**Tessellation Shaders**

I once won an OpenGL t-shirt at the SIGGRAPH conference by knowing how to correctly spell “tessellation”. (It's 2 s's and 2 l's.)

The only reason I knew it was that the week before I had been experimenting with the newly-released tessellation shaders for the first time and my program wouldn't compile because I had misspelled tessellation. The week of the conference, it was still fresh in my mind how to spell it correctly.

Please don't tell anyone. Spelling correctly undermines one's CS credibility… 😊

---

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

---

**The Tessellation Shaders: Where Do They Fit in the Pipeline?**

- Tessellation Control Shader
- Tessellation Primitive Generator
- Tessellation Evaluation Shader
- Vertex Shader
- Primitive Assembly
- Geometry Shader
- Primitive Assembly
- Rasterizer
- Fragment Shader

If in use, it is always the first stop after the Vertex Shader
If in use, it is always the last stop before the Rasterizer
The **Tessellation Control Shader (TCS)** optionally can transform the input coordinates (but usually doesn’t). It also computes the required tessellation level based on distance to the eye, screen space spanning, hull curvature, or displacement roughness. There is one TCS execution per 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**. Nobody asked.)

The **Tessellation Evaluation Shader (TES)** turns the TPG’s u-v-w coordinates into x-y-z. It can apply displacements. There is one TES execution per generated vertex.

There is a new “Patch” primitive:

```gl
glBegin( GL_PATCHES )
```

followed by some number of `glVertex3f( )` calls. There is no implied purpose, number of vertices, or vertex ordering – those are given by you in how you write the shader.

**In the OpenGL Program**

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

Check the OpenGL extension:

“GL_ARB_tessellation_shader”

**In GLSL:**

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

---

### TCS Inputs

- `gl_in[ ]` is an array of structures:
  ```gl
  struct
  {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[ 6 ];
  } gl_in[ ];
  ```
- `gl InvocationID` tells you which vertex you are working on. This is 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).
gl_out[] is an array of structures:

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

layout( vertices = n ) out; Used to specify the number of vertices sent to the TPG.

**TCS Outputs**

- `gl_TessLevelOuter[4]` is a built-in array containing up to 4 outside edges of tessellation levels.
- `gl_TessLevelInner[2]` is a built-in array containing up to 2 inside 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 use:

```c
layout( vertices = 16 ) 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".

```c
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.
- 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.
**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.

**TES Output Topologies: the Isolines 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.

Top line not drawn

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

**TES Output Topologies: the Triangle 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.

**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 the surfaces have been tessellated.

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

The OpenGL tessellation can do 1D curves. Just set OL0 == 1.

2. The Tessellation Primitive Generator generates $u$ 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 and $P_0$, $P_1$, $P_2$, and $P_3$.

$$P(u) = (1-u)^3P_0 + 3u(1-u)^2P_1 + 3u^2(1-u)P_2 + u^3P_3$$

where $P$ is an abbreviation for $\{x\ y\ z\}$.
In an OpenGL Program

In InitGraphics( ):

Pattern.Init( );
Pattern.Create( "pattern.vert", "pattern.tcs", "pattern.tes", "pattern.frag" );
Pattern.SetUniformVariable( "uOuter0", 20 );
Pattern.SetUniformVariable( "uOuter1", 10 );

In Display( ):

glPatchParameter( 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

###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
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
uniform int uOuter0, uOuter1;
layout( vertices = 4 ) out;
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

Outer1 = 5

Outer1 = 50

Example: A Bézier Surface

Bézier Surface Parametric Equations

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

In an OpenGL Program

```glsl
// Set patch parameters
glPatchParameteri(GL_PATCH_VERTICES, 16);

// Begin GL_PATCHES
glBegin(GL_PATCHES);

// Set vertices
glVertex3f(x_{00}, y_{00}, z_{00});
glVertex3f(x_{10}, y_{10}, z_{10});
glVertex3f(x_{20}, y_{20}, z_{20});
glVertex3f(x_{30}, y_{30}, z_{30});
glVertex3f(x_{01}, y_{01}, z_{01});
glVertex3f(x_{11}, y_{11}, z_{11});
glVertex3f(x_{21}, y_{21}, z_{21});
glVertex3f(x_{31}, y_{31}, z_{31});
glVertex3f(x_{02}, y_{02}, z_{02});
glVertex3f(x_{12}, y_{12}, z_{12});
glVertex3f(x_{22}, y_{22}, z_{22});
glVertex3f(x_{32}, y_{32}, z_{32});
glVertex3f(x_{03}, y_{03}, z_{03});
glVertex3f(x_{13}, y_{13}, z_{13});
glVertex3f(x_{23}, y_{23}, z_{23});
glVertex3f(x_{33}, y_{33}, z_{33});
```

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
#extension GL_ARB_tessellation_shader : enable

layout( triangles, equal_spacing, ccw ) in;
out vec3 telnor;

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;
    float3 pos = vec3( gl_TessCoord.x * p00.z + gl_TessCoord.y * p01.z + gl_TessCoord.z * p02.z + gl_TessCoord.w * p03.z );
    gl_Position = pos;
}
```

In the TCS Shader

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

layout( vertexes = 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
// 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 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:
float bu0v0 = bu0 * bv0;
float bu1v0 = bu1 * bv0;
float bu2v0 = bu2 * bv0;
float bu3v0 = bu3 * bv0;
float bu0v1 = bu0 * bv1;
float bu1v1 = bu1 * bv1;
float bu2v1 = bu2 * bv1;
float bu3v1 = bu3 * bv1;
float bu0v2 = bu0 * bv2;
float bu1v2 = bu1 * bv2;
float bu2v2 = bu2 * bv2;
float bu3v2 = bu3 * bv2;
float bu0v3 = bu0 * bv3;
float bu1v3 = bu1 * bv3;
float bu2v3 = bu2 * bv3;
float bu3v3 = bu3 * bv3;
float bu0v4 = bu0 * bv4;
float bu1v4 = bu1 * bv4;
float bu2v4 = bu2 * bv4;
float bu3v4 = bu3 * bv4;
float bu0v5 = bu0 * bv5;
float bu1v5 = bu1 * bv5;
float bu2v5 = bu2 * bv5;
float bu3v5 = bu3 * bv5;
float bu0v6 = bu0 * bv6;
float bu1v6 = bu1 * bv6;
float bu2v6 = bu2 * bv6;
float bu3v6 = bu3 * bv6;
float bu0v7 = bu0 * bv7;
float bu1v7 = bu1 * bv7;
float bu2v7 = bu2 * bv7;
float bu3v7 = bu3 * bv7;
float bu0v8 = bu0 * bv8;
float bu1v8 = bu1 * bv8;
float bu2v8 = bu2 * bv8;
float bu3v8 = bu3 * bv8;
float bu0v9 = bu0 * bv9;
float bu1v9 = bu1 * bv9;
float bu2v9 = bu2 * bv9;
float bu3v9 = bu3 * bv9;
float bu0v10 = bu0 * bv10;
float bu1v10 = bu1 * bv10;
float bu2v10 = bu2 * bv10;
float bu3v10 = bu3 * bv10;
float bu0v11 = bu0 * bv11;
float bu1v11 = bu1 * bv11;
float bu2v11 = bu2 * bv11;
float bu3v11 = bu3 * bv11;
float bu0v12 = bu0 * bv12;
float bu1v12 = bu1 * bv12;
float bu2v12 = bu2 * bv12;
float bu3v12 = bu3 * bv12;
float bu0v13 = bu0 * bv13;
float bu1v13 = bu1 * bv13;
float bu2v13 = bu2 * bv13;
float bu3v13 = bu3 * bv13;
float bu0v14 = bu0 * bv14;
float bu1v14 = bu1 * bv14;
float bu2v14 = bu2 * bv14;
float bu3v14 = bu3 * bv14;
float bu0v15 = bu0 * bv15;
float bu1v15 = bu1 * bv15;
float bu2v15 = bu2 * bv15;
float bu3v15 = bu3 * bv15;
```

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 Normal, given a u and v

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

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

\[ \text{teNormal} = \text{normalize}( \text{cross( dpdu } . \text{xyz}, \text{dpdv } . \text{xyz}) ) ; \]

Tangent Vectors obtained by differentiating the position equation with respect to u and v.

Vector cross product to get the perpendicular normal to the two tangent vectors.

Tessellation Levels and Smooth Shading

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

**spheresubd.vert**

```cpp
#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 fill in the coords later
}
```

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

**spheresubd.tcs**

```cpp
#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;
```

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

**spheresubd.tes**

```cpp
//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 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. );
```

Turning u and v into spherical coordinates

### Details

- **Detail=50, Scale=2.5**
- **Detail=30, Scale=1.**
- **Detail=50, Scale=1.**

---

### Example: Whole-Sphere Subdivision

**Computer Graphics**

Example: Whole-Sphere Subdivision

**Computer Graphics**

Example: Whole-Sphere Subdivision

**Computer Graphics**

Example: Whole-Sphere Subdivision
Making the Whole-Sphere Subdivision Adapt to Screen Coverage, I

```c
#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;
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;
}
```

Extreme points of the sphere in Clip space

Extreme points of the sphere in NDC space

How long are the lines between the extreme points?

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.)

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

```c
//version 400 compatibility
#extension GL_ARB_tessellation_shader : enable
layout( quads, equal_spacing, ccw ) in;
patch in float tcRadius;
ipatch 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 *= tcRadius;
    xyz += tcCenter;
    gl_Position = gl_ModelViewMatrix * vec4( xyz, 1. );
}
```

Original

Triangles Shrunk

Zoomed In

Zoomed Out

Rotated

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.

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

Spherical coordinates

No longer uses uScale

Making the Whole-Sphere Subdivision Adapt to Screen Coverage, IV
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.

Observation: triangles are usually passed in with points (P) and normals (N). Using this method, those triangles can be broken into a series of smoother triangles internally. AMD actually had this in their firmware before tessellation shaders made it unnecessary.


Example: PN Triangles

Vec3xyz = gl_Vertex.xyz;
xyz *= uScale;
g_Position = gl_ModelViewMatrix * vec4( Vec3xyz, 1. );
VNormal = normalize( gl_NormalMatrix * gl_Normal );
}

Example: PN Triangles

Vec3xyz = gl_Vertex.xyz;
xyz *= uScale;
g_Position = gl_ModelViewMatrix * vec4( Vec3xyz, 1. );
VNormal = normalize( gl_NormalMatrix * gl_Normal );
}

Example: PN Triangles

Vec3xyz = gl_Vertex.xyz;
xyz *= uScale;
g_Position = gl_ModelViewMatrix * vec4( Vec3xyz, 1. );
VNormal = normalize( gl_NormalMatrix * gl_Normal );
}

Example: PN Triangles

Vec3xyz = gl_Vertex.xyz;
xyz *= uScale;
g_Position = gl_ModelViewMatrix * vec4( Vec3xyz, 1. );
VNormal = normalize( gl_NormalMatrix * gl_Normal );
}
Example: PN Triangles

```glsl
#version 400

uniform float uShrink;
in vec3 texNormal[];
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(texNormal[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;
    CG = (V[0] + V[1] + V[2]) / 3.;
    ProduceVertex(0);
    ProduceVertex(1);
    ProduceVertex(2);
}
```

### Example: PN Triangles

```glsl
#version 400

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.