04 GLSL Intro Lab

Lab 22/01/2015
Vertices and Fragments

- A vertex is a location in space
  - The 3D (or 4D) points that describe what we are modelling
- A fragment can be considered to be a potential pixel that carries with it information including its colour and location and depth information, that is used to update the corresponding pixel in the frame buffer
  - A proto-pixel
- Fragments are generated as part of the rasterization process (some models consider the processing of fragments to occur separately).
**Vertex Processor**

- **Input from:**
  - User-defined variables
  - Uniform variables (built-in or user defined)
  - Texture maps
- **Can also access**
  - Built-in constants
- **Output:**
  - Built-in special variables
  - User defined **varying** variables
  - Special vertex shader output variables
Vertex Shaders

- Allow us to define operations to be performed on vertices
  - Basically a short program in (usually) a C-like language
  - Executed on each vertex as it passes down the pipeline, while the shader is loaded.
  - have access to the OpenGL State

- If defined completely replaces the fixed function pipelines' processing
  - This means that for each vertex the vertex shader must output all the information that the rasterizer needs to do its job

- The inputs are the vertex itself and any application information from the calling application that affects rendering on a per-vertex basis
A Simple GLSL Vertex Shader

```cpp
void main (void)
{
    gl_position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

- GLSL includes its own data types such as mat4 which represents a 4x4 matrix
- The above code reproduces some functionality of the fixed pipeline.
  - positions the vertex based on the modelview matrix and performs a projection based on the Projection Matrix
  - it does not set colour or any other attributes (colour could later be set by the fragment vertex shade)
  - N.B. the variables prefixed with gl are part of the OpenGL state and need not be declared
Vertex Shader Architecture

- Every time `glVertex3f(...)` or some variant is called the shader executes
Fragment Processor

Special input variables:
- gl_FragCoord
- gl_FrontFacing

User-defined in variables:
- Normal
- ModelCoord
- RefractionIndex
- Density etc.

Built-in uniform variables:
- gl_DepthRange.near, etc.

User-defined uniform variables:
- ModelScaleFactor, EyePos, Epsilon,
  LightPosition, WeightingFactor1, etc.

Special output variables:
- gl_FragDepth

User-defined output:
- FragColor, etc.

Provided directly by application
Provided indirectly by application
Produced by rasterization
Produced by the fragment processor

Texture maps

Diagram showing the flow of data through the fragment processor with inputs, outputs, and variables.
Fragment Shader Architecture
A Simple Fragment Shader

- In GLSL they have the same syntax as vertex shader programs

```c
void main()
{
    gl_FragColour = gl_FrontColor;
}
```

- But they work slightly differently to a vertex program.
  - `gl_FrontColor` is not a value generated by the vertex shader, but is produced by interpolating the colours between vertices
Using Shader Programs

- The Shader Program needs to be loaded and linked from the OpenGL Application.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create (empty) shader objects by calling <code>glCreateShader</code></td>
<td></td>
</tr>
<tr>
<td>Provide source code for these shaders by calling <code>glShaderSource</code></td>
<td></td>
</tr>
<tr>
<td>Compile each of the shaders by calling <code>glCompileShader</code></td>
<td></td>
</tr>
<tr>
<td>Create program object by calling <code>glCreateProgram</code></td>
<td></td>
</tr>
<tr>
<td>Attach all the shader objects to the program by calling <code>glAttachShader</code></td>
<td></td>
</tr>
<tr>
<td>Link the Program by calling <code>glLinkProgram</code></td>
<td></td>
</tr>
<tr>
<td>Install the executable program as part of OpenGL’s current state by using <code>glUseProgram</code></td>
<td></td>
</tr>
</tbody>
</table>
Minimum Code

GLuint myProgObj;
myProgObj = glCreateProgram();
GLchar vertexProg[] = "my_vertex_shader_filename";
GLuint myVertexObj;
myVertexObj = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(myVertexObj, 1, vertexProg, NULL);
glCompileShade(myVertexObj);
glAttachObject(myProgObj, myVertexObj);
...
glLinkProgram(myProgObj);
glUseProgram(myProgObj);

- Once the above code has been run subsequent graphics primitives will be drawn with the shaders you provide rather than with OpenGL's defined fixed functionality pipeline
Shader Data Types

- Shaders can be considered small independent programs which can be loaded onto hardware.
- They have some of their own data types
  - float, bool, int
  - Vectors of length 2, 3 or 4
    - vec2, vec3, vec4
  - Matrices, 2x2, 3x3, 4x4
    - mat2, mat3, mat4
- To access the first 3 components (i.e. A vec3) of variable myVec defined as vec4
  - myVec.xyz
Attribute Types

- **Vertex Attributes**
  - Information which only apply to vertices.
  - *Generic vertex attributes*: e.g. normal, color, texture coords
  - Custom Attributes can also be declared.
  - Attributes can be read and set in the application and shader

- **Special Output Variables**
  - gl_Position, gl_PointSize, gl_ClipDistance

- **Uniforms** are variables which hold true across a whole primitive
  - Reserved prefix gl_
  - Cannot be updated within a vertex shader
Varying

- **Varying** variables are used to convey data from a vertex shader to a fragment shader

```cpp
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
varying vec4 color_out;
void main(void)
{
    gl_position = gl_ModelViewProjectionMatrix * gl_vertex;
    color_out = red
}
```

- The Corresponding Fragment Shader

```cpp
varying vec4 color_out;
void main(void)
{
    gl_FragColor = color_out;
}
```
Built in functions

- **Trigonometry** ... Sin cos tan, convert from degrees to radians etc
  - genType `sin (genType angle)`

- **Maths**
  - genType `pow (genType x, genType y)`
    - returns x to the power of y
  - genType `sqrt (genType x)`
  - `mod, min max floor etc`

- **Geometric**
  - vec3 `cross (vec3 x, vec3 y)`
  - float `dot (genType x, genType y)`
  - `distance, length, normalize face forwards`
  - genType `reflect (genType I, genType N)`
  - genType `refract (genType I, genType N, float eta)`

- For more see the GLSL spec
Built in variables

- **gl_Position**
  - Must be defined in every vertex shader
- **gl_FragColor**
  - Must be defined in every fragment shader
- **gl_Vertex**
  - Stores the position of the incoming vertex (in local coordinates)
- **gl_FrontColor**
  - The front color of the fragment
- **gl_ModelViewMatrix**
- **gl_ProjectionMatrix**
- **gl_ModelViewProjectionMatrix**
- **vec4 ftransform(void);**
  - replicates fixed function
Samplers [Not for this lab]

- For accessing textures
  - They basically create a pointer handle to a texture to use when looking up what texture colour is to be used
  - e.g. `sampler1D, sampler2D, sampler3D`
  - `samplerCube` - for cube map textures
  - `sampler1DShadow, sampler2DShadow` - for shadow maps

- Uniform sampler2D source

  ```
  vec2 texcoords(0.7, 0.5);
  gl_FragColor = texture2D(source, texcoords)
  ```

- There are many texture lookup functions such as `texture2D, texture3D` etc which take samplers as a parameter
Default Shader

//vertex shader
void main(void)
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

//fragment shader
void main(void)
{
    gl_FragColor = vec4( 0.4, 0.0, 0.9, 1.0 );
}
Diffuse Shader: VS

varying vec3 N;
varying vec3 v;

void main(void)
{
    v = vec3(gl_ModelViewMatrix * gl_Vertex);
    N = normalize(gl_NormalMatrix * gl_Normal);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
Diffuse Shader: FS

```glsl
varying vec3 N;
varying vec3 v;

void main(void)
{
    vec3 L = normalize(gl_LightSource[0].position.xyz - v);
    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
    Idiff = clamp(Idiff, 0.0, 1.0);

    gl_FragColor = Idiff;
}
```

From: http://www.opengl.org/sdk/docs/tutorials/Clockwork Coders/lighting.php
Phong Shader: VS

```c
varying vec3 N;
varying vec3 v;

void main(void)
{
    v = vec3(gl_ModelViewMatrix * gl_Vertex);
    N = normalize(gl_NormalMatrix * gl_Normal);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

From: http://www.opengl.org/sdk/docs/tutorials/Clockwork Coders/lighting.php
varying vec3 N;
varying vec3 v;

void main (void)
{
    vec3 L = normalize(gl_LightSource[0].position.xyz - v);
    // we are in Eye Coordinates, so EyePos is (0,0,0)
    vec3 E = normalize(-v);
    vec3 R = normalize(-reflect(L,N));

    //calculate Ambient Term:
    vec4 Iamb = gl_FrontLightProduct[0].ambient * gl_FrontMaterial.ambient;

    //calculate Diffuse Term:
    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0) * gl_FrontMaterial.diffuse;
    Idiff = clamp(Idiff, 0.0, 1.0);

    // calculate Specular Term:
    vec4 Ispec = gl_FrontLightProduct[0].specular * pow(max(dot(R,E),0.0), gl_FrontMaterial.shininess);
    Ispec = clamp(Ispec, 0.0, 1.0);

    // write Total Color:
    gl_FragColor = Iamb + Idiff + Ispec;
}
Aside: Varying Variables in GLSL

- **Varying** variables are used to convey data that needs to be interpolated from a vertex shader to a fragment shader:
  - defined at each vertex and interpolated across a graphics primitive to produce a perspective correct value at each fragment.
  - must be declared in both the vertex shader and the fragment shader with the same type.
  - output values from vertex shaders and the input values for fragment shaders.

Example vertex shader:
```glsl
const vec4 red = vec4(1.0,0.0,0.0,1.0);
varying vec4 color_out;
void main(void)
{
    gl_position = gl_ModelViewProjectionMatrix *gl_vertex;
    color_out = red
}
```

And the corresponding fragment shader:
```glsl
varying vec4 color_out;
void main(void)
{
    gl_FragColor = color_out;
}
```
The Lab

Basic GLSL
Basic Setup

- **Start up a Win32 Console Application in Visual C++ 201X**
  - In the application wizard, under “Application Settings” make sure you tick “Empty Project”

- **You will need,**
  - Best get the versions from here:
    - [https://www.scss.tcd.ie/Michael.Manzke/CS7055/gl](https://www.scss.tcd.ie/Michael.Manzke/CS7055/gl)
  - Download files and place them in the following locations:
    - Glut.h to C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include\GL
    - Glut.lib to C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\lib\Glut32.dll to C:\Windows\System32\ OR C:\Windows\SysWOW64\ (64 Bit)
    - bin/glew32.dll to %SystemRoot%\system32
    - lib/glew32.lib to C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\lib\include/GL/glew.h to C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include\GL
    - include/GL/wglew.h to C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include\GL
Sample Code

- Get the sample code from:
  - [https://www.scss.tcd.ie/Michael.Manzke/CS7055/Lab1](https://www.scss.tcd.ie/Michael.Manzke/CS7055/Lab1)
  - This includes
    - Base code:
      - Basic vertex shader: base.vert
      - Basic fragment shader: guraud.frag

- It should show you the basic setup for an OpenGL/GLUT program with GLSL.
- Also included is a code for a basic vertex shader and pixel shader
- Compile and run this.
Assignment 1

- Worth ~6%, should approximately take 5 hours.
- Set-up a OpenGL/GLUT program with GLSL and run some basic shaders.
- You should create a demo of a scene with a simple rotating object rendered with 3 different shaders.
- All the work is expected to be done in the shader, although you may pass in variables from the application. Note that you are not required to use textures (yet)
- Some examples:
  - Toon rendering
  - Basic Phong or Gouraud
  - Gooch Shading
  - Blinn-Phong Shading
- 80% for the lab requirements (see also next slide) and 20% will go for any novel use of the shader, any interesting shaders, placement in a more complex scene (N.B. try not to get too carried away)
Submission

- This is a small assignment worth ~6% of the module
- You should demo it briefly in the lab next week at 15:00 29th January.
- For all labs henceforth there will be a 20% penalty for each unexplained day late of the demo (you should notify me by email in advance of the deadline)
- Also make a short youtube video of it. Email me the link.

- At this stage, we just want to make sure you have everything set up to do this
- You can use a camera capture program like camstudio OR (better) find a library that will export GL frames directly to video or image sequences (some examples are GL2AVI or AVIGenerator)
Notes

- Almost everything you need is provided in the sample for the application program. You may need to pass in some values for some of the shaders.

- For further details you can follow the lighthouse3D tutorial at

- Some Builds Errors may result due to incompatible version of GLEW and GLUT libs/dlls present on your system. You may have to get these from respective sites and possibly recompile them for your individual setup.
Further things to look at [Optional]

- **Textures:**
  - we will look at this in a later lab but you may want to skip ahead if you have time.
  - Unfortunately OpenGL does not have image-file data structures or loaders so you have to find some yourself. E.g. DevIL
  - The last part of the lighthouse3D tutorial discusses what you need to do in OpenGL/GLSL to set this up.

- **Geometry:**
  - You might want to see if you can create/load your own 3D geometry using vertex primitives. Current practice is to use vertex buffer objects.
Resources

- Lighthouse 3D GLUT/GLSL Tutorials
  - http://www.lighthouse3d.com/opengl/glut/
  - http://www.lighthouse3d.com/opengl/glsl/

- GLEW
  - Automatically sets up whichever OpenGL extensions are available (i.e. Enables calls to shader hardware if it exists)
  - http://glew.sourceforge.net/

- nVidia OpenGL SDK
  - Contains Examples (which also use the GLEW library)

- DevIL – an open image library (e.g. for textures)
  - http://openil.sourceforge.net/