Rendering Geometry

Introduction to GLSL and Vertex Shaders

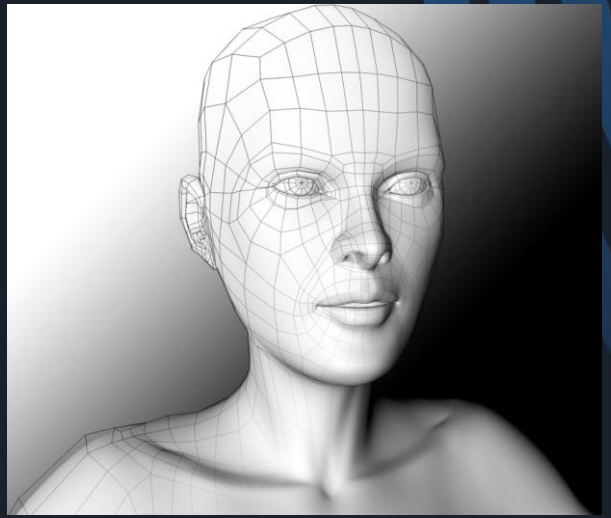
Programming – Computer Graphics

Contents

- Geometry
  - Vertex Buffers
  - Index Buffers
- Programming the GPU
  - GLSL Variables
  - GLSL Flow Control
  - GLSL Uniforms
  - GLSL Input and Output
- Vertex Shaders
  - Vertex Shader Uses

Geometry

- All rasterised computer graphics are created by projecting geometry onto the screen
  - There are other forms of CG rendering, but the most commonly used technique within games is still rasterisation
- We define geometry as a collection of primitive geometry elements
  - Points, lines, triangles
- Primitives then are used to create more complex forms
  - Surface topology for a mesh, such as a character or terrain
  - Smoke particles
- To create a complex mesh we must first understand these basic components



Vertex Buffers

- All geometry on the GPU is represented by a buffer of vertices
  - A Vertex is a structure that defines all the information at a point in a primitive
- Vertices can contain lots of information, but most importantly they usually hold the position of the point in 2D or 3D space
- Vertices are use to represent the shape of a primitive
- Primitives can be
  - Single points
  - A line consisting of two points
  - A triangle consisting of three points
  - Various subsets of the above types

```
struct Vertex {
  glm::vec4 position;
  glm::vec4 normal;
  glm::vec4 colour;
  glm::vec2 texCoord;
};
```

Example Vertex Structure

Primitives

- Primitives dictate the shape of the geometry represented by the vertex buffer

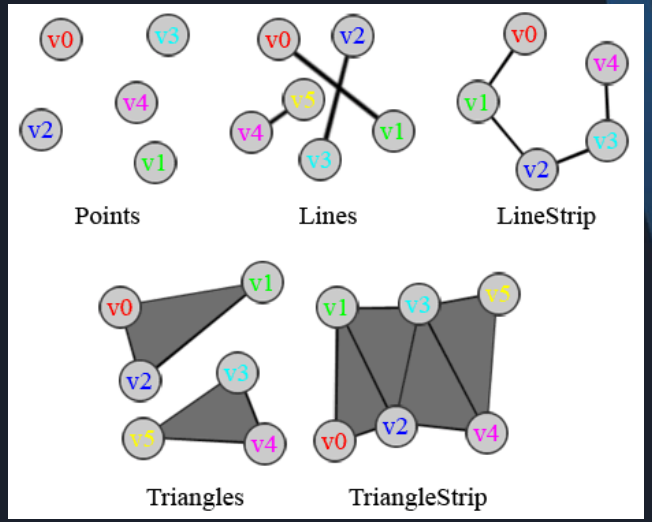
Points
  - Each vertex represents an individual point

Lines
  - Each pair of vertices represent a single line

Line Strip
  - The first two vertices represent a line, then every additional vertex uses the last to represent another line

Triangles (the most common in 3D meshes)
  - Every three vertices represent a different triangle

Triangle Strip
  - The first three vertices represent a triangle, then every additional vertex uses the last two to represent another triangle



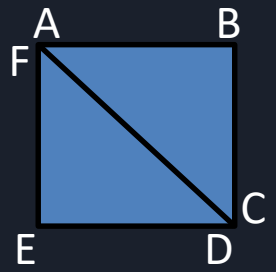
Index Buffers

- Complex geometry, such as a character mesh, can contain thousands of triangles
  - Using vertices we would need to have triple the amount of vertices as there are triangles
  - Many of the vertices would represent the exact same data!
  - Instead we can make use of an Index Buffer to help reduce the vertex data

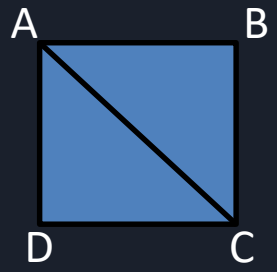
Index Buffers

- When using an index buffer, primitives are no longer defined by the vertex buffer's ordering
- Instead the vertex buffer merely represents the data at each point
- The index buffer then contains integers indexing vertices within the vertex buffer, and the primitive layout is instead defined by the index buffer ordering
  - This results in more elements but less data as an index is usually smaller that a vertex
  - Typically unsigned int vs a complex vertex structure

Vertex Buffer
A
B
C
D (same as C)
E
F (same as A)



Vertex Buffer	Index Buffer
A	A
B	B
C	C
D	C
	D
	A



Programming the GPU

- DirectX 8 era hardware gave developers the ability to write programs that could run on the GPU to replace certain stages in the pipeline
  - Programs for the GPU were called shaders as they typically effected the colouring of a scene
  - Initially just very basic vertex and fragment shaders using assembly code
- Games could now differentiate how they look and weren't restricted by the fixed pipeline
- Each GPU generation exposed more and more features and drastically increased the performance and abilities exposed to the developers

Programming the GPU

- The first shader programs were written in an assembly-style language

```
vs.1.4
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3
```

DirectX Assembly Shader performing a vector-matrix multiply transform with dot products

- Eventually higher-level languages were developed to allow programmers to write shaders using C-like code functions
  - Cg (C for Graphics) by Nvidia, used by many platforms and engines
  - HLSL (High-Level Shading Language) by Microsoft for DirectX
  - GLSL (OpenGL Shading Language) for OpenGL

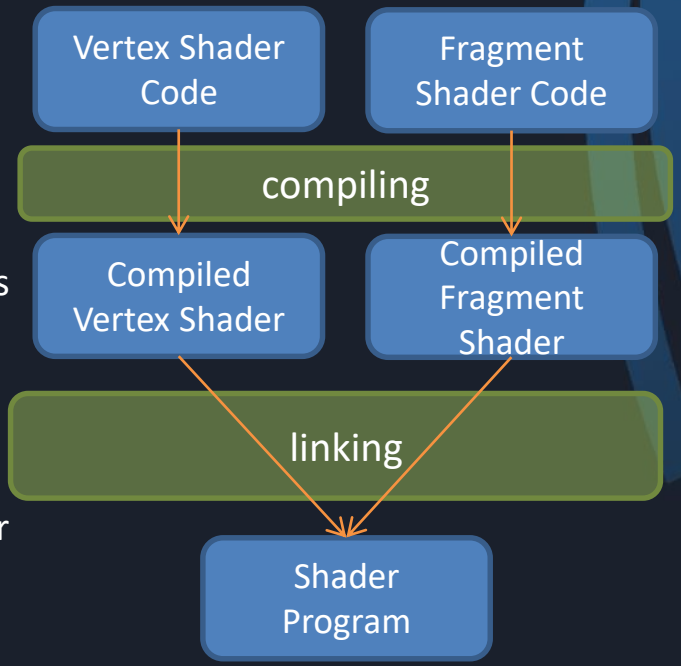
```
#version 410

in vec4 Position;

uniform mat4 ProjectionViewWorldMatrix;

void main( ) {
     gl_Position = ProjectionViewWorldMatrix * Position;
}
```

Performs the same task as the assembly code above, but in GLSL!

- Some shader versions even have C++ features, including classes and inheritance!

OpenGL Shading Language - GLSL

- GLSL is a very C-like language, including much of the same syntax, but with a few extra syntax tweaks
  - Written as text and then compiled into code that can execute on the GPU
- Compiled shader code for different programmable stages can then be linked together into a single shader program
  - A program then controls the entire render pipeline for any geometry drawn while the program is bound
  - Compiled shaders can be used in multiple programs
- Code for a shader stage consists of an entry point, similar to C's main() function, and any number of additional functions



GLSL Variables

- GLSL contains many of the same variable types as C++
  - bool, int, uint, float, double
- It also contains a few complex data types
  - vec2, vec3, vec4 (floating-point vector objects)
  - mat2, mat3, mat4 (floating-point NxN matrix objects, ie 2x2, 3x3, 4x4)
  - A few extra vector and matrix types exist
  - Vector types also have a feature called swizzling!
- Many of the standard C operators also exist
  - Except for pointer-related operators

```
// vectors have a union of x,y,z,w and r,g,b,a
vec2 v2 = vec2(0,0); // xy or rg
vec3 v3 = vec3(1,2,3); // xyz or rgb
vec4 v4 = vec4( v3, 0 ); // xyzw or rgba

dvec2 d2; // vector 2 of doubles
ivec3 i3; // vector of int

float f = v2[0]; // vectors can be used like arrays
f = v2.y;

// swizzling!
vec2 xx = v4.xx; // returns vec2( v4.x, v4.x )
vec3 zyx = v3.zyx;
vec4 rrra = v4.rrra;

mat2 m2;
mat3 m3;
mat4 m4;

mat3x4 m34; // 3 column and 4 row matrix

vec3 col1 = m4[1] * v3;

f = m4[1].w;
f = m4[1][3];

v4 = m4 * v4;
v4 = m4 * vec4( v3, 0 );
```

GLSL Structures and Const Variables

- GLSL can also make use of structs much like C

```
struct Light {
 vec4 position;
 vec3 colour;
};

Light myLight;
myLight.colour = vec3( 0, 1, 0 );

Light myOtherLight = Light( vec4( 10, 10, 10, 1 ), vec3( 1, 0, 0 ) );
```

- We can also create const variables that are usable within the shader and wont change
  - These variables are only accessible within the shader

```
const float PI = 3.14159f;
const vec3 RED = vec3( 1, 0, 0 );
```

GLSL Flow Control

- GLSL also contains some C-style flow control
  - if / else
  - for

```
float d = dot( v3, v4.xyz );

if ( d <= 0 )
      colour = vec4( 0, 0, 0, 1 );
else {
      colour = vec4( d, d, d, 1 );
}

for ( int i = 0 ; i < 5 ; ++i ) {
      colour.rgb += texture( diffuse, uv + vec2(-2 + i, 0 ) ).rgb;
}
```

- However GPUs aren't designed to handle dynamic branching very well
  - CPUs handle if / else / for fairly well, but a GPU is designed to run through thousands of mathematical operations rather than make dynamic decisions
  - Most loops and branches are unwound during compilation, which can make for slow cumbersome shaders
- Older hardware, especially mobile devices, sometimes do not support flow controls

GLSL Versions

- There have been many version of GLSL, each released with a version of OpenGL from OpenGL 2.0 onwards
  - OpenGL 2.0 had GLSL version 1.10 (110)
  - OpenGL 3.0 had GLSL 1.30 (130)
- From OpenGL 3.3 onwards the GLSL version matched the GL number, i.e. 3.3 and 3.30 (330)
  - Current OpenGL 4.5 has GLSL 4.50 (450)
- Each version added more and more features
- You can specify the version your shader file is using with a #version tag at the top
  - Not all features are supported with each version!

```
#version 410

in vec4 Position;

uniform mat4 ProjectionViewWorldMatrix;

void main( )
{
     gl_Position = ProjectionViewWorldMatrix * Position;
}
```

GLSL Uniforms

- Uniforms are global constant variables within shader code, that have their value set within the application code rather than the shader code
  - They are global in that any scope of the shader they are in can read them
  - They are constant in that they are read-only

```
#version 410

in vec4 Position;

uniform mat4 ProjectionViewWorldMatrix; // global constant!

void main( )
{
     gl_Position = ProjectionViewWorldMatrix * Position;
}
```

- Uniform variables are set in the application before a draw call is made
  - Within the application code we can access a handle to a uniform variables, similar to a pointer
  - We can then bind variables to their location that match the data type

- Uses for uniforms could include
  - Passing the shaders the colour of a mesh
  - Letting the shaders know the time for some animated effect

GLSL Input and Output

- Input to a shader is controlled with global variables with an in prefix
  - Input variables come from previous pipeline stages
  - In the case of the vertex shader the input comes from the vertex buffer
  - They don't need to be set like uniform variables do
- Output is usually controlled with a global variable with an out prefix
  - Output variables get passed on to the next stage in the pipeline, becoming input

```
#version 410

in vec4 VertexColour; // input to this example fragment shader from the previous stage

out vec4 PixelColour; // output from the shader goes to the next stage in the pipeline

void main( ) {
      PixelColour = VertexColour;
}
```

GLSL Input and Output

- In some cases there are pre-defined global variables
- For example, gl_Position is a pre-defined keyword for the output vertex position
  - This variable must be set at some point during the render pipeline before the Rasterisation stage, but not necessarily from the vertex shader!

```
#version 410

in vec4 Position;

uniform mat4 ProjectionViewWorldMatrix;

void main( ) {
     // stores the projected screen-space position in gl_Position for the Rasteriser
     // so it can plot the primitive's pixels
     gl_Position = ProjectionViewWorldMatrix * Position;
}
```

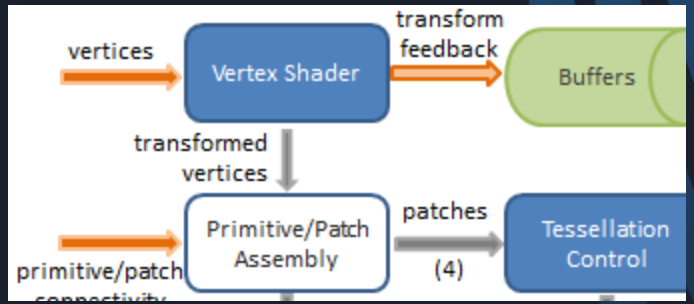
GLSL Functions

- Apart from the shader entry points, functions work very similar to C
  - They can have parameters
  - They can return variables
- There are no references, but a similar effect can be achieved with the out prefix or an inout prefix
  - out means the variable can be written to but not read from
  - inout means the variable has valid data that can be read, but it can also be written to
- There are also many built-in functions, most of which are math based

```
#version 410

uniform float totalTime;

in vec4 inColour;
out vec4 pixelColour;

vec4 warpColour1( vec2 colour )
{
	return colour * sin( totalTime );
}

void warpColour2( in vec4 original, out vec4 new )
{
	new = original * sin( totalTime );
}

void warpColour3( inout vec4 colour )
{
	colour *= sin( totalTime );
}

void main( )
{
	vec4 colour = warpColour1( inColour );
	warpColour2( inColour, colour );
	warpColour3( colour );
	pixelColour = colour;
}
```

Vertex Shader Stage

- Remembering back to a previous session, the Vertex Shader stage is a programmable stage
  - One of the first programmable stages available
  - First available on DirectX 8 era hardware
- Input to this stage consists of the vertices from the vertex buffer
  - It receives a single vertex at a time, but is completely parallel and can process thousands at once!



Vertex Shader Uses

- Commonly the vertex shader's job is to take the vertices for a mesh and transform them from Local / Model / Object Space into Screen Space
  - Vertices are usually in their own local space
  - The vertex is simply multiplied against a matrix to put it into global world space
  - From world space it is multiplied against a view space matrix, simply the inverse of the camera's global world space matrix
  - From view space it is multiplied against a screen space projection matrix

- These matrices are bound to the vertex shader as uniforms

```
#version 410

in vec4 position;

uniform mat4 worldMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

void main( ) {
      gl_Position = projectionMatrix * viewMatrix * worldMatrix * position;
}
```

Vertex Shader Uses

- When rendering, all the meshes in a 3D scene will typically use the same View Space and Screen Space
  - They each still have their own World Space
- What this means is we could pre-combine the view and projection matrices in our application and bind that to the shader as one matrix instead of two
  - This reduces how many matrix multiplies are run on the GPU
  - As an example, 50 objects with about 2000 vertices each would save us 99999 matrix multiplies
  - We could also combine the world matrix, but there are many uses for the world matrix in shaders so we can still include it separately

```
#version 410

in vec4 position;

uniform mat4 worldMatrix;
uniform mat4 projectionViewMatrix;

void main( ) {
      gl_Position = projectionViewMatrix * worldMatrix * position;
}
```

Other Vertex Shader Uses

- Just because the vertex shader commonly transforms vertex positions into Screen Space doesn't mean that's all it does!
  - We could manipulate and morph vertices

```
#version 410
in vec4 position;
uniform float time;
uniform mat4 worldMatrix;
uniform mat4 projectionViewMatrix;

void main( ) {
      vec4 warped = position * vec4( sin( time ), cos( time ), sin( time ), 1 );
      gl_Position = projectionViewMatrix * worldMatrix * warped;
}
```

- The vertex shader can also output more or less data than it receives, passing on data to later stages

```
#version 410
in vec4 position;
out vec4 colour;   // outputting colour that the shader generates
uniform float time;
uniform mat4 worldMatrix;
uniform mat4 projectionViewMatrix;

void main( ) {
      float pulse = sin( time ) * 0.5f + 0.5f;
      colour = vec4( pulse, pulse, pulse, 1 );
      gl_Position = projectionViewMatrix * worldMatrix * position;
}
```

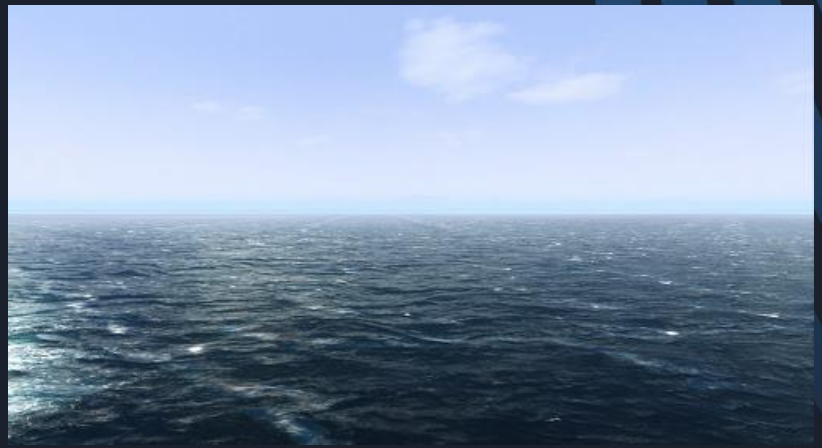
Other Vertex Shader Uses

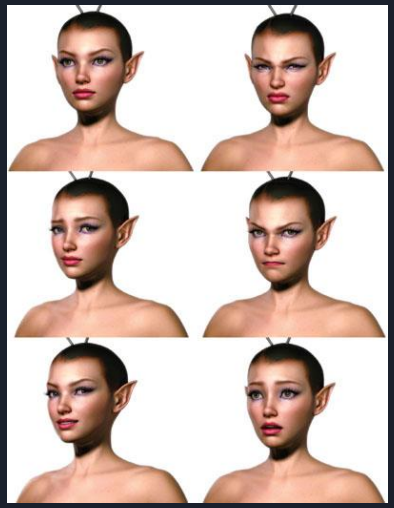
- In many instances the vertex shader is used to just pass data on to the rest of the pipeline
  - Certain Vertex Buffer elements might not be needed by the Vertex Shader, like Colour
  - Passed on data goes to the next programmable stage in the pipeline, whichever that may be

```
#version 410

in vec4 position;
in vec4 colour;
in vec2 texCoord;

out vec4 vColour;
out vec2 vTexCoord;

uniform mat4 worldMatrix;
uniform mat4 projectionViewMatrix;

void main( ) {
	vColour = colour;
	vTexCoord = texCoord;
	gl_Position = projectionViewMatrix * worldMatrix * position;
}
```

Other Vertex Shader Uses

- Vertex Shaders can be used for some very complex effects
  - Simulated ocean waves
  - 3D skeletal/morph animation
  - Lighting





Summary

- OpenGL Shading Language allows us to program the render pipeline with GLSL code
  - Other APIs exist, such as Cg and HLSL
  - Newer Vulkan API can be set up to use almost any shader language
- The Vertex Shader is the first programmable stage on the GPU
  - Typically transforms vertex positions into screen-space and passes data through to the other stages
  - Receives its input data from the vertex buffer

Further Reading

- Haemel, N, Sellers, G, Wright, R, 2014, *OpenGL SuperBible*, 6th Edition, Addison Wesley

- Wolff, D, 2013, *OpenGL 4 Shading Language Cookbook*, 2nd Edition, PACKT Publishing