**Lecture Outline**

- Reading for Last Class: §2.5, 2.6.1 – 2.6.2, 4.3.2, 20.2, Eberly 2*
- Reading for Today: §2.7, Eberly 2*; Direct3D handout
- Reading for Next Class: §2.6.3, 20.3 – 20.4, Eberly 2*
- Last Time: Intro to Illumination and Shading
  - Local vs. global models
  - Illumination (vertex shaders) vs. shading (fragment/pixel shaders)
  - Phong Illumination equation: introduction to shading
- Previously: Projections, Modelview, Viewing, Clipping & Culling
- Today: Direct3D Design Overview, Programming Intro
- Hardware Rendering: Application Programmer Interfaces (APIs)
- Shaders in Modern Pipeline
  - Vertex shaders: vertex attributes to illumination at vertices
  - Pixel shaders: lit vertices to pixel colors, transparency
- Next: Texture Mapping Explained; Shading in OpenGL

**Vertex Shaders vs. Pixel Shaders**

- **Classical Fixed-Function Pipeline (FFP):** Per-Vertex Lighting, MVT + VT
  - Largely used in mobile computing
- **Modern Programmable Pipeline:** Per-Pixel Lighting
  - Vertex Shaders (FFP and Programmable)
    - Input: per-vertex attributes (4-D, object space position, normal)
    - Output: lighting model terms (e.g., diffuse, specular, etc.)
  - **Pixel Shaders (Programmable Only)**
    - Input: output of vertex shaders (lighting aka Illumination)
    - Output: pixel color, transparency (R, G, B, A)
- Brief Digression
  - Note: vertices are lit, pixels are shaded
  - "Pixel shader": well-defined (iff "pixel" is)
  - "Vertex shader": misnomer (somewhat)
  - Most people refer to both as "shaders"

**Direct3D**

- Popular 3-D Graphics Library of DirectX (Sometimes Called “DirectX”)
  - Top market share due to Windows & Xbox platforms
  - Many developers (has overtaken OpenGL except on smartphones)
- Can Implement Many Effects We Have Studied/Will Study Such As:
  - Cube map (Lecture 12)
  - Particle system (Lecture 28)
  - Instancing (Lecture 20 & Advanced CG)
  - Shadow volume (Lecture 12)
  - Displacement Mapping (Lecture 12)

**Acknowledgements**
OpenGL State Machine (Simplified) from Wikipedia:

- HLSL: Shader language and API developed by Microsoft, only usable from within a DirectX application.
- Cg: Shader language and API developed by Nvidia, usable from within a DirectX and OpenGL application. Cg has a stark resemblance to HLSL.
- GLSL: Shader language and API developed by the OpenGL consortium and usable from within an OpenGL application.

DirectX

- DirectX is a collection of APIs developed by Microsoft for multimedia and game applications.
- They utilize the Component Object Model (COM), allowing you to use them across a wide variety of languages.
- The entire DirectX API is available freely from Microsoft, and works with any current Visual Studio compiler (and many others).
- Today we'll be focusing on the API for 3D rendering, Direct3D.

Abstraction

- The primary purpose of Direct3D is to provide a uniform interface for working directly with the graphics hardware without having to go through windows.
- Direct3D applications actually render to a display surface within the confines of a window – they don’t use the Window API.
- For best performance, you want to use full-screen, exclusive mode. In this mode, Windows does no rendering whatsoever, allowing you to leverage the full resources of the computer.

Direct3D Device

- Direct3D places most of its rendering control in a Direct3D Device object.
- Essentially, the Direct3D Device is a wrapper around the graphics hardware, and exposes most of its functionality to the programmer through a common interface.
- You can poll the Direct3D Device to learn of the graphics hardware's capabilities.

Direct3D Rendering Pipeline

- The process of transforming model geometry into on-screen images is called the rendering pipeline.
- In DirectX, the pipeline runs entirely on the graphics hardware – there is no software fallback.
- Other than that difference, it shares many characteristics with other 3D rendering libraries.
Input Assembly

- Input is sent from the DirectX application in the form of Vertex data (both buffered and unbuffered), shader code and variables, and textures.
- When possible, these sources are stored in the on-board graphics card memory.

Vertex Formats [1]: Flexible Vertex Format (FVF)

- You also will define the format of vertices that you will send to the graphics card.
- We use the Flexible Vertex Format (FVF) to do this.
- FVF is a two part process – we define a struct representing our vertex format, and define a constant describing the structure of the format.

Vertex Formats [2]: Using FVF

```c
#define POSITION_NORMAL_TEXTURE_FVF(D3DFVF_XYZ
| D3DFVF_NORMAL | D3DFVF_TEX1)

struct PositionNormalTextureVertex
{
    float x, y, z;
    float nx, ny, nz;
    float tu, tv;
};
```

Vertex Formats [3]: For More Details on FVF

- Flexible Vertex Formats allow us to send the bare minimum of vertex data for our needs to the graphics card.
- Since the pipeline from our CPU to the graphics hardware tends to be a bottleneck, this is a must when speed is a concern.
- More detail on what flags exist for the Direct3D FVF can be found in the DirectX API documentation.

Vertex Buffers

- Vertex Buffers are used to hold vertex data and to stream this data to the graphics card.
- Buffers may be located in Video memory or in RAM, depending on your device setup.
- Buffers are a shared resource – both the CPU and the GPU will be accessing them at times. Because of this, we will need to lock the buffer before we place our data into it.
- To minimize overhead, it is common practice to create your vertex data in CPU-only memory, then do a fast copy (memcpy) into a locked Vertex buffer.
First, you need to set it as a stream source on your Direct3D Device.
Then you can call DrawPrimitive on your device, indicating what portion of the Vertex Buffer should be drawn, along with what kind of primitive you are rendering.
This starts the vertex data on the Rendering Pipeline.

Index Buffers
- Index Buffers are an array of indices which reference an associated Vertex Buffer.
- Index Buffers allow us to re-use Vertex data elements when creating meshes – this is especially useful because triangles in most meshes are sharing their vertices with adjacent triangles.
- By using an index buffer, we only have to define a vertex once, then we can define the triangle by indices; and since an index is usually quite a bit smaller than a vertex’s data, we send less data into our pipeline.

Rasterization
- Rasterization is the process by which geometry is converted into pixels for rendering.
- It is much the same with any rendering process, in general, refers to all vector-to-raster stages.
- Sometimes equated with scan conversion (final step), Direct3D maintains useful buffers for it, such as depth buffers and stencil buffers.

Vertex Shaders [1]
- The next step down the pipeline are the Vertex Shaders.
- Vertex shaders allow us to manipulate our vertex data on the graphics card.
- This is important, because it means we’ll be using the GPU.

Vertex Shaders [2]
- There are a number of reasons we want to manipulate our vertex data on the GPU rather than the CPU:
  - The GPU is highly optimized for Vector- and Matrix-based operations. The CPU is not.
  - Because these operations operate independently from the CPU, our CPU is now free for other operations.
  - Vertex Shaders are essentially little programs that run against every vertex in the data we’ve sent down the rendering pipeline.
  - We write vertex shaders using Microsoft’s High-Level Shader Language.

High-Level Shader Language [1]
- HLSL was originally developed as part of a joint partnership between Nvidia and Microsoft, which also was responsible for Cg.
- It sports a very C-like syntax, but also supports arbitrary sized vectors and matrices:
  - e.g., 4x4 matrix of floats can be defined by float4x4
  - 3-element vector of floats can be defined by float3
- We can also pass variables into our shaders from DirectX as externs.
High-Level Shader Language [2]

- HLSL also sports two vector component namespaces, which provide helpful shorthand when dealing with positions and colors: xxyzw and rgba
  - e.g., color.r is equivalent to color[0]
  - position.y is equivalent to position[1]
- We can also #define our vectors – access more than one component in a single call
  - e.g., position.y is equivalent to position[1]
- Finally, HLSL provides a lot of useful functions – you can read the API documentation for more

Uses for Vertex Shaders

- Vertex shaders are primarily used to transform our vertices.
- Some Examples:
  - Dynamic deformation of models
  - Displacement Mapping
  - Billboards
  - Fluid modeling
  - Particles
- Vertex shaders can be further combined with other shader types for a wide variety of possible effects

Geometry Shaders [1]: Implementation

- Geometry Shaders are a new feature for DirectX 10
- They aren’t supported by the Xbox 360, and most computers at the moment
- But they offer some exciting possibilities for the near future

Geometry Shaders [2]: Uses

- The primary purpose of a Geometry Shader is to create new geometry using the vector data supplied from the Vector Shader
- This means they can be used for
  - Generating point sprites
  - Extruding shadow volumes (to use in the pixel shader)
  - Further tessellation operations – i.e. increasing the level of tessellation in models (more triangles = smoother models)
  - Subdivision surfaces (e.g., terrain)
  - Single-pass cube map generation

Rasterization Revisited: Per-Pixel Operations

- Pixel Shaders allow further transformation on the rasterized pixel data
- Like Vertex Shaders, Pixel Shaders are written in HLSL, use the same syntax and function libraries, and can have variables set by DirectX

Pixel Shaders [1]: Implementation
Pixel Shaders [2]:

- Pixel Shaders can be used in a number of ways:
  - To control how multiple textures are blended when texturing a surface
  - To alter the color of a material
  - For customized lighting
  - To apply specular highlights

“Managed” Direct3D

- Direct3D offers the programmer a lot of control, but careful attention must be paid to memory and resource management.
- Microsoft piloted a “Managed” DirectX effort at one point. This, combined with the library they developed for the original X-Box, developed into XNA, a shared library, framework, and IDE for building game applications in C#.

Direct3D vs. OpenGL

- Most people find OpenGL easier to learn to start with.
- OpenGL uses standard C interfaces.
- Direct3D has a steeper initial learning curve and is based on C++ interfaces (COM).
- It is more difficult using OpenGL to do lower level operations than Direct3D. However, this does mean you are less likely to crash an OpenGL app than a Direct3D one.
- The vast majority of PC games are written using Direct3D.
- OpenGL can run on multiple platforms whereas as Direct3D is Windows based.
- Direct3D is updated frequently (every 2 months), allowing a standard interface to new features sooner than they appear in core OpenGL. However, OpenGL has an extension mechanism allowing immediate access to new features as graphics manufacturers create them, rather than having to wait for a new API version to appear.
- Direct3D has a lot of helper functions for common game coding issues (the D3DX set of functions)
- In terms of performance there is little difference between the two.
Preview:
Pixel & Fragment Shaders

Next Time:
Texture Mapping!

Summary
- Last Time: Intro to Illumination and Shading
  - Local vs. global models
  - Illumination (vertex shaders) vs. shading (fragment/pixel shaders)
  - Phong illumination equation: Introduction to shading
  - Previously: Classical Fixed-Function Pipeline
  - Today: Shaders in Modern Pipeline
    - Vertex shaders
      - Input: per-vertex attributes (e.g., object space position, normal)
      - Output: lighting model terms (e.g., diffuse, specular, etc.)
    - Pixel shaders
      - Input: output of vertex shaders (lighting aka illumination)
      - Output: pixel color, transparency (R, G, B, A)
  - Hardware Rendering: Application Programmer Interfaces (APIs)
  - Next: Shader Languages – (O)GLSL, HLSL / Direct3D, Renderman

Terminology
- Direct3D: Graphics Component of Microsoft DirectX Library
  - Similarities to OpenGL: C/C++, 3-D polygons-to-pixels pipeline
  - Differences vs. OpenGL: Windows, COM, some lower-level ops
- Classical Fixed-Function Pipeline (FFP): Per-Vertex Lighting, MVT + VT
- Modern Programmable Pipeline: Per-Pixel Lighting
  - Vertex Shaders (FFP and Programmable)
    - Input: per-vertex attributes (e.g., object space position, normal)
    - Output: lighting model terms (e.g., diffuse, specular, etc.)
  - Pixel Shaders (Programmable Only)
    - Input: output of vertex shaders (lighting aka illumination)
    - Output: pixel color, transparency (R, G, B, A)
- Shader Languages
  - Domain-specific programming languages
  - Geared towards hardware rendering
  - (O)GLSL, HLSL / Direct3D, Renderman