Lecture 11:

OpenGL Shading Language (GLSL)

CS 175: Computer Graphics

November 14, 2024

Problems with "Old" OpenGL

- You have been working with the "Old" OpenGL specification (also known as the "fixed pipeline" OpenGL model) in Assignments 1 and 2.
  - With Assignments 3 and 4 we didn't use OpenGL at all
- There are many problems with it… Most notably the feeling that the developer doesn't have full "control" over the output
- For example, with shading, with Assignments 1 and 2, we were at the mercy of the graphics card developer – sometimes the shading / coloring are just different
- With Ray Tracing, things are better in terms of "developer control" – you decide the color of each pixel.
  - But the downside is that it's run entirely in software and not taking advantage of a GPU!

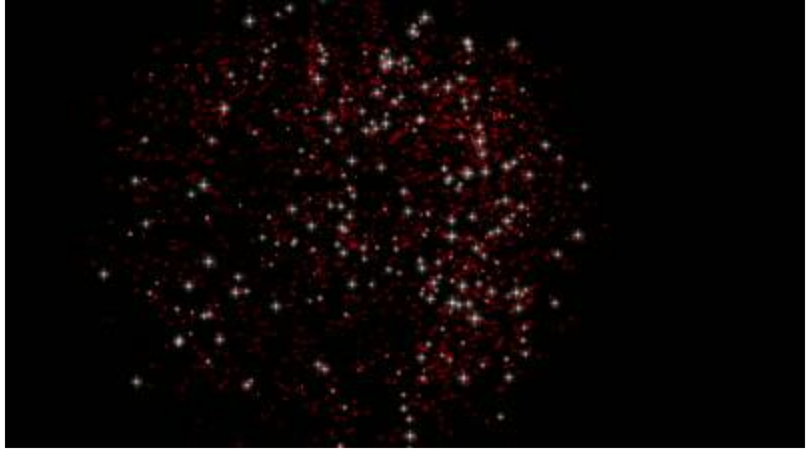
Inefficiencies in “Old” OpenGL

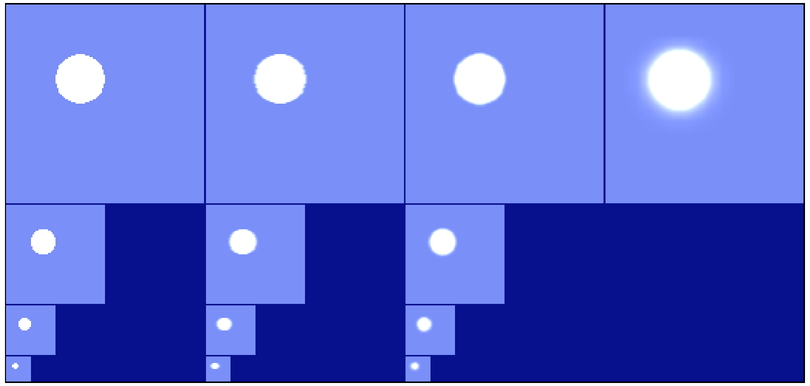
- There are many ways in which OpenGL is not very efficient…
  - For example, currently we send the entire content of an object (e.g. a PLY file), including vertex data, normals, etc. from CPU to GPU at 60 frames per second
- Over the years OpenGL kept adding awkward commands to:
  - Improve efficiency
  - While maintaining developer control…
- For example, in “old” OpenGL, we can “store” the entire “draw” function on GPU (yay! No more wasted data transfer)
- But if we wanted to change any part of that “draw” function (e.g., to remove a part of an object), then the whole thing falls apart

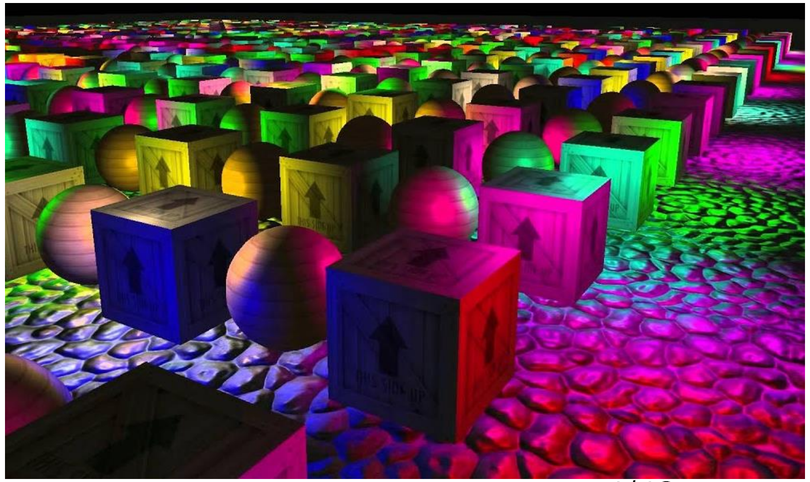
Also, Textures…

- Now that you see how complicated (and slow) real rendering is, how do you do some common techniques that appear “cool” but run in real time?

- For example:
- A glowing floating ball?
  - https://www.youtube.com/watch?v=PCE0k6PcDcw
  - Short answer: fake it with texture maps
- Moving light sources?
  - https://www.youtube.com/watch?v=nSL8cOxtsz4
  - Short answer: deferred shading







Advanced Rendering Techniques

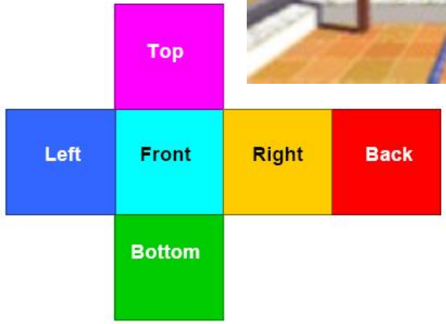
(and why we need shaders)

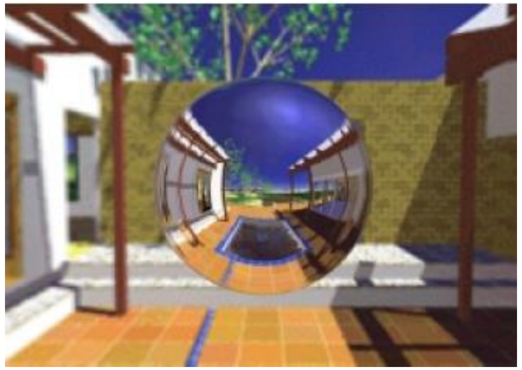
- Omitted due to time limit
- Will revisit in the next lecture!

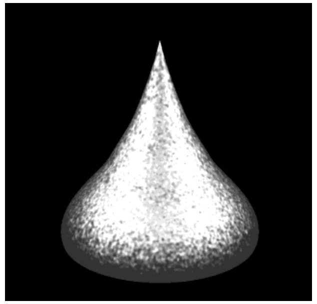
So what are shaders?

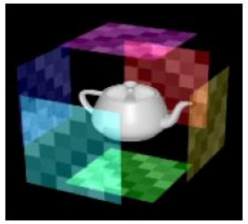
Motivation for GLSL

- It turns out, all of these “advanced” techniques are really hard to do using the current OpenGL workflow and API
- This is why GLSL was crated…









Graphics History

- Software Rendering
  - Old-fashioned way (Use ray casting)
- Fixed-Function
  - Immediate Mode (What we used in Assignments 1 and 2)
- Programmable Pipeline (~2001)
  - GL Shading Language (What we will learn today)
- CUDA/OpenCL (GPGPU) (~2008)
  - General Purpose GPU Programming
- Vulkan (~2016)
  - Multi-threading, Ray Tracing support

Software Rendering (some Ray Casting)

- Wolfenstein
- Quake 1

Fixed-Function

- Quake 3
- Half-Life

Programmable Pipeline (Shaders)

- Modern game (until real time ray tracing)… Fortnite, Star Wars Jedi: Fallen Order

Real-Time Ray Tracer

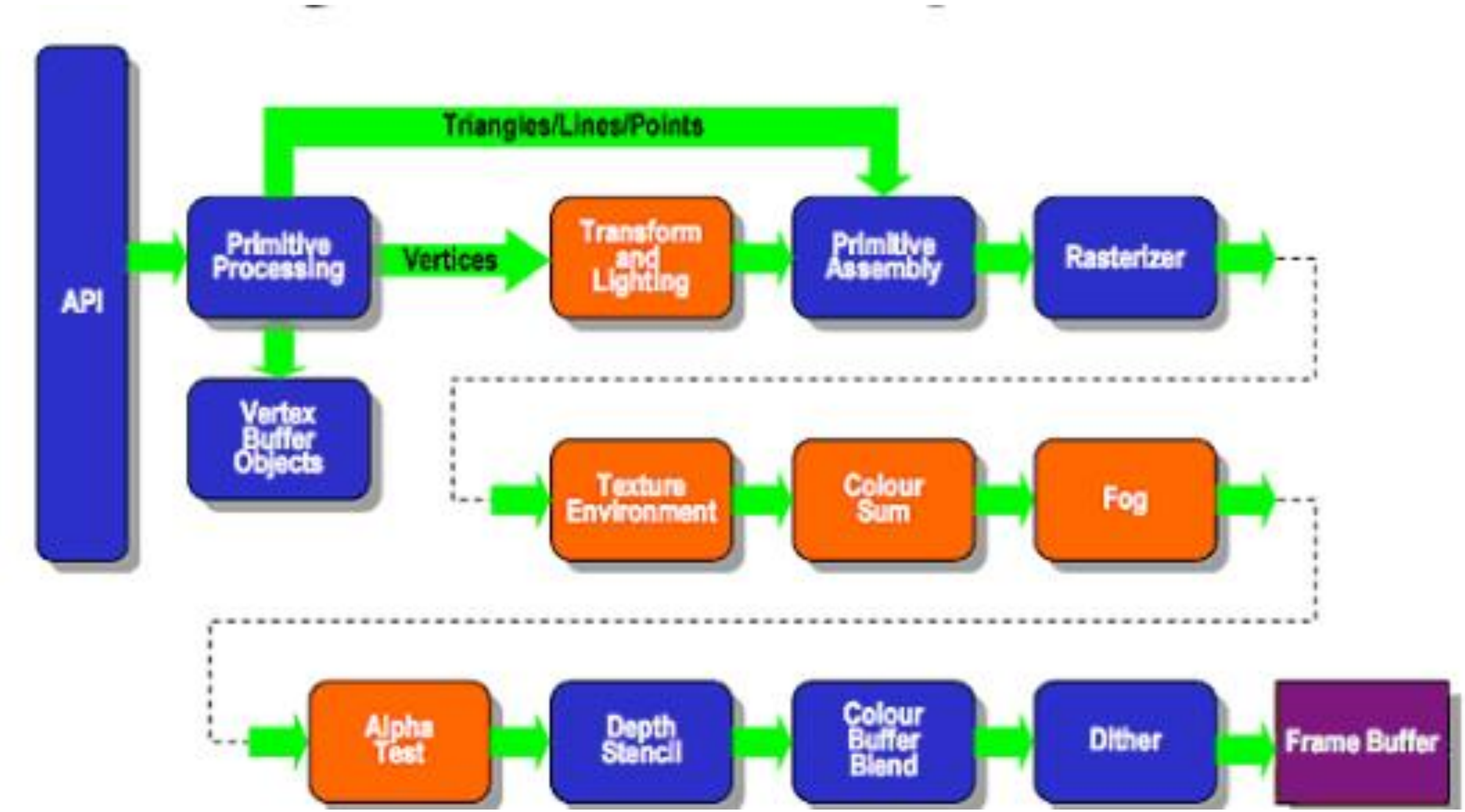
- Control
- Battlefield 5

- https://www.youtube.com/watch?v=476N4KX8shA

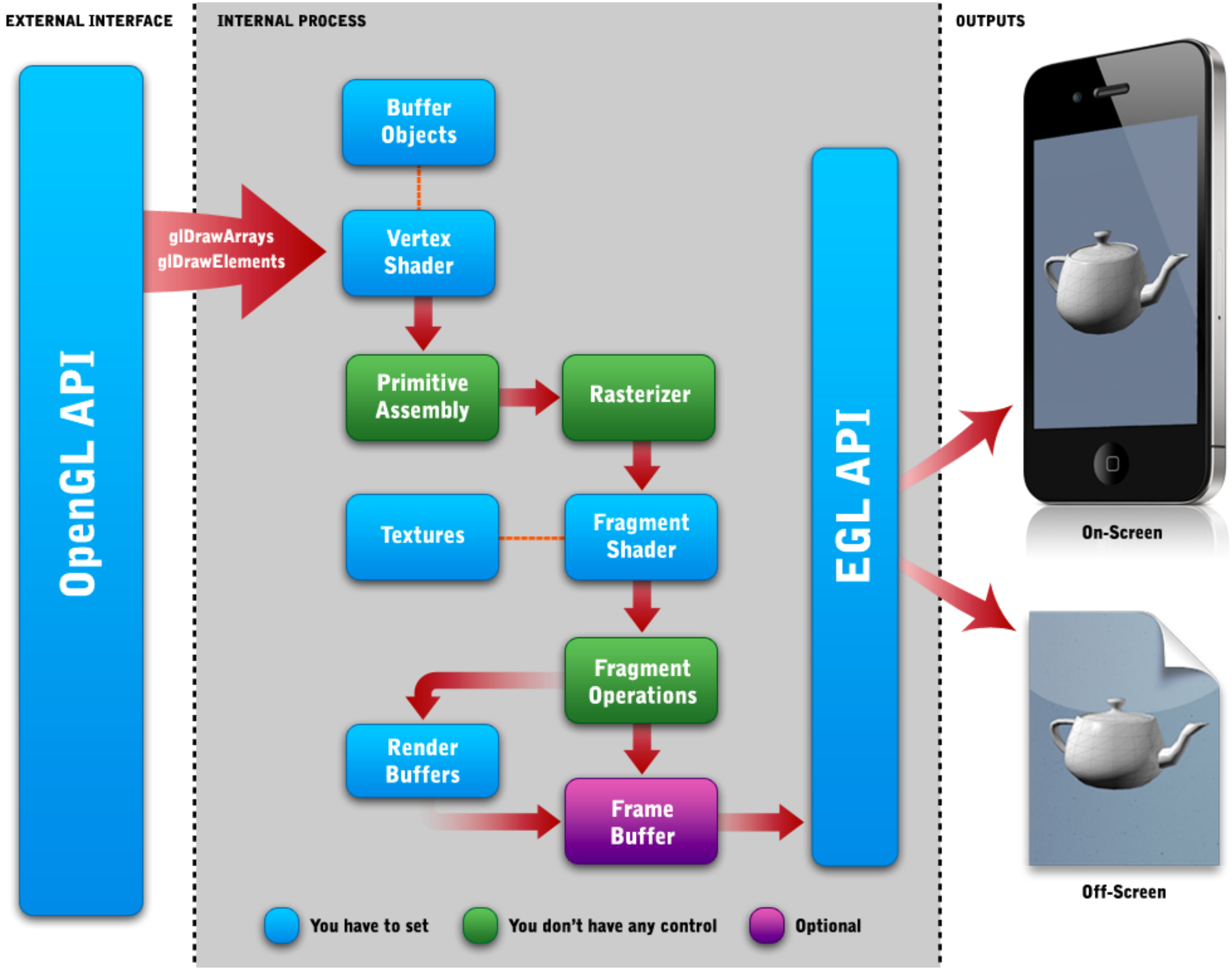
GPGPU

- General Purpose Graphics Processing Unit
- Utilize graphics card for computation
  - Do work other than graphics

Old Fixed-Function

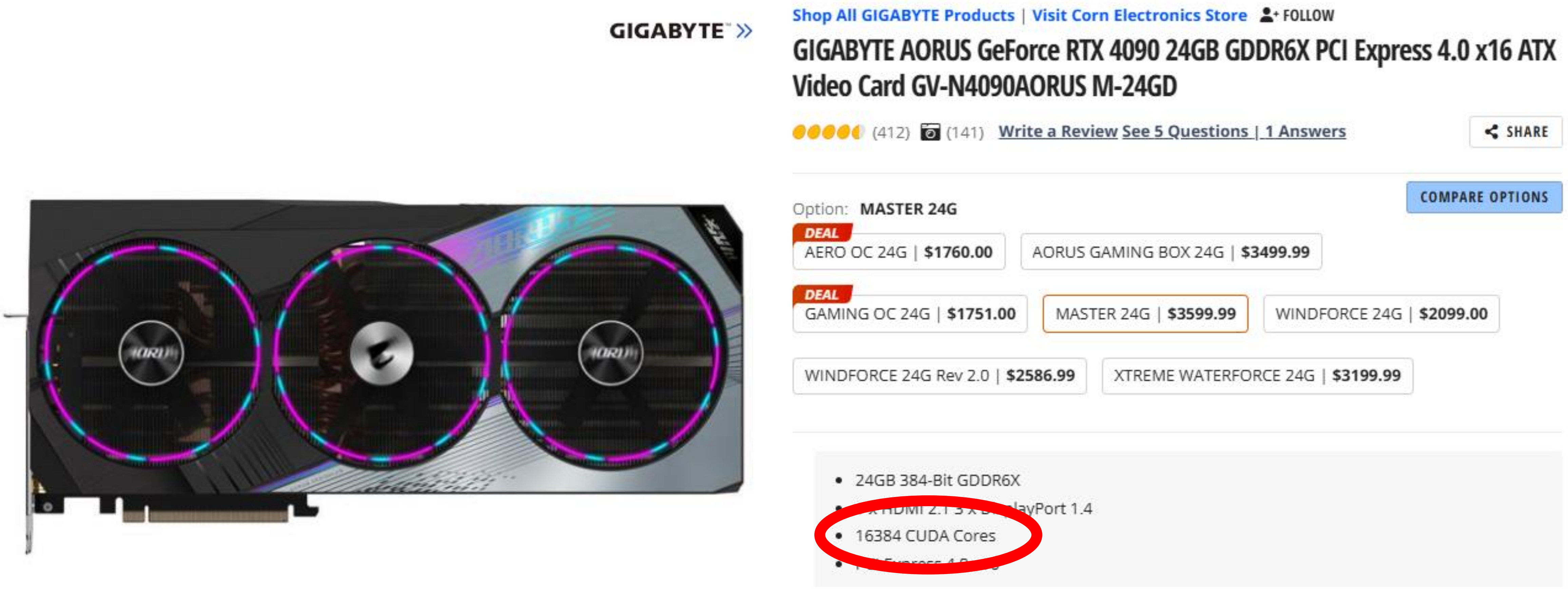


Programmable Pipeline



Why GPU?

- Having two (CPU and GPU) is better than one…
- GPU is dedicated to floating point operations (a traditionally expensive operation on a CPU)
  - It is especially optimized for linear algebra (e.g. matrix manipulation)!
- GPU is multi-core (multi-processor)
  - Imagine that you have one processor per pixel, how would you program what the processor should do?
- GPU is now optimized to work with textures
  - Modern GPUs come with large amounts of memory, almost comparable to CPUs


GIGABYTE
Shop All GIGABYTE Products | Visit Corn Electronics Store FOLLOW
GIGABYTE AORUS GeForce RTX 4090 24GB GDDR6X PCI Express 4.0 x16 ATX Video Card GV-N4090AORUS M-24GD
(412) (141) Write a Review See 5 Questions | 1 Answers
SHARE
Option: MASTER 24G
COMPARE OPTIONS
DEAL
AERO OC 24G | $1760.00
AORUS GAMING BOX 24G | $3499.99
DEAL
GAMING OC 24G | $1751.00
MASTER 24G | $3599.99
WINDFORCE 24G | $2099.00
WINDFORCE 24G Rev 2.0 | $2586.99
XTREME WATERFORCE 24G | $3199.99
24GB 384-Bit GDDR6X
16384 CUDA Cores

Versions of OpenGL and GLSL (and why you need to care)

Confusions galore…

Many versions of OpenGL

- OpenGL 2.0 (2004) introduces the idea of shaders
  - OpenGL SL 1.0 (or 1.1) is a part of OpenGL 2.0
- Currently, OpenGL is at 4.6 (2017). GLSL is also at 4.6
  - When OpenGL 3.1 was introduced, GLSL changed versioning to match (to 3.1), which was previous 1.3.1
- 3.0 -> 3.1 (2009) is said to be the "big move"
  - where fixed pipeline is considered deprecated.
  - Mac does split support (only 3.0+ or only <3.0). Mixed syntax is not allowed

GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59[1]	2.0	30 April 2004	#version 110
1.20.8[2]	2.1	07 September 2006	#version 120
1.30.10[3]	3.0	22 November 2009	#version 130
1.40.08[4]	3.1	22 November 2009	#version 140
1.50.11[5]	3.2	04 December 2009	#version 150
3.30.6[6]	3.3	11 March 2010	#version 330
4.00.9[7]	4.0	24 July 2010	#version 400
4.10.6[8]	4.1	24 July 2010	#version 410
4.20.11[9]	4.2	12 December 2011	#version 420
4.30.8[10]	4.3	7 February 2013	#version 430
4.40.9[11]	4.4	16 June 2014	#version 440
4.50.7[12]	4.5	09 May 2017	#version 450
4.60.5[13]	4.6	14 June 2018	#version 460

Open GL ES

- OpenGL ES (ES = Embedded Systems) is meant for smart phones, devices, etc.
  - OpenGL ES 1.0 (2003) implemented OpenGL 1.3
  - OpenGL ES 2.0 (2007) is a reduced set of OpenGL 2.0
  - But ES 2.0 “jumped ahead” and eliminated almost all fixed-pipeline operations (e.g. glBegin, glEnd)
- OpenGL ES 2.0 is not backward compatible with previous versions
- Current version is ES 3.2 (2015)

WebGL

- WebGL is OpenGL for browsers, in particular HTML5 canvas element.
- WebGL 1.0 is based on OpenGL ES 2.0 (2011)
- Current version is WebGL 2.0 (2017)
  - based on OpenGL ES 3.0
  - Largely backward compatible

CUDA / OpenCL

- Made for general purpose graphics programming (GPGPU)
- Popular known use is bitcoin mining
- CUDA is a language specific to NVIDIA (2007)
  - Obviously different from GLSL in that there are no graphics-related keywords or concepts. But the general idea (of parallel computing) is similar
- OpenCL is an open standard version (2009) of CUDA led by the Khronos Group
  - The Khronos Group is a non-profit consortium founded by major graphics companies
  - Currently Khronos is in charge of the OpenGL, OpenCL, OpenGL ES, WebGL, WebCL, and a bunch of related APIs.

Moving Forward in this Class…

- Mac is a pain… It only supports up to OpenGL 4.1
  - Check your computer: https://support.apple.com/en-us/HT202823
- Mac is a pain… It has stopped supporting OpenGL as of June 2018 and now only supports their proprietary graphics language called "Metal".
  - As a result, there are a lot of warnings when compiling OpenGL
- Mac is a pain… The use of either gl.h or gl3.h means that there's no backward compatibility
  - When programming in gl3.h, any fixed function call results in an error (e.g. glPushMatrix(), or glRotate3f(), etc.)
- Windows is a pain… There are many graphics card vendors
  - … the graphics card vendors don't support all aspects of OpenGL equally
  - As a result, the use of the GLEW library is almost strictly required

Why Do I Care?? GLSL 1.2 vs. 3.3+

- There's a rather large departure between GLSL 1.2 and 3.3
- GLSL 1.2 is easier to learn and get a program running.
  - However, it doesn't work for GL 3.0 (so Mac is a problem because of its split support)
- GLSL 3.3+ is more "clean", but we need to learn about Vertex Buffer Object (VBO).
  - We'll discuss later in passing… It's a lot of:
    - Memory packing and alignment into the correct format for GLSL
    - Magic incantation to run it
- Because of the differences in syntax, you need to specify the shader version in your GLSL code!

Accessing Programmable Pipeline

Introducing Shaders!

CPU + GPU

- CPU: Typical C++ code
- GPU: OpenGL Shading Language (GLSL)

1. CPU passes data to GPU

2. Some processing happens on the CPU:
   - Load and parse data from disk
   - Get mouse movement

3. Some processing happens on the GPU:
   - Transform point from Object to World space
   - Solve lighting equation

4. GPU Renders an image

CPU + GPU (Continued)

- Think of this as two separate processors. What we'll learn today are:

1. What the GPU should do (and what the CPU should do).
   - In particular different "shaders" to fit into the different parts of the graphics pipeline

2. How to communicate between CPU and GPU
   - Sharing "chunks of memory"
   - Passing variable **names** and **values**
   - (and how to pass variables between different shaders)

3. The "language" used by the GPU (i.e. OpenGL Shading Language, or OpenGL SL, or GLSL).

Types of Shaders

- Most common:
  - Vertex
  - Fragment (Pixel)

- Newer:
  - Geometry
  - Tessellation
  - Tessellation
  - Compute
  - (Ray Tracing)

What is a "Shader"?

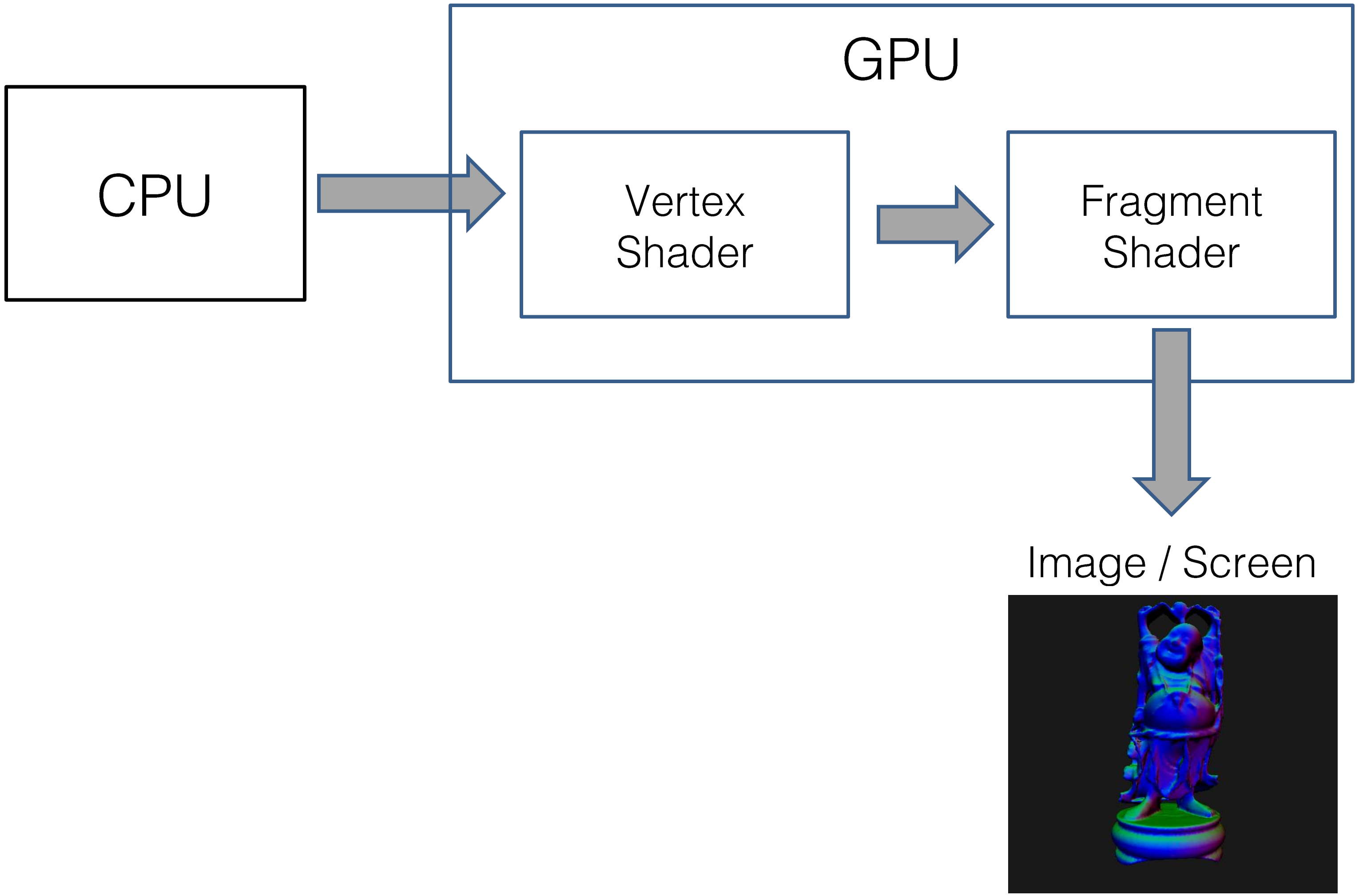
- A vertex shader is a (small) chunk of code that is run for each vertex of the object
- A fragment shader (sometimes referred to as pixel shader) is a (small) chunk of code that is run for each pixel on the screen

- The spirit of "shaders" is that each "shader" for a vertex or a pixel is run in parallel – much like threads – where there is NO COMMUNICATION between the execution of the other shaders
  - This takes advantage of the fact that graphics cards have many cores, which makes it easy to run programs (shaders) in parallel

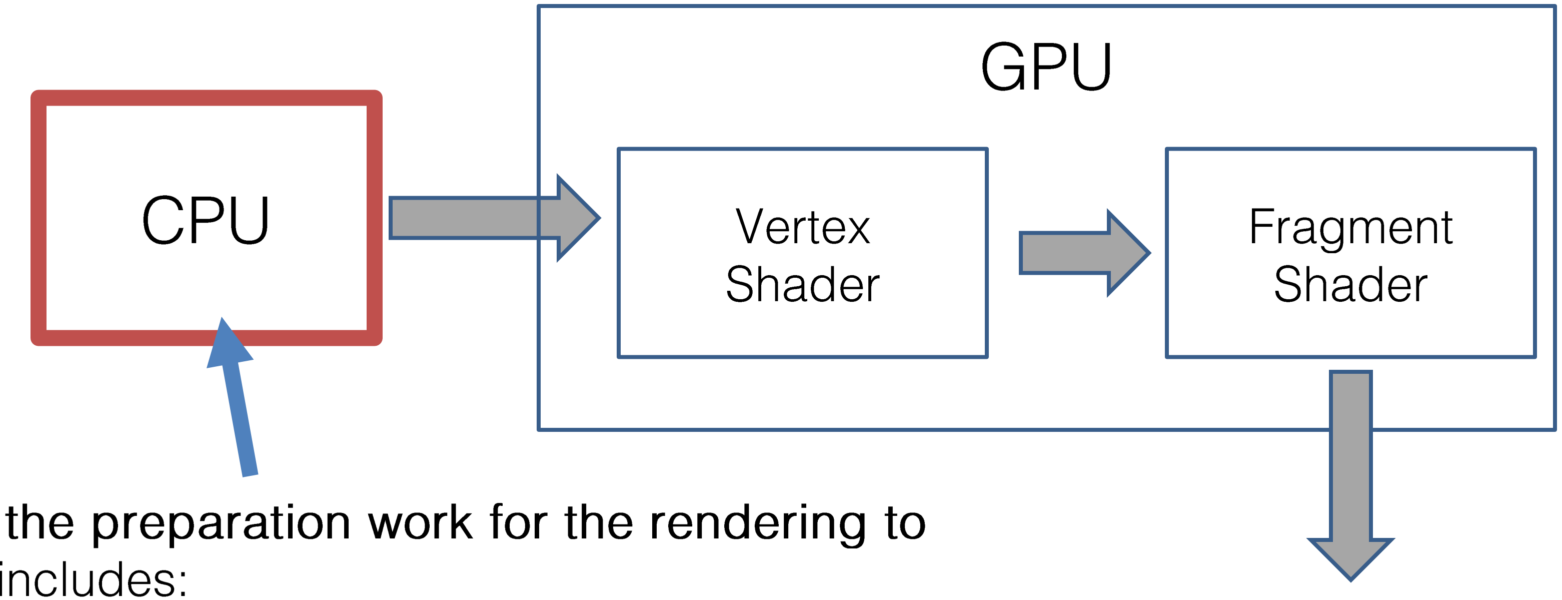
More Specifically…

- Vertex Shader
  - This program is run for each of the vertices
  - The “output” of this program is the position of the vertex in the “clipping plane” space
    - That is to say, not just camera space, but camera space after “unhinge” is applied to provide (perspective or orthographic) transform
- Fragment (Pixel) Shader
  - This program is run for each of the pixels
  - The “output” of this program is the color of that pixel

Control Flow



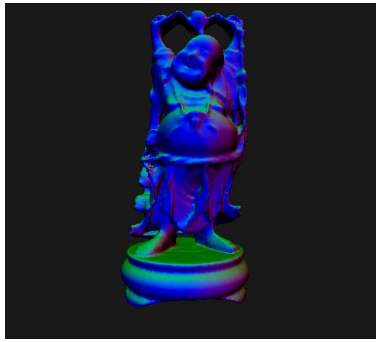
Control Flow



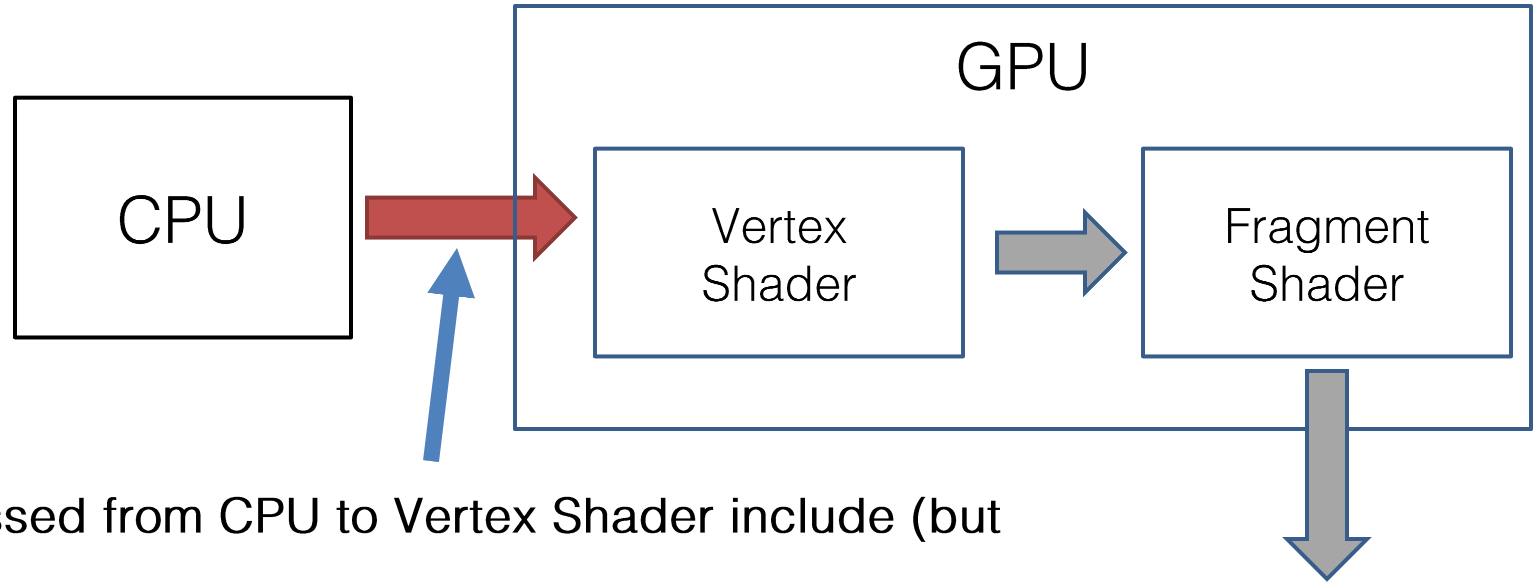
CPU does all the preparation work for the rendering to happen. This includes:

1) Read a PLY file from disk
2) Prepare the data (vertex, normal, texture, etc.) to pass to the GPU
3) Still has a “render” loop… In the render loop, we can do animation stuff (like changing camera position, lighting info, etc.)

Image / Screen



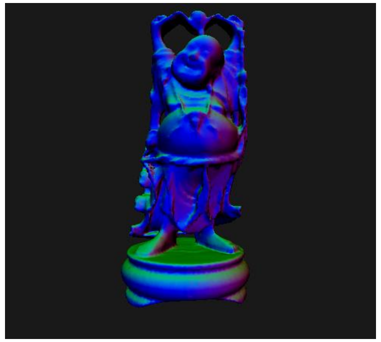
Control Flow



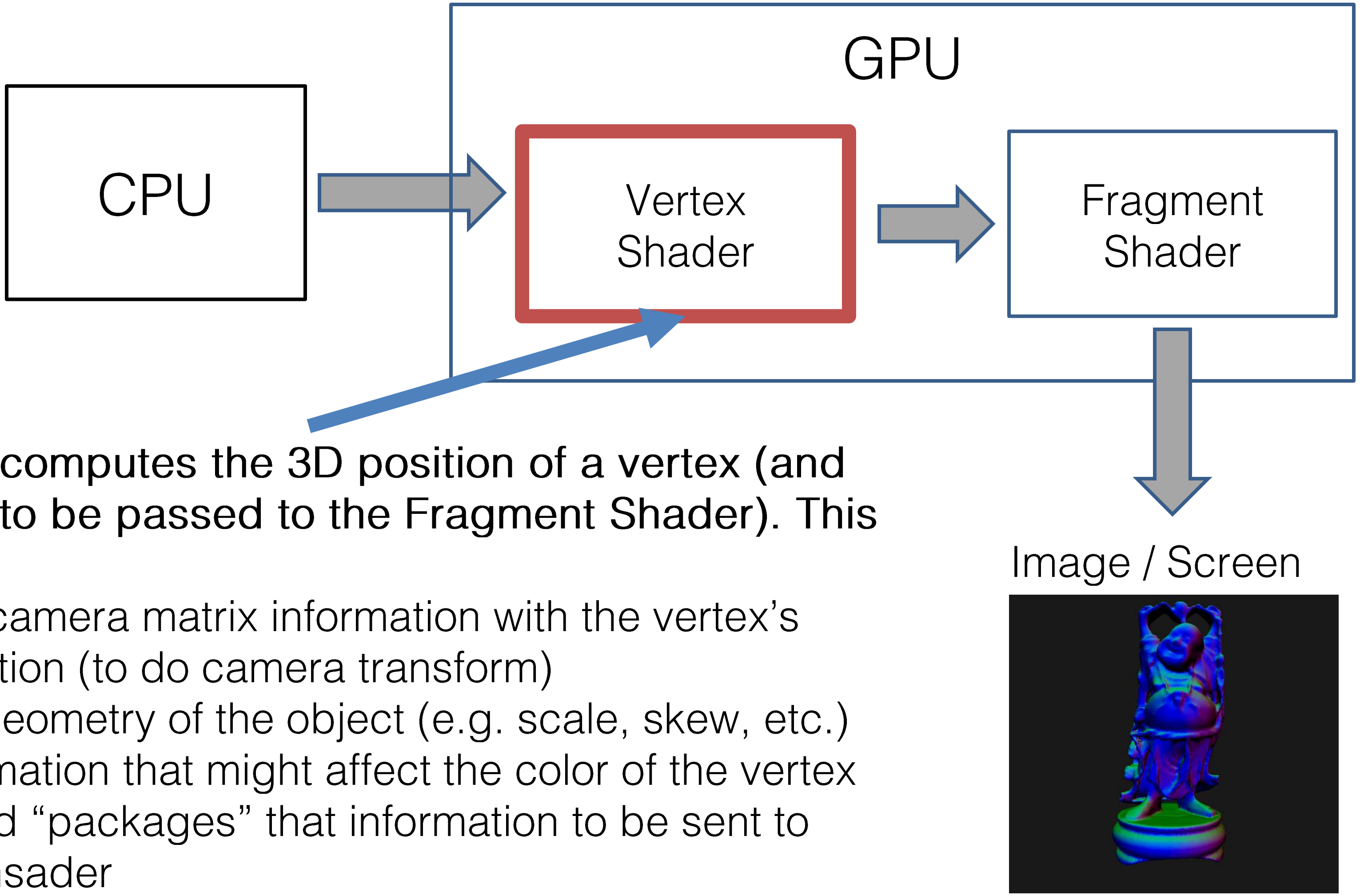
Data being passed from CPU to Vertex Shader include (but not limited to):

1) Camera's modelview matrix and perspective matrix
2) Position of the vertex
3) … and also things that the vertex shader doesn't care about but the fragment shader would care about. For example, color or texture

Image / Screen



Control Flow



Vertex Shader computes the 3D position of a vertex (and prepares data to be passed to the Fragment Shader). This includes:

1) Combining camera matrix information with the vertex's original position (to do camera transform)
2) Modify the geometry of the object (e.g. scale, skew, etc.)
3) Derive information that might affect the color of the vertex (or pixel) and "packages" that information to be sent to Fragment shsader

Control Flow

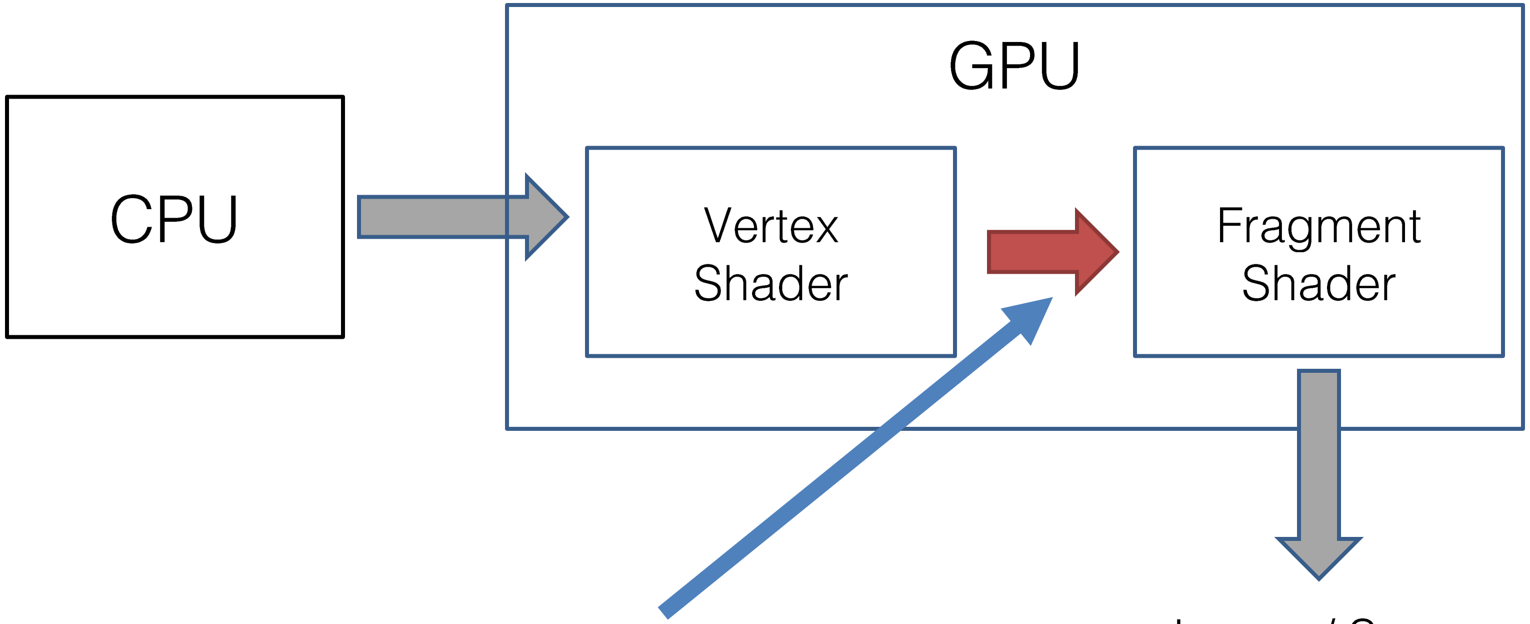
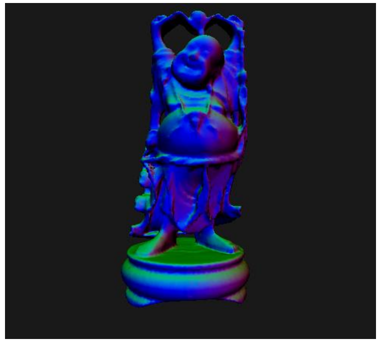
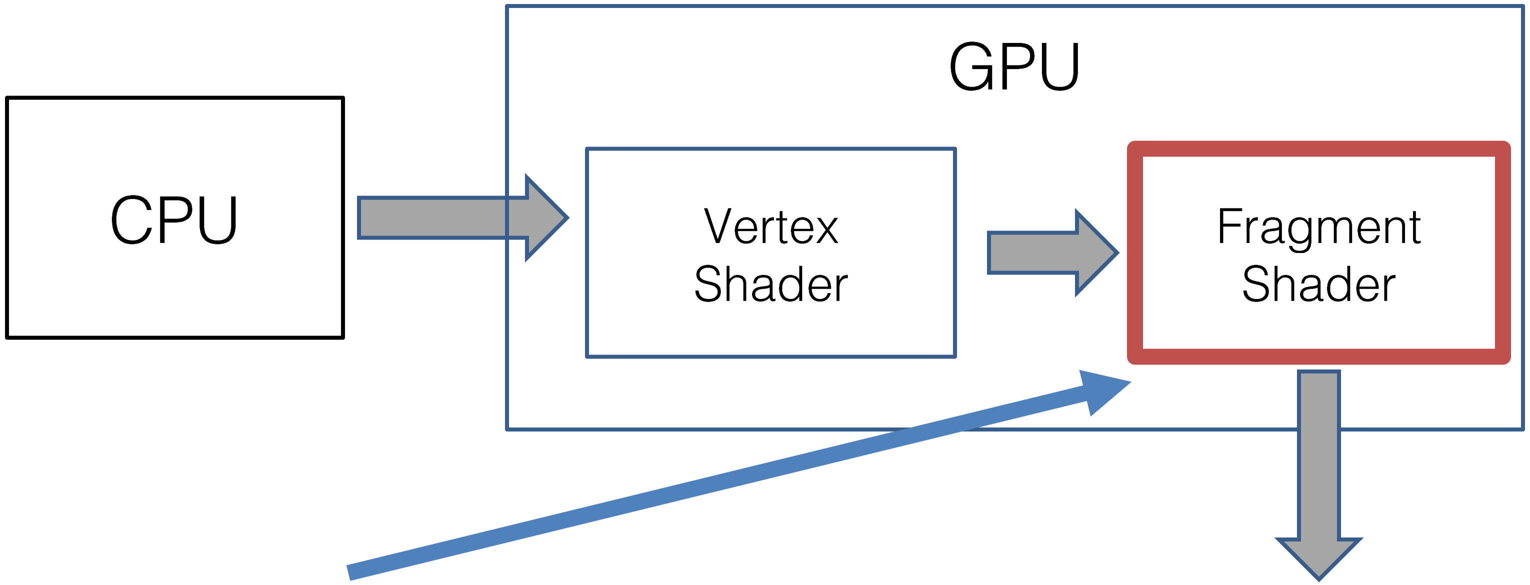


Image / Screen



Data being passed include (but not limited to):
1) Normals, color, texture, texture coordinate, etc.

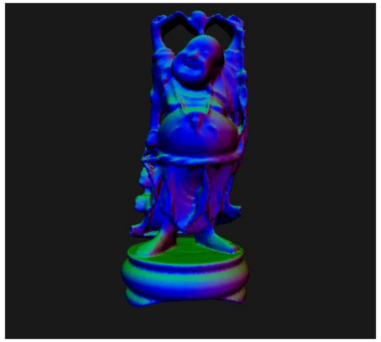
Control Flow



Determines the color of the pixel

1) For example, apply diffuse shading
2) Or, apply texture to the object
3) This is also where some of the “mapping” tricks we talked about before would happen (like normal mapping)

Image / Screen



Quick Demo of Shaders

Before We Start…

- There are 2 types of variables passed from C++ to shaders:
  - "uniform" – this is similar to a "global" variable where there's one copy of this variable (e.g. there's only one camera, so only one perspective matrix)
  - "in" – this is a "per vertex" (or "per pixel") variable. For example, there's a position and a normal per vertex
- To pass info between Vertex to Fragment shaders:
  - "out" – this is "per vertex" that passing information from vertex to fragment shader
  - "in" – on the receiving side, a fragment shader can receive the corresponding information
- Be Careful!! The names have to match EXACTLY (between C++ and Shader and between Vertex to Fragment shader). Capitalization Matters!!

Concepts to Cover…

- Quick VBO overview
- Rendering a basic outline
- Fragment Shader
  - Change color
  - Color by normal (pass variable from vertex to fragment)
  - Add diffuse lighting
- Vertex Shader
  - Add matrix transform (perspective first, then add modelview matrix)
  - Basic transform (scale, translate, etc.)
  - Funky stuff
    - Move points along normal
    - Modulate by timer