battle Interface*. And it is what makes it possible.



Trouble Bug Lv. 15
1: in C++, a For loop uses three control
2: statements to perform the same code a
3: certain number of times.

have received hp_small [4]

Figure 5.C.xii-1: An Example of the Battle Interface

The *Battle Interface* that is being used with the *Battle System* consists of several parts and will be constructed through a series of methods located in the constructor. Lists of methods that are responsible to create this *Battle Interface* are

```
self.createMainQuizFrame()

self.createBugTabBar()

self.createHideResponsesButton()
```

shown below.

createMainQuizFrame Method

This method constructs the canvas called the main frame which will be served as the parent of all *DirectGui* objects. The main frame will be using the *DirectFrame* module that is the most basic of all *DirectGui* objects as shown below.

```
self.mainFrame = DirectFrame( frameColor = (0.0, 0.0, 0.0, 0.2),
                              frameSize = (-0.6, 0.6, -0.7, 0.775),
```

As you may have noticed, the main frame will be hidden once it is created. The reason for this is that the *Battle Interface* should only display itself if the player is in battle and returns back hidden when not. Because we're just displaying whenever it's needed, this can save a lot of overhead that may be generated if we were to always keep destroying and recreating the *Battle Interface*. Also in this method, there is something called a title bar as shown below.

```
self.titleBar = DirectFrame( frameColor = (0.0, 0.0, 0.0, 0.3),
                             frameSize = (-0.6, 0.6, -0.025, 0.025),
                             pos = (0.0, 0.0, 0.8),
```

This title bar can be seen in other pop up interfaces such as when talking to an NPC, displaying the friends list, character information window, etc. And like every other *DirectGui* objects that are being used for the *Battle Interface* will be parented to the main frame, so that if we need to hide the main frame, all of its child objects will also be hidden. Even when we need to destroy the main frame for unloading purposes, we just simply call the *self.mainFrame.destroy* method that will destroy itself and its child interface objects, so that we do not have to destroy every single one individually.

createBugTabBar Method

This method is mainly used to create the tabs that are required to be displayed above in the *Battle Interface*. As mentioned in the constructor initialization part, this method will make use of the *self.bugTab* container as shown below.

```
self.bugTab.append(DirectButton( frameColor = (0.0, 0.0, 0.0, 0.1),
                                 frameSize = (-0.05, 0.05, -0.05, 0.05),
                                 relief = DGG.FLAT,
                                 command = self.selectBug,
```

This small code snippet is located inside a loop that will create a series of tabs using the *DirectButton* module. Whenever this tab is clicked upon, it will call the *self.selectBug* method that will allow you to target that specific bug and display a question from it. The *self.selectBug* method will be explained in further details below. You can also see that all of these tabs will be parented onto the main frame. Because these *DirectButton* objects are being stored inside the *self.bugTab* container, these tabs can be referenced back just by using its corresponding index value.

And also in this method is where the health bars are being created as shown below.

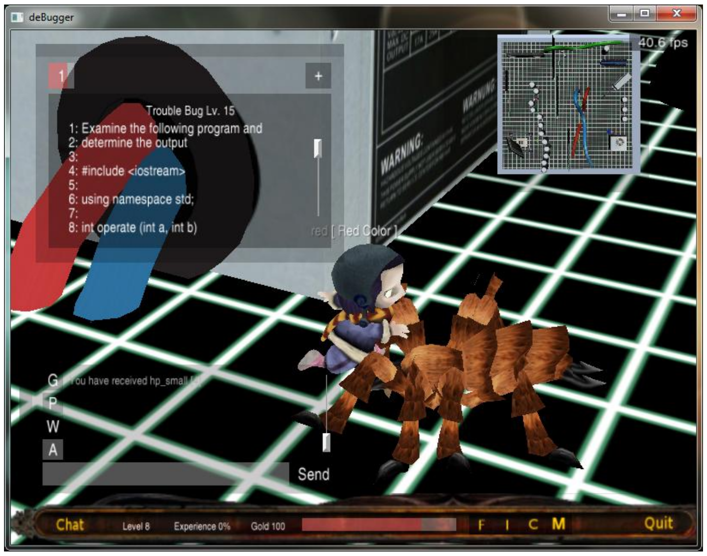
```
self.bugHealthBar.append(DirectWaitBar( text = str(i + 1),
```

Similar to the tabs, the health bars are being stored within its specific container, but this time, these *DirectWaitBar* objects are not being parented to the main frame, but instead onto its corresponding tab which will also be labeled with its index.

createHideResponsesButton Method

In this particular method, it is responsible for creating a *DirectButton* on the upper-right corner of the *Battle Interface* as you might see to the right with a *'+'*. Whenever the responses are visible, the button will simply change to a *'-'*.

The purpose of this button is to toggle the displaying of the responses so that only the question will remain on the box. To achieve this, whenever a player clicks on this button, it will call the method, *self.toggleResponses*, which will simply perform all the work to hide the bottom portion of the *Battle Interface*. You can easily reveal back the responses by clicking the button again. This feature does not really affect the gameplay at all. This is just more of a convenience for the player so that they won't accidently click on a response when trying to move around the environment.



Trouble Bug Lv. 15

```
1: Examine the following program and
2: determine the output
3:
4: #include <iostream>
5:
6: using namespace std;
7:
8: int operate (int a, int b)
```

You have received hp_small [4]

Figure 5.C.xii-2: Demonstrating the Hiding Responses Button

createQuestionBox Method

This is another method that is used to help construct the *Battle Interface*. But this method is one of most important part because this box is in charge of displaying the question properly onto different lines. And while adding a scroll bar so that a player, in case, would have to scroll through a long question. To get started is to first create a *DirectLabel* object, which will serve as the canvas for displaying the question.

Within this question box, at the very top, there is actually another *DirectLabel* object that is responsible for displaying the bug's name and level. Again, all of these objects will be parented to the main frame as shown below.

```
self.questionBox.reparentTo(self.mainFrame)
```

Now moving onto the harder part, which is to create a series of lines on the question box so that if a question had multiple lines of code, it will be formatted in such a way where the lines will be number and that it will be printed onto its corresponding line. This will involve a loop as shown below.

```
for i in range(self.maxQuestionLines):
    questionLine = DirectLabel( text = '',
                                text_pos = (0.0, 0.0),
                                text_scale = 0.055,
                                text_fg = (1.0, 1.0, 1.0, 1.0),
                                frameColor = (0.0, 0.0, 0.0, 0.0),
```

Because we are working with a limited amount of space, there must be a maximum amount of lines we can be displaying at once in the *Battle Interface*, which is determined by the *self.maxQuestionLines* variable. Each line of text will be created using the *DirectLabel* object. As mentioned in the constructor, this where it will make use of the *self.vQuestion* container, which is in charge of storing this set of *DirectLabel* objects for quick reference later on.

As mentioned, we are working with a set number of lines, so we need to toss in a scroll bar to display the remaining parts of the question as shown below.

```
self.questionScrollBar = DirectSlider( range = (1, 0),
                                       ...
```

You may be wondering if the displaying of the remaining lines of the question is automatic. It is not. There are pre-defined *DirectGui* objects that contain a scrolling feature, but if it is an interface that you don't like, such for this case, everything must be done manually from the construction to the displaying of this scrollable interface. The method that we will be using to perform the scrolling feature is called the *self.scrollQuestion* method, which will be explained further on.

createAtkDelayBar Method

In this method, we will be constructing something called the attack delay bar, which is simply just a loading bar that will delay the display of the responses. The duration of this delay varies from character to character depending on their attack speed. The purpose of this bar is to create a form of penalty whenever a player answers a question wrong as shown below.

\

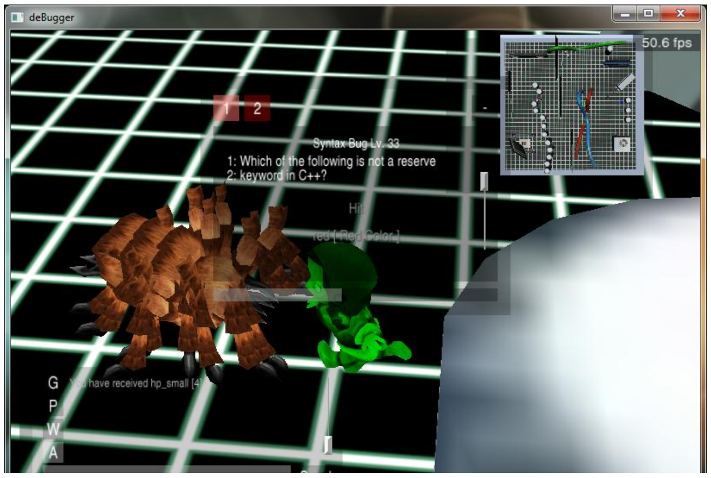


Figure 5.C.xii-3: Demonstrating Attack Delay Bar

The delay bar is simply a *DirectWaitBar* object that contains a sequence and method, which is used to adjust the value of the delay bar over a period of time. When the delay bar finishes, remaining responses should reappear.

createResponseButtons Method

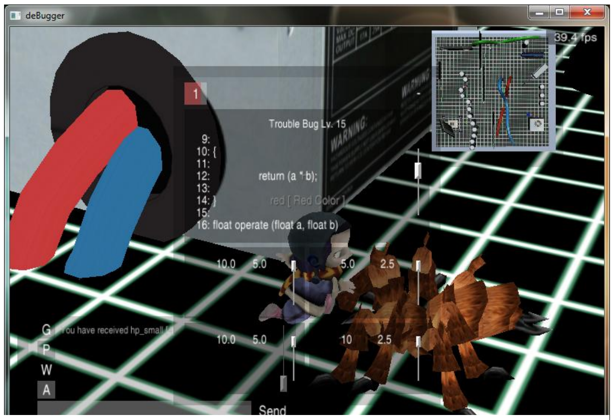
If you would simply look back to the start of this section, you will notice that below the question box contains a few buttons that will display any possible choices for the player to answer the question. Each *DirectButton* object is similar to how the question box is constructed, which is that each button consists of a series of *DirectLabel* objects that are treated as lines. And because we're working with limited space, a scroll bar is added for the player to scroll through an answer if it takes up more space than the button. As shown below is a small snippet of code that was used to create these buttons.

```
self.responseButton.append(DirectButton( text = '',

                                          ...

                                          command = self.sendResponse,

                                          extraArgs = [i] ))



...
```

Each *DirectButton* object will be treated as a response button that whenever it is clicked, it will call the *self.sendResponse* method, which will also be sending in the current index of the button for processing. The response button is then stored within the *self.responseButton* container for easy reference. Similar to the question box, you can see that another loop is used to create a series of *DirectLabel* objects as lines for the responses. The *self.vResponse* container will be responsible for keeping track of all these lines. Do note that the *self.vResponse* container is a list of lists where each index stores a list of lines for that button. And now we add the scrollbars for each button as shown below.

```
self.responseScrollBar.append(DirectSlider( range = (1, 0),

                                             ...

                                             command = self.scrollResponse,
```

As with other scrollbars, the *DirectSlider* module will be used to create the scrollbars, which will then be stored inside the *self.responseScrollBar* container for easy reference. Do note that every time you scroll the bar, it will call the method, *self.scrollResponse*, to scroll that particular button. And below is another screenshot of the finished *Battle Interface* with the question, responses and everything else.



```
9:
10: {
11:
12:         return (a * b);
13:
14: }
15:
16: float operate (float a, float b)
```

Figure 5.C.xii-4: Battle Interface Scrollbars

scrollQuestion Method

This method is used in conjunction with the question box to scroll through a question with multiple lines of text. This method is always executed whenever a player is moving the scrollbar. The first step that this method must do is to retrieve the current scrollbar value as shown below.

```
sliderValue = int(self.questionScrollBar['value'])
```

The reason for the type casting is to simply ignore the floating point values because we are only going to display full lines of text, not partial. After this step, it will try to retrieve a question in text format as shown below.

```
question = self.questionObject.getQuestion()
```

The question we will be retrieving has already been formatted and parsed in such a way that each line of the question is stored into a list so that each element is equivalent to one line of the entire question and don't worry, it has been formatted to the length of the question box, so that it will not be too short nor too long. After doing so, it will try to figure out its line number and then simply inserts each line of the question into those *DirectLabel* objects that was created earlier as shown below.

```
if (len(question) > self.maxQuestionLines):

    self.vQuestion[i]['text'] = lineNumber + question[sliderValue + i]
```

scrollResponse Method

This is exactly similar to the previous method, *scrollQuestion*, except that this method has an extra parameter for the response button number, the index. This method will always be called whenever the player moves the scrollbar on any of the response buttons. Then like before, it will retrieve the scrollbar value and the corresponding response, which is again formatted into a list, so that it will have almost the same length as the button itself. Then it will simply insert the text into those *DirectLabel* objects as shown below.

```
for i in range(maxItems):

    if (len(response) > self.maxResponseLines):

        self.vResponse[rButtonNum][i]['text'] = response[sliderValue + i]
```

How does the Battle System work?

The whole *Battle System*'s main functionality relies entirely on network protocols. The *Battle System* is initiated a little bit different depending on the whether the bug is aggressive or passive. Whenever a player encounters an aggressive bug, the bug will simply send a request, an invitation, to the player, similar to a PvP System, in which the player will automatically respond back so that the bug can chase the player. When the conditions are met such as if the player and bug is within attack range, the battle interface will pop up and present a random question and along with responses that has already been preloaded on the client-side when the player first loaded into the environment. Because the question and responses are already on the client-side, the validation should be taken place on server-side, but it's actually on client-side. If the validation is a success, the client will simply send an attack request so that the server will calculate and deduct the health points of the bug. If the player succeeds in defeating the bug, the player will simply be rewarded by chance. For passive bugs, the whole cycle is almost exactly the same except that the player triggers the battle by clicking on the bug.

The player will also be given the opportunity to run away from the bugs to escape for safety and after a certain amount of time, the bug will simply remove itself from the *Battle System* and walk away. As you can see to the right in the figure, the *Battle Interface* will remain visible as long as the bug is chasing the player.

And because every player has a certain amount of health points, just like the bug, a player can also be defeated if they took too much damage from the bug as shown below. But once the player has been defeated, they will be given the opportunity to respawn back to the first environment.

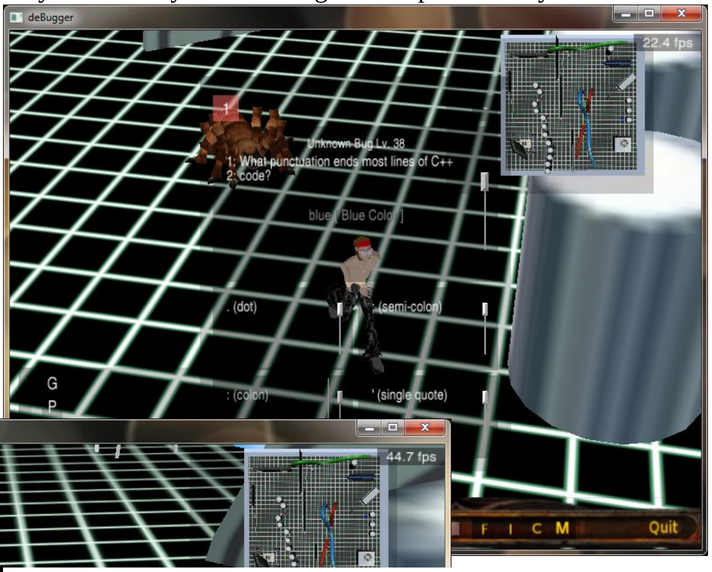


Figure 5.C.xii-5: Player Running Away

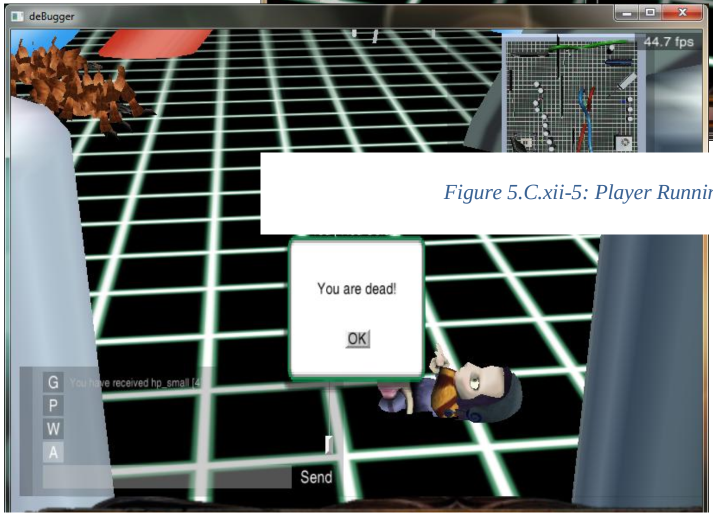


Figure 5.C.xii-6: Defeated!

addBug Method

This method will be used very often because this is the first method that will always be executed whenever a player encounters a bug. As shown below, the *self.bugList* container will be used to store all bugs that are trying to attack the player and it will simply return a confirmation.

```
self.bugList.append(bObject)
```

Once the bug has been added to the container, it will automatically check whether the bug contains more bugs than a player should be able to have. If the player already has the maximum number of bugs it can engage, the *Battle System* will simply remove the farthest bug in the container. This is to prevent players from mobbing up too many bugs at once.

```
self.removeBug(farthestTarget, True)
```

createBattle Method

Whenever the *self.bugList* container is not empty, the *Battle System* will simply try and create a battle between the player and the bug and stores it inside the *self.battleList* container. This container is one of the most important of all the containers because it keeps track of all the active battles that the player is in. This method makes use of the *Battle* class that simply tracks the bug target and the questions to be used within the *Battle System* as shown below.

```
self.battleList.append(Battle(len(self.battleList), bugObject, qObject))
```

executeAttack Method

This method will only be called whenever the player had successfully answered a question. This will simply tell the character to send an attack request, flag the question as completed and simply refreshes the battle interface for another question as shown below.

```
self.charObject.charAttack()

self.questionObject.setComplete()
```

selectBug Method

This method is responsible for updating the battle interface whenever a new bug is encountered or that the player had clicked on one of the tabs. It will simply refresh the battle interface, so that the bug name will appear. It also ensures that the player has selected the bug as the target as shown below.

```
self.bugObject = battleObject.getBug()
```

Once the bug has been selected as the target, it will try to access the *Battle* instance that corresponds to this bug and retrieves the question object out of it to display the corresponding question and responses.

```
self.questionObject = battleObject.getQuestion()
```

showAtkDelayBar Method

This method simply just display and fill up the attack delay bar as shown before to penalize the player for answering an incorrect response to a question. This is to just to slow down the player from trying again. It simply uses a method provided from the *Interval* module called *LerpFunc* as shown below.

```
self.atkDelayBarInterval = LerpFunc(incAtkDelayBar,

                                    fromData = 0,

                                    toData = 100,
```

The *LerpFunc* is like a loop that simply executes over and over for a certain period as shown above. It uses the player's attack speed as the timer. Whenever it sets the value, the attack delay bar will simply and visibly adjust its fill volume.

checkResponse Method

This is a very important method that is used to access the *Question* instance to validate the answer as shown below.

```
if (self.questionObject.checkResponse(self.responseButtonNum)):

    self.isCorrect = True
```

If the response is validated as correct, the *Battle System* will simply flag it to stop the validation process because the correctness has already been determined.

checkAvailableBattles Method

In order for a *Battle* instance to be created, this method must be executed from a loop in the main routine of the *Battle System* class. It basically checks if the *self.bugList* contains a bug and is within attack range and will simply try to create a *Battle* between the player and the bug to initiate the battle sequence as shown below.

```
if (destDistance <= self.charObject.atkRange):
```

checkEndedBattles Method

As mentioned very early about the *DirectObject* module using the method *accept*, whenever the *'endBattle'* event is generated from a *Battle* instance, the *accept* method will pick up this event and execute this exact method to remove that *Battle* instance from the *self.battleList* as shown below.

```
self.removeBattle(self.battleList[i])
```

battleSystemRoutine Method

Similar to the *Character* class, there is also a main routine that takes care of running certain methods to perform the necessary tasks to make the *Battle System* work. The only main purpose of this routine is to execute certain methods to validate a particular response for a question and also to create battles whenever needed. As shown below, the first thing this routine does is to call the *checkAvailableBattles* method.

```
self.checkAvailableBattles()
```

As mentioned above, this method will simply check against the *self.bugList* container for any existing bugs that have been targeted you and creates a *Battle* instance between the player and bug to initiate a battle. If a battle has been successfully initiated and if a player has attempted to answer a question, this routine would execute the necessary methods to validate the response as shown below.

```
if (len(self.battleList) > 0):

    ...

    if (self.toCheckResponse):
```

And if all bugs that currently exist in the *self.bugList* have been defeated, the *Battle Interface* will simply hide itself until another battle takes place.

unload Method

Like the *Character* class, the *Battle System* class also has an unload method that stops the *battleSystemRoutine* method from running and destroys the *Battle Interface* object from memory as shown below.

```
taskMgr.remove('battleSystemRoutine')
```

The only time that this method is called in the game is whenever a player switches map and logging off. For everything else, it simply hides itself visibly.

The Chat module defined in Chat.py file is located in main.World.Gui.Chat package.

The Chat module consists of two main classes:

CHAT

Class Chat

The Chat class consists of the following functions:

1. def **__init__**(*self*, world)
2. def **createMainFrame**(*self*)
3. def **createButtons**(*self*)
4. def **createLog**(*self*)
5. def **createEntry**(*self*)
6. def **setChatMode**(*self*,chatMode,focus = True)
7. def **startDragMode**(*self*, param)
8. def **stopDragMode**(*self*, param)
9. def **sendEvent**(*self*)
10. def **setFocus**(*self*,state = None)
11. def **toggleChatMode**(*self*,direction)
12. def **sendMsg**(*self*)
13. def **parseMsg**(*self*,name,type,msg,send = True)
14. def **addMessage**(*self*,name,type,msg)
15. def **updateScrollBar**(self)
16. def **scrollChatLog**(self)
17. def **toggleVisibility**(*self*)
18. def **filterChat**(self)
19. def **receiveMsg**(self,user_id = -1, name = ‘’, type = -1, msg = ‘’)
20. def **clearChatEntry**(self)
21. def **clearChatBox**(self)
22. def **chatRoutine**(self,task)
23. def **unload**(self)

Following is a brief description of each of the above listed functions.

1. def **__init__**(*self*, world)

The __init__(*self*, world) is the constructor for the Chat class. The chat system in Debugger consists of 4 types of chat i.e. global chat, party chat, whisper chat and public chat. Due to which the constructor defines several chat variables to differentiate between these chat modes. At any point of time a game player can switch into any chat mode by pressing its equivalent button either by making use of mouse or by using keyboard. Thus, the chat box is accessible using keyboard controls which are defined in the constructor. As seen in the below code snippet, the constructor of the Chat class makes use of **'!'**,

'%', *''* and *'/'* operators to distinguish whether the game player is in global, party, public or whisper chat mode. Also to distinguish between the chat types, the text of each chat type is attributed with a different color, for example, *self*.globalColor = (1.0, 0.0, 0.0, 1.0).

The constructor also declares various functions, for example, *self*.createMainFrame() and *self*.createButtons() as the name indicates are used for creating the interface (frame) and the buttons of the chat box. By default, when a player first logs into the game his default chat mode is set to public chat as defined by *self*.setChatMode(Constants.CMSG_PUBLIC_CHAT, False) function. The function *self*.createLog() is used for logging the chat entries into an array. The constructor also declares 2 functions namely startDragMode(self, param) and stopDragMode(self, param) through which the chat box can be dragged and placed anywhere on the screen. Lastly, the constructor declares a task named *'chatRoutine'* which is handled by the global Task Manager object referred in Panda3D as taskMgr. The task function

self.chatRoutine is added to the task list by calling taskMgr.add() which takes 2 input arguments i.e. task function and task name.

```
def __init__(self, world):

    self.world = world

    # Chat Box Variables
    self.isFocus = False

    self.chatEntry = None
    self.globalButton = None
    self.globalGlowSequence = None
    self.mainFrame = None
    self.partyButton = None
    self.partyGlowSequence = None
    self.publicButton = None
    self.publicGlowSequence = None
    self.scrollBar = None
    self.sendButton = None
    self.titleBar = None
    self.whisperButton = None
    self.whisperGlowSequence = None

    self.chatLog = []
    self.filterLog = []

    self.chatButtons = []
    self.vChatLog = []

    self.globalColor = (1.0, 0.0, 0.0, 1.0)
    self.partyColor = (1.0, 0.7, 0.0, 1.0)
    self.publicColor = (1.0, 1.0, 1.0, 1.0)
    self.rewardColor = (0.7, 0.7, 0.7, 1.0)
    self.systemColor = (1.0, 1.0, 0.0, 1.0)
    self.whisperColor = (0.0, 0.4, 1.0, 1.0)

    self.globalOp = '!'
    self.partyOp = '%'
    self.publicOp = ''
    self.whisperOp = '/'
    self.newWhisperOp = self.whisperOp

    self.chatOp = (self.globalOp,
            self.partyOp,
            self.whisperOp)

    self.chatMode = -1
    self.lastFilterSize = 0
    self.lastType = -1
    self.maxItems = 30
    self.maxItemsVisible = 6
    self.modeIndex = -1
    # End of Chunk

    # Create Chat Box
    self.createMainFrame()
    self.createButtons()
    self.createLog()
    self.createEntry()
    self.setChatMode(Constants.CMSG_PUBLIC_CHAT, False)
    # End of Chunk

    # Allow Draggable Window (Using Title Bar)
    self.titleBar.bind(DGG.B1PRESS, self.startDragMode)
    self.titleBar.bind(DGG.B1RELEASE, self.stopDragMode)
    # End of Chunk

    # Chat Box Control Scheme
    self.accept('enter', self.sendEvent)
    self.accept('mouse1', self.setFocus, [False])
    self.accept('tab', self.toggleChatMode, [1])
    self.accept('shift-tab', self.toggleChatMode, [-1])
    # End of Chunk

    taskMgr.add(self.chatRoutine, 'chatRoutine')
```

2. def **createMainFrame**(*self*)

The createMainFrame(*self)* function as the name indicates is used for creating the interface or outer frame of the chat box. As seen in the below code snippet of createMainFrame(*self)* function creates two DirectFrame objects named self.mainFrame and self.titleBar. The self.titleBar object is made to appear on self.mainFrame object by using the reparentTo() method.

```
def createMainFrame(self):

    # Create Main Frame
    self.mainFrame = DirectFrame( frameColor = (0.0, 0.0, 0.0, 0.2),
                                  frameSize = (-0.6, 0.6, -0.3, 0.2),
                                  pos = (-0.65, 0.0, -0.50))

    self.titleBar = DirectFrame( frameColor = (0.0, 0.0, 0.0, 0.3),
                                frameSize = (-0.025, 0.025, -0.3, 0.2),
                                pos = (-0.625, 0.0, 0.0),
                                state = DGG.NORMAL )
    self.titleBar.reparentTo(self.mainFrame)
    # End of Ch
```

3. def **createButtons**(self)

The createButtons(*self)* function is used for creating four different buttons by clicking which a player can toggle between the 4 types of available chat mode. Below is the partial code snippet of createButtons(*self)* function which shows the creation of one of the buttons i.e. global button which on clicked a player enters into global chat mode. The button object self.globalButton is of type DirectButton whose constructor takes various parameters such as the text that should appear on the button, the foreground color, position and scale of the text, frame color, size and position of the button. Whenever the global button is clicked the button executes the **setChatMode**(self,chatMode,focus = True) which is passed an argument of Constants.CMSG_GLOBAL_CHAT as defined in extraArgs. The self.globalButton is made to appear on the self.mainFrame object using reparentTo() method. Finally the global button is added to an array named self.chatButtons as declared in the constructor of the Chat class.

```
def createButtons(self):
```

```
# Nested Function
def setFrameColor(chatButton, color):
    chatButton['frameColor'] = color
# End of Chunk

# Create Global Button
self.globalButton = DirectButton( text = 'G',
                                  text_fg = (1.0, 1.0, 1.0, 1.0),
                                  text_pos = (-0.002, -0.02),
                                  text_scale = 0.065,
                                  frameColor = (0.0, 0.0, 0.0, 0.2),
                                  frameSize = (-0.04, 0.04, -0.04, 0.04),
                                  pos = (-0.52, 0.0, 0.14),
                                  command = self.setChatMode,
                                  extraArgs = [Constants.CMSG_GLOBAL_CHAT],
                                  relief = DGG.FLAT )
self.globalButton.reparentTo(self.mainFrame)

self.globalGlowSequence = Sequence( Func(setFrameColor, self.globalButton, (1.0, 1.0, 1.0, 0.3)),
                                    Wait(0.3),
                                    Func(setFrameColor, self.globalButton, (0.0, 0.0, 0.0, 0.2)),
                                    Wait(0.3),
                                    Func(setFrameColor, self.globalButton, (1.0, 1.0, 1.0, 0.3)),
                                    Wait(0.3),
                                    Func(setFrameColor, self.globalButton, (0.0, 0.0, 0.0, 0.2)),
                                    Wait(0.3),
                                    Func(setFrameColor, self.globalButton, (1.0, 1.0, 1.0, 0.3)),
                                Wait(0.3),

                                Func(setFrameColor, self.globalButton, (0.0, 0.0, 0.0, 0.2)) )


        self.chatButtons.append(self.globalButton)

        # End of Chunk
```

4. def **createLog**(self)

The createLog(*self)* function creates an object chatLogLine of type DirectLabel which is basically a text string. The maximum number of lines visible in the chat frame is equal to 6 as defined by the *self*.maxItemsVisible declared in the Chat constructor. The for loop creates 6 number of DirectLabel object whose position is changed in text_pos.

By making use of reparentTo() method the chatLogLine object is made to appear on self.mainFrame object. The chatLogLine is then appended to an array self.vChatLog as declared in Chat constructor. The createLog(*self)* function also creates a scrollbar whose orientation is set to VERTICAL. The self.scrollBar calls the scrollChatLog(self) function when the value of the scrollbar changes. Finally, the scrollbar is made to appear on self.mainFrame object by calling the reparentTo() method.

```
def createLog(self):


        # Create Chat Box Log

        for i in range(self.maxItemsVisible):

            chatLogLine = DirectLabel( text = '',

                                text_pos = (-0.459, 0.125 - i * 0.05),

                                text_scale = 0.045,

                                text_fg = (1.0, 1.0, 1.0, 1.0),

                                frameColor = (0.0, 0.0, 0.0, 0.0),

                                text_align = TextNode.ALeft,

                                textMayChange = 1 )

            chatLogLine.reparentTo(self.mainFrame)

            self.vChatLog.append(chatLogLine)
        self.scrollBar = DirectSlider( range = (1, 0),
                                value = 1,
                                scale = 0.15,
                                pos = (0.545, 0.0, 0.01),
                                thumb_frameSize = (-0.1, 0.1, -0.25, 0.25),
                                pageSize = 1,
                                scrollSize = 1,
                                orientation = DGG.VERTICAL,
                                command = self.scrollChatLog )
        self.scrollBar.reparentTo(self.mainFrame)
        # End of Chunk
```

5. def **createEntry**(*self*)

The createEntry(*self*) function first creates a self.chatEntry field of type DirectEntry. It is basically a textbox wherein a game player can enter chat text. As can be seen in the below code snippet of createEntry(*self*), one of the arguments of DirectEntry is focusInCommand which is set to self.setFocus function. This function gets called whenever the self.chatEntry field gains focus. The textbox is made to appear on self.mainFrame object by calling the reparentTo() method.

The createEntry(*self*) function also creates a Send button which when clicked calls the self.sendEvent function. The Send button is made to appear on self.mainFrame object by making use of the reparentTo() method.

```
def createEntry(self):

        # Create Chat Box Entry
        self.chatEntry = DirectEntry( text = '',
                               frameColor = (0.8, 0.8, 0.8, 0.4),
                               pos = (-0.555, 0.0, -0.248),
                              scale = 0.05,
                              width = 19.0,
                              focusInCommand = self.setFocus )
        self.chatEntry.reparentTo(self.mainFrame)


        self.sendButton = DirectButton( text = 'Send',
                               text_fg = (1.0, 1.0, 1.0, 1.0),
                               text_pos = (-0.003, -0.02),
                               text_scale = 0.065,
                               frameColor = (0.0, 0.0, 0.0, 0.2),
                               frameSize = (-0.08, 0.08, -0.045, 0.045),
                               pos = (0.495, 0.0, -0.225),
                               command = self.sendEvent,
                               relief = DGG.FLAT )
        self.sendButton.reparentTo(self.mainFrame)
        # End of Chunk
```

6. def **setChatMode**(self,chatMode,focus = True)

The setChatMode(*self*,chatMode,focus = True) function is called from the constructor of the Chat class. One of the arguments passed to this function is Constants.CMSG_PUBLIC_CHAT which is initialized with a value of 68 and is defined in Constants.py located in common package. Thus when a player logs into the game his/her default chat mode is set to public chat. The second argument that is passed on to this function sets the focus of the chat box to True. Depending on the chat mode that is selected the setChatMode() function initializes the field glowButtonSequence as shown in the below code snippet.

```
if (chatMode == Constants.CMSG_GLOBAL_CHAT):
      glowButtonSequence = self.globalGlowSequence
elif (chatMode == Constants.CMSG_PARTY_CHAT):
      glowButtonSequence = self.partyGlowSequence
```

```
elif (chatMode == Constants.CMSG_PUBLIC_CHAT):
        glowButtonSequence = self.publicGlowSequence
elif (chatMode == Constants.CMSG_PRIVATE_CHAT):
        glowButtonSequence = self.whisperGlowSequence
```

Depending on the chat mode selected, it also sets the frame color of the chat button as shown in the below code snippet for global chat.

```
if (chatMode == Constants.CMSG_GLOBAL_CHAT):
    self.globalButton['frameColor'] = (1.0, 1.0, 1.0, 0.3)
    self.modeIndex = self.chatButtons.index(self.globalButton)
    chatOp = self.globalOp
```

The setChatMode() function also initializes the range and value of the chat scroll bar as shown below:

```
self.scrollBar['range'] = (1, 0)
self.scrollBar['value'] = 1.0
```

7. def **startDragMode**(self, param)

The startDragMode(*self*, param) function is used for dragging the chat box window anywhere on the game screen with the help of a mouse. The constructor of the Chat class contains the following line:

```
self.titleBar.bind(DGG.B1PRESS, self.startDragMode)
```

used for binding the self.titleBar DirectFrame object to the command *self*.startDragMode for the event B1PRESS. Below is the code snippet of startDragMode(*self*, param) function which is used for binding the parent frame i.e. self.mainFrame to mouse.

```
def startDragMode(self, param):
    # Bind Main Frame (Parent Frame) To Mouse
    self.mainFrame.wrtReparentTo(self.world.mPicker.aspect2dMouseNode)
    # End of Chunk
```

8. def **stopDragMode**(self, param)

The stopDragMode(*self*, param) function drops the chat box window at the location where the mouse was released on the game screen. The constructor of the Chat class contains the following line:

self.titleBar.bind(DGG.B1RELEASE, self.stopDragMode)

used for binding the self.titleBar DirectFrame object to the command *self*.stopDragMode for the event B1RELEASE. Below is the code snippet of stopDragMode(*self*, param) function which is used for releasing the parent frame i.e. self.mainFrame from mouse.

```
def stopDragMode(self, param):

    # Release Main Frame (Parent Frame) From Mouse
    self.mainFrame.wrtReparentTo(aspect2d)
    # End of Chunk
```

9. def **sendEvent**(self)

Debugger's chat box has few keyboard controls associated with it. "Enter" key is one such control which when pressed calls the sendEvent(*self*) function. This function sets the focus to True if it is False or else will send the chat message typed by the player to the required player(s) within a chat mode.

```
def sendEvent(self):
    # Focus / Send Message
    if (self.isFocus is False):
        self.setFocus(True)
    else:
        self.sendMsg()
    # End of Chunk
```

10. def **setFocus**(*self*, state = None)

In addition to keyboard control, debugger's chat box focus can be set depending on whether the cursor is placed in the chat text box by using a mouse. If the cursor is placed in the chat's text box, the textbox appears highlighted and when mouse is clicked anywhere outside the chat box then it the textbox appears in dull shade color. Below is the code snippet for setFocus() function in which depending on the state, the self.chatEntry's focus and frame color is changed accordingly.

```
def setFocus(self, state = None):

    # Set Chat Box Entry Focus
    if (state is False):
        self.chatEntry['focus'] = 0
        self.isFocus = False
        self.chatEntry['frameColor'] = (0.8, 0.8, 0.8, 0.4)
        if (self.lastType < 0):
            self.world.charHero.hideChatBubble()
    else:
        if (state is True):
            self.chatEntry['focus'] = 1
        self.isFocus = True
        self.chatEntry['frameColor'] = (0.8, 0.8, 0.8, 0.9)
    # End of Chunk
```

11. def **toggleChatMode**(*self*,direction)

The toggleChatMode() function implements toggling through the 4 different chat modes using keyboard keys i.e. by either using Tab key or Shift-Tab key. The Chat class constructor contains the following lines of code:

```
self.accept('tab', self.toggleChatMode, [1])

self.accept('shift-tab', self.toggleChatMode, [-1])
```

Thus when either tab/shift-tab event is sent it calls the toggleChatMode() function which takes a single argument named direction with a value of 1 when tab key is pressed and -1 when shift-tab key is pressed. Below is the code snippet of toggleChatMode() function which depending on which chat button is selected sets the chat mode by calling the setChatMode() function.

```
def toggleChatMode(self, direction):

    # Toggle Through Chat Modes Using Tab / Shift-Tab
    if (self.isFocus is True):
        if (direction < 0):
            self.modeIndex -= 1
            if (self.modeIndex < 0):
                self.modeIndex = len(self.chatButtons) - 1
        elif (direction > 0):
            self.modeIndex += 1
            if (self.modeIndex > len(self.chatButtons) - 1):
                self.modeIndex = 0

        bObject = self.chatButtons[self.modeIndex]
        if (bObject is self.globalButton):
```

```
        self.setChatMode(Constants.CMSG_GLOBAL_CHAT)
    elif (bObject is self.partyButton):
        self.setChatMode(Constants.CMSG_PARTY_CHAT)
    elif (bObject is self.publicButton):
        self.setChatMode(Constants.CMSG_PUBLIC_CHAT)
    elif (bObject is self.whisperButton):
        self.setChatMode(Constants.CMSG_PRIVATE_CHAT)
  # End of Chunk
```

12. def **sendMsg**(self)

The sendMsg() function as the name indicates is used to send the chat message to other game player(s) depending on the chat mode. The sendMsg() function parses the text message by calling parseMsg() function. It then clears the entry in the chat text box by calling clearChatEntry() function. As seen in the below code snippet, sendMsg() function sends a request to the server using the following line of code:

```
if (type == Constants.CMSG_GLOBAL_CHAT):
    self.world.main.cManager.sendRequest(Constants.CMSG_GLOBAL_CHAT, rContents)
```

in which arguments to the sendRequest() function contains the constant field –

Constants.CMSG_GLOBAL_CHAT = 14 defined in Constants.py file located in common package and the chat message. The sendMsg() function calls the addMessage() function which is used for adding/displaying the message in the chat box. It finally sets the chat bubble in which the typed message appears and disappears after some time.

```
def sendMsg(self):

    # Send Message To Others
    msg = self.chatEntry.get()
    result = self.parseMsg(self.world.charHero.getName(), self.chatMode, msg)
```

```
self.clearChatEntry()

if (result != None):
    (name, type, msg, targetUser, newMsg) = result

    self.lastType = type

    rContents = {'msg': msg}
    if (type == Constants.CMSG_GLOBAL_CHAT):
        self.world.main.cManager.sendRequest(Constants.CMSG_GLOBAL_CHAT, rContents)
    elif (type == Constants.CMSG_PARTY_CHAT):
        self.world.main.cManager.sendRequest(Constants.CMSG_PARTY_CHAT, rContents)
    elif (type == Constants.CMSG_PUBLIC_CHAT):
        self.world.main.cManager.sendRequest(Constants.CMSG_PUBLIC_CHAT, rContents)
    elif (type == Constants.CMSG_PRIVATE_CHAT):
        rContents = {'targetName': targetUser,
                     'msg': msg}
        self.world.main.cManager.sendRequest(Constants.CMSG_PRIVATE_CHAT, rContents)

    self.addMessage(name, type, newMsg)

    if (type != -1):
        self.world.charHero.setChatBubble(type, msg)
    self.setFocus(False)
elif (self.isFocus is False):
    self.setFocus(True)
else:
    self.setFocus(False)

    self.world.charHero.hideChatBubble()

# End of Chunk
```

13. def **parseMsg**(*self*,name,type,msg,send = True)

The parseMsg() function is used by sendMsg() function as show below

```
result = self.parseMsg(self.world.charHero.getName(), self.chatMode, msg)
```

The input arguments to parseMsg() function consists of the name of the player, the chat mode the player is in and the chat message. Since Debugger makes use of *'!'*, *'%'*, *''* and *'/'* operators to distinguish whether the game player is in global, party, public or whisper chat mode, parseMsg() first strips out the appropriate operator depending on the chat mode and finally returns the new message. The return type also includes name of the player, chat type and receiver(s) of the message.

14. def **addMessage**(*self*, name, type, msg)

The addMessage() function is used for adding messages into the chat box. Its input parameters include name of the player, chat type and chat message. It creates a ChatObject which is added to the chat log. The addMessage() function finally calls updateScrollBar() and scrollChatLog() functions. Below is the code snippet of the same.

```
def addMessage(self, name, type, msg):

    # Add Message into Chat Box

    textNode = TextNode('text')

    textNode.setText(msg)

    textNode.setWordwrap(21.0)

    pMessage = textNode.getWordwrappedText().split('\n')
```

```
        for mObject in pMessage:
            if (len(self.chatLog) == self.maxItems):
                self.chatLog.pop(0)

            self.chatLog.append(ChatObject(name, type, mObject))

            if (self.chatMode != Constants.CMSG_PUBLIC_CHAT):
                self.filterChat()

            self.updateScrollBar()
            self.scrollChatLog()
        # End of Chunk
```

15. def **updateScrollBar**(*self*)

The updateScrollBar() function is used to update the range and value of the scroll bar when the text to be displayed in the chat box exceeds that of the maximum number of visible items i.e. 6. Below is the code snippet for the updateScrollBar() function.

```
def updateScrollBar(self):

        if (self.chatMode == Constants.CMSG_PUBLIC_CHAT):
            chatLog = self.chatLog
        else:
            chatLog = self.filterLog

        if (len(chatLog) > self.maxItemsVisible):
            scrollRange = len(chatLog) - self.maxItemsVisible
```

```
    else:
        scrollRange = 1

    lastRange = self.scrollBar['range'][0]
    self.scrollBar['range'] = (scrollRange, 0)

    if (round(self.scrollBar['value']) >= lastRange):
        self.scrollBar['value'] = scrollRange
    elif (round(self.scrollBar['value']) > 0.0):
        if (self.chatMode == Constants.CMSG_PUBLIC_CHAT and len(chatLog) >= self.maxItems or
                self.chatMode != Constants.CMSG_PUBLIC_CHAT and len(chatLog) ==
self.lastFilterSize):
            self.scrollBar['value'] = round(self.scrollBar['value']) - 1.0
```

16. def **scrollChatLog**(self)

The scrollChatLog() function incorporates the scrolling capability for the chat log.

```
def scrollChatLog(self):

    # Scrolling Capability for Chat Log
    sliderValue = int(round(self.scrollBar['value']))

    if (self.chatMode == Constants.CMSG_PUBLIC_CHAT):
        chatLog = self.chatLog
    else:
        chatLog = self.filterLog

    if (len(chatLog) < self.maxItemsVisible):
```

```
    maxItems = len(chatLog)
else:
    maxItems = self.maxItemsVisible


for i in range(maxItems):
    chatType = None


    if (len(chatLog) > self.maxItemsVisible):
        self.vChatLog[i]['text'] = chatLog[sliderValue + i].msg
        chatType = chatLog[sliderValue + i].type
    else:
        self.vChatLog[i]['text'] = chatLog[i].msg
        chatType = chatLog[i].type


    if (chatType == Constants.CMSG_GLOBAL_CHAT):
        self.vChatLog[i]['text_fg'] = self.globalColor
    elif (chatType == Constants.CMSG_PARTY_CHAT):
        self.vChatLog[i]['text_fg'] = self.partyColor
    elif (chatType == Constants.CMSG_PUBLIC_CHAT):
        self.vChatLog[i]['text_fg'] = self.publicColor
    elif (chatType == Constants.SMSG_REQ_ITEM):
        self.vChatLog[i]['text_fg'] = self.rewardColor
    elif (chatType == Constants.CMSG_PRIVATE_CHAT):
        self.vChatLog[i]['text_fg'] = self.whisperColor
    else:
        self.vChatLog[i]['text_fg'] = self.systemColor
# End of Chunk
```

17. def **filterChat**(self)

The filterChat() function is used to filter out logs of chat which are not of type public chat.

```
def filterChat(self):

    self.lastFilterSize = len(self.filterLog)

    if (self.lastFilterSize > 0):
        self.filterLog = []

    for cObject in self.chatLog:
        if (cObject.type == self.chatMode or cObject.type == -1):
            self.filterLog.append(cObject)
```

18. def **toggleVisibility**(self)

The toggleVisibility() function implements the show/hide feature of chat box. By default the chat box is visible when a player logs into the game and can be hidden by clicking on the Chat button. Below is the code snippet of the same:

```
def toggleVisibility(self):

    # Toggle Visibility
    if (self.mainFrame.isHidden()):
        self.mainFrame.show()
    else:
        self.mainFrame.hide()
    # End of Chunk
```

19. def **receiveMsg**(self,user_id = -1, name = ‘’, type = -1, msg = ‘’)

The receiveMsg() function implements the functionality of receiving chat messages from other game player(s). In addition to starting the glowButtonSequence depending on the type of chat mode, receiveMsg() function also calls the addMessage() function which adds the message into the chat box.

20. def **clearChatEntry**(self)

The clearChatEntry() function is used for clearing the chat box entry. Depending on the chat mode it sets the self.chatEntry’s text with either of these '!', '%', '' and '/' values depending on whether the game player is in global, party, public or whisper chat mode.

```
def clearChatEntry(self):

    # Clear Chat Box Entry
    if (self.chatMode == Constants.CMSG_GLOBAL_CHAT):
        self.chatEntry.enterText(self.globalOp)
    elif (self.chatMode == Constants.CMSG_PARTY_CHAT):
        self.chatEntry.enterText(self.partyOp)
    elif (self.chatMode == Constants.CMSG_PUBLIC_CHAT):
        self.chatEntry.enterText('')
    elif (self.chatMode == Constants.CMSG_PRIVATE_CHAT):
        self.chatEntry.enterText(self.newWhisperOp)
    # End of Chunk
```

21. def **clearChatBox**(self)

The clearChatBox function clears the text in *self*.vChatLog array to empty string. Below is the code snippet of the same.

```
def clearChatBox(self):

    # Clear Chat Box
    for cObject in self.vChatLog:
        cObject['text'] = ''
    # End of Chunk
```

22. def **chatRoutine**(self)

The chatRoutine() function is a task function which is added to the task list by calling taskMgr.add() in the Chat's constructor. The function returns task.cont which indicates that the chatRoutine() function will be called again in the next frame. This function keeps on running until the game is quit. Below is the code snippet of the same.

It always sets the chat mode to the default public chat. Whenever a player types into the chat text box it parses the chat message by calling the parseMsg() function. When a player is typing in the chat text box and is not done yet the text *'Typing...'* is displayed over the character's model.

```
 def chatRoutine(self, task):

    if (self.chatMode != Constants.CMSG_PUBLIC_CHAT):
        if (len(self.chatEntry.get()) == 0):
            self.setChatMode(Constants.CMSG_PUBLIC_CHAT)

    if (self.isFocus is True):
        type = None
        msg = self.chatEntry.get()
        result = self.parseMsg(self.world.charHero.getName(), self.chatMode, msg, send = False)

        if (result != None):
            type = result[1]
            msg = result[2]
        else:
            msg = 'Typing...'

        if (type != -1 and msg != self.world.charHero.getChatBubble()):
            self.world.charHero.setChatBubble(-1, msg, fade = False)
            self.lastType = -1

    return task.cont
```

23. def **unload**(self)

The def unload() function is analogous to a destructor or can be viewed as a cleanup function when the game player exits from the game. It is used for destroying all the objects that have been used in the chat i.e. it removes the mainFrame by using the removeNode() method. It also deletes the content from the chat log and initializes it as an empty array. Finally it also pauses the glowing sequence of chat buttons. Below is the code snipped for unload() function.

```
def unload(self):

    # Unload Chat
    self.ignoreAll()

    self.mainFrame.removeNode()
```

```
        for i in range(len(self.chatLog)):
            del self.chatLog[0]

        self.chatLog = []

        for i in range(len(self.vChatLog)):
            del self.vChatLog[0]

        self.vChatLog = []

        if (self.globalGlowSequence != None):
            self.globalGlowSequence.pause()

        if (self.partyGlowSequence != None):
            self.partyGlowSequence.pause()

        if (self.publicGlowSequence != None):
            self.publicGlowSequence.pause()

        if (self.whisperGlowSequence != None):
            self.whisperGlowSequence.pause()
        # End of Chunk
```

Class ChatObject

The ChatObject class consists a constructor as shown below is made use in the addMessage() function defined in the Chat class.

```
class ChatObject:

    def __init__(self, name, type, msg):

        self.name = name
        self.type = type
        self.msg = msg
```

DEBUGGER WEBSITE

Foreword:

The deBugger website was written in PHP with the main presentation layer in HTML and CSS. The upside to the way the site was designed allows this part of the documentation be very short, as everything is simple and repeats itself. Therefore, we will be explaining it once, as the other parts of the site will follow the same format with only one change; content.

Overview:

We will briefly talk about how to create an executable, because packpanda itself has a self explanatory manual once you run the command line argument. Then we shall talk about the general structure of the website, and go into the coding used to display the one section of the website. This will give you the reader the knowledge required to add more sections, since they will be simple changes to the code.

PackPanda:

Pack Panda allows you to create an installer in either windows (.exe) or linux (.rpm) depending on what environment you install. We will go over the .exe version because presumably, the rpm version is identical.

Running the packpanda command from command line gives you:

```
packpanda

PANDA located at C:\Panda3D-1.6.2

You must specify the --dir option.
```

packpanda usage:

```
  --dir x        Name of directory containing game

  --name x       Human-readable name of the game

  --version x    Version number to add to game name

  --rmdir x      Delete all directories with given name

  --rmext x      Delete all files with given extension
```

--fast	Use fast compression instead of good compression
--bam	Generate BAM files, change default-model-extension to BAM
--pyc	Generate PYC files

The important options are the –dir, --name, --rmdir, --rmext, --bam, and –pyc. Only of which –dir is required.

--dir tells panda what directory has the Main.py file. Our game used Launcher.py to launch the game, packpanda does NOT have an option to set this, so you must rename the file to Main.py for the installer to be packed. You may rename it back if you wish, but this is the required setting of packpanda.

--name lets you pick the option of giving your game a name.

--rmdir allows you to remove certain directories. In this case, since we were using the production version of the game, we had to remove the svn files.

--rmext removes files with the extension. We would like to remove the .py and .egg files because these are source files and should not be made readily available.

--bam and –pyc files both compile the .egg and .py files into their compiled forms. This makes the game a lot faster to load.

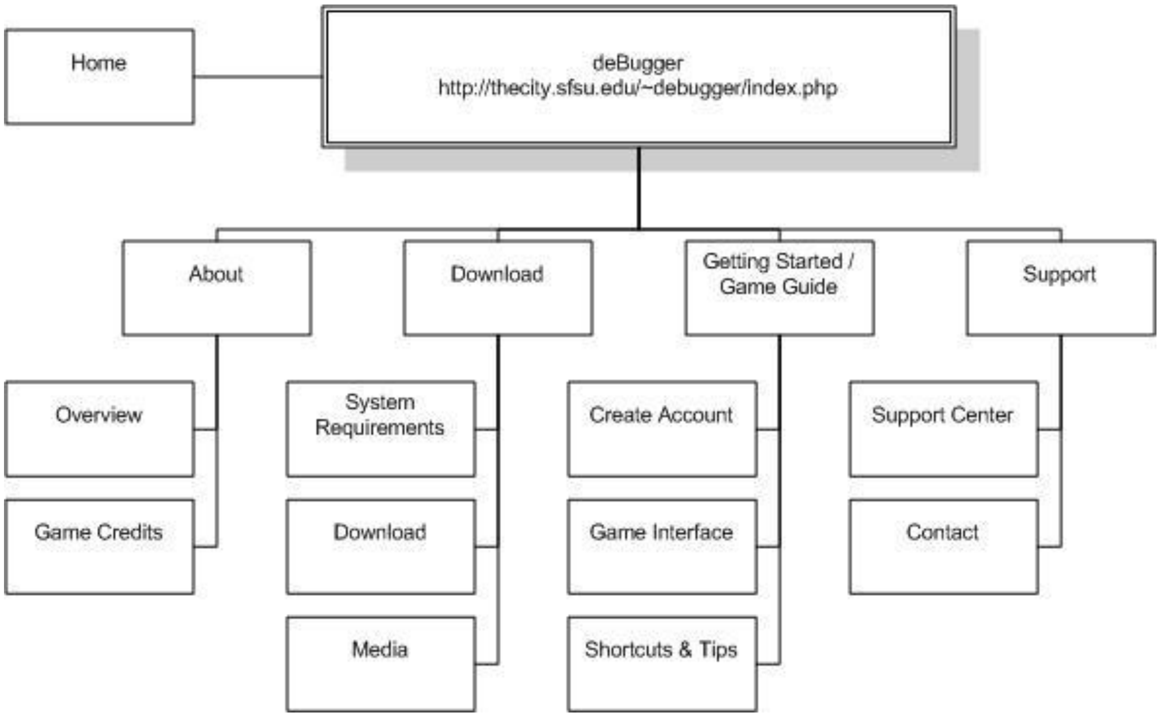
Using these options, the proper command to run from the top of the src directory is:

```
Packpanda –dir src –name “deBugger – SFSU” –bam –rmext egg –pyc –rmext py –rmdir svn
```

This will give the game a name, compile and remove all .egg and .py files, and remove svn directories from the packaged game.

Website Model:

Below is a tree structure diagram of the website with all of the directories (first layer) and files (second layer). We shall be doing the Download section of the website (Directory and File)



5-1 – Tree Structure of deBugger's website.

Index.php:

deBuggers index contains a few php calls that includes various files. These files handle the header and footer of the website. By changing these, you will change all headers and footers, thus making it easy to change a header or footer:

```
<body>

    <?php include("ssi/heading.html"); ?>

    <div id="main_content_1">

        <div id="main_content_2">



            <div id="left_column">
```

```
                <?php include("ssi/home.html"); ?>

            </div>



            <div id="right_column">



            </div>



            <div id="middle_column">

                <div class="column_inner">

                <h2>Welcome to Debugger</h2>

                <p>The Debugger gaming project by the CSC 631/ CSC 831 course is a class effort
to create a functional, enjoyable, and educational Massively Multiplayer Online Role Playing G$



                <p>Debugger was developed during the Multiplayer Game Development class (CSC
631/CSC 831) at San Francisco State University taught by Dr. Ilmi Yoon during the Fall semester o$



                <p>Debugger was developed using Panda3D 1.6.2 - a 3D Game Engine which
provides a library of subroutines for 3D rendering and game development. Debugger's client was
develope$

                </div>

            </div>

        </div>

    </div>

    <div class="cleaner">&nbsp;</div>

    <?php include("ssi/footer.html"); ?>

</body>
```

Also note that the main colum_inner div is made up of the news section. Again, we could simply use a php include to make it load from a file.

Heading.html:

The heading file contains all of the links that you see on the main page once you go to index.php. With this file, you can add new tabs and new subsections. Below is the section that handles all of the tabs and sections.

```
<div id="navigation">

……

        <li><a
href="http://thecity.sfsu.edu/~debugger/download/download.php">Download<span></span></a>

            <ul>

                <li><a href="http://thecity.sfsu.edu/~debugger/download/sysrequire.php">System
Requirements</a></li>

                <li><a
href="http://thecity.sfsu.edu/~debugger/download/download.php">Downloads</a></li>

                <li><a
href="http://thecity.sfsu.edu/~debugger/download/media.php">Media</a></li>

            </ul>

        </li>

……

</div>
```

The typical format for adding a new section is to:

```
      <li>

            <ul>For a new section

                  <li>Each section</li>

            </ul>

      </li>
```

Download/Download.php:

The final part of the process is to make the directory for the new section. Since we will be using the download directory as the example, we will simply paste the code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

……..


<body>

    <?php include("../ssi/heading.html"); ?>

    <div id="main_content_1">

        <div id="main_content_2">


            <div id="left_column">

            </div>

…….


                <div class="column_inner">

                    <h2>Learn 2 deBugger</h2>

                    <p>Diving right into the game and getting started with deBugger is quite simple,
we already have an installer made just for you!</p>

                    <p>All you need to do is download the included setup file, and run it, which will
install given the various prompts.</p>

                    <p>We'll walk you through the process of installing the game, just to make it a
bit simpler. We promise, installing a real deBugger will not be this easy!</p>

                    <p>So click on the following link, and run the setup file to get started.</p>

                    <a href="deBugger.exe">Click to Download deBugger</a>

…..
```

As you notice, this file formatting takes on the same formatting of index.php. There are better ways of doing this, such as using php code to load the index.php and simply including the file, which is what many sites do. But for simplicity and time constraints, this is how the website was designed.

Note:

When you log into the debugger account, you must make sure all directories are properly chmodded to 755, as well as all files. Otherwise, you will get a permission error.

Conclusion:

The website is very basic in its current form. However, it has all of the required information that any gaming website would have, in a layout as well as organized in a matter which is concise and easy to understand. We would hope that future implementers add some Web 2.0 into the site, to allow more functionality such as forums, including the Question Creator, and AJAX capabilities.

NETWORK COMMUNICATION

Purpose

This document describes how the network communication is implemented in the debugger game. This document describes how the protocols are sent from the game client and bug server and received from the java server.

A computer network allows computers to communicate with other computers and share resources and information. There are different types of networks (Local Area Network [LAN], Campus Area Network [CAN], Metropolitan Area Network [MAN], Wide Area Network [WAN], Global Area Network [GAN], Virtual Private Network [VPN], etc.).

The "deBugger" has been tested to work over a WAN. There is some lag. However, the game still functions as planned. This document describes in detail how the "deBugger" game client sends data to the Java Server through a network and how it receives data from the server through the network.

We use a TCP connection to send and receive data to/from the server. TCP provides reliable, ordered delivery of a stream of bytes from a program on one computer to another program on another computer. TCP provides a communication service at an intermediate level between an application program and the Internet Protocol (IP). TCP detects unpredictable network behavior, requests retransmission of lost packets, rearranges out of order packets, and helps minimize network congestion to reduce occurrence of other problems.

TCP is optimized for accurate delivery rather than timely delivery; hence, at times there is lag between a request and a response of data. TCP guarantees delivery of a data packet sent from one host without duplication or loss of data.

Specifications

Connection Manager

The connection manager, as the name suggests, manages all the requests sent by the client and the responses sent by the server.

```
def __init__(self, main):


        self.main = main


        self.cManager = QueuedConnectionManager()

        self.cListener = QueuedConnectionListener(self.cManager, 0)
        self.cReader = QueuedConnectionReader(self.cManager, 0)

        self.cWriter = ConnectionWriter(self.cManager, 0)


        self.connection = None
```

This is the constructor of the Connection manager. The QueuedConnectionReader accepts incoming packages. The QueuedConnectionManager is used to open a connection between the client and the server. The QueuedConnectionListener is used to specify a certain port address to listen to and the number of connections that can be ignored before an error is generated. We have not used this feature in our application though it can be used later. The ConnectionWriter is used to send data back to the server.

```
def startConnection(self):


      try:

            if (self.connection == None):

                self.connection =
self.cManager.openTCPClientConnection(Constants.SERVER_IP,


Constants.SERVER_PORT, 1000)

                self.cReader.addConnection(self.connection)

                taskMgr.add(self.readTask, 'readTask', -39)


                return True

        except:

            return False
```

This function is used to create a connection with the java server. The server ip and port is stored in the Constants.py file. After the connection has been created a readTask is added to the Python's task manger. The task manager calls this method once every frame. If the connection has been established successfully startConnection returns True, else it returns False.

```
def readTask(self, task):


        if (self.cReader.dataAvailable()):

            datagram = NetDatagram()

            if (self.cReader.getData(datagram)):
                data = PyDatagramIterator(datagram)

                responseCode = data.getUint16()

                if (responseCode != Constants.MSG_NONE):

                    self.handleResponse(responseCode, data)


        return task.cont
```

This task is used to check if there is any response sent by the server. The data is retrieved from the QueuedConnectionReader and stored in a NetDatagram. The contents of the package sent by the server are retrieved using the PyDatagramIterator class. The iterator class acts as the complement of the PyDatagram class; its methods can be used to retrieve the content that was encoded using PyDatagram. In the protocol section we have stated that the first data in the packet will be the Short that is unique to the response sent by the server. Hence, we extract this short and pass the data and the short extracted i.e. the response code to the handleResponse function.

```
def handleResponse(self, responseCode, data):


        response = ServerResponseTable.get(responseCode)

        if (response != None):

            response.set(self.main)

            response.execute(data)
```

The handleResponse function is called whenever a response is received by the client from the server. Each response code is unique to the response sent by the server. We know the code for the response before-hand (Check Constants.py for a list of response codes. Response codes have a prefix of SMSG i.e. Server Message). We also have a ResponseTable that associates the response code with its respective class. We get an instance of the associated class and pass the data to the execute function. The response class can now process the data anyway it wants to.

```
def sendRequest(self, requestCode, args = {}):


        request = ServerRequestTable.get(requestCode)

        if (request != None):

            request.set(self.cWriter, self.connection)

            request.send(args)
```

The sendRequest function is used to send a request to the server. This function is called from any other part of the game that requires a request to be sent to the server. The request code (Check Constants.py for a list of request codes. Request codes have a prefix of CMSG i.e. Client Message) and a key, value pair of the data that needs to be added to the packet needs to be passed to this function.   An instance of the appropriate request class is retrieved from the Request table depending on the requestCode. The connection object and the connection writer object is then set in the request class. Then the send function of the request class is called and the arguments specified are passed.

```
def closeConnection(self):


        if (self.connection != None):

            taskMgr.remove('heartbeatTask')

            taskMgr.remove('readTask')


            self.cManager.closeConnection(self.connection)

            self.connection = None
```

This function is used to close the connection with the server. The hearbeatTask is added to Python’s taskmanager when the user has been authenticated. When the user logs off, this task needs to be removed from the task manager. Hence, upon closing the connection it is removed from the task queue along with the readTask.

This is how the Game Client handles the sending and receiving of network packets. The data in the packets that are received by the client, will be in the same order they have been sent. Hence, if we have a packet that contains the following data {Short, String, Int, Float, Float}, we have to do a packet.getUint16(), packet.getString(), packet.getUint32(), packet.getFloat32(), packet.getFloat32(), to retrieve the last float.

ServerRequestTable/ServerResponseTable

These two .py files store the requests and responses that the client is allowed to send and receive respectively. They basically implement a key/value pair for each request/response.

In the constructor of each file, all the requests/responses are added to the hash map. As the request/response tables are similar, I will explain just the request table.

Example,

```
def init():

        ServerRequestTable.add(Constants.CMSG_SEND_BUDDIES, 'RequestBuddies')
        ServerRequestTable.add(Constants.CMSG_CHAT, 'RequestChat')

        ServerRequestTable.add(Constants.CMSG_REQ_DEAD, 'RequestDead')

        ServerRequestTable.add(Constants.CMSG_GLOBAL_CHAT,
      'RequestGlobalChat')

@staticmethod

    def add(constant, name):

        if (name in globals()):

            ServerRequestTable.requestTable[constant] = name

        else:

            print 'Add Request Error: No module named ' + str(name)
```

This is the add function used in the constructor to add the request to the hashtable.

```
@staticmethod

    def get(requestCode):

        serverRequest = None


        if (requestCode in ServerRequestTable.requestTable):

            serverRequest =
globals()[ServerRequestTable.requestTable[requestCode]]()

        else:

            print 'Bad Request Code: ' + str(requestCode)


        return serverRequest
```

The get function is used to create an instance of the request corresponding to the requestCode specified. It is used in the connection manager described before this section. An error is printed in the console. If the request exists in the table, the globals() function is used to call the constructor of the request. Thus, an instance of the request class is created and then returned to the connection manager.

Protocols

The protocols that are being used by the Game Client and the Server have been defined in the protocols section. Please go through the protocol section for a list of all the protocols. In this section, we will describe the working of just the "Login" protocol. The other protocols are handled in a similar manner. Once you read this explanation, you should be able to easily determine how the other protocols work.

The working of Login:

The login request is sent in the Login.py file. Login.py is located in the Login package under the main package i.e. src->main->Login->Login.py. The Login page takes in the username and the password entered by the user and sends it to the server.

```
def submit(self):


        # Process Login Form

        if (len(self.usernameEntry.get()) == 0):

            self.createErrorBox('Username Required')

            self.setFocus(0)

        elif (len(self.passwordEntry.get()) == 0):

            self.createErrorBox('Password Required')

            self.setFocus(1)

        elif (self.main.startConnection() is True):

            rContents = {'username': self.usernameEntry.get(),

                         'password': self.passwordEntry.get()}

            self.main.cManager.sendRequest(Constants.CMSG_AUTH, rContents)

        # End of Chunk
```

This is the submit function that's called after clicking the login button on the login screen. Some validations are done to check if the user has entered an empty value or not. If the user has entered some values in the username as well as the password field, a key/value pair i.e. rContents is created with "username" and "password" as keys. The values for these keys are taken from the text boxes in which the user has entered the values.

The sendRequest function of the connection manager is called by passing the RequestCode (in this case it is Constants.CMSG_AUTH) and by passing the data that is to be sent to the server.

```
def sendRequest(self, requestCode, args = {}):


        request = ServerRequestTable.get(requestCode)

        if (request != None):
            request.set(self.cWriter, self.connection)

            request.send(args)
```

The above function is the sendRequest function that was explained in the connection manager section above. Here in this example, requestCode corresponds to Constants.CMSG_AUTH and args corresponds to rContents.

The RequestLogin class is associated with the CMSG_AUTH request code. Hence, the ServerRequestTable class initializes the RequestLogin class and returns the instance to the connection manager.

The connection manager now sets the connection manager and the connection of the RequestLogin instance and then calls the send function of RequestLogin

```
def send(self, args):


        try:

            pkg = PyDatagram()

            pkg.addUint16(Constants.CMSG_AUTH)

            pkg.addString(args['username'])

            pkg.addString(args['password'])


            self.cWriter.send(pkg, self.connection)


            self.log('Sent [' + str(Constants.CMSG_AUTH) + '] Authentication
Request')

        except:

            self.log('Bad [' + str(Constants.CMSG_AUTH) + '] Authentication
Request')

            print_exc()
```

This function is the send function of RequestLogin. The data is extracted from the key/value pair and added to the packet accordingly. We first add the short that corresponds to the request being sent (in this case it is Constants.CMSG_AUTH). Then the username and password are added to the packet. The packet is then sent using the connection writer.

Now, the server will receive this packet, process it and respond with a ResponseAuth (we call it ResponseLogin for uniformity in naming conventions).

```
def readTask(self, task):
        if (self.cReader.dataAvailable()):

            datagram = NetDatagram()

            if (self.cReader.getData(datagram)):

                data = PyDatagramIterator(datagram)

                responseCode = data.getUint16()

                if (responseCode != Constants.MSG_NONE):

                    self.handleResponse(responseCode, data)


        return task.cont
```

The connection manager readTask detects that the server has sent data to the client. Using the connection reader, the data is extracted in the form of a NetDatagram. The NetDatagram is then used in a PyDatagramIterator (explained in the Connection Manger section above) for extraction of data.

As this function is used for all responses, we need to know what kind of response has been sent by the server. Hence, we extract the short that identifies the type of response from the packet and pass it to the handleResponse function.

```
def handleResponse(self, responseCode, data):


        response = ServerResponseTable.get(responseCode)

        if (response != None):

            response.set(self.main)

            response.execute(data)
```

The handleResponse function retrieves the appropriate instance of the response class from the response table. For now let us assume that the packet sent was the ResponseAuth response. Hence, the ServerResponseTable will return an instance of ResponseLogin. The main class is sent in ResponseLogin and the execute function of ResponseLogin is called.

```
def execute(self, data):


        try:

            if ('Login' in self.main.envMap):

                flag = data.getUint32()
                if (flag == 1):
                    self.main.charData = {'user_id'     : data.getUint32(),
                                          'username'    : data.getString(),
                                          'student_id'  : data.getUint32(),
                                          'firstName'   : data.getString(),
                                          'lastName'    : data.getString(),
                                          'gender'      : data.getUint16(),
                                          'email'       : data.getString(),
                                          'userState'   : data.getUint16(),
                                          'avatar_id'   : data.getUint16(),
                                          'health'      : data.getUint32(),
                                          'maxHealth'   : data.getUint32(),
                                          'moveSpeed'   : data.getFloat32(),
                                          'charLevel'   : data.getUint16(),
                                          'experience'  : data.getUint32(),
                                          'gold'        : data.getUint32(),
                                          'map_id'      : data.getUint16(),
                                          'lastX'       : data.getFloat32(),
                                          'lastY'       : data.getFloat32(),
                                          'lastZ'       : data.getFloat32()}

                    self.main.switchEnvironment('World')

                    taskMgr.doMethodLater(0.0, self.main.heartbeatTask,
'heartbeatTask')
                else:
                    self.main.envMap['Login'].showAlert()

            self.log('Received [' + str(Constants.SMSG_AUTH_RESPONSE) + ']
Authentication Response')
```

```
        except:

            self.log('Bad [' + str(Constants.SMSG_AUTH_RESPONSE) + ']
Authentication Response')

            print_exc()
```

Here the data is extracted for use in the game. As you can see, the packet contains a lot of data (user id, username, student id, firstname, lastname, etc.). All this data is extracted and then set as character data in the main class. The only time we get all this data is when the user has been successfully authenticated. Hence, we now switch to the Main scene. This is done by calling the switchEnvironment() function. Also, we can now start the heartbeatTask which basically tells the Java server that the client is still logged in.

If the user is not authenticated successfully, an alert box is shown to the user.

CHAT BUBBLE

One of the key reasons why MMO games are so successful is the reliability of the chat modules and functionalities. This game has two primary methods to display messages from users. The first of which is the chat box, which can keep track of several different types of chat from many different sources. The second method that the game displays messages between users is through the chat bubbles. The chat bubbles appear graphically in game, not requiring a user to divert their eyes towards the chat box and away from the main environment.

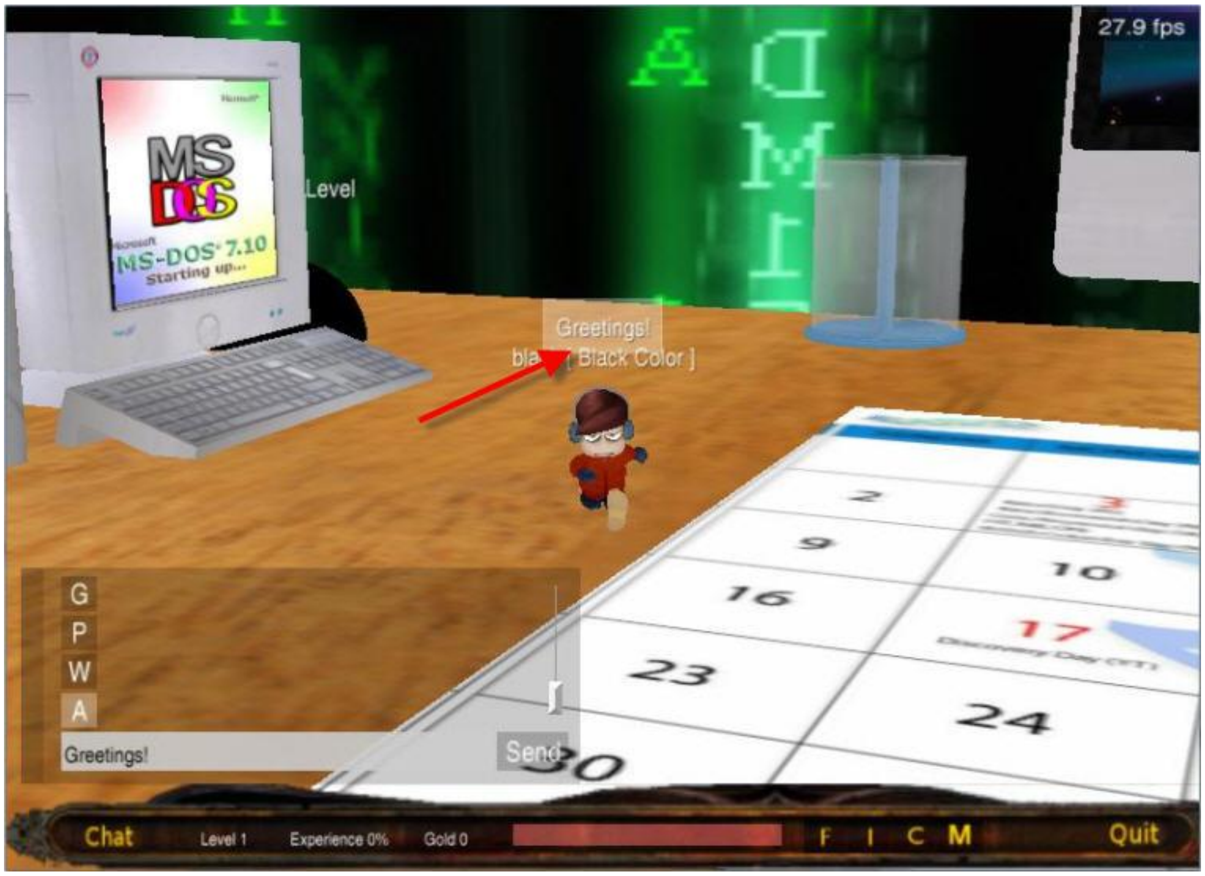


Figure 4.c.iv.1 - A player's message will appear over their head locally.

One of the characteristics about the chat bubble is that it displays over the source character's head. This allows players to easily find the source of the message without having to match names via the chat box. This prevents the pace of the game from being disrupted in the circumstance that there are critical events going on in the virtual world environment. Another characteristic about the chat bubble is that it displays messages from characters that are immediately visible to the player. This is different from the chat box, which has a wider range of access to messages from many characters.

In order to display messages in a chat bubble a user can simply input a message to send through the chat box. This will show the message in both the chat box and in the chat bubble. While the message is being created, the bubble will be filled with the text “Typing..”. This alerts other players that a character may be busy typing a message to notice something on screen or perform any immediate actions.

After a short amount of time or when a new message is being sent, the chat bubble and its contents will disappear from over the character.

Chat Bubble – Understanding The Code

ChatBubble.py is located within the main\World\Bubble package. This file is in charge of creating and displaying on-screen graphic text bubbles over character's avatar for players to see. These messages originate from those that a user sends through chat.

ChatBubble Class

This class creates bubbles and borders above a character's model. It then sets text inside the bubble that other players can see. After the message is displayed, the bubble is hidden.

Data Members

xObject	This object provides access to world variables for the current user.
chatBubbleSequence	Panda 3D Sequence Object. This provides a sequence of events for the bubble to perform.
lastDistance	This float variable holds the last distance.

lastPosition	This 3 point variable holds the last position.
textColor	This 4 point variable holds the color and alpha values for the text in the bubble.
defaultBoxColor	This 4 point variable holds the default box color and alpha values.
globalBoxColor	This 4 point variable holds the global box color and alpha values.
partyBoxColor	This 4 point variable holds the party box color and alpha values.
publicBoxColor	This 4 point variable holds the public box color and alpha values.
whisperBoxColor	This 4 point variable holds the whisper box color and alpha values.
defaultBorderColor	This 4 point variable holds the default border color and alpha values.
globalBorderColor	This 4 point variable holds the global border color and alpha values.
partyBorderColor	This 4 point variable holds the party border color and alpha values.
publicBorderColor	This 4 point variable holds the public border color and alpha values.
whisperBorderColor	This 4 point variable holds the whisper border color and alpha values.
cBubble	Panda 3D text node. This bubbles holds the appropriate text color, shadow.
cBubbleNodePath	This holds the path for cBubble.

Member Functions

__init__(self, xObject)

This function initializes xObject and the default values for all box, text and border colors. It then initializes cBubble and sets it to cBubbleNodePath. Afterwards, it adds cBubbleRoutine to tasMgr.

setText(self, type, msg, fade)

This function takes in a chat type (global, party, etc), a string msg and fade as arguments. This function checks the chat type being sent.

```
if (type == Constants.CMSG_GLOBAL_CHAT):
    self.cBubble.setCardColor(self.globalBoxColor)
    self.cBubble.setFrameColor(self.globalBorderColor)
elif (type == Constants.CMSG_PARTY_CHAT):
    self.cBubble.setCardColor(self.partyBoxColor)
    self.cBubble.setFrameColor(self.partyBorderColor)
elif (type == Constants.CMSG_PUBLIC_CHAT):
    self.cBubble.setCardColor(self.publicBoxColor)
    self.cBubble.setFrameColor(self.publicBorderColor)
elif (type == Constants.CMSG_PRIVATE_CHAT):
    self.cBubble.setCardColor(self.whisperBoxColor)
    self.cBubble.setFrameColor(self.whisperBorderColor)
else:
    self.cBubble.setCardColor(self.defaultBoxColor)
    self.cBubble.setFrameColor(self.defaultBorderColor)
```

It will assign cBubble the appropriate color based on the chat type. Following this, it sets cBubble margins. Afterwards, it creates cBubbleSequence if fade is true. Otherwise, it writes msg into cBubble.

```
if (fade is True):
    self.chatBubbleSequence = Sequence(Func(self.cBubble.setText, msg),
                                       Wait(5.0),
                                       Func(self.cBubble.setText, ''))
    self.chatBubbleSequence.start()
else:
    self.cBubble.setText(msg)
```

getText(self)

This function returns the text currently set to cBubble.

hideBubble(self)

This function checks if there is a current chatBubbleSequence. If there is, then the sequence is paused. At completion, this function resets cBubble's text.

cBubbleRoutine(self, task)

This function takes a task as an argument. It starts by getting new drawing positions if they are different from lastPosition. It then sets these positions to cBubbleNodePath and updates lastPosition to reflect the new values.

```
        self.cBubbleNodePath.setPos(self.xObject.getPos())
        self.cBubbleNodePath.setZ(self.xObject.getObject(), center.getZ() + radius * 1.3)

        self.lastPosition = self.xObject.getPos()
```

It then sets a new scale to cBubbleNodePath and updates lastDistance if these have changed.

unload(self)

This function removes the cBubbleRoutine from taskMgr. It then clears cBubbleNodePath. Afterwards, if there's a chatBubbleSequence the functions pauses the sequence.

BUG SERVER

Foreword: As previously stated, the bug server was developed as an extension of the client, with various modifications to make it possible to provide some autonomous functions to bugs. As such, many of the files that were included in the bug server is a direct port from the client. In many of these files, the only changes were the commenting or removal of the logic for that specific protocol, hereby called event. Certain events such as adding a bug to a players buddy list was removed because there would be no point in doing so. For such changes as these, we will be aggregating those files into one section so we may concentrate more heavily on the crucial parts of the bug server.

Overview: The bug server is made up of primarily two parts. The logic and the events that are carried out once the logic has satisfied the various conditions. Because of the simplistic design, and the fact that we have used the client architecture in the implementation of the bug server, there is only two areas in the code that needed to be modified. One exception would be the disabling of the GUI, which is turned off because the bug server does not control a bug directly. However, it is left to the implementers to turn it on in case it is needed for debugging purposes. Below we have included a list of the files that make up the bug server at its last revision. Again, as noted, this looks very close to the work that is on the client. The most important change would be the addition of Bug inside of the World directory. This is where the logic for the bugs are held. Comparatively, the logic for players would be in Character. We shall be going through the code by file in order of typical execution. We are writing this understanding from a logical point of view so that you the reader is able to understand where the code is going, and thus in order to add the relevant code, you need to know the relevant place in the code to modify. We will briefly give an introduction to the files mentioned: Constants.py – The location for various constants, needed to debug at times. Main.py – The place where the disabling of the GUI is done. World.py – Where bugs shall be spawned. This should be done once per environment. BugGenerator.py – Where bug spawning actually takes place. Bug.py – Where most logic takes place. [Various Events] – They will be listed out as they are encountered. [Other non functional files] – Just files that may be modified to add extensibility.
[Note: All Events have a Response and Request. They will simply be talked about in terms of responses. Requests events are all handled within the logic, the Requests only package up the information]

Constants.py – [Major contributor: Client Team] The only thing to really note in this file is the constants that matter most: IP and Port. SERVER_IP = 'thecity.sfsu.edu' SERVER_PORT = 9996 These two lines give the possibility of running the bug server locally as to test out your changes (if you have a server running locally, and a client running locally) **Main.py – [Major contributor: Gary. Minor: Alan]** When the client first runs the game, they have to enter their login in order to authenticate themselves. In regards to the bug server, we have used the login green, and password green. So in this case, the bug server starts the connection immediately: if (self.cManager.startConnection()): print 'Connection Successful!' rContents = {'username': 'green', 'password': 'green'} self.cManager.sendRequest(Constants.CMSG_AUTH, rContents) else: return task.again The last line is very important; it will attempt reconnecting to the server until it is able to connect. There is no point trying to log in otherwise. **ResponseLogin.py – [Major Contributor: Client Team]** As per reading the documentation, the word heartbeat gets used a lot, and here is where the work for the heartbeat is started. self.main.switchEnvironment('World') taskMgr.doMethodLater(0.0, self.main.heartbeatTask, 'heartbeatTask') Since the bug server is a single map only server, we have to switch the environment so that we can load the bugs onto the server. The heartbeat is also started as a task so we can begin getting requests from the server. **World.py – [Major Contributor: Gary. Minor: Alan]** Once the environment is loaded, the next step is to load the entities (Bugs) onto the map. This is done through World.py. self.map_id = 3 self.bugGenerator = BugGenerator(self) self.remoteCharacterGenerator = RemoteCharacterGenerator(self) if (self.charData['map_id'] != self.map_id): rContents = {'map_id': self.map_id} self.main.cManager.sendRequest(Constants.CMSG_SEND_SCENEID, rContents) self.switchMap(self.map_id) We make sure that green is still on the environment that it has to load the bugs. Why? Because the server needs to know which scene the bugs are in to properly send the bug information to players that go onto the proper scene. Otherwise you will see bugs on scenes that aren't the one they should be on. In this case, scene 3. This initialization function also calls the Bug Generator. This is where the bugs actually spawn on the maps. The function load entities does most of this work: def **loadEntities**(self): # Load Other Entities (i.e. Bugs, NPCs, etc.) if (self.map_id == 2): bugData = {'model_id' : 2, 'scale' : 0.75, 'move_speed' : 0.75, 'count' : 10, 'mode' : True, 'boss' : True, 'pos_x' : -101.783, 'pos_y' : 10.8503, 'pos_z' : -0.315859, 'radius' : 5, 'money' : 100} self.bugGenerator.generate(bugData) Through this, you can see the various options that can be set. Furthermore, to extend it, you can have these attributes loaded from a file, versus set inside of World.py. BugGenerator.py – [Major Contributor: Gary. Minor: Alan] The bug generator simply has a few functions that are used to get and remove bugs, as well as to generate bugs that were called in World.py. This is also where the bug ID gets generated. Without the bug ID, or with clashing IDs, you will have bugs jumping around the map due to two bus using the same ID, so this piece of code prevents that: self.bugMap[self.object_id] = Bug(self.world, args) self.object_id += 1 Once that is done, bugs will soon be spawned onto the map. **Bug.py – [Major Contributor: Gary, Alan]** We will be listing out the code by the function name, simply because it will be easier to do due to the fact that all logic is done in this file.

__init__: The initialization function will set the attributes of the bug based on the list that the bug received from the corresponding callers (BugGenerator and World). self.model_id = args['model_id'] self.moveSpeed = args['move_speed'] self.object_id = args['object_id'] self.position = Point3(args['pos_x'], args['pos_y'], args['pos_z']) There is not much to explain because many of these attributes are similar to players. The biggest part of the code comes at the bottom: rContents = {'bug_id' : self.object_id, 'type' : self.name, 'level' : self.level, 'scale' : self.scale, 'model_id' : self.model_id, 'map_id' : self.world.map_id, 'x' : self.actorObject.getX(), 'y' : self.actorObject.getY(), 'z' : self.actorObject.getZ(), 'h' : self.actorObject.getH(), 'isBoss' : self.isBoss, 'maxHealth' : self.maxHealthPoints, 'healthPoints' : self.healthPoints, 'money' : self.money} self.world.main.cManager.sendRequest(Constants.CMSG_REQ_REGISTER_BUG, rContents) # Initiate Routine Task taskMgr.add(self.bugRoutine, 'bugRoutine-' + str(self.object_id)) Again, we are going to be calling an event to register the bug, and with that, also running a bug routine that shall start. This routine allows a bug to move. Search for the event RequestRegisterBug to learn about its workings. **bugAttack:** Collisions as well as attacking is done server sided. Therefore, the logic required to be attacked by a bug comes in when the routine is called. Reading through the code, you will see various logical checks to see that a bug is able to attack, such as this one: if (eTarget != None and not eTarget.isDead): rContents = {'attacker_id' : self.object_id, 'target_id' : user_id, 'damage' : atkDamage, 'type' : type} self.world.main.cManager.sendRequest(Constants.CMSG_REQ_ATTACK, rContents)

There is a packet sent when a bug is able to attack the player. And this is how a player gets damaged in game. **bugChase:** Bug chase is called before a bug is able to attack a player. They have to run towards the player so that they are close enough to attack. Like shown above, it’s only when a bug is in range can it attack: destDistance = (self.destPoint - self.actorObject.getPos()).length() if (destDistance > self.atkRange): self.inAtkRange = False else: self.inAtkRange = True Make note of the math done in order to determine distance. This shall be used later on to initiate the attacks. **bugEngage:** This function simply does a logical check on the bugs status, and if it’s currently attacking a player, and the bug isn’t dead, set its flag to be in battle. if (objectTarget != None and not self.isDead): if (not objectTarget.isDead and self.canAttack and self.inAtkRange): if (not self.inBattle): self.inBattle = True **bugCollide:** bugCollide simply moves the bug on top of terrain, so that it does not fall through the floor. self.actorObject.setZ(collisionEntry.getSurfacePoint(render).getZ()) Also checks to see if they can move after reaching a destination: else: self.bugStop() self.canRoam = True self.actorObject.setPos(self.position) **bugMove:** The function was copied from client, and changed to show the animations of the bugs moving. This is only syntactic. **bugRoam:**

bugRoam generates a new point for the bug to move to. It uses the Pythagorean theorem to move a bug, so that it can be a bit more efficient. if (self.canMove): xPos = self.spawnPos.getX() yPos = self.spawnPos.getY() newDistance = uniform(self.minRoamRadius, self.maxRoamRadius) randX = uniform(-newDistance, newDistance) randY = sqrt(pow(newDistance, 2) - pow(randX, 2)) rSign = choice([-1, 1]) newDestPoint = Point3(xPos + randX, yPos + rSign * randY, self.position.getZ()) self.setDestPoint(newDestPoint) randX and randY gives it the heading, due to the knowledge of the hypotenuse, and randX, we can solve for randY **bugStop:** bugStop is again, taken from the client. Simply stops the bug. **dropTargetTask:** With the in game logic, when a bug sees that a player is too far, this function is called in order to send a request to the server that a bug has stopped attacking: def **dropTarget**(): if (self.objectTarget != None): rContents = {'attacker_id' : self.object_id, 'target_id' : self.objectTarget.getID()} self.world.main.cManager.sendRequest(Constants.CMSG_REQ_BATTLE_REMOVE, rContents) **getClosestTarget:** getClosestTarget tries to obtain, and set the destination of a bug's future move. This is done to reach the destination where the player is headed, not going to where the player currently is. if (self.actorObject != None): targetDistance = (targetPos - self.actorObject.getPos()).length() targetPos.setY(targetPos.getY()+sqrt(abs(targetDistance))) We want the targetPosition to be a bit of distance in front of the player. This will make us get closer to the player because they themselves, are getting closer to us. And once the player is within range, they will be attacked through an event raising: if (minDistance <= self.sightRadius): self.setCanSeekDelay()

rContents = {'attacker_id' : self.object_id, 'target_id' : closestTarget.getID()} self.world.main.cManager.sendRequest(Constants.CMSG_REQ_BATTLE_ADD, rContents) **setTarget:** We are simply setting the target of the bug to a specified object. **regenQuestionTask:** This function will delay the quiz "attack" of the player. This prevents a bug from simply letting a player infinite chances with no wait. self.regenQuestionSequence = Sequence(Wait(self.regenSpeed), Func(regenQuestion)) **respawn:** This function calls the initialization function, which will restore the bug as if it just loaded. **setCanAttackDelay:** This function delays the bug attacking a player, so that players do not die instantly. **setCanSeekDelay:** This function again delays a bugs seeking, so that they do not repeatedly seek. **setPassiveMode:** This function prevents aggroing. It delays the seeking of the target. **setDestPoint:** This function sets the bugs' destination point. Used by the targeting function and other functions such as roam. **setRoamTask:** This function basically restarts the roam sequence when the bug reaches its destination. It's so bugs stand still for a bit before moving again. **takeDamage:** This function simply checks a bug for its status such as health, and if they are in a dead state, set the various flags to support such. Here the bug will check to see that if it is dead, it will respawn after a certain amount of time:

else: taskMgr.doMethodLater(self.respawnTime, self.respawn, 'respawnBug-' + str(self.object_id)) self.unload() **bugRoutine:** This function here mainly deals with every other function in the Bug file. From the bugRoutine, the bugs are able to act autonomously. It sets the various flags and runs the various functions that will tell a bug what to do next: if (not self.isDead): if (self.objectTarget != None and not self.objectTarget.isDead): self.bugChase() self.bugEngage() self.bugAttack() elif (self.inBattle): self.setPassiveMode() self.bugRoam() self.bugMove() Here you can add your own logical tests so that they can do other things. **unload:** Simply removes all of the tasks set by the bug. This is so that no stray sequences such as roaming still run while the bug is in the unloaded state. Just checks to see that each sequence that is possible for a bug is properly paused. **updatePosition:** updatePosition is called while the bug is in motion. This is done in bugRoutine. Bugs that are in motion send events off to the server so that the server can update other players on the bugs new position: if (currentPos != self.lastPos or currentH != self.lastH or self.wasMoving): rContents = {'object_id': self.object_id, 'x' : currentPos.getX(), 'y' : currentPos.getY(), 'z' : currentPos.getZ(), 'h' : currentH, 'speed' : self.moveSpeed, 'isMoving' : self.isMoving} self.world.main.cManager.sendRequest(Constants.CMSG_MOVE, rContents) **ResponseMap.py – [Major Contributor: Client Team]** This simply sets the scene ID of the entire bug server. This is why a bug server only resides on a single scene, because scenes are based on connection, not per object/player ID.

pkg = PyDatagram() pkg.addUint16(Constants.CMSG_SEND_SCENEID)
pkg.addUint32(args['map_id'])

In order to extend the game to allow more than one map per server instance, this has to change to track the ID of the map changing player/bug. **ResponseAttack.py – [Major Contributor: Gary]** Once a player answers a quiz question correctly, a bug takes damage. This is how a bug knows that it will die. bObject = self.main.envMap['World'].bugGenerator.getBug(target_id) if (bObject != None): bObject.takeDamage(eTarget, damage) **ResponseBattleAdd.py – [Major Contributor: Gary]** Once a player attacks, or gets attacked and is able to add itself to the characters list of bugs attacking, this is sent back from the client telling the bug to add itself. But it uses the closest target. self.main.envMap['World'].bugGenerator.getBug(target_id).getClosestTarget() **ResponseBattleAcc.py – [Major Contributor: Gary]** Part two of the battle sequence, this comes from a player and directly tells a bug that it has been successful in its attempt at attacking. Once this occurs, the bugs target is now the target that it wanted to attack. self.main.envMap['World'].bugGenerator.getBug(attacker_id).setTarget(target) **ResponseBattleRemove.py – [Major Contributor: Gary]** This event can either be called by a player, or by a bug. If it's called by a bug, the logic would be in Bug.py. This request is sent by the player to tell the bug that it is no longer being attacked. bObject = self.main.envMap['World'].bugGenerator.getBug(attacker_id) if (bObject != None): if (bObject.objectTarget != None and bObject.objectTarget.getID() == target_id): bObject.setPassiveMode() **RequestRegisterBug.py – [Major Contributor: Gary. Minor: Alan]** RequestRegisterBug is called by Bug.py when it first spawns in a map. It is also called after respawning. This event is a modification of the player RequestRegister event. Most importantly, the bug then begins to send its position when it's moving using a task: taskMgr.doMethodLater(1.0, bugObject.updatePosition, *'updatePosition-'* + str(bug_id))

Notes: The other various events that are included in the bug are simply there to complete the protocols. Otherwise, there will be an error thrown. It does not mean that future implementers cannot use those protocols to add functionality into the game.

Conclusions: The bug server heavily relied on the porting of the client code, and this allowed it to easily mimic human movement by simply re using the code. Much of the main work is done in Bug.py. The events that are called are in the same layout as the client, and if there are features that must be ported over such as running, it would be trivially easy to modify and move over to the bug server.

GAMESERVER

The Nursetown team did a lot of the code and the following paragraphs are from Zoran Milic who helped developed the Nursetown server as lead developer. At the time of the semester ending, Fall 2009, the DeBugger team wasn't able to implement a second part of our game, which was to have a board game but Nursetown was able too and it's still possible too and can be easily added in the future. I left these parts in Zoran master report in case future developers need some kind of information about it. In each section I added a * to show there was a changed from Zoran master report. At the end of each section if there was a change from the Nursetown server to the DeBugger I wrote our own edition to the server to explain some of the changes.

5.1 Zoran Master Report

Game Server application was designed with the idea of being the key cohesive element of the Serious Game. It was supposed to support all the functions of the Game Client: registration of new users, authentication of the users connected through the Game Client, managing of the users, relaying of the information between them, enabling users to participate in the Board Game (Quiz), and maintaining the persistence of the "Nursetown" Game World. Game Server was also designed as a way for Game Clients to record vital information of the user's participation in the game, such as points users accumulate during the game or their health status as it changes over time. It was also planned, as a part of some future development of the game, to enable recording of information regarding game users, their interaction with other users, and their participation in the educational elements of the Game (for instance, Game Quiz). This information was supposed to be used for the analysis of the impact that a game like this would have on the educational process, once it was introduced.

In an effort to improve the stability and reliability of the existing code, I did a review of the existing protocols, cleaned some of the logical and functional errors. I also modified the code so that thrown Exceptions would be caught and properly handled. Uncaught exceptions were the source of many errors reported on the Server side. They were also a source of instability and unreliability, since each unhandled exception meant the unexpected termination of the corresponding process, and consequentially loss of data and discontinuation of the player's game session. Furthermore this laid the foundation for future programming policy regarding handling potentially erroneous portions of the code (DB insertions, etc.)

One of the changes made before I started working on the Game Server was introduction of Server-side data containers. Idea behind this modification was to reduce the number of accesses to database by keeping some of the data in the memory, so that it can be accessed more quickly. Also, because of the way these data are organized, it was possible to access them in a more convenient way then if it was done by directly querying database. Initial version of these Server-side data containers was done by one of the contributing students, Genki Kuroda, but it had design problem. One of the data containers was superfluous in a way that it contained information that was already part of another collection, and hence was unnecessary. In order to remove this design flaw, I performed several modifications to the "NurseGameServer" class.

5.2 Design and performance improvements

Another change made by the same student who added server-side data containers was introduction of GamePacket class. Its purpose was to encapsulate big-endian conversion when data is being sent from Game Server (written in Java, big-endian) to Game Client (written in Python, little-endian). Before this class was added, conversion was done inside each of the Request/Response classes individually. This addition automated procedure and made sure that no error could be introduced in the process by omission.

5.3 Logout and auto-logout procedures*

In cooperation with one of the exchange students who worked on this project, Abra Viagbo, I perfected the currently existing logout sequence and implemented new auto-logout sequence for the server-side thread that was handling the Game Client application.

I completed a Logout sequence on the Server side. Before this change was made, when Client logged out Game Server would terminate the thread that was servicing that Client, and would inform other Clients of that. My change was to update the user's information in the Server-side database, more precisely to remove scene id from the scene_id column in "User" table. This addition helps in maintaining the accurate status of the database during the game, which is important in case that server tries to use the database to make any inquire regarding user's status. It is also related to some other changes that were subsequently made to the Game Server.

This, however, did not completely solve the issue. Here is the basic problem Game Server had in operating GameClient class threads: the thread was based on an infinite loop that would read the incoming package sent by the client; if size of the package was equal to 0 (zero), the rest of the loop would be skipped, and thread would go back to reading the incoming package. This classic polling mechanism works fine as long as there are incoming Client's packages. First problem, however, came from the fact that Game Client was not sending proper "logout signal" when player would decide to exit the game. This would cause the Game Server to enter an infinite polling loop, which would effectively tie up huge CPU resources on server side. On the other hand, other players in the game would not be aware of the fact that one of them has left the game. All they would see is that the player has become unresponsive, standing still. In effect, they were not notified that player has left the virtual world, and that his avatar should be removed from the virtual world.

Solution for this problem was found in cooperation with the developers working on Game Client. Client's code was modified so that it sends proper logout request. The fact that player has left the virtual world would then be broadcasted to other players, and they would know that they need to remove his avatar from their virtual world.

The code that is handling this procedure on the Game Server's side was done, in part, by Abra Viagbo.

This, however, was not the end of the problem. If Game Client would, for any reason, discontinue his functioning without sending proper logout signal (unexpected termination or loss of network connectivity, for instance), the thread that was servicing that Client would, again, enter the infinite polling loop.

This kind of situation would have to be done entirely on the Server side. The solution that I implemented would have a two-step approach. But before describing that approach, I have to explain in brief the way Client communicates with the Server.

Game Client sends on regular intervals "Heart Beat" signal, indicating on one hand that he is still present and functioning, but on the other hand requesting to be updated. These updates contain all information that Server designated as information that is to be "broadcasted" to the Client. Updates are sent as a response to a Heart Beat request. These Heart Beat requests are sent on average each 15 milliseconds. This regularity of update requests is what I used as a basis for my improvement.

First step of my approach would be a temporary suspension: if a Client would stop sending data packages to the Server, Server's thread would put itself to sleep for a certain period of time (right now it is set to 20 milliseconds). After said time, thread would again try to read incoming data. If there was no communication from the Client, thread would enter sleeping cycle again. Also, each time a thread enters sleep cycle, a designated counter is incremented.

If thread goes through specified number of consecutive sleep cycles (in our case 3000 cycles, which is equal to 1 minute), the only conclusion that can be drawn is that Client became unresponsive. After determining that, thread can move to the next step, forcing the Client to logout and terminate itself.

If, however, the thread resumes receiving the data from the Client, sleep-cycle counter is re-initialized, and work resumes like before the temporary suspension.

This modification solves the problem by putting the Client "on hold", so to speak, for a short period of time, re-examining its responsiveness until it can be determined with certainty if the Client is or isn't working. This modification, also, saves the server-side CPU resources by making considerably less number of unsuccessful polling cycles. Finally, if Client is really terminated, thread terminates itself also, only properly logging out the Client before that.

In the new game DeBugger logout and login, to keep the world persistence, we added new fields to the User table. Some of these things were the last x,y,z coordinates, money, experience, equipment, a user_state. In Nursetown people would logout and end up at the same area where they had first started. We changed this in DeBugger. With the last x,y,z coordinates and logout scenen id. If someone logged out of map 2 in the left corner of the map, that where they we end up when they logged back in.

5.4 CONNECTION POOL

Next very important modification was introduction of the Connection Pool. Game Server, being an important part of the "Nursetown" Serious Game, for most of its functions depends on reliable connection to the database. At its initial versions, while it was still operating as a test application, Game Server was functioning with only one database connection, one that would be set up when server would start up. But, the server often has need for multiple connections at the same time. Creating new database connection can be time consuming. Keeping the connections valid, testing them, and refreshing them when needed can be a complicated task. Having a dedicated application that does the function of the connection pool became a necessity.

There are several open source solutions, packages and libraries that can be used for this purpose. Having personally good experience with the open source software developed by Apache Software Foundation, I opted for their Jakarta DataBase Connection Pool (DBCP).

In order to implement Jakarta DB Connection Pool, I had to make use of two of their libraries, both developed under "Apache Commons" project. The libraries are Commons Pool and Commons DBCP. Commons Pool is a component that provides generic pool interface, a toolkit for creating modular object pools, and several general purpose pool implementations. [31] Object being pooled in this case are JDBC resources. Commons DBCP relies on the Commons Pool package to provide underlying object pool mechanism that it utilizes.

Sequence of steps to set up the Connection Pool goes like this:

- First we need an ObjectPool that serves as the actual pool of connections. We use a GenericObjectPool instance, although any ObjectPool implementation will suffice. Code snippet:

```
ObjectPool connectionPool = new GenericObjectPool( null,
GenericObjectPool.DEFAULT_MAX_ACTIVE,
GenericObjectPool.DEFAULT_WHEN_EXHAUSTED_ACTION,
GenericObjectPool.DEFAULT_MAX_WAIT, GenericObjectPool.DEFAULT_MAX_IDLE,
true, false, 3600000, GenericObjectPool.DEFAULT_MAX_ACTIVE,
GenericObjectPool.DEFAULT_MIN_EVICTABLE_IDLE_TIME_MILLIS, true);
```

Parameters of the pool's constructor are:

  - PoolableObjectFactory – Factory used to create, validate and destroy objects. If null is given as a parameter, it will be set to default.
  - maxActive – Maximum number of objects that can be borrowed from the pool at one time. Default is 8, which is what I used. To have a limitless pool, this parameter should be non-positive number.
  - whenExhaustedAction – Specifies the behavior of the borrowObject() method when the pool is exhausted. Default is BLOCK.
  - maxWait – How long (in milliseconds) will the pool block the request for new connection if pool is exhausted. If non-positive value given, blocks indefinitely. Default value -1.

    - maxIdle – The maximum number of idle objects in the pool. Default is 8.
    - testOnBorrow – Will or will not the pool attempt to validate each object before it is returned from the borrowObject() method.
    - testOnReturn – Will or will not the pool attempt to validate each object before it is returned to the pool in the returnObject(java.lang.Object) method.
    - timeBetweenEvictionRunsMillis – The amount of time (in milliseconds) to sleep between examining idle objects for eviction. Default is -1, which stands for "not performing any evictions". I've set it to 3,600,000 milliseconds (60 minutes).
    - numTestsPerEvictionRun – The number of idle objects to examine per run within the idle object eviction thread (if any). Default is 3. I set the parameter to a bit higher number, but again not too high, default value for "maxActive" parameter.
    - minEvictableIdleTimeMillis – The minimum number of milliseconds an object can sit idle in the pool before it is eligible for eviction. Default is 1,800,000 milliseconds (30 minutes).
    - testWhileIdle – Whether or not to validate objects in the idle object eviction thread, if any.
- Next is to create a ConnectionFactory that the pool will use to create Connections. We'll use the DriverManagerConnectionFactory, using the connect string passed by GameDB. Code snippet:

```
DriverManagerConnectionFactory
connectionFactory = new
DriverManagerConnectionFactory(connectURI, null);
```

Second parameter is a list of string tag/value pairs that serve as connection arguments. In this case null instead of it.

- Now we'll cr3eate the PoolableConnectionFactory, which wraps the "real" Connections created by the ConnectionFactory with the classes that implement the pooling functionality. Code snippet:

```
PoolableConnectionFactory
poolableConnectionFactory = new PoolableConnectionFactory
(connectionFactory, connectionPool, null,
"SELECT * FROM user LIMIT 0, 1", false, true);
```

Parameters are:

    - connFactory – The ConnectionFactory from which to obtain base Connections
    - pool – The ObjectPool in which to pool those Connections
    - stmtPoolFactory – The KeyedObjectPoolFactory to use to create KeyedObjectPools for pooling PreparedStatements, or null (as in my case) to disable PreparedStatemments.
    - validationQuery – A query to use to validate Connections. Should return at least one row. Using null turns off validation.
    - defaultReadOnly – The default "read only" setting for borrowed Connection.
    - defaultAutoCommit – The default "auto commit" setting for returned Connections.
- Finally, we create the PoolingDriver itself, passing in the object pool we created. Code snippet:

```
PoolingDataSource dataSource = new PoolingDataSource(connectionPool);
```

Data Source that was created is returned to the calling class, DataDB, and can be used for fetching of JDBC objects, when needed. Code snippet:

```
connection = dataSource.getConnection();
```

After the interaction with the database is done, to return the connection back to Connection Pool it is enough simply to "close" it.

5.6 IMPROVING GAME CLIENT'S USABILITY*

I reviewed and completed implementation of the "New Account" feature. Even though "New Account" button was present at the Game Client's main page, it was not functional. New players could not be created, since the mechanism on the Server's side that should handle that, RequestRegister class, was not finished and tested. In order to get it to work, I first had to modify structure of the database table "User", to allow columns to have default values (all columns except 'user_id', 'username', and 'password'). This allowed creation of new users knowing only their 'username' and 'password'. To be sure that players have distinct usernames, prior to creating new user entry in the database check is performed to make sure that given username does not already exist. Next thing was to create a ResponseRegister class that would handle sending the message back to the client, informing him if creation of new user was successful or not. This required addition of new Response ID number in the Constants class on Server and Client side. Only thing that remains to be done is to implement reception of this response on the Game Client's side, and to present a proper interpretation of that response to the player.

I added the modification that prevents user from logging in the Game World twice at the same time. Primary intention of this correction was to eliminate the possibility of accidental duplicated login. Such situation could happen if, when logging in, user clicks twice very fast on "Play" button. Later, however, after appropriate modification was made to the Game Client to remove this possibility, there still was a possibility of double login. For instance, if two different Game Client applications would to try to login using same user name, they would have succeeded. Needless to say, having two identical characters in the game at the same time can cause confusion. Modification I made prevents that from happening. Solution I implemented is simple: during the authentication procedure, Server will check the value in scene_id column of "User" table; since this value is being set to NULL every time player logs out, it should be the same when he tries to log back in again; so, if the value is some integer number, that would indicate that user with that username is already logged in.

The DeBugger server was able to get this functioning with the game client. Now when the player wants to create a new character they will be able to from the game client. Things that were added in the registration are

- User Name
- Password
- First Name
- Last Name
- Avatar Model
- Email

We finished up the registration and were able to implement this so now it will be easier for players to make new character if they want too.

5.7 CONFIGURABILITY*

I introduced a configuration file for the Game Server. Initial purpose of this file was to relocate some of the hard-coded parts of the code outside the Server, more precisely database connection settings and listening port number. But, the purpose of this file, in some future development of the Server, could be to hold all the data that configures the Server and its function.

To introduce this configuration file, I created a dedicated class, "NurseGameServerConfig.java". When Game Server is started, during its initialization, this class reads the configuration file, "game_server_config.txt", and extracts all the information. Later, during the Server's operation, this class returns the information as requested by other classes.

Nothing really changed to the configuration file, except we added another part to NurseGameServerConfig.java. Now it will also look for BugServerPath. What this is the BugServers absolute path. This was needed because first we don't know where the person is storing the bug server folder and it's launcher.py. I didn't want to assume they would store everything in the same place. Second is the bug server will not launch correct if it's not launched from within it's own directory. This took a long time to figure out, but lucky there was a class in java that would let a process execute from within a certain directory.

5.8 PERSISTENCE*

Some modifications to the Game Server were made to enable recording of information regarding game user's interaction with other users and their participation in the educational elements of the Game (for instance, Game Quiz), all for further analysis.

I modified Game Server to record information about user's participation in the quiz. Some of the information were directly stored in the database, others could be found by cross-referencing information in two or more tables. Data that was recorded were: quiz player participated in, list of all players that participated in the quiz, name of the winner, time when quiz was scheduled, board-map that quiz was played on, quiz game duration, player's movement across the game board, questions player answered, correctness of player's answer. All the changes required to record these data were made to GameDB.java class, which is main database class, and to number of "RequestQuiz", "RequestQuest", and "RequestAnswer" classes.

Number of modifications to the Game Server was made having in mind future development of the game. One important addition that was planned was expansion of Game World's persistence (persistence would be Game World's trait to have data that would outlive the execution of the program that created that data [32]).

One of the modifications was to make available list of players that are in a particular scene. Purpose of this was to enable players to "land" anywhere in the Game World and be immediately aware of all other players that are there with them.

Another modification, in relation to the first one, was to record the scene_id number for the scene player logged out from. Idea behind this was to enable players to login to the same scene they left from last time they were playing the game.

Also, I programmed Game Server to record logout time, each time player leaves the game. This piece of data is not used at the moment. One possible use in the future can be so that Server could update player/Client with global, Game World's events that had happened since the last time player was in the game.

In DeBugger to keep with persistances new fields were added, and changes to the user login and logout. They will now login where they last logged out and also be on the game map.

5.9 SUPPORT FOR ADDITIONAL CLIENT'S FEATURES

Furthermore, I added a set of protocols for the "Buddy list". Buddy list is a register of player's friends in the "Nursetown" Serious Game. Players should be able to see the list as part of the tool-bar, at the bottom of the Game-window. From there, player should be able to add new friends, remove them from the list, or select those he wants to send a message to.

Protocols I implemented for Buddy list are for adding a new buddy, removing of the existing buddy, and for returning the list of user's buddies upon request. All of the protocols are fully functioning, but are still not used on the client's side.

Another set of protocols that was implemented on the Server side only are protocols that enable Client's to record information regarding player's level of health and points/money.

Also, I implemented protocol for broadcasting of the player's avatar animation ID. More precisely, if and when player's avatar makes a special preprogrammed gesture, animation, this should initiate a RequestAnimate signal sent to the Game Server, sending animation ID. Server will, in return, broadcast this ID to all other players. Knowing what is the model of the player who makes the animated move, and having the ID of the animation, all other player's Clients will be able to show the identical animation of the originating player's avatar to their players. This, naturally, required addition of new Animation Request and Response ID number in the Constants class on Server and Client side.

One of the plans that made for the Game Client was adding map of the Game world to the Game Client's window. Idea was to enable player to see whereabouts of other players in the game, and not only those that are in "visual" vicinity. To enable that, I created new class, RequestChangeScene, and corresponding ResponseChangeScene class. Whenever Client would change scene, he would inform Server of that. These classes made sure this change was relayed to all other Game Clients, so that they could update the map, once that map is implemented.

Finally, we have added simple web interface for control of the Game Server's starting up and shutting down, a page through which Server can be started and stopped. In case Server is to be stopped, it can be done within 60 seconds or 10 minutes. That gives Server enough time to save user data and properly logout players, after notifying them about the shutdown. Once proper notification system is implemented on the Client's side as well, as it is already on the Server's side, this feature will still needs to be fully tested.

Other things that were added in DeBugger a global chat. People can now talk to each other over different maps instead of just talking locally. Movement speed so that players can now run at a different speed depending on the equipment they get.

5.10 Bugs

This is a new feature added in DeBugger. New protocols were set up for both the client and the game server. Also a modified game client was used to control the bugs. The game server job in this was to find a way to store these bugs in the game server but also let the players interacted with them since they are attacking person with quiz questions. If a player defeats a bug the bug might drop an item and gold. In order for a bug to be created a request and response was created. RequestRegisterBug and ReponseCreateBug are the two protocols that create a bug on the server. The bug server sends a RequestReigsterBug, this protocol basically creates a game user object with some minor differences then a player game user object. It's got an aggressive flag, and a dead flag. Also a scale flag so that it can be in the game the bug can be big or small. This request will put this bug game user object in a list of active bugs for the server to keep track of. It's put in a different list then the active game users who are the real players because players are having their queues being updated consistently. Bugs don't really need their queues to be updated; they don't care about things like who is sending messages for help then a player getting beat by a bug. Once a bug is registered a response is sent back to the bug server and it will control the bugs action from there.

This game won't work if there are no bugs to kill which one the game server needs to launch the bug server as soon as it starts. This is done from within the NurseGameServer.java with the help a class called BugServerProcess.java. BugServerProcess.java is a runnable which is basically a thread. In the NurseGameServer code as soon as the NurseGameServer is created it will run a thread process. From here the BugServerProcess class takes over. The BugServerProcess class does a couple of things. It gets the path of the BugServer folder. Then it checks to see if the bug server Launcher.py is at the path given. If not it will print out the location of where it couldn't find the Launcher.py. If it does it will start a process using a java library called ProcessBuilder. ProcessBuilder was used it let the command that needs to be executed run from within a certain directory. This is a major thing, as the BugServer will not launch correctly if it's run from whatever other then within it's own directory. What I mean by this is, if it's one directory above the BugServer folder and we want to execute the Launcher.py. If the command python BugServer/Launcher.py is executed the BugServer will fail. Also the command to run the BugServer is not just python but "python –u" this makes the outstream,errorstream,inputstream unbuffered. I don't know why they need to be unbuffered but if using the ProcessBuilder class with out the command "-u" will make the process hang and the bug server will not function correctly.

FILE STRUCTURE

***BugServer* –** The bug server code manages bugs for the user to interact with. This code is primarily written in python and was implemented in conjunction with the client code, but on smaller scale with some different functionality.

BugServer/ **-** Root directory. Holds files for launching the code and other directories for managing code.

Launcher.py

runGame.bat

runGameDebug.bat

BugServer/common directory – This directory holds common files that can be easily accessed by other code modules.

Constants.py

DatabaseHelper.py

Entity.py

BugServer/db directory – This directory holds files related to using the local database.

SeriousGames.db

SeriousGames.sql

BugServer/db/bin directory – Directory holding binary for database.

SQLite Database Browser.zip

BugServer/main directory – This directory contains files and directories necessary for running the virtual environment.

Main.py

BugServer/main/World directory – This directory contains the functionality used for loading and controlling many aspects of the virtual environment.

World.py

BugServer/main/World/Battle directory – This directory contains the battle functionality and events for bugs.

Question.py

BugServer/main/World/Bug directory – This directory holds files for creating and keeping track of bugs.

Bug.py

BugGenerator.py

BugServer/main/World/Character directory – This directory holds files for creating and keeping of remote characters for the bug to potentially interact with.

RemoteCharacter.py

RemoteCharacterGenerator.py

BugServer/main/World/Environment directory – This directory contains files for managing the environment.

Environment.py

BugServer/models directory – This directory contains all the folders and resource files required to graphically create the environment and characters.

BugServer/models/bugs directory – This directory contains all the files and resource files for graphically creating bugs.

BugServer/models/bugs/bug1 directory – This directory holds resource files for the character bug1.

attack.egg.pz

attack2.egg.pz

bug1.egg.pz

bug-walk.egg.pz

death.egg.pz

death2.egg.pz

readme.txt

runWithAttack.egg.pz

spider_01.jpg

spider_01_blue.jpg

spider_01_green.jpg

spider_01_red.jpg

spider_01_yellow.jpg

BugServer/models/bugs/panda directory – This directory holds resource files for the character panda.

panda.egg.pz

panda.jpg

panda-walk.egg.pz

BugServer/models/characters directory – This directory holds resource files for creating in the virtual environment.

BugServer/models/characters/ralph directory – This directory holds resource files for creating the ralph directory.

ralph.egg.pz

ralph.jpg

ralph-run.egg.pz

ralph-walk.egg.pz

BugServer/models/scenes directory – This directory holds resource files for the environments and scenes.

BugServer/models/scenes/dungeonEasy directory – This directory holds all the resource files for creating the easy dungeon.

dungeonEasy.egg.pz

BugServer/models/scenes/dungeonEasy/dungeon directory – This directory holds all the resource files for creating dungeon.

digits_1024.png

digits_1024_blue.png

matrix.jpg

tron_green_white_1024.jpg

BugServer/models/scenes/easyLevelScene directory – This directory contains all the resource files required for creating the easy level environment.

easyLevel.egg.pz

BugServer/models/scenes/easyLevelScene/assets directory – This directory contains all the asset folders required to compose the scene.

BugServer/models/scenes/easyLevelScene/assets/textures directory – This directory contains the texture files used for detailing the scene.

BugServer/models/scenes/easyLevelScene/assets/textures/cpu directory – This directory contains all the resource files needed to create a CPU graphically in the scene.

cpu-front.jpg

cpu-back.jpg

grey-texture.jpg

mb.jpg

Plastic_Ola_Grey.jpg

BugServer/models/scenes/easyLevelScene/assets/textures/EasyBoard directory – This directory contains all the resource files needed to create the easy board background in the scene.

a4000tmb_rev4_4.jpg

digits_1024.png

digits_1024_blue.png

HD_Matrix_Wallpaper_by_aNdre_W.jpg

pixel_wallpaper3.png

tron_green_1024.jpg

tron_green_white_1024.jpg

BugServer/models/scenes/easyLevelScene/assets/textures/hard_drive directory – This directory holds all the files necessary to create a hard drive in the scene.

coral-red.jpg

Plastic_Ola_Black.jpg

silverTexture.jpg

strapTexture.jpg

textureHD.jpg

<u>BugServer/models/scenes/easyLevelScene/assets/textures/ports</u> directory – This directory holds all the files necessary to create a port in the scene.

Matrix tut 2.jpg

Plastic_Ola_Black.jpg

portBlock.jpg

silverTexture.jpg

sTex2l.jpg

sTexture.jpg

<u>BugServer/models/scenes/easyLevelScene/assets/textures/power_supply</u> directory – This directory holds all the resource files required for drawing the power supply in the scene.

back.jpg

black.png

coral-red.jpg

front.jpg

Medium Black Texture.jpg

Plastic_Ola_Black.jpg

ps_back.jpg

ps_top.jpg

side.jpg

side2.jpg

side3.jpg

sky-blue.jpg

texturePS.jpg

BugServer/models/scenes/easyLevelScene/assets/textures/ram directory – This directory holds all the resource files required for drawing the RAM in the scene.

ram.jpg

BugServer/models/scenes/easyLevelScene/assets/textures/resistors directory – This directory holds all the resource files required for drawing the resistors in the scene.

resistor.jpg

resistor_green.jpg

resistor_red.jpg

BugServer/models/scenes/easyLevelScene/assets/textures/wires directory – This directory holds all the resource files required for drawing the wires in the scene.

wire_blue.jpg

wire_green.jpg

wire_red.jpg

wire_yellow.jpg

BugServer/models/scenes/mainScene directory – This directory holds all the files required to create the main scene.

mainScene.egg.pz

BugServer/models/scenes/mainScene/assets directory – This directory holds all the folders designated to creating the main scene.

BugServer/models/scenes/mainScene/assets/textures directory – This directory holds all the folders containing all the different items that can be drawn in the main scene.

BugServer/models/scenes/mainScene/assets/textures/CD_Spindle directory – This directory holds all the resource files required for creating a CD spindle in the main scene.

grey-texture.jpg

Plastic_Ola_Grey.jpg

Plastic_texturerns.jpg

BugServer/models/scenes/mainScene/assets/textures/futureComputer directory – This directory holds all the resource files required for creating a futuristic computer in the main scene.

alienSilver.jpg

black.JPG

grey-texture.jpg

LinuxWallpaper.jpg

map-bw.jpg

metal_texture.jpg

transparency.jpg

BugServer/models/scenes/mainScene/assets/textures/imac2009 directory – This directory holds all the resource files required for drawing an imac in the main scene.

aluminumimac01.jpg

BA_big.jpg

black.jpg

brushed_silver_matt_prima_385x286.jpg

desktop.jpg

desktop-sharing-l.jpg

imac-early-2009.jpg

Silver-Texture.jpg

Sim Brushed Aluminum.jpg

BugServer/models/scenes/mainScene/assets/textures/mug directory – This directory holds all the resource files required for drawing a mug in the main scene.

red_ceramics.jpg

textureCeramic.jpg

textures4ever_vol2_sample1.jpg

BugServer/models/scenes/mainScene/assets/textures/oldComputer directory – This directory holds all the resource files required for drawing an older computer in the main scene.

computer.jpg

doslogo.jpg

keyboard.jpg

monitor.jpg

BugServer/models/scenes/mainScene/assets/textures/printer directory – This directory holds all the resource files required for drawing a printer in the main scene.

1010421407.jpg

BA_big.jpg

black.JPG

brushed_silver_matt_prima_385x286.jpg

Copy of multifuncional frente.jpg

f7393f7fb0e67a446a69226eec6b904d.jpg

Fabric_texture_white_by_Patterns_stock.jpg

frontTexture.jpg

HP logo.JPG

hpShop.jpg

hpShop.png

multifuncional frente.jpg

Silver-Texture.jpg

Sim Brushed Aluminum.jpg

texturise_plastic_texture_0005-500x363.jpg

BugServer/models/scenes/mainScene/assets/textures/stiffy_3.5 directory – The directory holds all the resource files required for drawing the stiffy object on the main scene.

stiffy back.tga

stiffy front.tga

BugServer/models/scenes/mainScene/assets/textures/tableTextures directory – This directory holds all the resource files required for drawing the table textures on the main scene.

2009_calendar.JPG

2009-AUGUST-print.jpg

deskwood.jpg

BugServer/models/scenes/mainScene/assets/textures/worldSphere directory – This directory holds all the resource files required for drawing the sphere shaped background in the main scene.

matrix-wallpaper.jpg

BugServer/models/scenes/world directory – This directory holds all the files necessary required to create a world scene.

ground.jpg

hedge.jpg

rock03.jpg

tree.jpg

world.egg.pz

<u>BugServer/net</u> directory – This directory holds all the files and folders required for making connections and updates to and from the server.

ConnectionManager.py

ServerRequestTable.py

ServerResponseTable.py

<u>BugServer/net/request</u> directory – This directory holds all the files making requests to the server.

RequestAttack.py

RequestBattleAcc.py

RequestBattleAdd.py

RequestBattleRemove.py

RequestBuddies.py

RequestBuddyAcc.py

RequestBuddyAdd.py

RequestBuddyRemove.py

RequestChat.py

RequestDead.py

RequestGlobalChat.py

RequestHeartbeat.py

RequestInventory.py

RequestInventoryAdd.py

RequestInventoryRemove.py

RequestItem.py

RequestLogin.py

RequestLogout.py

RequestMap.py

RequestMove.py

RequestPartyChat.py

RequestPrivateChat.py

RequestPublicChat.py

RequestQuestion.py

RequestRegister.py

RequestRegisterBug.py

ServerRequest.py

BugServer/net/response directory – This directory holds all the functionality for taking responses from the server.

ResponseAttack.py

ResponseBattleAcc.py

ResponseBattleAdd.py

ResponseBattleRemove.py

ResponseBuddies.py

ResponseBuddyAcc.py

ResponseBuddyAdd.py

ResponseBuddyRemove.py

ResponseChangeMap.py

ResponseChat.py

ResponseCreate.py

ResponseCreateBug.py

ResponseDead.py

ResponseGlobalChat.py

ResponseInventory.py

ResponseInventoryAdd.py

ResponseInventoryRemove.py

ResponseItem.py

ResponseLogin.py

ResponseLogout.py

ResponseMap.py

ResponseMove.py

ResponsePartyChat.py

ResponsePrivateChat.py

ResponsePublicChat.py

ResponseQuestion.py

ResponseRegister.py

ResponseRegisterBug.py

ResponseRemoveUser.py

ServerResponse.py

***Gaming* –** This directory holds all the code implemented to create the client application. The client application is what players will use to connect and interact with the game.

src directory – This directory is the main directory that holds all the folders containing functionality and graphic resources. It also holds files for launching the application.

Launcher.py

runGame.bat

runGameDebug.bat

Gaming/src/common directory - This directory holds common files that can be easily accessed by other code modules.

Constants.py

DatabaseHelper.py

Entity.py

Gaming/src/db directory - This directory holds files related to using the local database.

--SeriousGames.db

--SeriousGames.sql

Gaming/src/db/bin directory - Directory holding binary for database.

SQLite Database Browser.zip

Gaming/src/main directory - This directory contains files and directories necessary for running the game environment.

Main.py

Gaming/src/main/Login directory – This directory contains files in charge of creating the login screen and functionality.

Login.py

Gaming/src/main/Register directory – This directory contains files in charge of creating the registration screen and functionality.

Register.py

Gaming/src/main/World directory – This directory contains files that initialize and regulate the in game world environment and functionality of different events and properties in the virtual world.

World.py

Gaming/src/main/World/Battle directory – This directory contains the functionality of the battle system that the user experiences.

Battle.py

BattleSystem.py

Question.py

Gaming/src/main/World/Bubble directory – This directory contains the files responsible for the functionality of event driven, graphic, informational text bubbles that appear above a user's avatar.

ChatBubble.py

DamageBubble.py

NameBubble.py

Gaming/src/main/World/Bug directory – This directory holds functionality for a bug appearing on a user's screen.

RemoteBug.py

RemoteBugGenerator.py

Gaming/src/main/World/Camera directory – This directory holds functionality for camera control.

Camera.py

Gaming/src/main/World/Character directory – This directory holds the functionality for user properties and showing other characters and creating characters.

Character.py

RemoteCharacter.py

RemoteCharacterGenerator.py

<u>Gaming/src/main/World/ControlScheme</u> directory – This directory contains the functionality for the control scheme.

ControlScheme.py

<u>Gaming/src/main/World/Environment</u> directory – This directory contains the information for creating, loading and maintaining the environment.

Environment.py

<u>Gaming/src/main/World/Gui</u> directory – This directory contains the files necessary for creating and loading user interface.

Gui.py

MainBar.py

<u>Gaming/src/main/World/Gui/CharacterInfo</u> directory – This directory contains files that creates and holds functionality for displaying character information.

CharacterInfo.py

<u>Gaming/src/main/World/Gui/Chat</u> directory – This directory contains files that create and maintain the chat box.

Chat.py

<u>Gaming/src/main/World/Gui/Friends</u> directory – This directory contains files that create and maintain the friends menu.

Friends.py

<u>Gaming/src/main/World/Gui/Inventory</u> directory – This directory contains files that create and maintain the inventory menu.

Inventory.py

Gaming/src/main/World/Gui/MiniMap directory – This directory contains files that create and maintain the mini map on the screen.

MiniMap.py

Gaming/src/main/World/MousePicker directory – This directory contains files that create and maintain the mouse picker.

MousePicker.py

Gaming/src/main/World/NPC directory – This directory contains files that maintains functionality for non playable characters that appear in the virtual environment.

Dialog.py

NPC.py

NPCLoader.py

Shop.py

Gaming/src/models directory **-** This directory contains all the folders and resource files required to graphically create the environment and characters. It also holds images used by user interface.

bottom_632.gif

bottom_632.png

button.png

CharacterInfo.png

ChatButton.png

DownArrow.png

edgespattern_632.gif

edgespattern_632.png

epicCharacter1.png

FriendsButton.png

InventoryButton.png

MainBar.png

MapButton.png

poc_title_signupplay_632.gif

poc_title_signupplay_632.png

Quit.png

rroll.png

shoe.png

signup.png

smiley.egg.pz

static_ralph.jpg

swk.png

UpArrow.png

Xbutton.png

Gaming/src/models/avatarimages directory – This directory holds the images used by the registration screen to scroll through avatar selections.

1.png

2.png

3.png

4.png

5.png

6.png

7.png

8.png

Gaming/src/models/bugs directory – This directory contains resource files for drawing the bugs/enemies in the game environment.

Gaming/src/models/bugs/bug1 directory – This directory contains resource files for drawing a bug1 character.

attack.egg.pz

attack2.egg.pz

bug1.egg.pz

bug1-walk.egg.pz

death.egg.pz

death2.egg.pz

readme.txt

runWithAttack.egg.pz

spider_01.jpg

spider_01_blue.jpg

spider_01_green.jpg

spider_01_red.jpg

spider_01_yellow.jpg

Gaming/src/models/bugs/panda directory – This directory contains resource files for drawing a panda character in the virtual environment.

panda.egg.pz

panda.jpg

panda-walk.egg.pz

Gaming/src/models/characters directory – This directory contains folders which hold resource files for drawing player characters in the virtual environment.

Gaming/src/models/characters/females directory – This directory contains folders which hold resource files for drawing female player characters.

<u>Gaming/src/models/characters/females/ralphSister</u> directory – This directory holds resource files for drawing the ralph sister character.

eve.egg.pz

eve.mb

eve.png

eve_screenshot.jpg

eveface.png

eve-jump.egg.pz

eve-offbalance.egg.pz

eve-run.egg.pz

eve-tireroll.egg.pz

eve-tireroll.mb

eve-walk.egg.pz

model-inf.dat

tire.png

<u>Gaming/src/models/characters/males</u> directory – This directory holds folders which hold resource files for drawing the male characters in the game environment.

readme.txt

<u>Gaming/src/models/characters/male/male1</u> directory – This directory holds resource files for drawing the male1 character.

male.egg.pz

male-run.egg.pz

man-black-black-grey.JPG

texture.jpg

Gaming/src/models/characters/male/male2 directory – This directory holds resource files for drawing the male2 character.

male.egg.pz

male-run.egg.pz

man-black-red-green.JPG

texture.jpg

Gaming/src/models/characters/male/male3 directory – This directory holds resource files for drawing the male3 character.

male.egg.pz

male-run.egg.pz

man-grey-green-blue.JPG

texture.jpg

Gaming/src/models/characters/male/male4 directory – This directory holds resource files for drawing the male4 character.

male.egg.pz

male-run.egg.pz

man-white-red-yellow.JPG

texture.jpg

Gaming/src/models/characters/male/male5 directory – This directory holds resource files for drawing the male5 character.

male.egg.pz

male-run.egg.pz

man-white-white-yellow.JPG

texture.jpg

Gaming/src/models/characters/male/ralph directory – This directory holds resource files for drawing the ralph character.

bvw-f2004--ralph-egg.zip

ralph.egg.pz

ralph.jpg

ralph-run.egg.pz

ralph-screenshot.jpg

ralph-walk.egg.pz

Gaming/src/models/characters/ralph directory – This directory holds resource files for drawing the ralph character.

ralph.egg.pz

ralph.jpg

ralph-run.egg.pz

ralph-walk.egg.pz

Gaming/src/models/characters/rockstar directory – This directory holds resource files for drawing the rockstar character.

rockstar.egg.pz

rockstar-run.egg.pz

Gaming/src/models/characters/rockstar/textures directory – This directory holds resource files for the rockstar character textures.

rockstar.bmp

Gaming/src/models/characters/smiley directory – This directory holds resource files for drawing the smiley in the game environment.

smiley.egg.pz

Gaming/src/models/minimap directory – This directory holds folders that contain resources for displaying the minimap graphic.

dungeonEasy.jpg

easyLevelMap.jpg

mainSceneMap.jpg

Gaming/src/models/minimap/1 directory – This directory contains resources for displaying the minimap graphic.

easyLevelMap.JPG

loginMap.jpg

mainSceneMap.JPG

minimapImage.jpg

plane.egg.pz

point.jpg

point2.png

Gaming/src/models/scenes directory – This directory contains folders which hold resource files for drawing different scenes.

Gaming/src/models/scenes/dungeonEasy directory – This directory contains resource files for the easy dungeon scene.

dungeonEasy.egg.pz

Gaming/src/modles/scenes/dungeonEasy/dungeon directory – This directory holds resource files for adding graphics to the easy dungeon scene.

digits_1024.png

digits_1024_blue.png

matrix.jpg

tron_green_white_1024.jpg

Gaming/src/models/scenes/easyLevelScene directory – This directory holds the resource files for creating the easy level scene.

easyLevel.egg.pz

Gaming/src/models/scenes/easyLevelScene/assets directory – This directory contains all the asset folders required to compose the scene.

Gaming/src/models/scenes/easyLevelScene/assets/textures directory - This directory contains the texture files used for detailing the scene.

Gaming/src/models/scenes/easyLevelScene/assets/textures/cpu directory – This directory contains all the resource files needed to create a CPU graphically in the scene.

cpu_front.jpg

cpu-back.jpg

grey-texture.jpg

mb.jpg

Plastic_Ola_Grey.jpg

Gaming/src/models/scenes/easyLevelScene/assets/textures/EasyBoard directory - This directory contains all the resource files needed to create the easy board background in the scene.

digits_1024.png

digits_1024.psd

digits_1024_blue.png

HD_Matrix_Wallpaper_by_aNdre_W.jpg

pixel_wallpaper3.png

tron_green_1024.jpg

tron_green_project.psd

tron_green_white_1024.jpg

Gaming/src/models/scenes/easyLevelScene/assets/textures/hard_drive directory - This directory holds all the files necessary to create a hard drive in the scene.

coral-red.jpg

Plastic_Ola_Black.jpg

silverTexture.jpg

strapTexture.jpg

texture.psd

textureHD.jpg

Gaming/src/models/scenes/easyLevelScene/assets/textures/ports directory - This directory holds all the files necessary to create a port in the scene.

Matrix tut 2.jpg

Plastic_Ola_Black.jpg

portBlock.jpg

silverTexture.jpg

sTex2l.jpg

sTexture.jpg

Gaming/src/models/scenes/easyLevelScene/assets/textures/power_supply directory - This directory holds all the resource files required for drawing the power supply in the scene.

back.jpg

black.png

coral-red.jpg

front.jpg

Medium Black Texture.jpg

Plastic_Ola_Black.jpg

ps_back.jpg

ps_top.jpg

side.jpg

side2.jpg

side3.jpg

sky-blue.jpg

texture.psd

texturePS.jpg

whole.psd

Gaming/src/models/scenes/easyLevelScene/assets/textures/ram directory **-** This directory holds all the resource files required for drawing the RAM in the scene.

ram.jpg

Gaming/src/models/scenes/easyLevelScene/assets/textures/resistors directory **-** This directory holds all the resource files required for drawing the resistors in the scene.

-resistor.jpg

resistor.psd

resistor_green.jpg

resistor_red.jpg

Gaming/src/models/scenes/easyLevelScene/assets/textures/wires directory **-** This directory holds all the resource files required for drawing the wires in the scene.

wire_blue.jpg

wire_green.jpg

wire_red.jpg

wire_yellow.jpg

Gaming/src/models/scenes/loginScene directory – This directory holds all the files required for creating the login scene.

loginScene.egg.pz

loginSceneModeled.mb

Gaming/src/models/scenes/loginScene/assets directory – This directory holds all the resource files required to draw assets for the login scene.

14-Motherboard side 2.jpg

alienSilver.jpg

matrix-wallpaper.jpg

metal_texture.jpg

monster.jpg

red.jpg

Gaming/src/models/scenes/mainScene directory – This directory holds all the files required to create the main scene.

mainScene.egg.pz

README.txt

Gaming/src/models/scenes/mainScene/assets directory **-** This directory holds all the folders designated to creating the main scene.

<u>Gaming/src/models/scenes/mainScene/assets/maps</u> directory – This directory holds maps for the main scene.

<u>Gaming/src/models/scenes/mainScene/assets/textures</u> directory – This directory holds all the folders containing all the different items that can be drawn in the main scene.

<u>Gaming/src/models/scenes/mainScene/assets/textures/CD_Spindle</u> directory - This directory holds all the resource files required for creating a CD spindle in the main scene.

grey-texture.jpg

Plastic_Ola_Grey.jpg

Plastic_texturerns.jpg

<u>Gaming/src/models/scenes/mainScene/assets/textures/CD_Spindle/.mayaSwatches</u> directory – This directory holds maya swatches for the CD spindle.

Plastic_Ola_Grey.jpg.swatch

Plastic_texturerns.jpg.swatch

<u>Gaming/src/models/scenes/mainScene/assets/textures/futureComputer</u> directory - This directory holds all the resource files required for creating a futuristic computer in the main scene.

alienSilver.jpg

black.JPG

grey-texture.jpg

LinuxWallpaper.jpg

map-bw.jpg

metal_texture.jpg

transparency.jpg

Gaming/src/models/scenes/mainScene/assets/textures/futureComputer/.mayaSwatches directory – This directory holds maya swatches for future computer.

alienSilver.jpg.swatch

LinuxWallpaper.jpg.swatch

metal_texture.jpg.swatch

Gaming/src/models/scenes/mainScene/assets/textures/imac2009 directory – This directory holds all the resource files required for creating an imac in the main scene.

aluminumimac01.jpg

BA_big.jpg

black.jpg

brushed_silver_matt_prima_385x286.jpg

desktop.jpg

desktop-sharing-l.jpg

imac-early-2009.jpg

Silver-Texture.jpg

Sim Brushed Aluminum.jpg

Gaming/src/models/scenes/mainScene/assets/textures/imac2009/.mayaSwatches directory – This directory holds maya swatches for imac2009.

BA_big.jpg.swatch

black.jpg.swatch

desktop.jpg.swatch

desktop-sharing-l.jpg.swatch

imac3.mb.swatches

imacFinal.mb.swatches

Silver-Texture.jpg.swatch

Sim Brushed Aluminum.jpg.swatch

Gaming/src/models/scenes/mainScene/assets/textures/mug directory – This directory holds all resource files for creating a mug on the main scene.

red_ceramics.jpg

textureCeramic.jpg

textures4ever_vol2_sample1.jpg

Gaming/src/models/scenes/mainScene/assets/textures/mug/.mayaSwatches directory – This directory holds maya swatches for mug.

red_ceramics.jpg.swatch

Gaming/src/models/scenes/mainScene/assets/textures/oldComputer directory – This directory holds all resource files for creating an old computer on the main scene.

computer.jpg

doslogo.jpg

keyboard.jpg

monitor.jpg

Gaming/src/models/scenes/mainScene/assets/textures/oldComputer/.mayaSwatches directory – This directory holds maya swatches for old computer.

computer.jpg.swatch

doslogo.jpg.swatch

keyboard.jpg.swatch

monitor.jpg.swatch

Gaming/src/models/scenes/mainScene/assets/textures/printer directory – This directory holds all resource files for creating a printer on the main scene.

1010421407.jpg

BA_big.jpg

black.JPG

brushed_silver_matt_prima_385x286.jpg

Copy of multifuncional frente.jpg

f7393f7fb0e67a446a69226eec6b904d.jpg

Fabric_texture_white_by_Patterns_stock.jpg

frontTexture.jpg

HP logo.JPG

hpShop.jpg

hpShop.png

multifuncional frente.jpg

Silver-Texture.jpg

Sim Brushed Aluminum.jpg

texturise_plastic_texture_0005-500x363.jpg

Thumbs.db

Gaming/src/models/scenes/mainScene/assets/textures/printer/.mayaSwatches directory – This directory holds maya swatches for printer.

1010421407.jpg.swatch

hpShop.png.swatch

Gaming/src/models/scene/mainScene/assets/textures/stiffy_3.5 directory – This directory holds all the resource files required to create stiffy in the main scene.

stiffy back.tga

stiffy front.tga

Gaming/src/models/scene/mainScene/assets/textures/stiffy_3.5/.mayaSwatches directory – This directory holds the maya swatches for stiffy.

stiffy back.tga.swatch

stiffy front.tga.swatch

Gaming/src/models/scene/mainScene/assets/textures/tableTextures directory – This directory holds all the resources files to create the table and its textures in the main scene.

2009_calendar.JPG

2009-AUGUST-print.jpg

deskwood.jpg

Gaming/src/models/scene/mainScene/assets/textures/tableTextures/.mayaSwatches directory – This directory holds the maya swatches for table textures.

2009_calendar.JPG.swatch

2009-AUGUST-print.jpg.swatch

deskwood.jpg.swatch

Gaming/src/models/scene/mainScene/assets/textures/worldSphere directory – This directory holds all the resource files for drawing the sphere shaped background in the main scene.

matrix-wallpaper.jpg

Gaming/src/models/scene/mainScene/textures directory – This directory holds all the textures for the main scene (deprecated).

Gaming/src/models/scene/world directory – This directory holds all the resource files for creating the outside world environment in the game.

ground.jpg

hedge.jpg

rock03.jpg

tree.jpg

world.egg.pz

<u>Gaming/src/models/screenshots</u> directory – This directory holds screenshots for the different scenes.

dungeon1.jpg

dungeon2.jpg

easyLevel.jpg

easyLevel2.jpg

easyLevel3.jpg

easyLevel4.jpg

loginScene0.jpg

mainScene1.jpg

mainScene2.jpg

mainScene3.jpg

mainScene4.jpg

mainScene5.jpg

<u>Gaming/src/net</u> directory - This directory holds all the files and folders required for making connections and updates to and from the server.

ConnectionManager.py

ServerRequestTable.py

ServerResponseTable.py

<u>Gaming/src/net/request</u> directory - This directory holds all the files making requests to the server.

RequestAttack.py

RequestBattleAcc.py

RequestBattleAdd.py

RequestBattleRemove.py

RequestBuddies.py

RequestBuddyAcc.py

RequestBuddyAdd.py

RequestBuddyRemove.py

RequestChat.py

RequestDead.py

RequestGlobalChat.py

RequestHeartbeat.py

RequestInventory.py

RequestInventoryAdd.py

RequestInventoryRemove.py

RequestItem.py

RequestLogin.py

RequestLogout.py

RequestMap.py

RequestMove.py

RequestPartyChat.py

RequestPrivateChat.py

RequestPublicChat.py

RequestQuestion.py

RequestRegister.py

RequestRegisterBug.py

ServerRequest.py

<u>Gaming/src/net/response</u> directory **-** This directory holds all the functionality for taking responses from the server.

ResponseAttack.py

ResponseBattleAcc.py

ResponseBattleAdd.py

ResponseBattleRemove.py

ResponseBuddies.py

ResponseBuddyAcc.py

ResponseBuddyAdd.py

ResponseBuddyRemove.py

ResponseChangeMap.py

ResponseChat.py

ResponseCreate.py

ResponseCreateBug.py

ResponseDead.py

ResponseGlobalChat.py

ResponseInventory.py

ResponseInventoryAdd.py

ResponseInventoryRemove.py

ResponseItem.py

ResponseLogin.py

ResponseLogout.py

ResponseMap.py

ResponseMove.py

ResponsePartyChat.py

ResponsePrivateChat.py

ResponsePublicChat.py

ResponseQuestion.py

ResponseRegister.py

ResponseRegisterBug.py

ResponseRemoveUser.py

ServerResponse.py

Server – The server acts as a mediator for clients, bugs and the database. The server facilitates different information it receives from bugs and clients and sends responses to each, updating them with different states, positions, and other information. The server also is responsible for retrieving information from the database, making it available to clients.

Nursetown_Server/ - Root directory. Holds different launch and configuration files for the server to utilize.

game_server_config.txt

game_server_config.txt.bak

GameServer.jar

makeJar.bat

makeJar.sh

manifest.txt

runServer.bat

runServer.sh

runServerDebug.bat

runServerDebug.sh

<u>Nursetown_Server/core</u> directory – This directory holds most constants and logic for mediating between different server functionalities. It also holds files for configuring and handling different types of connections made to the server.

BugServerProcess.java

ConnectionPool.java

Constants.java

CountingDownSec.java

GameClient.java

GameDB.java

GameItem.java

GamePacket.java

GamePacketStream.java

GameQuestion.java

GameRequestTable.java

GameUser.java

NurseGameServer.java

NurseGameServerConfig.java

NurseGameServerShutdown.java

<u>Nursetown_Server/lib</u> directory – This directory holds different connection libraries for the server code to use.

commons-dbcp-1.2.2.jar

commons-pool-1.5.3.jar

mysql-connector-java-5.1.10-bin.jar

<u>Nursetown_Server/request</u> directory – This directory holds functionality for requests made from the server. These usually are from world events that happen. Each different file specifies how to handle and mediate each type of request so the server can process them.

GameRequest.java

RequestAnimate.java

RequestAnswer.java

RequestAttack.java

RequestBattleAcc.java

RequestBattleAdd.java

RequestBattleRemove.java

RequestBoardMove.java

RequestBuddies.java

RequestBuddyAcc.java

RequestBuddyInvite.java

RequestBuddyRemove.java

RequestChatGlobal.java

RequestDead.java

RequestDice.java

RequestHealthInsert.java

RequestHeartbeat.java

RequestImg.java

RequestInventory.java

RequestInventoryRemove.java

RequestItem.java

RequestLogin.java

RequestLogout.java

RequestMove.java

RequestMsg.java

RequestPrivateChat.java

RequestPublicChat.java

RequestQuesLoad.java

RequestQuestion.java

RequestQuiz.java

RequestQuizAcc.java

RequestQuizEnd.java

RequestQuizStart.java

RequestRegister.java

RequestRegisterBug.java

RequestSceneId.java

RequestSendActive.java

RequestSetState.java

RequestTransition.java

Nursetown_Server/response directory – This directory contains files that deal with the functionality of the server sending responses to different targets. Each file sends updates based on events, such as game events, in order for other targets to be updated in a near real time process.

GameResponse.java

ResponseAddGroup.java

ResponseAnimate.java

ResponseAnswer.java

ResponseAttack.java

ResponseAuth.java

ResponseBattleAcc.java

ResponseBattleAdd.java

ResponseBattleRemove.java

ResponseBoardMove.java

ResponseBuddies.java

ResponseBuddyAcc.java

ResponseBuddyRemove.java

ResponseChangeScene.java

ResponseChatGlobal.java

ResponseChatPrivate.java

ResponseCreate.java

ResponseCreateBug.java

ResponseDead.java

ResponseDice.java

ResponseHealthInsert.java

ResponseImg.java

ResponseInventoryRemove.java

ResponseItem.java

ResponseLogout.java

ResponseMove.java

ResponseMsg.java

ResponsePublicChat.java

ResponseQuesLoad.java

ResponseQuestion.java

ResponseQuiz.java

ResponseQuizAcc.java

ResponseQuizEnd.java

ResponseQuizStart.java

ResponseRegister.java

ResponseRegisterBug.java

ResponseRemoveGroup.java

ResponseRemoveUser.java

ResponseSceneId.java

ResponseSendActive.java

ResponseSendBuddyInvite.java

ResponseSetState.java

ResponseTransition.java

ResponseUpdate.java

Nursetown_Server/sql directory – This directory deals with different functionalities for updating the server.

buddy.sql

bug.sql

user.sql