MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

# 159.709 Computer Graphics

## Tesselation

Daniel Playne
d.p.playne@massey.ac.nz

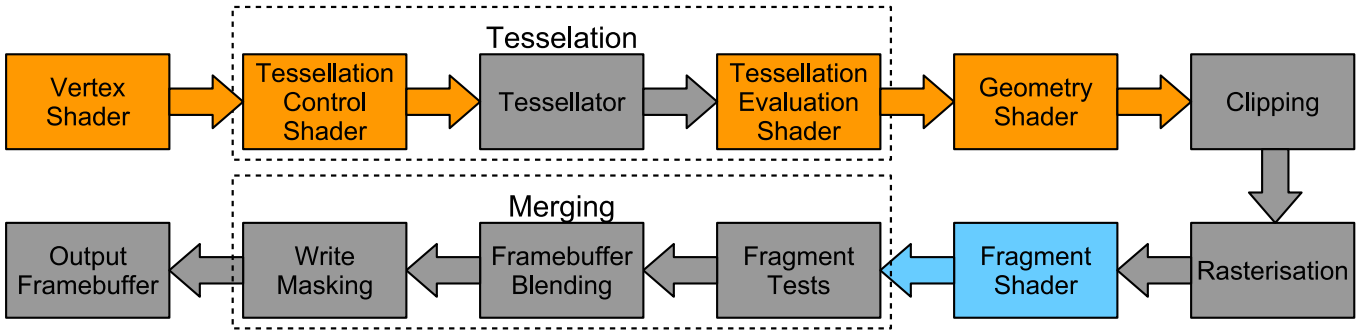
# Tessellation

A ***tessellation*** is an arrangement of geometric shapes according to a set of rules that will fill a plane without leaving any gaps. Tessellations have been used in mosaics and artworks for thousands of years.

Source: Wikipedia, Author: Mbellaccini, License: CC BY-SA 4.0

# Tessellation

In OpenGL, tessellation is the process of taking *patches* of vertex data and subdividing them further into smaller primitives.



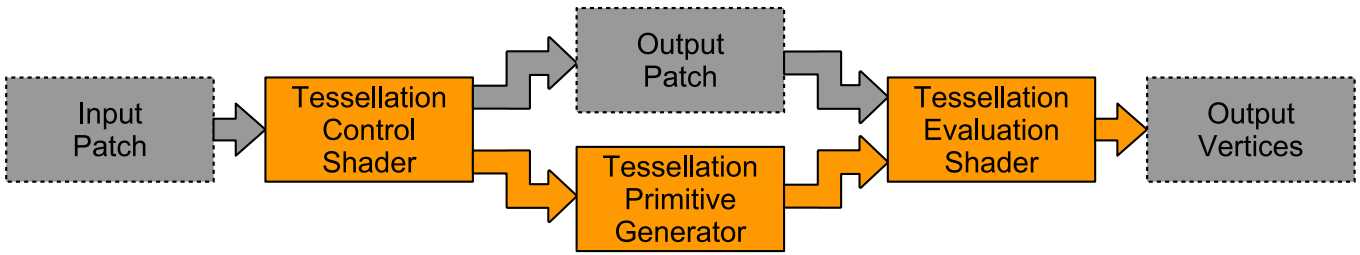

The ***tessellation*** stage consists of two programmable stages (the tessellation ***control*** and ***evaluation*** shaders) and a fixed function stage (available since OpenGL 4.0).

# Tessellation

The general process for OpenGL tessellation involves taking a *patch* of some type and subdividing it into more primitives.

This is achieved by creating new vertexes and calculating their vertex position, colour, normal etc using information from the original vertexes (and possibly other input).

Each stage of the tessellation pipeline performs a part of this process.



# Patches

Tessellation shaders work with a different type of primitive called patches (`GL_PATCHES`) which represent an arbitrary piece of geometry defined by the programmer.

These patches define the control points on a curve or surface. They can be configured by setting the number of vertices per patch.

```
1 glPatchParameteri(GL_PATCH_VERTICES, numPatchVertices);
```

# Patches

The number of vertexes per patch must be in the range `[1, maxPatchVertices]`, which can be found with:

```
1 int maxPatchVertices;
2 glGetIntegerv(GL_MAX_PATCH_VERTICES, &maxPatchVertices);
```

All patches are a set of `numPatchVertices` different vertexes (there is no *patch strip* or *patch loop* equivalent). Patches can be drawn with indexed rendering.

# Tessellation Control Shader

The first stage of the tessellation pipeline is the ***tessellation control shader*** (TCS) which is invoked *per-vertex* of the *output patch*.

The *input patch* consists of the vertexes from the ***vertex shader*** which are aggregated into arrays based on the size of the input patch.

The TCS may (or may not) transform these vertexes before passing them onto the *output patch*. The TCS will also produce output to the ***tessellation primitive generator*** on how to subdivide the output patch.

# Tessellation Control Shader

The inputs from the ***vertex shader*** are aggregated into arrays (as are the outputs).

```
1  // OpenGL 4.0
2  #version 400
3
4  // Input to Control Shader
5  in vec4 ctrl_Position[];
6  in vec4 ctrl_Colour[];
7
8  // Output from Control Shader
9  out vec4 eval_Position[];
10 out vec4 eval_Colour[];
11 ...
```

# Tessellation Control Shader

Each instance of the tessellation control shader can only write *per-vertex* output to the output variable that corresponds to its `gl_InvocationID` (it can read any of the inputs).

```
1  ...
2  // Input to Control Shader
3  in vec4 ctrl_Position[];
4  in vec4 ctrl_Colour[];
5
6  // Output from Control Shader
7  out vec4 eval_Position[];
8  out vec4 eval_Colour[];
9
10 void main () {
11    // Per-Vertex Output
12    eval_Position[gl_InvocationID] = ctrl_Position[gl_InvocationID];
13    eval_Colour[gl_InvocationID]   = ctrl_Colour[gl_InvocationID];
14    ...
15 }
```

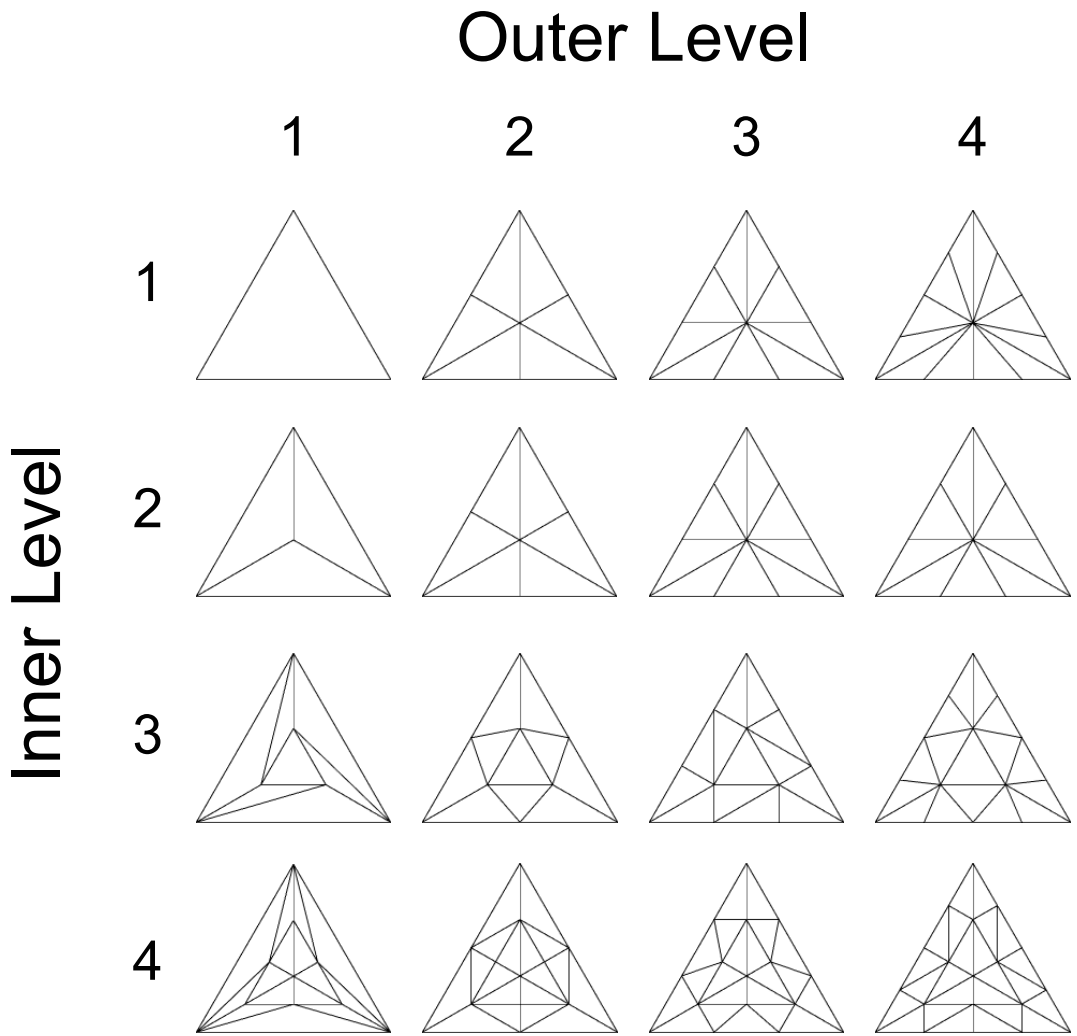
# Tessellation Control Shader

The tessellation control shader also sets the tessellation levels of the patch. The tessellation levels can be configured by writing to `gl_TessLevelInner[]` and `gl_TessLevelOuter[]`. These only need to be written once per patch.

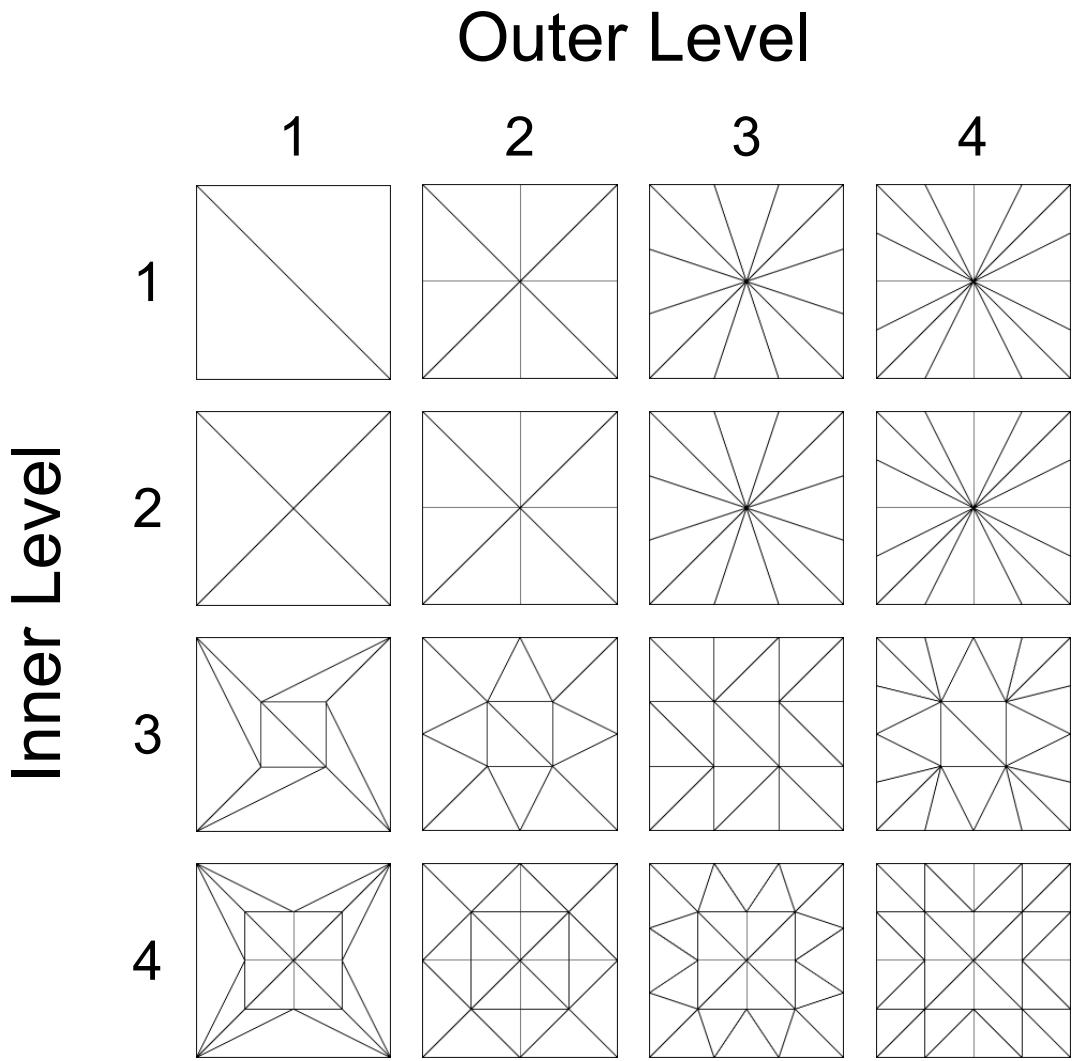
The tessellation levels determine how to break the patches up. The *outer* levels determine how many segments the edges of the abstract patch should be split into and the *inner* levels define how many internal segments it should be broken into.

- ***Triangles*** have three *outer* levels and a single *inner* level.
- ***Quads*** have four *outer* levels and two *inner* levels.

# Tessellation Levels - Triangles



# Tessellation Levels - Quadrangles



# Tessellation Control Shader

Simple *Tessellation Control Shader* - Triangles:

```
1  // OpenGL 4.0
2  #version 400
3
4  // Triangles
5  layout (vertices = 3) out;
6
7  void main() {
8     ...
9     if(gl_InvocationID == 0) {
10       // Set Inner Tessellation Level
11       gl_TessLevelInner[0] = u_TessLevelInner;
12
13       // Set Outer Tessellation Level
14       gl_TessLevelOuter[0] = u_TessLevelOuter;
15       gl_TessLevelOuter[1] = u_TessLevelOuter;
16       gl_TessLevelOuter[2] = u_TessLevelOuter;
17    }
18 }
```

# Tessellation Control Shader

Simple *Tessellation Control Shader* - quads:

```
1  // OpenGL 4.0
2  #version 400
3
4  // Quads
5  layout (vertices = 4) out;
6
7  void main() {
8     ...
9     if(gl_InvocationID == 0) {
10       // Set Inner Tessellation Level
11       gl_TessLevelInner[0] = u_TessLevelInner;
12       gl_TessLevelInner[1] = u_TessLevelInner;
13
14       // Set Outer Tessellation Level
15       gl_TessLevelOuter[0] = u_TessLevelOuter;
16       gl_TessLevelOuter[1] = u_TessLevelOuter;
17       gl_TessLevelOuter[2] = u_TessLevelOuter;
18       gl_TessLevelOuter[3] = u_TessLevelOuter;
19    }
20 }
```

# Tessellation Primitive Generator

The ***tessellation primitive generator*** (TPG) will produce output primitives for the tessellation stages - the behaviour of this stage is controlled by:

- The tessellation levels, usually defined by TCS.
- The spacing of the tessellated vertices defined by TES.
- The primitive type, defined by TES.
- The winding order, defined by TES.

Each generated vertex has parameter space coordinates in the *abstract patch*. These are passed onto the tessellation evaluation shader.

# Tessellation Evaluation Shader

The ***tessellation evaluation shader*** (TES) is invoked once *per-vertex* produced by the TPG.

The input to the TES is the abstract patch data produced by the TPG and the parameter space coordinates `gl_TessCoord` which define position of the vertex inside this patch.

The TES is responsible for computing the properties for each vertex that will be passed onto the fragment shader.

# Tessellation Evaluation Shader

The TES specifies several input layout qualifiers that determine the behaviour of the TPG. The qualifiers are defined at the start of the shader.

```
1 // OpenGL 4.0
2 #version 400
3
4 layout (param1, param2, ...) in;
```

- Abstract patch type - `isolines`, `triangles`, `quads`
- Spacing - `equal_spacing`, `fractional_even_spacing`, `fractional_odd_spacing`
- Winding Order - `cw`, `ccw`

# Tessellation Qualifiers

The ***abstract patch type*** defines what type of geometry will be generated for the TES.

The ***spacing*** defines how the inner and outer levels are distributed across the patch.

The ***winding order*** simply defines the winding order of the primitives output by the tessellation pipeline.

# Spacing

The three types of tessellation ***spacing*** provide different options for tessellating patches. The examples we have seen use `equal_spacing` which splits the lines into equal sections to evenly distribute the vertexes.

The tessellation level is rounded up to the nearest integer and the corresponding edge will be subdivided into that number of equal segments.

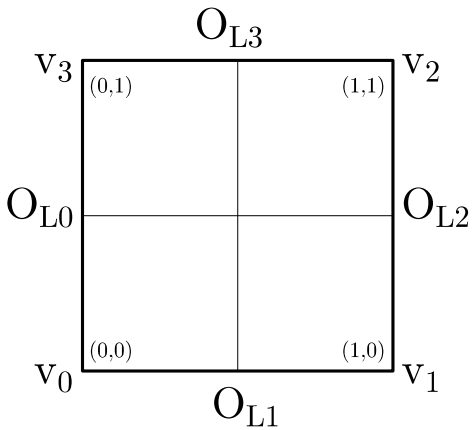
# Spacing

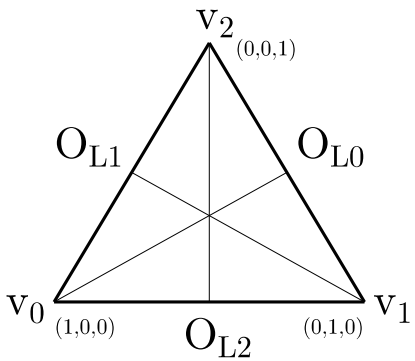
The fractional tessellation levels `fractional_odd_spacing` and `fractional_even_spacing` will round the tessellation up to the nearest odd and even integers (respectively) and divide the edge into that number of (possibly different) segments.

These spacing modes will segment the edges differently based on the fractional value of the tessellation level. This allows a smooth transition between different tessellation levels and avoids the problem of vertexes suddenly *popping* into existence.

# Tessellation Evaluation Shader

For triangle primitives, `gl_TessCoord` represents the vertex position as barycentric coordinates in the input patch. For quads, only the `x` and `y` components are valid and represent the position inside the input patch.



# Tessellation Evaluation Shader

Example: simple tessellation evaluation shader for triangles.

```
1  // OpenGL 4.0
2  #version 400
3
4  // Tessellation Qualifiers
5  layout (triangles, equal_spacing, ccw) in;
6
7  // Input from Evaluation Shader
8  in vec4 eval_Position[];
9
10 // Output to Fragment Shader
11 out vec4 frag_Position;
12
13 void main() {
14    // Calculate Position
15    vec3 pos0 = gl_TessCoord.x * eval_Position[0].xyz;
16    vec3 pos1 = gl_TessCoord.y * eval_Position[1].xyz;
17    vec3 pos2 = gl_TessCoord.z * eval_Position[2].xyz;
18
19    // Output to Fragment Shader
20    frag_Position = pos0 + pos1 + pos2;
21 }
```

# Tessellation Evaluation Shader

Example: simple tessellation evaluation shader for quads.

```
1  // OpenGL 4.0
2  #version 400
3
4  // Tessellation Qualifiers
5  layout (quads, equal_spacing, ccw) in;
6
7  // Input from Evaluation Shader
8  in vec4 eval_Position[];
9
10 // Output to Fragment Shader
11 out vec4 frag_Position;
12
13 void main() {
14    // Calculate Position
15    vec3 pos_a = mix(eval_Position[0].xyz, eval_Position[1].xyz, gl_TessCoord.x);
16    vec3 pos_b = mix(eval_Position[3].xyz, eval_Position[2].xyz, gl_TessCoord.x);
17
18    // Output to Fragment Shader
19    frag_Position = mix(pos_a, pos_b, gl_TessCoord.y);
20 }
```

# Level-of-Detail

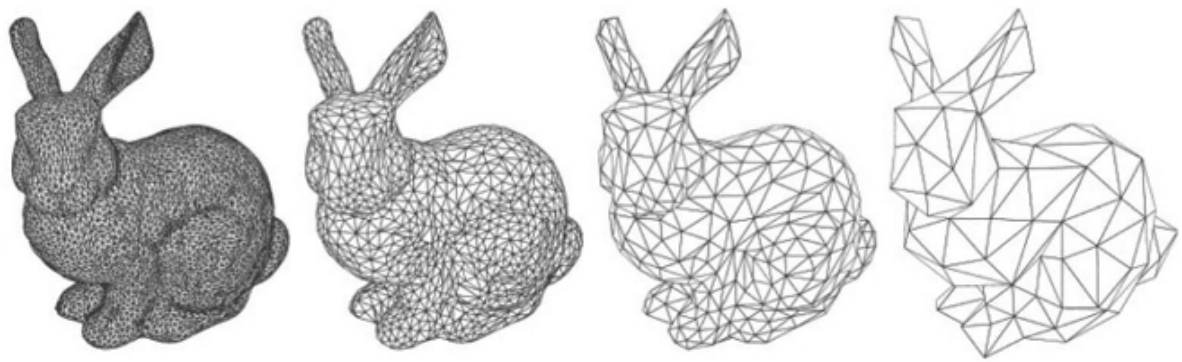
Hardware tessellation is one method that can be used to implement *level of detail* (LOD).

The basic idea is to use simpler versions of an object as it makes a smaller contribution to the rendered frame.

When the object occupies a large area of the screen it may be drawn with a very high quality model (with say millions of triangles). When the object takes up less space, drawing it at the same high detail would be wasteful as the details would not be visible.

# Level-of-Detail

An example of the same object represented by different models with four different levels of detail.



At long distance, these different models would have approximately the same appearance but the simple model would be much faster to render.

# Level-of-Detail

*Level-of-detail* algorithms generally consist of three major parts:

- ***generation*** - creates the different representations of the models.
- ***selection*** - chooses the level of detail based on some criteria.
- ***switching*** - changes the object from one level of detail to the next.

# Tessellation - Level-of-Detail

The hardware tessellation pipeline can be used to implement level-of-detail algorithms by selecting and generating the necessary level of detail for the model.

For example - a waving flag.

# Tessellation LOD - Generation

The *tessellation evaluation shader* can be used to dynamic *generate* the geometry of the flag at different levels of detail. For this example it can easily be achieved by tessellating a quad and offsetting the position of each vertex according to a simple sine wave function.

```
1 // Position
2 vec3 pos_a = mix(eval_Position[0].xyz, eval_Position[1].xyz, gl_TessCoord.x);
3 vec3 pos_b = mix(eval_Position[3].xyz, eval_Position[2].xyz, gl_TessCoord.x);
4 vec3 position = mix(pos_a, pos_b, gl_TessCoord.y);
5
6 // Add Sine wave
7 position.z += 0.1f*sin(position.x * 6.28f + u_time*2.0f);
```

# Tessellation LOD - Selection

The *tessellation control shader* can be used to *select* different levels of detail by choosing the appropriate tessellation levels.

One very simple approach for doing this is to look at the distance between the viewer and the vertexes.

```
1  void main () {
2     // Vertex Positions (eye space)
3     vec4 v0 = u_View * u_Model * ctrl_Position[0];
4     vec4 v1 = u_View * u_Model * ctrl_Position[1];
5     vec4 v2 = u_View * u_Model * ctrl_Position[2];
6     vec4 v3 = u_View * u_Model * ctrl_Position[3];
7
8     // Near & Far Planes (Projection Matrix)
9     float zNear = -0.2f;
10    float zFar = -50.0f;
11
12    // Vertex Distances
13    float z0 = (v0.z - zNear) / (zFar - zNear);
14    float z1 = (v1.z - zNear) / (zFar - zNear);
15    float z2 = (v2.z - zNear) / (zFar - zNear);
16    float z3 = (v3.z - zNear) / (zFar - zNear);
17    ...
```

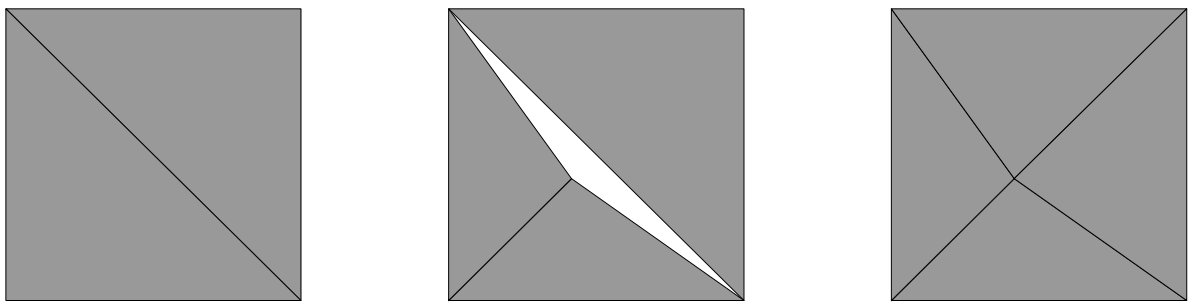
# Tessellation LOD - Selection

This distance can then be used to determine the tessellation levels:

```
1  float distanceTessLevel(float z0, float z1) {
2     float p = 25;
3     return max(pow(1.0f - z0, p), pow(1.0f - z1, p)) / 1.0f;
4  }
5
6  void main () {
7     ...
8     //-----------------------------------------------
9     // Calculate Tessellations Levels - Quad
10    float outer0 = distanceTessLevel(z0, z3);
11    float outer1 = distanceTessLevel(z1, z0);
12    float outer2 = distanceTessLevel(z2, z1);
13    float outer3 = distanceTessLevel(z3, z2);
14
15    // Calculate inner Tessellation Levels
16    float inner1 = max(max(outer0, outer1), max(outer2, outer3));
17    float inner2 = max(max(outer0, outer1), max(outer2, outer3));
18    ...
```

# Tessellation LOD - Selection

It is important to ensure that two different patches that share an edge will generate the same tessellation level for that edge.



To avoid this, it should be ensured that tessellation level calculations are consistent regardless of which patch is performing the calculation.

# Tessellation LOD - Switching

The *spacing* qualifier can be set to choose which *switching* method is used for the tesselllation.

The `even_spacing` qualifier will instantly switch between different levels of detail and may cause *popping*.

The fractional spacing qualifiers try to smooth the transition by splitting the edges unevenly so that new vertexes are added very close to existing ones.