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 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 patterns!

What built-in patterns can the Tessellation shaders produce?

- Lines
- Quads (subsequently broken into triangles)
- Triangles

The Tessellation Shaders: Where Do they Fit in the Pipeline?

- Fixed Function
- Programmable

In the OpenGL Program

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

// If you have a TCS, you must also have a Vertex Shader

// These have no implied topology – they will be given to you in an array. It's up to your shader to interpret the order
GLuint tcs = glCreateShader(GL_TESS_CONTROL_SHADER);
GLuint tes = glCreateShader(GL_TESS_EVALUATION_SHADER);
```

In GLSL:

```glsl
if (GL_ARB_tessellation_shader) {
  // If you have a TCS, you must also have a Vertex Shader
  // These have no implied topology – they will be given to you in an array. It's up to your shader to interpret the order
  GLuint tcs = glCreateShader(GL_TESS_CONTROL_SHADER);
  GLuint tes = glCreateShader(GL_TESS_EVALUATION_SHADER);
}```

Check the OpenGL extension:

- "GL_ARB_tessellation_shader"
Computer Graphics

TCS Inputs

| gl_in | is an array of structures:
|-------|---------------------|
| struct| {
|       | vec4 gl_Position;
|       | float gl_PointSize;
|       | float gl_ClipDistance[6];
|       | } gl_in[];
| gl_InInvocationID | 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)

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;

TCS Outputs

| gl_out | is an array of structures:
|-------|---------------------|
| struct| {
|       | vec4 gl_Position;
|       | float gl_PointSize;
|       | float gl_ClipDistance[6];
|       | } gl_out[];

All invocations of the TCS have read-only access to all the output information.

layout( vertices = n ) out; | Used to specify the number of vertices output to the TPG

RGG Inputs

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

Tessellation Primitive Pattern Generator (TPG)

- The TPG is “fixed-function”, i.e., you can’t change its operation except by setting parameters
- Consumes all vertices from the TCS and emits vertices for the triangles, quads, or isolines patterns
- TPG outputs a series of vertices as coordinates in barycentric (u,v,w) parametric space
- All three coordinates (u,v,w) are used for triangles
- Just (u,v) are used for quads and isolines

TES Inputs

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

teslaout(path) in;

TES Output Topologies: the Quad Pattern

| gl_TessLevelOuter[4] | is an array containing up to 4 edges of tessellation levels
| gl_TessLevelInner[2] | is an array containing up to 2 edges of tessellation levels
Examples

In these examples:

1. We are 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

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

   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

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

3. The Tessellation Evaluation Shader computes the x,y,z coordinates based on the TPS's \( u \) values

\[ P(u) = u^3(-P_0 + 3P_1 - 3P_2 + P_3) + u^2(3P_0 - 6P_1 + 3P_2) + u(-3P_0 + 3P_1 + P_2) + P_3 \]

In an OpenGL Program

```glsl
glPatchParameteri(GL_PATCH_VERTICES, 4);
glBegin(GL_PATCHES);
glVertex3f(x0, y0, z0);
glVertex3f(x1, y1, z1);
glVertex3f(x2, y2, z2);
glVertex3f(x3, y3, z3);
glEnd();
```

In a .glib File

```glsl
#Open GL GLIB
Perspective 70
Vertex beziercurve.vert
Fragment beziercurve.frag
TessControl beziercurve.tcs
TessEvaluation beziercurve.tes
Program BezierCurve uOuter0 <0 1 5>
  uOuter1 <3 5 50>
Color 1.  .5  0. 1.
NumPatchVertices 4
gBegin gl_patches
gVertex 0. 0. 0.
gVertex 1. 1. 1.
gVertex 2. 1. 0.
gVertex 3. 0. 1.
gEnd
```

In the TCS Shader

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

In the TES 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;
  // the basis functions:
  float b0 = (1.-u) * (1.-u) * (1.-u);
  float b1 = 3. * u * (1.-u) * (1.-u);
  float b2 = 3. * u * u * (1.-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

Outerr1 = 5

Outerr1 = 50
Example: A Bézier Surface

P00  P10  P20  P30
P31  P32  P33
P12  P22  P23
P13  P03  P02
P01  P11  P21

Bézier Surface Parametric Equations

\[ P(u,v) = \begin{bmatrix} (1-u)^3 & 3u(1-u)^2 & 3u^2(1-u) & u^3 \end{bmatrix} \begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \]

In an OpenGL Program

```glsl
glPatchParameteri(GL_PATCH_VERTICES, 16);
```

```glsl
glBegin(GL_PATCHES);
glVertex3f(x00, y00, z00);
glVertex3f(x10, y10, z10);
glVertex3f(x20, y20, z20);
glVertex3f(x30, y30, z30);
glVertex3f(x01, y01, z01);
glVertex3f(x11, y11, z11);
glVertex3f(x21, y21, z21);
glVertex3f(x31, y31, z31);
glVertex3f(x02, y02, z02);
glVertex3f(x12, y12, z12);
glVertex3f(x22, y22, z22);
glVertex3f(x32, y32, z32);
glVertex3f(x03, y03, z03);
glVertex3f(x13, y13, z13);
glVertex3f(x23, y23, z23);
glVertex3f(x33, y33, z33);
glEnd();
```

This order is not set by OpenGL. It is set by you. Pick a convention yourself and stick to it! GLSL doesn't care as long as you are consistent.

In the _glib File

```glsl
##OpenGL GLIB
Perspective 70
Vertex beziersurface.vert
Fragment beziersurface.frag
TessControl beziersurface.tcs
TessEvaluation beziersurface.tes
Geometry beziersurface.geom
Program BezierSurface uOuter02 <1 10 50>  uOuter13 <1 10 50>  uInner0 <1 10 50>  uInner1 <1 10 50>  
 uShrink <0. 1. 1.>                                                                                             
 u LightX <-10. 0. 10.>  u LightY <-10. 10. 10.>   uLightZ <-10. 10. 10. >
Color 1. 1. 0. 1.
NumPatchVertices 16
```

In the TCS Shader

```glsl
#version 400
#extension GL_ARB_tessellation_shader : enable
uniform float uOuter02, uOuter13, uInner0, uInner1;
layout( vertices = 16 )  out;
void
main( )
{
    gl_out[ gl_InvocationID ].gl_Position = gl_in[ gl_InvocationID ].gl_Position;
    gl_TessLevelOuter[0] = gl_TessLevelOuter[2] = uOuter02;
    gl_TessLevelInner[0]  = uInner0;
    gl_TessLevelInner[1]  = uInner1;
}
```

In the TES Shader

```glsl
#version 400 compatibility
#extension GL_ARB_tessellation_shader : enable
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;
    Assigning the intermediate pij's is here to make the code more readable. From what I've seen, the compiler will optimize this away.
In the TES Shader – Computing the Position, given a u and v

```plaintext
// 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:
gl_Position = bu0 * ( bv0*p00 + bv1*p01 + bv2*p02 + bv3*p03 )
+ bu1 * ( bv0*p10 + bv1*p11 + bv2*p12 + bv3*p13 )
+ bu2 * ( bv0*p20 + bv1*p21 + bv2*p22 + bv3*p23 )
+ bu3 * ( bv0*p30 + bv1*p31 + bv2*p32 + bv3*p33 );
```

Example: A Bézier Surface

```plaintext
Example: Whole-Sphere Subdivision
```

Example: Whole-Sphere Subdivision

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

spheresubd.tcs

Example: Whole-Sphere Subdivision

spheresubd.tes

Making the Whole-Sphere Subdivision Adapt to Screen Coverage

sphereadapt.tcs

Making the Whole-Sphere Subdivision Adapt to Screen Coverage

sphereadapt.tes
Making the Whole-Sphere Subdivision Adapt to Screen Coverage

Example: PN Triangles

Example: PN Triangles

Example: PN Triangles

Example: PN Triangles

Example: PN Triangles

Example: PN Triangles

Example: PN Triangles
Example: PN Triangles

```
#pragma version 400 compatibility

in float gLightIntensity;
const vec3 COLOR = vec3(1., 1., 0.);

void main()
{
    gl_FragColor = vec4(gLightIntensity * COLOR, 1.);
}
```

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

```
uOuter = 1, uInner = 1
```

```
uOuter = 2, uInner = 1
```

```
uOuter = 2, uInner = 2
```

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 8x8 (the maximum allowed).
This is pretty good-looking, but doesn't come close to using the full 4096x2048 resolution available for the bump-map.