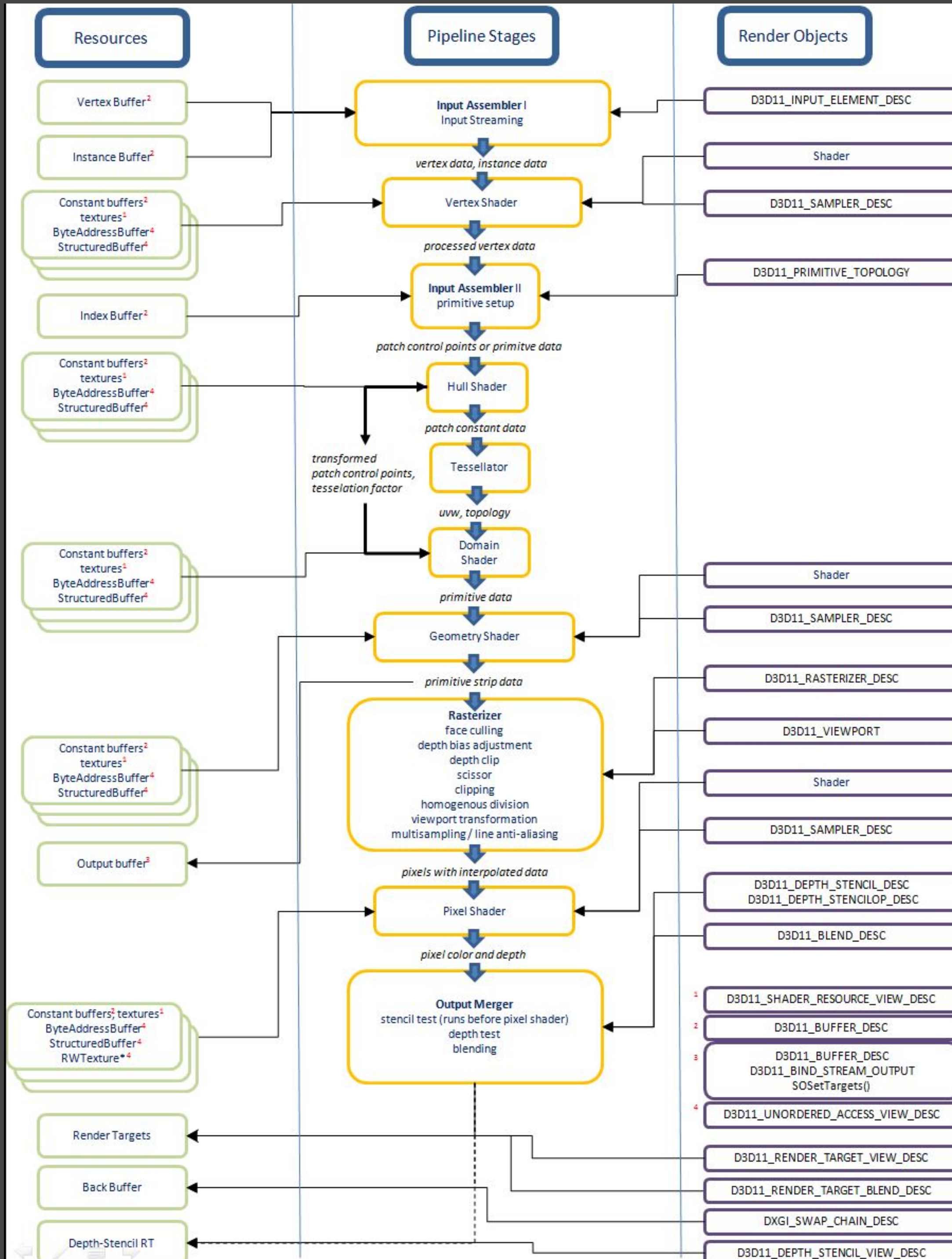
CS195V Week 2

Modern OpenGL


Resources
Pipeline Stages
Render Objects
Vertex Buffer
Instance Buffer
Constant buffers
textures
ByteAddressBuffer
StructuredBuffer
Index Buffer
Constant buffers
textures
ByteAddressBuffer
StructuredBuffer
Constant buffers
textures
ByteAddressBuffer
StructuredBuffer
Constant buffers
textures
ByteAddressBuffer
StructuredBuffer
Output buffer
Constant buffers, textures
ByteAddressBuffer
StructuredBuffer
RWTexture
Render Targets
Back Buffer
Depth-Stencil RT
Input Assembler I
Input Streaming
vertex data, instance data
Vertex Shader
processed vertex data
Input Assembler II
primitive setup
patch control points or primitve data
Hull Shader
patch constant data
Tessellator
transformed
patch control points,
tesselation factor
uvw, topology
Domain
Shader
primitive data
Geometry Shader
primitive strip data
Rasterizer
face culling
depth bias adjustment
depth clip
scissor
clipping
homogenous division
viewport transformation
multisampling / line anti-aliasing
pixels with interpolated data
Pixel Shader
pixel color and depth
Output Merger
stencil test (runs before pixel shader)
depth test
blending
D3D11_INPUT_ELEMENT_DESC
Shader
D3D11_SAMPLER_DESC
D3D11_PRIMITIVE_TOPOLOGY
Shader
D3D11_SAMPLER_DESC
D3D11_RASTERIZER_DESC
D3D11_VIEWPORT
Shader
D3D11_SAMPLER_DESC
D3D11_DEPTH_STENCIL_DESC
D3D11_DEPTH_STENCILOP_DESC
D3D11_BLEND_DESC
1 D3D11_SHADER_RESOURCE_VIEW_DESC
2 D3D11_BUFFER_DESC
3 D3D11_BUFFER_DESC
D3D11_BIND_STREAM_OUTPUT
SOSetTargets()
4 D3D11_UNORDERED_ACCESS_VIEW_DESC
D3D11_RENDER_TARGET_VIEW_DESC
D3D11_RENDER_TARGET_BLEND_DESC
DXGI_SWAP_CHAIN_DESC
D3D11_DEPTH_STENCIL_VIEW_DESC

OpenGL?!

- Open Graphics Library
- You already know basically what it can do
- CS123 used OpenGL 2.x (2004ish)
- We are going to use 4.x (2011)
  - Hip and happening!
- Some major changes in how you write graphics programs
- This week - Vertex Buffers, Vertex Arrays
- Next week - Shaders

Things that are deprecated

- Among other things...
- GLSL 1.1 and 1.2 (what we used in CS123)
- glBegin/glEnd primitive specification (more on this next)
- Fixed function vertex and fragment processing
  - No more default shader!
- Quad/polygon primitives
- Pixel/raster operations
- Old pixel formats
- Accumulation buffers
- Display lists
- Full list at (really long) http://www.opengl.org/wiki/History_of_OpenGL#Deprecation_Model
- When in doubt, check http://www.opengl.org/wiki/

Dealing With All this Versioning

- Computers have different versions of the OpenGL spec depending on hardware and os
  - Ex. Windows only ships with OpenGL 1.x spec :(
- How do we know what's supported and what's not (and how do we get write code that runs crossplatform)?
  - Ex. glBindFramebuffer might be part of EXT on one platform and core on another and ARB on yet another platform
- Most method sigs. can be found in GL/gl.h and GL/glext.h. (when including glext, you may want to define GL_GLEXT_PROTOTYPES)
  - But these signatures are not the same from computer to computer - which means compiler errors everywhere

Compiler Errors Everywhere

- GLEW (GL Extension Wrangler) and GLee (GL Easy Extension library)
  - Two different libraries with similar purpose: provide a cross platform extension loading library for GL
  - We will be using GLEW, but they essentially do the same thing
- Both allow for easy checking of supported extensions on a platform
  - In theory, you would use these libraries to check for supported methods (ex. via glewIsSupported(...)), but this is time consuming and boring, so we'll mostly ignore this
  - What we will use GLEW for is to avoid having to figure out whether to write glBindFramebufferEXT or glBindFramebuffer
  - With GLEW, you should be able to just write glBindFramebuffer

What Does EXT, ARB, etc. Mean?

- Each graphics vendor has a specific abbreviation they will use for new capabilities they add to the OpenGL core specification (for example, NVIDIA uses NV)
- If more than one vendor agrees on a certain extension, the abbreviation EXT will be used instead
- Finally, if the OpenGL Architecture Review Board (ARB) approves of the extension, the ARB abbreviation is used
- Before using any extensions, it is good practice, (but really annoying) to check to make sure that extension exists (which is why we have libraries like GLEW)

A bit on GLUT..

- GLUT (The OpenGL Utility Toolkit) provides several useful GL functions
  - You may have used it in CS123
- Important notes:
  - It's old and we don't really need it - instead the support does all its windowing and context management via Xlib

The glEnd of the glBegin-ning

- In the past, when you wanted to draw something...
  - glBegin
  - glColor, glNormal, glVertex, glTexCoord, etc.
  - glEnd
  - Run every time you paint to the screen
    - This means all the geom. data is uploaded every frame - complete waste of memory bandwidth
    - Bottleneck is the GPU-CPU memory interface
- OpenGL 3.1 and later removed these methods (and the rest of the fixed function pipeline)
  - All drawing is now expected to be done using shaders
  - If you've played around with OpenGL ES, you know this already
- The new way to describe geometry is through Vertex Buffer Objects (VBOs)

4 Steps to OpenGL Success!

1. Generate!
2. Bind!
3. ???
4. Profit!

- Many of the features of OpenGL follow this format

Vertex Buffer Objects

- A VBO is a collection of data representing your object
    - Positions, normals, colors, texture coordinates, etc.
- Faster because you can store in GPU memory for fast access
- Setting it up takes a little work...
    1. glGenBuffers
    2. glBindBuffer
    3. glBufferData

VBO Example

- Say we have a triangle mesh with points, normals, and texture coordinates
- We could make a struct for a vertex
  - Contains point, normal, texture coordinates
- Make a VBO to store an array of these structs
- Make another one to store the triangle indices
- To draw (pass to shader)...
  1. Bind the VBO for the vertex data
  2. glVertexAttribPointer to set up uniforms
  3. Bind the VBO for the indices
  4. glDrawElements

Vertex Array Objects (VAO)

- A convenient way to work with VBOs
- You can set vertex attributes for the VAO to point to the different data fields of your struct and the proper uniforms of the shader
- You can then bind your VBOs to your VAO
- If you tell the VAO to draw, it will properly pass the necessary information from its associated VBOs to the shader

Create VAO/VBO Example

```
void CreateVBO(void)
{
    GLfloat Vertices[] = {
        -0.8f, -0.8f, 0.0f, 1.0f,
         0.0f,  0.8f, 0.0f, 1.0f,
         0.8f, -0.8f, 0.0f, 1.0f
    };
    GLfloat Colors[] = {
        1.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 1.0f, 0.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f
    };

    glGenVertexArrays(1, &VaoId);
    glBindVertexArray(VaoId);

    glGenBuffers(1, &VboId);
    glBindBuffer(GL_ARRAY_BUFFER, VboId);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glGenBuffers(1, &ColorBufferId);
    glBindBuffer(GL_ARRAY_BUFFER, ColorBufferId);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Colors), Colors, GL_STATIC_DRAW);
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);
}
```

source: http://openglbook.com/the-book/chapter-2-vertices-and-shapes/#toc-object-creation

Additional Notes

- glVertexAttribPointer does not set any data! It just tells OpenGL how to find a specific attribute given a VBO later
- Working with VBOs is a lot like working with textures
  - glGen* creates a handle to access the object later
  - glBind* activates an object to be used/modified
  - For a texture, you had to call glTexImage2D to load the image data. For a VBO, you call glBufferData to load the actual vertex data.
  - Again, you have to bind the texture, VBO, or VAO before you use it in your drawing

Static, Dynamic, and Stream

- When specifying your vertex attribute pointers, you must select a usage mode - this usage mode will depend on how you plan to use the VBO
  - Make sure to choose the correct usage mode for best performance

- STREAM_DRAW - The data store contents will be specified once by the application, and used at most a few times as the source for GL drawing and image specification commands.
- STREAM_READ - The data store contents will be specified once by reading data from the GL, and queried at most a few times by the application.
- STREAM_COPY - The data store contents will be specified once by reading data from the GL, and used at most a few times as the source for GL drawing and image specification commands.
- STATIC_DRAW - The data store contents will be specified once by the application, and used many times as the source for GL drawing and image specification commands.
- STATIC_READ - The data store contents will be specified once by reading data from the GL, and queried many times by the application.
- STATIC_COPY - The data store contents will be specified once by reading data from the GL, and used many times as the source for GL drawing and image specification commands.
- DYNAMIC_DRAW - The data store contents will be specified repeatedly by the application, and used many times as the source for GL drawing and image specification commands.
- DYNAMIC_READ - The data store contents will be specified repeatedly by reading data from the GL, and queried many times by the application.
- DYNAMIC_COPY - The data store contents will be specified repeatedly by reading data from the GL, and used many times as the source for GL drawing and image specification commands.

Additional Notes (continued)

- VBOs are extremely flexible in the data that you hand to them (the example previous gives it data in the format (VVVV) (NNNN) (CCCC)
  - Note that in this format we must create a VBO for each attribute
- We can also give data to the VBO in two other formats:
  - (VVVVNNNNCCCC) - block
  - (VNCVNCVNCVNC) - interleaving
- Which way is best?
  - In general, the first choice is usually the least optimal - since it means we have multiple VBO bind calls and it's better to combine them into one
  - Usually, non interleaved formats give best results due to memory locality (although this may depend on hardware - usually ~5% perf. difference)
    - But at the same time, if we want to update the VBO information, we need to pull the attribute we want to update into its own VBO
  - Don't forget that when drawing a block
- TLDR
  - Use block ordering if you don't plan to update the VBO often, and use interleaved otherwise
  - Note that the support code uses interleaved data

Additional Notes (continued)

- You can tell GL your VBO format via
- void **glVertexAttribPointer** (GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer);
- For interleaved data, you would modify the stride parameter and the pointer offset
- For block data, you would simply pass it the offset pointer address
- When specifying vertex index data, use 16 bit unsigned integers (short), or unsigned byte (if you have <=256 vertices)
  - The last option is 32 bit unsigned integer if you have more than 65535 vertices

Transform Feedback

- Transform feedback is a method of storing vertices which have been processed by a vertex and/or geometry shader back into a VBO
  - Happens before rasterization or clipping
  - Equivalent to DirectX's Output Stream functionality
  - This allows you recursively modify VBO geometry data
- We won't really be using transform feedback much though...
  - But you're welcome to try it out!

Our First Project

- Remember Life from CS31?
- Life in GLSL
  - Each texel will be a cell
  - At each stage, we can operate on the entire texture to figure out the next state of the simulation
  - Afterward, visualize it in a cool way
- Pushing the computation to the GPU allows for a huge simulation space and super speed
- This (relatively) easy project should get you caught up with modern OpenGL and GLSL (don't use deprecated features!)

Handling the Matrix Stack (VSML)

- OpenGL no longer supports the fixed function matrix operations (glMultMatrix, etc.)
- We will be keeping track of the matrix stack ourselves using a library called VSML (Very Small Matrix Library)
- You can call transformations like rotate, translate, scale on the VSML instance similar to the old OpenGL commands
- You can call vsmlOrtho() or vsmlPerspective() to quickly switch between orthographic and perspective projections (see support code)

Data Structures in OpenGL

- The main way we manipulate data in OpenGL is through textures/framebuffers
- A texture is basically a 2D array, so any data structure you can implement in an array, you can implement on the GPU using textures
  - Octree textures, mesh colors?!
- If you draw a full screen quad, you can use a fragment shader to apply some computation to every element of the texture
  - OpenGL 4.2 introduced image load stores allowing read/writes to arbitrary locations of an image in any shader stage (more on this in a later lecture)
- Your basic compute pipeline will be to bind a framebuffer, render a quad, and then read the framebuffer data

Basic Shader Example

```
#version 400 core

uniform mat4 modelviewMatrix;
uniform mat4 projMatrix;

#ifdef _VERTEX_
in vec3 in_Position;
in vec3 in_Normal;
void main(void) {
    gl_Position = projMatrix * modelviewMatrix * vec4(in_Position,1.0);
}

#endif

#ifdef _FRAGMENT_
out vec4 out_Color;
void main() {

    out_Color = vec4(1.0, 1.0, 1.0, 1.0);
}
#endif
```

A Word on Texture Formats

- □In general, it is best practice to use the lowest memory texture format you can get away with
  - Note that supported texture formats may vary depending on manufacturer implementation
  - This can potentially save a lot of memory bandwidth
- For most computation you probably want a floating point format
  - These formats all end with the suffix F (ex. GL_RGBA32F) would be a 4 channel, 32-bit per channel texture
  - If you want to use a floating point format and don't need 32-bits of precision, use 16 bit precision formats instead (ex. GL_RGB16F)
  - Dont forget about integer formats (and use them where appropriate)!

Common OpenGL Mistakes

- Depth buffer precision
  - Depth buffers are not stored as floating point data - this is a common mistake (it is usually 16, 24, or 32 bit integer)
  - Depth values in the clip region are usually from 0.0 to 1.0 - these can be converted to the integer format by multiplying by the maximum value of the integer format
- Generating mipmaps
  - The correct non-deprecated way to generate mipmaps is to call glGenerateMipmap(target)
  - *Do not use gluBuild2DMipmaps*
  - If you're targeting OpenGL < 3.0, use GL_GENERATE_MIPMAP, otherwise use the first way
- Never create any GL resources in a draw loop (kittens will cry if you do)
  - This is like newing stuff in a loop - don't do it

Aside: OpenGL and Direct3D History

Most information from http://programmers.stackexchange.com/questions/60544/why-do-game-developers-prefer-windows/88055#88055

It was was 1990-something...

In the world that I grew up in...

- SNES and Sega Genesis out in the console market
- Most gaming on PCs went through DOS
  - Extremely low level programming, like the consoles at the time
  - However, unlike consoles, no known hardware configuration
- Microsoft was getting ready to launch the Windows platform
- To promote game development on Windows, they needed a fast, low level API that could work across all different kinds of hardware
- This was the beginning of DirectX

Meanwhile in OpenGL-land

- In the early 90's Silicon Graphics's Iris GL API was the industry standard for workstation graphics
- It competed against the open standard PHIGS
- As more graphics hardware manufacturers entered the market (supporting PHIGS), hurting SGI's market share
- In response, SGI decided to convert Iris GL into an open API
- Thus, OpenGL was born, first version released in 1992

Back to DirectX...

- Consumer 3D accelerators started coming out soon after the release of DirectX
- As of then, DirectX only had a 2D graphics component called DirectDraw
- Bought RenderMorphics to build their 3D API
- The result: Direct3D version 3
    - People allegedly hated it
- Microsoft concurrently developed OpenGL support for Windows, primarily for workstation applications

How new features are implemented

- Direct3D managed by Microsoft
  - New releases every 6-12 months or so
- OpenGL managed by the OpenGL Architectural Review Board (ARB)
  - Extensions mechanism allows for implementation of new hardware features as soon as they are available
  - However, can cause problems of their own
  - You may remember seeing EXT and ARB in some of your GL code
    - EXT is an extension provided by a specific hardware vendor
    - ARB is an 'official' extension which has been blessed by the ARB

Direct3D vs. OpenGL, the early days

- Direct3D was first to have programmable shaders in D3D8
- nVidia Geforce3, ATI Radeon 8500 release
  - D3D had to release a new shader version (1.1) to use the 8500's features
  - OpenGL had to have a different set of extensions for both cards...
- OpenGL originally focused on adding features to the fixed function pipeline instead of doing things in shaders
- 3DLabs designed the first OpenGL Shading Language
  - Lots of compiler and optimization problems...
  - Ultimately had a lot of things right, but not for the time

More recently...

- Direct3D
  - Dominant in the PC game market
  - D3D9.0c available for Xbox360
  - Some porting to Windows Phone
- OpenGL
  - More common in professional applications
  - Only choice for Mac/Linux
  - PS3 can use an OpenGL wrapper
  - OpenGL ES for embedded devices (smartphones)
- In general though, game console developers like to use native APIs to maximize performance
- Both currently offer similar functionality - since the department here runs Linux, we'll be using OGL

Writing Your First GL program

- We will be using minimal 3rd party libraries (basically GLEW and maybe an image loading library)
- This means no Qt, etc.
- In CS123, Qt magically created a window and OpenGL drawing surface for you
- The support code we have provided uses minimal 3rd party libraries (basically just GLEW)
    - Windowing is done through X - and because of this is not cross platform
        - Although I do have a Win32 version somewhere (so if anyone wants to try developing on Windows...)
    - Support code also support for some useful GL stuff (Textures, Framebuffers...)
    - I don't guarantee correctness of the support code...

A Very Quick Intro to CMake

- The support code uses CMake (Cross platform Make)
- CMake is a cross-platform tool for generating build chains
  - Ex. it can generate Makefiles, Visual Studio Solutions...
  - Used by many projects (OpenCV, LLVM, Clang, Blender, KDE4...)
- In each folder you should see a CMakeLists.txt, which specifies how to build the files in that folder
  - If you add new source files or libraries make sure to modify the relevant information in CMakeLists
- Example

Case Study : Tone Mapping

References:
Real-Time Rendering, 3rd Edition
http://developer.nvidia.com/node/183

HDR / Tonemapping

- Most of you are probably sick of hearing about HDR / tone mapping
  - HDR is the process of capturing high dynamic range information (in OpenGL, this corresponds to luminance values stored in a floating texture outside the [0..1] range)
  - Tone mapping is the process of remapping these values back to 0...1
- It is trivial to implement in shader code a simple tone mapping equation such as ($L = Y / (Y + 1)$), where Y is radiance
  - But this gives relatively poor results (images look washed out)
  - What about a slightly more complex one (the above tone mapping equation gives mediocre results at best)?

HDR / Tonemapping

- According to Reinhard et al., when summing up pixel values it is usually better to take the logarithm of the luminance and average these values, then convert back, such that
  - Lavg = exp(1/N * sum(log(epsilon + L(i,j)))
  - Where Lvg is the log average luminance, and L(i,j) is the luminance at pixel (x,y)
- GL Implementation
  - The above tonemapping equation is relatively simple, except we need to find the average luminance among pixels
  - One way to do this would be to read the pixels back to the CPU and compute the average before passing the computed average back as a uniform
  - Remember that memory transfer is slow...
    - Can we do better?

HDR / Tonemapping

- Essentially we want to perform a reduce operation on the pixel values
  - OpenGL and DirectX have this functionality built in for summing and averaging
- Solution: mipmap the texture
  - Mipmap the texture down to its lowest level (1x1) glGenerateMipmap, and then sample (use textureLod to sample using integer coordinates at a specified mipmap level)
- When possible, use built-in functionality!

Color Object Detection

- Being able to quickly average an images values can be used in other applications
- See GPU Gems 3: Chapter 26. Object Detection by Color: Using the GPU for Real-Time Video Image Processing
    - They use it to find the centroid of pixel values with a certain color