GPU

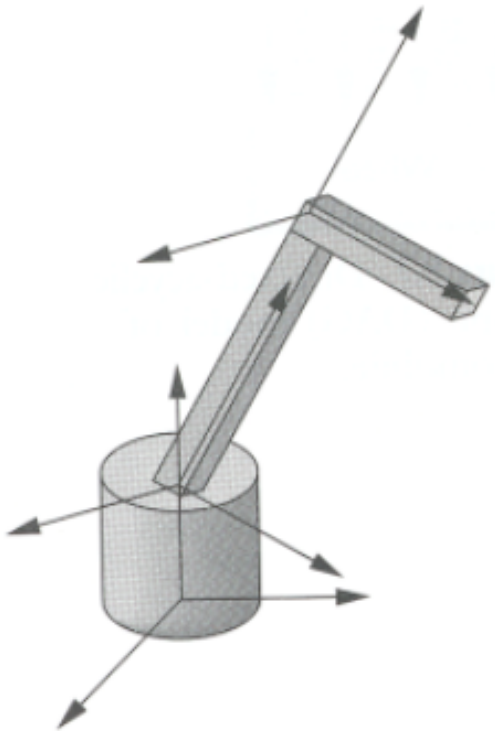
Slide Credits

- Marc Olano (UMBC)
  - SIGGRAPH 2006 Course notes
- David Luebke (University of Virginia)
  - SIGGRAPH 2005, 2007 Course notes
- Mark Kilgard (NVIDIA Corporation)
  - SIGGRAPH 2006 Course notes
- Rudolph Balaz and Sam Glassenberg (Microsoft Corporation)
  - PDC 05
- Randy Fernando and Cyril Zeller (NVIDIA Corporation)
  - I3D 2005

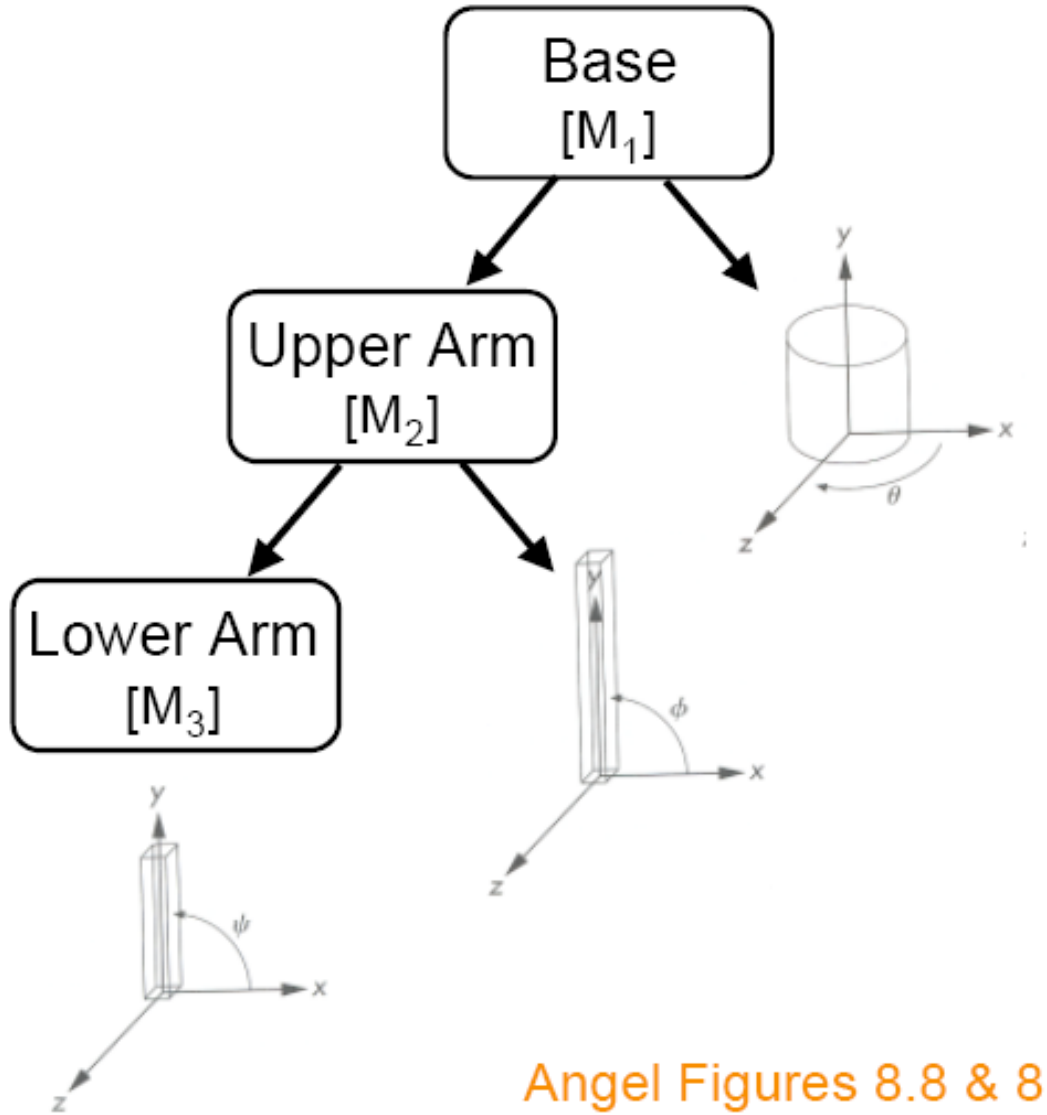
What Does GPU do?

- Let’s try to understand Where the GPU was born from...
  - Graphics Card / Video Card
- What does Graphics card do and Why did people develop it?
  - You need to understand Graphics (Rendering) Pipeline

Transform Example - Robot Arms

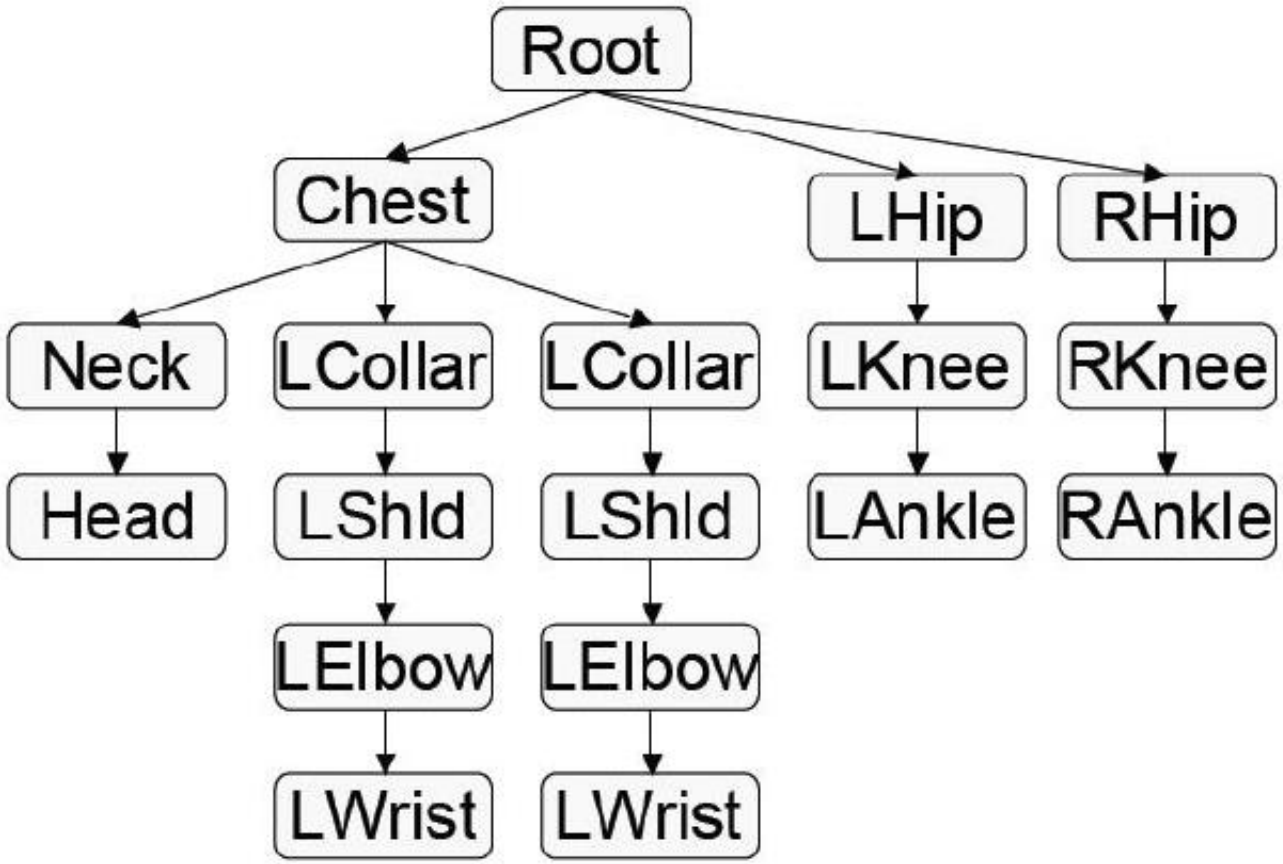


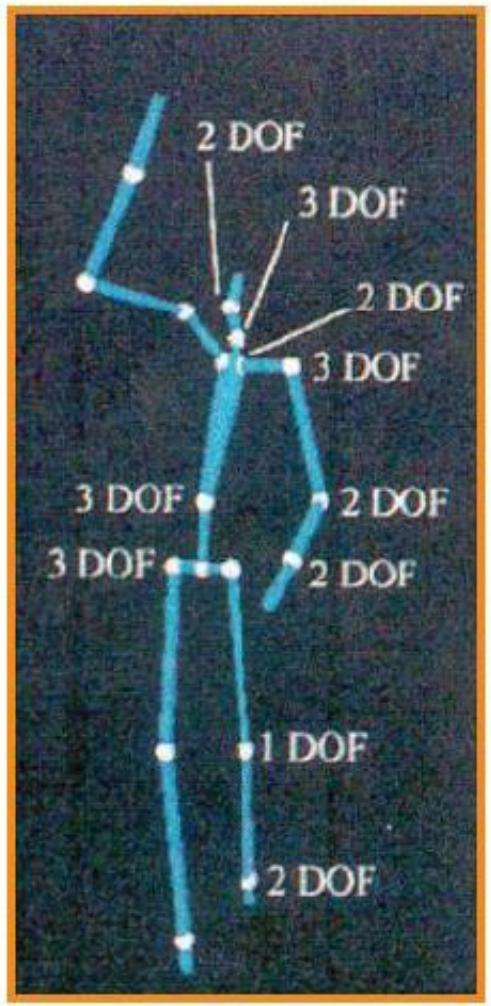
Robot Arm



Angel Figures 8.8 & 8.9

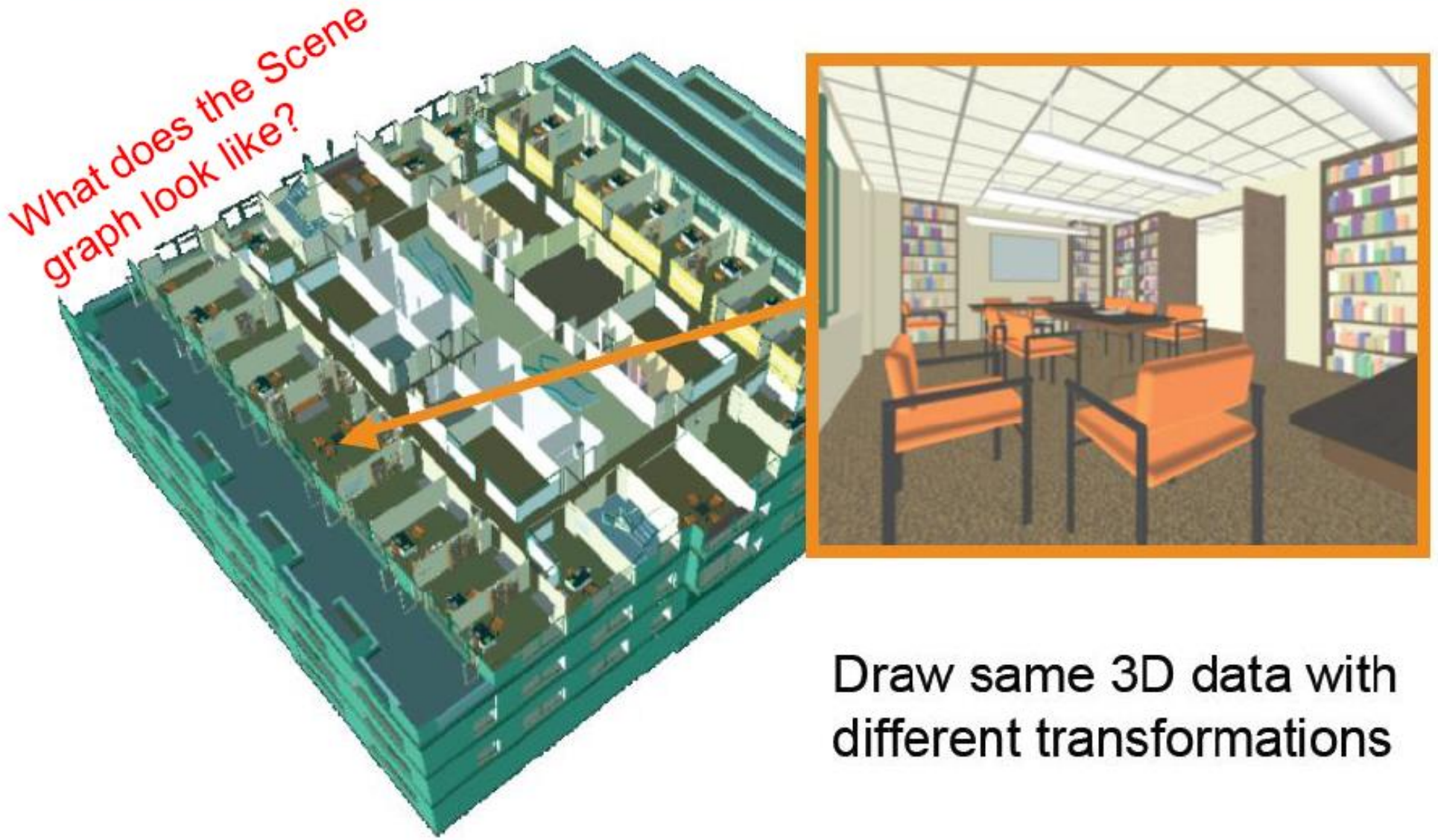
Example – Humanoid Character





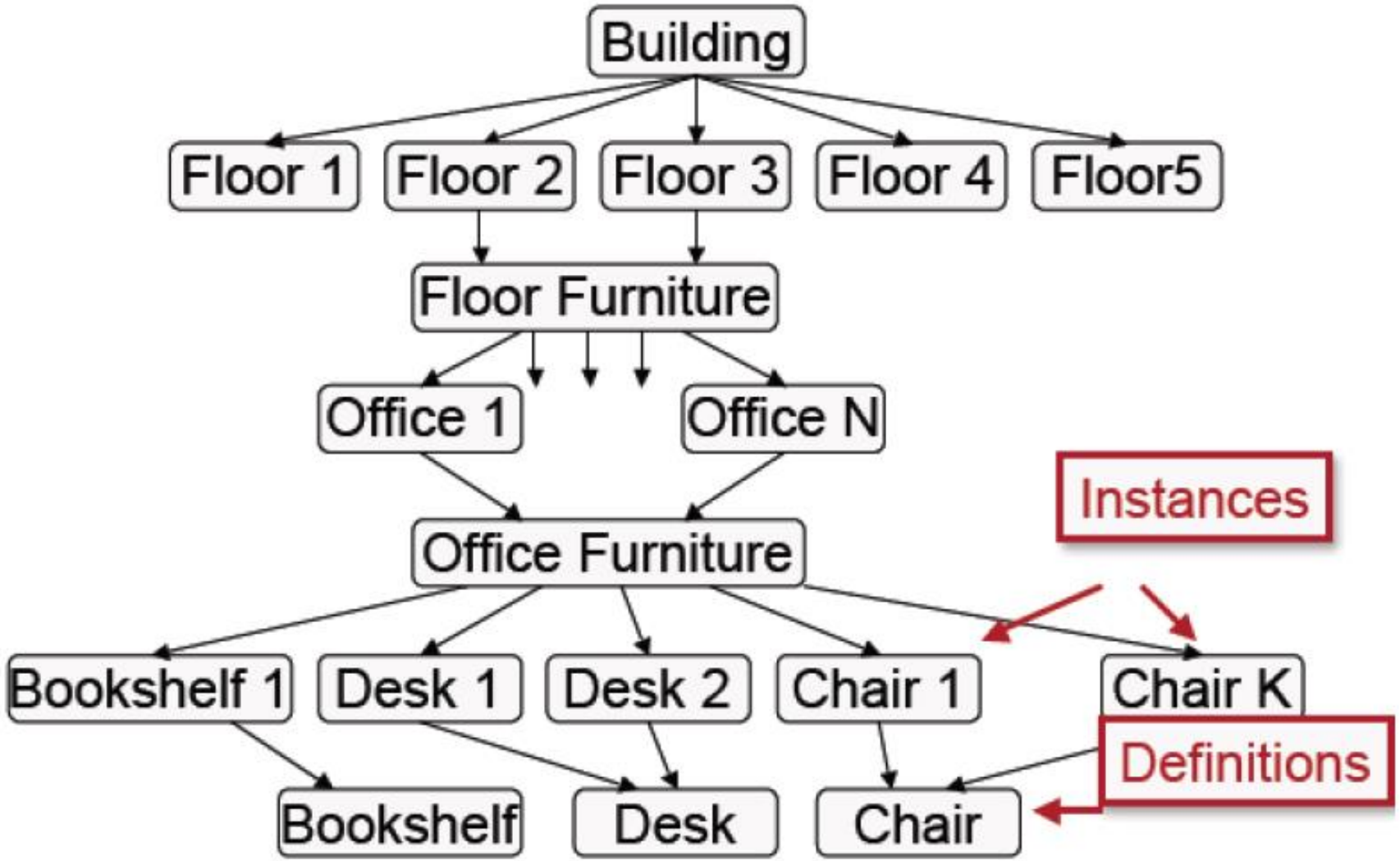
Rose et al. `96

Example – Complex Scenes

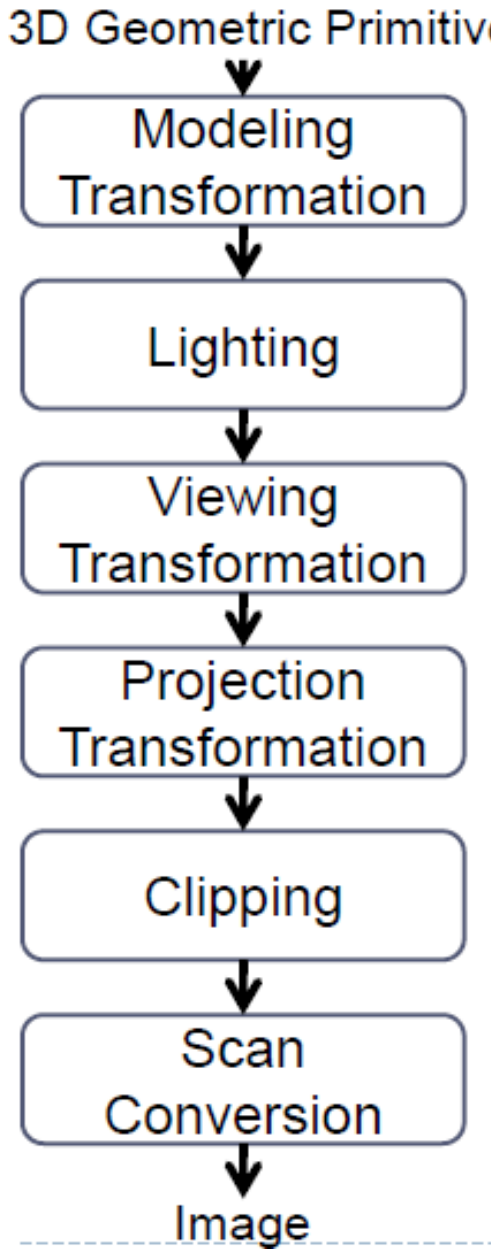


Draw same 3D data with different transformations

Example – Complex Scene Graph

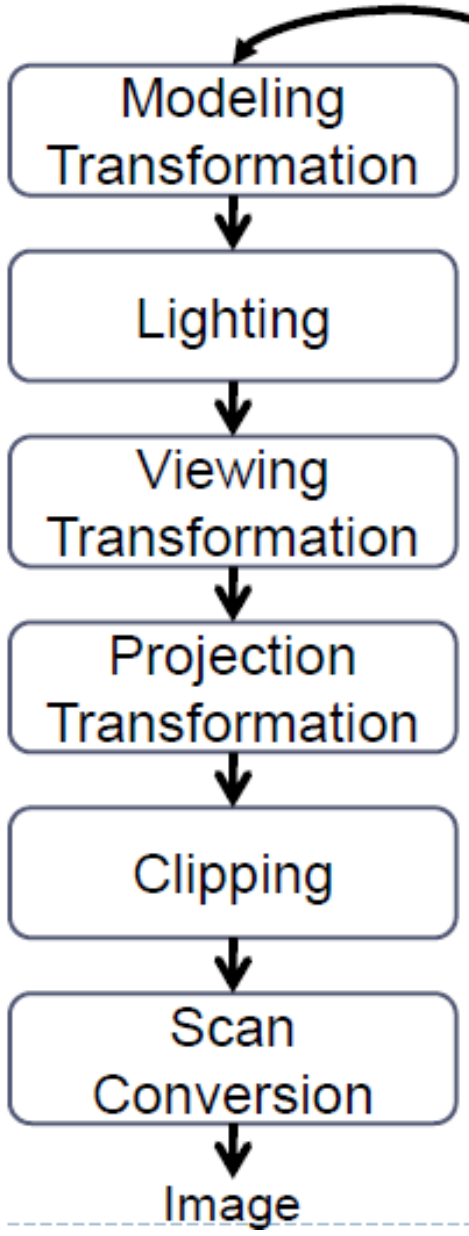


3D Rendering Pipeline (direct illumination only)



- A pipelined sequence of operations to draw a 3D primitive onto a 2D image

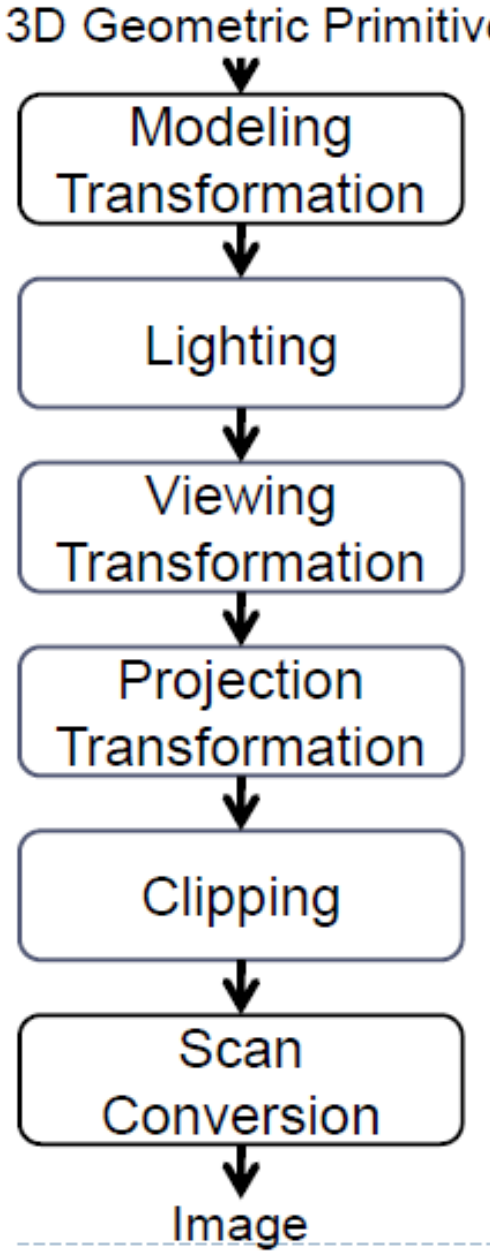
OpenGL Example




```
glBegin(GL_POLYGON);
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(1.0, 0.0, 0.0);
glVertex3f(1.0, 1.0, 1.0);
glVertex3f(0.0, 1.0, 1.0);
glEnd();
```

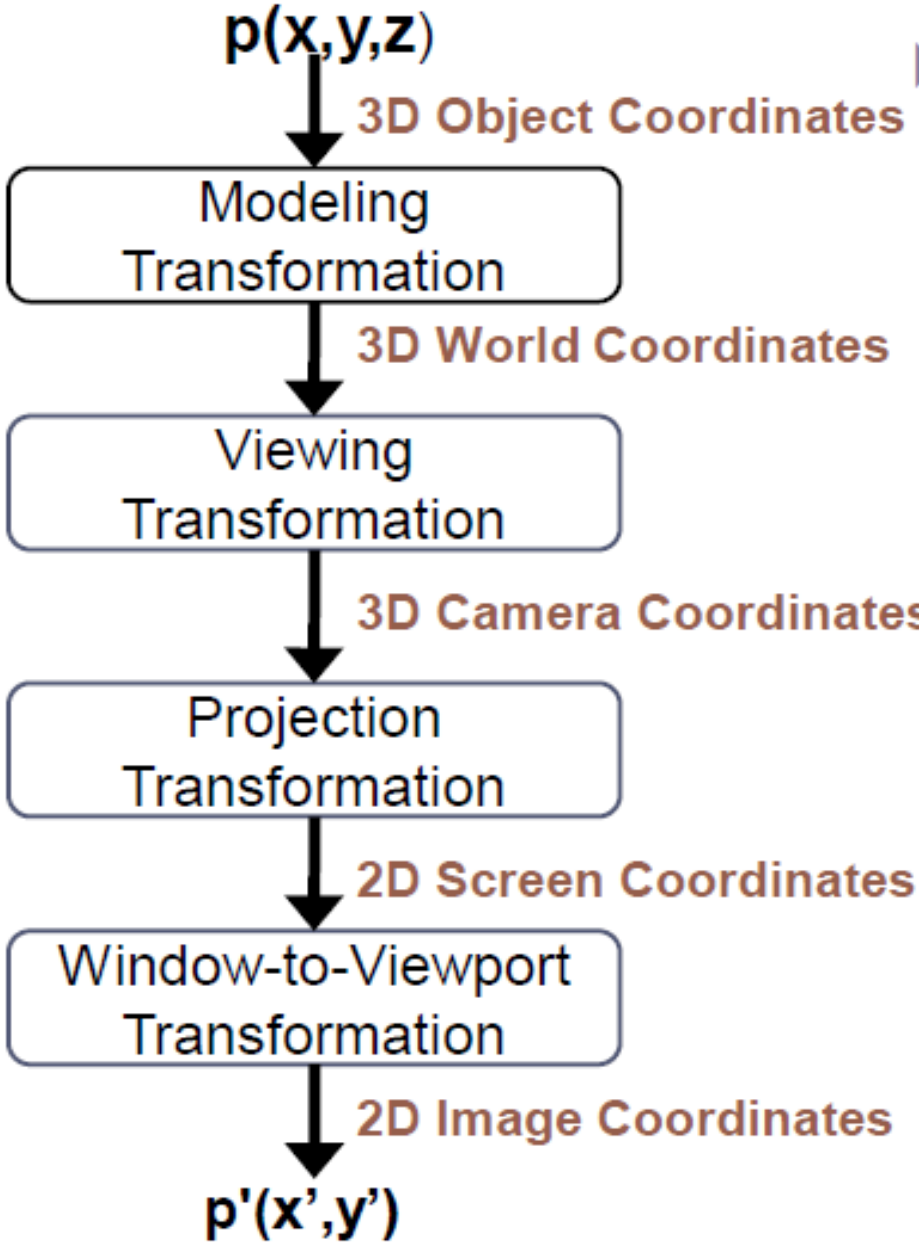
OpenGL runs the 3D rendering pipeline for each polygon

3D Rendering Pipeline (direct illumination only)

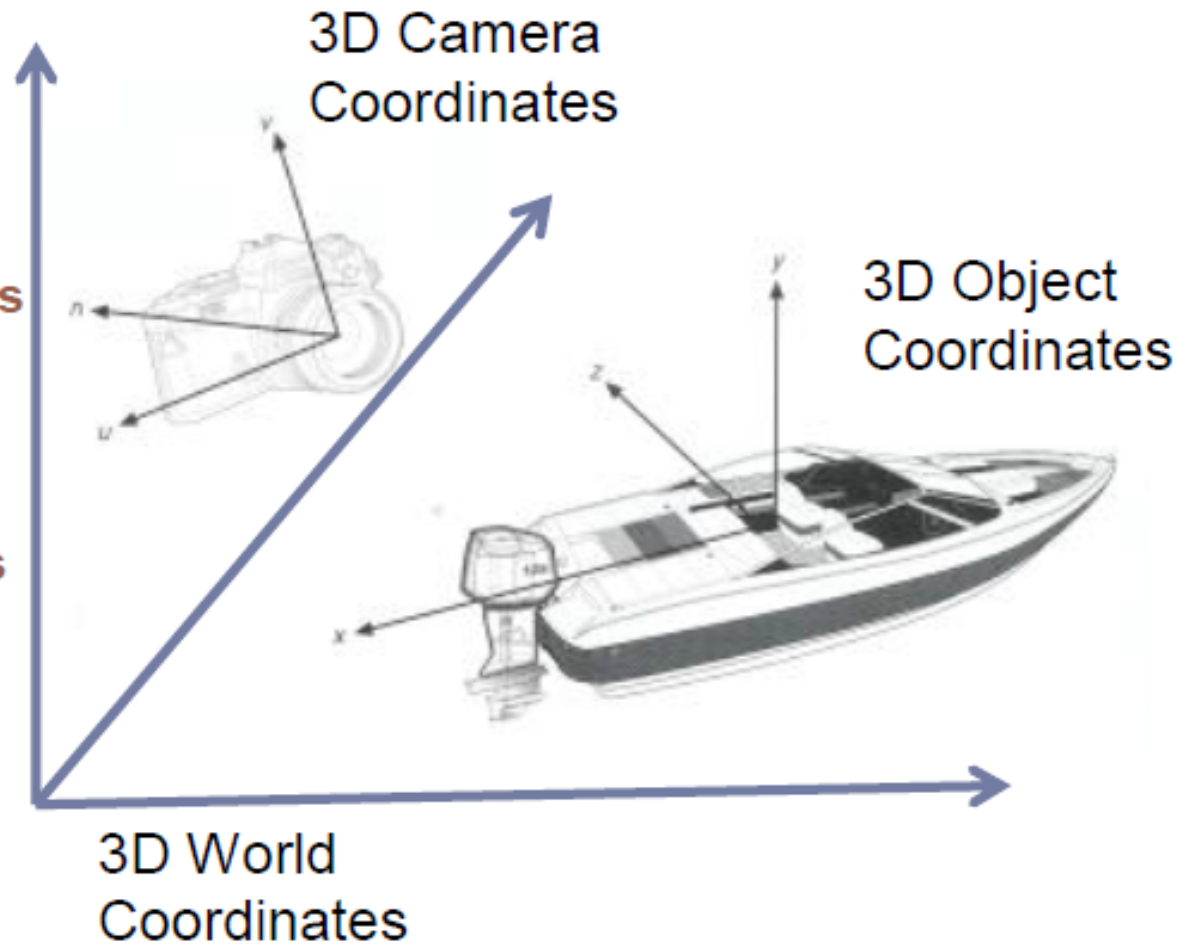


- Transform in 3D word coordinate system
- Illuminate according to lighting and reflectance
- Transform into 3D camera coordinate system
- Transform into 2D screen coordinate system
- Clip primitives outside camera's view
- Draw Pixels (also texturing, hidden surface, ...)

Transformations

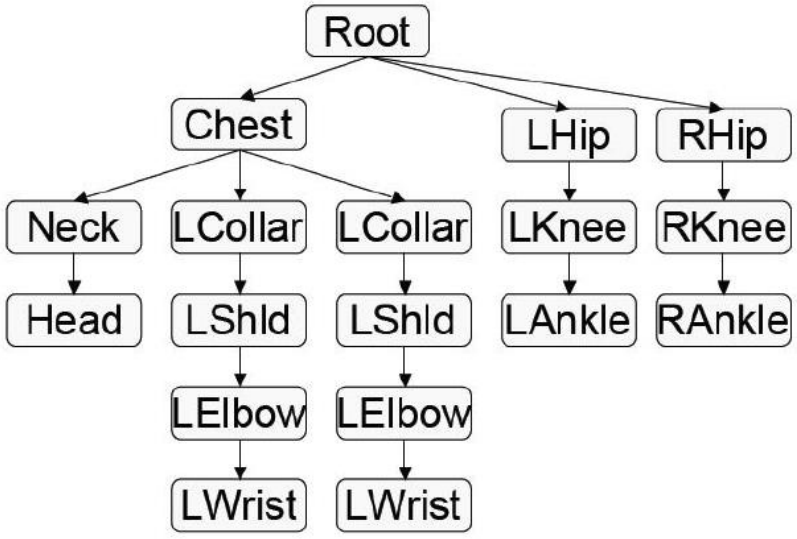


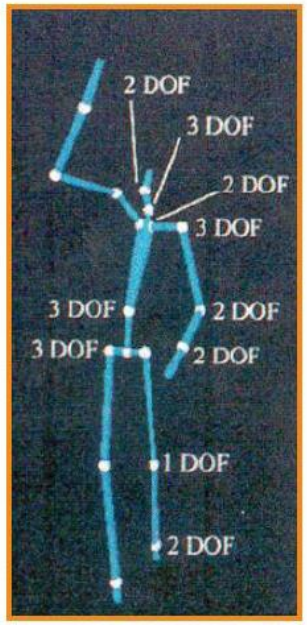
- Transformations map points from one coordinate system to another



Can you see the relation?

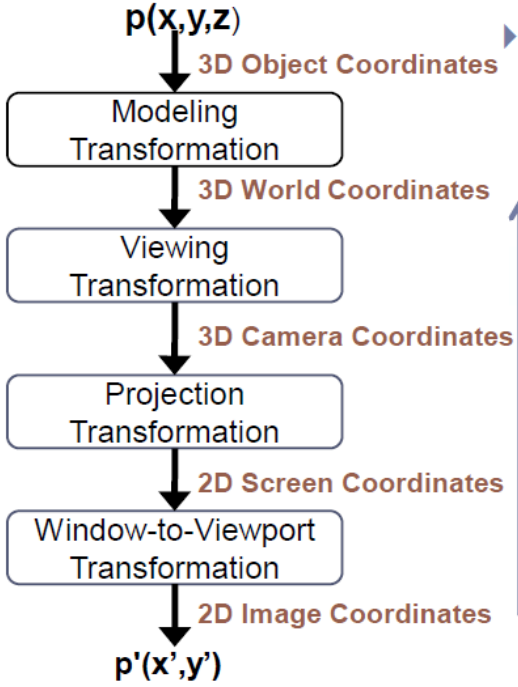
Example – Humanoid Character



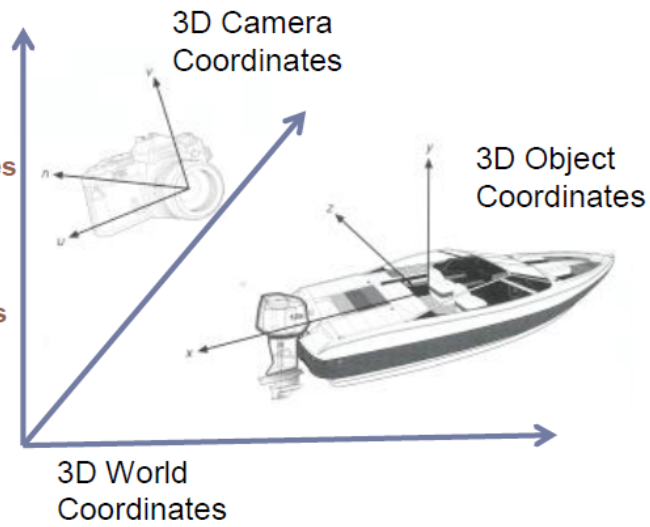


Rose et al. `96

Transformations



- Transformations map points from one coordinate system to another



Finding the Viewing Transformation

- We have the camera (in world coordinates)
- We want *T* taking objects from world to camera

$$P^c = T * P^w$$

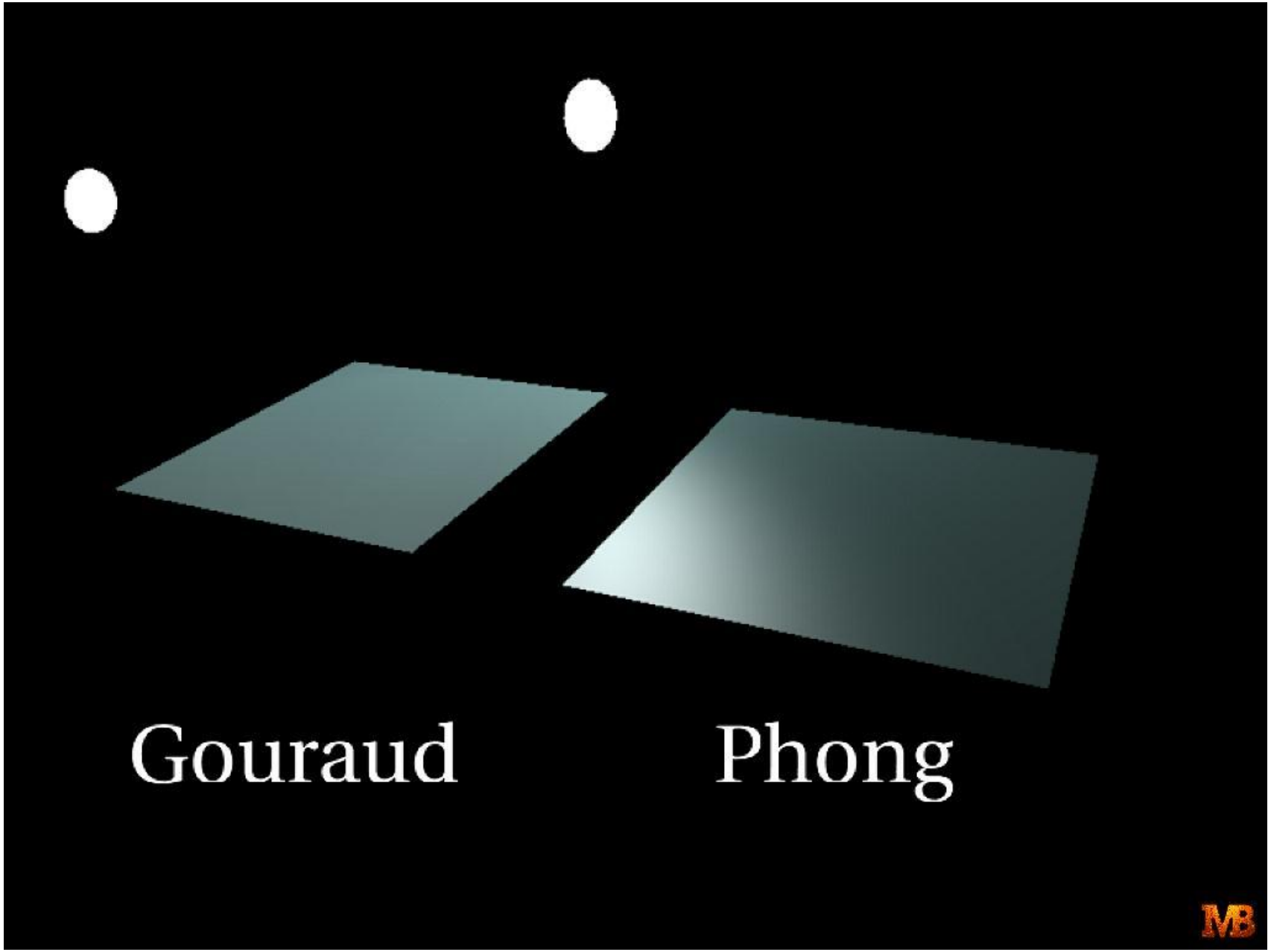
- Trick: Find T-1 taking objects from camera to world

$$P^w = T * P^c$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Power of Homogeneous Coordinate System

- So, where is the most computation intensive part?
- What did graphics card do about it?
- How was GPU evolved from Graphics Card?


Gouraud
Phong
MB

Interpolation

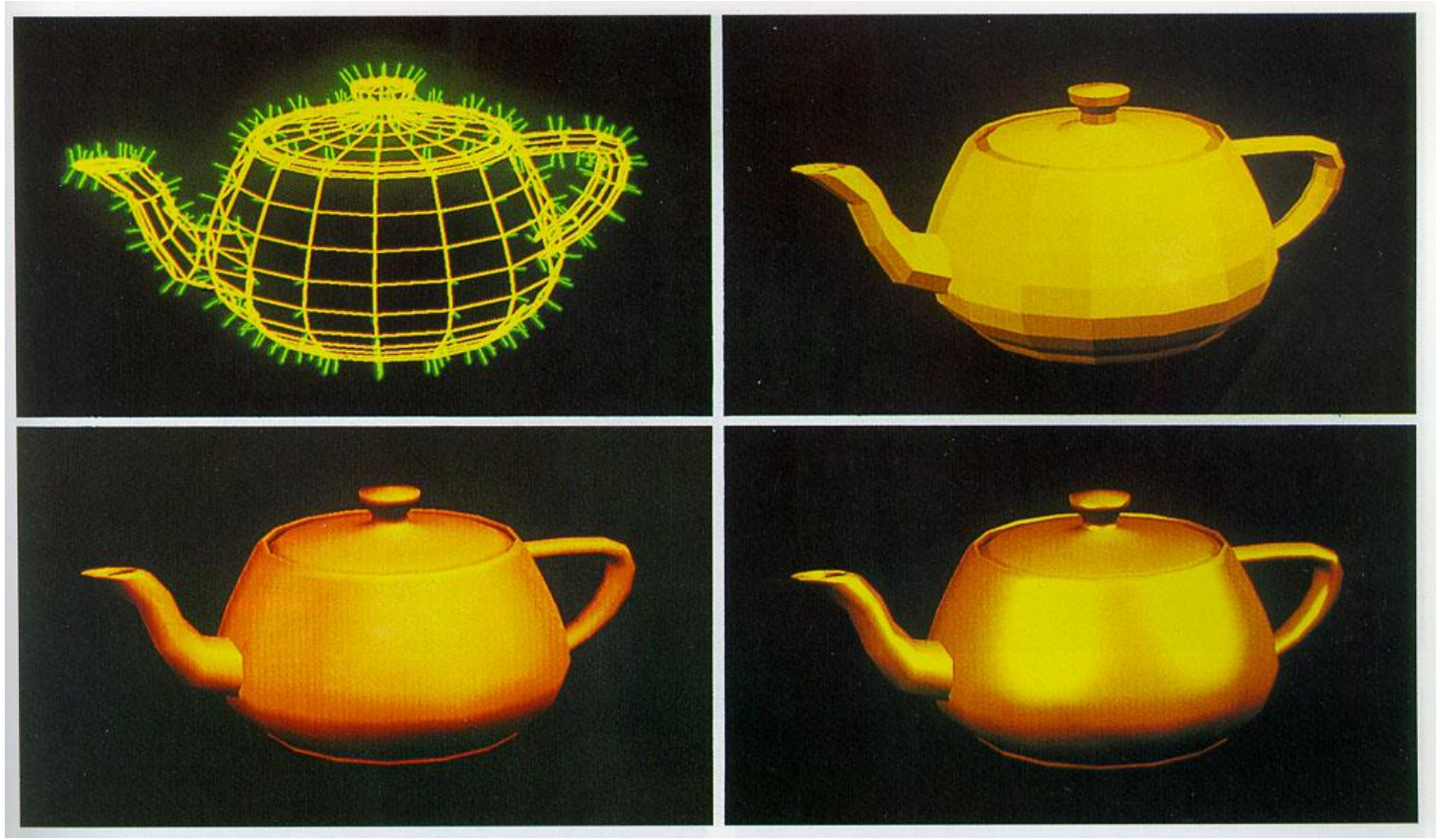
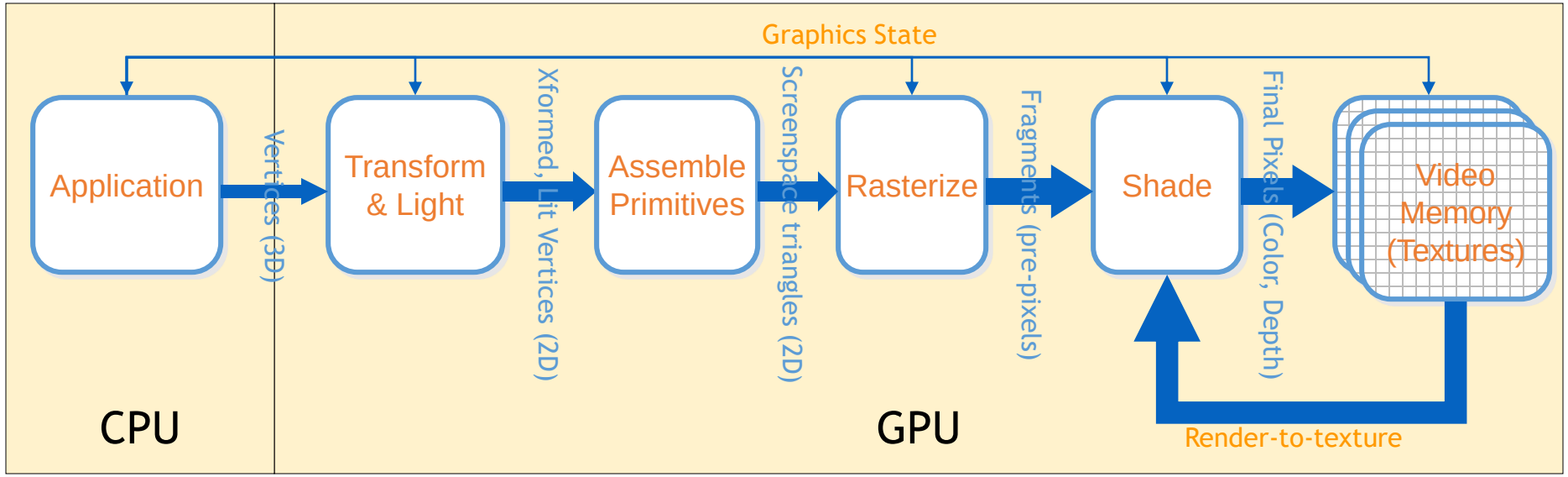


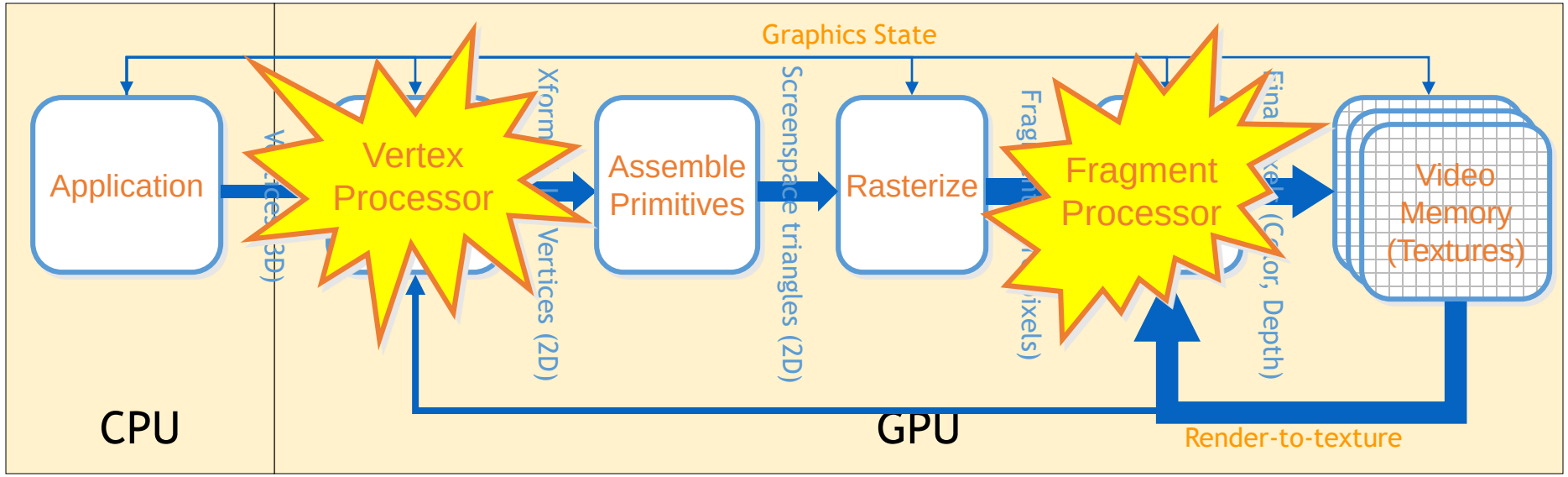
Image courtesy of Watt & Watt, Advanced Animation and Rendering Techniques

GPU Fundamentals: Graphics Pipeline



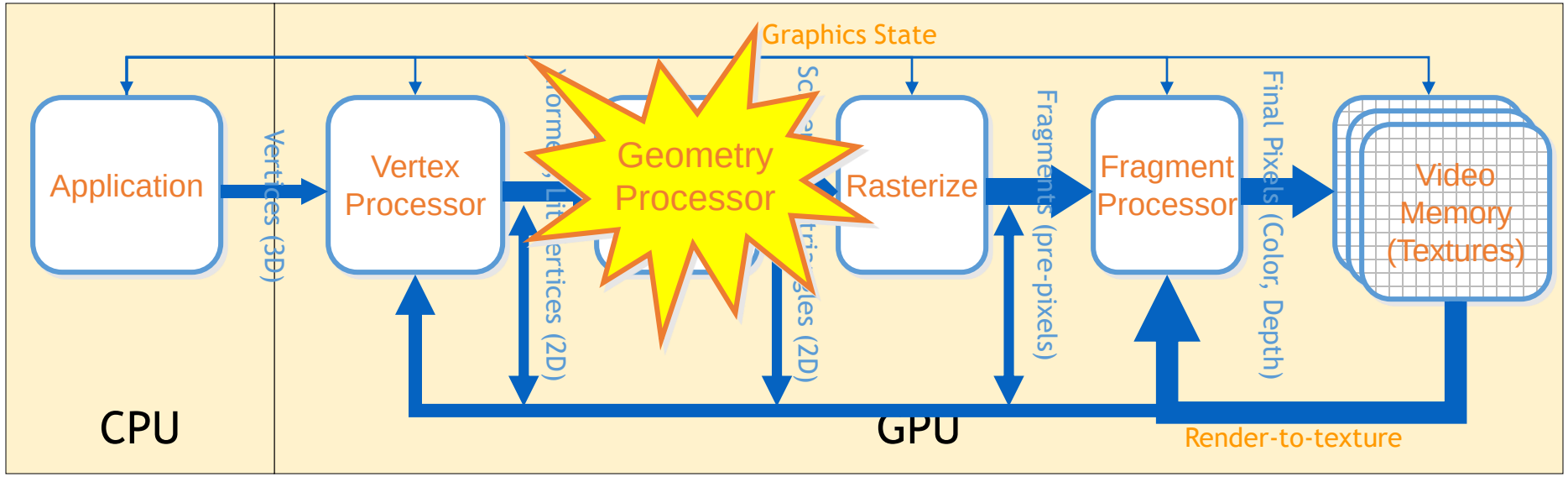
- A simplified graphics pipeline
  - Note that pipe widths vary
  - Many caches, FIFOs, and so on not shown

GPU Fundamentals: *Modern* Graphics Pipeline



- Programmable vertex processor!
- Programmable pixel processor!

GPU Fundamentals: *Modern* Graphics Pipeline



- Programmable primitive assembly!
- More flexible memory access!

GPU vs CPU

- A GPU is tailored for highly parallel operation while a CPU executes programs serially
- For this reason, GPUs have many parallel execution units and higher transistor counts, while CPUs have few execution units and higher clockspeeds
- A GPU is for the most part deterministic in its operation (though this is quickly changing)
- GPUs have much deeper pipelines (several thousand stages vs 10-20 for CPUs)
- GPUs have significantly faster and more advanced memory interfaces as they need to shift around a lot more data than CPUs

The GPU pipeline

- The GPU receives geometry information from the CPU as an input and provides a picture as an output
- Let’s see how that happens

Host Interface

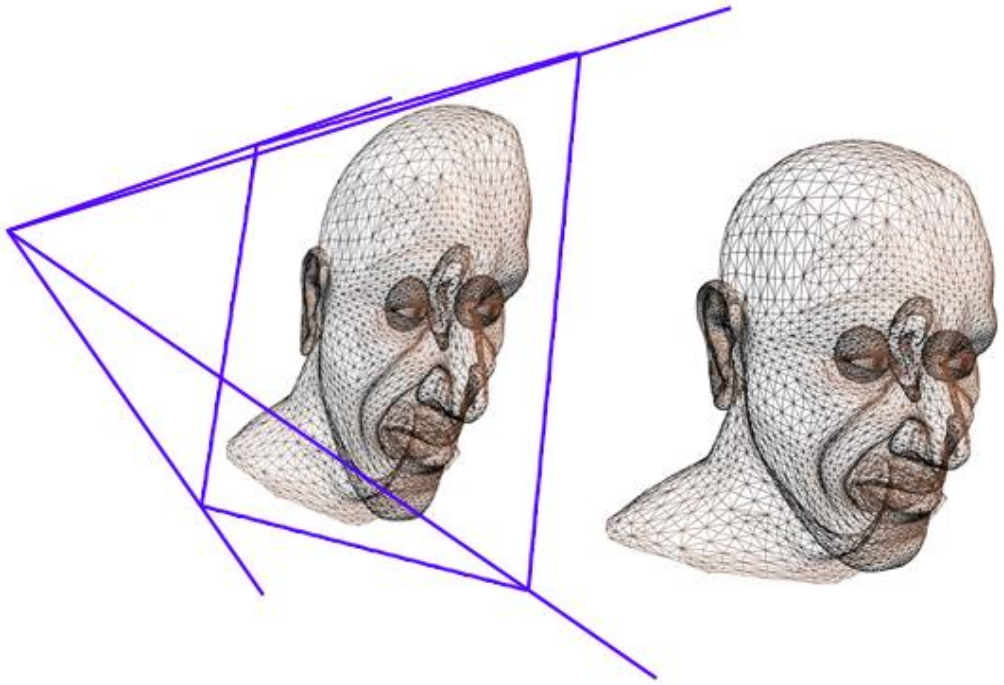
- The host interface is the communication bridge between the CPU and the GPU
- It receives commands from the CPU and also pulls geometry information from system memory
- It outputs a *stream* of vertices in object space with all their associated information (normals, texture coordinates, per vertex color etc)

Vertex Processing

- The vertex processing stage receives vertices from the host interface in object space and outputs them in screen space
- This may be a simple linear transformation, or a complex operation involving morphing effects
- Normals, texcoords etc are also transformed
- No new vertices are created in this stage, and no vertices are discarded (input/output has 1:1 mapping)

GPU Pipeline: Transform

- Vertex processor (multiple in parallel)
  - Transform from “world space” to “image space”
  - Compute per-vertex lighting



Example

http://www.lighthouse3d.com/opengl/glsl/

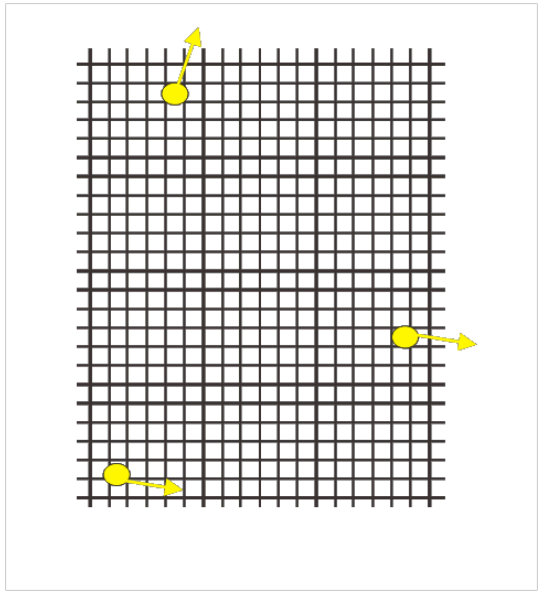
- Vertex shader

```
void main()
{
	vec4 v = vec4(gl_Vertex);
	v.z = 0.0;
	gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

- Pixel shader

```
void main()
{
	gl_FragColor = vec4(0.8,0.4,0.4,1.0);
}
```

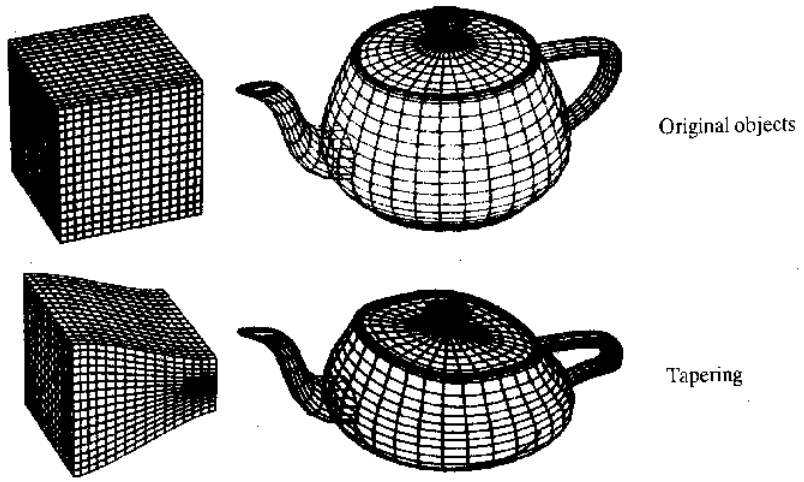
Vertex Shader



- One element in / one out
- No communication
- Can select fragment address
- Input:
  - Vertex data (position, normal, color, …)
  - Shader constants, Texture data
- Output:
  - Required: Transformed clip-space position
  - Optional: Colors, texture coordinates, normals (data you want passed on to the pixel shader)
- Restrictions:
  - Can’t create new vertices

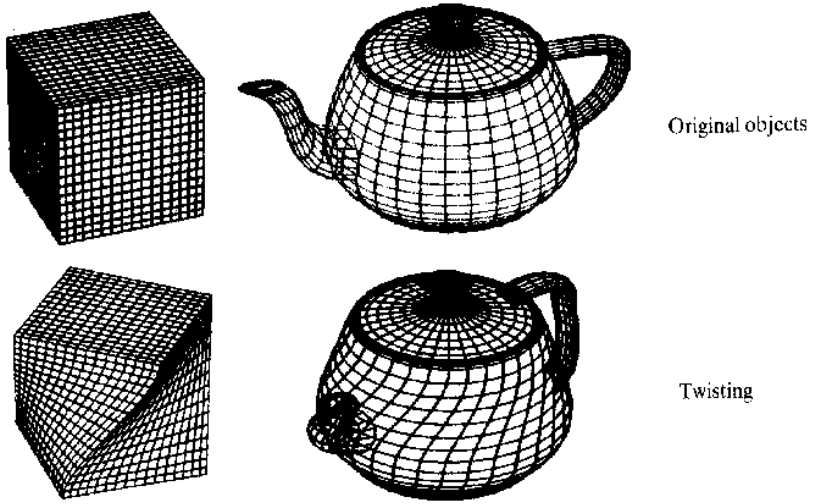
Tapering

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & f(x) & 0 & 0 \\ 0 & 0 & f(x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



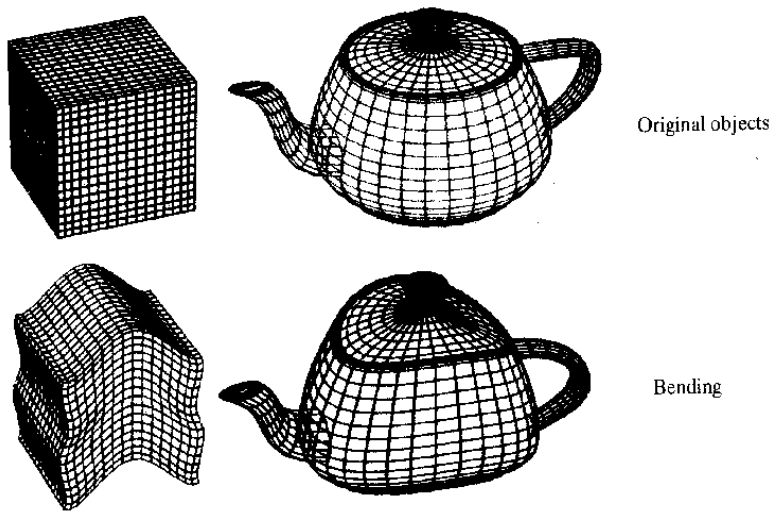
Twisting

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta(y)) & 0 & \sin(\theta(y)) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta(y)) & 0 & \cos(\theta(y)) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



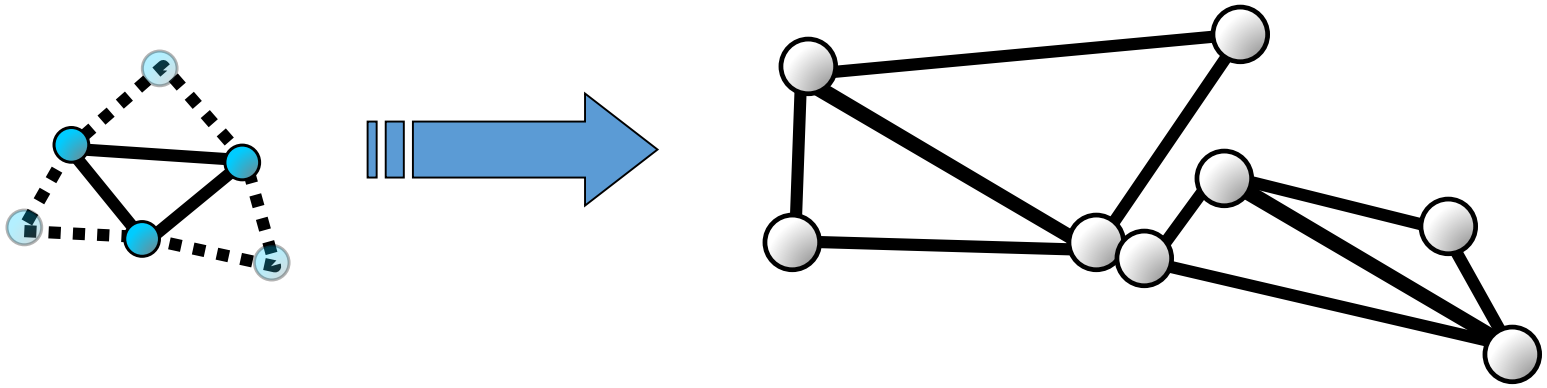
Bending

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & f(y) & g(y) & 0 \\ 0 & h(y) & k(y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



GPU Pipeline: Assemble Primitives

- Geometry processor
  - How the vertices connect to form a primitive
  - Per-Primitive Operations

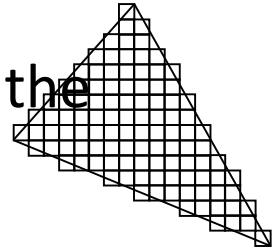


Triangle setup

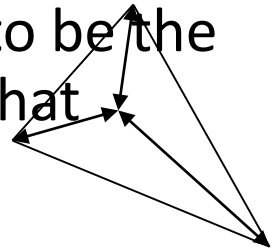
- In this stage geometry information becomes raster information (screen space geometry is the input, pixels are the output)
- Prior to rasterization, triangles that are backfacing or are located outside the viewing frustrum are rejected
- Some GPUs also do some hidden surface removal at this stage

Triangle Setup (cont)

- A fragment is generated if and only if its center is inside the triangle

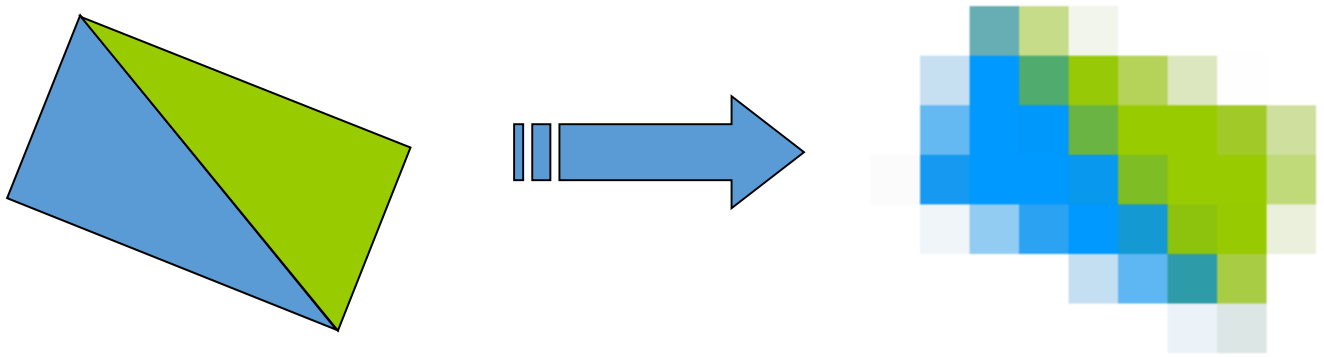


- Every fragment generated has its attributes computed to be the perspective correct interpolation of the three vertices that make up the triangle



GPU Pipeline: Rasterize

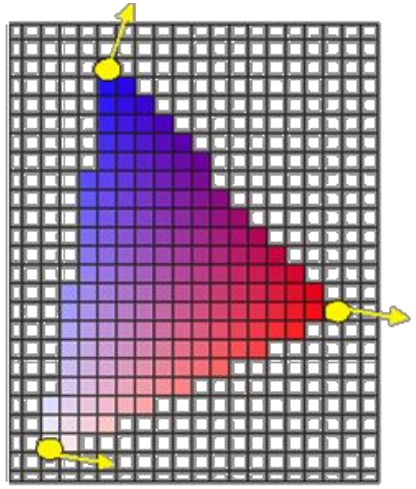
- Rasterizer
  - Convert geometric rep. (vertex) to image rep. (fragment)
    - Pixel + associated data: color, depth, stencil, etc.
  - Interpolate per-vertex quantities across pixels



Fragment Processing

- Each fragment provided by triangle setup is fed into fragment processing as a set of attributes (position, normal, texcoord etc), which are used to compute the final color for this pixel
- The computations taking place here include texture mapping and math operations
- Typically the bottleneck in modern applications

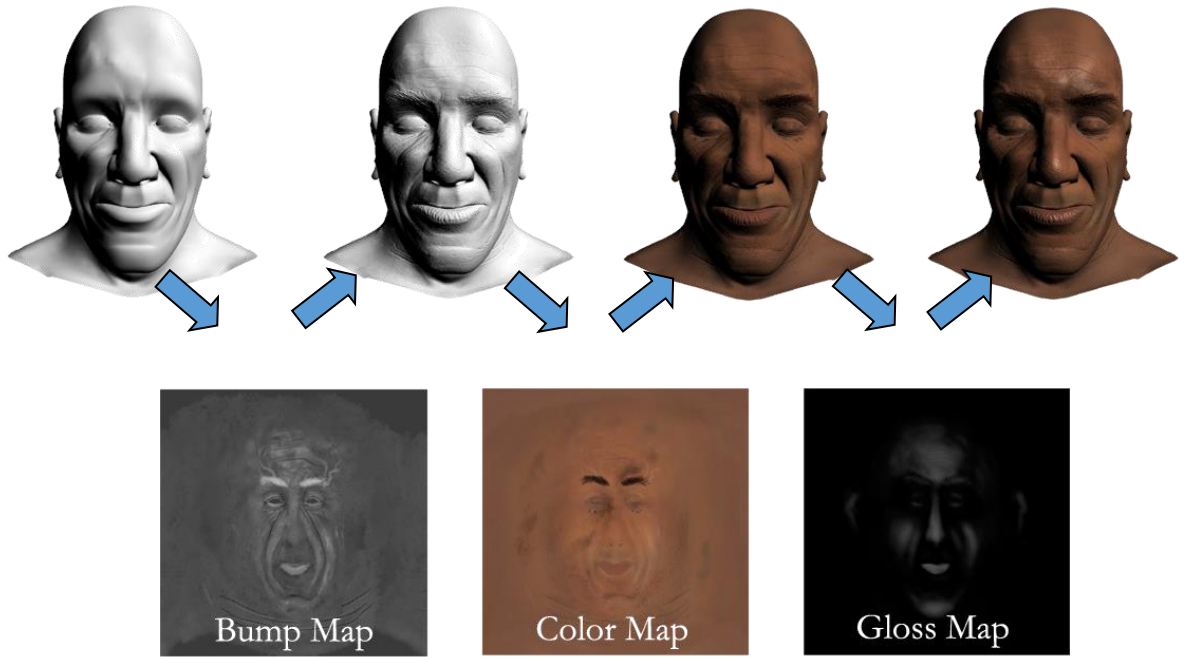
Pixel Shader



- Biggest computational resource
- One element in / 0 – 1 out
- Cannot change destination address
- No communication
- Input:
  - Interpolated data from vertex shader
  - Shader constants, Texture data
- Output:
  - Required: Pixel color (with alpha)
  - Optional: Can write additional colors to multiple render targets
- Restrictions:
  - Can’t read and write the same texture simultaneously

GPU Pipeline: Shade

- Fragment processors (multiple in parallel)
  - Compute a color for each pixel
  - Optionally read colors from textures (images)



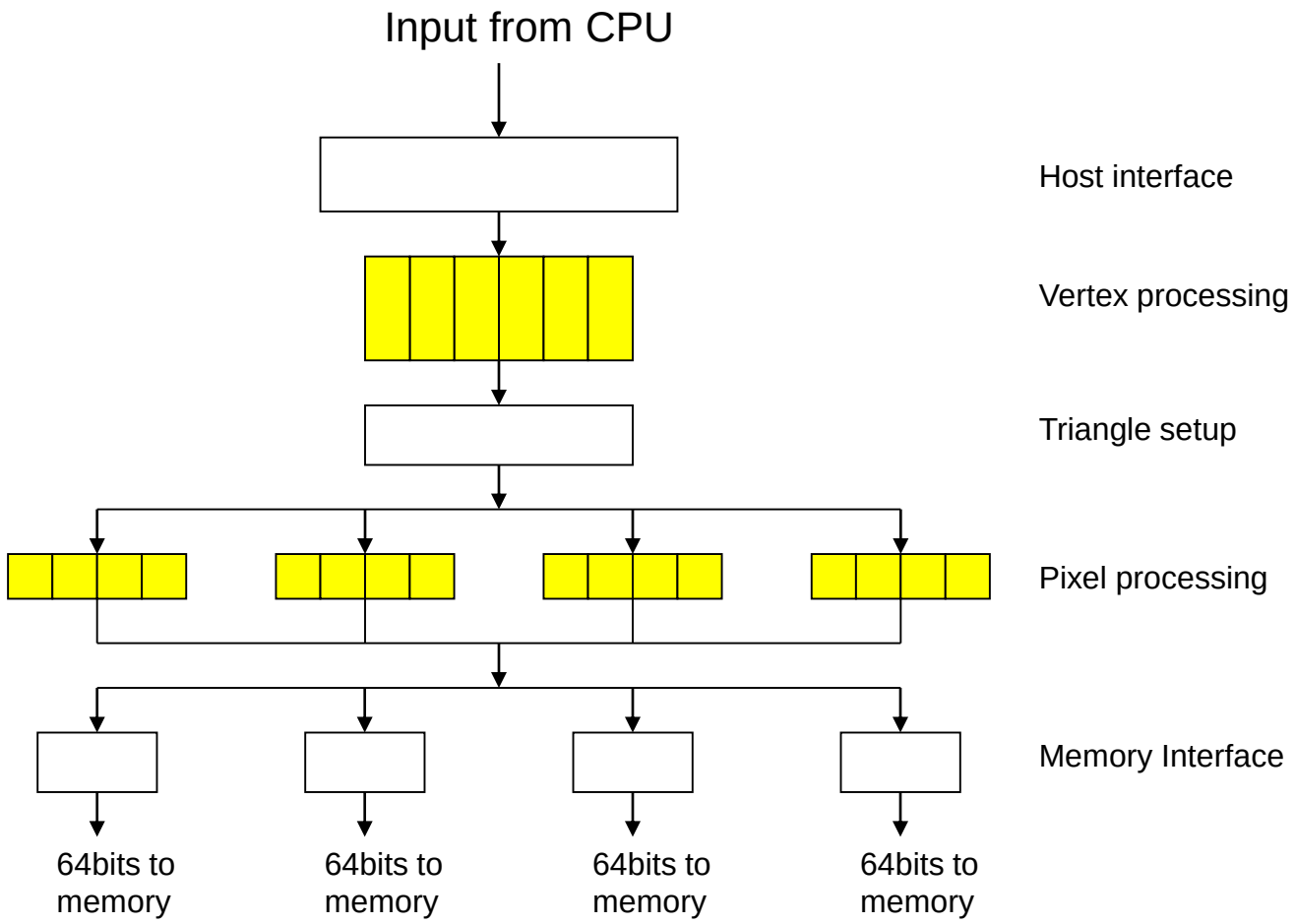
Memory Interface

- Fragment colors provided by the previous stage are written to the framebuffer
- Used to be the biggest bottleneck before fragment processing took over
- Before the final write occurs, some fragments are rejected by the zbuffer, stencil and alpha tests
- On modern GPUs, z and color are compressed to reduce framebuffer bandwidth (but not size)

Programmability in the GPU

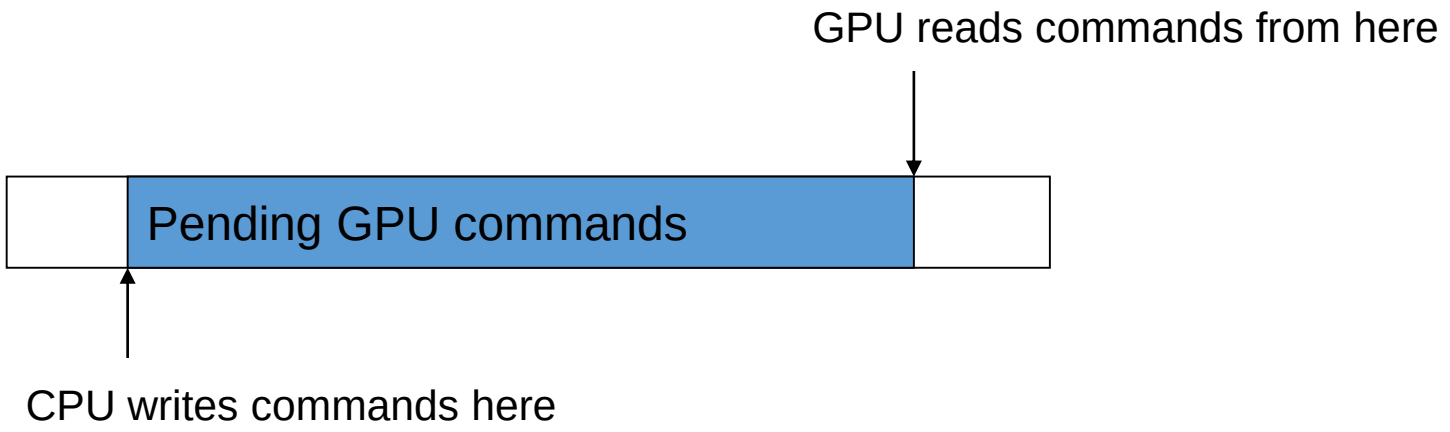
- Vertex and fragment processing, and now triangle set-up, are programmable
- The programmer can write programs that are executed for every vertex as well as for every fragment
- This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of older 3D applications

Diagram of a modern GPU



CPU/GPU interaction

- The CPU and GPU inside the PC work in parallel with each other
- There are two “threads” going on, one for the CPU and one for the GPU, which communicate through a command buffer:

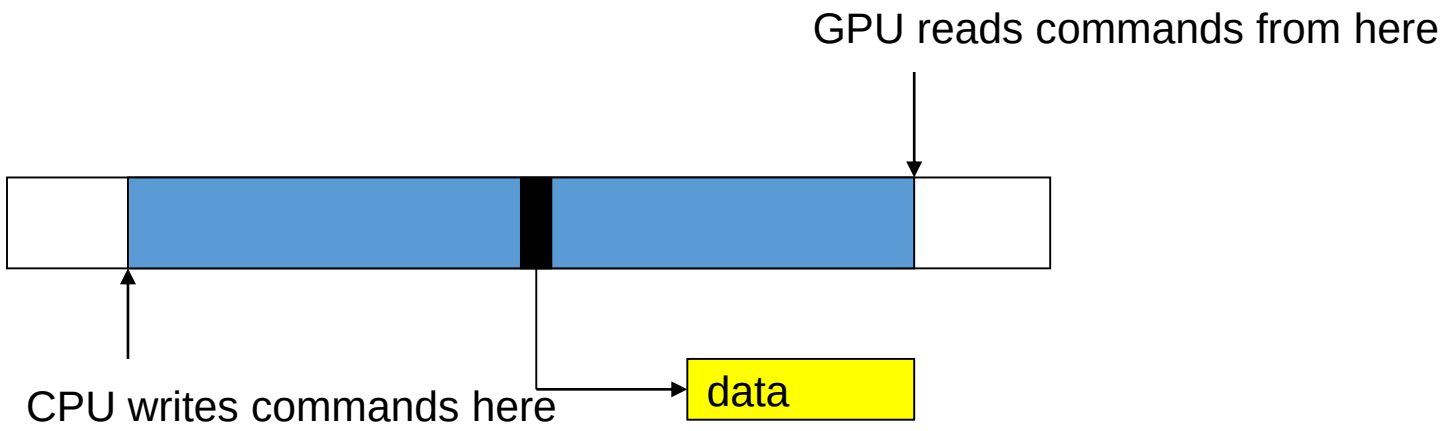


CPU/GPU interaction (cont)

- If this command buffer is drained empty, we are CPU limited and the GPU will spin around waiting for new input. All the GPU power in the universe isn’t going to make your application faster!
- If the command buffer fills up, the CPU will spin around waiting for the GPU to consume it, and we are effectively GPU limited

Synchronization issues

- This leads to a number of synchronization considerations
- In the figure below, the CPU must not overwrite the data in the “yellow” block until the GPU is done with the “black” command, which references that data:



Some more GPU tips

- Since the GPU is highly parallel and deeply pipelined, try to dispatch large batches with each drawing call
- Sending just one triangle at a time will not occupy all of the GPU's several vertex/pixel processors, nor will it fill its deep pipelines
- Since all GPUs today use the zbuffer algorithm to do hidden surface removal, rendering objects front-to-back is faster than back-to-front (painters algorithm), or random ordering
- Of course, there is no point in front-to-back sorting if you are already CPU limited

Computational Power

- *Why are GPUs getting faster so fast?*
  - Arithmetic intensity
    - the specialized nature of GPUs makes it easier to use additional transistors for computation
  - Economics
    - multi-billion dollar video game market is a pressure cooker that drives innovation to exploit this property

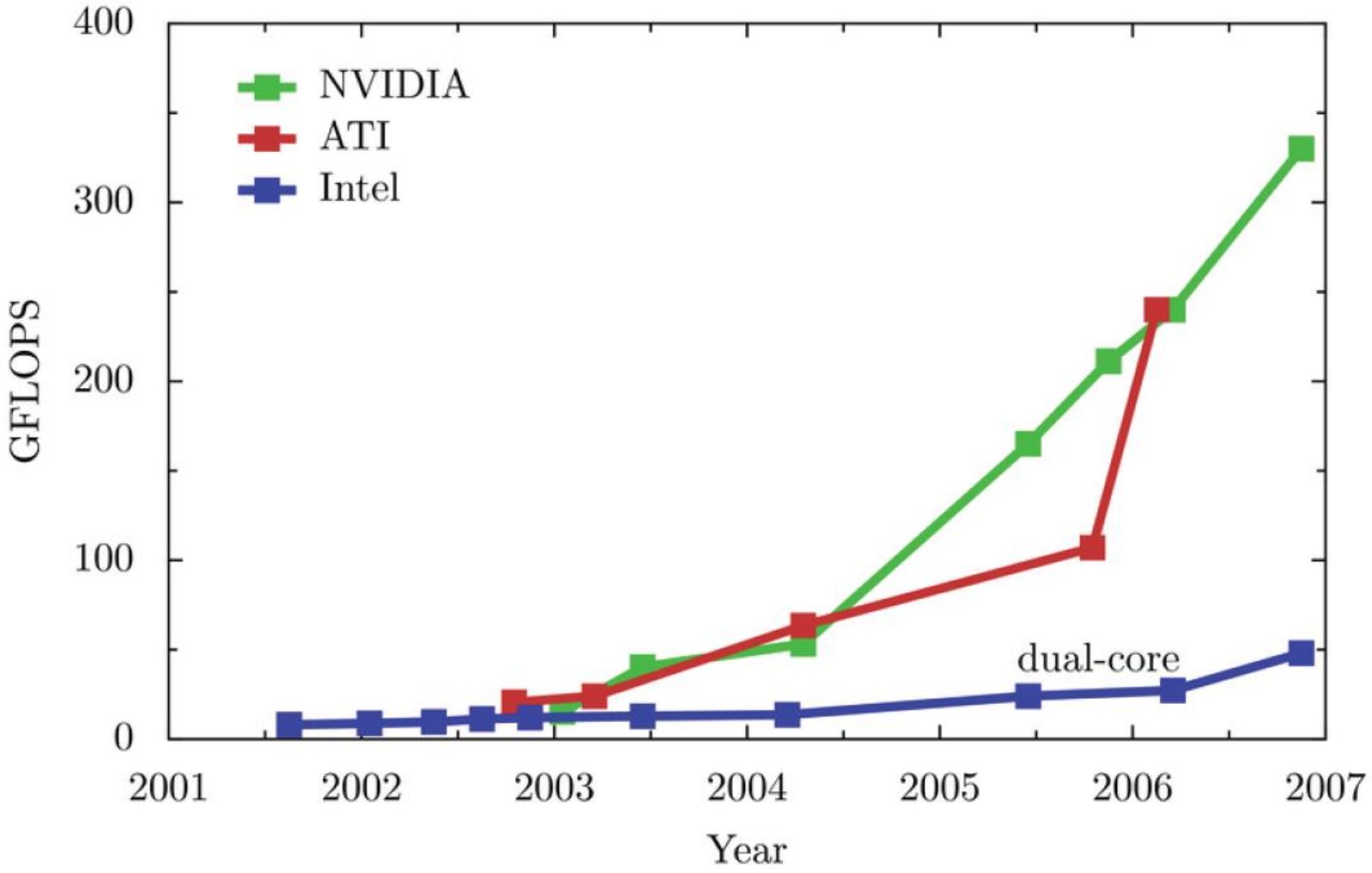
Modern GPU has more ALU's

Figure 1-2. The GPU Devotes More Transistors to Data Processing

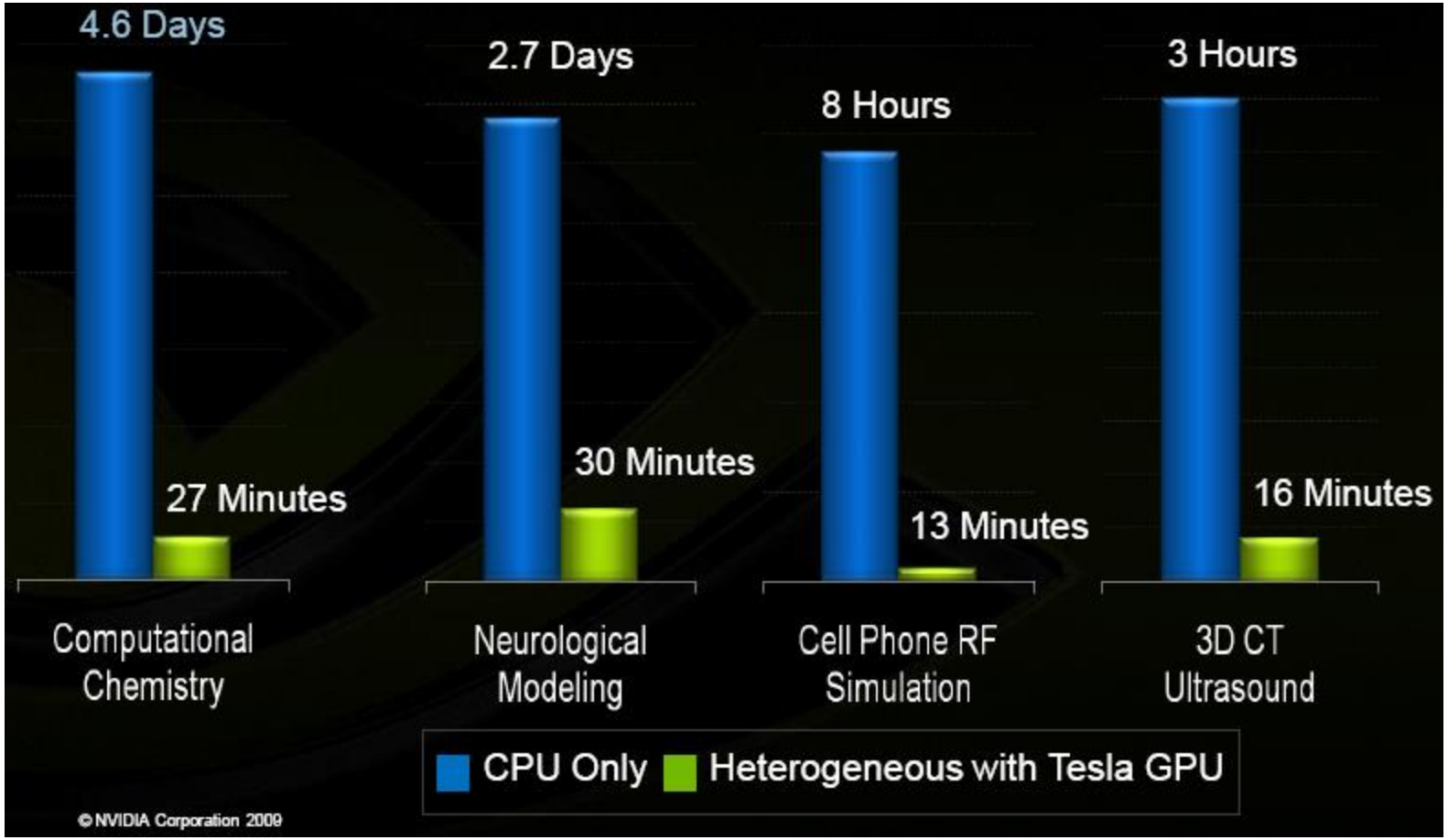
Computational Power

- GPUs are fast…
  - 3.0 GHz Intel Core2 Duo (Woodcrest Xeon 5160):
    - Computation: 48 GFLOPS peak
    - Memory bandwidth: 21 GB/s peak
    - Price: $874 (chip)
  - NVIDIA GeForce 8800 GTX:
    - Computation: 330 GFLOPS observed
    - • Memory bandwidth: 55.2 GB/s observed
    - • Price: $599 (board)
- GPUs are getting faster, faster
  - CPUs: 1.4× annual growth
  - GPUs: 1.7×(pixels) to 2.3× (vertices) annual growth

Computational Power



CPU v/s GPU



Flexible and Precise

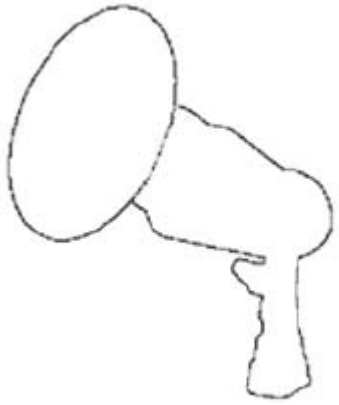
- Modern GPUs are deeply programmable
  - Programmable pixel, vertex, and geometry engines
  - Solid high-level language support
- Modern GPUs support “real” precision
  - 32 bit floating point throughout the pipeline
    - High enough for many (not all) applications
    - Vendors committed to double precision soon
  - DX10-class GPUs add 32-bit integers

Example

Depth buffer

Normal buffer

Final result



Silhouettes

Creases

Image Space Silhouette Extraction Using Graphics Hardware [Wang 2005]

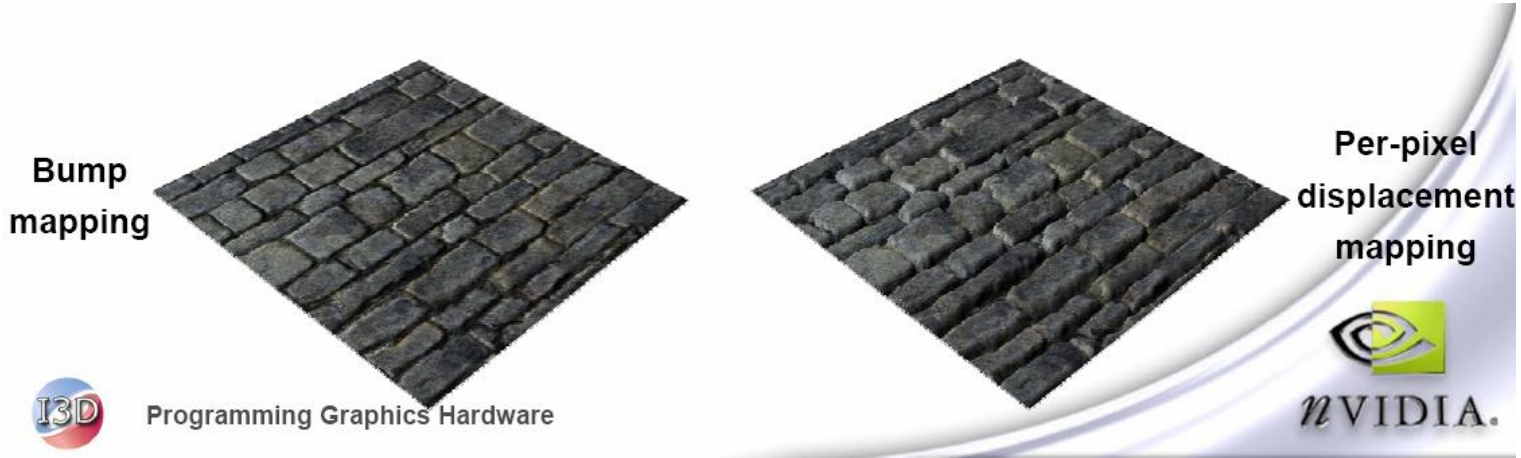
GPU Applications

- Bump/Displacement mapping

Diffuse light without bump

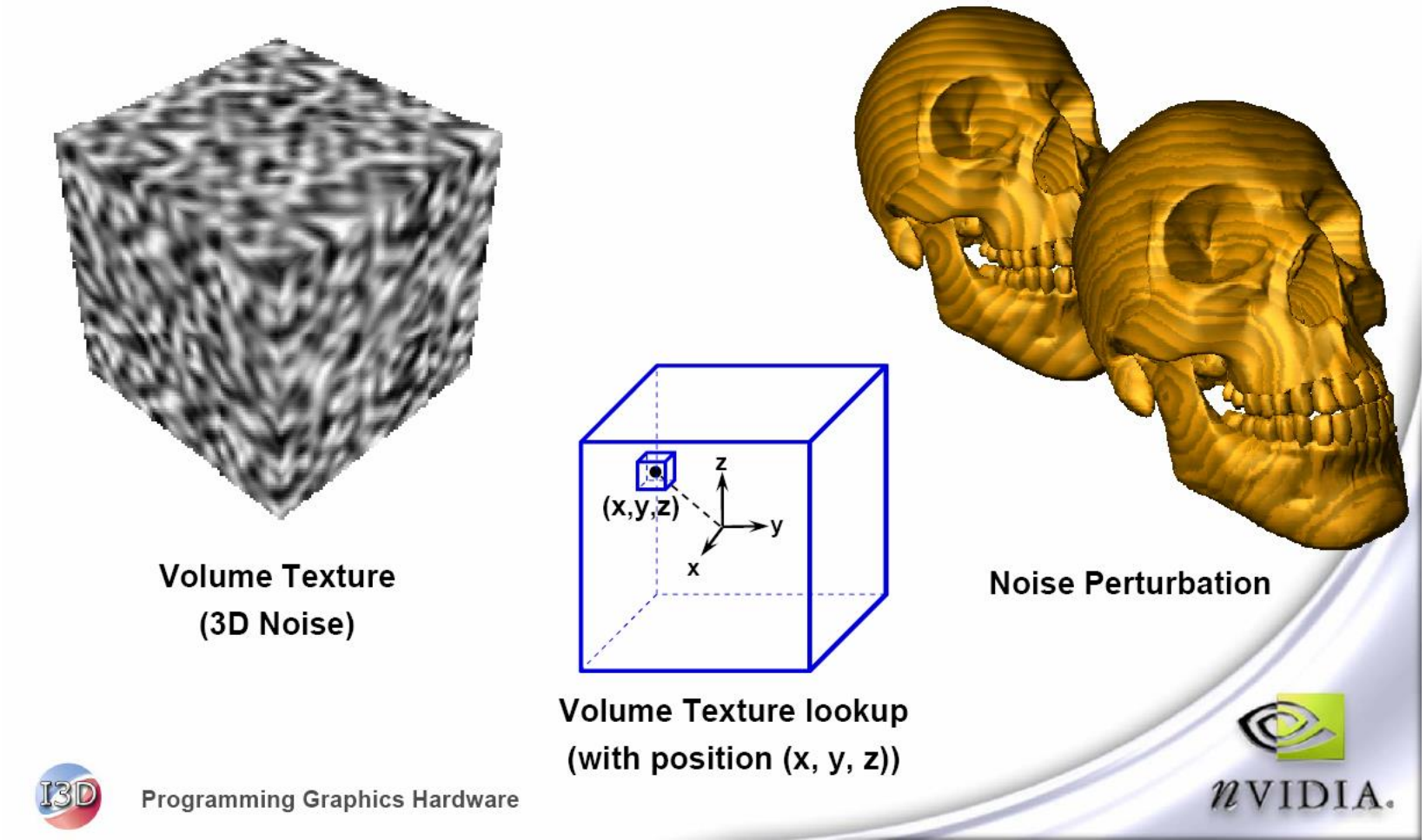
Height map

Diffuse light with bump



GPU Applications

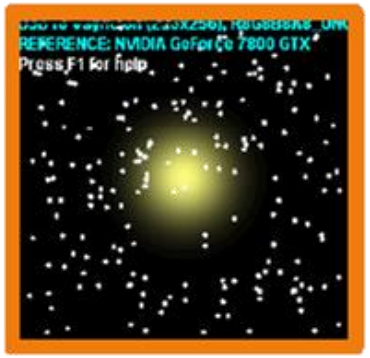
- Volume texture mapping



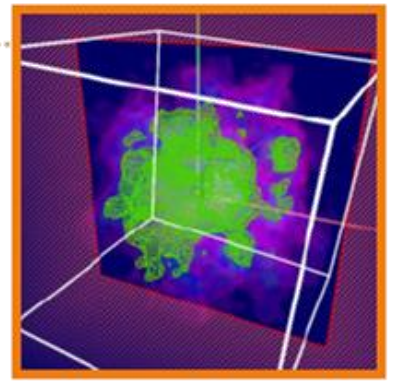
GPU Applications

Deep Waves

Sparkling Sprites

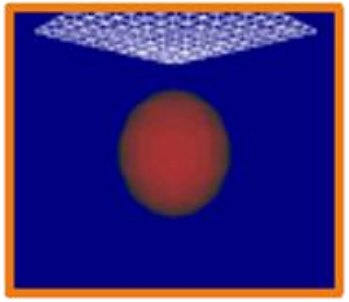


GPU Fluid Simulation



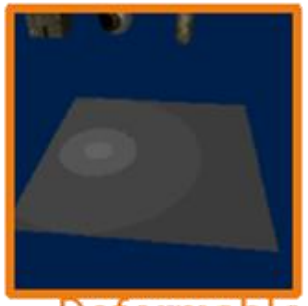
GPU Marching Cubes

Table-free Noise



GPU Cloth

Styled Line Drawing



Deformable Collisions

SIGGRAPH2006

GPU Applications

- Soft Shadows

Percentage-closer soft shadows [Fernando 2005]

GPGPU

- How?
  - CUDA
  - OpenCL
  - DirectCompute

What is CUDA?

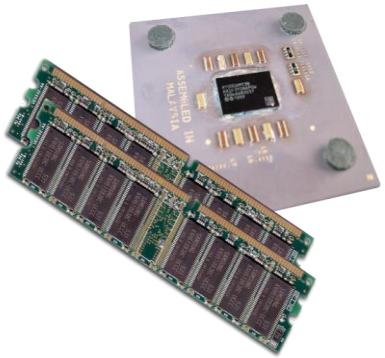
- ‘Compute Unified Device Architecture’

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance

- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

Heterogeneous Computing

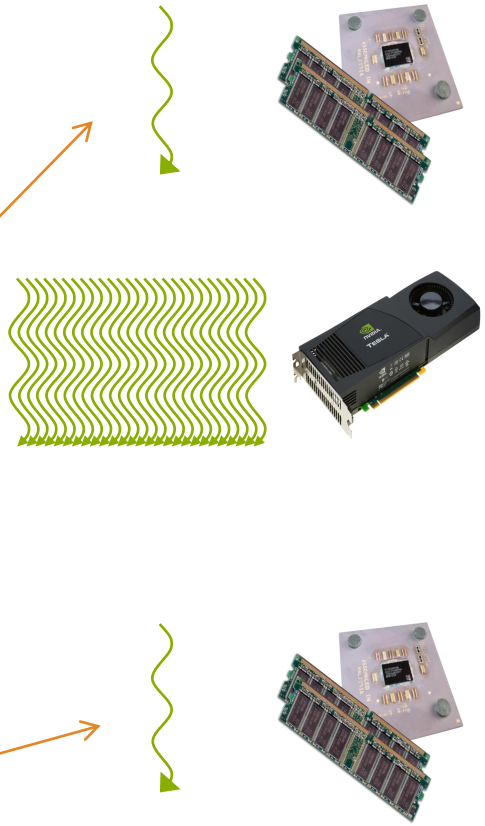
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

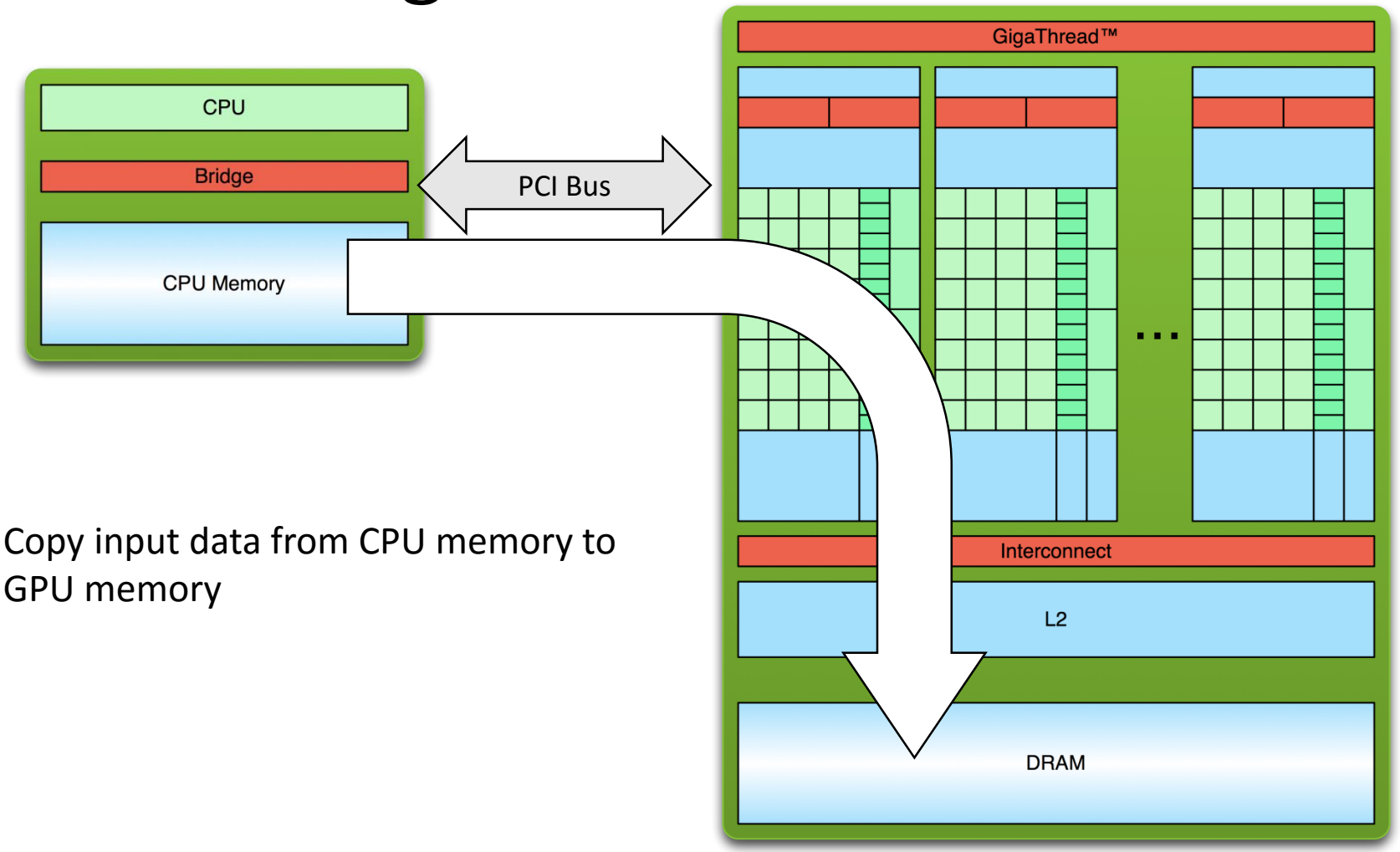
serial code

parallel code

serial code

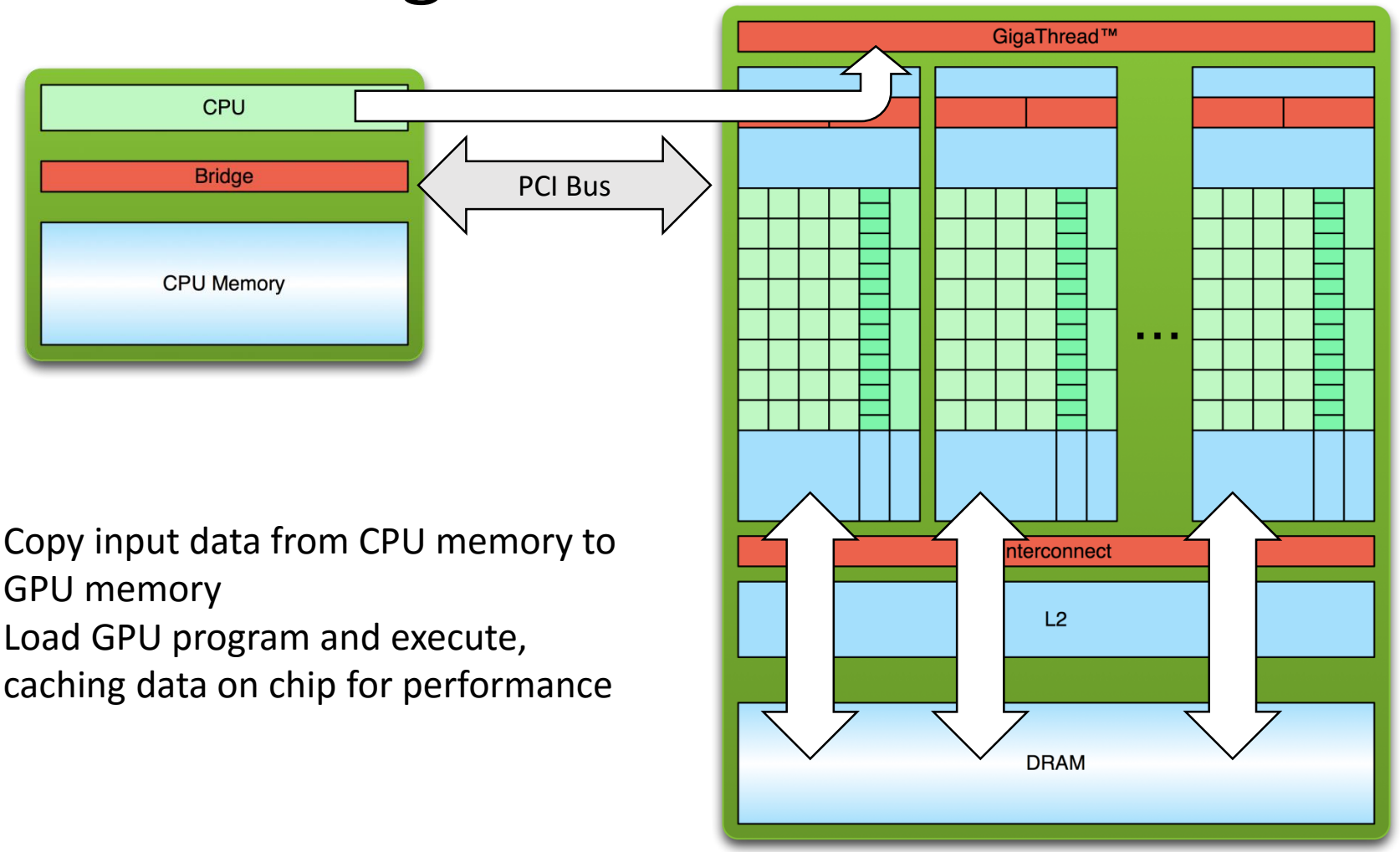


Simple Processing Flow



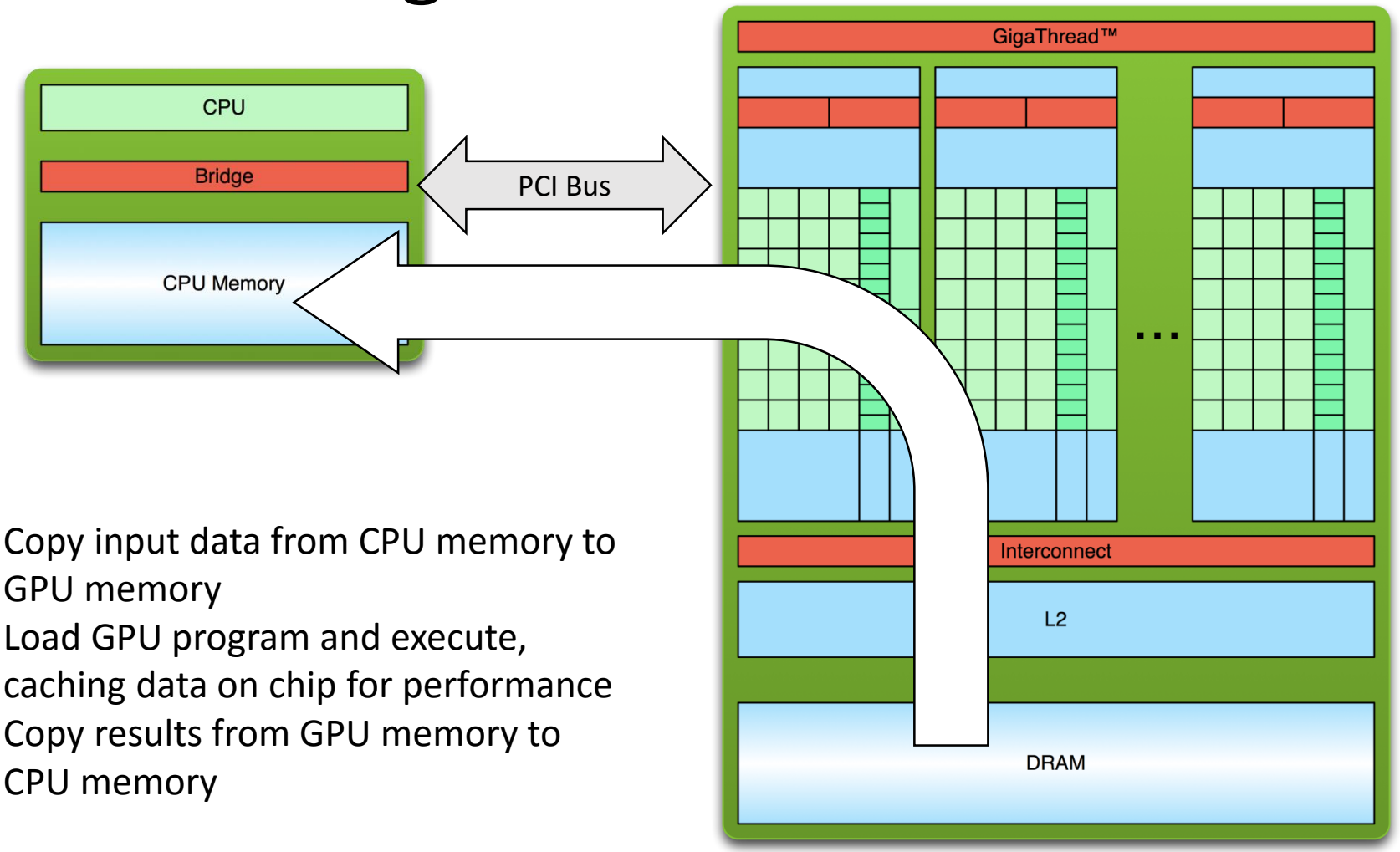
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory