Tessellation Shaders

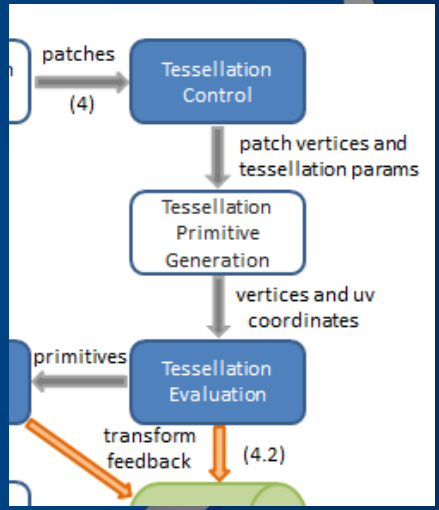
Entering the confusing domain of the tessellated patch!

Tessellation Stages

- As mentioned these are an optional set of three stages
  - Tessellation Control Stage (Programmable)
  - Tessellation Primitive Generation Stage (Fixed)
  - Tessellation Evaluation Stage (Programmable)

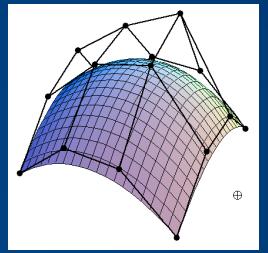
- The tessellator takes in patch primitives and outputs standard primitives
  - A patch is a primitive consisting of 1 or more vertices
  - The Control Stage receives the input patch and can manipulate it
  - The Primitive Generation Stage then generates a tessellated domain
  - The Evaluation Stage defines the ouput primitive type and receives the manipulated patch and the tessellated domain information, then uses them to calculate the tessellated vertices

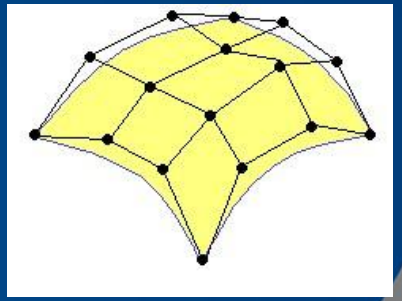
- The newest of the pipeline stages
  - Not available on mobile (yet!)
  - Can be very confusing!



Patch Primitives

- A Patch primitive is simply a primitive with 1 or more vertices
  - All patches in a mesh must have the same number of vertices
- However, a patch does not have to represent the geometry
  - The vertices in a patch are Control Points that define a “hull” for a curved surface
- Before a draw call the application must specify how many vertices there are in each patch
  - OpenGL 4 supports up to 32 vertices per patch primitive
  - A Bezier Patch would have 16 for example
- Each control point goes through the Vertex Shader as usual



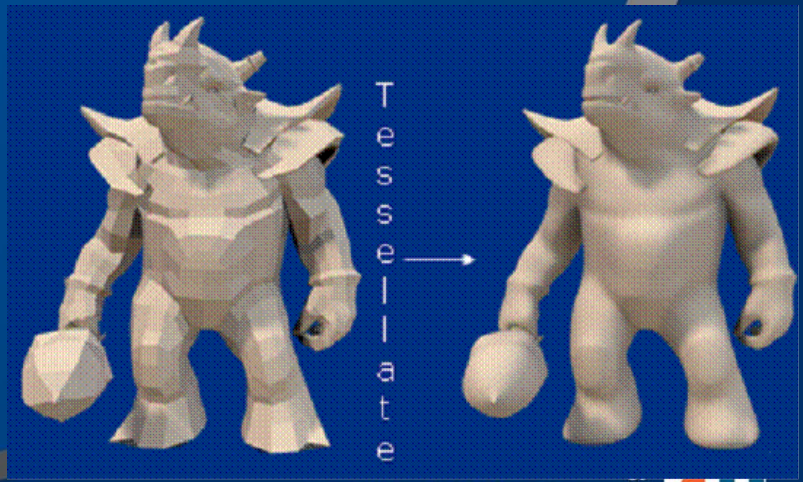


Tessellation Stages

- Using this patch “hull” the Control Stage generates an output patch
  - Can manipulate the input control points or simply output them
  - Can output more or less control points than the input patch had
- The Primitive Generation stage just generates a tessellated domain
  - More on the domain later
- The Evaluation Stage then receives the Control Stage’s output patch and the tessellated domain, executing for every point in the domain, and calculates the vertices for the primitive type it defines

Tessellation Uses

- Tessellations main use in games is to increase the primitive count in meshes
  - We could have a low detail character mesh for old hardware, and then rather than having another high detail mesh for newer hardware we could simply use the low detail one and have the tessellator make it high detail
- We could also manipulate these new primitives to add more detail to a mesh, rather than just adding more triangles
  - This is called Displacement Mapping

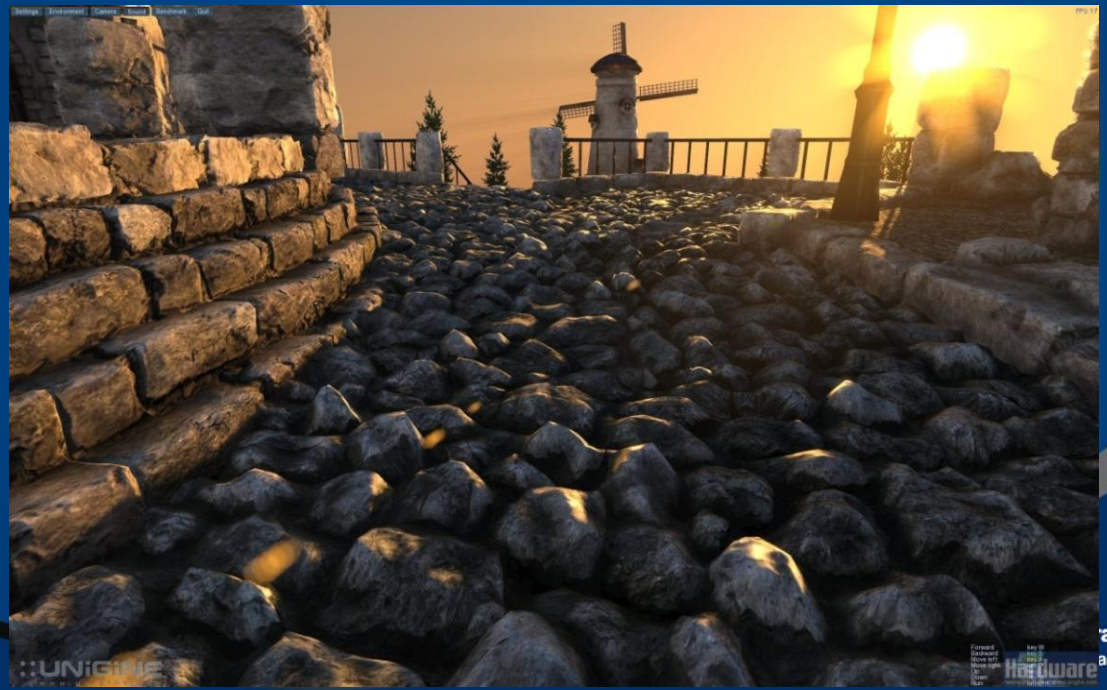


Displacement Mapping

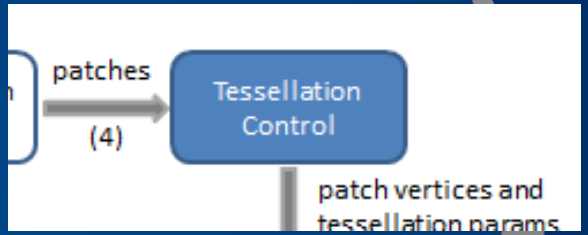
- Displacement Mapping is typically when we offset the vertex position in a mesh, usually by sampling a texture
  - The texture refers to an offset amount rather than a colour, ie the Red channel could refer to an offset scale of 0.0 to 1.0 rather than the colour red
  - Applying this to 3D models can have astonishing effects!

Displacement Mapping

- When we tessellate we have access to the newly generated vertices in the evaluation stage
  - In this stage we can sample a texture and offset the new vertices before they go to the next stage



Tessellation Control



- The Tessellation Control stage is right after the Vertex Shader
  - It receives all the vertex data for each control point in the patch
  - Executes for EACH control point
  - A global called **gl_InvocationID** specifies the index of the current control point being processed in the patch
- The Control Stage takes the input patch and must generate an output patch
  - Uses an output layout, similar to a Geometry Shader, to define the number of control points in the output patch
- It is also used to specify tessellation levels for the Primitive Generation Stage
  - Inner and Output subdivision amounts can be set within the shader, or the application, and can be used to perform dynamic tessellation based off things such as view distance
- This is an optional stage; without it the input patch is used directly as output!

Tessellation Levels

- Tessellation Levels define how many times to subdivide a patch domain
  - Used by the Primitive Generation Stage (more on the domain soon)
- gl_TessLevelInner and gl_TessLevelOuter are GLSL globals defining arrays of floats
  - The array size depends on the domain type specified in the Tessellation Evaluation Stage; usually Triangle or Quad
- The numbers represent how many times to subdivide
  - Inner internally subdivides the domain and Outer subdivides edges
  - For example 1.0 in an Outer would mean the edge is represented by 1 edge, 2.0 would mean cut an edge in two, 3.0 in three, etc
  - An outer of 0.0 is possible, and would mean the domain is degenerate and wont render, but a 0.0 inner counts as 1.0
- Quads use 2 Inner levels and 4 Outer levels, and Triangles use 1 Inner and 3 Outer levels

Triangle Tessellation Level Example

Thick lines are the original edges of a patch defined with 3 control points.

Inner: 1
Outer: 1

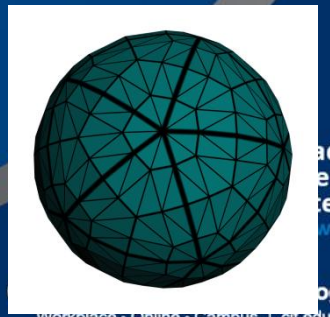
Inner: 1
Outer: 2

Inner: 1
Outer: 3

Inner: 2
Outer: 1

Inner: 3
Outer: 1

Inner: 4
Outer: 4



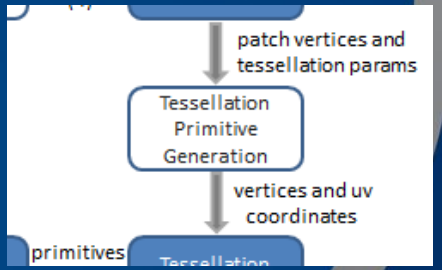
Tessellation Control Example

- Here is a simple Control Shader that receives a patch and outputs a patch with 3 control points, defining no subdivision for its domain
  - Note use of the global **gl_out;** it matches **gl_in** and is available in all stages before Rasterisation
  - The globals **gl_TessLevelOuter** and **gl_TessLevelInner** define how many times to subdivide the domain
  - All usual shader built-ins are available in Control Shaders

```
layout( vertices = 3 ) out;

void main( )
{
	gl_out[ gl_InvocationID ].gl_Position = gl_in[ gl_InvocationID ].gl_Position;

	gl_TessLevelOuter[ 0 ] = 1.0;
	gl_TessLevelOuter[ 1 ] = 1.0;
	gl_TessLevelOuter[ 2 ] = 1.0;
	gl_TessLevelInner[ 0 ] = 1.0;
}
```

Tessellation Primitive Generation

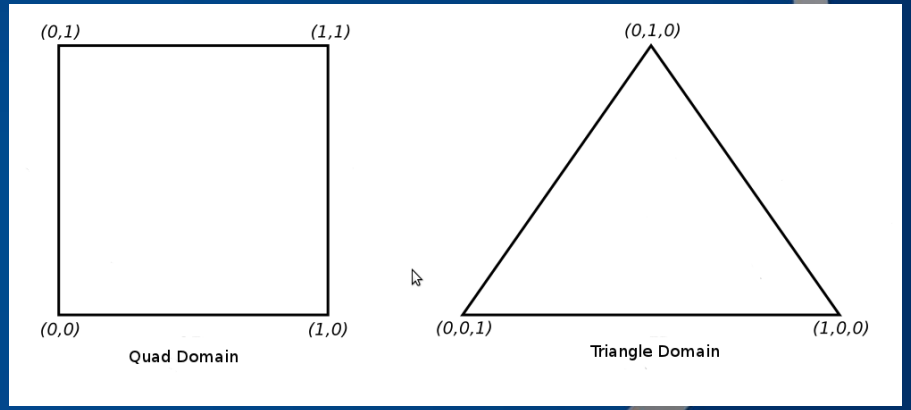
- Once the Control Stage has executed (or if it wasn't defined) the Primitive Generation Stage executes



- This is a fixed stage between the Control Stage and Evaluation Stage
  - Can be a bit hard to understand what actually goes on here
- Generates a tessellated "domain" based on the Tessellation Levels defined in the Control Shader, and the primitive type specified by the Evaluation Shader
  - The tessellated domain creates new blank vertices for the Evaluation Stage
  - Domain information, along with the Control Stage's output patch, are then passed to the Evaluation Stage so it can "fill in" the blank vertices

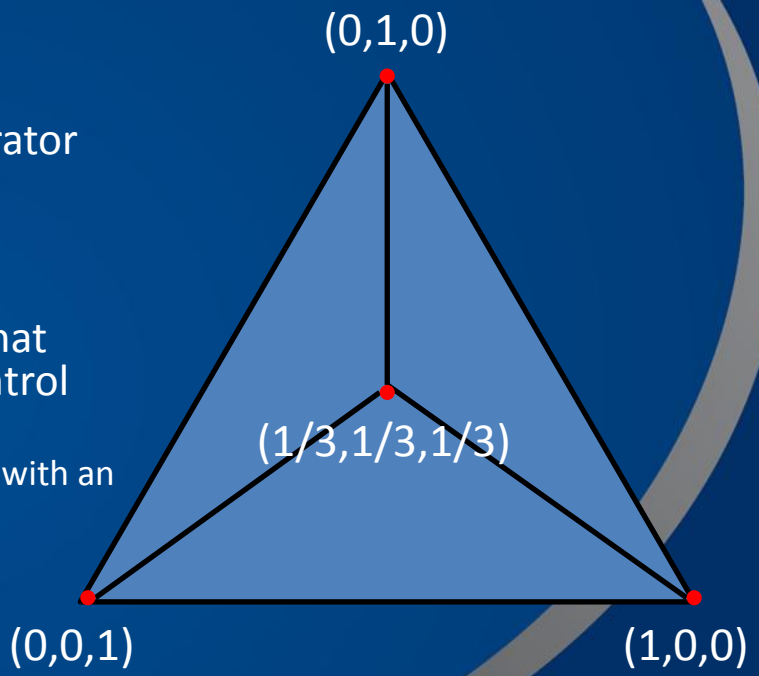
Patch Domains

- A domain is an area defined by Control Points
  - Does not use the Control Stage's patch, which could have any number of control points, even 1!
- Two example domains used by the Evaluation Stage are Quad and Triangle
  - Quad uses 4 Control Points
  - Triangle uses 3
  - Also available is an isoline
- The Primitive Generation Stage generates vertices based on the tessellated domain, defined by the Tessellation Levels



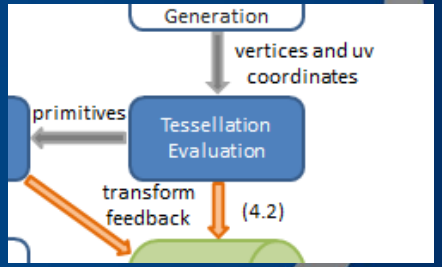
Patch Domains

- For each tessellated vertex the Primitive Generator creates a Tessellation Coordinate
  - 2D for a quad, 3D for a triangle, 1D for an isoline
- This coordinate holds interpolated “weights” that specify its position relative to the domain’s control points
  - The example on the right shows a triangle domain, with an inner tessellation level of 2
  - The newly create point in the middle has weighted values of a 3rd
- This domain has no information regarding the Control Stage’s output patch!
  - The domain is calculated blind of any patch

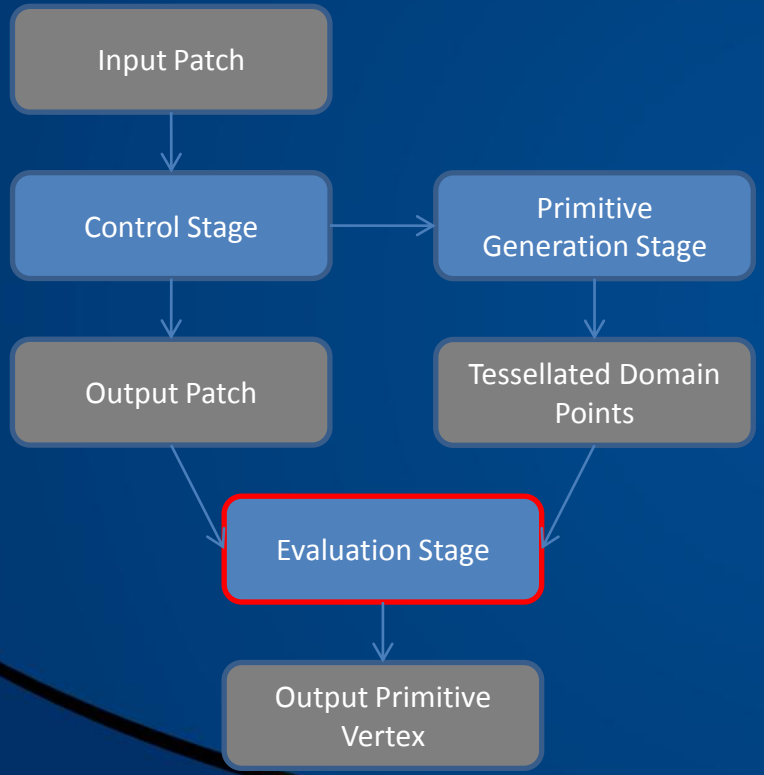


Tessellation Evaluation

- Finally we hit the last stage in the tessellation pipeline
- The Evaluation Stage executes for every tessellated point in the domain created by the Primitive Generation Stage
  - It receives all the control points in the output patch from the Control Stage each time it executes
  - Also receives the tessellated domain coordinate for the domain point currently being processed
- It's job is to generate the actual vertices used in the final tessellated primitives



Tessellation Evaluation



- The Evaluation Stage specifies
  - The primitive type to output
  - The vertex winding order of the generated primitive
  - How the domain handles fractional Tessellation Levels from the Control Stage
- The Primitive Generation Stage creates its domain based on the Evaluation Stage's settings
- Finally the Evaluator calculates the newly generated interpolated vertices

Tessellation Evaluation Example

- The most important bit of information in an Tessellation Evaluation Shader is the domain point's weighted coordinate, stored in the global **gl_TessCoord**
  - This can be used to blend between patch control points from the Tessellation Control Shader
- Also important is the input layout definition
  - Defines the primitive type to use, and vertex winding order
  - Also lets the Primitive Generation Stage know how to handle fractional Tessellation Levels

```
layout( triangles, equal_spacing, ccw ) in;

void main( )
{
	vec4 p0 = gl_in[ 0 ].gl_Position;
	vec4 p1 = gl_in[ 1 ].gl_Position;
	vec4 p2 = gl_in[ 2 ].gl_Position;
	vec3 p = gl_TessCoord.xyz;
	gl_Position = p0 * p.x + p1 * p.y + p2 * p.z;
}
```

Tessellation Evaluation Example

- In this example we are defining that we are generating triangle primitives, and that the winding order is counter clock-wise
- We are also defining that the domain should use **equal_spacing**
  - There are three spacing levels, including **fractional_even_spacing** and **fractional_odd_spacing**, that affect how the subdivision works
- We are also assuming that the patch from the Control Stage has 3 points that we are accessing with **gl_in**
- As we are using triangles **gl_TessCoord** is 3-dimensional and stores how much of each control point affects the final output vertex

```
layout( triangles, equal_spacing, ccw ) in;

void main( )
{
      vec4 p0 = gl_in[ 0 ].gl_Position;
      vec4 p1 = gl_in[ 1 ].gl_Position;
      vec4 p2 = gl_in[ 2 ].gl_Position;
      vec3 p = gl_TessCoord.xyz;
      gl_Position = p0 * p.x + p1 * p.y + p2 * p.z;
}
```

Tessellation

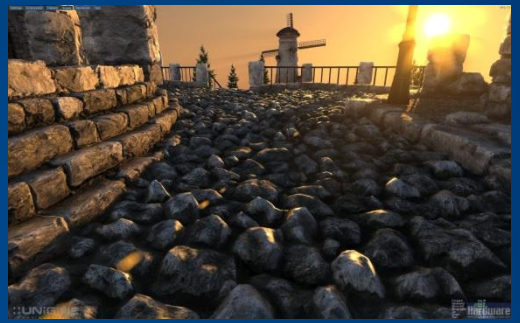
- Tessellation can be rather complex, and our example so far has been rather simple yet complex to understand
  - Just receives input patches as triangles, sends them through the tessellator, and outputs triangles
- If we increase the Tessellation Levels in the Control Shader our triangle could have hundreds or triangles
  - But this would be a little pointless as it would still look like a flat triangle

```
layout( vertices = 3 ) out;

void main( )
{
	gl_out[ gl_InvocationID ].gl_Position = gl_in[ gl_InvocationID ].gl_Position;

	gl_TessLevelOuter[ 0 ] = 5.0; // tessellate 5 times, but visibly no different
	gl_TessLevelOuter[ 1 ] = 5.0;
	gl_TessLevelOuter[ 2 ] = 5.0;
	gl_TessLevelInner[ 0 ] = 5.0;
}
```

```
// we haven't changed how we decide the final vertex possition!
layout( triangles, equal_spacing, ccw ) in;

void main( )
{
	vec4 p0 = gl_in[ 0 ].gl_Position;
	vec4 p1 = gl_in[ 1 ].gl_Position;
	vec4 p2 = gl_in[ 2 ].gl_Position;
	vec3 p = gl_TessCoord.xyz;
	gl_Position = p0 * p.x + p1 * p.y + p2 * p.z;
}
```

Tessellation Displacement Mapping

- We need to push and pull and smooth our tessellated primitives ourselves, and one way is by using displacement mapping!
  - We would need to add texture coordinates to out patch control points
  - We could then sample a texture within the Evaluation Shader and offset our new vertices!

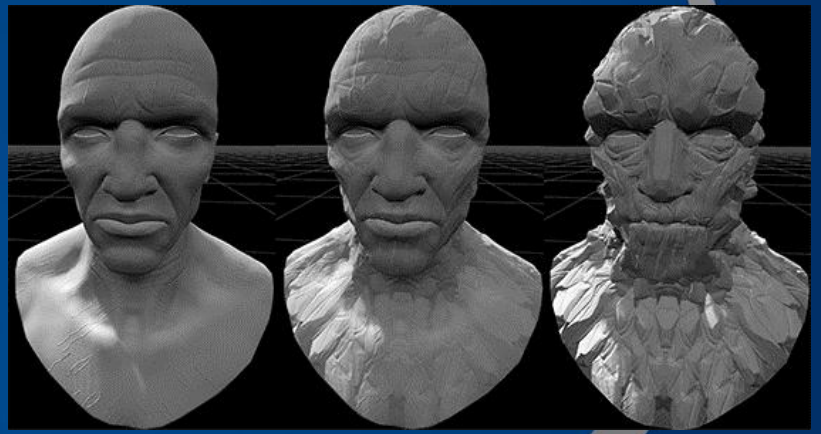


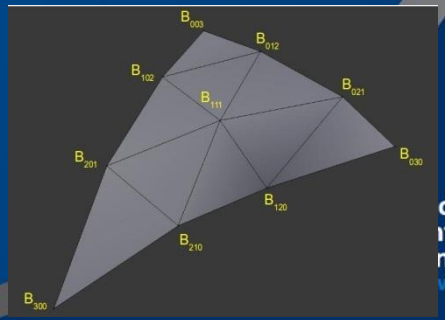
```
layout( vertices = 3 ) out;

in vec2 vTexCoord[];
out vec2 csTex[];

void main( )
{
	gl_out[ gl_InvocationID ].gl_Position = gl_in[ gl_InvocationID ].gl_Position;
	csTex[ gl_InvocationID ] = vTexCoord[ gl_InvocationID ];

	gl_TessLevelOuter[ 0 ] = 5.0; // subdivide 5 times!
	gl_TessLevelOuter[ 1 ] = 5.0;
	gl_TessLevelOuter[ 2 ] = 5.0;
	gl_TessLevelInner[ 0 ] = 5.0;
}
```

```
layout( triangles, equal_spacing, ccw ) in;

in vec2 csTex[];

uniform sampler2D displacementMap;
uniform float scale;

void main( )
{
	vec4 p0 = gl_in[ 0 ].gl_Position;
	vec4 p1 = gl_in[ 1 ].gl_Position;
	vec4 p2 = gl_in[ 2 ].gl_Position;
	vec3 p = gl_TessCoord.xyz;

	vec2 uv = csTex[0] * p.x + csTex[1] * p.y + csTex[2] * p.z;
	float dist = texture2D( displacementMap, uv ).r;

	gl_Position = p0 * p.x + p1 * p.y + p2 * p.z;
	gl_Position.y += dist * scale;
}
```

Mesh Smoothing With Tessellation

- Smoothing a complex mesh is a lot harder than displacement mapping
- If we wanted to do tessellation the right way we would literally treat all our primitives as curved patch hulls
  - Bezier Curves for example
- Then during the Evaluation Stage we would correctly calculate a curved position
- There are a few very complex algorithms out there for correctly tessellating complex meshes
  - Bezier Patches
  - Gregory Patches
  - PN Triangles (most common in games)
  - Their scope is beyond this introductory session!





Summary

- Tessellation is a complex topic
  - Simple enough to implement basic displacement mapping, but complex to implement more complete tessellation algorithms
- The Tessellation stages work very differently to the other stages
  - But still have access to all the same GLSL functionality
- Fairly new technology, but slowly moving to the mobile space
  - ARM have demonstrated a prototype chip able to achieve basic DirectX 11 tessellation on mobile!

http://www.opengl.org/wiki/Tessellation