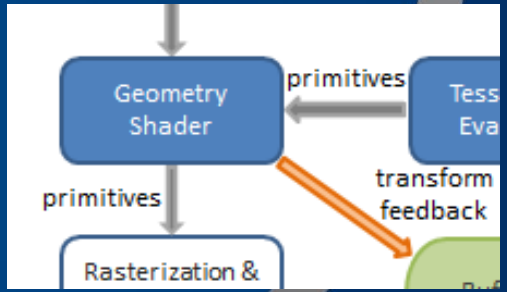
Geometry Shaders

Manipulating the world one primitive at a time

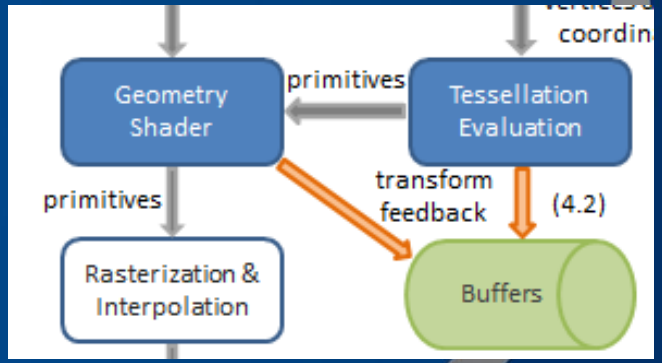
Geometry Shader Stage

- An optional programmable stage
- Is executed for each primitive in a mesh
- Receives the vertex data for all vertices used in the primitive
  - 1 for a point, 2 for a line, 3 for a triangle
- Can output more or less primitives than it receives
  - Can even change the type of the primitives!
  - And can output 0 primitives!
  - Does not automatically output the received primitive

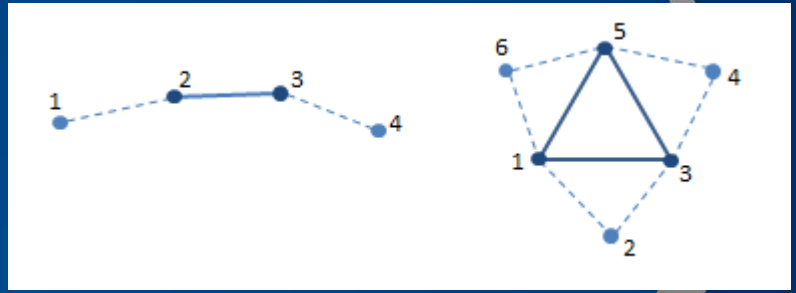


Geometry Shader Input and Output

- Despite there being multiple primitive types when rendering, the Geometry Shader only has 3
  - Points
  - Lines
  - Triangles
- It can also access adjacency information for primitives near the one being processed
  - Lines with adjacency and Triangles with adjacency
- It also has a limited set of output primitives
  - Points
  - Line Strips
  - Triangle Strips
- The Geometry Stage has one other advanced feature; it can output the processed vertices back to the application, without sending them to the Rasteriser to be drawn!



Geometry Shader Input and Output



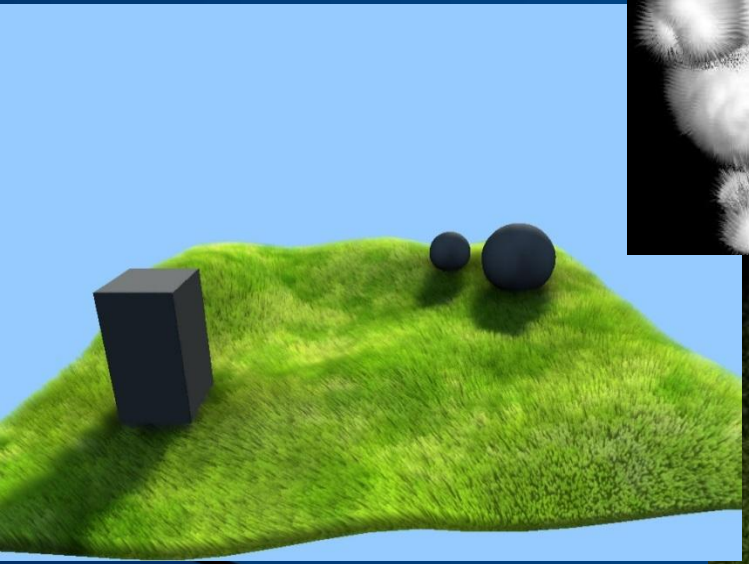
- Adjacency primitives are available when rendering primitives with adjacency information
  - These can be used instead of Line and Triangle primitives, but require more vertices per primitive

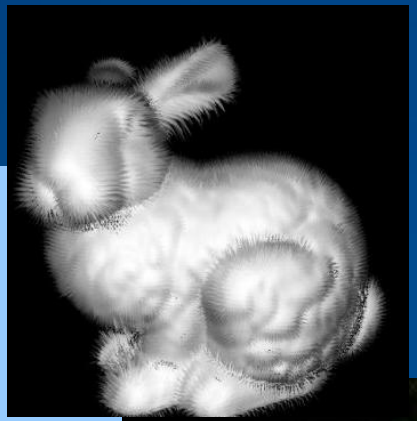
- The input adjacency primitives are as follows
  - Lines consist of 4 vertices rather than 2, with the 2nd and 3rd being the line itself, and the 1st and 2nd being the preceding line and 3rd and 4th being the following line
  - Triangles consist of 6 vertices with 1st 3rd and 5th being the triangle itself and the other points combine with the edges

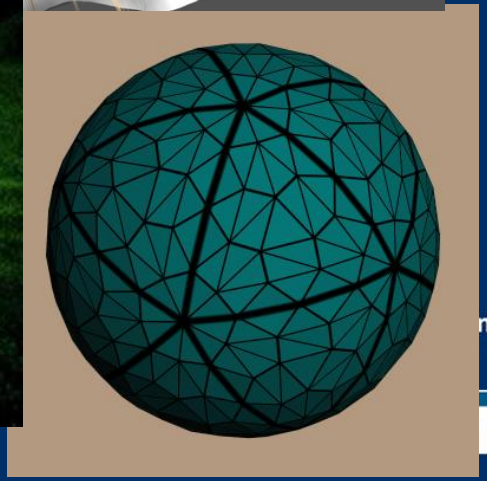
Geometry Shader Uses

- Before the Tessellation Stage was available people used to use the Geometry Shader for similar uses
  - Since it can output more primitives than it receives people could perform basic tessellation and displacement mapping
- Because it can output different primitive types than it receives, there are many other effects it enables
  - Simulating particles as point primitives in the Vertex Shader and having the Geometry Shader convert them to screen-aligned quads
  - Adding grass / hair / fur “fins” to create hairy geometry
  - Debug rendering features, such as colouring the edges of triangle primitives to add a wireframe overlays

Geometry Shader Uses







Writing Geometry Shaders

- Writing Geometry Shaders is again similar to the other shader stages, except that it requires a bit of extra work
  - You must define the input primitive layout and the output primitive layout
  - You must also define the maximum number of output vertices

```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
```

- During the shader you must notify when a vertex has been defined, and when it should end the primitive
  - **EmitVertex()** notifies the shader that elements of the current vertex have been set
  - **EndPrimitive()** notifies that a primitive is complete, such as a triangle strip, but more primitives can be defined in the shader after **EndPrimitive()**

Writing Geometry Shaders

- Here is an example of a Vertex Shader passing through the screen-space position
  - The Geometry Shader is receiving triangles
  - It is outputting triangle strips with a max of 3 vertices, so just 1 triangle

```
in vec4 position;

out vec4 vPosition; // output position to next stage, not rasteriser

uniform mat4 worldmatrix;
uniform mat4 projectionViewMatrix;

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

```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

in vec4 vPosition[];

void main( )
{
      gl_Position = vPosition[ 0 ];
      EmitVertex();
      gl_Position = vPosition[ 1 ];
      EmitVertex();
      gl_Position = vPosition[ 2 ];
      EmitVertex();
      EndPrimitive();
}
```

Writing Geometry Shaders

- Here is an example Geometry Shader turning points in to 2 triangles
  - The input vertices are still in local space so that the new triangles get converted to screen space
  - Since we're outputting triangle strips we only need to define 4 vertices

```
layout(points) in;
layout(triangle_strip, max_vertices = 4) out;

in vec4 vPosition[];

uniform float size;
uniform mat4 pvwMatrix;

void main( )
{
	float halfSize = size * 0.5f;
	gl_Position = pvwMatrix * (vPosition[0] + vec4( -halfSize, -halfSize, 0, 0 ));
	EmitVertex();
	gl_Position = pvwMatrix * (vPosition[0] + vec4( -halfSize, halfSize, 0, 0 ));
	EmitVertex();
	gl_Position = pvwMatrix * (vPosition[0] + vec4( halfSize, halfSize, 0, 0 ));
	EmitVertex();
	gl_Position = pvwMatrix * (vPosition[0] + vec4( halfSize, -halfSize, 0, 0 ));
	EmitVertex();
	EndPrimitive();
}
```

Geometry Shaders Global Variables

- The Render Pipeline has a few different global variables available in the programmable stages
  - We've used **gl_Position** already, though there are more for other stages!

- The Geometry Shader has a few extra ones, including
  - **gl_PrimitiveID** : the current integer index of the primitive being drawn
  - **gl_in[]** : an array of **gl_PerVertex** structures, with the array size being the number of vertices in the primitive being accessed
  - The **gl_PerVertex** structure contains a few variables, one important one being **gl_Position** for each input vertex if it was set in an earlier stage

```
out gl_PerVertex {
   vec4 gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
};
```

Writing Geometry Shaders

- An example using gl_in[]
  - This example pushes all triangles out away from their mesh, almost in an “exploding” manner

```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

uniform float time;
uniform mat4 pvwMatrix;

void main( )
{
      vec4 dir = vec4( cross(gl_in[1].gl_Position.xyz – gl_in[2].gl_Position.xyz,
                             gl_in[1].gl_Position.xyz – gl_in[0].gl_Position.xyz),0);
      dir = normalize(dir);

      for ( int i = 0 ; i < 3 ; ++i )
      {
            gl_Position = pvwMatrix * (gl_in[i].gl_Position + dir * time);
            EmitVertex();
      }
      EndPrimitive();
}
```

Writing Geometry Shaders

- An example using gl_PrimitiveID

```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

out vec4 gColour;

void main( )
{
	// fract returns the fractional part of a number
	float greyShade = fract( gl_PrimitiveID / 255.0f );

	for ( int i = 0 ; i < 3 ; ++i )
	{
		gColour = vec4(greyShade, greyShade, greyShade, 1);
		gl_Position = gl_in[i].gl_Position;
		EmitVertex();
	}
	EndPrimitive();
}
```

Summary

- The Geometry Shader receives all vertices for a primitive
  - Points, Lines, Triangles, and adjacency vertices for lines and triangles if specified

- The only stage that can change primitives from one type to another
  - Or output nothing at all if desired