Why do we need a Tessellation step right in the pipeline?

- You can perform adaptive subdivision based on a variety of criteria (size, curvature, etc.)
- You can provide coarser models, but have finer ones displayed (≈ geometric compression)
- You can apply detailed displacement maps without supplying equally detailed geometry
- You can apply detailed normal maps without supplying equally detailed geometry
- You can adapt visual quality to the required level of detail
- You can create smoother silhouettes
- You can do all of this, and someone else will supply the geometric patterns for you!

What built-in patterns can the Tessellation shaders produce?
- Lines
- Quads (subsequently broken into triangles)
- Triangles

Tessellation Shader Organization

The Tessellation Control Shader (TCS) transforms the input coordinates to a regular surface representation. It also computes the required tessellation level based on distance to the eye, screen space spanning, hull curvature, or displacement roughness. There is one invocation per output vertex.

The Fixed-Function Tessellation Primitive Generator (TPG) generates semi-regular u-v-w coordinates in specific patterns. (In fact, if it had been up to me, this would have been called the Tessellation Pattern Generator.)

The Tessellation Evaluation Shader (TES) evaluates the surface in u,v,w coordinates. It interpolates attributes and applies displacements. There is one invocation per generated vertex.

There is a new “Patch” primitive – it is the face and its neighborhood: `glBegin( GL_PATCHES ); // # of vertices in each patch` `gl Vertex3f( ... );` `glEnd();` These have no implied topology – they will be given to you in an array. It’s up to your shader to interpret the order.

In the OpenGL Program

```glsl
#version 400
#extension GL_ARB_tessellation_shader : enable

GLuint tcs = glCreateShader( GL_TESS_CONTROL_SHADER );
GLuint tes = glCreateShader( GL_TESS_EVALUATION_SHADER );

if you have a TCS, you must also have a Vertex Shader

GLuint tcs = glCreateShader( GL_TESS_CONTROL_SHADER );
GLuint tes = glCreateShader( GL_TESS_EVALUATION_SHADER );

if you have a TCS, you must also have a Vertex Shader

In the OpenGL Program:

```
TCS Inputs

- `gl_in[]` is an array of structures:
  ```
  struct {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[6];
  } gl_in[n];
  ```
- `glInvocationID` tells you which output vertex you are working on. This must be the index into the `gl_in[]` array.
- `gl_PatchVerticesIn` is the number of vertices in each patch and the dimension of `gl_in[]`.
- `gl_PrimitiveID` is the number of primitives since the last ` glBegin()` (the first one is #0).

TCS Outputs

- `gl_out[]` is an array of structures:
  ```
  struct {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[6];
  } gl_out[n];
  ```
- `layout( vertices = n ) out;` Used to specify the number of vertices output to the TPG.

In the TCS

- User-defined variables defined per-vertex are qualified as "out".
- User-defined variables defined per-patch are qualified as "patch out".

Defining how many vertices this patch will output:

- `layout( vertices = n ) out;`

TES Inputs

- Reads one triplet of 0. <= (u,v,w) <= 1. coordinates in the built-in variable `vec3 gl_TessCoord`.
- User-defined variables defined per-vertex are qualified as "out".
- User-defined variables defined per-patch are qualified as "patch out".

- `gl_in[]` is an array of structures coming from the TCS:
  ```
  struct {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[6];
  } gl_in[n];
  ```

TES Output Topologies: the Quad Pattern

- `gl_TessLevelOuter[4]` is an array containing up to 4 outside edges of tessellation levels.
- `gl_TessLevelInner[2]` is an array containing up to 2 inside edges of tessellation levels.

Tessellation Primitive Pattern Generator (TPG)

- The TPG is "fixed-function", i.e., you can’t change its operation except by setting parameters.
- The TPG consumes all vertices from the TCS and emits vertices for the triangles, quads, or isolines patterns.
- The TPG outputs a series of vertices as coordinates in barycentric, i.e., in terms of the parameters (u,v,w).
- Really, only (u,v) are unique: for triangles w = 1. – u – v.
- Just (u,v) are used for quads and isolines patterns.
Examples

In these examples:

1. I am using glman to run them. The only necessary input files are the glman.glib file and the shader files. If you aren’t using glman, you can do this from a full OpenGL program.

2. All of the surface examples use the Geometry Shader triangle-shrink shader. This isn’t necessary, but is educational to really see how much and where the surfaces have been tessellated.

Example: A Bézier Curve

\[ P(u) = (1-u)^3P_0 + 3u(1-u)^2P_1 + 3u^2(1-u)P_2 + u^3P_3 \]

1. You program the Tessellation Control Shader to decide how much to tessellate the curve based on screen area, curvature, etc.

   Can even tessellate non-uniformly if you want, such as using more points where the curvature is higher

   The OpenGL tessellation can also do 1D curves. Just set \( OL0 = 1 \).

2. The Tessellation Primitive Generator generates \( u,v,w \) values for as many subdivisions as the TCS asked for.
Example: A Bézier Curve

3. The Tessellation Evaluation Shader computes the x, y, z coordinates based on the TPG's u values.

The basis functions:

\[ P(u) = u^3P_0 + 3u^2(1-u)P_1 + 3u(1-u)^2P_2 + (1-u)^3P_3 \]

where \( P \) is actually \( (x, y, z) \).

In an OpenGL Program:

```glsl
glPatchParameteri(GL_PATCH_VERTICES, 4);
gBegin(GL_PATCHES);
gVertex3f(x0, y0, z0);
gVertex3f(x1, y1, z1);
gVertex3f(x2, y2, z2);
gVertex3f(x3, y3, z3);
gEnd();
```

In a .glib File:

```glsl
#version 400
#extension GL_ARB_tessellation_shader: enable
uniform int uOuter0, uOuter1;
layout( vertices = 4 ) out;
void main()
{
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
    gl_TessLevelOuter[0] = float(uOuter0);
    gl_TessLevelOuter[1] = float(uOuter1);
}
```

In the TCS Shader:

```glsl
#version 400
#extension GL_ARB_tessellation_shader: enable
layout(isolines, equal_spacing) in;
void main()
{
    vec4 p0 = gl_in[0].gl_Position;
    vec4 p1 = gl_in[1].gl_Position;
    vec4 p2 = gl_in[2].gl_Position;
    vec4 p3 = gl_in[3].gl_Position;
    float u = gl_TessCoord.x;
    float b0 = (1.0 - u) * (1.0 - u) * (1.0 - u);
    float b1 = 3.0 * u * (1.0 - u) * (1.0 - u);
    float b2 = 3.0 * u * u * (1.0 - u);
    float b3 = u * u * u;
    gl_Position = b0*p0 + b1*p1 + b2*p2 + b3*p3;
}
```

Assigning the intermediate pi's is here to make the code more readable. From what I have seen, the compiler will optimize this away.

Example: A Bézier Curve
Example: A Bézier Surface

Bézier Surface Parametric Equations

\[ P(u,v) = \left( 1-v \right)^2 u^2 \left( \begin{array}{c} P_{10} \\ P_{20} \\ P_{30} \\ P_{00} \end{array} \right) + \left( 1-u \right)^2 v^2 \left( \begin{array}{c} P_{10} \\ P_{20} \\ P_{30} \\ P_{00} \end{array} \right) + (1-u)(1-v) u v \left( \begin{array}{c} P_{10} \\ P_{20} \\ P_{30} \\ P_{00} \end{array} \right) \]

In an OpenGL Program

```c
void main()
{
    gl_out[glInvocationID].gl_Position = gl_in[glInvocationID].gl_Position;
    gl_TessLevelOuter[0] = gl_TessLevelOuter[2] = uOuter02;
    gl_TessLevelInner[0] = uInner0;
    gl_TessLevelInner[1] = uInner1;
}
```

In the TCS Shader

```c
uniform float uOuter02, uOuter13, uInner0, uInner1;
layout( vertices = 16 ) out;
void main()
{
    gl_out[glInvocationID].gl_Position = gl_in[glInvocationID].gl_Position;
}
```

In the TES Shader

```c
layout( quads, equal_spacing, ccw ) in;
out vec3 teNormal;
void main()
{
    vec4 p00 = gl_in[0].gl_Position;
    vec4 p10 = gl_in[1].gl_Position;
    vec4 p20 = gl_in[2].gl_Position;
    vec4 p30 = gl_in[3].gl_Position;
    vec4 p01 = gl_in[4].gl_Position;
    vec4 p11 = gl_in[5].gl_Position;
    vec4 p21 = gl_in[6].gl_Position;
    vec4 p31 = gl_in[7].gl_Position;
    vec4 p02 = gl_in[8].gl_Position;
    vec4 p12 = gl_in[9].gl_Position;
    vec4 p22 = gl_in[10].gl_Position;
    vec4 p03 = gl_in[12].gl_Position;
    vec4 p13 = gl_in[13].gl_Position;
    vec4 p23 = gl_in[14].gl_Position;
    vec4 p33 = gl_in[15].gl_Position;
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    teNormal = ...;
}
```
In the TES Shader – Computing the Position, given a u and v

// the basis functions:
float bu0 = (1.-u) * (1.-u) * (1.-u);
float bu1 = 3. * u * (1.-u) * (1.-u);
float bu2 = 3. * u * u * (1.-u);
float bu3 = u * u * u;

float dbu0 = -3. * (1.-u) * (1.-u);
float dbu1 =  3. * (1.-u) * (1.-3.*u);
float dbu2 =  3. * u *      (2.-3.*u);
float dbu3 =  3. * u *      u;

float bv0 = (1.-v) * (1.-v) * (1.-v);
float bv1 = 3. * v * (1.-v) * (1.-v);
float bv2 = 3. * v * v * (1.-v);
float bv3 = v * v * v;

float dbv0 = -3. * (1.-v) * (1.-v);
float dbv1 =  3. * (1.-v) * (1.-3.*v);
float dbv2 =  3. * v *      (2.-3.*v);
float dbv3 =  3. * v *      v;

// finally, we get to compute something:
vec4 dpdu = dbu0 *  ( bv0*p00 + bv1*p01 + bv2*p02 + bv3*p03 )
+ dbu1 * ( bv0*p10 + bv1*p11 + bv2*p12 + bv3*p13 )
+ dbu2 * ( bv0*p20 + bv1*p21 + bv2*p22 + bv3*p23 )
+ dbu3 * ( bv0*p30 + bv1*p31 + bv2*p32 + bv3*p33 );

vec4 dpdv = bu0 *  ( dbv0*p00 + dbv1*p01 + dbv2*p02 + dbv3*p03 )
+ bu1 * ( dbv0*p10 + dbv1*p11 + dbv2*p12 + dbv3*p13 )
+ bu2 * ( dbv0*p20 + dbv1*p21 + dbv2*p22 + dbv3*p23 )
+ bu3 * ( dbv0*p30 + dbv1*p31 + dbv2*p32 + dbv3*p33 );

tNormal = normalize( cross( dpdu.xyz, dpdv.xyz ) );

Tangent Vectors

Example: A Bézier Surface

Example: Whole-Sphere Subdivision

Using the x, y, z, and w to specify the center and radius of the sphere
Example: Whole-Sphere Subdivision

Making the Whole-Sphere Subdivision Adapt to Screen Coverage
Making the Whole-Sphere Subdivision Adapt to Screen Coverage

Example: PN Triangles

Example: PN Triangles

Example: PN Triangles

Example: PN Triangles
Example: PN Triangles

```
#version 400 compatibility
in float gLightIntensity;
const vec3 COLOR = vec3( 1., 1., 0. );
void main( )
{
    gl_FragColor = vec4( gLightIntensity*COLOR, 1. );
}
```

The Difference Between Tessellation Shaders and Geometry Shaders

By now, you are probably confused about when to use a Geometry Shader and when to use a Tessellation Shader. Both are capable of creating new geometry from existing geometry. See if this helps.

**Use a Geometry Shader when:**
1. You need to convert an input topology into a different output topology, such as in the silhouette and hedgehog shaders (triangles→lines) or the explosion shader (triangles→points).
2. You need some sort of geometry processing to come after the Tessellation Shader (such as how the shrink shader was used).

**Use a Tessellation Shader when:**
1. One of the built-in tessellation patterns will suit your needs.
2. You need more than 6 input vertices to define the surface being tessellated.
3. You need more output vertices than a Geometry Shader can provide.

Demonstrating the Limits of Tessellation Shaders

This tessellation is using 64x64 (the maximum allowed).

This is pretty good-looking, but doesn’t come close to using the full 4096x2276 resolution available for the bump-map.

The Cow’s Tail is a Good Example of using PN Triangles

Notice how much improvement there is just by increasing the outer tessellation. This is because smooth shading already helps the inner parts of triangles, but does nothing for the silhouettes.