Tessellation Shaders

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)

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

Transformed xy Patch Vertices from the Vertex Shader

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 );
glEnd();
```

In the OpenGL Program
```
gBegin( GL_PATCHES );
gVertex3f( ... );
gEnd();
```

Check the OpenGL extension:
```
"GL_ARB_tessellation_shader"
```

In GLSL:
```
#extension GL_ARB_tessellation_shader : enable
```
gl_in[] is an array of structures:

```c
struct {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[6];
} gl_in[];
```

- `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).

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

```c
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

gl_out[] is an array of structures:

```c
struct {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[6];
} gl_out[];
```

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:

```c
layout( vertices = 16 ) out;
```

In the TCS:

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

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 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.

Tessellation Evaluation Shader (TES):

- 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".

```
struct {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[6];
} gl_in[];
```

```
layout(                     ,                                              ,               , point_mode ) in;
```

- `triangles`:
- `quads`:
- `isolines`:
- `equal_spacing`:
- `fractional_even_spacing`:
- `fractional_odd_spacing`:

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.

```
layout( vertices = n ) out;
```

Used to specify the number of vertices output to the TPG.
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 OL0 = 1.
Example: A Bézier Curve

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

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

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

In an OpenGL Program

```opengl
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

```glib
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.  .5  0. 1.
NumPatchVertices 4

In the TCS Shader
```

```c
version 4.0
extension GL_ARB_tessellation_shader: enable

void main() {
    gl_TessLevelOuter[0] = float( uOuter0 );
    gl_TessLevelOuter[1] = float( uOuter1 );
}
```

```c
In the TES Shader
```

```c
version 4.0
extension GL_ARB_tessellation_shader: enable

layout( isolines, equal_spacing ) in;

void main() {
    // 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 p[i]'s here to make the code more readable. We assume that the compiler will optimize this away.
Example: A Bézier Curve

Outer1 = 5

Example: A Bézier Surface

Bézier Surface Parametric Equations

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

In an OpenGL Program

```cpp
glPatchParameteri(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

```
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

```cpp
#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

Assigning the intermediate pij's is here to make the code more readable. We assume that the compiler will optimize this away.

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

Computing the Normal, given a u and v

Example: A Bezier Surface

Example: Whole-Sphere Subdivision
Example: Whole-Sphere Subdivision

spheresubd.vert

version 400 compatibility
out vec3 vNormal;
out float vRadius;
void main()
{
    vCenter = gl_Vertex.xyz;
    vRadius = gl_Vertex.w;
    vec3 xyz = vec3( cosphi*cos(theta), sin(phi), cosphi*sin(theta) );
    float cosphi = cos(phi);
    float theta = 2. * PI * ( v - .5 );
    float phi = PI * ( u - .5 );
    float w = gl_TessCoord.z;
    float v  = gl_TessCoord.y;
    float u  = gl_TessCoord.x;
    vec3 p = gl_in[0].gl_Position.xyz;
    gl_out[ gl_InvocationID ].gl_Position = gl_in[ 0 ].gl_Position; // (0,0,0,1)
    vRadius = gl_Vertex.w;
    vCenter = gl_Vertex.xyz;
    vec4 mz = vec4( vCenter[0] - vec3( 0., 0., vRadius[0] ), 1. );
    vec4 my = vec4( vCenter[0] - vec3( 0., vRadius[0], 0. ), 1. );
    vec4 pz = vec4( vCenter[0] + vec3( 0., 0., vRadius[0] ), 1. );
    vec4 py = vec4( vCenter[0] + vec3( 0., vRadius[0], 0. ), 1. );
    vec4 px = vec4( vCenter[0] + vec3( vRadius[0], 0., 0. ), 1. );
    vec4 mx = vec4( vCenter[0] - vec3( vRadius[0], 0., 0. ), 1. );
    tcRadius = vRadius[0];
    tcCenter = vCenter[0];
}

Example: Whole-Sphere Subdivision

spheresubd.tcs

version 400 compatibility
sphereadapt.tcs

void main()
layout( vertices = 1 ) out;
uniform float uScale;
uniform float uDetail;
patch out vec3 tcCenter;
patch out float tcRadius;
in vec3 vCenter[];
in float vRadius[];
#extension GL_ARB_tessellation_shader : enable
#version 400 compatibility
spheresubd.tcs

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Making the Whole-Sphere Subdivision Adapt to Screen Coverage

spheresubd.tcs

version 400 compatibility
using GL_ARB_tessellation_shader enable
in float vScale[];
in vec3 vCenter[];
patch out float tcRadius;
patch out vec3 tcCenter;
layout( quads, equal_spacing, ccw) in;
uniform float uScale;
#extension GL_ARB_tessellation_shader : enable
#version 400 compatibility
spheresubd.tcs

Making the Whole-Sphere Subdivision Adapt to Screen Coverage

spheresubd.tcs

version 400 compatibility
using GL_ARB_tessellation_shader enable
in float vScale[];
in vec3 vCenter[];
patch out float tcRadius;
patch out vec3 tcCenter;
layout( vertices = 1 ) out;
uniform float uScale;
#extension GL_ARB_tessellation_shader : enable
#version 400 compatibility
spheresubd.tcs

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision

Example: Whole-Sphere Subdivision
Create the Bézier control points by using the surface normals at the corner vertices. The Bézier patch equation can then be interpolated into a triangular Bézier patch.

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

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.
Example: PN Triangles

```glsl
#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 );
}
```

pntriangles.frag

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

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

Example: PN Triangles

```glsl
#version 400 compatibility
// 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

```
\begin{itemize}
  \item uOuter = 1, uInner = 1
  \item uOuter = 2, uInner = 1
  \item uOuter = 2, uInner = 2
\end{itemize}
```

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 hedgerow 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.
```