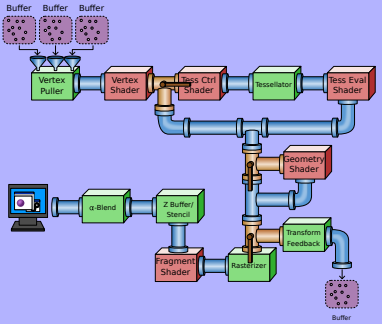
Tessellation Shaders

Motivation

- We've seen tessellation shaders for lines
- Now we'll discuss using them with triangles

Review

- Recall GPU pipeline



TCS & TES

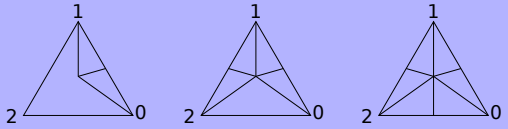
- Recall: TCS and TES work together to direct tessellator actions
- TCS says how much subdivision to do
- TES says what kind of subdivision to do
- For this discussion, we'll assume triangular tessellation

TCS

- TCS must write to gl_TessLevelOuter[0...2]
  - Tells how much to subdivide each edge of the triangle
- Also must write to gl_TessLevelInner[0]
  - Tells how much to subdivide interior of triangle
- If zero written to any of these: Entire patch is discarded

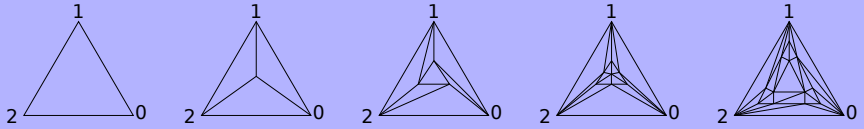
Examples

- gl_TessLevelInner is 1 in all of these
- gl_TessLevelOuter is {2,1,1}, {2,2,1}, {2,2,2} here



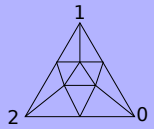
Examples

- glTessLevelOuter is {1,1,1} for all of these
- gl_TessLevelInner is 1, 2, 3, 4, and 5



Example

- Outer is {2,2,2} and inner is 3



TES

- TES should begin with line:
  layout(triangles)in;

Code

- First, we'll develop program that just draws input mesh
- C++ code:

```
void draw(){
    cam->setUniforms();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    Program::setUniform("worldMatrix", mat4::identity() );
    meshprog->use();
    glPatchParameteri(GL_PATCH_VERTICES,3);
    M->draw(GL_PATCHES);
    fpsText->draw();
}
```

VS

- Vertex shader: Just a passthrough:

```
layout(location=0) in vec3 position;
layout(location=1) in vec2 texCoord;
layout(location=2) in vec3 normal;

out vec2 v_texCoord;
out vec3 v_normal;
out vec3 v_position;

void main(){
   v_texCoord = texCoord;
   v_normal = normal;
   v_position = position;
}
```

TCS

- TCS: Notice vertices=3
- Since patch size is 3, system will pull 3 vertices off input, run TCS 3 times, then feed result to tessellator

```
layout(vertices=3) out;

in vec3 v_position[];
in vec2 v_texCoord[];
in vec3 v_normal[];

out vec3 tcs_position[];
out vec3 tcs_normal[];
out vec2 tcs_texCoord[];

void main(){
    gl_TessLevelOuter[0] = 1;
    gl_TessLevelOuter[1] = 1;
    gl_TessLevelOuter[2] = 1;
    gl_TessLevelInner[0] = 1;
    tcs_position[gl_InvocationID] = v_position[gl_InvocationID];
    tcs_normal[gl_InvocationID] = v_normal[gl_InvocationID];
    tcs_texCoord[gl_InvocationID] = v_texCoord[gl_InvocationID];
}
```

TES

- For TES, we'll need to define some helper functions:

```
vec3 interpolate(float u, float v, float w, vec3 a, vec3 b, vec3
    c ){
     //simple barycentric interpolation
     vec3 p0 = u * a;
     vec3 p1 = v * b;
     vec3 p2 = w * c;
     return (p0+p1+p2);
}
vec2 interpolate(float u, float v, float w, vec2 a, vec2 b, vec2
    c ){
     vec2 p0 = u * a;
     vec2 p1 = v * b;
     vec2 p2 = w * c;
     return (p0+p1+p2);
}
```

TES

- Now for the declarations.
- Notice we have this declared as taking triangles (first line):

```
layout(triangles) in;

in vec3 tcs_position[];
in vec3 tcs_normal[];
in vec2 tcs_texCoord[];

out vec3 tes_worldPos;
out vec2 tes_texCoord;
out vec3 tes_normal;

void main(){
    ...
}
```

TES

- Determine position of points:

```
vec3 p1 = interpolate(
    gl_TessCoord[0], gl_TessCoord[1], gl_TessCoord[2],
    tcs_position[0], tcs_position[1], tcs_position[2] );
vec3 n1 = interpolate(
    gl_TessCoord[0], gl_TessCoord[1], gl_TessCoord[2],
    tcs_normal[0], tcs_normal[1], tcs_normal[2] );
tes_texCoord = interpolate(
    gl_TessCoord[0], gl_TessCoord[1], gl_TessCoord[2],
    tcs_texCoord[0], tcs_texCoord[1], tcs_texCoord[2] );
```

TES

- Rest of the code would look very familiar in a vertex shader:

```
vec4 p = vec4(p1,1.0);
p = p * worldMatrix;
tes_worldPos = p.xyz;
p = p * viewMatrix;
p = p * projMatrix;
gl_Position = p;

vec4 n = vec4(n1,0.0);
n = n * worldMatrix;
tes_normal = n.xyz;
```

TES

- gl_TessCoord[0...2] will have the u,v,w barycentric coordinates
- We need to interpolate all three items (position, normal, texture coordinates)
- Also output data for FS

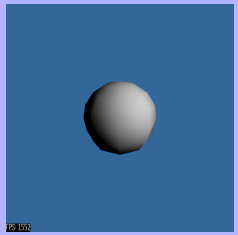
FS

- Standard FS:

```
in vec2 tes_texCoord;
in vec3 tes_normal;
in vec3 tes_worldPos;

out vec4 color;
layout(binding=0) uniform sampler2DArray tex;

void main(){
    vec4 c = texture( tex, vec3(tes_texCoord,0.0) );
    vec3 L = normalize(lightPosition - tes_worldPos);
    vec3 N = normalize(tes_normal);
    float dp = max( dot(L,N), 0.0 );
    color.rgb = dp * diffuse * c.rgb;
    color.a = c.a;
}
```

Result

- We’re using a fairly coarse mesh here with smoothed normals:



Applications

- What can we do with triangle tessellation?
- One use: Adaptive surface refinement
- First method we'll look at: Implicit Surfaces

Implicit Surfaces

- Implicit surface: Surface is described by mathematical equation
- Example: Sphere of radius 1
- We begin by rendering an octahedron
- TCS sets subdivision level
- TES pushes each resulting vertex to correct distance from origin and computes radius

VS

- Nothing special here: Just pass data through
- We've already seen this, so no need to show it again.

TCS

- TCS copies data and defines tessellation level
- main() is the only change from what we had previously:

```
void main(){
    //tessFactor is a uniform
    gl_TessLevelOuter[0] = tessFactor;
    gl_TessLevelOuter[1] = tessFactor;
    gl_TessLevelOuter[2] = tessFactor;
    gl_TessLevelInner[0] = tessFactor;
    tcs_position[gl_InvocationID] = v_position[gl_InvocationID];
    tcs_normal[gl_InvocationID] = v_normal[gl_InvocationID];
    tcs_texCoord[gl_InvocationID] = v_texCoord[gl_InvocationID];
}
```

TES

- TES is where the main work happens

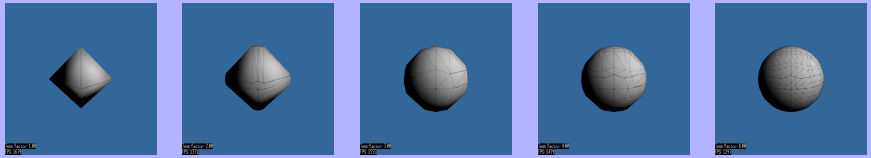
```
void main(){
    vec3 p1 = interpolate(gl_TessCoord[0], gl_TessCoord[1],
       gl_TessCoord[2],
        tcs_position[0], tcs_position[1], tcs_position[2] );
    p1 = normalize(p1);
    vec3 n1 = p1;
    tes_texCoord = interpolate(gl_TessCoord[0], gl_TessCoord[1],
       gl_TessCoord[2],
        tcs_texCoord[0], tcs_texCoord[1], tcs_texCoord[2] );
    ...multiply p1 with worldMatrix, viewMatrix, projMatrix...
    ...write p1 to gl_Position...
    ...multiply n with worldMatrix, write to tes_normal
}
```

FS

- FS is as usual

Results

- Tessellation levels of 1, 2, 3, 4, 8



Problem

- Usually, we'd use a lower level of tessellation for model when it's far away and switch to more detailed level as it gets closer
- Problem: Can cause "popping" effect when levels change
  - Very distracting visually
- We can solve this with fractional tessellation values
  - Make sure first line of TES is:
    layout(triangles,fractional_odd_spacing) in;
- Now, fractional values give intermediate positions
- Allows smooth transition
- fractional.ogv

Phong Tessellation

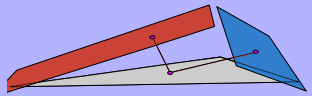
- Implicit surfaces work well if object can be described via simple mathematical formula
  - Ex: Sphere, rounded cube, cone, cylinder, ...
- But what if we have a more complex object? (Such as a mesh)
- Another method of refinement: Phong tessellation

Phong Tessellation

- Given: Three vertices of triangle, v_0, v_1, v_2 with normals n_0, n_1, n_2
- Recall: We can define a plane using a point and a normal
  - Plane equation: Ax+By+Cz+D=0
  - At vertex i: A,B,C=n_i and D=$-(n_i \cdot v_i)$
- So we can define three planes, one at each vertex of the triangle
- We project original surface point to each plane then interpolate its position

Idea

- Each vertex has a smooth normal
- That normal + the vertex location define a plane
- Project surface point to that plane (do three times: Once for each vertex)
- Here, only two planes are shown.



Projection

- How to project a point p to an arbitrary plane (A,B,C,D)?
- We know distance of p from plane is given by $\delta = Ap_x + Bp_y + Cp_z + D$
  - This is a standard property of plane equation, assuming A,B,C is unit vector
- So if we compute $p - \delta n$ units (where n = plane normal), we have projected p to the plane
- Why subtraction? If we are 4 units in front of plane, we need to move 4 units in negative normal direction

Projection

- We can wrap this in a GLSL function:

```
vec3 project( vec3 v, vec3 n, vec3 p ){
    //project p to plane defined by point v and normal n
    vec4 planeEquation = vec4(n, -dot(v,n) );
    float pdistance = dot( vec4(p,1),planeEquation );
    return p - pdistance * n;
}
```

Procedure

- Once we've computed three projected points, take barycentric average of their positions
- This gives final surface location
- Note: Original developers of PT technique recommended taking $\frac{3}{4}/\frac{1}{4}$ mix of new point and old point
- Normal is just the usual interpolated normal (also using barycentric interpolation)

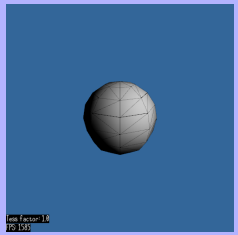
TES

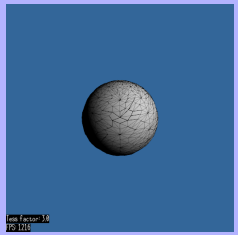
- TES is where the main Phong tessellation work happens

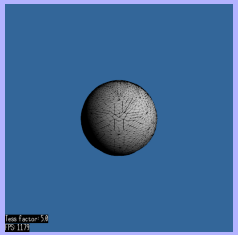
```
vec3 p1 = interpolate(
    gl_TessCoord[0], gl_TessCoord[1], gl_TessCoord[2],
    tcs_position[0], tcs_position[1], tcs_position[2] );

vec3 pr1 = project( tcs_position[0], tcs_normal[0], p1 );
vec3 pr2 = project( tcs_position[1], tcs_normal[1], p1 );
vec3 pr3 = project( tcs_position[2], tcs_normal[2], p1 );
vec3 p1a = interpolate(
    gl_TessCoord[0], gl_TessCoord[1], gl_TessCoord[2],
    pr1,pr2,pr3 );

float alpha=0.75;
p1 = (1.0-alpha)*p1 + alpha * p1a;
...interpolate normal and texcoord as before...
...multiply by world/view/proj matrices...
```

Results

- Sphere with tessFactor=1, 3, 5

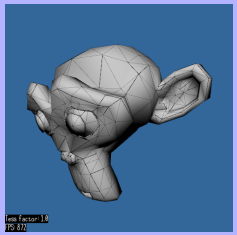


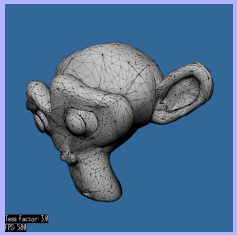


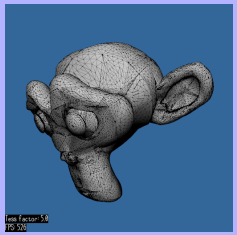


Results

- Using the Suzanne model from Blender, tessFactor=1, 3, 5

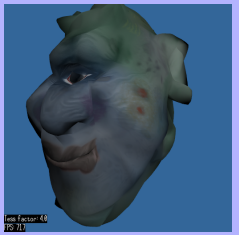






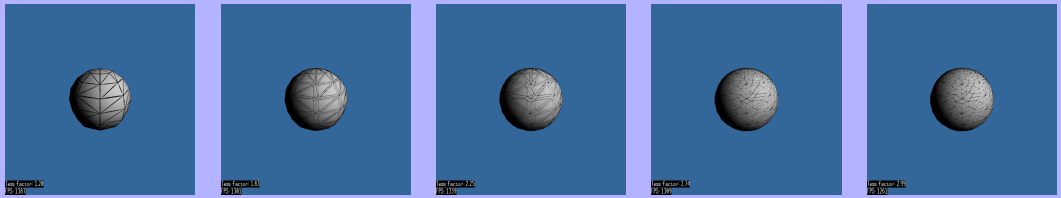
Results

- Using a decimated version of Keenan Crane's Jerry model: tessFactor=1, 2, 3, 4



Fractional Spacing

- We can use fractional spacing here to reduce popping
- Example of fractional tessellations: 1.2, 1.8, 2.25, 2.75, 3.0



Note

- We could apply more subdivision to silhouette edges vs faces viewed head-on
- This can improve appearance of model without imposing too much rendering cost

Sources

- OpenGL Wiki. https://www.opengl.org/wiki/ Tessellation_Control_Shader
- OpenGL Wiki. https://www.opengl.org/wiki/ Tessellation_Evaluation_Shader
- OpenGL Wiki. https://www.opengl.org/wiki/ Tessellation
- Philip Rideout. Triangle Tesselation with OpenGL 4.0. http://prideout.net/blog/?p=48
- Philip Rideout. Quad Tesselation with OpenGL 4.0. http://prideout.net/blog/?p=49
- http://onrendering.blogspot.com/2011/12/tessellation-on-gpu-curved-pn-triangles.html
- http://www.ludicon.com/castano/blog/2009/01/10-fun-things-to-do-with-tessellation/
- Brandon Wang. Smooth GPU Tessellation. https://people.eecs.berkeley.edu/~sequin/CS284/PROJ_12/Brandon/Smooth%20GPU%20Tessellation.pdf
- Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, Sébastien Hillaire. Real-Time Rendering (Fourth Edition). CRC Press.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org