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
- Triangles
- Quads (subsequently broken into 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 );
```
followed by some number of `glVertex3f( )` calls. There is no implied function, number of vertices, or vertex ordering – those are given by you.

### In the OpenGL Program

```
glBegin( GL_PATCHES );
glVertex3f( ... );
glVertex3f( ... );
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.

```
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

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

**Check the OpenGL extension:**

```
"GL_ARB_tessellation_shader"
```

In GLSL:
```
#version 400
#extension GL_ARB_tessellation_shader : enable
```

### TCS Inputs

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

  `gl_INVOCATIONID` 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

- `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

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

```
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
```
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 = 16 ) out;

Tessellation Primitive Generator

Is "fixed-function", i.e., you can't change its operation except by setting parameters

Consumes all vertices from the TCS and emits tessellated triangles, quads, or lines

Outputs positions as coordinates in barycentric (u,v,w)

All three coordinates (u,v,w) are used for triangles

Just (u,v) are used for quads and isolines

TES Inputs

Reads one vertex of 0 <= (u,v,w) <= 1 coordinates 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[ ];

Tessellation Primitive Generator (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 Output Topologies: the Quad Pattern

OL0 == 1 implies that you just want to draw a single curve.

TES Output Topologies: the Isolines Pattern

OL0 == 1 implies that you just want to draw a single curve.

TES Output Topologies: the Triangle Pattern

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

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

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.

The OpenGL tessellation can also do 1D curves. Just set OLS = 1.

Example: A Bézier Curve

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.

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

```c
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 an `.glib` File

```
#OpenGL 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. 0.5 0.1.
NumPatchVertices 4
gBegin gl_patches
gVertex 0. 0. 0.glVertex 1. 1. 1.glVertex 2. 1. 0.glVertex 3. 0. 1.
gEnd
```

In the TCS Shader

```
#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 TES Shader

```
#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. We assume that the compiler will optimize this away.
Example: A Bézier Curve

Outer1 = 5

Outer1 = 50

Example: A Bézier Surface

Example: A Bézier Surface

Bézier Surface Parametric Equations

In an OpenGL Program

\[
P(u, v) = \begin{bmatrix} (1-u)^3 & 3u(1-u)^2 & 3u^2(1-u) & u^3 \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} \begin{bmatrix} (1-v)^3 \\ 3v(1-v)^2 \\ 3v^2(1-v) \\ v^3 \end{bmatrix}
\]

In an OpenGL Program

\[
gPatchParameteri( GL_PATCH_VERTICES, 16 );
gBegin( GL_PATCHES );
gVertex3f( x00, y00, z00 );
gVertex3f( x10, y10, z10 );
gVertex3f( x20, y20, z20 );
gVertex3f( x30, y30, z30 );
gVertex3f( x01, y01, z01 );
gVertex3f( x11, y11, z11 );
gVertex3f( x21, y21, z21 );
gVertex3f( x31, y31, z31 );
gVertex3f( x02, y02, z02 );
gVertex3f( x12, y12, z12 );
gVertex3f( x22, y22, z22 );
gVertex3f( x32, y32, z32 );
gVertex3f( x03, y03, z03 );
gVertex3f( x13, y13, z13 );
gVertex3f( x23, y23, z23 );
gVertex3f( x33, y33, z33 );
gEnd();
\]

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
#version 400
precision highp float;
layout( tokens = 16 ) out;

float
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 TCS Shader

```glsl
#version 400
precision highp float;
layout( tokens = 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
precision highp float;
layout( tokens = 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;
}
```

Assigning the intermediate pj's is here to make the code more readable. We assume that the compiler will optimize this away.
In the TES Shader – Computing the Normal, given a u and v

\[
\text{vec4 dpdu} = \text{dbu0} \cdot (\text{bv0*p00} + \text{bv1*p01} + \text{bv2*p02} + \text{bv3*p03}) + \text{dbu1} \cdot (\text{bv0*p10} + \text{bv1*p11} + \text{bv2*p12} + \text{bv3*p13}) + \text{dbu2} \cdot (\text{bv0*p20} + \text{bv1*p21} + \text{bv2*p22} + \text{bv3*p23}) + \text{dbu3} \cdot (\text{bv0*p30} + \text{bv1*p31} + \text{bv2*p32} + \text{bv3*p33});
\]

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

\[
\text{teNormal} = \text{normalize(cross(dpdu.xyz, dpdv.xyz))};
\]
**Example: Whole-Sphere Subdivision**

```glsl
#version 400 compatibility

out vec3 vCenter;
out float vRadius;

void main()
{
    vCenter = gl_Vertex.xyz;
    vRadius = gl_Vertex.w;

    gl_Position = vec4( 0., 0., 0., 1. ); // doesn't matter now – we will in the cords later
}
```

**spheresubd.vert**

Using the x, y, z, and w to specify the center and radius of the sphere

**Example: Whole-Sphere Subdivision**

```glsl
#version 400 compatibility

#extension GL_ARB_tessellation_shader : enable

in float vRadius[];
in vec3 vCenter[];

patch out float tcRadius;
patch out vec3 tcCenter;

uniform float uDetail;
uniform float uScale;

layout( vertices = 1 ) out;

void main()
{
    gl_out[ gl_InvocationID ].gl_Position = gl_in[ 0 ].gl_Position; // (0,0,0,1)
    tcCenter = vCenter[ 0 ];
    tcRadius = vRadius[ 0 ];

    gl_TessLevelOuter[0] = 2.;
    gl_TessLevelOuter[1] = uScale * tcRadius * uDetail;
    gl_TessLevelOuter[2] = 2.;
    gl_TessLevelOuter[3] = uScale * tcRadius * uDetail;

    gl_TessLevelInner[0] = uScale * tcRadius * uDetail;
    gl_TessLevelInner[1] = uScale * tcRadius * uDetail;
}
```

**spheresubd.tcs**

Using the scale and the radius to help set the tessellation detail

Outer[0] and Outer[2] are the number of divisions at the poles.
Outer[1] and Outer[3] are the number of divisions at the vertical seams. Inner[0] and Inner[1] are the inside sphere detail.

**Example: Whole-Sphere Subdivision**

```glsl
#version 400 compatibility

#extension GL_ARB_tessellation_shader : enable

uniform float uScale; layout( quads, equal_spacing, ccw ) in;

patch in float tcRadius;
patch in vec3 tcCenter;

out vec3 teNormal;
const float PI = 3.14159265;

void main()
{
    vec3 p = gl_in[0].gl_Position.xyz;
    float u  = gl_TessCoord.x;
    float v  = gl_TessCoord.y;
    float w = gl_TessCoord.z;

    float phi = PI * ( u - .5 );
    float theta = 2. * PI * ( v - .5 );
    float cosphi = cos(phi);
    vec3 xyz = vec3( cosphi*cos(theta), sin(phi), cosphi*sin(theta) );

    teNormal = xyz;
    xyz *= ( uScale * tcRadius );
    xyz += tcCenter;

    gl_Position = gl_ModelViewMatrix * vec4( xyz,1. );
}
```

**spheresubd.tes**

Turning u and v into spherical coordinates

```
// π ≤ φ ≤ +π/2
// -π ≤ θ ≤ +π

xyz = vec3( uScale * tcRadius );
xyz += tcCenter;
```
### Making the Whole-Sphere Subdivision Adapt to Screen Coverage

**spheredapt.tcs, I**

Version 400 compatibility: WebGL extension `GL_ARB_tessellation_shader` enabled.

```glsl
in float vRadius[];
in vec3 vCenter[];
patch out float tcRadius;
patch out vec3 tcCenter;
uniform float uDetail;
layout( vertices = 1 ) out;
void main( )
{
    gl_out[ gl_InvocationID ].gl_Position = gl_in[ 0 ].gl_Position; // (0,0,0,1)
    tcCenter = vCenter[0];
    tcRadius = vRadius[0];
    vec4 mx = vec4( vCenter[0] - vec3( vRadius[0], 0., 0. ), 1.);
    vec4 px = vec4( vCenter[0] + vec3( vRadius[0], 0., 0. ), 1.);
    vec4 my = vec4( vCenter[0] - vec3( 0., vRadius[0], 0. ), 1.);
    vec4 py = vec4( vCenter[0] + vec3( 0., vRadius[0], 0. ), 1.);
    vec4 mz = vec4( vCenter[0] - vec3( 0., 0., vRadius[0] ), 1.);
    vec4 pz = vec4( vCenter[0] + vec3( 0., 0., vRadius[0] ), 1.);
    mx = gl_ModelViewProjectionMatrix * mx;
    px = gl_ModelViewProjectionMatrix * px;
    my = gl_ModelViewProjectionMatrix * my;
    py = gl_ModelViewProjectionMatrix * py;
    mz = gl_ModelViewProjectionMatrix * mz;
    pz = gl_ModelViewProjectionMatrix * pz;
    mx.xy /= mx.w;
    px.xy /= px.w;
    my.xy /= my.w;
    py.xy /= py.w;
    mz.xy /= mz.w;
    pz.xy /= pz.w;
    float dx = distance( mx.xy, px.xy );
    float dy = distance( my.xy, py.xy );
    float dz = distance( mz.xy, pz.xy );
    float dmax = sqrt( dx*dx + dy*dy + dz*dz );
    gl_TessLevelOuter[0] = 2.;
    gl_TessLevelOuter[1] = dmax * uDetail;
    gl_TessLevelOuter[2] = 2.;
    gl_TessLevelOuter[3] = dmax * uDetail;
    gl_TessLevelInner[0]  = dmax * uDetail;
    gl_TessLevelInner[1]  = dmax * uDetail;
}
```

We no longer use `uScale` or `tcRadius`. But, we do use `uDetail` to provide a way to convert from NDC to Screen Space or to indicate the quality you'd like (i.e., `uDetail` depends on how good you want the spheres to look and on how large the window is in pixels.)

**spheredapt.tcs, II**

```glsl
const float PI = 3.14159265;

void main( )
{
    vec3 p = gl_in[0].gl_Position.xyz;
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    float w = gl_TessCoord.z;
    float phi = PI * ( u - .5 );
    float theta = 2. * PI * ( v - .5 );
    float cosphi = cos(phi);
    vec3 xyz = vec3( cosphi*cos(theta), sin(phi), cosphi*sin(theta) );
    xyz *= ncRadius;
    xyz += tcCenter;
    gl_Position = gl_ModelViewMatrix * vec4( xyz, 1. );
}
```

No longer uses `uScale`

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

**spheredapt.tes**

Original

Triangles Shrunk

Rotated

Zoomed In

Notice that the number of triangles adapts to the screen coverage of each sphere, and that the size of the tessellated triangles stays about the same, regardless of radius or transformation.
General idea: turn each triangle into a triangular Bézier patch. Create the Bézier control points by using the surface normals at the corner vertices. The Bézier patch equation can then be interpolated to any level of tessellation.

Example: PN Triangles

Example: PN Triangles
Example: PN Triangles

```
#version 400 compatibility
#extension GL_gpu_shader4: enable
#extension GL_geometry_shader4: enable

uniform float  uShrink;
in vec3              teNormal[ ];
out float          gLightIntensity;

const vec3 LIGHTPOS = vec3( 5., 10., 10. );
vec3 V[3];
vec3 CG;

void
ProduceVertex( int v )
{
    gLightIntensity = abs( dot( normalize(LIGHTPOS - V[v]), normalize(teNormal[v]) ) );
    gl_Position = gl_ProjectionMatrix * vec4( CG + uShrink * ( V[v] - CG ), 1. );
    EmitVertex( );
}

void
main( )
{
    V[0]  =   gl_PositionIn[0].xyz;
    V[1]  =   gl_PositionIn[1].xyz;
    ProduceVertex( 0 );
    ProduceVertex( 1 );
    ProduceVertex( 2 );
}
```

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 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.
```

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.