Abertay University

Tessellation

CMP301 Graphics Programming with Shaders

This week

- Overview
- The tessellation stages
  - Hull Shader
  - Tessellator
  - Domain shader
- Example

Overview

- Tessellation
  - The process of subdividing geometry
  - Main purpose of tessellation
    - Add detail to geometry
    - Dynamic LOD
      - Using tessellation to have high poly near camera and low poly in the distance
    - Physics and animation efficiency
      - Do calculations on a low poly model then add detail
    - Memory optimisation
      - Store low poly in memory adding detail at runtime
      - Sending less data to GPU

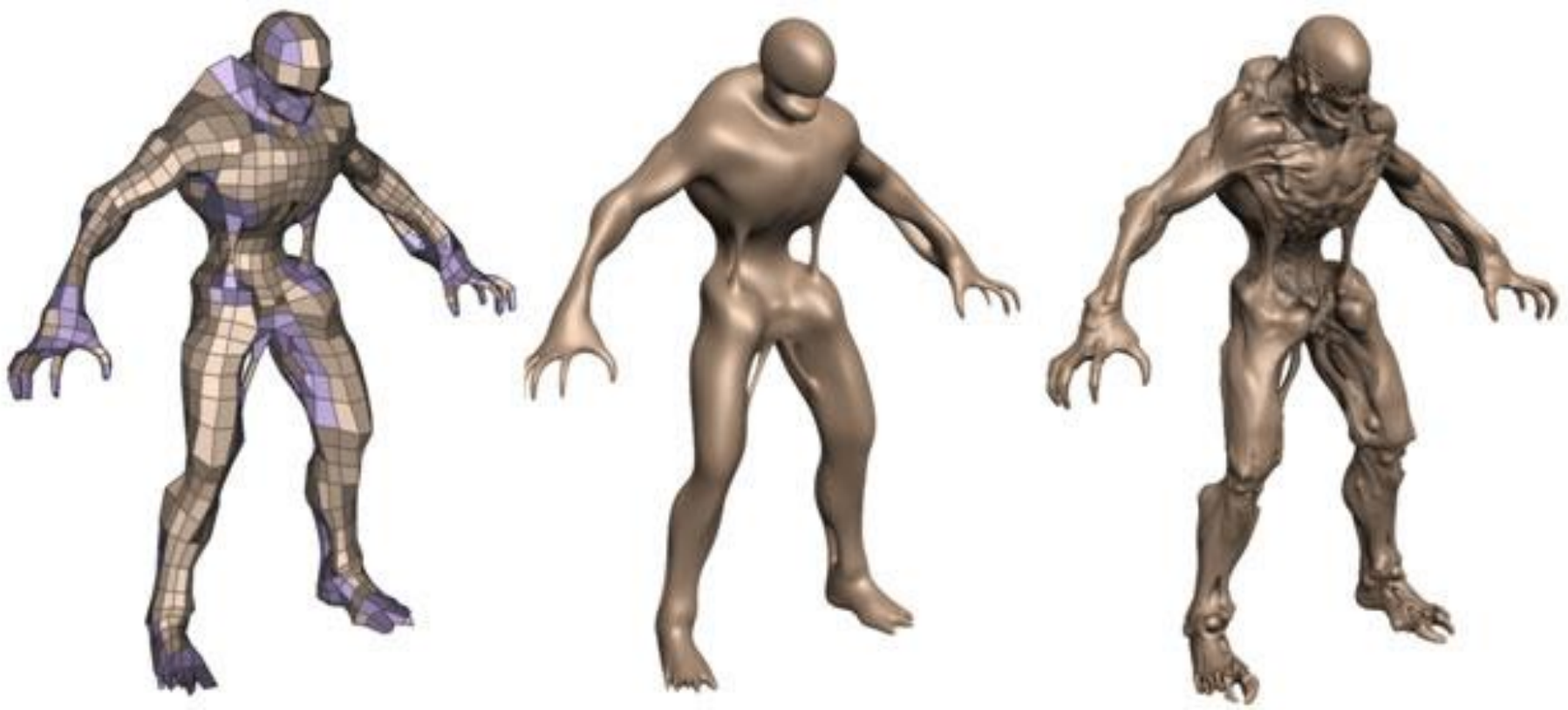
Overview

- Example
  - Take a quad and cut it across the diagonal
    - You have just tessellated the quad (into two triangles)
  - The challenge
    - It doesn’t matter if a square is rendered as two triangles or two thousand triangles
    - This doesn’t gain us more detail
    - We need the new triangles to depict new information

Overview

- The most popular technique for using these extra triangles is displacement mapping
- Displacement maps require a large number of vertices
- We can only form detailed surfaces with sufficient vertices
  - Think back to the vertex manipulation with less vertices it doesn’t look great

Overview



Low poly model

Tessellated smoothed model

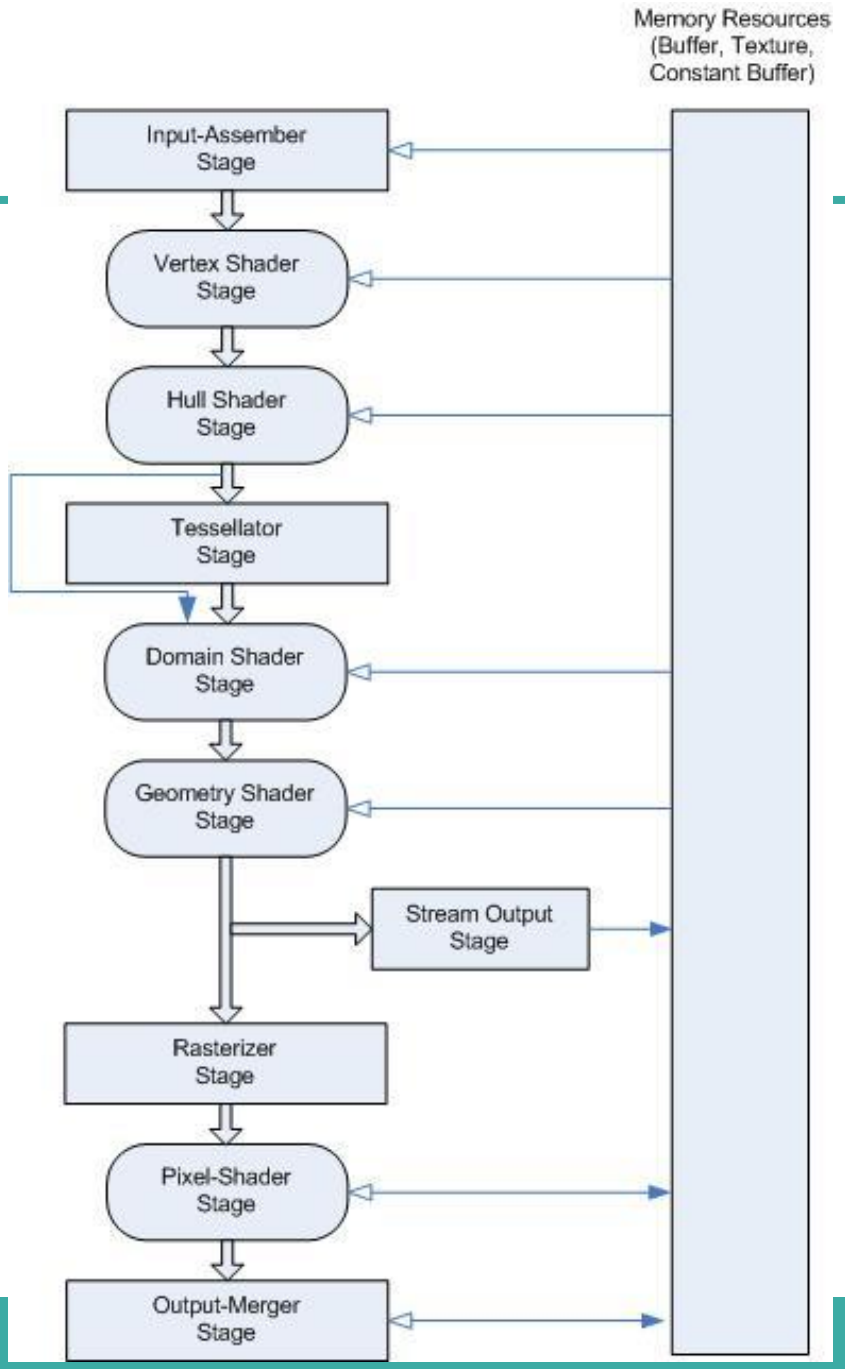
Displacement mapping applied

Overview

- When to tessellate?
- How much to tessellate?
  - Distance from camera
  - Screen coverage
    - If an object takes up lots of screen real-estate it should probably get more tessellation
  - Orientation
  - Roughness
    - Rough surfaces require more details than smooth surfaces
  - Performance
  - Batch tessellation render
    - Turning tessellation on and off is expensive

Where does this happen

- The tessellation stages
- Mentioned back at the start when we looked at the pipeline
- The hull-shader, tessellator and domain shader comprise the tessellation stages
- Can render seriously high detailed geometry by having the hardware sub-divide the geometry



Tessellation primitives

- The tessellation stages do not process vertices or triangles
  - Currently we pass vertices as a triangle list for rendering
  - This will have to change
- For tessellation we submit “control points” that make up “patches”
- A patch can have 1-32 control points

Tessellation primitives

- A triangle can be treated as a patch with 3 control points
- D3D11_PRIMITIVE_3_CONTROL_POINT_PATCH
- This way we can use current mesh data for tessellation
  - Just need to change the primitive type
- Quad patch has 4 control points
- D3D11_PRIMITIVE_4_CONTROL_POINT_PATCH
- Will be tessellated into triangles

Vertex shader

- If tessellation happens after the vertex shader, what does the vertex shader do?
  - Process the control points
  - Can do physics and animation calculations on low poly models, prior to tessellation
- Any vertex manipulation you were doing has to be moved to the domain shader
  - So it happens after tessellation
  - And we have all the geometry
  - Matrix transforms, or other manipulation

Hull shader

- Two components to a Hull shader
  - Constant hull function
  - Control point function
- Constant hull function
  - Evaluates each patch
  - Outputting tessellation factors
  - Tessellation factors instruct the tessellator how to divide up the patch
    - Internal and external values for subdivision

Hull shader – Constant Function

- Receives all the control points for a patch based on the output of the vertex shader
  - Inputpatch <InputType, 3>
- 3 is the number of control points for the patch
  - Could be 3, 4, etc
- Also receives a patchID
  - Identifies the patch in the draw call
  - Can be stored if needed

Hull shader – Constant Function

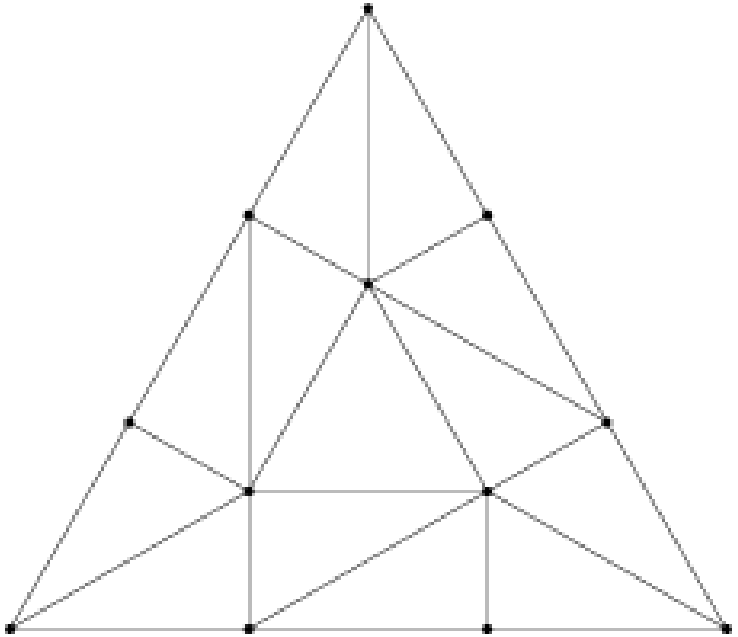
- The constant hull shader's purpose is to output the tessellation factors
  - Based on patch topology
- Tessellating a patch requires two parameters
  - Edge tessellation factor
    - How much to tessellate each edge
    - 3 for a triangle patch
  - Interior tessellation factor
    - How much to tessellate the a patch
    - 1 for a triangle patch

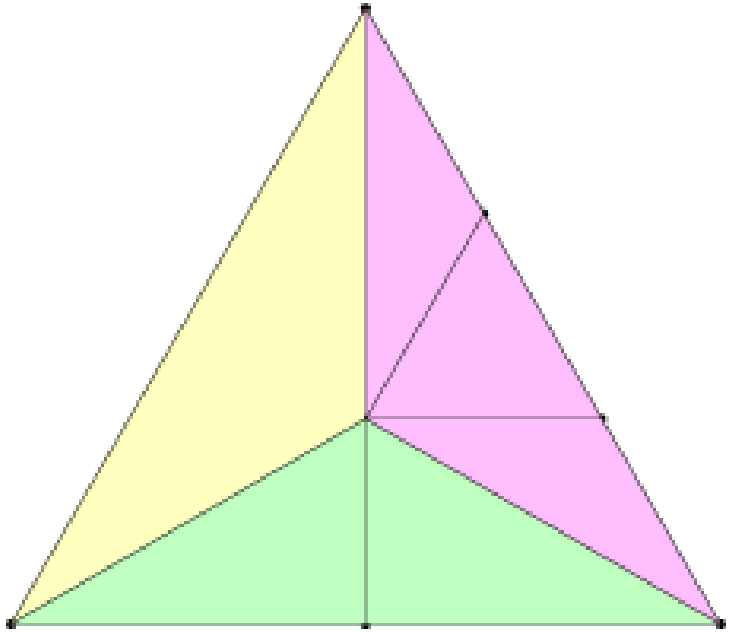
Hull shader – Constant Function

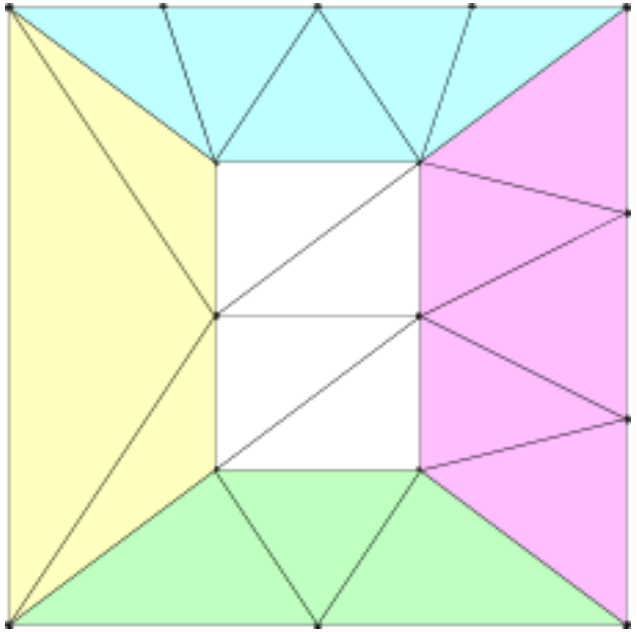
- Tessellation factors
  - For a triangle
    - 3 edge factors
    - 1 interior
  - For a quad
    - 4 edge factors
    - 2 interior factors
      - Horizontal and vertical dimensions of the quad
- Maximum tessellation factor is 64
  - If tessellation factor is zero, the patch is rejected
  - If one, no tessellation occurs

Images

- Tessellation factors don’t have to be the same







Hull shader

```
ConstantOutputType PatchConstantFunction(InputPatch<InputType, 3> inputPatch, uint patchId :
SV_PrimitiveID)
{
    ConstantOutputType output;



    // Set the tessellation factors for the three edges of the triangle.
    output.edges[0] = 3;
    output.edges[1] = 3;
    output.edges[2] = 3;

    // Set the tessellation factor for tessellating inside the triangle.
    output.inside = 3;

    return output;
}
```

Hull shader

- Control point hull function
  - Invoked once per control point
  - Can modify control points
    - Add control points
  - Most cases only need a simple pass-through shader
    - No modifying
- Has a number attributes that need be set

Hull shader – Control Point Function

- Control point attributes
  - Domain
    - The patch type (tri, quad, isocline)
    - [domain(“tri”)]
  - Partitioning
    - Defines the subdivision to be made
      - Integer [1-64]
      - fractional_even [2-64]
      - fractional_odd [1-63]
    - Integer tessellation factor (ignores fractions)
    - [partitioning(“integer”)]
    - Fractional uses floats, can avoid detail popping

Hull shader – Control Point Function

- Control point attributes
  - Outputtopology
    - Winding order (triangle_cw, triangle_ccw, line)
    - [outputtopology(“triangle_cw”)]
    - Calculated in UV space (y-axis is positive downward)
  - Outputcontrolpoints
    - Number of times the hull shader executes
    - One control point per output
    - Semantic SV_outputControlPointID gives the index of current control point
    - [outputcontrolpoints(3)]

Hull shader – Control Point Function

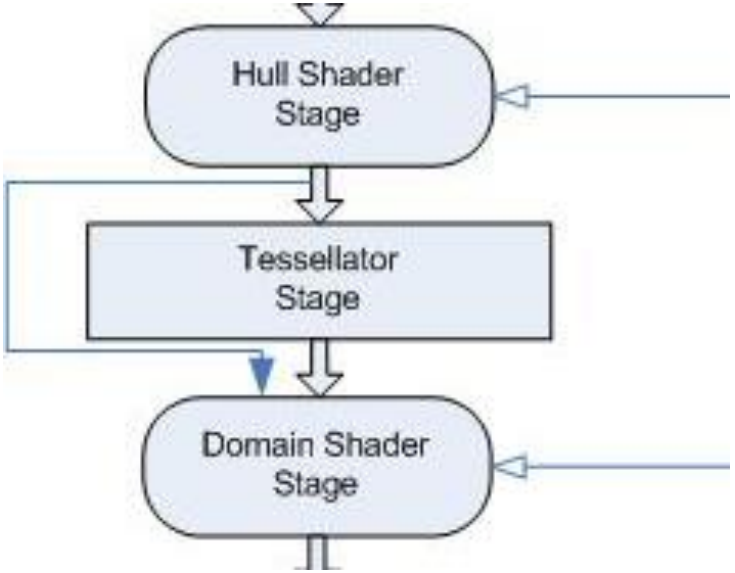
- Control point attributes
  - Patchconstantfunc
    - Points to the constant hull shader function
    - [patchconstantfunc(“PatchConstantFunction”)]
  - Maxtessfactor
    - Specifies the maximum tessellation factor
    - Potential optimisation by the hardware if the upper bound is known
    - For D3D11 max is 64
    - [maxtessfactor(64.0f)]

Hull shader – Control Point Function

```
[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("PatchConstantFunction")]
OutputType main(InputPatch<InputType, 3> patch, uint pointId : SV_OutputControlPointID, uint patchId : SV_PrimitiveID)
{
    OutputType output;

    // Set the position for this control point as the output position.
    output.position = patch[pointId].position;

    // Set the input colour as the output colour.
    output.colour = patch[pointId].colour;

    return output;
}
```

The tessellator

- Fixed function stage
- We have no direct control over it
- Subdivides the patches based on the tessellation factors
- Outputs new created vertices
  - This is the first time we are dealing vertices



Domain shader

- Receives data from both Hull shader and Tessellator
- Invoked once per vertex
  - Like the vertex shader
  - Acts as the vertex shader
- Receives per patch data from the constant hull shader
  - Including
    - Tessellation factor
    - Additional data we add
    - Patch control points
    - U, V coordinates of the tessellated vertex positions

Domain shader

- We have to use the UV coordinates and control point data to derive the 3D position of the vertex
- A couple ways to do this
  - For tri patches
    - The UVW represent weightings
    - By combining with control point positions we can determine the vertex position
  - For quad patches
    - We use bilinear interpolation between points using the UV values

Domain shader

```
[domain("tri")]
OutputType main(ConstantOutputType input, float3 uvwCoord : SV_DomainLocation, const OutputPatch<InputType, 3> patch)
{
    float3 vertexPosition;
    OutputType output;

    // Determine the position of the new vertex.
    // Invert the y and Z components of uvwCoord as these coords are generated in UV space and
    therefore y is positive downward.
    // Alternatively you can set the output topology of the hull shader to cw instead of ccw
    vertexPosition = uvwCoord.x * patch[0].position + uvwCoord.y * patch[1].position + uvwCoord.z *
                     patch[2].position;

    // Calculate the position of the new vertex against the world, view, and projection matrices.
    output.position = mul(float4(vertexPosition, 1.0f), worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Send the input colour into the pixel shader.
    output.colour = patch[0].colour;

    return output;
}
```

An example

- A slightly different mesh
- New shaders
- Update to shaders
- PICTURES

Mesh

- Within the framework there is a tessellation mesh
  - Simple “triangle” mesh
  - Very similar to normal/other meshes
  - BUT the topology has been changed
    - Note the render function
- Set the type of primitive that should be rendered from this vertex buffer
- Not a triangle list anymore
- deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST);

- When rendering this shape it is recommend you turn on wireframe mode

Shaders

- 4 Shaders
  - Vertex
  - Hull
  - Domain
  - Pixel

Vertex Shader

```
struct InputType
{
    float3 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};
struct OutputType
{
    float3 position : POSITION;
    float4 colour : COLOR;
};
OutputType main(InputType input)
{
    OutputType output;

 // Pass the vertex position into the hull shader.
    output.position = input.position;

    // Pass the input colour into the hull shader.
    output.colour = float4(1.0, 0.0, 0.0, 1.0);

    return output;
}
```

Hull Shader

```
// Tessellation Hull Shader
// Prepares control points for tessellation

struct InputType
{
    float3 position : POSITION;
    float4 colour : COLOR;
};

struct ConstantOutputType
{
    float edges[3] : SV_TessFactor;
    float inside : SV_InsideTessFactor;
};
```

Hull shader

```
struct OutputType
{
    float3 position : POSITION;
    float4 colour : COLOR;
};

ConstantOutputType PatchConstantFunction(InputPatch<InputType, 3> inputPatch, uint patchId :
SV_PrimitiveID)
{
    ConstantOutputType output;


    // Set the tessellation factors for the three edges of the triangle.
    output.edges[0] = 3;
    output.edges[1] = 3;
    output.edges[2] = 3;

    // Set the tessellation factor for tessellating inside the triangle.
    output.inside = 3;

    return output;
}
```

Hull shader

```
[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("PatchConstantFunction")]
OutputType main(InputPatch<InputType, 3> patch, uint pointId : SV_OutputControlPointID, uint patchId :
SV_PrimitiveID)
{
    OutputType output;

    // Set the position for this control point as the output position.
    output.position = patch[pointId].position;

    // Set the input colour as the output colour.
    output.colour = patch[pointId].colour;

    return output;
}
```

Domain Shader

```
cbuffer MatrixBuffer : register(cb0)
{
    matrix worldMatrix;
    matrix viewMatrix;
    matrix projectionMatrix;
};

struct ConstantOutputType
{
    float edges[3] : SV_TessFactor;
    float inside : SV_InsideTessFactor;
};

struct InputType
{
    float3 position : POSITION;
    float4 colour : COLOR;
};

struct OutputType
{
    float4 position : SV_POSITION;
    float4 colour : COLOR;
};
```

Domain shader

```
[domain("tri")]
OutputType main(ConstantOutputType input, float3 uvwCoord : SV_DomainLocation, const OutputPatch<InputType, 3> patch)
{
    float3 vertexPosition;
    OutputType output;

    // Determine the position of the new vertex.
    vertexPosition = uvwCoord.x * patch[0].position + uvwCoord.y * patch[1].position + uvwCoord.z *  patch[2].position;

    // Calculate the position of the new vertex against the world, view, and projection matrices.
    output.position = mul(float4(vertexPosition, 1.0f), worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Send the input colour into the pixel shader.
    output.colour = patch[0].colour;

    return output;
}
```

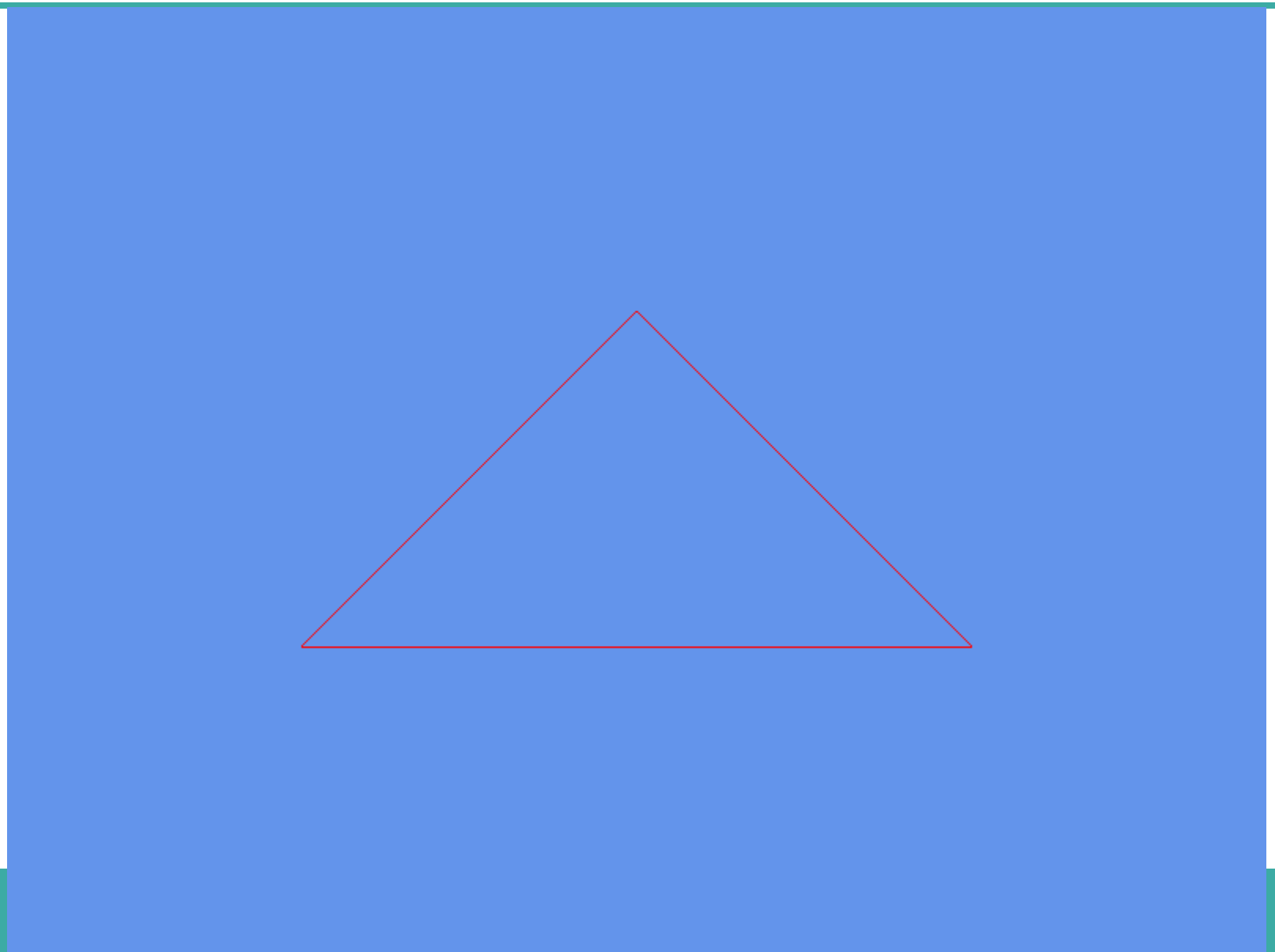
Pixel Shader

```
struct InputType
{
    float4 position : SV_POSITION;
    float4 colour : COLOR;
};

float4 main(InputType input) : SV_TARGET
{
    return input.colour;
}
```

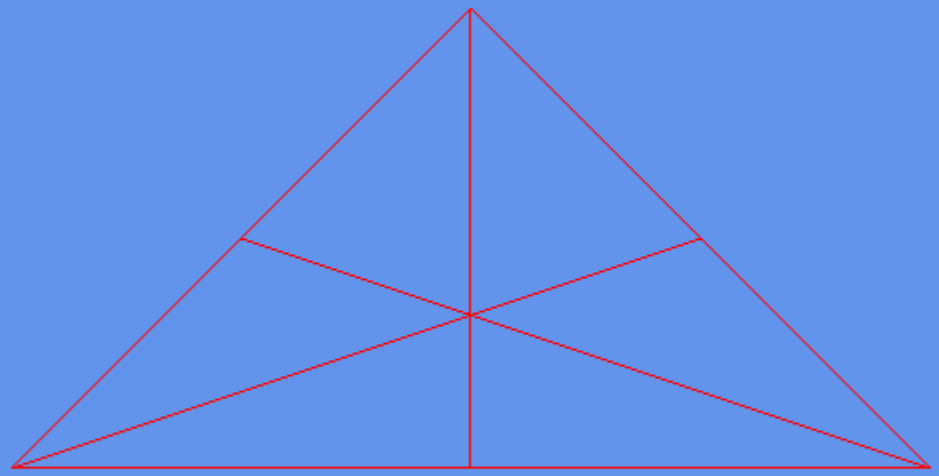
Tessellation Shader Class

- Loads all 4 shaders
  - Vertex
  - Hull
  - Domain
  - Pixel
- It has a constant buffer so we can pass a tessellation amount into the shaders
  - This allows us to control the tessellation
- Rendering a simple triangle using our new set of shaders

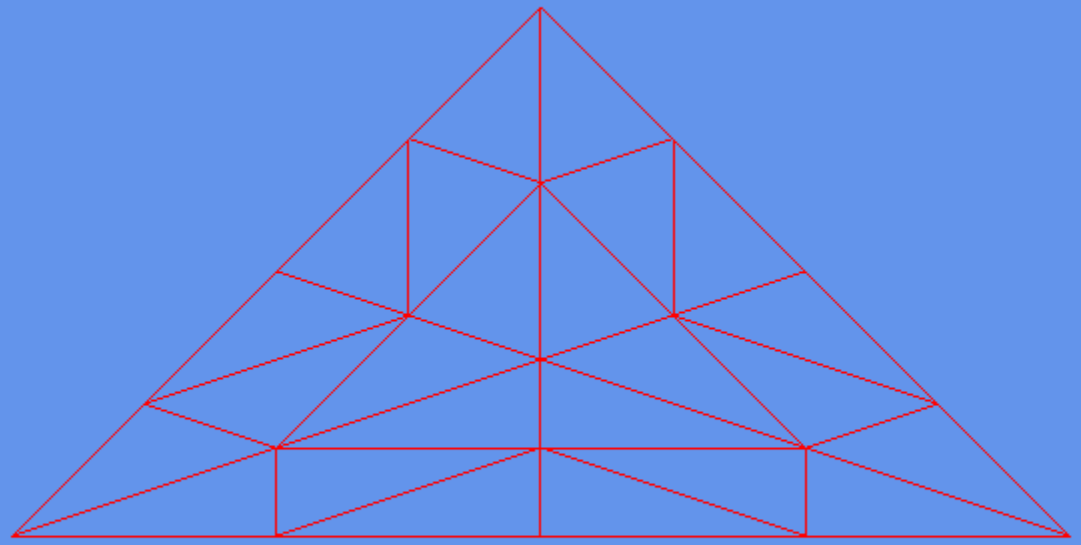
Tess factor 1



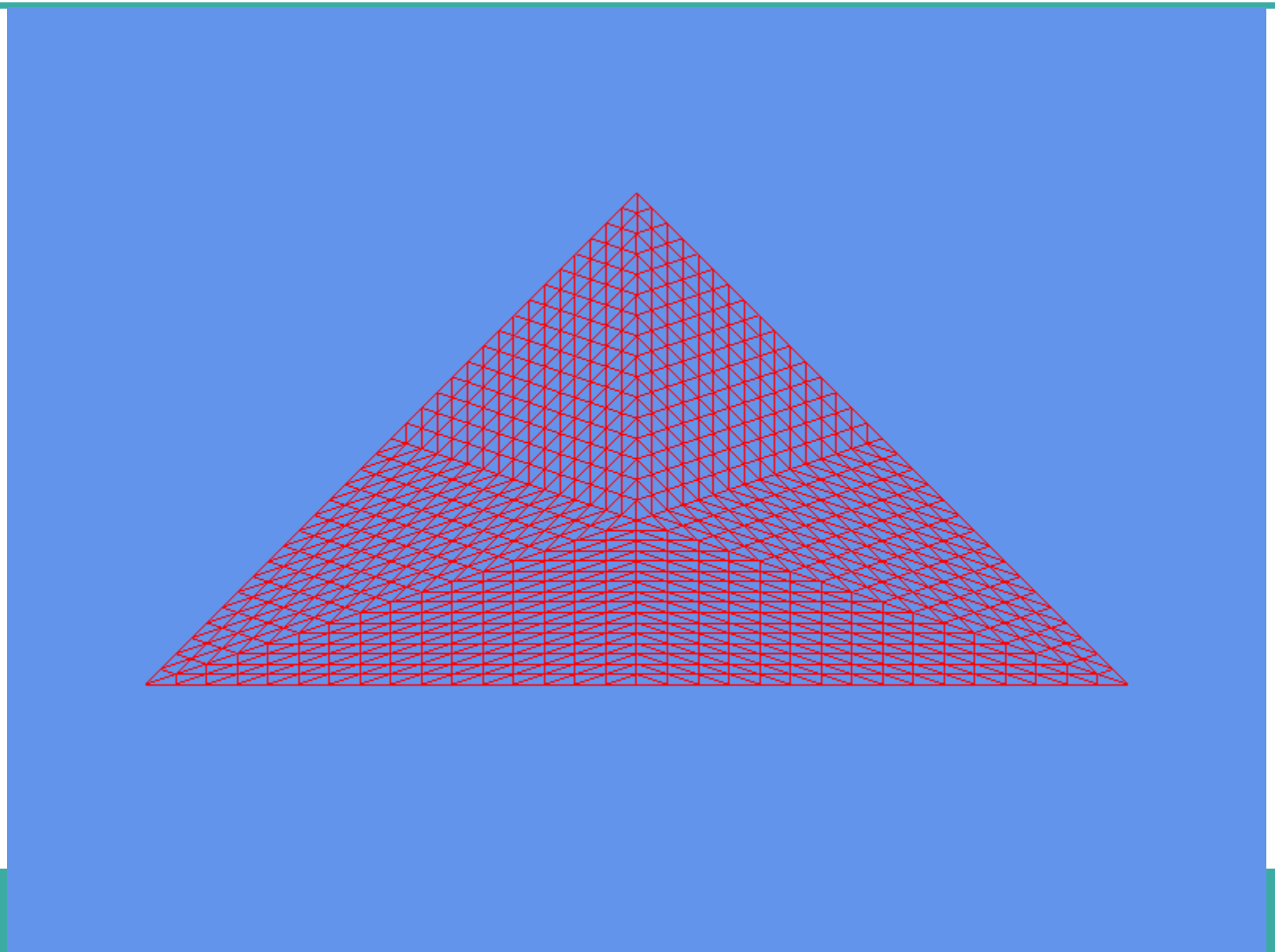
Tess factor 2



Tess factor 4



Tess factor 32



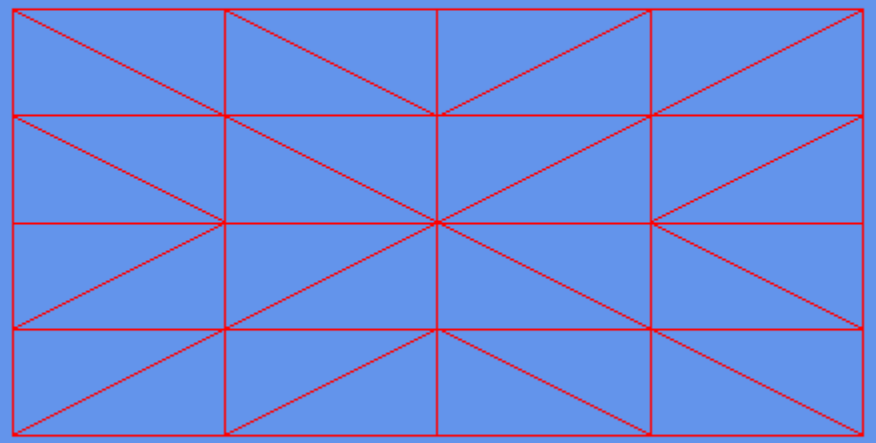
Tessellation factors

- For quads we need more edge and interior factors

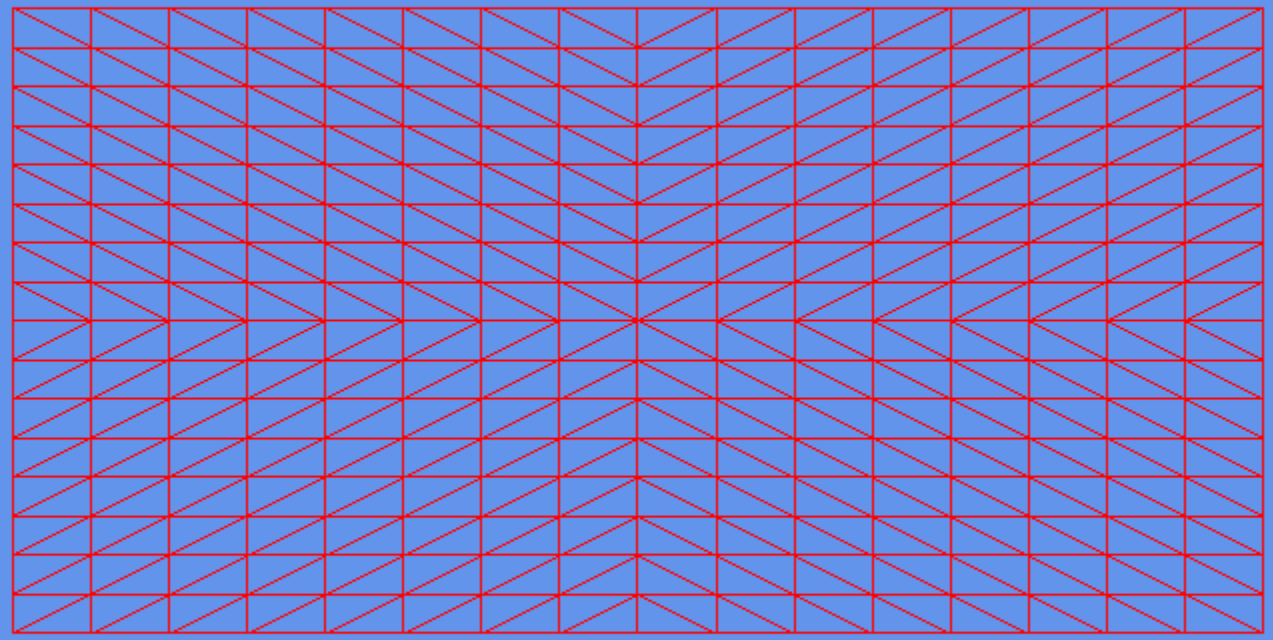
```
output.edges[0] = 2;
output.edges[1] = 2;
output.edges[2] = 4;
output.edges[3] = 4;

output.inside[0] = 2;
output.inside[1] = 4;
```

- For any tessellation the tessellation factors don’t have to match
  - Can produce uneven tessellation

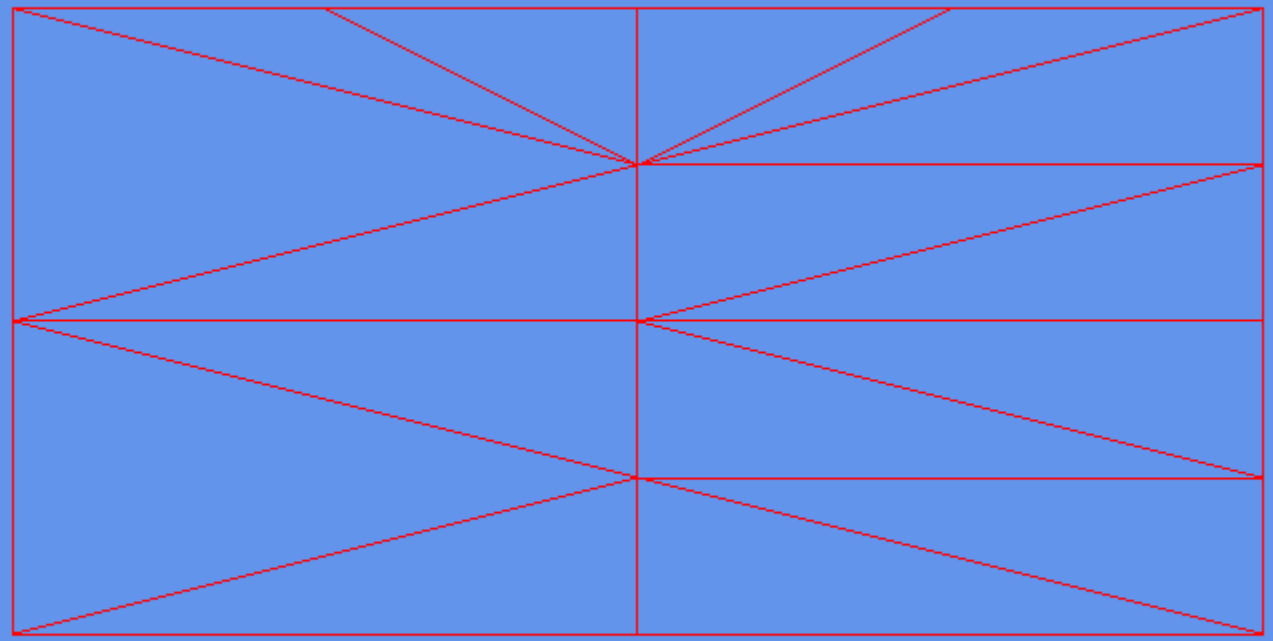
Quad tess factor 4



Quad tess factor 16



Quad tess factor 2 and 4



Partial quad hull shader

```
ConstantOutputType PatchConstantFunction(InputPatch<InputType, 4> inputPatch, uint patchId : SV_PrimitiveID)
{
    ConstantOutputType output;


    // Set the tessellation factors for the three edges of the triangle.
    output.edges[0] = tessellationFactor;
    output.edges[1] = tessellationFactor;
    output.edges[2] = tessellationFactor;
    output.edges[3] = tessellationFactor;

    output.inside[0] = tessellationFactor;
    output.inside[1] = tessellationFactor;

    return output;
}
```

Partial quad hull shader

```
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("PatchConstantFunction")]
OutputType main(InputPatch<InputType, 4> patch, uint pointId : SV_OutputControlPointID, uint patchId : SV_PrimitiveID)
{
    OutputType output;


    // Set the position for this control point as the output position.
    output.position = patch[pointId].position;

    // Set the input colour as the output colour.
    output.colour = patch[pointId].colour;

    return output;
}
```

Partial quad domain shader

```
[domain("quad")]
OutputType main(ConstantOutputType input, float2 uvwCoord : SV_DomainLocation, const OutputPatch<InputType, 4> patch)
{
    float3 vertexPosition;
    OutputType output;

    float3 v1 = lerp(patch[0].position, patch[1].position, 1-uvwCoord.y);
    float3 v2 = lerp(patch[2].position, patch[3].position, 1-uvwCoord.y);
    vertexPosition = lerp(v1, v2, uvwCoord.x);

    // Calculate the position of the new vertex against the world, view, and projection matrices.
    output.position = mul(float4(vertexPosition, 1.0f), worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Send the input color into the pixel shader.
    output.colour = patch[0].colour;

    return output;
}
```

In the labs

- Investigating tessellation
  - Some code provided

- Note
  - Current examples show tessellation in action
  - Requires some modification to be useful
    - Adding texture coordinates, normals, etc