# DEBUGGER

- Documentation -

*Prepared By: Gary Ng*
*Last Updated: 10-19-2014*

# Table of Contents

**INTRODUCTION** .......... **3**
GAME OVERVIEW .......... 3

**GETTING STARTED** .......... **3**
INSTALLATION GUIDE .......... 3
CLIENT-SIDE DEVELOPMENT .......... 4
*Game Client Requirements & Resources* .......... *4*
*Quick Start Guide* .......... *5*
SERVER-SIDE DEVELOPMENT .......... 5
*Game Server Requirements & Resources* .......... *5*
*Bug Server Requirements & Resources* .......... *6*
*Quick Start Guide* .......... *6*

**ARCHITECTURE DESIGN** .......... **7**
QUICK OVERVIEW .......... 7
GAME CLIENT .......... 8
GAME SERVER .......... 8
BUG SERVER .......... 9
FLOW LOGIC .......... 9

**GAME CLIENT OVERVIEW** .......... **11**
GETTING STARTED .......... 11
*Main* .......... *12*
*Login* .......... *12*
*World* .......... *12*
GLOBAL VARIABLES .......... 13
*Custom Global: main* .......... *13*
EXTENDED PANDA3D DIRECT CLASSES .......... 13
DATABASE .......... 13
*SQLite Overview* .......... *13*
*Usage* .......... *13*
*Convert to Python Script* .......... *14*
*Localization* .......... *14*
WINDOW MANAGEMENT .......... 14
TASKS .......... 14
NETWORKING .......... 15
*Connection Manager* .......... *15*

**GAME SERVER OVERVIEW** .......... **16**
GETTING STARTED .......... 16
*Game Server* .......... *16*
*Bug Server Monitor* .......... *16*
*Game Script Handler (NPCs, Shops, etc.)* .......... *17*
*Game Client* .......... *17*

DATABASE ........ 17
*Overview* ........ *17*
*Table: user* ........ *17*
*Table: questions* ........ *18*
NETWORKING ........ 18

**NETWORKING OVERVIEW ........ 18**
EXAMPLE: LOGGING IN ........ 18

**CLIENT DISTRIBUTION ........ 19**
OVERVIEW ........ 19
INSTRUCTIONS ........ 19

**GAME FEATURES ........ 21**
PARTY ........ 21
BOARD GAME ........ 21
SURVIVAL ........ 22
DUEL ........ 23
NPCS ........ 23
HOTKEYS ........ 25
EQUIPMENT ........ 25
ITEMS ........ 26
CHAT ........ 26
FRIENDS ........ 27

**REFERENCES ........ 28**
IMPORTANT LINKS ........ 28
OTHER RESOURCES ........ 28

# Introduction

The deBugger project was originally conceived in a game development course that was set out to design a game to help students go over Computer Science material that they've already or currently being covered in class. Players are dealt with multiple-choice questions relating to the concepts and fundamentals of Computer Science to further improve their learning and understanding in the subject.

deBugger utilizes the Panda3D engine for the client side that is connected to a Java-based server for network connectivity allowing online interaction between players. Developers will need to gain knowledge in Panda3D, Python, Java, and MySQL to further develop this project.

## Game Overview

deBugger, similar to traditional MMOGs (Massively Multiplayer Online Games), connects everyone together inside a virtual world using avatars. At its core, players will destroy bugs by successfully answering a series of multiple-choice questions. These questions become more difficult as bugs get tougher. Other game types include playing a board game, filling in variable values under a time limit, code completion and more.

Players can also level up by destroying bugs. Higher-level players gain access to areas with bugs that challenge them with more difficult questions, but before they can get there, they must go through the lower level areas with simpler questions.

Socializing is a very important aspect in this game. Using the built-in text chatting system, players can communicate with others that are connected. This can be used to share interests, ask for help or even turn the game into a virtual meeting for discussions. Party system is also available for those that want to explore the world together.

# Getting Started

## Installation Guide

Once you have unzipped the deBugger Source package, you should have in front of you GameClient.zip, GameServer.zip, and BugServer.zip. Proceed to unzip those packages.

Before you can get started with development, you will need to prepare your environment by first grabbing the necessary tools such as the Panda3D SDK, IDEs, and MySQL server mentioned further below in their sections.

Starting with the GameServer and BugServer, you will have to execute a MySQL script found in the GameServer to initialize and setup your database. Be sure to double-check the tables once the scripted has been executed. Once the database is properly set up, you will also need to modify a configuration file, again, found in the GameServer, so you can decide on the port number and set up the database connection information. When all is said and done, you should be able to run both the GameServer and BugServer.

As for the GameClient, the one included in this package is the full source code and not the self-extracting installer found in the deBugger website. You will need to configure it to connect to the GameServer you have running either locally or on another machine.

You can create a self-extracting installer for the GameClient using the source code to distribute to other users, so anytime there are changes to the GameClient such as localization and any other modifications, you will need to create a new installer every time. Instructions for this process can be found under *Client Distribution*.

For a more detailed explanation of this whole procedure, please refer to the following sections below called *Client-Side Development* and *Server-Side Development*.

## Client-Side Development

### Game Client Requirements & Resources

**Platforms**

- Microsoft Windows, Mac OS X, Linux

**Panda3D Manual**

- http://www.panda3d.org/manual/index.php/Main_Page

**Panda3D SDK**

- Panda3D SDK 1.8.1 or later version is required.
    - http://www.panda3d.org/download.php?sdk
- Panda3D SDK 1.9.0 Experimental Build (Resolves 1.8.x issues on Mac OS X)
    - http://rdb.name/Panda3D-1.9.0.dmg

**Panda3D Runtime (For Client Distribution)**

- Panda3D Runtime 1.0.4 or later version is required.
    - http://www.panda3d.org/download.php?runtime

**Development Tools**

- IDE w/ Python support
    - Eclipse – http://www.eclipse.org/
        - PyDev Plugin
            - http://pydev.org/manual_101_install.html
    - NetBeans – http://www.netbeans.org/
        - Python Plugin
            - https://blogs.oracle.com/geertjan/entry/python_in_netbeans_ide_8

### Quick Start Guide

Setting up the environment differs depending on your development platform. After doing so, you should be able to launch the client directly from your IDE. Refer to the links above to download the necessary tools mentioned below.

1. Install the latest Panda3D SDK.

2. If you don't already have an IDE w/ Python support, install one of the two suggested IDEs, Eclipse or NetBeans. You will have to install the Python plugin afterwards such as PyDev for Eclipse.

3. Go ahead and import the source code into your IDE by creating a new project. If you already have an existing Python interpreter, you will need to point your project to the Python interpreter (ppython) that comes packaged with the Panda3D SDK for full compatibility. For help relating to your IDE, please refer to its documentation.

4. You will need to configure the connection settings such as host and port, which are found in *Constants*.py under the *common* directory. These settings must match with the server you are connecting to. You can look for the server's setup guide in the next section.

5. There are two ways to run the client during development. You can execute the game within your IDE by running *Launcher.py* as the main driver. The other option is by executing one of the scripts found in the source root with a prefix of *runGame* depending on your platform.

6. You may have to set permissions for these scripts to be executable. For Windows, you shouldn't have to do anything. For OSX, you will need to use the Terminal to perform the following:

```
chmod 744 runGame*.command
```

7. You should have now successfully set up your environment for client development.

## Server-Side Development

### Game Server Requirements & Resources

**Platforms**

- Microsoft Windows, Mac OS X, Linux

**Development Tools**

- IDE w/ Java support
    - Eclipse – http://www.eclipse.org/
    - NetBeans – http://www.netbeans.org/

**Database Tools**

- MySQL
    - MySQL Community Server – http://www.mysql.com/
    - phpMyAdmin – http://www.phpmyadmin.net/
        - MAMP (For OSX) – http://www.mamp.info/
        - XAMPP (For Windows) – http://www.apachefriends.org/

## Bug Server Requirements & Resources

*Please refer to Game Client Requirements & Resources under Client-Side Development.*

## Quick Start Guide

There are 3 major components needed for server-side development.

### Game Server Component

The main component of server-side is simply what we called the Game Server. It's Java-based that utilizes a MySQL database for storage and a Bug Server component that takes care of the Bugs' logic you see in the game.

1. If you haven't already installed an IDE w/ Java support, you should choose from either Eclipse or NetBeans.

2. Create a new project and import the source folder into your IDE.

3. Once you have your project imported, you'll now have to import additional libraries located in the *lib* directory since the server need these dependencies. The option should be located in our project properties.

4. You will need to configure the *game_server_config.txt* found in the root directory. You must choose an available port number to use. The database information as well as the Bug Server information can be added later once you reach to the bottom of this section.

5. To run the Game Server from your IDE, you'll need to set the *GameServer.java* found in the *core* directory as the main driver. You still need to configure the database and Bug Server first before it can run. There are scripts similar the Game Client that compiles and runs the server from outside the IDE.

### Database Component

The MySQL database backend is required for the Game Server to run, so you'll need access to a MySQL database server.

1. Install one of the following listed above under Database Tools. Installing either MAMP or XAMPP will give you access to phpMyAdmin to manage your database from the browser. Refer to their respective documentation for further instructions to get you set up.

2. You'll need to first create a new database so you can populate it with the necessary tables in the next step. Create the database using *utf8_general_ci* collation as the default so it can support Unicode text if needed. Existing tables that are set using another collation may need to be changed to support Unicode as well.

3. Import the database schema found in the *sql* directory in your Game Server. Look for a SQL script called *deBuggerDB.sql*. You will want to import that into your empty database you created.

4. Modify the *game_server_config.txt* file found in the Game Server with the database information so it can connect properly.

**Bug Server Component**

The Bug Server's main purpose is to centralize the bug's logic and decision making over on the server side as oppose to the player's client side so that all connected players will see the same actions happening. The Game Server initiates it, but needs to be configured as well so it'll know how to connect as if it was a game client.

1. Similar to the Game Client, you'll need to install the Panda3D SDK if you haven't already.

2. If you've changed the default port number the server uses, you'll need to edit *Constants.py* under the *common* directory and ensure it matches the server port you're using.

3. Once again, the Game Server will start the Bug Server as long as the absolute path configured in *game_server_config.txt* is set correctly. By default, the Bug Server is active as long as the main server is running. Applying *-s* as an argument when running the server will activate the Bug Server's Smart Mode, which means that it will run only if there are any users logged in.

## Architecture Design

### Quick Overview

deBugger as a whole is comprised of three major components—Game Client, Game Server and Bug Server. Each component is designed in such a way to be able to communicate between each other over a network through requests and responses.

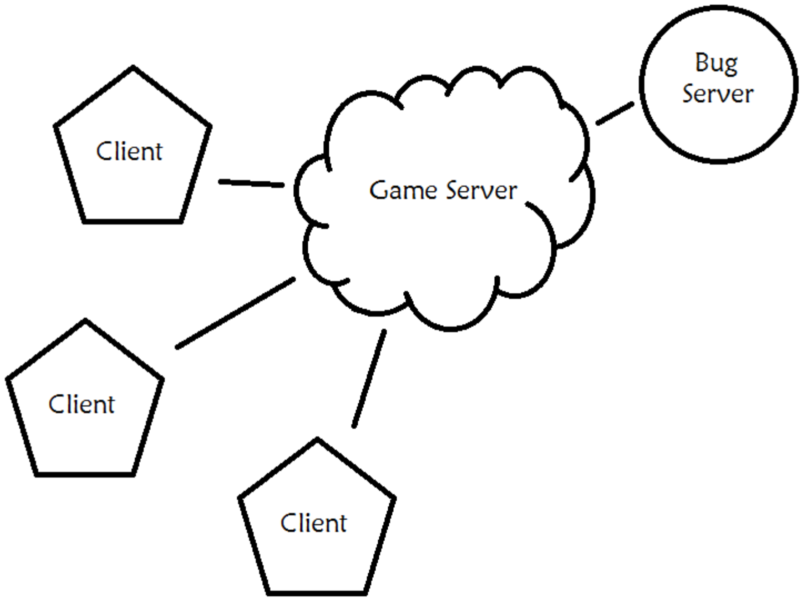

Figure 1

## Game Client

The Game Client is written in Python to utilize the Panda3D game engine. This allows the game engine to take care of various aspects of the game such as rendering, collision detection, networking and much more. In relation to the Game Server, the client is responsible to read user input and sends any relevant information to the server for processing.

Users are capable of performing character movements, menu interactions, chat and other various forms of actions that are sent to the server in the form of packets. For example, a request called a heartbeat is sent every frame to poll the server for newly updated information. It is also used to maintain a connection with the server. Without the heartbeat, any results or responses from your actions will not be retrieved and can result in a timeout situation, which leads to a disconnection.

## Game Server

The Game Server's core functionality is written in Java, while incorporating a little bit of Python for situations where scripting is required. The use of Java allows better ease of creating a network application such as this server. The server is required in order to act as a central unit to allow players to log into the game and perform tasks such as keeping track of player and bug movements, spawning, death, as well as sending the questions, item drops, chat notices, etc. to players.

It utilizes a MySQL database separate from the server to store and retrieve information in a persistent manner, which it connects through using JDBC. Information such as questions, stats and various other things lies within the database for easy retrieval.

For every client that is connected, a separate thread is created and reserved for each connection that handles all of its incoming requests and outgoing responses. Each incoming request is processed as it comes, whereas the outgoing responses are being held in a queue. In order to release these responses, a heartbeat request must come from the pertaining client.

## Bug Server

The Bug Server is basically a modified client with its graphics window disabled that works closely with the Game Server. It manages the movement of bugs walking in a scene while making use of Panda3D's collision system to determine the exact location in 3D space. These positions gathered will be used to update all connected clients the exact locations of each bug. The Bug Server keeps track of other various pieces of information about a bug as well such as its health points, level, aggression as well as initiating attacks against players. Unlike the Game Client, the Bug Server can support multiple maps at one given time. That way, only one instance of a Bug Server is needed.

## Flow Logic

The initial point of the deBugger game starts at the client, as shown below. From there, the client goes through multiple steps to work its way through the cycle. This cycle begins the moment the client makes connection with the server and repeats several times as long as the game is running. The main loop of the cycle makes use of either an event dictionary (Game Client) or an event hash map (Game Server) that will look up the events, as each receives it, and produce results on either end.

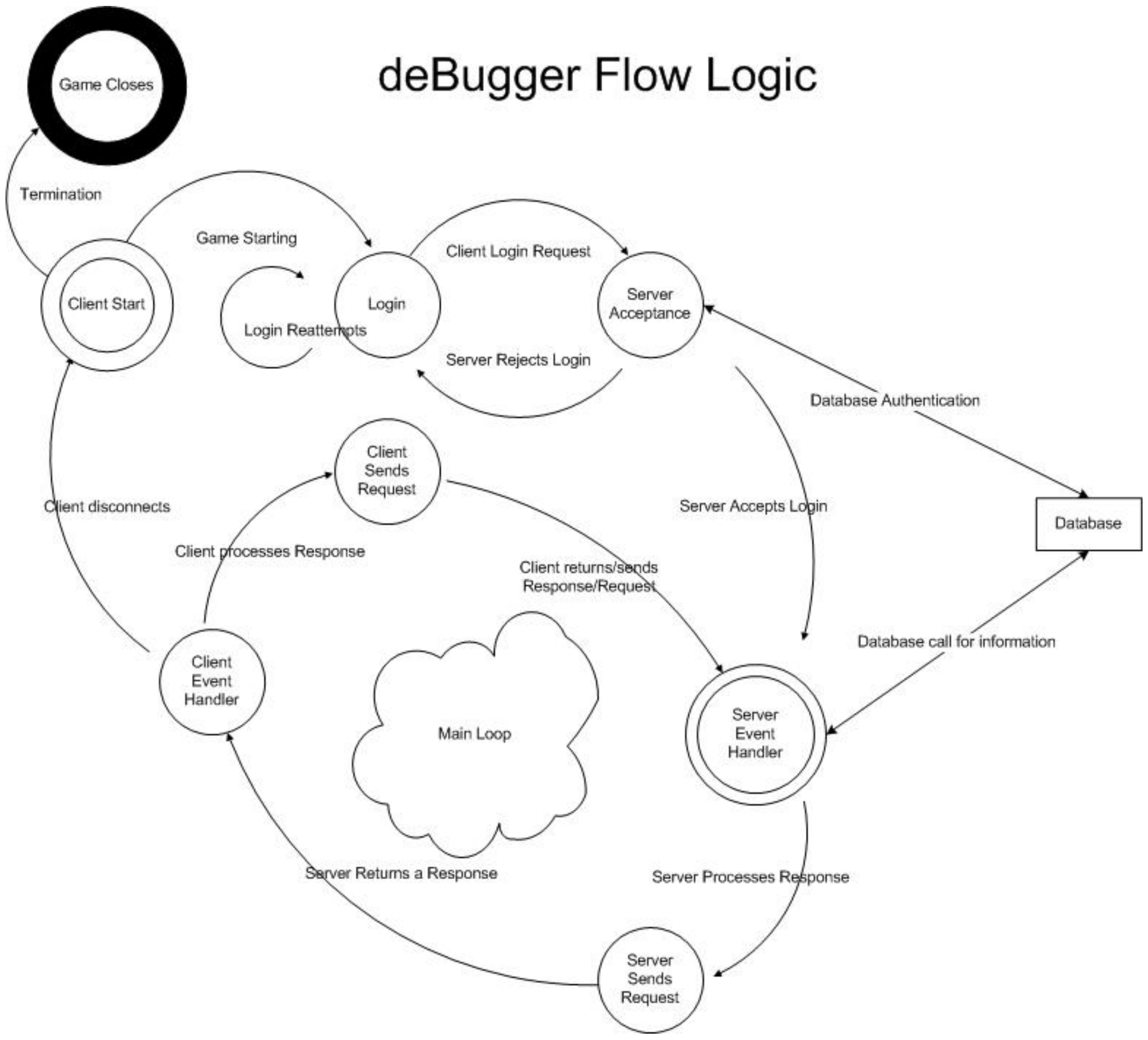

Figure 2

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 player can proceed to play the game.

## Client

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 a surface 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 another event with its current position. This is done through a task that occurs at regular intervals.

## Server

Whenever a connection has been established, the server creates a thread for each of those connections. This allows the server to focus on forming and removing connections, while allowing each of the threads to deal with the packets sent by the client. Each of these connections listens to the specified connection that was created when the player logged in with specifically for updates from that client. Depending on the event, the server will update every other connected client and places this information into the queue that each thread maintains until the next heartbeat. At each heartbeat, the server will send the client all of the queued updates since the last heartbeat request.

**Communication**

Information is being passed between client and server in the form of packets or also known as datagrams in Panda3D. A typical packet will look something like the following:

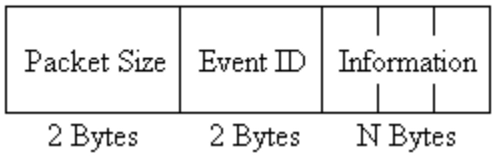

Figure 3

Using TCP for our networking, the packets will be read in the same order as they were sent. The size of the packet determines how much information included needs to be read. Following the size is what we called an event ID. This helps distinguish what type of request or response contained within the packet so that both the client and server will know how to process the remaining information.

Due to the nature of Java, the server, running on top of the Java Virtual Machine, must perform an extra step to process the incoming data before being read. It's something described as the data's endianness or also known as the byte order. Data can be in either big-endian (Server), where the most significant byte is ordered first or little-endian (Client), where the least significant byte is ordered first. Because of these differences, the server must covert the incoming bytes by reversing, through shifting bits, its order for it to read properly otherwise it'll produce unexpected results. This conversion process does not happen on client-side, so outgoing data from the server must be reversed before being sent to the client.

## Game Client Overview

### Getting Started

This section will provide you with details of certain modules that are connected to get the game running. Take a look at the following diagram for these specific modules:

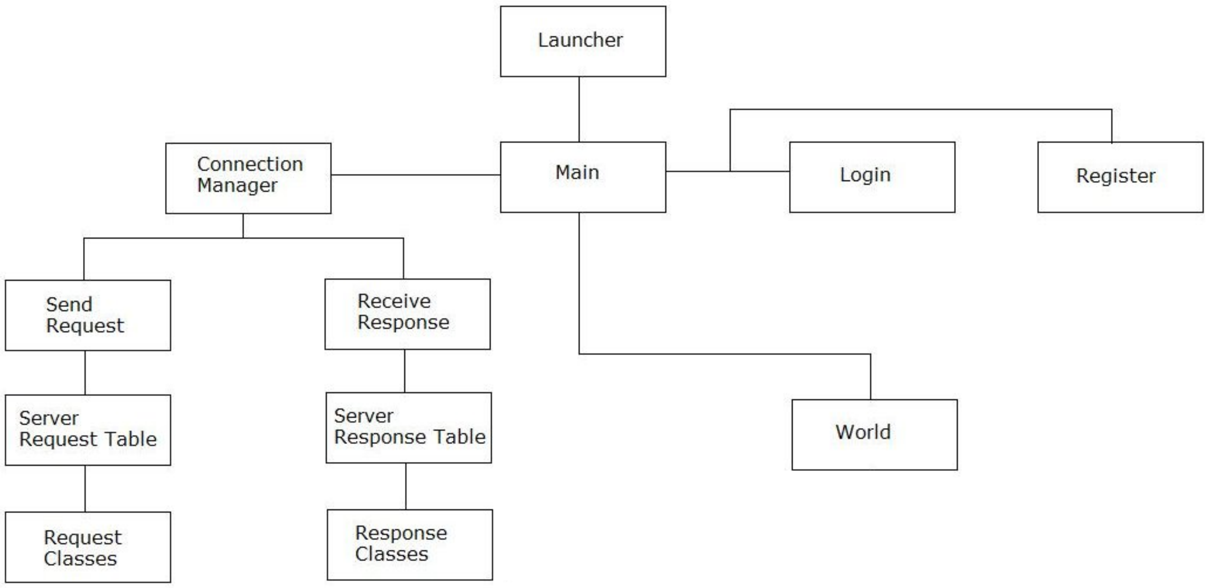

Figure 4

The first module called *Launcher* serves as a simple driver that initiates the *Main* module. If there were any extra command arguments found, these arguments would be processed before initiating the *Main* module.

## Main

The *Main* module is responsible for initializing certain variables used for the display window such as resolution, title name, aspect ratio, etc. It is also responsible for loading the *Connection Manager* module, which is used to handle all requests that go out and responses that come in. Not only does it handle the creation of the connection manager, it contains a task that is responsible to send a heartbeat every so often to retrieve data back from the server. Another major functionality of the *Main* module is that it controls the switching between *Login*, *Register* and *World*.

## Login

The *Login* module will be the first to be loaded to accept login information from the user, which passes it to the connection manager to send off to the server for authentication. Once authentication is successful, an event will be generated to switch from *Login* to *World*.

## World

The *World* module is responsible for many things. It loads up several different instances that include the character, map, camera, menus, etc. It manages almost everything that occurs in the game including the existence of all characters, bugs and NPCs.

## Global Variables

There are several global variables provided by Panda3D such as render, aspect2d, loader, base, base.camera, task and many more. These variables provide quick access to some of the functionality that Panda3D has to offer.

### Custom Global: main

There is only one important global variable created for this game, which is called *main*. This variable provides quick access to the very top of the hierarchy that contains references to many different objects and such. Doing so, there's no need to keep passing reference of the root to any other modules that may need access to a method in a particular module. Initialization of this variable is located in the *Main.py* script.

## Extended Panda3D Direct Classes

A few classes found in the *common* folder should look very similar to existing Panda3D classes with names beginning with *Direct*. These classes are simply the same found in Panda3D except that they include extended functionality that allows consistency with what a basic window should look like specifically in this game, dealing issues with text entry that Panda3D hasn't got around fixing, simple window management and many more. Due to the lack of built-in styling and skinning, these classes provide a unique look from standard built-ins.

## Database

### SQLite Overview

Similar to a MySQL database, SQLite allows the client to store key information such as model path mappings, level music, item descriptions, etc. Found in the *db* folder, there is a file called *SeriousGames.db*, which is essentially the database storing all these tables used by SQLite.

But due to the somewhat problematic client packaging process for distribution, there is also another file called *SeriousGames.py* served as an alternative which is simply the database tables converted by a script called *Database2Python.py* and stored as a Python script using dictionaries. This is currently being used over the database file for distribution.

### Usage

Data must be inserted into *SeriousGames.db* using a tool called *SQLite Database Browser* found in the *bin* folder. Here you can find a version for Microsoft Windows or Mac OS X, so unpack one if you haven't already. Once unpacked, you simply load up the tool and open *SeriousGames.db*.

You'll find the following tables: *avatar*, *bug*, *space_item*, *map_object*, *game*, *item*, *map*, *map_type*, *msg*, *npc*, *skill*, *tip*, *view*.

Not all tables are being used such as *space_item* and *map_object.* Tables such as *item* and *msg* contains strings used by the client and others such as *avatar* and *bug* are used to map the models with an ID.

### Convert to Python Script

For distribution purposes, you will have to run the *Database2Python.py* script from your IDE or command-line to convert the database into a Python script to produce *SeriousGames.py*. The reason for this is Panda3D isn't able to locate the *SeriousGames.db* file when packaged up, so this is the alternative solution until it is fixed.

### Localization

For localization purposes, tables such as *item* and *msg* will be need to be modified by replacing the individual strings with its localized form. Some strings may not be stored inside the database so it will require modifying the strings found inside the source code. Other strings such as questions found in-game exist on the server's database, so changes are dealt there.

## Window Management

Using our extended classes of *DirectFrame*, a simple window management system is created to determine which window needs to be focused and its properties enabled, while those in the background should have their certain functionalities disabled to prevent any unwanted interruptions due to events being triggered.

Located within *Main*, there are a few methods that will keep track of all existing windows and will be responsible to maintain the order at which they were created, so that whenever a window needs to be removed, a previous window will automatically be focused for use.

Since only one window can be focused at one time, this will prevent an issue where multiple separate windows would pick up the same event such as one created by the *Enter* key, which is not acceptable. One example would be each window containing a text field, the *Enter* key event is designed to focus onto a text field to allow typing, but it just happens that both text fields will be focused instead of the one you want and this is where this window management allows better control.

## Tasks

One characteristic that most modules have in common is a method, a task, called *updateRoutine*.

For example, the one shown below is taken from the *Main* module that contains a task responsible for sending a heartbeat to the server once per frame.

```
def updateRoutine(self, task):
    self.cManager.sendRequest(Constants.CMSG_HEARTBEAT)
    return task.again
```

These similar tasks exist for characters, camera, combat and many more as well. It is simply a method that runs like a loop to perform actions required by a specific module. For instance, a character needs to walk across from one place to another. An *updateRoutine* method similar to the one above will be responsible to move the character forward by a small distance every frame. Over time, the character will finally reach its destination.

The main loop of the game will simply manage all tasks that were created and call each task by the order it was created. All of these tasks will simply be executed once per frame. For more details about tasks, please go through the Panda3D manual.

## Networking

### Connection Manager

A *Connection Manager* exists for one main reason, which is to create a connection with the server to exchange data from the client to the server and back. Types of data being sent fall under one of two categories, request or response. A request is simply used to send data from the client to the server. A response is data received from the server, which will be processed to perform said actions.

The following method and format is required to prepare a request through the connection manager:

```
rContents = {'username' : username,
             'password' : md5(password).hexdigest()}
self.cManager.sendRequest(Constants.CMSG_AUTH, rContents)
```

This example is taken from the *Main* module to show that the login request requires exactly two arguments of information, username and password. A request type and a dictionary of both arguments will be passed into the *sendRequest* method located in *ConnectionManager* for further preparation before sending it off to the server.

The *Connection Manager* is now responsible to wait for any information in the form of a response being sent from the server. An *updateRoutine* method will be constantly running every frame to check for any incoming data.

Once the response is received, the data contained within the packets will be read in the order it was prepared by the server. Using the above example, if the authentication was successful, the client will load the character, world and everything else that is required as a result of this particular response.

# Game Server Overview

## Getting Started

This section will provide you with details of how certain classes are connected to get the server running. Take a look at the following diagram for these specific classes:

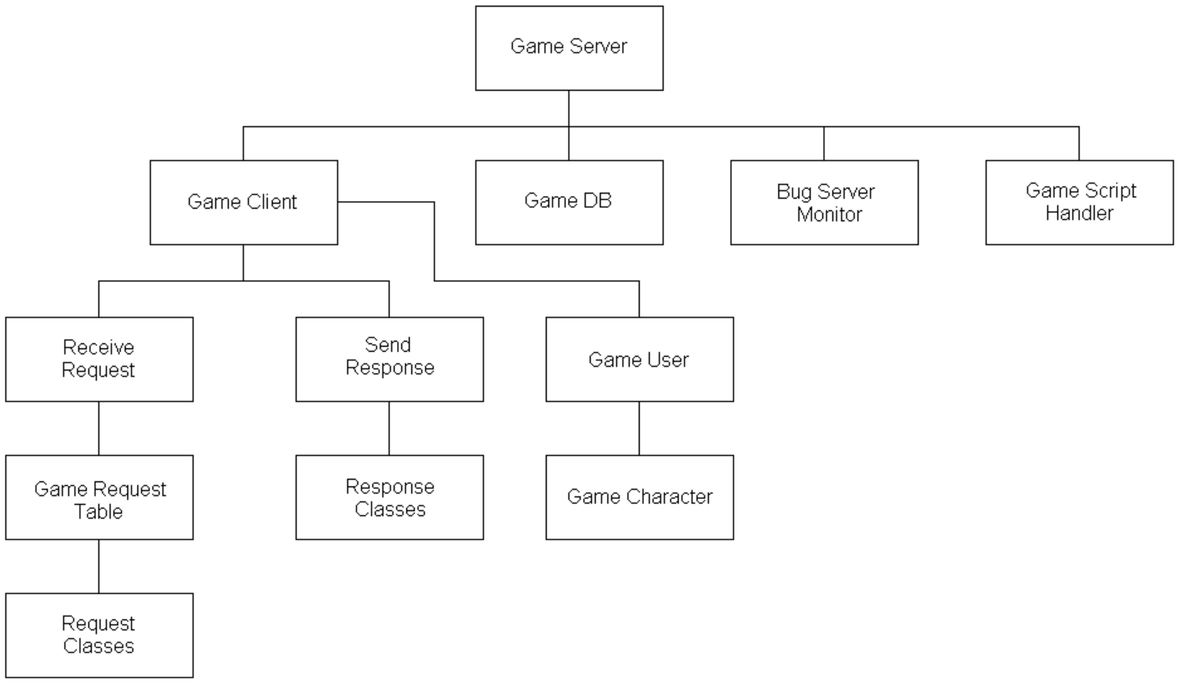

Figure 5

### Game Server

The top most class called *Game Server* is considered the main component of the server. It is responsible for creating connections with connecting clients. The class provides many other important functions that are widely used by other classes such as *Game Client*, *Game DB*, *Bug Server Monitor*, *Game Script Handler*, etc.

Since the server utilizes a MySQL database for storing and retrieving data, a class called *GameDB* is created to provide different methods to accomplish these actions.

### Bug Server Monitor

The *Bug Server Monitor* class is responsible for locating the Bug Server and booting it up whenever needed. The Bug Server can either stay active at all times or only whenever at least one user is connected. If the Bug Server is booted up in *Smart Bug Mode*, this class will start it up whenever a user is detected and closes whenever no users are detected.

### Game Script Handler (NPCs, Shops, etc.)

Another important class is the *Game Script Handler*. It is responsible for integrating the Python language into the Java written server. Using Python, scripts can be written to give NPCs a certain degree of A.I. to perform specific actions requested by the users.

These scripts are found in the *npc* folder. Notice that you see a series of function calls to allow this dynamic scripting behavior integrated into Java. These functions themselves are declared within *GameScriptHandler.java*. Because it is Python, you have to adhere to the tab indentation rule as well.

One of these scripts such as *npc.txt* allows you to create a NPC and designate it at a location with a certain conversation dialog that accepts user inputs. Every script is different since it follows a different formatting.

### Game Client

There are many other important classes that aren't mentioned, but the last important class as seen in the diagram is called the *Game Client*. Whenever a user connects to the server, a connection is created. Along with that connection, a separate thread called the *Game Client* is created to handle all actions, requests, made by the user. So basically, for every user, there's a *Game Client* thread created for a particular user. Within each *Game Client*, a *Game User* is created to store all account-related information for the user. For every *Game User*, a *Game Character* is created to store all character-related information. There's one exception, whenever a Bug Server connects to the server, a *Game User* is created, but not a *Game Character* since it does not use a character after all.

## Database

### Overview

The Game Server uses a MySQL database to store accounts, player progress, questions, etc. There are several dozen tables found in the database and its all access by using queries stored inside the *GameDB* class found under the *core* directory.

### Table: user

The *user* table is responsible for storing all accounts created by the players. There are a few important accounts to keep in mind.

| Username | Password | Description |
|---|---|---|
| admin | password | Reserved strictly for the Bug Server to connect |
| 1 | 1 | Dummy Account "Red" |
| 2 | 2 | Dummy Account "Green" |

| | | |
|---|---|---|
| 3 | 3 | Dummy Account “Blue” |

The first account should not be removed otherwise the Bug Server will have trouble connecting.

#### Table: questions

The *questions* table stores all questions being used to display to the player during bug encounters and including other game mode. Any changes such as adding or even localizing must be done within this table. Questions are not stored on the client’s database. They are delivered from sever to client as they are requested in the game.

### Networking

As briefly mentioned before, a *Game Client* instance is created upon each unique connection made with the server. Within this very instance, there exists a similar loop found within the *Game Server* class that runs over and over except that it is a little more complicated than just creating a thread for each connection.

When the loop is running, it will check whether there’s any data within the buffered stream to be processed. Remember that each packet contains three major components—Packet Size, Event ID and Information. Here are the steps of the process:

1. Check buffer if any packet data exists.
2. Retrieve the size of each packet by reading the first 2 bytes.
3. Extract that specific size amount of information from the buffer to be processed.
4. From the extraction, the first 2 bytes will be read as the Event ID.
5. Any more information read from the extraction will be process by their Request class.
6. Repeat until user disconnects.

Most requests will always have at least some information that needs to be sent to the client, so a single to several responses using a Response class of their own will be created within those requests.

Now how does that work? Each response created will either be sent right away to the client or it will be inserted into a list located within *GameUser*. The list will keep track of all existing responses until a heartbeat request from the client is received. Once it receives a heartbeat request, which occurs about every few milliseconds decided by the client, all existing responses would be dumped and sent straight to the client for processing.

## Networking Overview

### Example: Logging In

Client-side:

- User inputs credentials handled by *Login*
- Client uses *ConnectionManager* to send a request called *RequestLogin* to the server

Server-side:

- Picks up a request by the loop within *GameClient*
- Determines this is a *RequestLogin* type request
- Processes credentials against what is stored within the database through *GameDB*
- Depending on the success, a response called *ResponseLogin* will be created
- Once all necessary information, such as user information, is prepared, it will send this response to the client
- Every other client connected will also be informed by another response that this particular user has logged in and will create a character representing this user

Client-side:

- The loop running within *ConnectionManager* will pick up a response
- Determines this is a *ResponseLogin* type request
- Perform all necessary measures to process incoming data such as user information

# Client Distribution

## Overview

Once your development version is ready, it needs to be packed up and distributed to users as an executable that they can install. To start off, you will need to install the Panda3D Runtime found in the Game Client Requirements & Resources section.

Although not perfect, the tool can in fact create a package for multiple platforms so for example, you can create a Mac client or even Linux client using a Windows machine. There are some know issues doing this on a Mac as it can fail to produce a Windows client.

Microsoft Windows is strongly recommended.

## Instructions

Some rules to create a proper executable you should adhere to:

1. Create a copy of the entire Game Client folder somewhere and perform the following modifications below in this copy.

2. In *Main.py*, remove or comment out the line at the very top few lines with *show-frame-rate-meter*. This is not necessarily needed for the users.

3. Remove all the scripts found in the root that begin with *runGame*.

4. Remember to convert your database to Python script if there were any changes made. Then remove the *bin* folder, *Database2Python.py*, *SeriousGames.db*, and *SeriousGames.sql*.

5. Open your Command Prompt (PC) or Terminal (Mac) and locate the copy of the Game Client directory. Execute the following command:

```
packp3d -o "deBugger.p3d" -d "deBugger" -m Launcher.py -r audio -r
morepy
```

   This will create a single P3D file containing the full client.

   Here's a brief explanation of each parameter:

| Parameter | Description |
|---|---|
| o | Output Filename |
| d | Client Directory Name |
| m | Launcher Script |
| r | Panda3D Libraries |

   For more details go to:
   https://www.panda3d.org/manual/index.php/Using_packp3d

6. Next step is to pack up the P3D into an executable for all relevant platforms such as Microsoft Windows, Mac OS X, and many others. Execute the following command:

```
pdeploy -N "deBugger" -v 1.03 -t width=1024 -t height=640 -P win32 -
P linux_i386 -P osx_i386 -s -A "SFSU" "deBugger.p3d" installer
```

   This will create an executable for Windows, OS X, and Linux. This will greatly depend on what platform, version, author you want to create.

   Here's a brief explanation of each parameter:

| Parameter | Description |
|---|---|
| N | Game Name |
| v | Version Number |
| P | Platform |
| A | Author |

   For more details go to:

https://www.panda3d.org/manual/index.php/Distributing_as_a_self-contained_installer

7. For Mac OS X clients, it's best practice to pack up the .PKG file generated into a .DMG image container. Execute the following command using a Mac:

```
hdiutil create -srcfolder "deBugger_1.03" -volname "deBugger" -fs
HFS+ -fsargs "-c c=64,a=16,e=16" -format UDBZ "./deBugger_1.03.dmg"
```

   Here's a brief explanation of each parameter:

| **Parameter** | **Description** |
|---|---|
| srcfolder | Folder Containing PKG |
| volname | Game Name |

8. Make sure you test each executable before distributing. You can now remove the Game Client copy.

## Game Features

### Party

A party comprises of up to 5 players that feel the need to form a group to keep each other in check, but is also required for certain situations such as starting a board game. Players can see what level other players are at and how much life they have left. It can also be used to see who's online.

To create a party, one player must find another player to bring up the Action Menu and select Add Party. Once the other player accepts the invitation, a party will automatically be created under the creator's name. The creator will now be assigned as the leader of the party, which has rights to invite more players, kick a player from party and lastly, being able to disassemble it.

Players are not allowed to invite others that are already in a party. And as stated, only the leader has rights to invite more players. Anyone that's not a leader can only leave the party and that's about it. If an existing party no longer has more than one player, it will be disassembled automatically.

Parties are really required only for playing the Board Game or Survival.

### Board Game

Aside from Bug Hunting, players in the same party are able to compete against each other in a board game. Just like an ordinary board game, players are given a predetermined path with a variety of tiles. The objective is simple, be the first to reach the end. But in order to do so, players must face obstacles in the form of questions to advance forward.

To access the board game, a player of a given party must talk to the NPC called *Student* located at the lower right of *The Table* map.

Each instance of a board game has two forms of customization, mode and topic. There are two different modes called Regular and Race:

Regular mode offers players an ordinary turn-based game. The roll of a dice determines the amount of necessary steps required to move along the path. Upon reaching the destination, the current turn's player will face a randomly selected question. Answer right, stay in new position. Answer wrong, return to previous position. Then it alternates to the next player and repeats until someone reaches the end.

Race mode is a bit different. Instead of players taking turns to roll a dice and answering questions, everything happens at the same time. A dice will be rolled by the game, not the players. Once the result is determined, everyone will face the same randomly selected question. A short delay is given at the exact moment to every player for a chance to read the question. Once the delay is over, the first to answer the question correctly will advance forward the amount steps the dice rolled. For those that failed to answer first correctly, their position stays the same. Again, same objective, first to reach the end wins.

The second form of customization relates to topic selection. The selected topic determines the type of questions to be asked. The higher the level required for that topic, the harder and more rewarding the questions are. But of course, all players in the party are required to be of a certain level for the selected topic.

The predetermined paths of every board game consist of regular and special tiles. The meaning of each tile is really simple as different tiles provide different effects. Regular tiles are marked as green, which only reward experience for answering correctly. Special tiles are basically non-green tiles, which can either give bonuses or bring you back to the start. The different tiles and description are as follows:

| | | |
|---|---|---|
| Red Tile | - | Starting Point |
| Yellow Tile | - | Finishing Point; Reward experience and money to winner |
| Green Tile | - | Reward experience |
| Blue Tile | - | Bonus experience modifier |
| Orange Tile | - | Reward money |
| Black Tile | - | Send player back to starting point; Regular Mode only |

## Survival

Survival puts players into a different scenario where they are fed with waves after waves of bugs. Each wave gets more difficult as you get higher up. Similar to the Board Game, the player sets the difficulty. A player can either go solo or with a party he/she chooses. But the objective is really simple, complete all the waves and defeat a boss at the very end. What make Survival special are the bosses as they are exclusive to this mode; not found anywhere else. Bosses are larger and harder than regular bugs, but will greatly reward the players who defeat it.

To access Survival, a player will have to talk to the NPC called *Technician* located at the upper-right of *The Table* map.

There may be two modes, Solo and Party, but both are the same. Choosing Solo will only warp yourself, whereas the other will warp your entire party that's around you. A party is recommended, as bosses will take quite some time to defeat. After selecting a mode, topic selection will be available. Topics determine what kind of bugs to spawn that affects the difficulty of each instance of Survival. Fighting bugs are no different than fighting them outside. Survival bugs are really just obstacles preventing you from getting to the boss where the excitement is really at. If you die at any point, you simply have no choice but to warp yourself back out and start over.

## Duel

Dueling allow players to challenge each other, testing their knowledge and power. Using knowledge for speed and power from weapons and leveling up should help you beat your opponent quite easily. So basically, destroy your opponent before the time runs out.

To access the Duel feature, both players must be at least level 10. The challenger will have to bring up the Action Menu and select a topic for consent to initiate the duel.

Upon selecting Duel from the Action Menu, the challenger will be given a list of topics. The topic selected for each duel determines what kind of questions both players are asked. More topics become available as players get higher level. The levels required to unlock these topics are exactly the same as every other game mode. To be fair, the actual list of topics being available is determined by the lowest level of both players. Once the duel begins, both players will have 10 minutes to fight it out. Just like fighting with bugs, players must answer questions correctly to inflict damage to their opponent. But there is one difference; players who answer incorrectly will not be punished with penalty damage. Since this is a duel between two players, only one will come out as the winner, but if the timer runs out, the duel ends with a draw. If one of the players bail out by switching map or logging off, it is considered a forfeit.

## NPCs

NPCs are non-playing characters. These characters are NPC for Wave game, Board Game, and for quick help. They guide you to the respective functions mentioned above.  The other three NPCs represent shop section. Shop section is divided in three parts namely item shop, weapon shop and armor shop.

The other NPCs are bugs, which are the "bad guys" of the game.

The basic working for NPC is as follows

*Server side Implementation:*

The information of all the NPCs is stored in one main file *scripts.txt*. This file defines the different types of NPCs we have in the game.

According to the type of NPC respective NPCs are loaded and the information from those NPCs is extracted. For e.g., if the NPC type is shop, the 'shop.txt' is loaded; the respective content is displayed.

All the working of NPCs is controlled via request – response protocols between client and server.

In order to activate the NPCs a request for 'NPC Talk' is sent from client.

```
rContents = {'npc_id'    : self.npc_id,
             'action'    : self.lastOption + 2}
self.world.main.cManager.sendRequest(Constants.CMSG_NPC_TALK, rContents)
```

A search is made to select the proper requested NPC according to the NPC id. Based on the respective NPC proper text file is selected using npc.getscript function which is written in a helper class named "GameNPC.java".

The content in the text file is the information and position of the NPC and the message it will display. Such content is extracted and sent back to the client via a response.

If the requested NPC is shop then the shop is loaded and the items in the shop are displayed accordingly.

If there is a request from client to sell an item from inventory, then the inventory content is checked and the corresponding item is removed from user's inventory and proper response is sent back to client.

If the request is for buying a particular item from the shop then the shop contents are checked and updated, also user is checked for eligibility to buy the item and his bytes are updated with a proper response to update client.

If the requested NPC is bug then a search is made in the existing bug list to check if the requested bug exists or not. If yes, then the contents are sent giving the information of spawning the bug.

A new bug is registered with the *RegisterBug* request and is added to the bug list.

*Client Side implementation:*

The client requests NPCs to load in the game via "request NPC Talk" along with NPC id. The information received from the server is used to load the model and display the respective content.

If the NPC is shop then respective shop with its id and name received from the server is loaded and the corresponding items are displayed along with their information such as ‘name’, ‘price to buy’, ‘the item effect’. For e.g., if the shop type is ‘gear’ then the items pertaining to ‘gear’ shop are displayed along with their description such as ‘name’, ‘effect’, ‘required level’ and ‘buy price’.

Similarly, in order to sell the item from inventory, the corresponding NPC id, inventory id and the amount, these contents are sent to the server with the ‘Request NPC sell’ request. The respective response from server contains the information to update the client.

For any item to purchase “request NPC buy” is sent along with ‘npc id’, ‘item id’ and ‘amount’. Upon receiving the response the client is updated.

Bugs are also NPCs, which are requested via *RequestBug* class. A response from the server sets out the information about the bug, which is used to spawn the bug and load it in the game.

## Hotkeys

Hotkeys are the items placed in the hotkey bar, for quick use of them. To create Hotkey, just drag the items into the hotkey bar; and drag them out of the bar/ or double click on them in order to make use of them.

The hotkeys are requested through “request hotkeys” and server sends back the hotkey list to the client in server.

This addition to the hotkey bar happens via request-response protocol between client and server. A request for addition of hotkey (item) is made through “request Hotkey Add.” Along with the request protocol, other information of item included are ‘type’, ‘item_id’ and ‘slot’ in the hotkey bar.

The server then updates the empty slots with the requested item and sends the updated information to the client who in turn updates the slot in the hotkey bar and displays the added item in the bar.

Similarly removal of the item is carried out with the request to “remove hotkey” and information of the item and slot number is sent to server. The server updates itself and sends the proper response.

## Equipment

Equipment are the weapons players a use in the game to fight against the bugs. Equipment help players in fighting bugs by providing them with some help. Equipment can be bought or earned. They are added in the inventory by client sending request to the server. In order to show the Equipment, a request, “request equipment”, is sent to the server.

At the server side, the equipment list is extracted according to the character requesting the equipment and sent it across. While adding equipment into inventory the information of the respective equipment such as id, type, price to buy, price to sell, amount, and equipment location is sent in the response. At the client side, the local database is updated with the new equipment.

In order to remove equipment, a request for "equipment remove" is sent along with inventory id where the equipment is placed. On server side the database is updated by removing the information of the equipment and updated response is sent to client, which updates local database and show the updated inventory to the player.

## Items

Items are the accessories player can make use of in the game. They can be of different types. Some items can help player enhance his/ her avatar and some can help him/her in the gameplay if used during the battle. Few items help player by giving him/her hints about the questions, while few help by supplying health portion.

Items can be acquired in following ways:

- Buy an item – Every item has a particular price and level associated with it. If a person satisfies these criteria then he/she can buy those items.
- Gain an item – Items can be gained by killing bugs. Some bugs drop items when they die, the player killing such bugs receives those items.
- Progressing in the game – Completing levels and/ or successfully finishing wave/ board game can help you climb to next levels. Upon getting into particular levels more items are open for buying.

Following are the items the game currently has:

- Health portions in small, medium, large, and mega sizes which restore health of player accordingly.
- Hint cards, which can be used only in battles. They are based no different chapters or levels. They reveal hint for a question upon usage.
- Weapons such as wooden sticks, books, shields, etc. can increase attack power or defense power.
- Accessories such as shirts, jeans, glasses etc., can be used for moving speed, attack time, defense power, health potion, or just for avatar customization.

## Chat

Chat is a way of communication between players in the game. It is helpful to share information with players who are online at the same time. Chat has following modes:

- Public:  Public chat is between players who are friends in the game.
- Private: Private chat is between two players without knowledge of any other player.

- Party: Party chat is between members of same party.
- Global: Global chat is open chat. Anybody who is online playing can participate in this chat.

All the types of chat follow the similar request-response protocol between server and client. Each type of chat is easy to recognize as they start with a particular operator. Global starts with '!', public with ' ', party with '%' and 'private with '/'. Either of these operators in the chat window describes what chat mode a player is into.

With each message the client also sends a code assigned to particular chat mode to the server in request. The server parses the request by checking which user the information is coming from, the message and the code for the chat. And then responds back to the client.

```
rContents = {'name' : targetUser,
             'msg'  : msg}
self.world.main.cManager.sendRequest(Constants.CMSG_PRIVATE_CHAT,
rContents)
```

In case of private chat mode, along with message and code, player also needs to write the name of the other player he needs to send the message. The request is sent in following format.

The client then sends this information to the server. Sever checks the status of the other player, if he/she is online that time and respectively send the message back to client else it send the response back to client with the indication of the other player being offline. Client then accordingly sends the message if the player is online or else informs the sender that other player is offline.

## Friends

All MMOs have game played with having friends in the game. Debugger also supports it. Thus in order to add friends, a player just needs to right click on the other player avatar and select "Add Friend"
When a player selects the a player to add as a friend, a request is sent from the client to server via "request add buddy"

```
rContents = {'name' : self.friendsEntry.get()}
self.world.main.cManager.sendRequest(Constants.CMSG_BUDDY_ADD, rContents)
```

Every time before sending this request the check is made if the requested player is already in friend list, or if the player is adding himself or if there are white spaces instead the name of the player. Once the request is sent to the server, the server then checks the online status of the requested player and prepares an invitation, which he sends it ack to the client. Client upon receiving the response from the server, with the online status of the requested player, sends the invitation and informs the requesting player that the requested player is unavailable or the invitation is sent.

Now, when a player receives the invitation, the client has a procedure to handle the invitation. This is also a request in following form.

```
rContents = {'invitation_id'     : invitation_id,
             'status'            : status}
self.world.main.cManager.sendRequest(Constants.CMSG_BUDDY_ACCEPT,
rContents)
```

The status is checked at the server and the corresponding invitation id, which reveals the sender's information, this information is sent over back to the client. Client then according to the status, displays the information to respective players.

## References

### Important Links

- deBugger Website
    - http://smurf.sfsu.edu/~debugger/
- Panda3D
    - http://www.panda3d.org/
- Videos
    - http://www.youtube.com/user/deBuggerSFSU

### Other Resources

- NetBeans
    - http://www.netbeans.org/
- Eclipse
    - http://www.eclipse.org/
- MySQL
    - http://www.mysql.com/