

Tessellation Shaders

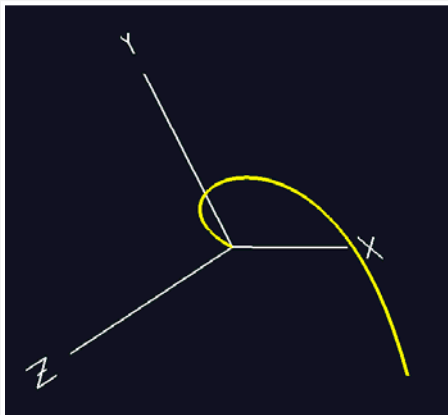
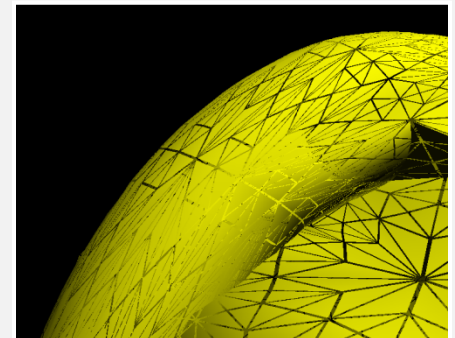


Oregon State
University
Mike Bailey

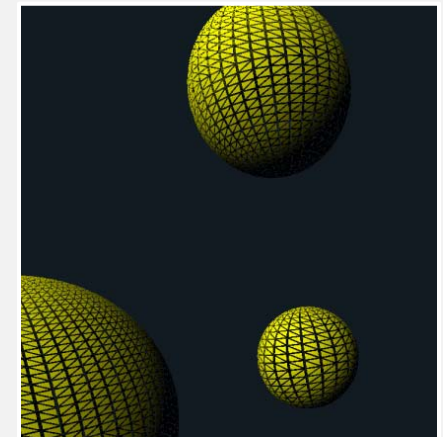
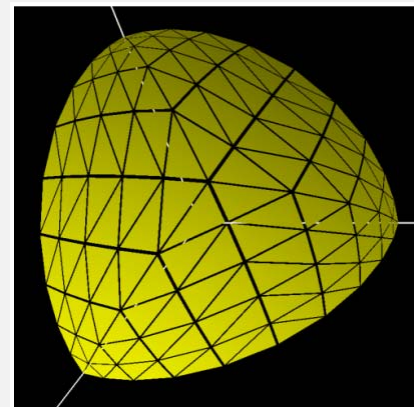
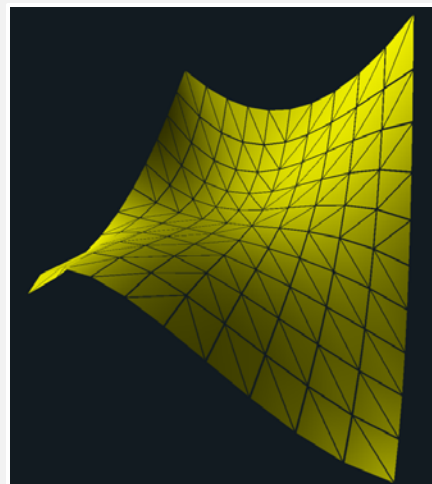
mjb@cs.oregonstate.edu



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/)



Computer Graphics



tessellation.pptx

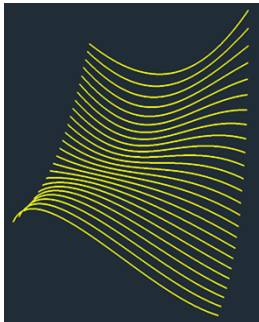
mjb – December 31, 2019

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

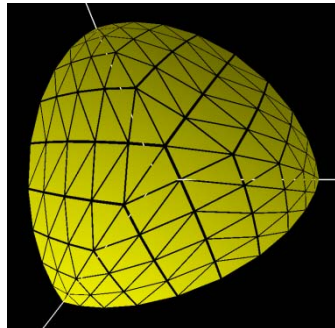
- You can perform adaptive subdivision based on a variety of criteria (size, curvature, etc.)
- You can provide coarser models, but have finer ones displayed (\approx geometric compression)
- You can apply detailed displacement maps without supplying equally detailed geometry
- You can adapt visual quality 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?

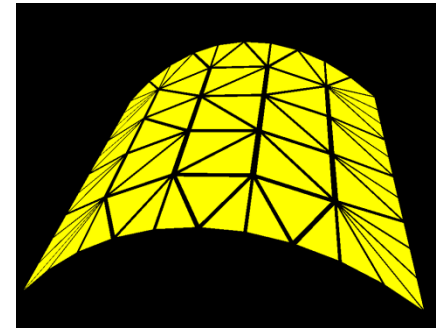
Lines



Triangles

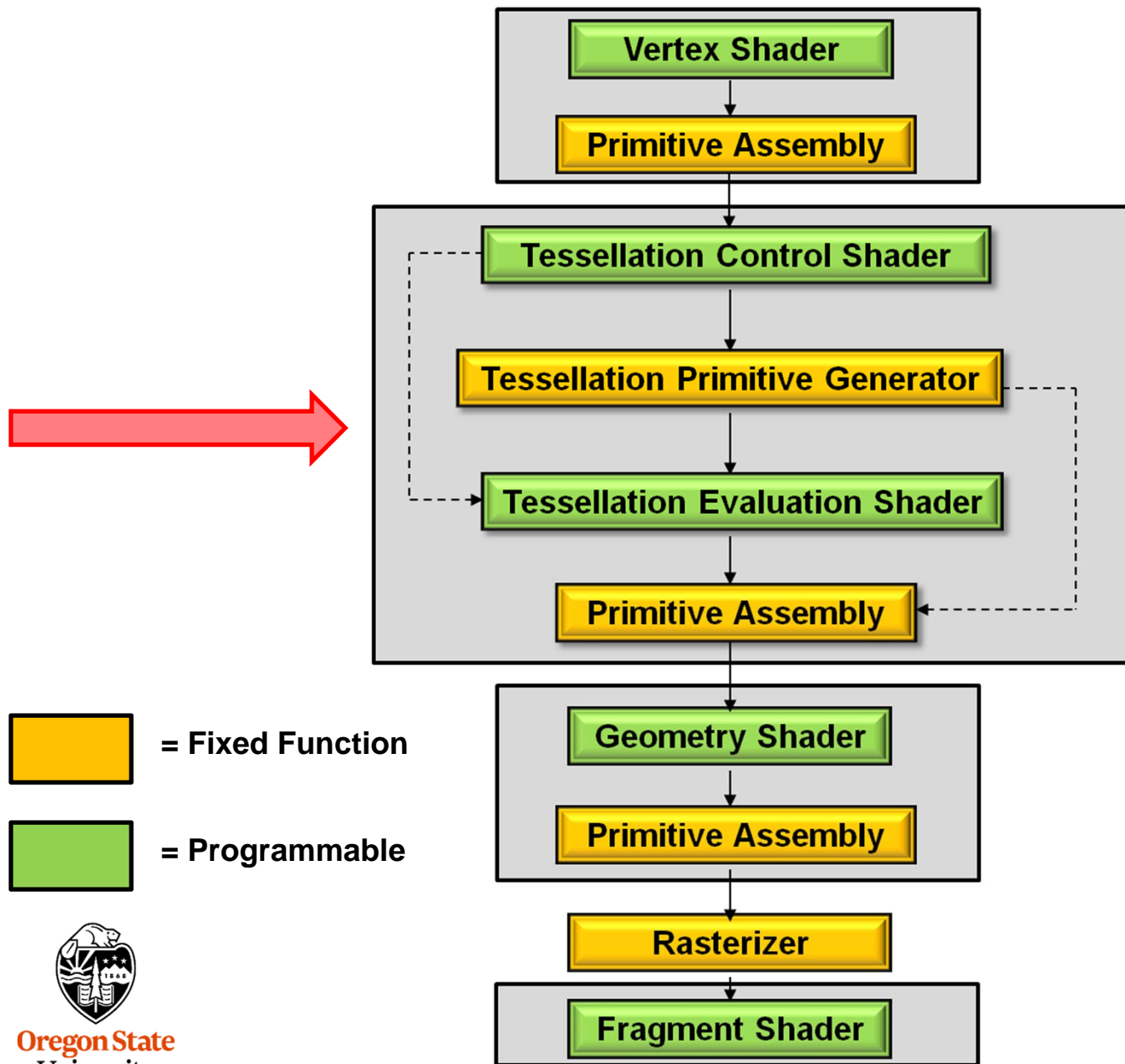


Quads (subsequently broken into triangles)



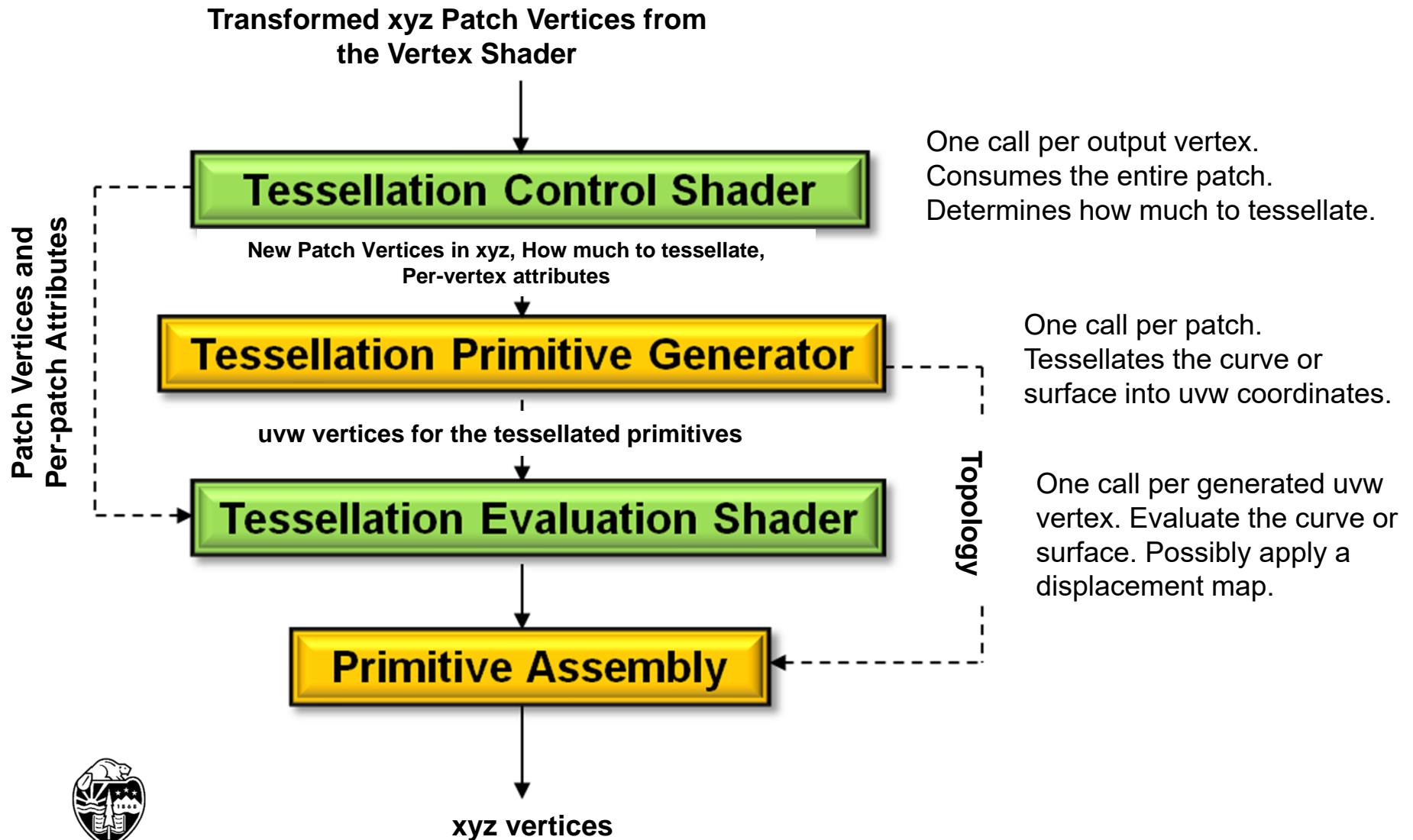
The Tessellation Shaders: Where Do they Fit in the Pipeline?

3



Tessellation Shader Organization

4



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 uvw 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 how you write the shader.



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

```
glPatchParameteri( GL_PATCH_VERTICES, num ); // #of vertices in each patch
```

```
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

Check the OpenGL extension:

“GL_ARB_tessellation_shader”

In GLSL:

```
#version 400
```

```
#extension GL_ARB_tessellation_shader : enable
```



`gl_in[]` is an array of structures:

```
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[]`

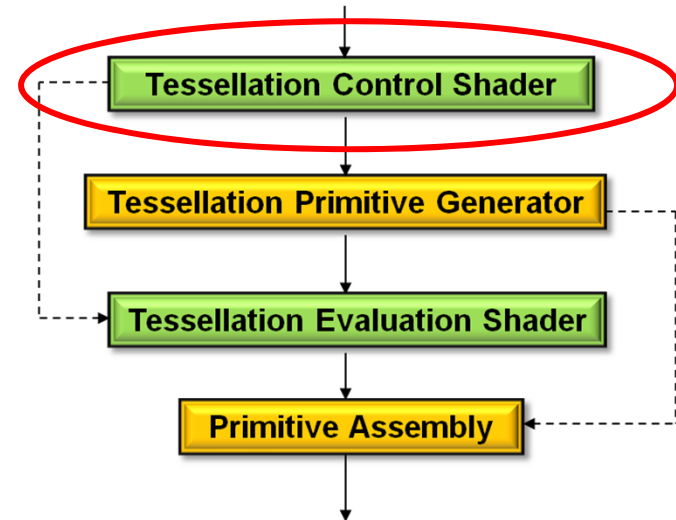
gl_PrimitiveID is the number of primitives since the last `glBegin()` (the first one is #0)

TCS Outputs

8

`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

`gl_TessLevelOuter[4]` is an array containing up to 4 edges of tessellation levels

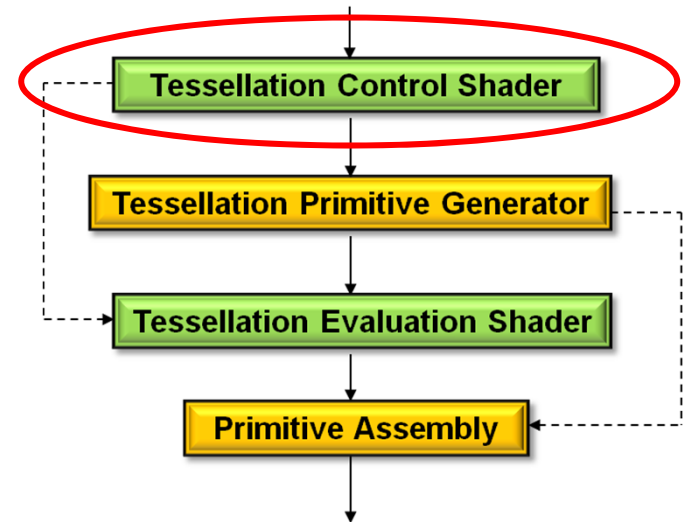
`gl_TessLevelInner[2]` is an array containing up to 2 edges of tessellation levels

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;
```



TES Inputs

10

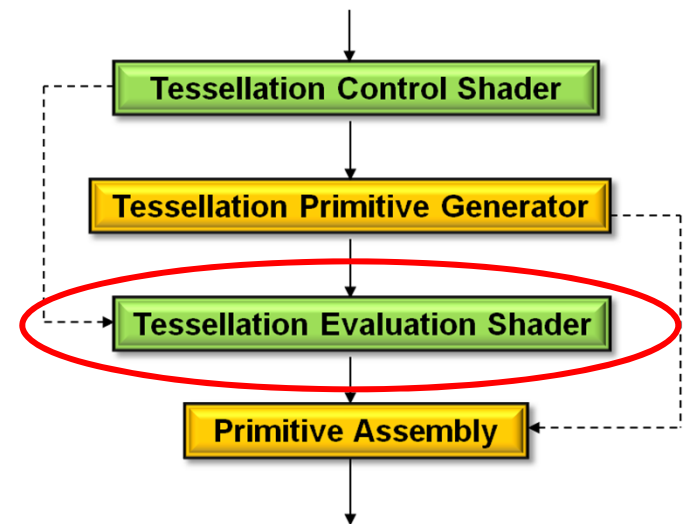
Reads one vertex of $0 \leq (u,v,w) \leq 1$ coordinates in variable *vec3 gl_TessCoord*

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[ ];
```



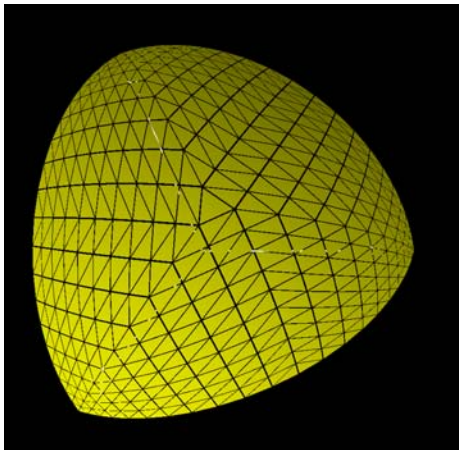
layout($\left\{ \begin{array}{l} \text{triangles} \\ \text{quads} \\ \text{isolines} \end{array} \right\}$, $\left\{ \begin{array}{l} \text{equal_spacing} \\ \text{fractional_even_spacing} \\ \text{fractional_odd_spacing} \end{array} \right\}$, $\left\{ \begin{array}{l} \text{ccw} \\ \text{cw} \end{array} \right\}$, point_mode) in;



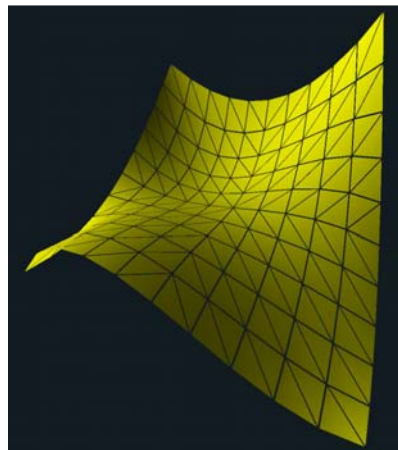
Tessellation Primitive Pattern Generator (TPG)

11

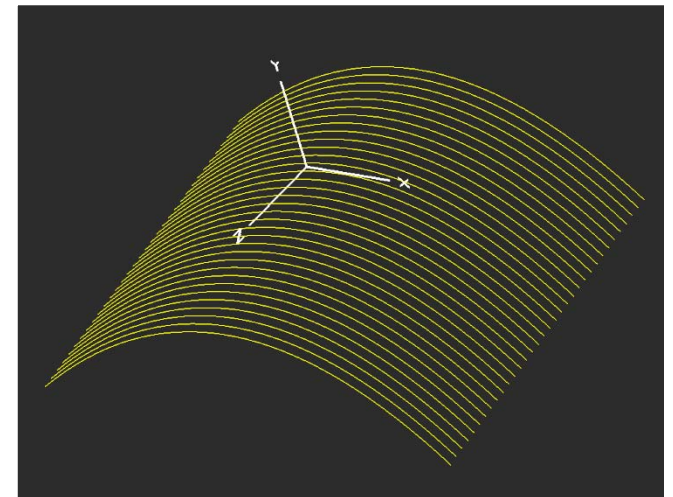
- The TPG is “fixed-function”, i.e., you can’t change its operation except by setting parameters
- Consumes all vertices from the TCS and emits vertices for the ***triangles***, ***quads***, or ***isolines*** patterns
- TPG outputs a series of vertices as coordinates in barycentric (u,v,w) parametric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



 Triangle
pattern
Oregon!
Univer
Computer Graphics



Quad
pattern

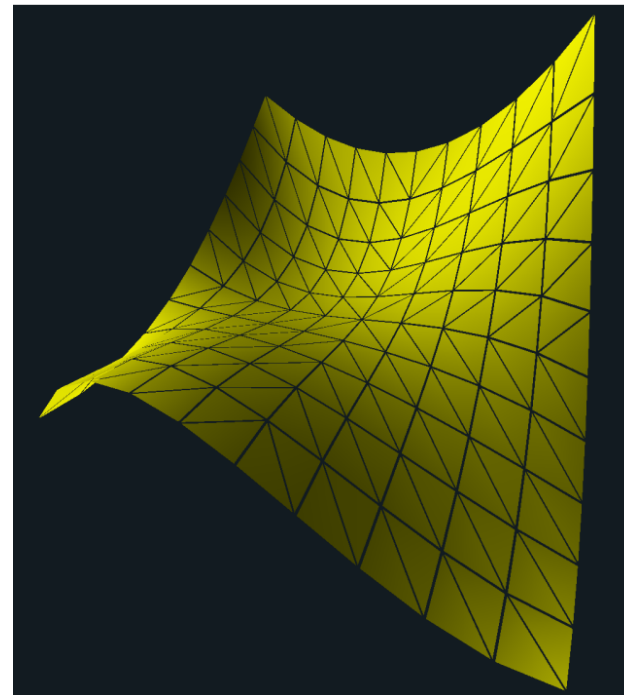
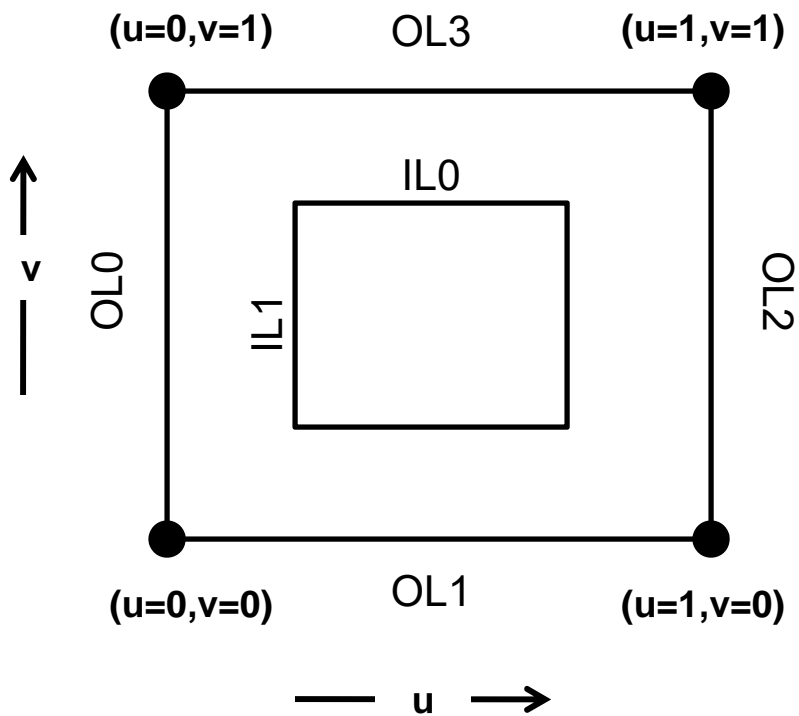


Isoline
pattern

TES Output Topologies: the Quad Pattern

12

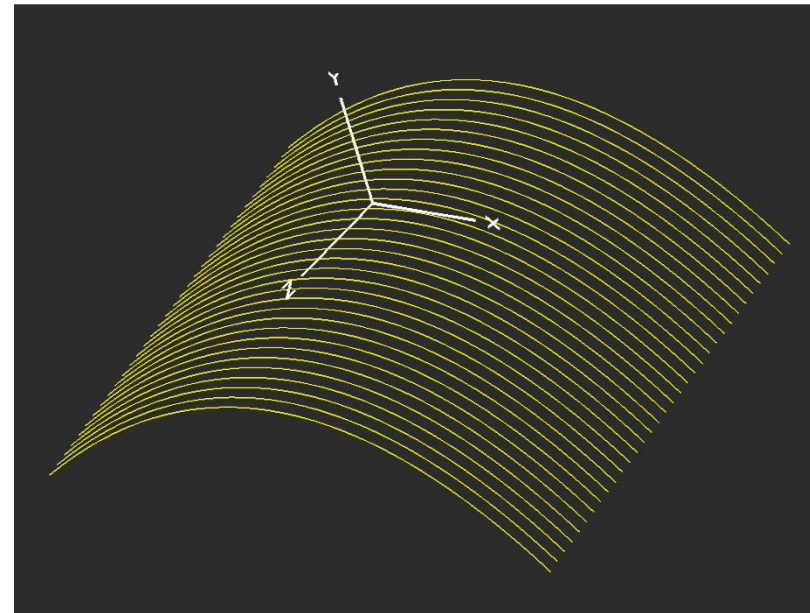
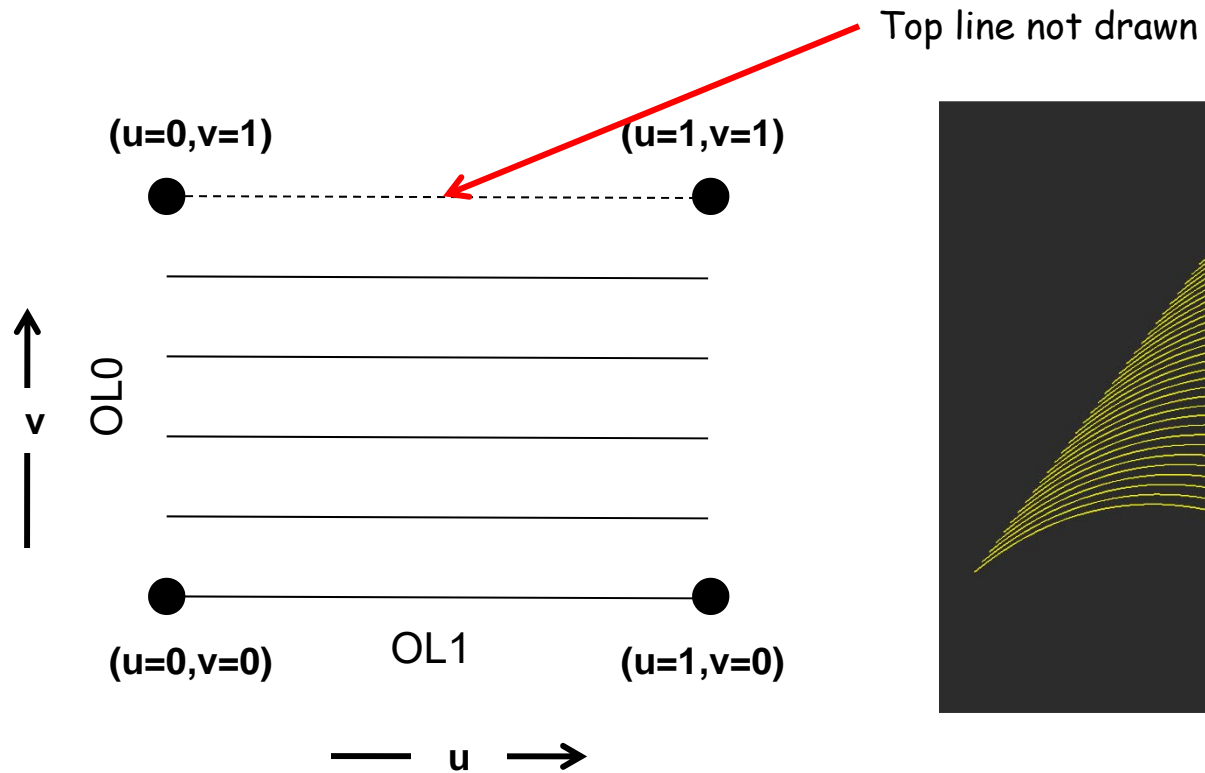
gl_TessLevelOuter[4] is an array containing up to 4 edges of tessellation levels.
gl_TessLevelInner[2] is an array containing up to 2 edges of tessellation levels.



TES Output Topologies: the Isolines Pattern

13

`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.

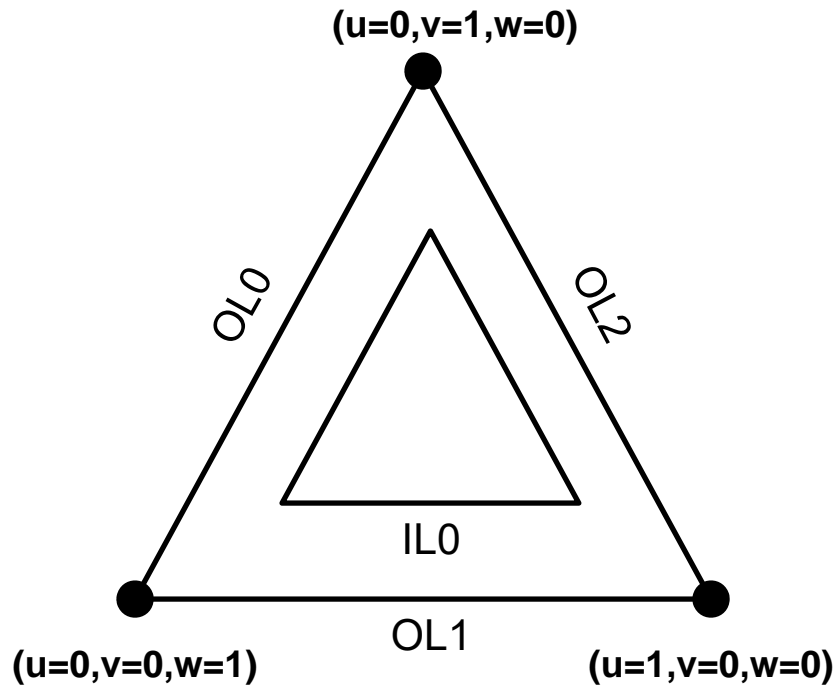


`OL0 == 1` implies that you just want to draw a single curve

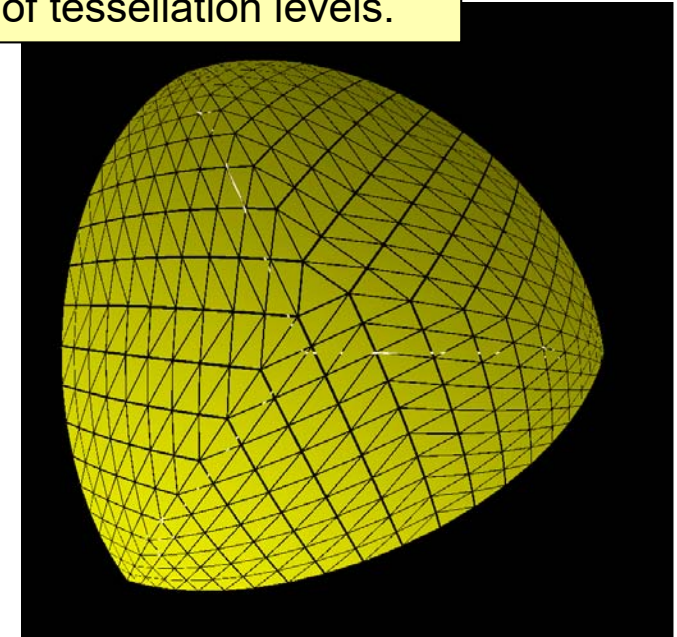
TES Output Topologies: the Triangle Pattern

14

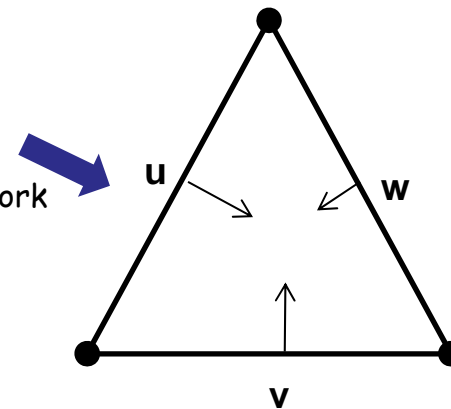
`gl_TessLevelOuter[4]` is an array containing up to 4 edges of tessellation levels.
`gl_TessLevelInner[2]` is an array containing up to 2 edges of tessellation levels.



$$u + v + w = 1$$



How triangle
barycentric
coordinates work



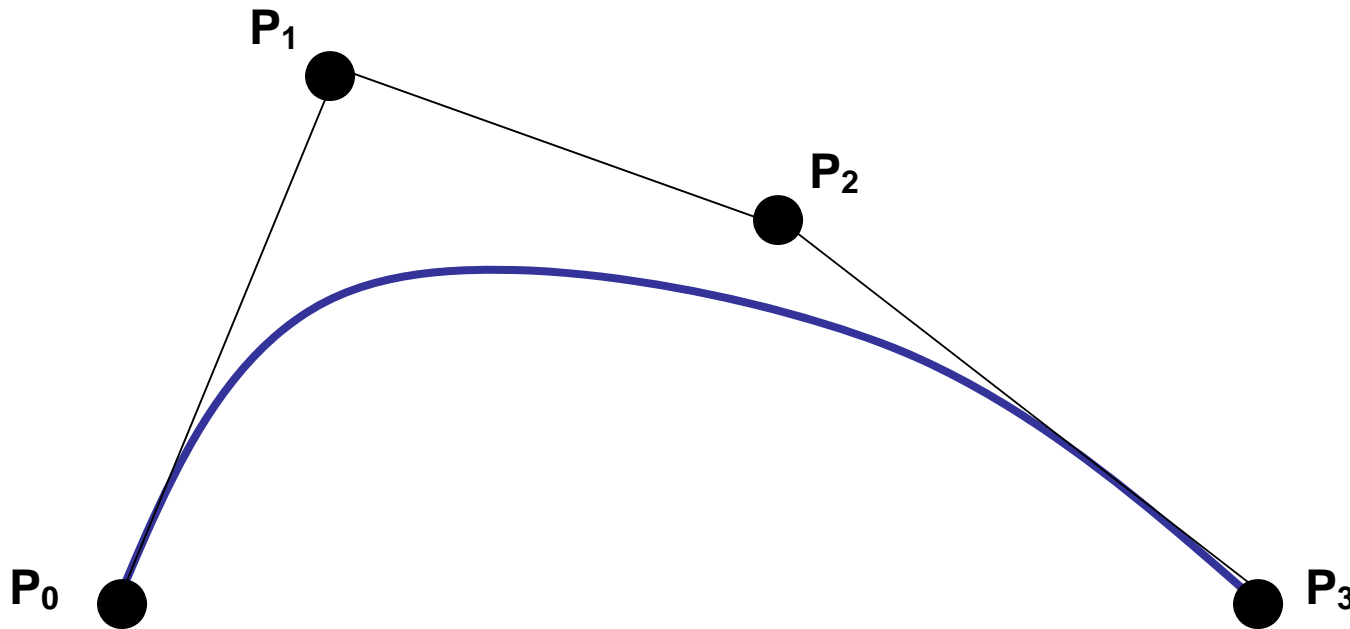
$$u + v + w = 1$$

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

16

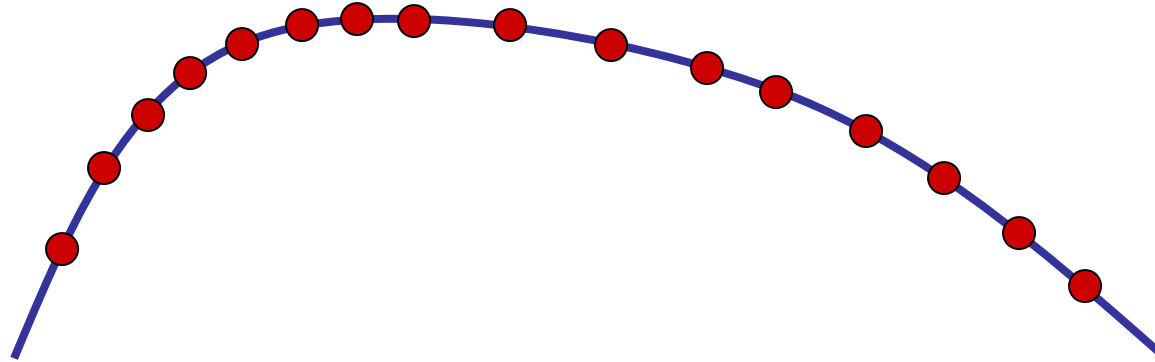


$$P(u) = (1-u)^3 P_0 + 3u(1-u)^2 P_1 + 3u^2(1-u)P_2 + u^3 P_3$$

Need to pass 4 points in to define the curve. Need to pass N points out to draw the curve.

Example: A Bézier Curve

17



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

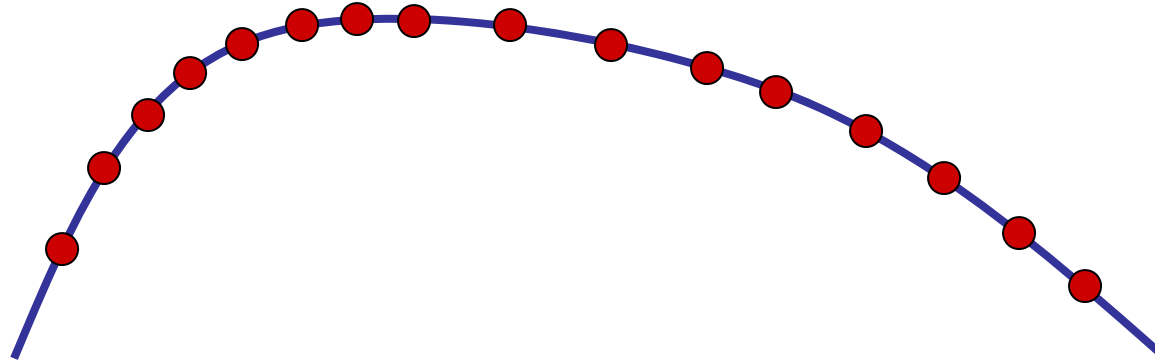
1. You program the Tessellation Control Shader to decide how much to tessellate the curve based on screen area, curvature, etc.

You can even tessellate non-uniformly if you want

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

Example: A Bézier Curve

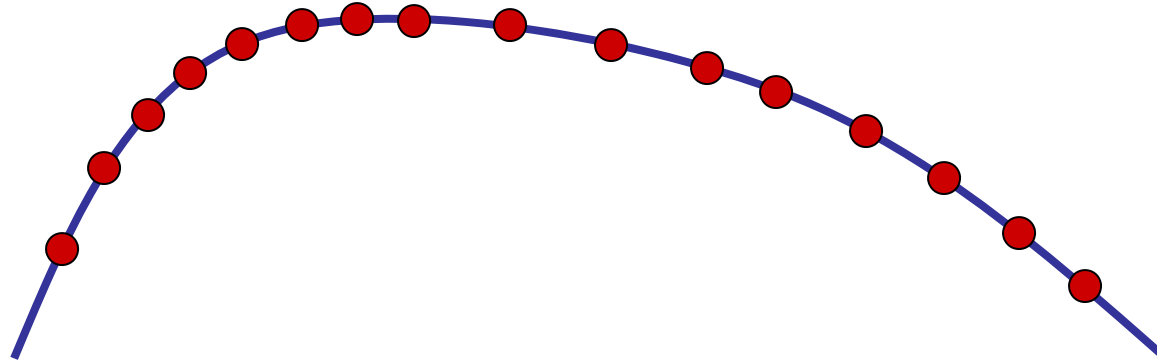
18



2. The Tessellation Primitive Generator generates $u[v,w]$ values for as many subdivisions as the TCS asked for.

Example: A Bézier Curve

19



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

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

$$P(u) = u^3(-P_0 + 3P_1 - 3P_2 + P_3) + u^2(3P_0 - 6P_1 + 3P_2) + u(-3P_0 + 3P_1) + P_0$$



In an OpenGL Program

20

```
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

21

```
##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
glBegin gl_patches
    glVertex 0. 0. 0.
    glVertex 1. 1. 1.
    glVertex 2. 1. 0.
    glVertex 3. 0. 1.
glend
```

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

```
uniform int uOuter0, uOuter1;
```

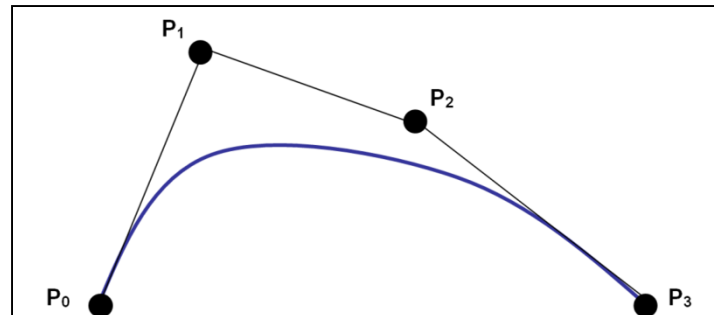
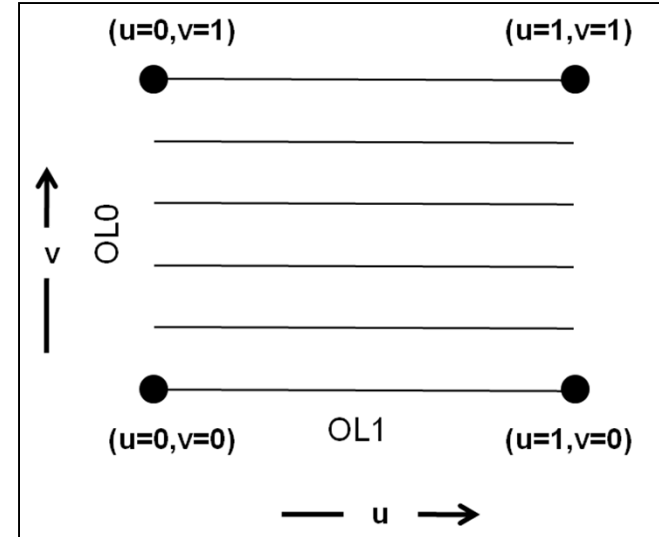
```
layout( vertices = 4 ) out;
```

```
void
main( )
{
```

```
    gl_out[ gl_InvocationID ].gl_Position = gl_in[ gl_InvocationID ].gl_Position;
```

```
    gl_TessLevelOuter[0] = float( uOuter0 );
    gl_TessLevelOuter[1] = float( uOuter1 );
```

```
}
```



```
#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;
```

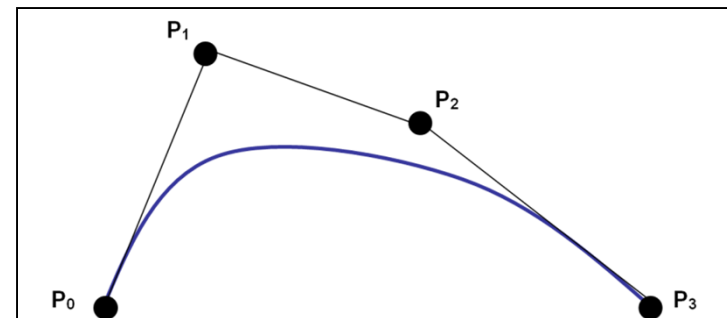
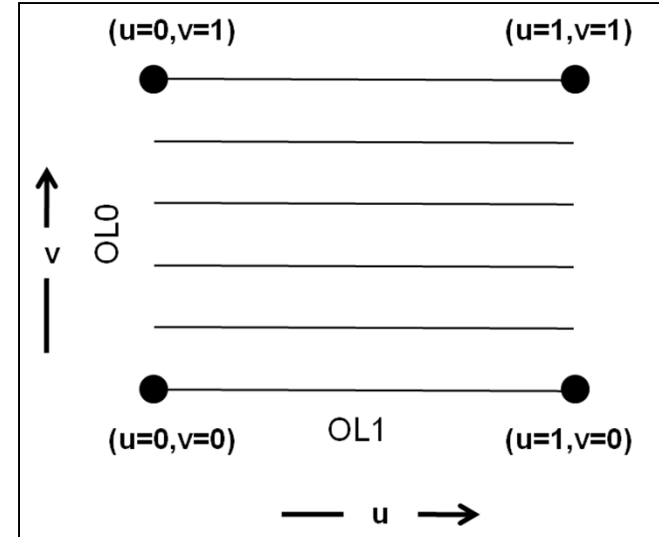
```
    float u = gl_TessCoord.x;
```

```
    // 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;
```

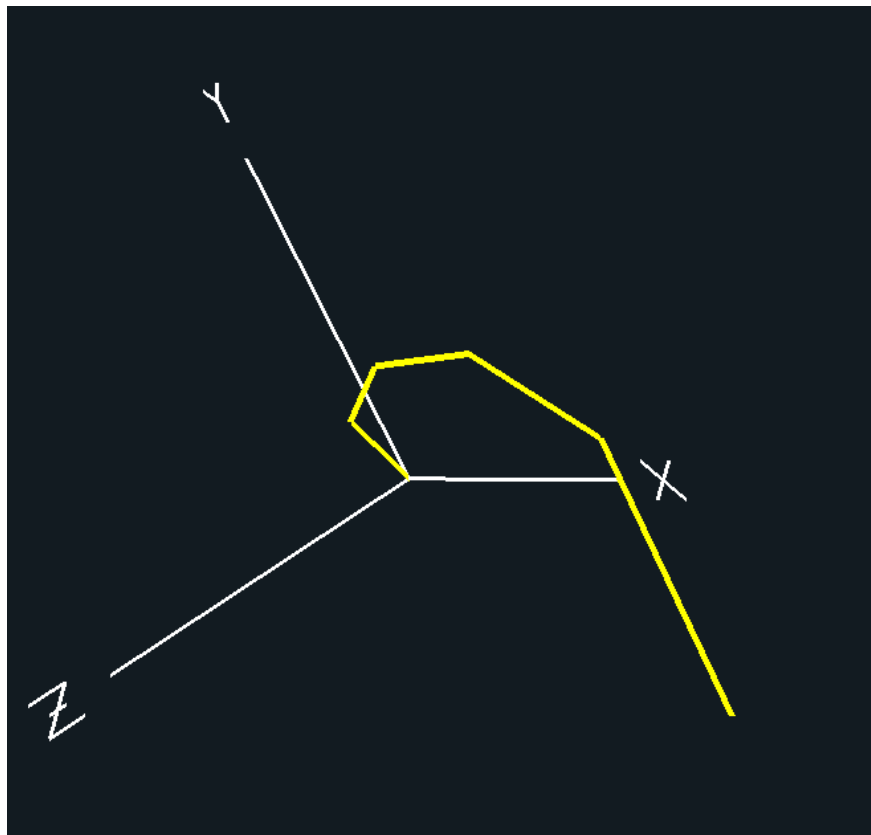
```
}
```



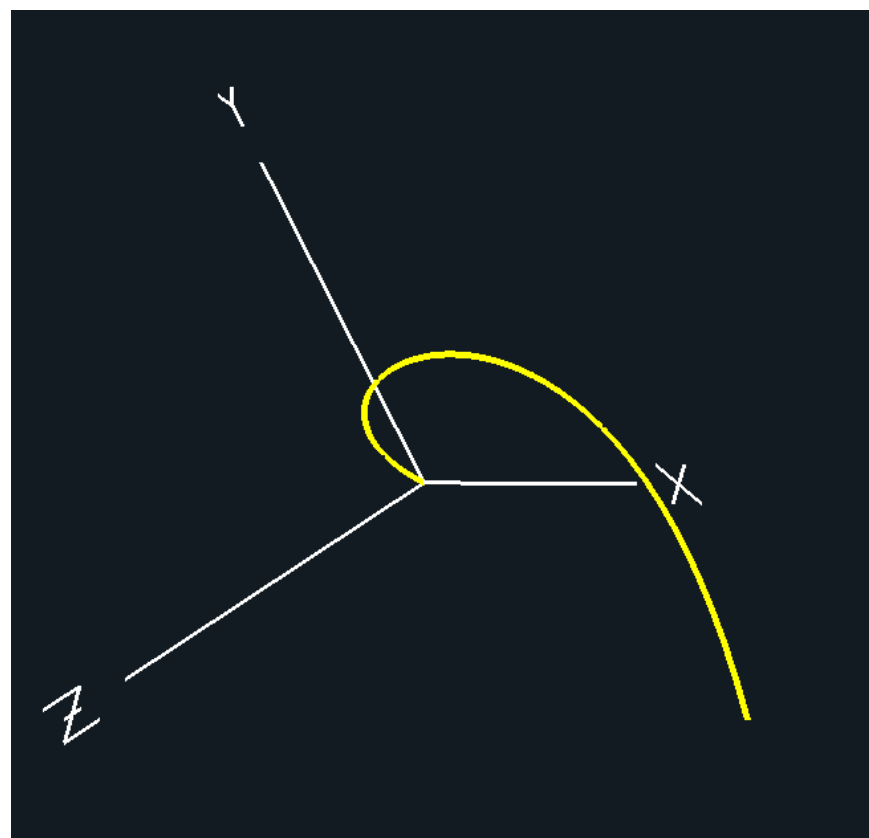
Assigning the intermediate pi's is here to make the code more readable.
From what I have seen, the compiler will optimize this away.

Example: A Bézier Curve

24



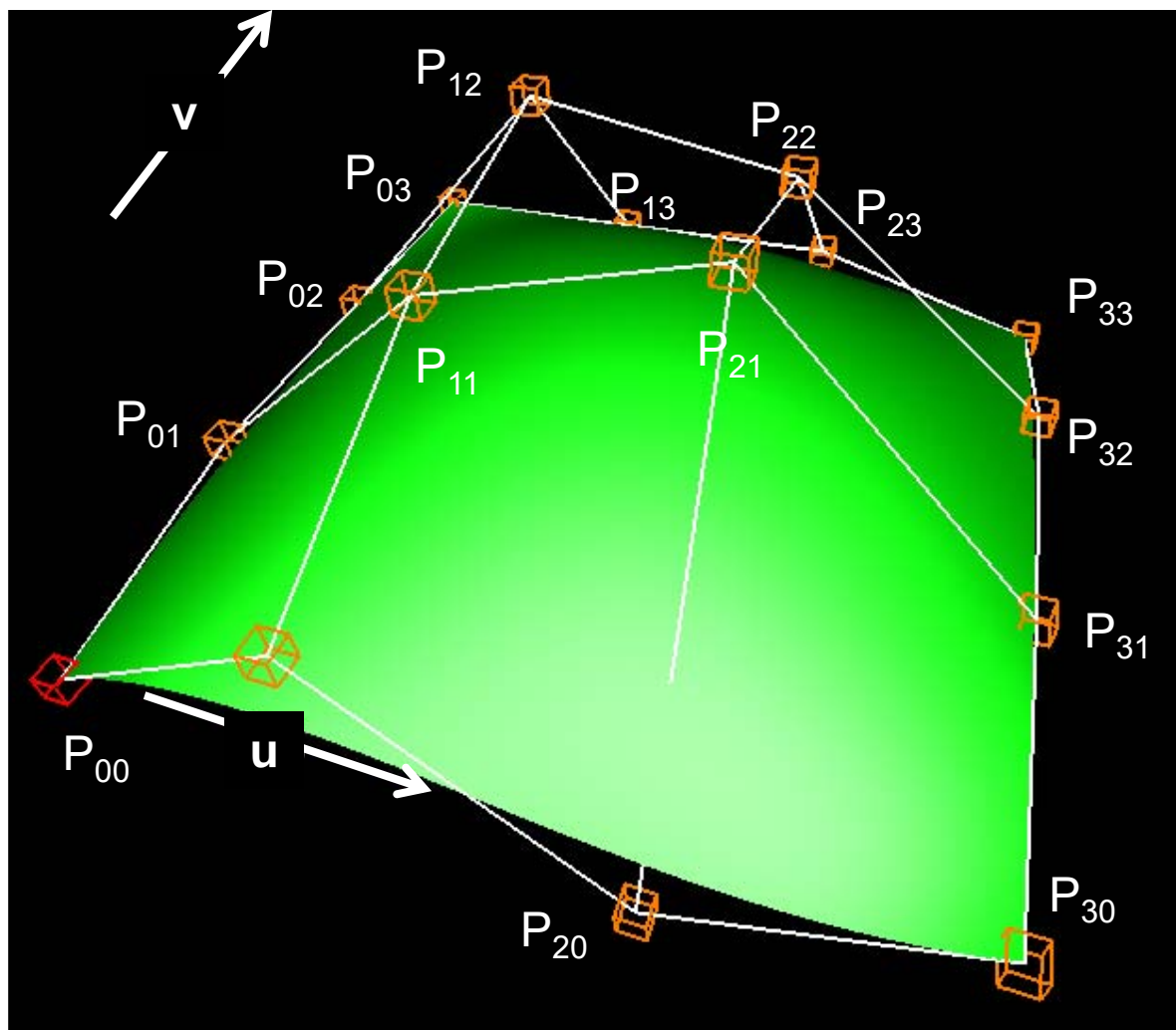
Outer1 = 5



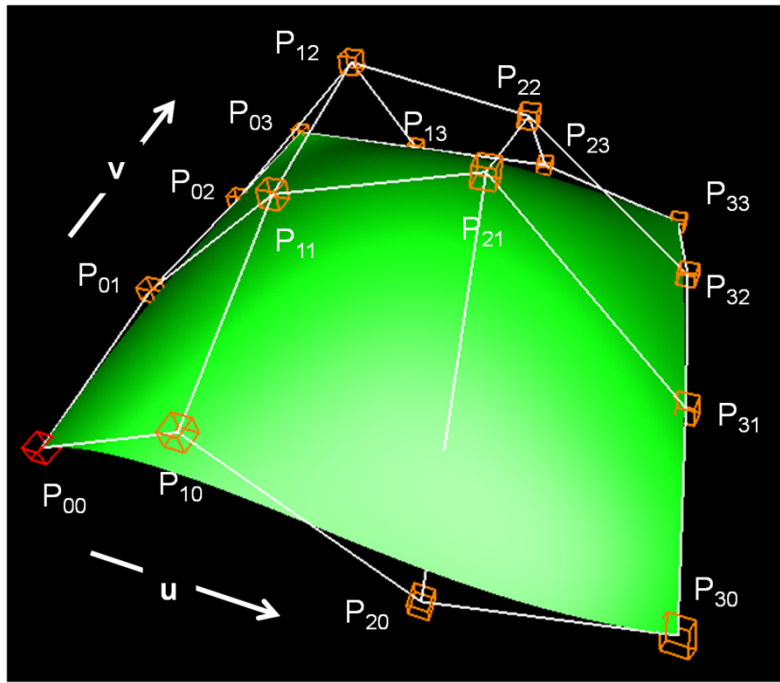
Outer1 = 50

Example: A Bézier Surface

25



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

27

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

```
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

28

```
##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.> u LightZ <-10. 10. 10. >

Color 1. 1. 0. 1.

NumPatchVertices 16

glBegin gl_patches
  glVertex 0. 2. 0.
  glVertex 1. 1. 0.
  glVertex 2. 1. 0.
  glVertex 3. 2. 0.

  glVertex 0. 1. 1.
  glVertex 1. -2. 1.
  glVertex 2. 1. 1.
  glVertex 3. 0. 1.

  glVertex 0. 0. 2.
  glVertex 1. 1. 2.
  glVertex 2. 0. 2.
  glVertex 3. -1. 2.

  glVertex 0. 0. 3.
  glVertex 1. 1. 3.
  glVertex 2. -1. 3.
  glVertex 3. -1. 3.
glEnd
```



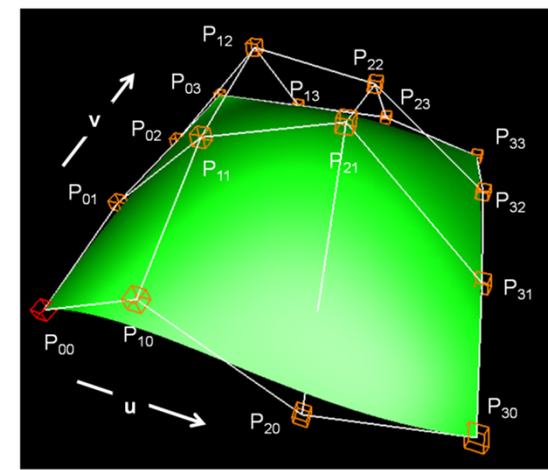
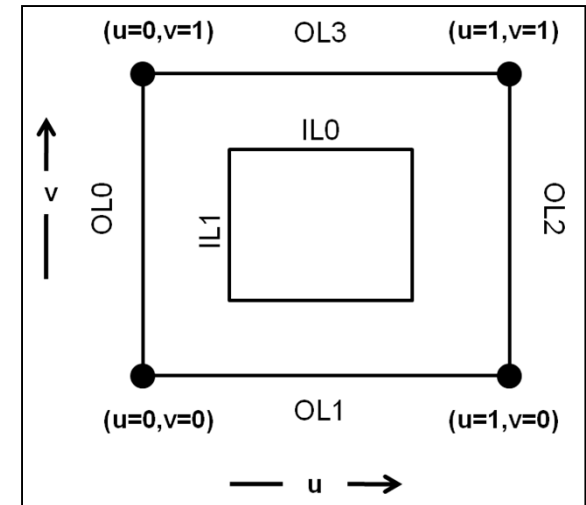
```
#version 400
#extension GL_ARB_tessellation_shader : enable

uniform float uOuter02, uOuter13, ulnner0, ulnner1;

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_TessLevelOuter[1] = gl_TesslevelOuter[3] = uOuter13;
    gl_TessLevelInner[0] = ulnner0;
    gl_TessLevelInner[1] = ulnner1;
}
```



In the TES Shader

30

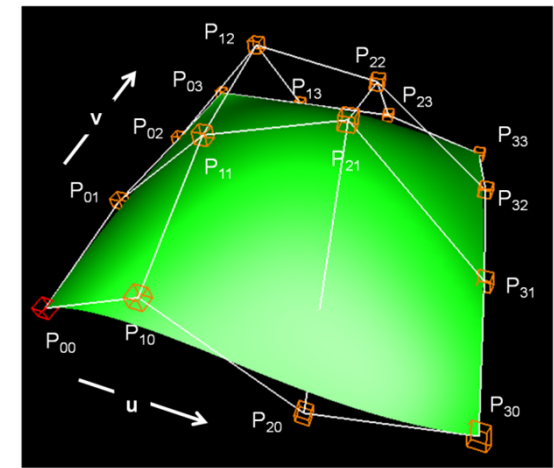
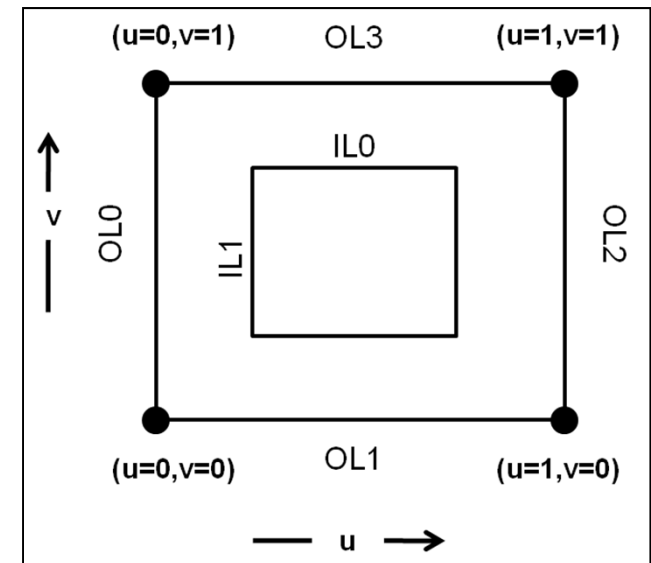
```
#version 400 compatibility
#extension GL_ARB_tessellation_shader : enable

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 p32 = gl_in[11].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

// 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;
```

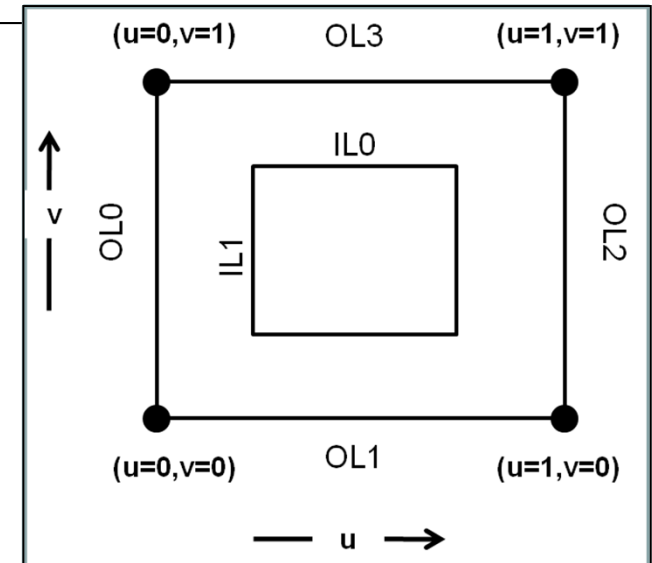
```
float dbu0 = -3. * (1.-u) * (1.-u);
float dbu1 = 3. * (1.-u) * (1.-3.*u);
float dbu2 = 3. * u * (2.-3.*u);
float dbu3 = 3. * u * u;
```

```
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;
```

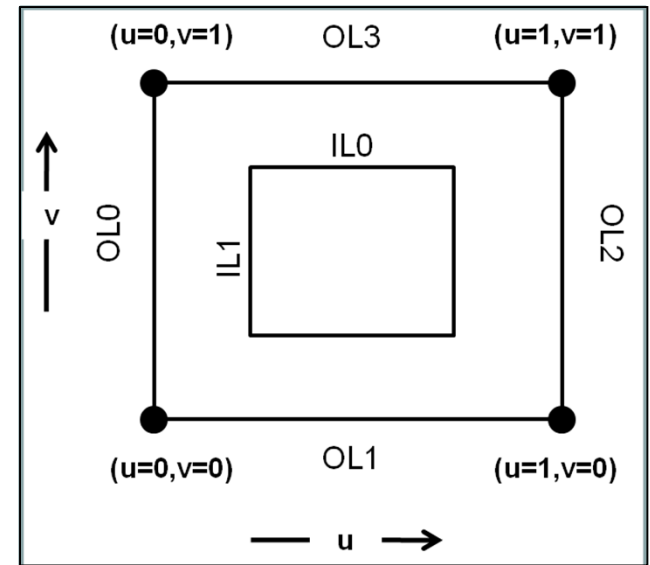
// finally, we get to compute something:

```
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 );
```



In the TES Shader – Computing the Normal, given a u and v

32



Tangent Vectors

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 );
```

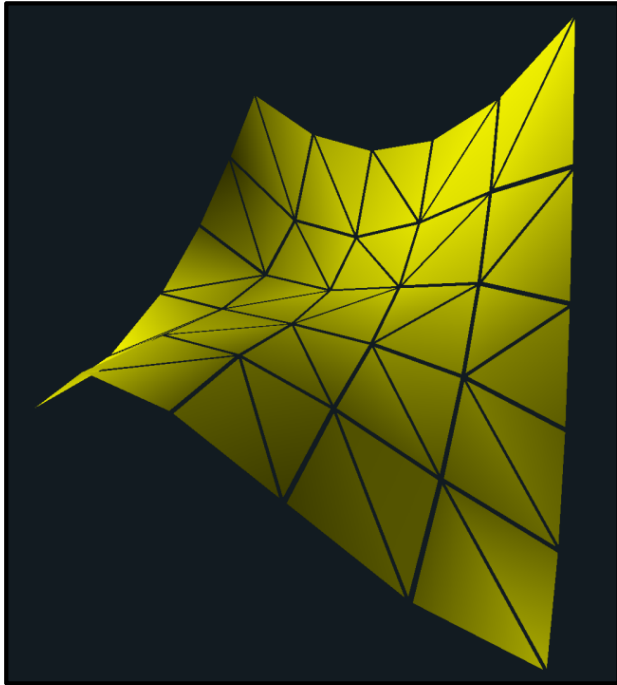
vec4 dpdv =

```
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 ) );
```

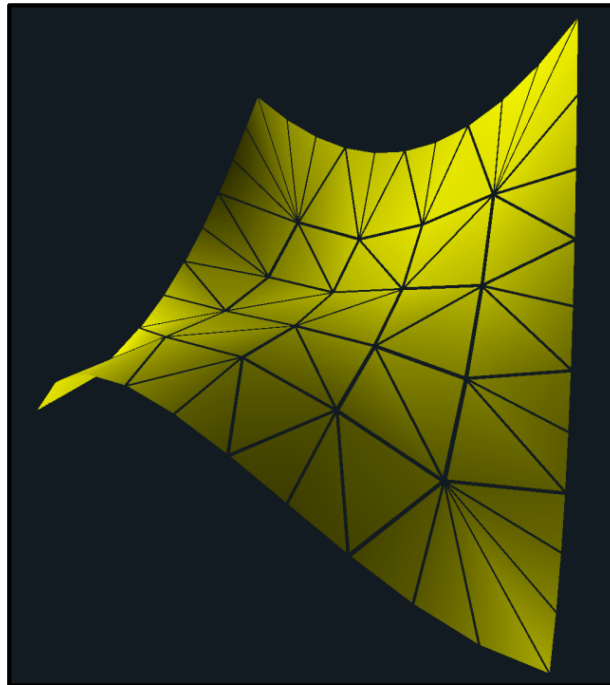
}

Example: A Bézier Surface

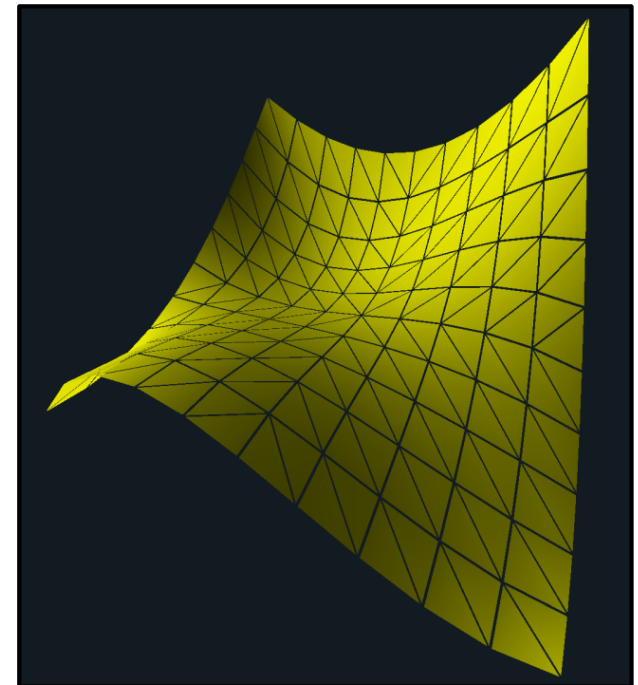


$$\begin{aligned} u_{\text{Outer}02} &= u_{\text{Outer}13} = 5 \\ u_{\text{Inner}0} &= u_{\text{Inner}1} = 5 \end{aligned}$$

$$\begin{aligned} u_{\text{Outer}02} &= u_{\text{Outer}13} = 10 \\ u_{\text{Inner}0} &= u_{\text{Inner}1} = 5 \end{aligned}$$



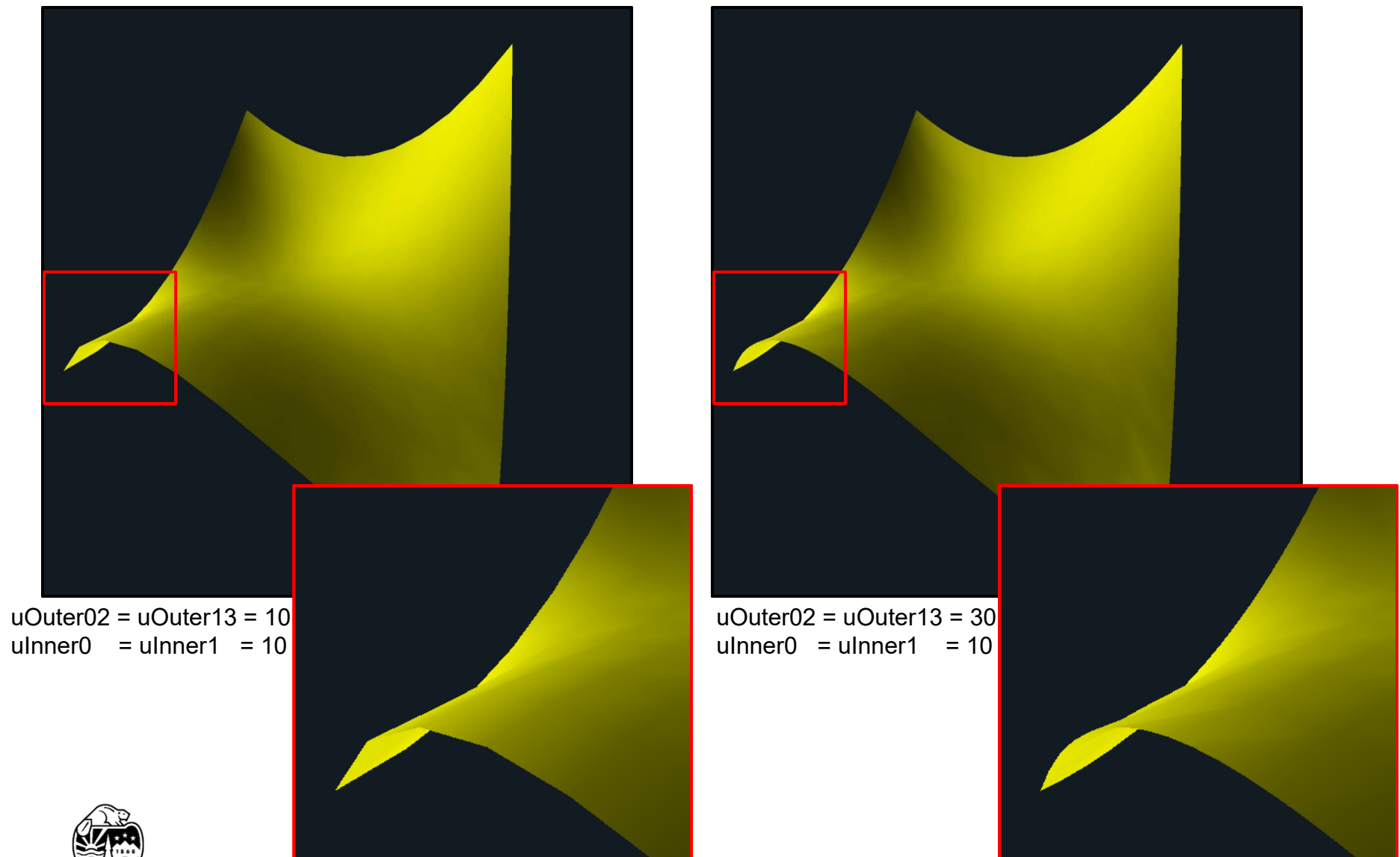
$$\begin{aligned} u_{\text{Outer}02} &= u_{\text{Outer}13} = 10 \\ u_{\text{Inner}0} &= u_{\text{Inner}1} = 10 \end{aligned}$$



mjb - December 31, 2019

Tessellation Levels and Smooth Shading

34



Smoothing edge boundaries is one of the reasons that you can set Outer and Inner tessellation levels separately

Example: Whole-Sphere Subdivision

35

spheresubd.glib

```
##OpenGL GLIB

Vertex      spheresubd.vert
Fragment    spheresubd.frag
TessControl  spheresubd.tcs
TessEvaluation spheresubd.tes
Geometry    spheresubd.geom
Program SphereSubd
    uDetail <1 30 200>
    uScale <0.1 1. 10.>
    uShrink <0. 1. 1.>
    uFlat <false>
    uColor {1. 1. 0. 0.}
    uLightX <-10. 5. 10.> uLightY <-10. 10. 10.> uLightZ <-10. 10. 10.>

Color 1. 1. 0.

NumPatchVertices 1

glBegin gl_patches
    glVertex 0. 0. 0. .2
    glVertex 0. 1. 0. .3
    glVertex 0. 0. 1. .4
glEnd
```

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



Example: Whole-Sphere Subdivision

36

spheresubd.vert

```
#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
}
```

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



Example: Whole-Sphere Subdivision

37

spheresubd.tcs

```
#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

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

38

spheresubd.tes

```
#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. );
```

```
}
```

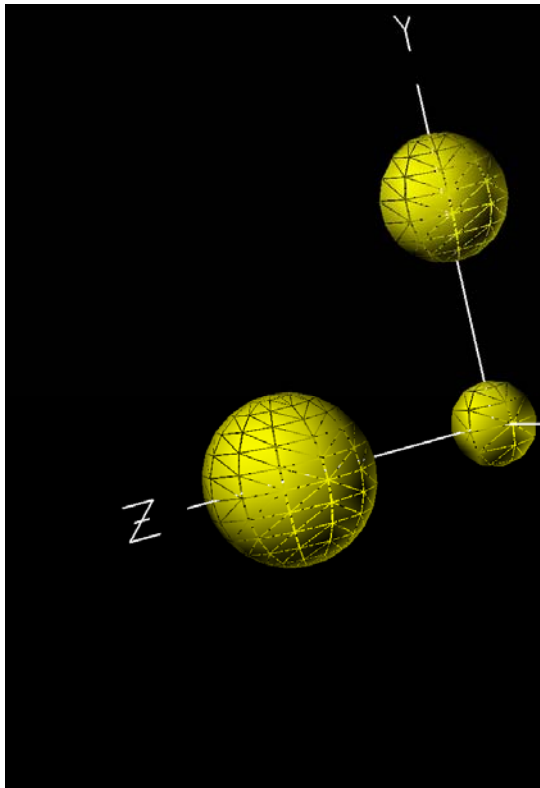
$$-\frac{\pi}{2} \leq \phi \leq +\frac{\pi}{2}$$

$$-\pi \leq \theta \leq +\pi$$

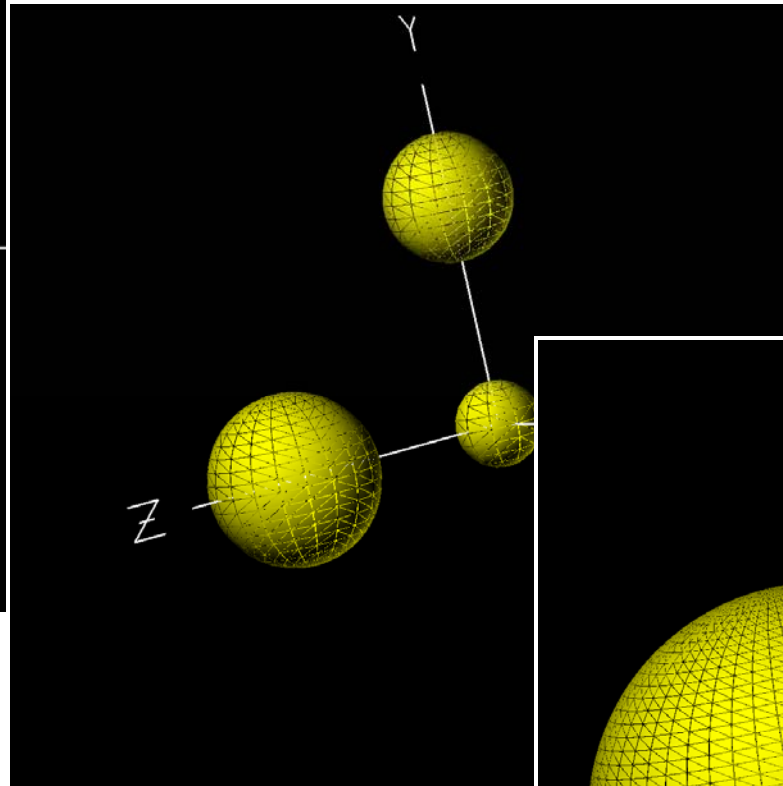
Turning u and v into
spherical coordinates

Example: Whole-Sphere Subdivision

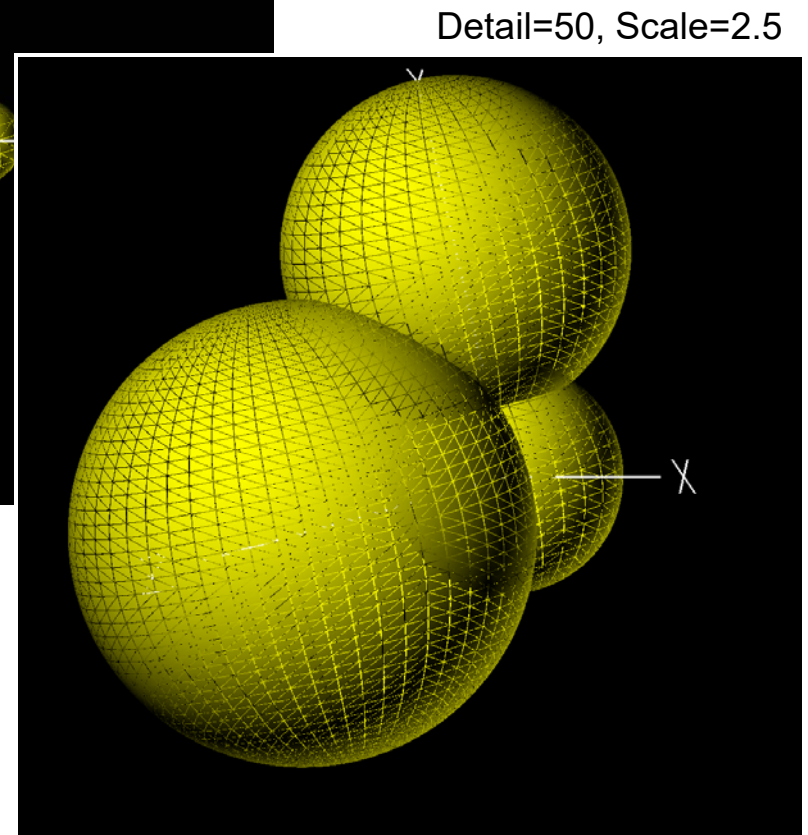
39



Detail=30, Scale=1.



Detail=50, Scale=1.



Detail=50, Scale=2.5

sphereadapt.tcs, I

```
#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. );
```

Extreme points of the sphere



sphereadapt.tcs, II

```
mx = gl_ModelViewProjectionMatrix * mx;  
px = gl_ModelViewProjectionMatrix * px;  
my = gl_ModelViewProjectionMatrix * my;  
py = gl_ModelViewProjectionMatrix * py;  
mz = gl_ModelViewProjectionMatrix * mz;  
pz = gl_ModelViewProjectionMatrix * pz;
```

Extreme points of the sphere in Clip space

```
mx.xy /= mx.w;  
px.xy /= px.w;  
my.xy /= my.w;  
py.xy /= py.w;  
mz.xy /= mz.w;  
pz.xy /= pz.w;
```

Extreme points of the sphere in NDC space

```
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 );
```

How large are the lines between the extreme points?

```
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.)



sphereadapt.tes

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

```
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 );
```

$$-\frac{\pi}{2} \leq \phi \leq +\frac{\pi}{2}$$

$$-\pi \leq \theta \leq +\pi$$

```
    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. );
```

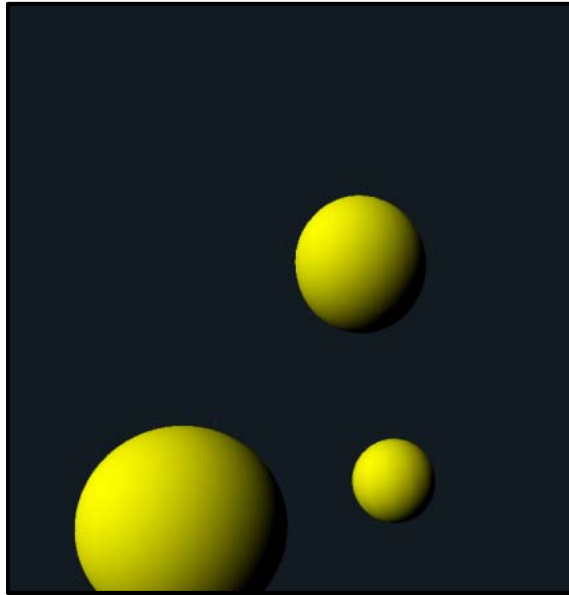
```
}
```

← Spherical coordinates

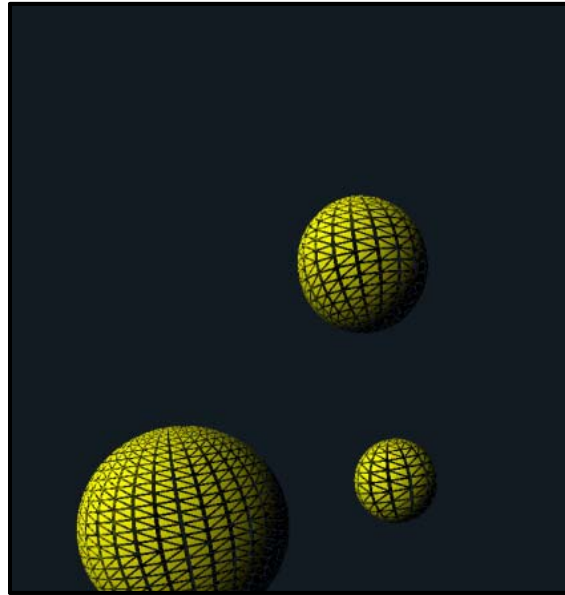
← No longer uses uScale

Making the Whole-Sphere Subdivision Adapt to Screen Coverage

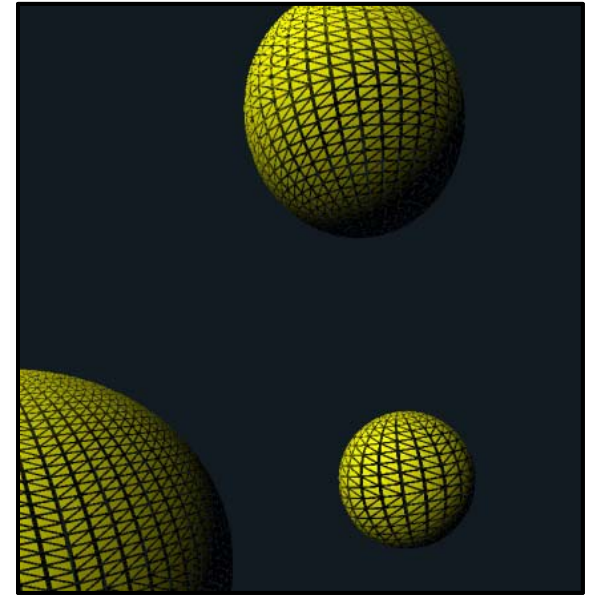
43



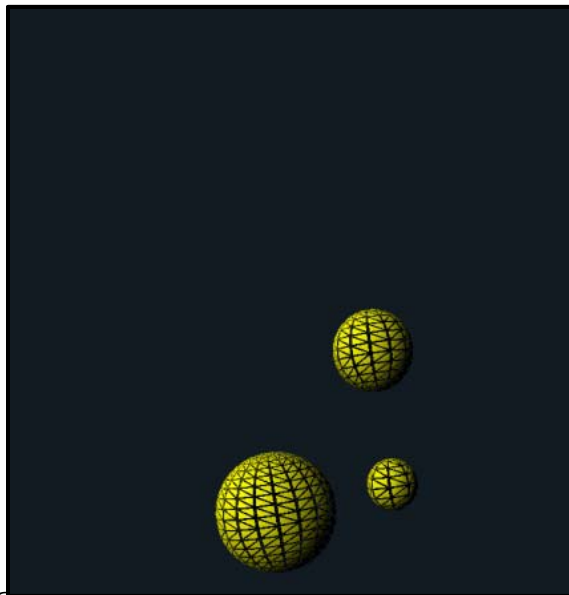
Original



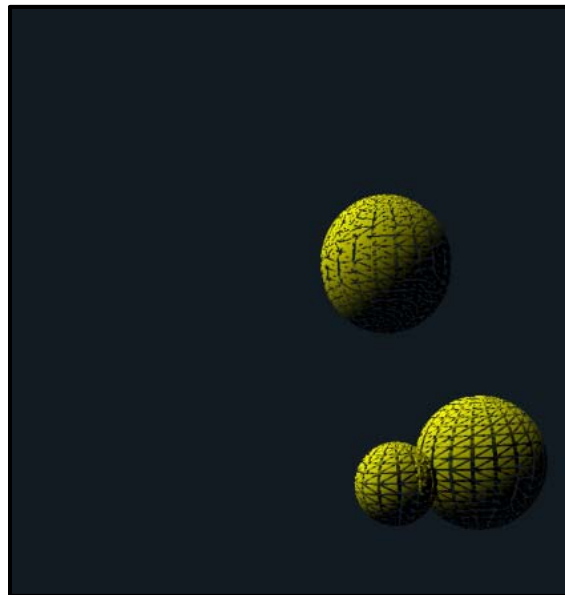
Triangles Shrunk



Zoomed In



Zoomed Out



Rotated

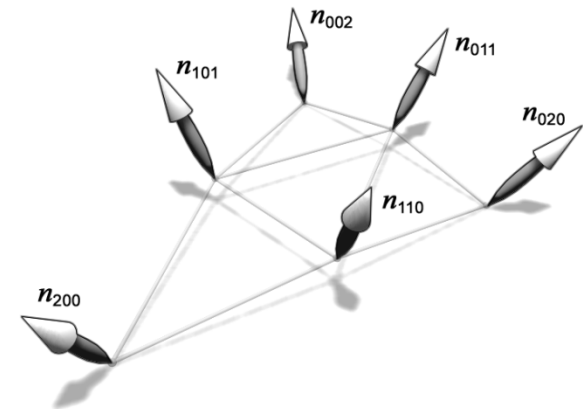
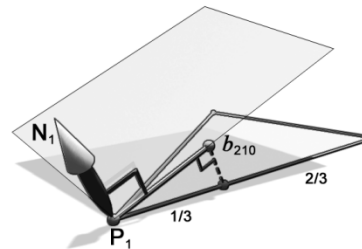
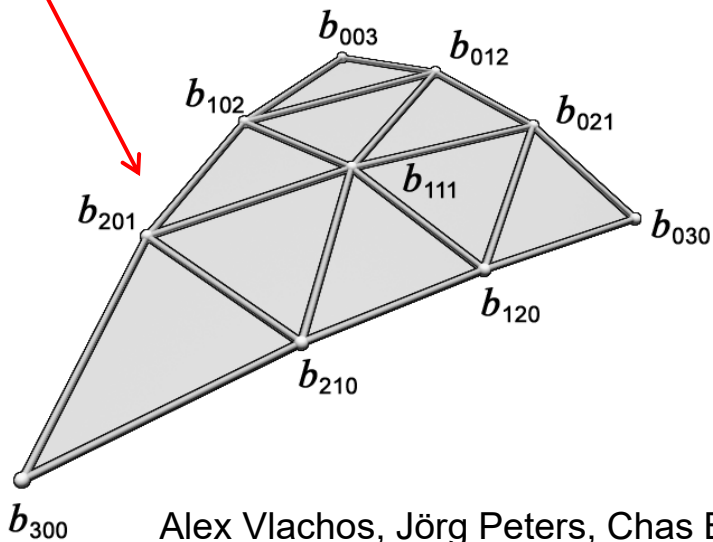
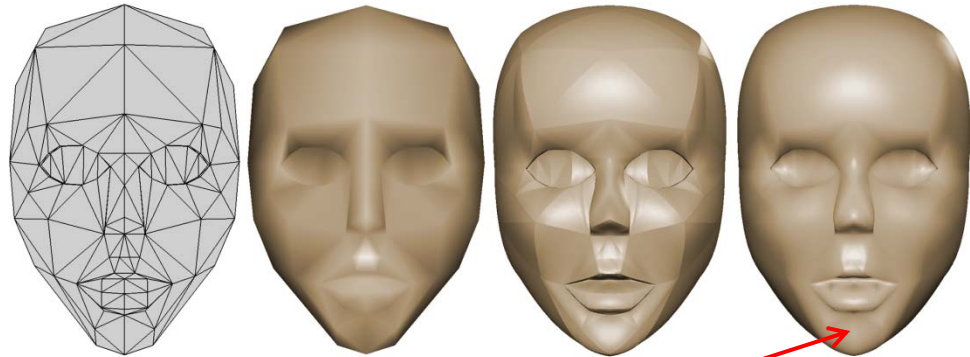
Notice that the number of triangles adapts to the screen coverage of each sphere, and that the size of the tessellated triangles stays about the same, regardless of radius or transformation

Compute

Example: PN Triangles

44

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.



Alex Vlachos, Jörg Peters, Chas Boyd, and Jason Mitchell, "Curved PN Triangles", *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pp.159 – 166.



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 drivers before tessellation shaders made it unnecessary.

Example: PN Triangles

pntriangles.vert

```
#version 400 compatibility
uniform float uScale;
out vec3      vNormal;

void
main( )
{
    vec3 xyz = gl_Vertex.xyz;
    xyz *= uScale;
    gl_Position = gl_ModelViewMatrix * vec4( xyz, 1. );
    vNormal = normalize( gl_NormalMatrix * gl_Normal );
}
```

pntriangles.tcs

```
#version 400 compatibility
#extension GL_ARB_tessellation_shader : enable
uniform int    uOuter, uInner;
uniform float  uScale;

layout( vertices = 3 ) out;
in vec3      vNormal[ ];
out vec3     tcNormals[ ];

void main( )
{
    tcNormals[ gl_InvocationID ] = vNormal[ gl_InvocationID ];
    gl_out[ gl_InvocationID ].gl_Position = gl_in[ gl_InvocationID ].gl_Position;

    gl_TessLevelOuter[0] = uScale * float(uOuter);
    gl_TessLevelOuter[1] = uScale * float(uOuter);
    gl_TessLevelOuter[2] = uScale * float(uOuter);
    gl_TessLevelInner[0] = uScale * float(uInner);
}
```

Example: PN Triangles

46

pntriangles.tes, I

```
#version 400 compatibility
#extension GL_ARB_tessellation_shader : enable
in vec3   tcNormals[ ];
out vec3  teNormal;

layout( triangles, equal_spacing, ccw) in;

void
main( )
{
    vec3 p1 = gl_in[0].gl_Position.xyz;
    vec3 p2 = gl_in[1].gl_Position.xyz;
    vec3 p3 = gl_in[2].gl_Position.xyz;

    vec3 n1 = tcNormals[0];
    vec3 n2 = tcNormals[1];
    vec3 n3 = tcNormals[2];

    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    float w = gl_TessCoord.z;

    vec3 b300 = p1;
    vec3 b030 = p2;
    vec3 b003 = p3;

    float w12 = dot( p2 - p1, n1 );
    float w21 = dot( p1 - p2, n2 );
    float w13 = dot( p3 - p1, n1 );
    float w31 = dot( p1 - p3, n3 );
    float w23 = dot( p3 - p2, n2 );
    float w32 = dot( p2 - p3, n3 );
```

Example: PN Triangles

pntriangles.tes, II

```

vec3 b210 = ( 2.*p1 + p2 - w12*n1 ) / 3.;
vec3 b120 = ( 2.*p2 + p1 - w21*n2 ) / 3.;
vec3 b021 = ( 2.*p2 + p3 - w23*n2 ) / 3.;
vec3 b012 = ( 2.*p3 + p2 - w32*n3 ) / 3.;
vec3 b102 = ( 2.*p3 + p1 - w31*n3 ) / 3.;
vec3 b201 = ( 2.*p1 + p3 - w13*n1 ) / 3.;

vec3 ee = ( b210 + b120 + b021 + b012 + b102 + b201 ) / 6.;
vec3 vv = ( p1 + p2 + p3 ) / 3.;
vec3 b111 = ee + ( ee - vv ) / 2.;

vec3 xyz = 1.*b300*w*w*w + 1.*b030*u*u*u + 1.*b003*v*v*v +
3.*b210*u*w*w + 3.*b120*u*u*w + 3.*b201*v*w*w +
3.*b021*u*u*v + 3.*b102*v*v*w + 3.*b012*u*v*v +
6.*b111*u*v*w;

float v12 = 2. * dot( p2-p1, n1+n2 ) / dot( p2-p1, p2-p1 );
float v23 = 2. * dot( p3-p2, n2+n3 ) / dot( p3-p2, p3-p2 );
float v31 = 2. * dot( p1-p3, n3+n1 ) / dot( p1-p3, p1-p3 );

vec3 n200 = n1;
vec3 n020 = n2;
vec3 n002 = n3;
vec3 n110 = normalize( n1 + n2 - v12*(p2-p1) );
vec3 n011 = normalize( n2 + n3 - v23*(p3-p2) );
vec3 n101 = normalize( n3 + n1 - v31*(p1-p3) );

Normal = n200*w*w + n020*u*u + n002*v*v +
n110*w*u + n011*u*v + n101*w*v;

gl_Position = vec4( xyz, 1. );
}

```

Example: PN Triangles

48

pntriangles.geom

```
#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;
    V[2] = gl_PositionIn[2].xyz;

    CG = ( V[0] + V[1] + V[2] ) / 3.;

    ProduceVertex( 0 );
    ProduceVertex( 1 );
    ProduceVertex( 2 );
}
```


Example: PN Triangles

49

pntriangles.frag

```
#version 400 compatibility

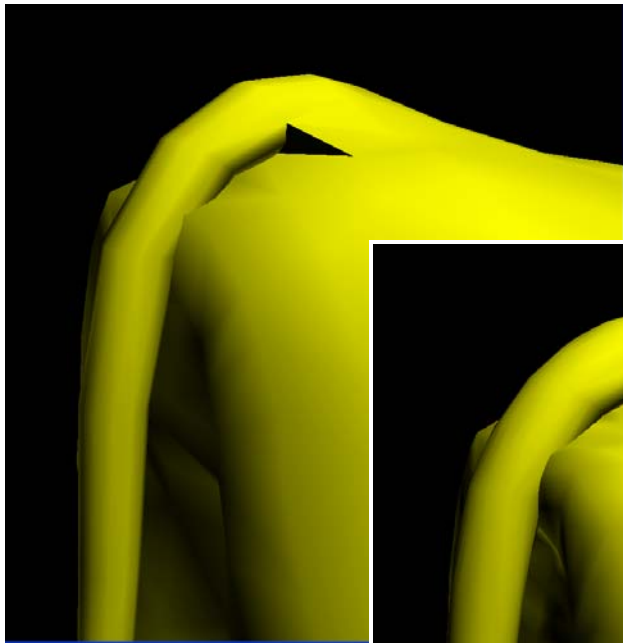
in float    gLightIntensity;

const vec3 COLOR = vec3( 1., 1., 0. );

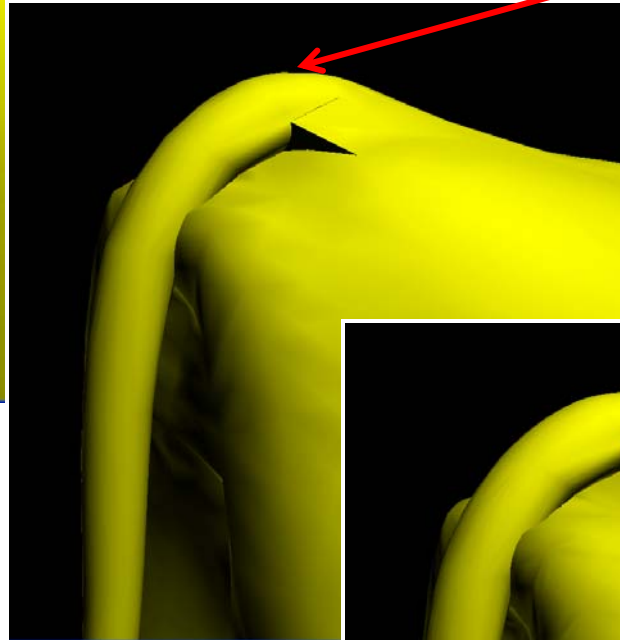
void
main( )
{
    gl_FragColor = vec4( gLightIntensity*COLOR, 1. );
}
```

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

50

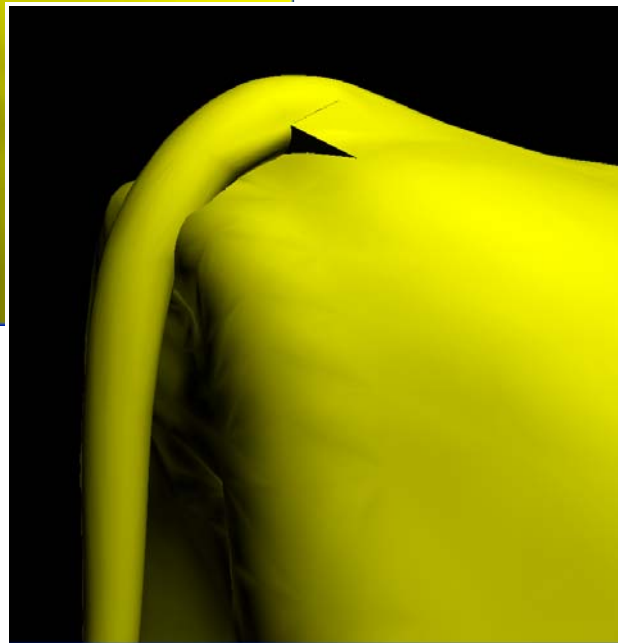


$u_{Outer} = 1, u_{lnner} = 1$

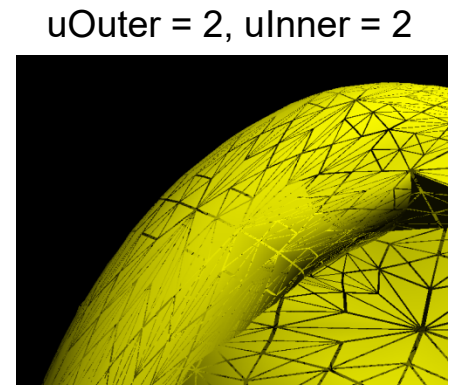


$u_{Outer} = 2, u_{lnner} = 1$

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.



$u_{Outer} = 2, u_{lnner} = 2$



$u_{Outer} = 2, u_{lnner} = 2$

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).

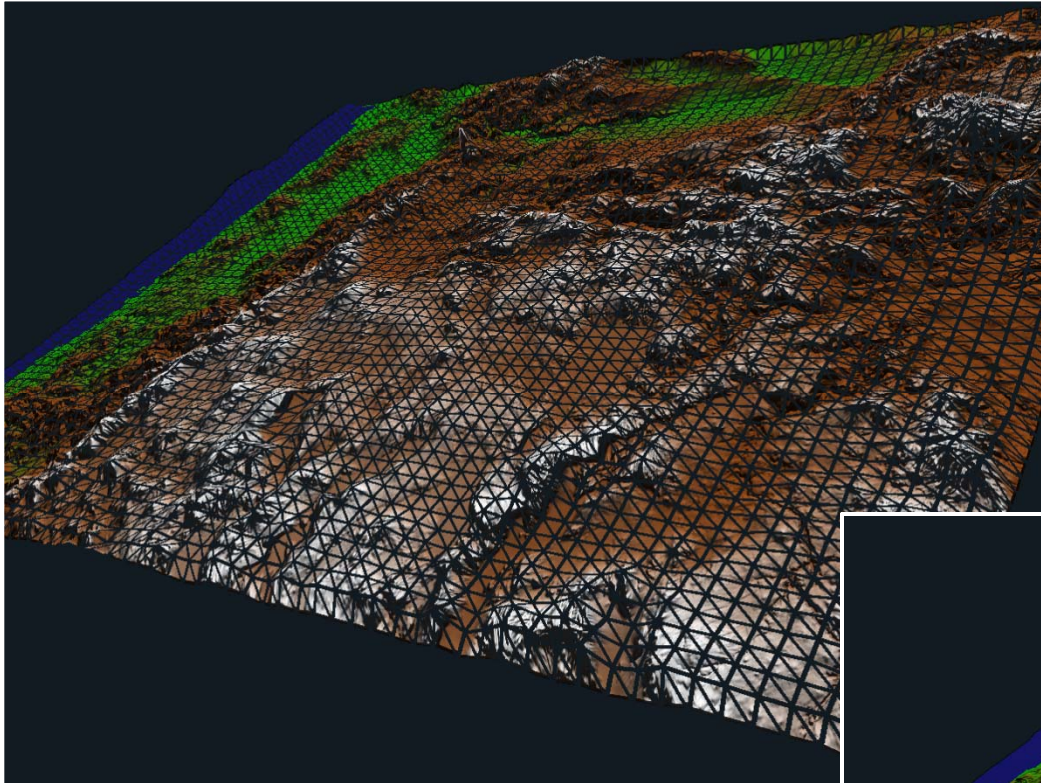
Use a **Tessellation Shader** when:

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

52



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.

