Intro to OpenGL
Rendering Objects

- Object has internal geometry (Model)
- Object relative to other objects (World)
- Object relative to camera (View)
- Object relative to screen (Projection)

Need to transform all geometry then draw…
The Graphics Pipeline

- Raytracing pipeline is too slow
  - Raytracers are irregular applications (difficult to parallelize)
- Better-looking ray tracers require numerous samples to converge
- Raster pipeline optimizes local light transport
  - Designed to accelerate rendering process
  - Focused on high throughput and parallelization
Rasterization

Objects composed of vertex data
Vertex data *tessellated* into primitives
Rasterization

Primitives have color and position

Color pixels on screen based on primitive projections

Embarrassingly parallel with great hardware support!
OpenGL

Open Graphics Library

• Standardized in 1992 by Silicon Graphics
• Currently managed by Kronos Group

Microsoft equivalent is DirectX
Simplified Graphics Pipeline

Application

Vertex batching & assembly

Clipping

NDC to window space

Rasterization

Fragment shading

Depth testing

Color update

OpenGL API

NDC = Normalized Device Coordinates, this is a \([-1,+1]^3\) cube

Depth buffer

Framebuffer
A Little Expanded...

Application

Vertex batching & assembly

Lighting

Texture coordinate generation

Triangle assembly

Vertex transformation

User defined clipping

View frustum clipping

NDC to window space

Perspective divide

Back face culling

Triangle rasterization

Fragment shading

Depth testing

Depth buffer

Color update

Framebuffer

Framebuffer
Old vs Modern OpenGL

Originally OpenGL was a “Fixed Function” Pipeline

• Exposed graphics hardware through user configurations
• Built-in math operations manipulate data accordingly
Old vs Modern OpenGL

OpenGL 3.0 is programmable allowing for greater flexibility and control.

Also changes hardware pipeline and how a programmer interacts with the GPU.
The modern rendering pipeline (blue stages are fully programmable)
Vertex Specification

Specify vertices GPU should process
  • One vertex/triangle at a time is slow
Specify how to process
  • **Attributes** inform vertex shader what data represents
Vertex Buffer Objects (VBOs)

• Source of data for vertex arrays
• `glBindBuffer` binds given buffer to global target
  • `GL_ARRAY_BUFFER` specifies Buffer Object is vertex attribute data
• `glVertexAttribPointer` specifies attribute data for these vertices
  • i.e. what are the data components and how are they arranged?
VBO Data

Contain data for:

- Vertex position
- Vertex colors
- Texture info
- Normal info
- etc
Vertex Array Objects (VAOs)

- OpenGL Objects associated with an OpenGL context (state of the instance)
- Stores attribute data and Buffer Objects for bussing to GPU
  - Can contain multiple VBOs
- VAOs allow switches between vertex attribute configurations without performance hit
- `glGenVertexArrays` creates VAO
- `glBindVertexArray` binds that VAO to target
Using VAOs

1. Create VAO with necessary information:
   1. Create VAO
   2. Bind VAO
   3. Generate and bind VBO
   4. Disable/unbind VAO and VBO

2. Rendering using VAO:
   1. Bind VAO
   2. Draw data in VBO
   3. Unbind VAO
Coordinate Systems

object → model matrix → world

model matrix

world → view matrix → camera

view matrix

camera → perspective matrix → normalized device

normalized device
Camera Coordinates

Note: Look down negative z direction
Normalized Device Coordinates

Note:
X and Y map to screen width and height
Z used for depth (deeper points are higher)
Except...

Screen coordinates use a different system!
Also...

`glViewport(x, y, width, height)` transforms NDC to window coordinates.

Allows for an aspect ratio in final display to screen after being normalized.

Incidentally, \((x, y)\) specifies the _lower_ left corner of the viewport.
Framebuffer

Memory region containing pixel data
Controlled by GPU

Layers:

• Color buffer (RGB)
• Depth buffer (Z axis position)
• Stencil buffer (extension of depth buffer)
Displaying a Framebuffer

CRTs: beam sweeps across screen to draw pixels (one pass every 1/60 secs)

LCDs: grab framebuffer (every 1/60 secs)
Flickering and Tearing

Framebuffer changes while monitor draws

How to solve?
When to Draw

On CRTs: wait for vertical retrace to swap
- “VSync”
- Occurs 1/60 sec
- Introduces lag

On LCDs: swap when not reading
Double-Buffering

Use two frame buffers

Render to **back buffer** while showing **front buffer**

Then swap
Triple Buffering and Beyond

Triple buffering can be used in conjunction with VSync to reduce double-buffering latency with less tearing than VSync.

Can also queue up $n$ frames generalizing notion of “double” or “triple” buffering.
Side Note: G-Sync and FreeSync

G-Sync (NVidia) and FreeSync (AMD) improve upon VSync by synchronizing refresh rates with frame rate.

Solves for VSync issues where fluctuating frame rates create tearing.
OpenGL Tutorial

Work through:

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/