Geometry and tessellation shaders

in OpenGL

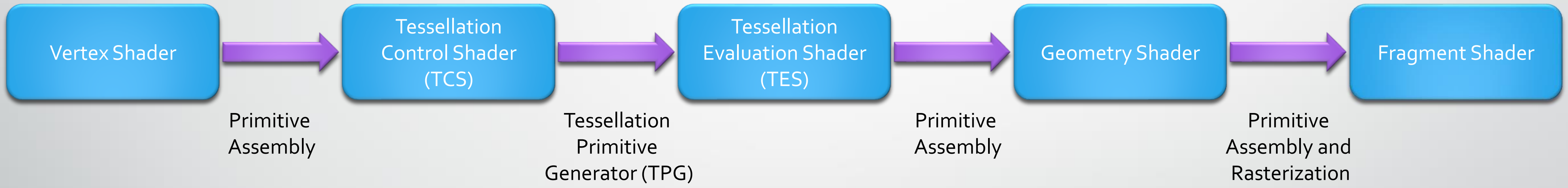
Introduction

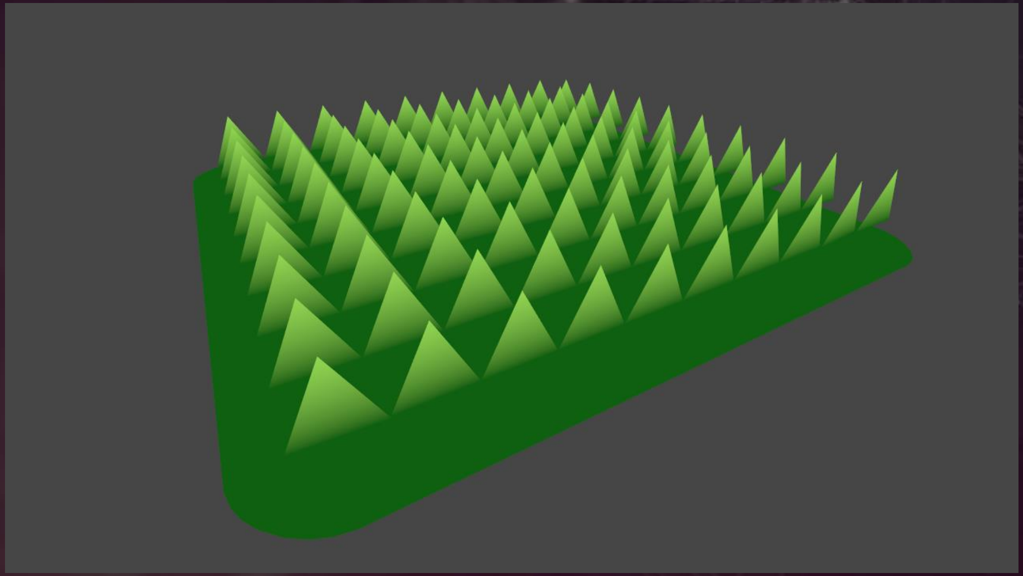
Geometry and tessellation shaders

Introduction

- Tessellation and geometry shaders provide programmers with additional ways to modify geometry as it progresses through the shader pipeline.
- Geometry shaders can be used to add, modify, or delete geometry in a very precise and user-controlled manner.
- Tessellation shaders can also be configured to automatically subdivide geometry to various degrees (levels of detail), potentially creating immensely dense geometry via the GPU.

The shader pipeline extended


Vertex Shader
Tessellation Control Shader (TCS)
Tessellation Evaluation Shader (TES)
Geometry Shader
Fragment Shader
Primitive Assembly
Tessellation Primitive Generator (TPG)
Primitive Assembly
Primitive Assembly and Rasterization



The geometry shader

In OpenGL

https://roystan.net/articles/grass-shader

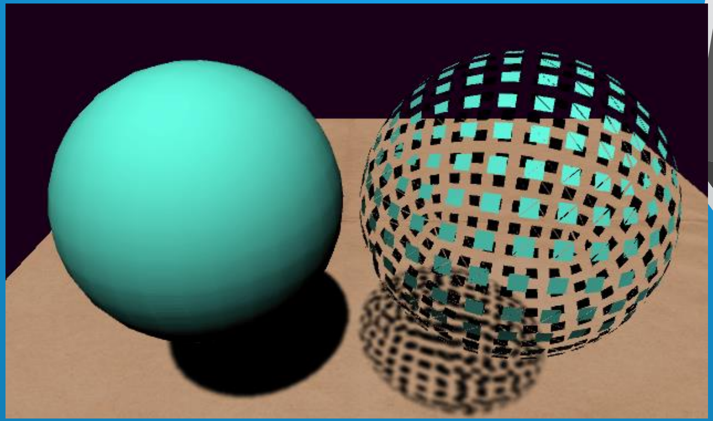
Geometry shader

- The **geometry shader (GS)** is designed to execute once for each primitive. It has access to all of the vertices of the primitive, as well as the values of any input variables associated with each vertex.
- If vertex shader provides an output variable, the geometry shader has access to the value of that variable for all vertices in the primitive. As a result, the input variables within the geometry shader are always arrays.
- The geometry shader can output zero, one, or more primitives. Those primitives need not be of the same kind that were received by the geometry shader.
- However, the GS can only output one primitive type. For example, a GS could receive a triangle, and output several line segments as a line strip, or a GS could receive a triangle and output zero or many triangles as a triangle strip.

Geometry shader

- A GS could be responsible for culling (removing) geometry based on some criteria, such as visibility based on occlusions.
- It could generate additional geometry to augment the shape of the object being rendered.
- The GS could simply compute additional information about the primitive and pass the primitive along unchanged
- The GS could produce primitives that are entirely different from the input geometry.

Geometry shader



- It has two main built-in functions, **EmitVertex** and **EndPrimitive**
- These two functions allow the GS to send multiple vertices and primitives down the pipeline. The GS defines the output variables for a particular vertex, and then calls **EmitVertex**
- After that, the GS can proceed to redefine the output variables for the next vertex, calls **EmitVertex** again and so on.
- After emitting all of the vertices for the primitive, the GS can call **EndPrimitive** to let the OpenGL system know that all the vertices of the primitive have been emitted
- The **EndPrimitive** function is implicitly called when the GS finishes execution. If GS does not call **EmitVertex** at all, the input primitive is effectively dropped (it is not rendered).

The Tessellation shaders

In OpenGL

Tessellation shaders

- When the tessellation shaders are active, we can only render one kind of primitive: the patch (**GL_PATCHES**)
- Rendering any other kind of primitive (such as triangles, or lines) while a tessellation shader is active is an error.
- The **patch primitive** is an arbitrary chunk of geometry (or any information) that is completely defined by the programmer. It has no geometric interpretation beyond how it is interpreted within the TCS and TES.
- The number of vertices within the patch primitive is also configurable.

Tessellation shaders

- The maximum number of vertices per patch is implementation-dependent, and can be queried via the following command:

```
glGetIntegerv(GL_MAX_PATCH_VERTICES, &maxVerts);
```

- We can define the number of vertices per patch with the following function:

```
glPatchParameteri( GL_PATCH_VERTICES, numPatchVerts );
```

- The patch primitive is never actually rendered; instead, it is used as additional information for the TCS and TES.

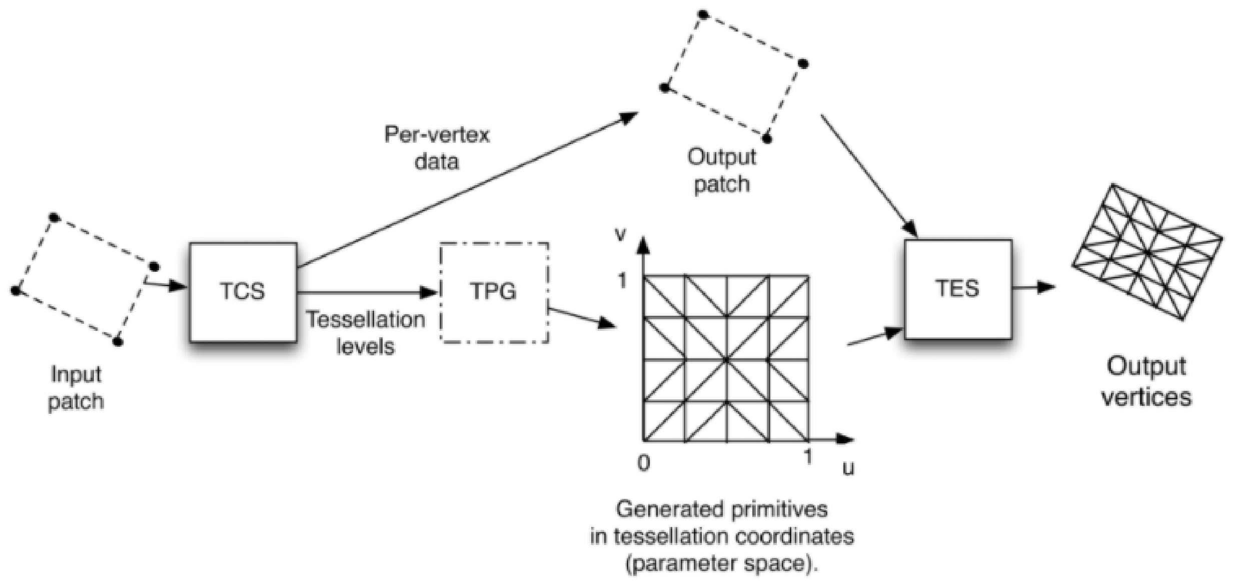
Tessellation primitive generator (TPG)

- The primitives that actually make their way further down the pipeline are created by the **tessellation primitive generator (TPG)**, which lies between the TCS and TES.
- Think of the tessellation-primitive generator as a configurable engine that produces primitives based on a set of standard tessellation algorithms.
- The **TCS** and **TES** have access to the entire input patch, but have fundamentally different responsibilities.

Tessellation primitive generator (TPG)

- The **TPG** generates primitives based on a particular algorithm (quads, isolines, or triangles).
- Each algorithm produces primitives in a slightly different fashion. Each vertex of the generated primitives is associated with a position in parameter space (**u, v, w**).
- Each coordinate of this position is a **number** that can range from **zero** to **one**. This coordinate can be used to evaluate the location of the vertex, often by interpolation of the patch primitive's vertices.
- The tessellation algorithms for **quads** and **isolines** make use of only the first two parametric coordinates: **u** and **v**.

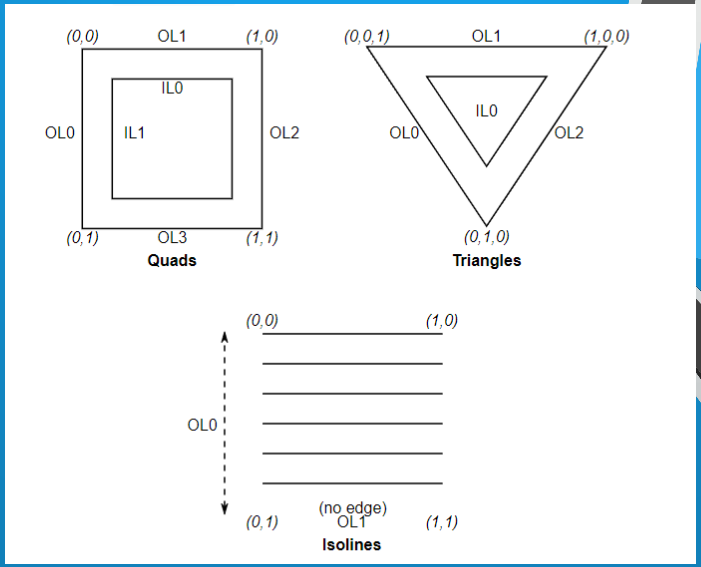
Tessellation primitive generator (TPG)



The process for an input and output patch consisting of 4 vertices (only u and v are used).

Image taken from: https://www.packtpub.com/product/opengl-4-shading-language-cookbook-third-edition/9781789342253

Tessellation control shader (TCS)



- The **Tessellation Control Shader (TCS)** has access to the entire input patch and is responsible for:
  - setting up the TPG
  - defining how the primitives should be generated by the TPG (how many and what algorithm to use)
  - producing per-vertex output attributes.
- The TCS is executed once for each vertex in a patch, but has access to all vertices of its associated patch
- It can **compute** additional information about the patch and pass it along to the TES using output variables.
- TCS main role is to tell the TPG how many primitives it should produce by defining tessellation levels via the **gl_TessLevelInner** and **gl_TessLevelOuter** (both arrays)

Tessellation evaluation shader (TES)

- The TES has the job of determining the position (and any other information) of each vertex of the primitives that are produced by the TPG.
  - For example, the TCS might tell the TPG to generate a line strip consisting of 100 line segments, and the TES is responsible for determining the position of each vertex of those 100 line segments.
- The TES is executed once for each parameter-space vertex that is generated by the TPG. The TES is actually the shader that defines the algorithm used by the TPG. It does so via its input layout qualifier.
- The TES uses the parametric coordinate (u,v) provided by the TPG along with the positions of all of the input patch vertices to determine the position of the vertex

Point sprites

with the geometry shader

Point sprites

- **Point sprites** are simple quads (usually texture mapped) that are aligned such that they are always facing the camera. They are very useful for particle systems in 3D.
- The point sprites are specified by the OpenGL application as single-point primitives, via the GL_POINTS rendering mode.
- This simplifies the process, because the quad itself and the texture coordinates for the quad are determined automatically. The OpenGL side of the application can effectively treat them as point primitives, avoiding the need to compute the positions of the quad vertices.
- The quad and texture coordinates are generated automatically (within the geometry shader) and aligned to face the camera

Point sprites

- When rendering point primitives using this mode, the points are rendered as screen-space squares that have a diameter (side length) as defined by the **glPointSize** function.
- OpenGL will automatically generate texture coordinates for the fragments of the square. These coordinates run from zero to one in each direction (horizontal and vertical), and are accessible in the fragment shader via the **gl_PointCoord**, built-in variable
- **glPointParameter** can define the origin of the automatically-generated texture coordinates or the alpha value for points when multi-sampling is enabled.

Point sprites

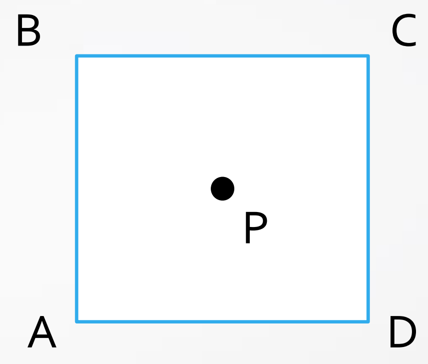
- The built-in support for point sprites does not allow the programmer to rotate the screen-space squares, or define them as different shapes but you can achieve similar effects with the creative use of textures and transformations of the texture coordinates
- We could transform the texture coordinates using a rotation matrix to create the look of a rotating object even though the geometry itself is not actually rotating.
- In addition, the size of the point sprite is a screen-space size. The point size must be adjusted with the depth of the point sprite if we want to get a perspective effect (sprites get smaller with distance).

Point sprites

- If these (and possibly other) issues make the default support for point sprites too limiting, we can use the geometry shader to generate our point sprites.
- In fact, this technique is a good example of using the geometry shader to generate different kinds of primitives than it receives. The basic idea here is that the geometry shader will receive point primitives (in camera coordinates) and will output a quad centred at the point and aligned so that it is facing the camera.
- The geometry shader will also automatically generate texture coordinates for the quad.
- We could generate other shapes, such as hexagons, or we could rotate the quads before they are output from the geometry shader.

Point sprites implementation

- Given a point P in camera coordinates we can generate by simply translating P in a plane parallel to the x-y plane:

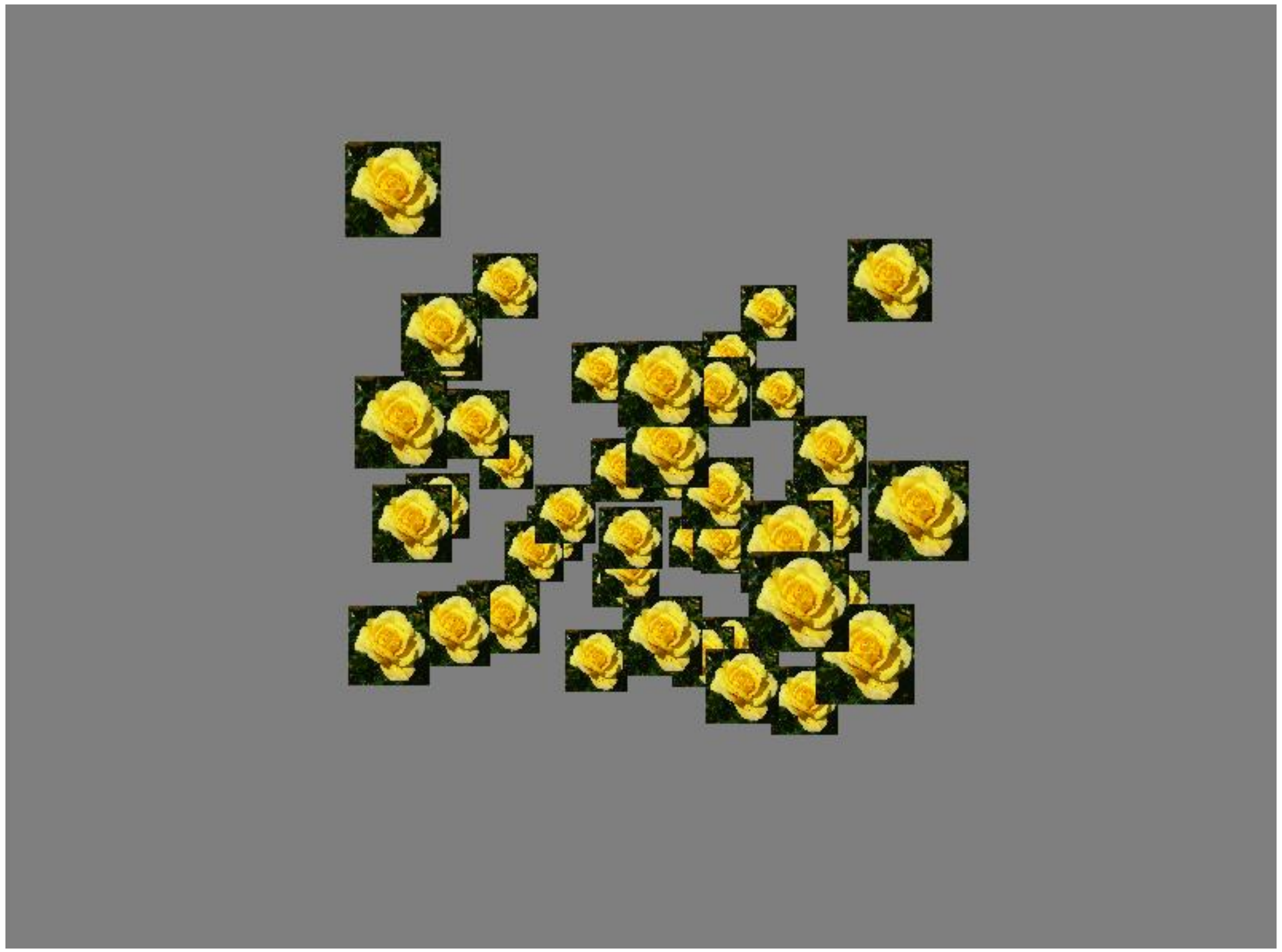


$$A = P + \left(-\frac{w}{2}, -\frac{w}{2}, 0\right)$$

$$B = P + \left(-\frac{w}{2}, \frac{w}{2}, 0\right)$$

$$C = P + \left(\frac{w}{2}, \frac{w}{2}, 0\right)$$

$$D = P + \left(\frac{w}{2}, -\frac{w}{2}, 0\right)$$



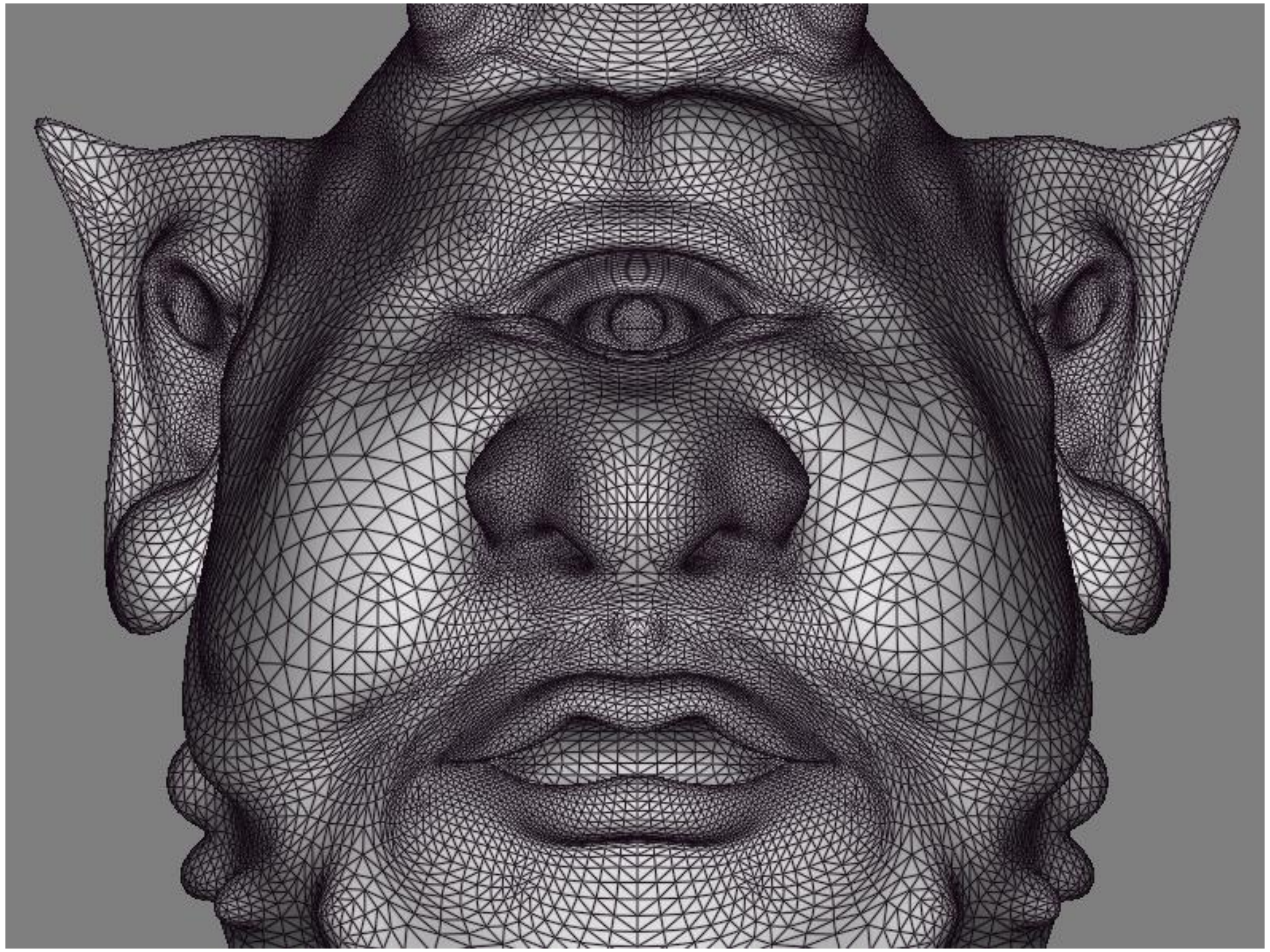
Point sprites implementation

Wireframe

on top of a shaded mesh

Wireframe

- We can use the geometry shader to produce a different variety of primitives than it received. Geometry shaders can also be used to provide additional information to later stages. Because we have access to all of the vertices of the primitive at once, and can do computations based on the entire primitive rather than a single vertex.
- This geometry shader that does not modify the triangle at all. It essentially passes the primitive along unchanged. However, it computes additional information about the triangle that will be used by the fragment shader to highlight the edges of the polygon. The basic idea here is to draw the edges of each polygon directly on top of the shaded mesh.
- This technique comes from an NVIDIA whitepaper published in 2007 : NVIDIA whitepaper in 2007 (**Solid Wireframe, NVIDIA Whitepaper WP-03014-001_V01** available at developer.nvidia.com).
- We make use of the geometry shader to produce the wireframe and shaded surface in a single pass. We also provide some simple anti-aliasing of the mesh lines that are produced



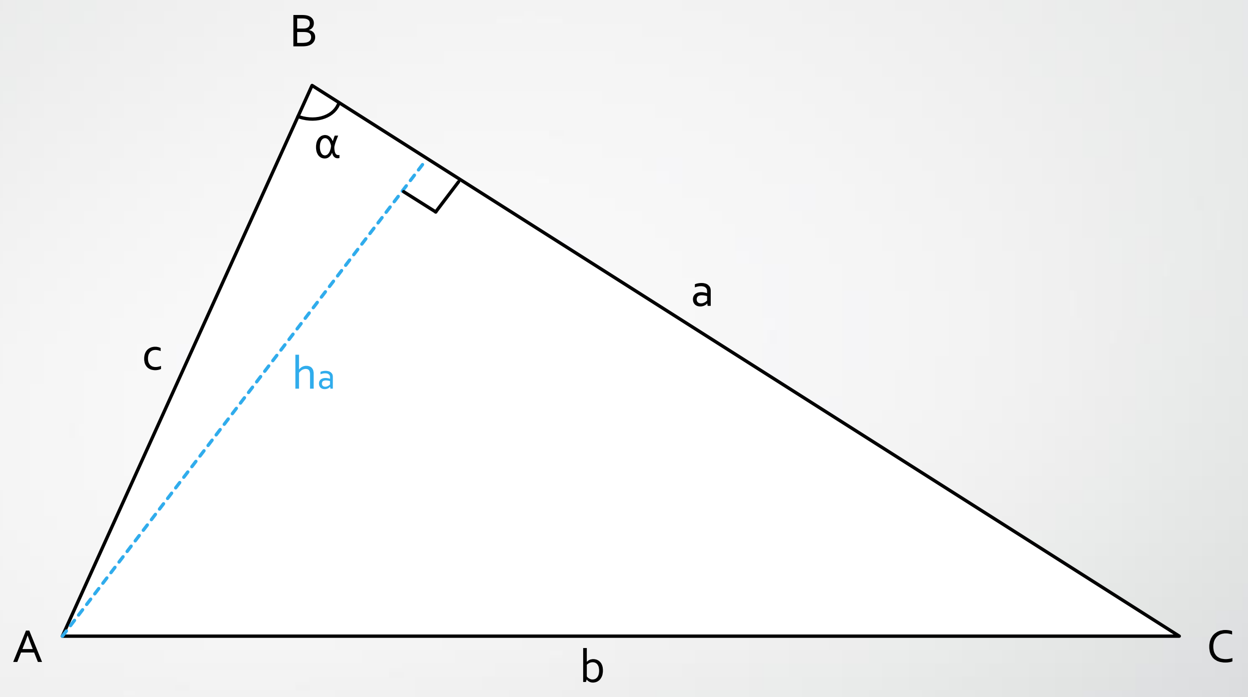
Wireframe

Wireframe

- To render the wireframe on top of the shaded mesh, we'll compute the distance from each fragment to the nearest triangle edge. When the fragment is within a certain distance from the edge, it will be shaded and mixed with the edge colour. Otherwise, the fragment will be shaded normally.

- To compute the distance from a fragment to the edge, we use the following technique. In the geometry shader, we compute the minimum distance from each vertex to the opposite edge (also called the **triangle altitude**). The distances are h_a, h_b and h_c

Wireframe

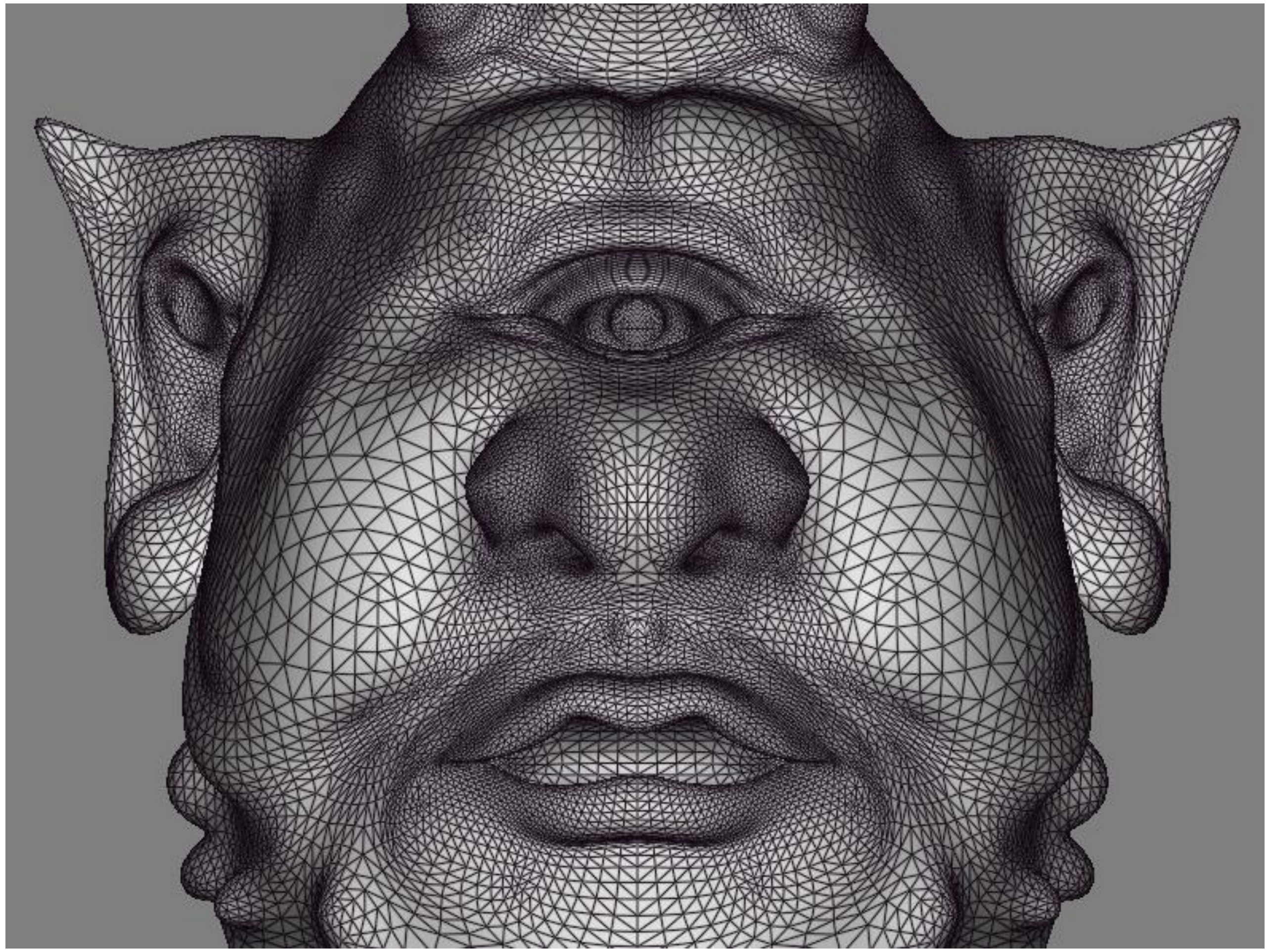


$h_a = c \sin\alpha$

$\alpha = cos^{-1}(\frac{a^2+c^2-b^2}{2ac})$

Wireframe

- Within the fragment shader, all we need to do is find the minimum of the three distances, and if that distance is less than the line width, we mix the fragment colour with the line colour.
- We'll scale the intensity of the line in a two-pixel range around the edge of the line. Pixels that are at a distance of one or less from the true edge of the line get 100% of the line colour, and pixels that are at a distance of one or more from the edge of the line get 0% of the line colour.
- The edge of the line itself is a configurable distance we'll call it Line.width from the edge of the polygon.



Wireframe

Silhouette lines

with the geometry shader

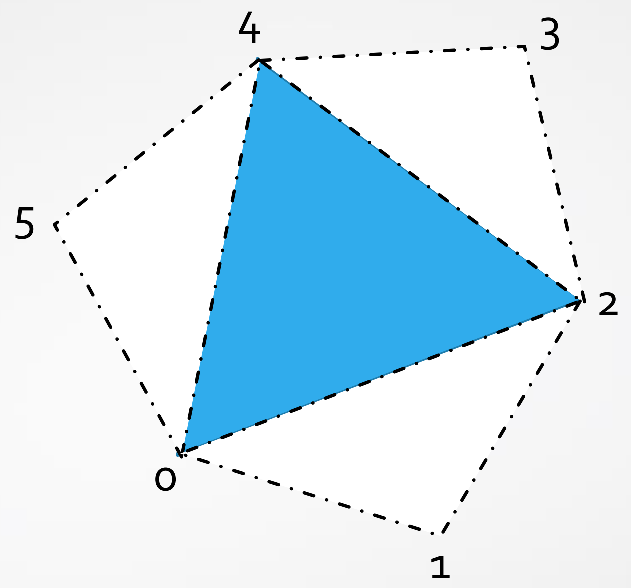
Silhouette lines

- In this example will cover drawing black outlines around the edges of a model using the geometry shader
- The geometry shader will approximate these lines by generating small, skinny quads aligned with the edges that make up the silhouette of the object.
- This is based on the technique published here: https://prideout.net/blog/?p=54 by Philip Rideout
- We'll implement the technique using a single pass without anti-aliasing or custom depth testing.
- One of the most important features of the geometry shader is that it allows us to provide additional vertex information beyond just the primitive being rendered.

Silhouette lines

- OpenGL has several additional primitive rendering modes called **adjacency modes.** They allow additional vertex data to be associated with each primitive.
- Typically, this additional information is related to the nearby primitives within a mesh, but there is no requirement that this be the case.
- The following list includes the adjacency modes:
  - **GL_LINES_ADJACENCY**: This mode defines lines with adjacent vertices (four vertices per line segment)
  - **GL_LINE_STRIP_ADJACENCY**: This mode defines a line strip with adjacent vertices (for n lines, there are n+3 vertices)
  - **GL_TRIANGLES_ADJACENCY**: This mode defines triangles along with vertices of adjacent triangles (six vertices per primitive)
  - **GL_TRIANGLE_STRIP_ADJACENCY**: This mode defines a triangle strip along with vertices of adjacent triangles (for n triangles, there are 2(n+2) vertices provided)

Silhouette lines



- The solid blue represents the triangle itself, and the dotted lines represent adjacent triangles. The first, third, and fifth vertices (0, 2, and 4) make up the triangle itself. The second, fourth, and sixth are vertices that make up the adjacent triangles.
- Mesh data is not usually provided in this form, so we need to pre-process our mesh to include the additional vertex information. Typically, this only means **expanding** the element **index array by a factor of two**. The position, normal, and texture coordinate arrays can remain unchanged.

Silhouette lines

- When a mesh is rendered with adjacency information, the geometry shader has access to all six vertices associated with a particular triangle. We can then use the adjacent triangles to determine whether a triangle edge is part of the silhouette of the object.
- The basic assumption is that an edge is a **silhouette edge** if the triangle is front-facing and the corresponding adjacent triangle is not front-facing.
- We can determine whether a triangle is front-facing within the geometry shader by computing the triangle's normal vector (using a cross product).
- If we are working within eye coordinates (or clip coordinates), the z coordinate of the normal vector will be positive for front-facing triangles.

Silhouette lines

- For a triangle with vertices A, B, and C, the z coordinate of the normal vector is given by the following equation:

$$n_z = (A_xB_y - B_xA_y) + (B_xC_y - C_xB_y) + (C_xA_y - A_xC_y)$$

- Once we determine which edges are silhouette edges, the geometry shader will produce additional skinny quads aligned with the silhouette edge.
- These quads, taken together, will make up the desired dark lines. After generating all the silhouette quads, the geometry shader will output the original triangle.
- In order to render the mesh in a single pass with appropriate shading for the base mesh, and no shading for the silhouette lines, we'll use an additional output variable. This variable will let the fragment shader know when we are rendering the base mesh and when we are rendering the silhouette edge.

Bloom effect implementation

Tessellating a curve

In OpenGL

Tessellating a curve

- We'll use tessellation for drawing a curve, a **cubic Bezier curve**.
- A **Bezier curve** is a parametric curve defined by four control points. The control points define the overall shape of the curve.
- The **first** and **last** of the four points define the start and end of the curve, and the **middle points** guide the shape of the curve, but do not necessarily lie directly on the curve itself.
- The curve is defined by interpolating the four control points using a set of blending functions. The blending functions define how much each control point contributes to the curve for a given position along the curve.

Tessellating a curve

- We will draw a cubic Bezier curve, which involves four control points (n = 3):

$$P(t) = B_0^3(t)P_0 + B_1^3(t)P_1 + B_2^3(t)P_2 + B_3^3(t)P_3$$

- Cubic Bernstein polynomials:

$$B_0^3(t) = (1 - t)^3$$
$$B_1^3(t) = 3(1 - t)^2 t$$
$$B_2^3(t) = 3(1 - t)t^2$$
$$B_3^3(t) = t^3$$

Tessellating a curve

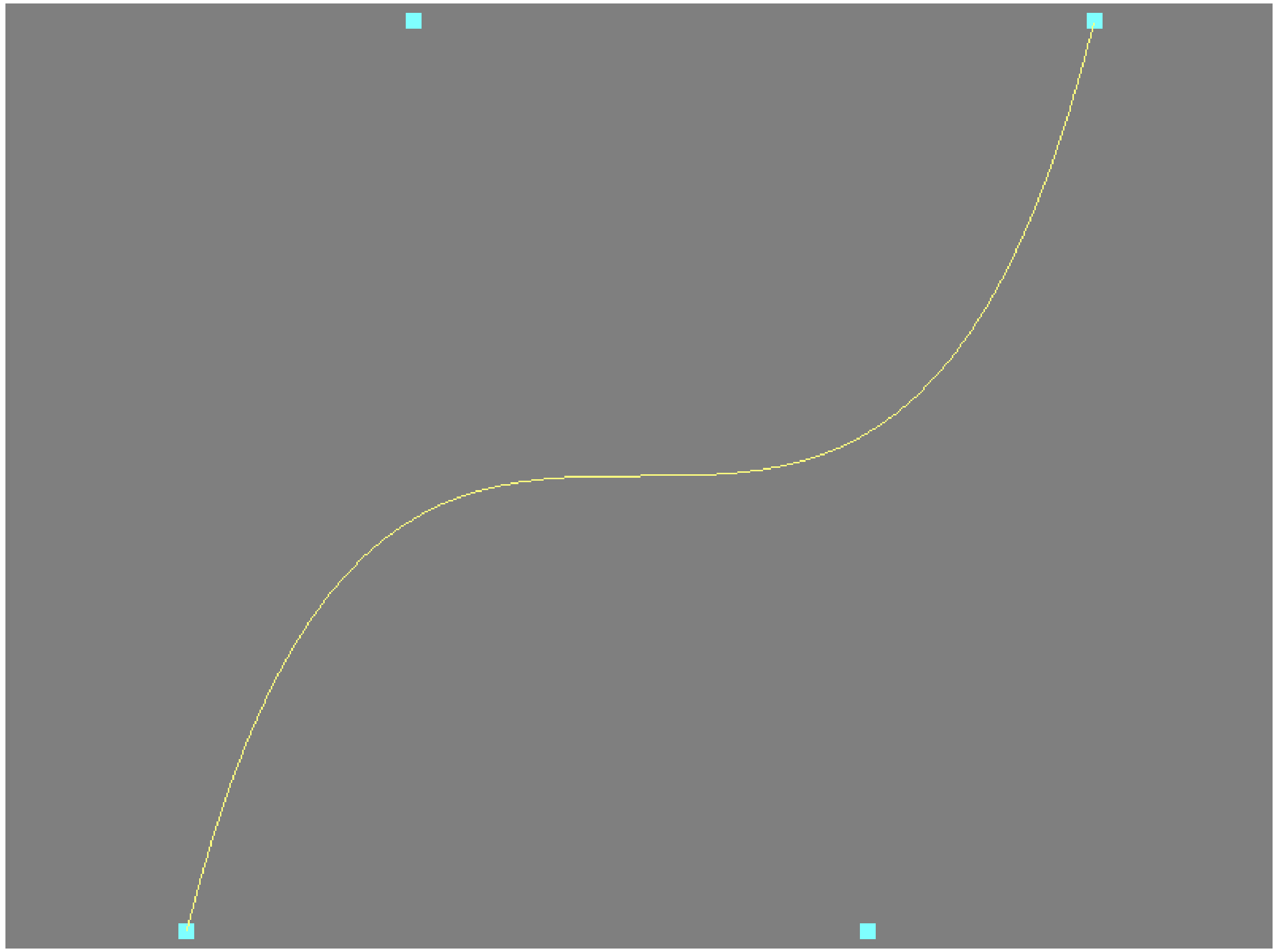
- We'll define the number of line segments for our Bezier curve within the **TCS** (by defining the outer tessellation levels), and evaluate the Bezier curve at each particular vertex location within the **TES**.
- The control points for the Bezier curve are sent down the pipeline as a **patch primitive** consisting of four vertices.
- **A patch primitive** is a programmer-defined primitive type. Basically, it is a set of vertices that can be used for anything that the programmer chooses.
- The **TCS** is executed once for each vertex within the patch, and the **TES** is executed, a variable number of times, depending on the number of vertices produced by the **TPG**. The final output of the tessellation stages is a set of primitives. In our case, it will be a line strip.

Tessellating a curve

- Part of the job of the **TCS** is to define the tessellation level, the number of vertices that will be generated.
- In our case, the **TCS** will be generating a line strip, so the tessellation level is the number of line segments in the line strip.
- Each vertex that is generated for this line strip will be associated with a tessellation coordinate that will vary between zero and one. We'll refer to this as the **u coordinate**, and it will correspond to the parametric **t parameter** in the preceding Bezier curve equation.
- The **TCS** will trigger the generation of a set of line strips called **isolines**. Each vertex in this set of isolines will have a **u** and a **v coordinate**.
- The number of distinct values of u and v is associated with two separate tessellation levels, the so-called **outer levels**. For this example, however, we'll only generate a single line strip, so the second tessellation level (for v) will always be one.

Tessellating a curve

- Within the TES, the main task is to determine the position of the vertex associated with this execution of the shader.
- We have access to the u and v coordinates associated with the vertex, and we also have (read-only) access to all of the vertices of the patch.
- We can then determine the appropriate position for the vertex by using the parametric equation, with u as the parametric coordinate (t in the preceding equation).



Tessellating a curve

Tessellating a quad

In OpenGL

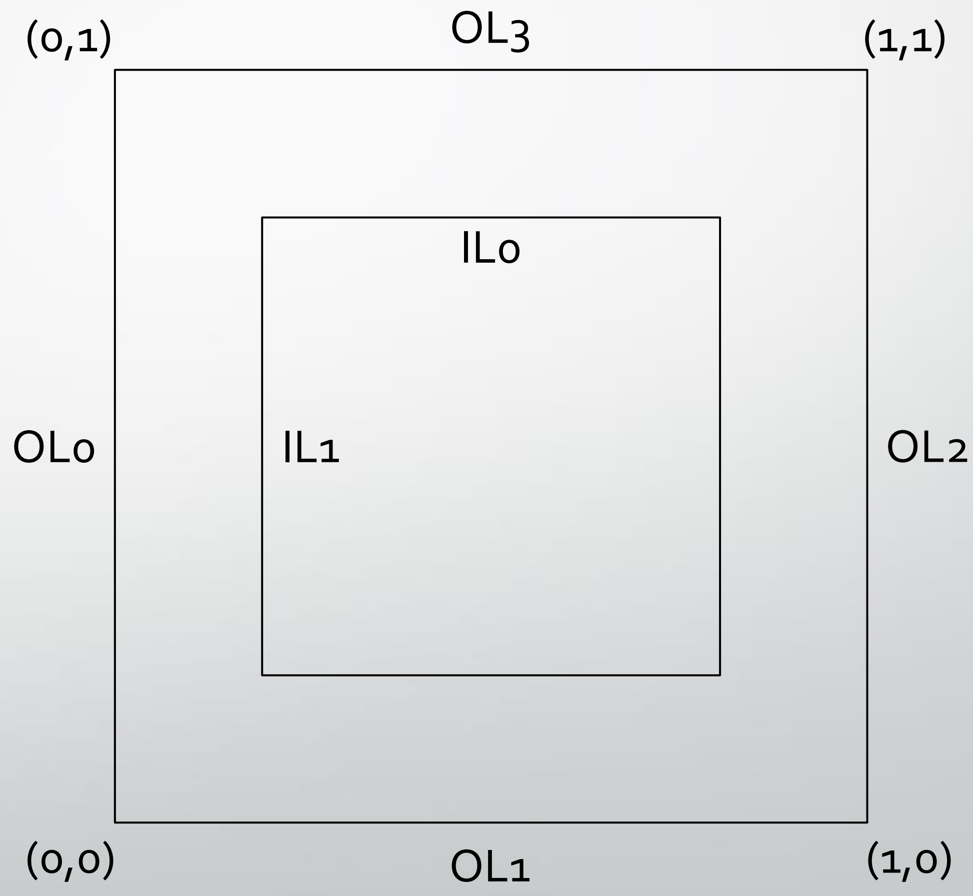
Tessellating a quad

When using **quad tessellation**, the tessellation primitive generator subdivides (u,v) parameter space into a number of subdivisions based on six parameters:

- **Outer level 0 (OL0)**: This is the number of subdivisions along the **v** direction where **u = 0**
- **Outer level 1 (OL1)**: This is the number of subdivisions along the **u** direction where **v = 0**
- **Outer level 2 (OL2)**: This is the number of subdivisions along the **v** direction where **u = 1**
- **Outer level 3 (OL3)**: This is the number of subdivisions along the **u** direction where **v = 1**
- **Inner level 0 (IL0)**: This is the number of subdivisions along the **u** direction for all internal values of **v**
- **Inner level 1 (IL1)**: This is the number of subdivisions along the **v** direction for all internal values of **u**

Tessellating a quad

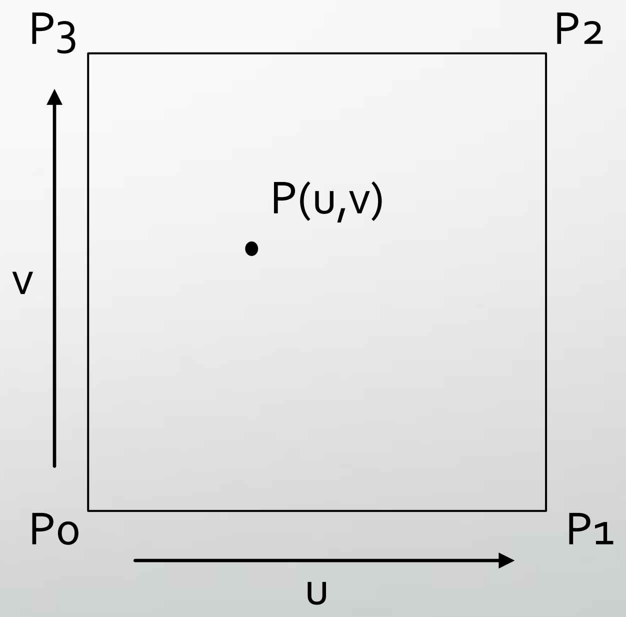
- These are the **inner tessellation levels** for u and v **inner level 0** and **inner level 1**) - gl_TessLevelInner
- The **outer tessellation levels** for u and v along both edges (**outer levels 0 to 3**) - gl_TessLevelOuter

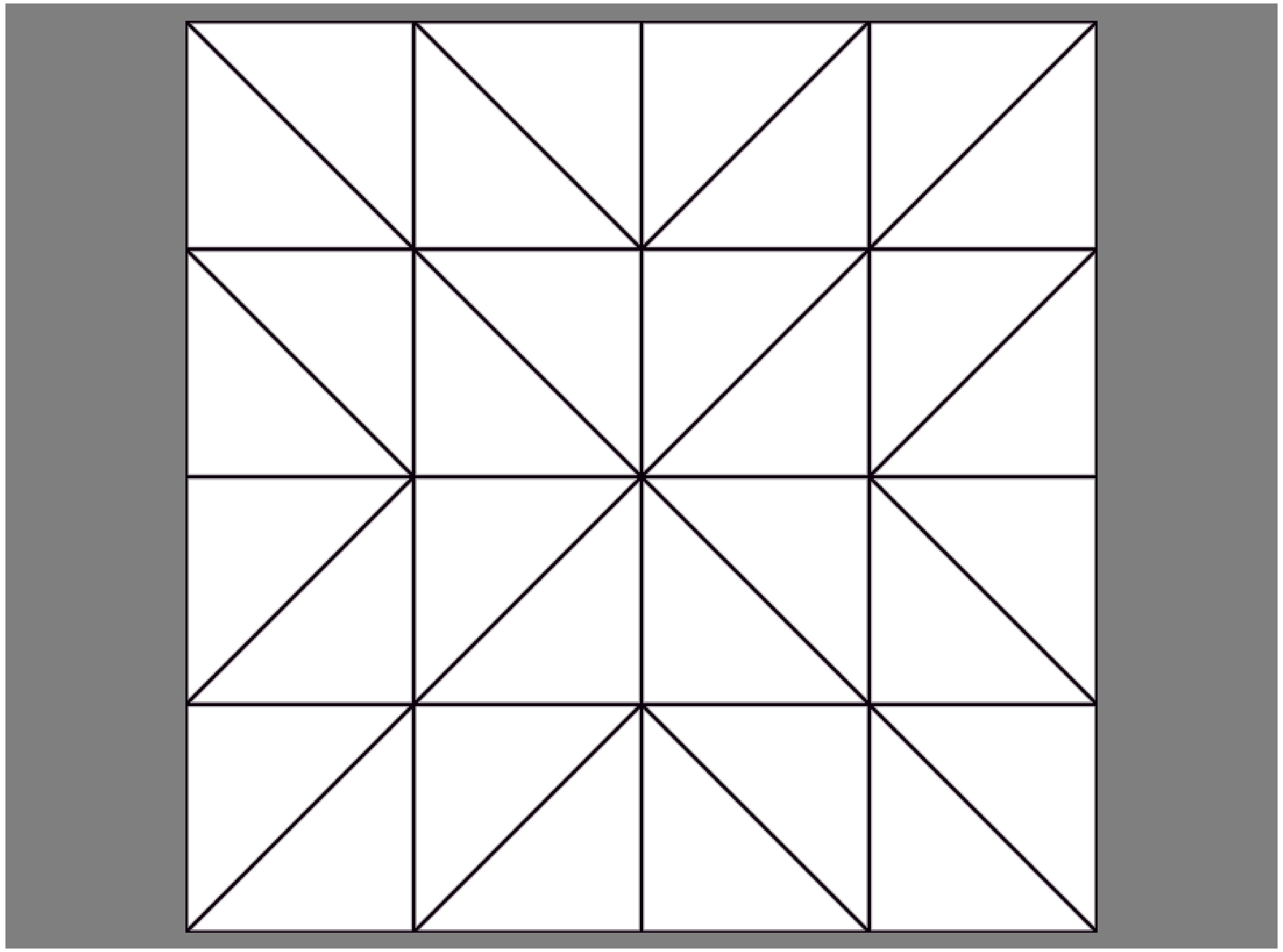


Tessellating a quad

Linear interpolation of a point P:

$$P(u,v) = P_0\,(1-u)(1-v) + P_1 u(1-v) + P_2 uv + P_3 v(1-u)$$





Tessellating a quad

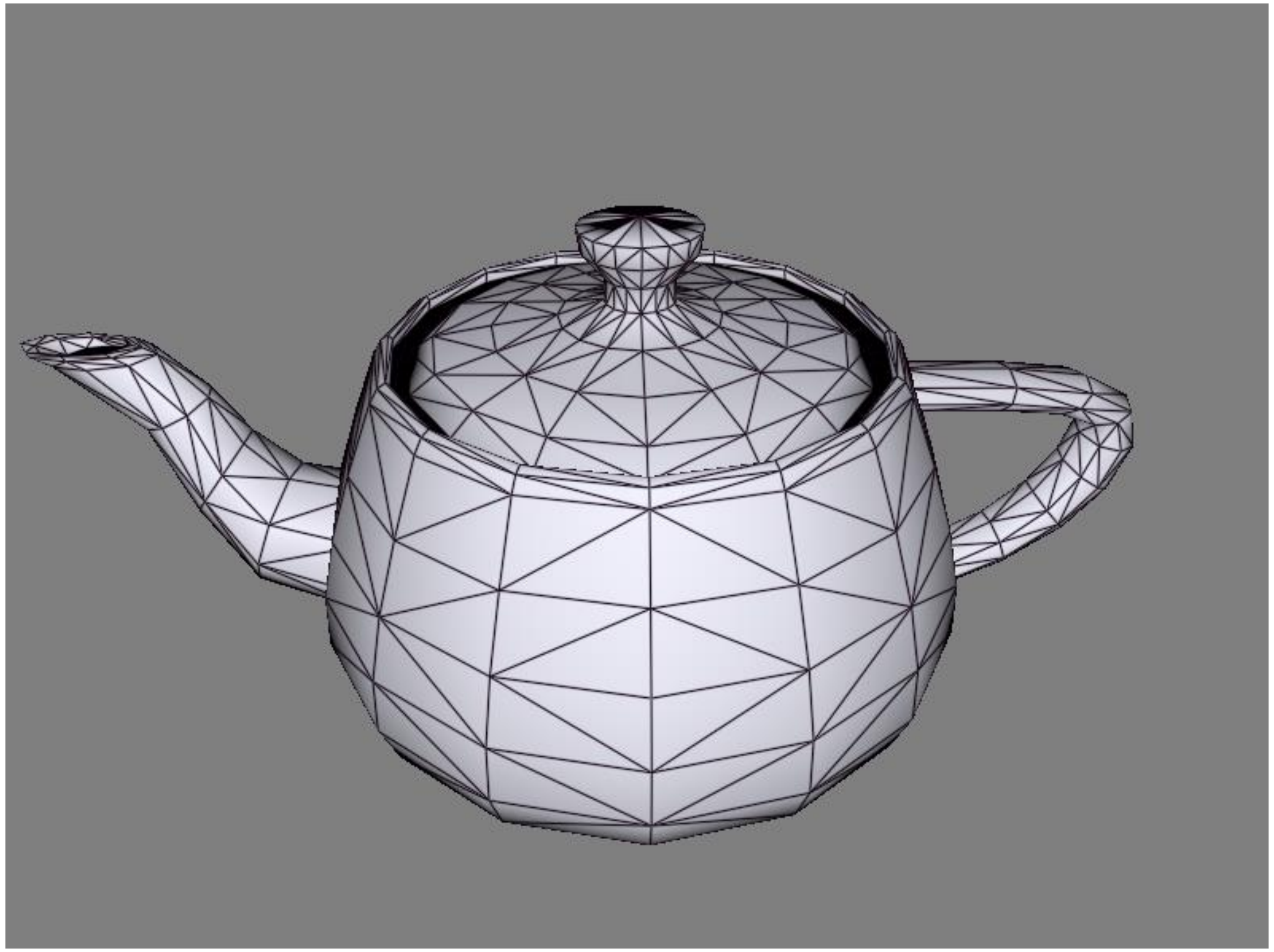
Tessellating a 3D surface

In OpenGL

Tessellating a 3D surface

- Teapot's dataset is actually defined as a set of 4 x 4 patches of control points, suitable for cubic Bezier interpolation. Therefore, drawing the teapot really comes down to drawing a set of cubic Bezier surfaces.
- We'll render each patch of 16 vertices as a patch primitive
- We use quad tessellation to subdivide the parameter space
- We implement the Bezier interpolation within the tessellation evaluation shader.
- The surface is defined by a set of 16 control points (laid out in a 4 x 4 grid) Pij, with i and j ranging from 0 to 3, the parametric Bezier surface is given by the following equation:

$$P(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i^3(u) B_j^3(v) P_{ij}$$



Tessellating a 3D surface

Useful links

- To read - Chapter 11 Geometry shaders (OpenGL Superbible – see link on the DLE)
- To read – Geometry shaders: https://learnopengl.com/Advanced-OpenGL/Geometry-Shader
- To read: Using Geometry and Tessellation shaders (OpenGL 4 Shading Language Cookbook).
- Primitives: https://www.khronos.org/opengl/wiki/Primitive
- glPointSize: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glPointSize.xhtml
- gl_PointCoord: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/gl_PointCoord.xhtml
- glPointParameter: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glPointParameter.xhtml
- Bernstein polynomial: https://en.wikipedia.org/wiki/Bernstein_polynomial
- Tessellation explanation: https://stackoverflow.com/questions/37647181/how-triangle-patch-tessellation-is-done-using-edge-and-inner-tessellation-factor