A Brief Introduction to OpenGL

Sean Middleditch
DigiPen Game Engine Architecture Club

All About Me

Sean Middleditch
DigiPen RTIS Senior

http://seanmiddleditch.com/

All About This Presentation

Some history and an idea of what OpenGL is

When and why you should or shouldn't use it

Hints on getting started

Not a how-to or tutorial

Past and Present

Description and history of OpenGL

What is OpenGL?

C API for accessing graphics hardware

Polygonal rendering model

Fixed or programmable pipeline

Low-level hardware abstraction

What isn't OpenGL?

Not a scene management API or "engine"

Not a movie/photorealistic rendering API

Not a raytracer

No sound, input, physics, etc.

Why is OpenGL Good?

Very simple API for small/prototype projects

Only API for iOS, Android, HTML5, OS X, Linux

Vendor extensions (sometimes get new features before D3D gets them)

Can choose fixed function or programmable

Why is OpenGL Bad?

API is ancient, clunky, difficult, error prone

Windows drivers are low quality vs D3D drivers

Less tools and existing tools are not great

No XBox360 support, limited PS3 or Wii support, requires user to install extra drivers on Windows to get full support

OpenGL History

Original version of OpenGL came out in 1992

Design for fixed function SGI hardware way beyond consumer graphics hardware capabilities

Only a few years after ISO C89 was released, and 5 years before ISO C++98 was released

Five Generations of GL

There are roughly five different "generations"

OpenGL 1.x - fixed function
OpenGL 2.x - early programmable
OpenGL 3.x/4.x - modern programmable,
    Core profile and deprecation
OpenGL ES 1.x - mobile fixed function
OpenGL ES 2.x - mobile programmable

OpenGL 1.x

Fixed function

Textured polygons, lighting, fog

3D games were all software rendered at this time

Quake vs QuakeGL, etc.

OpenGL 2.x

Programmable pipeline

ARB assembly and GLSL programs

Designed by 3DLabs with "future looking" functionality (features hardware couldn't do)

Same crufty API, just added new functions

OpenGL 3.x / 4.x

Tons of new features, D3D10/11 capabilities

Added Core Profile and deprecation, which optionally breaks compatibility and forces programmable pipeline mode (OS X forces Core Profile to get 3.x features, everyone else enabled Compatibility Profile by default)

Longs Peak controversary

OpenGL ES 1.x

Stripped down, cleaned up OpenGL 2.x

Fixed function only

Designed for early mobile graphics hardware

OpenGL ES 2.x

Stripped down, cleaned up OpenGL 3.x

No deprecated functions from older versions

Programmable pipeline only, GLSL ES shaders mandatory to draw anything

The Future

OpenGL 5.x... direct state access?

OpenGL ES 3.x... more full OpenGL 3 features?

Highly unlikely that the API will be changed or improved in any way

Fixed vs Programmable

What shaders are and what they do

Fixed Function

"Fixed function" means the math is hardcoded

Lighting model, fog, texturing, etc.

Features can be turned on or off, some values can be tweaked, but otherwise every app must render the same way

Programmable

Originally, allowed small lighting model changes (cel shading, for example)

Modern hardware allows almost the entire pipeline to be replaced by user-written code

Highly customizable lighting, texture effects, and even geometry modification

Remnants of Fixed Function

Even modern programmable hardware still has fixed-function parts

- Triangle rasterizer, perspective correction
- Texture filtering
- Early depth testing
- Blending and alpha cut-off
- Various other bits

Why Fixed Function Today?

Fixed function is easier

All the advanced stuff is taken care of for you

Most simple rendering doesn't need any special shaders to work well

Hardware is cheaper (Wii, 3DS, dumb phones)

OpenGL & Fixed Function

OpenGL 1.x and OpenGL ES 1.x are fixed function only

OpenGL 2.x retains fixed function

OpenGL 3.x Core Profile and OpenGL ES 2.x do NOT have any fixed function support

(Direct3D 7 is last fixed function only version)

OpenGL & Programmable

OpenGL 2.x added programmable pipelines

OpenGL 3.x Core Profile and OpenGL ES 3.x REQUIRE programmable pipelines

OpenGL 2.x and 3.x Compatibility Profile allow optional or mixed usage of programmable

(Direct3D 10 and up require programmable)

OpenGL ARB Programs

ARB assembler programs were original OpenGL programmable pipeline language

Similar to D3D assembler shaders

Still used sometimes for old hardware support (usually through Cg)

OpenGL Shading Language

Commonly called GLSL

High level C-like language for shaders

Similar in scope to HLSL sans FX language

New version with each GL version, deprecates old language features/syntax frequently

NVIDIA Cg

NVIDIA's Cg language is an HLSL-like language

It can compile down to HLSL, GLSL, ARB assembly, a few others

Useful for abstracting between D3D and OpenGL

Does NOT support OpenGL ES (yet)

Using OpenGL 1

Getting started with the basics

Warning Warning Warning

The following slides are advice on OpenGL 1.x

This API is great for simple, small apps and games that don't need high performance

The API should NOT be used in "real engines"

But seriously, it's totally cool to use for most Sophomore DigiPen games and saves time

1. Creating a Context

Step 1 is to create a "context"

Equivalent to an ID3D9Device object

Easy option: use GLUT, GLFW, SFML, SDL

GLUT is the easiest (in my opinion), SDL is most popular (based on no statistical evidence), SFML is cleanest (so sayeth the Internet)

2. Projection Matrix

Step 2 is to create a projection matrix

3D (and 2D) pipelines rely on a matrix to transform coordinates in the game world to points on the screen

OpenGL uses a matrix stack to manage this

GLU contains helper functions for projection

3D Perspective Matrix

Select the matrix stack you want to modify

```
glMatrixMode(GL_PROJECTION);
```

Set the top element to the identity matrix

```
glLoadIdentity();
```

Multiply the top element with a perspective mtx

```
gluPerspective(fov, aspect, near, far);
```

2D Orthographic Matrix

Select the matrix stack you want to modify

```
glMatrixMode(GL_PROJECTION);
```

Set the top element to the identity matrix

```
glLoadIdentity();
```

Multiply the top element with a perspective mtx

```
gluOrtho2D(0, width, height, 0);
```

Matrix Stack - Model View

glMatrixMode(GL_MODELVIEW);

glTranslate, glRotate, glScale

Otherwise, exact same as perspective

Matrix Stack

glMatrixMode selects active stack

Use glPushMatrix and glPopMatrix

glLoadIdentity and glLoadMatrix replace top

Pretty much every other matrix function modifies the existing top entry on the active stack

3. Rendering Triangles

Step 3 is to render something

```
glBegin(GL_TRIANGLES);
    glVertex3f(1.f, 1.f, 1.f);
    glVertex3f(1.f, 2.f, 3.f);
    glVertex3f(2.f, 1.f, 3.f);
glEnd();
```

Understanding Drawing

glBegin must be called to start drawing

glBegin species how vertices are interpreted - are they making triangles, lines, or points?

glEnd finalizes the drawing

Many OpenGL functions illegal to call between glBegin and glEnd

Vertex Attributes

A vertex is submitted for rendering when glVertex is called

Other attributes are part of a drawing state

glColor, glTexCoord, glNormal

Vertex Function Suffixes

glVertex3f - vertex with 3 floating point components

glColor4ub - color with 4 unsigned byte components

Basically, number of components, then type of components

4. Texturing

Step 4 is adding texturing to triangles (optional)

OpenGL gives you no help in loading textures from disk

SDL, GLFW, SFML all have helper functions

Can also use libpng (yuck) or another helper lib

Creating Textures

Textures must be created

All objects in OpenGL have integer handles called "names" in GL parlance

```
GLint tex;
glGenTextures(1, &tex);
```

Binding Textures

OpenGL objects must be "bound" to use them

glBindTexture(GL_TEXTURE2D, myTex);

Must be called before calling any other function that modifies a texture, or any function that renders with a texture

Uploading Texture Data

Once you have the texture loaded, use glTexImage to upload the data

Must specify exact format of data

```
glTexImage2D(GL_TEXTURE2D, 0,
GL_RGBA8, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, ptrToPixelBuffer);
```

Explaining the Tricky Bits

The seventh is the layout of components, e.g. GL_RGBA, GL_RGB, GL_RGBA, etc.

The eighth is the size of the components

The third is the "internal format" GL should use

Third used to be number of components, but that use is deprecated

Using Textures

Enabling texturing

```
glEnable(GL_TEXTURE2D)
```

Passing texture coordinatees

```
glTexCoord2f(u, v);
```

glTexCoord affects the next glVertex call

Cleaning Up Textures

When done using a texture for now, unbind it

```
glBindTexture(GL_TEXTURE2D, 0);
```

When done using a texture forever, delete it

```
glDeleteTextures(1, &myTex);
```

5. Flipping Buffers

Step 5 is to flip the back buffer to the front buffer

All modern GL stacks effectively require double buffering

The swap buffer command is not in OpenGL!

D3D has DXGI, OpenGL has WGL, AGL, GLX, EGL, and SDL and friends have wrappers

Samples for Flipping

wglSwapBuffers()

SDL_SwapBuffers()

glfwSwapBuffers()

mySfmlWindow->swapBuffers()

6. Clearing the Screen

Step 6 is to clear the screen

Must be done so your next frame doesn't look like garbled trash

```
glClearColor(0.f, 0.f, 0.f, 1.f);
glClear(GL_COLORBUFFER_BIT);
```

Depth Testing

Depth testing is not turned on by default

For 3D, you almost certainly want this
For 2D, it's probably not that helpful

```
glEnable(GL_DEPTH_TEST);
glClear(GL_COLORBUFFER_BIT |
    GL_DEPTHBUFFER_BIT);
```

2D Sprites

Sprites are just quads (GL_QUADS)

Rendering them is the same as rendering any other shape in OpenGL

Render all points with the same z value

Transparency in Sprites

Sprites generally have transparent parts

Transparent: 100% see-through
Translucent: 1-99% see-through
Opaque: 0% see-through

Transparent is much faster than translucent

Alpha Test

Transparent is faster thanks to alpha testing

Alpha testing is mandatory if using depth testing!

```
glEnable(GL_ALPHA_TEST);
glAlphaFunc(GL_GREATER, 0.f);
```

Modern OpenGL

Using OpenGL 2.x/3.x

Vertex Buffers

Using glBegin/glEnd is slow

Vertex buffers replace them

glGenBuffers, glDeleteBuffers
glBufferData
glVertexAttribPointer
glDrawArrays

Shaders

Once you get into shaders, there's little more to OpenGL itself

GLSL is where all the action is

And that... is another lecture

Shader Functions to Know

glCreateProgram
glCreateShader
glCompileShader
glAttachShader
glLinkProgram
glDeleteShader
glDeleteProgram

Types of Shaders

Two parts in OpenGL: Program and Shader

A Program is what the hardware runs

A Shader is a function for a single stage
(Vertex Shader, Fragment Shader, etc.)

OpenGL shaders are non-separable until 4.x

Creating Shaders

Need a Program, and two Shaders

```
GLint program, vertexShader, fragmentShader;
program = glCreateProgram();
vertexShader = glCreateShader(
    GL_VERTEX_SHADER);
fragmentShader = glCreateShader(
    GL_FRAGMENT_SHADER);
```

Loading Shader Code

Must write your own function to load shader source files from disk (fopen(), iostreams, etc.)

```
glShaderSource(vertexShader, 1,
    (GLchar**)&vertexSource,
    strlen(vertexShader));
```

Compiling a Shader

Compiling a shader doesn't actually do a whole lot in OpenGL, just checks basic syntax

glCompileShader(vertexShader);

Checking for Errors

```
GLint status;
glGetShader(vertexShader,
    GL_COMPILE_STATUS, &status);
if (!status) {
    GLchar log[1024];
    GLsizei length;
    glGetShaderInfoLog(vertexShader,
        sizeof(log), log, &length);
    printf("Error %.*s\n", length, log);
}
```

Attaching Shaders Together

Shaders from each stage must be linked together into a Program

Once a shader is attached, it can be deleted if it won't be linked to any other Programs, since they're internally reference counted

```
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);
```

Linking the Program

Linking is when the program is actually compiled, and this is where the GPU machine code is generated

glLinkProgram(program)

And Check for Errors

```
GLint status;
glGetProgram(program,
    GL_LINK_STATUS, &status);
if (!status) {
    GLchar log[1024];
    GLsizei length;
    glGetProgramInfoLog(program,
        sizeof(log), log, &length);
    printf("Error %.*s\n", length, log);
}
```

Using the Program

Applying the shader program is very simple

```
glUseProgram(program);
```

And to stop, equally simple

```
glUseProgram(0);
```

Vertex Attributes

Shaders have custom vertex attributes, and modern shaders require the programmer to explicitly specify even the "built-in" attributes like position

There have been several ways to do this, ranging from awful to good

Vertex Attributes V1

Built-in attributes, glColor or glTexCoord

Or the slightly better way, glVertexPointer, glTexCoordPointer, glColorPointer, glNormalPointer, etc.

Vertex Attributes V2

Custom attributes are identified by a numeric index, passed to glVertexAttribPointer()

These indices are retrived AFTER linking by glGetAttribLocation()

But this function can return "unknown" even for attributes that exist in your shaders (if unused) !

Vertex Attributes V3

The programmer can explicitly specify an attribute index, which is then always a valid index

Call glBindAttribLication() BEFORE linking

This is a vastly superior alternative to glGetAttribLocation(), but sadly it seems nobody knows about this

Vertex Attributes V4

Specify the attribute indices directly in the shader, like HLSL has done for years

Requires OpenGL 3.3+ (sigh)

In the shader:

```
layout(location=0) in vec4 Position;
```

Uniforms

Uniforms are "programmable constants"

Again, several ways to set them

Uniforms V1

Get the uniform's index with glGetUniform() AFTER linking

There is no version to call before linking

Use one of the glUniform*() functions to set value, naming is similar to the glVertex*() functions from ealier, e.g. glUniform1f()

Uniforms V2

Use uniform buffers, like constant buffers in D3D

This is a pain in the butt, and only supported on GL 3.1+ (sigh)

Framebuffer Objects

OpenGL equivalent of D3D's Render Targets

Huge source of errors

Can attach textures to render into or render buffers (which are textures you can render into but then throw away... basically just a bad API for depth buffers)

Create the Framebuffer

```
GLint fbo;
glGenFramebuffers(1, &fbo);

glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

Attach a Texture

You must have created the texture, and you must have given it a dimension!

glTexImage2D(), but pass NULL for the data pointer

```
glFramebufferTexture2D(GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0,
    GL_TEXTURE2D, myTex, 0);
```

Create a Renderbuffer

Used for depth buffer, not much else

Depth buffers can also be regular textures, in the event that you want to use it for a later stage (like deferred shading)

```
GLint depthBuffer;
glGenRenderbuffers(1, &depthBuffer);
```

Attach a Renderbuffer

```
glBindRenderbuffer(GL_RENDERBUFFER,
    depthBuffer);
glRenderBufferStorage(GL_RENDERBUFFER,
    GL_DEPTHSTENCIL24_8,
    width, height);
```

IMPORTANT: Width and height must be the same as the color buffer texture's size!

Using a Framebuffer

Bind a framebuffer to use it

```
glBindFramebuffer(GL_FRAMEBUFFER,
    fbo);
```

To draw to the screen again, just unbind it

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Check for FBO Errors

Bind the FBO and then call glCheckFramebufferStatus

It will return GL_FRAMEBUFFER_COMPLETE if things are correct

If it fails, it probably means your textures and renderbuffers are not the same sizes

OpenGL vs D3D

Vague answers to the inevitable question

OpenGL vs Direct3D

- Direct3D is easier for medium to difficult stuff
- Direct3D drivers are better
- Direct3D has better tools
- HLSL less annoying than GLSL

- OpenGL is easier for simple stuff
- OpenGL runs on more platforms
- OpenGL has more jobs (mobile/GLES)

OpenGL's Problems

The OpenGL API has severe problems with error reporting and debuggability

The OpenGL API is not thread-friendly

The OpenGL API is error prone

The OpenGL API screws up Intellisense

Direct3D's Problems

Not available anywhere except Windows / XBox

... Some people hate Microsoft, I guess

OpenGL Complaints #1

OpenGL uses an integer to identify objects, so it's easy to pass an object to completely incorrect functions

OpenGL confuses Intellisense, thanks both to the integer naming and the way that OpenGL functions are defined in the header files

OpenGL Complaints #2

OpenGL has very weak error reporting, making it difficult to tell what you did wrong (and the API's design means you will do a lot of things wrong)

OpenGL's tools, like gDEBugger, are lacking in features and are buggy

OpenGL Complaints #3

OpenGL objects are mutable when they shouldn't be, which makes it easy to screw things up, and imposes some performance penalties

OpenGL requires many sequential function calls to change even simple things, any one of which could fail or cause later functions to fail

OpenGL Complaints #4

OpenGL's API is based on a state machine, which means that function calls affect what following functions do

This also requires the glBind*() function calls, which are a huge source of confusion, errors, and performance problems even for advanced users

OpenGL Complaints #5

OpenGL provides only a small number of helper functions, all math related, so even basic tasks like loading a texture or a shader requires third-party libraries or error-prone user-written code

The helpers that OpenGL provides are often slow or incomplete, and best avoided in any non-trivial game engine or renderer

OpenGL Complaints #6

OpenGL's state machine and weak OS bindings make threading nearly impossible to do, and when it is done the result in clunky and difficult to use

OpenGL's weak deprecation model means features and limitations from the 1980's negatively affect what the API does or how it works today

OpenGL Complaints #7

The drivers are crap (slow)

The drivers are crap (buggy)

The drivers are crap (old GL versions)

Intel's driver is exceptionally bad crap (all the above times 10)

Summary of Comparison

If you know you're targeting PC's, use D3D

If you know you're targeting mobile, use OpenGL (ES)

Big AAA engines need to support D3D, OpenGL (ES), and console-specific APIs, so they must abstract everything away anyway

More Resources

Getting help with OpenGL

Internet Resources

http://www.arcsynthesis.org/gltut/

http://www.opengl.org/discussion_boards/activity.php

http://gamedev.stackexchange.com/

http://openglbook.com/

http://www.gamedev.net/forum/25-opengl/

Printed Resources

OpenGL SuperBible (5th Edition)

OpenGL ES 2.0 Programming Guide

Game Engine Architecture

OpenGL Programming Guide (8th Edition)

Thank You

Questions? Even "How Do I...?" Questions