Tessellation Shaders

Review

- We saw vertex and fragment shaders in ETGG 2801/2802
  - Vertex shader: Takes one vertex as input, outputs one vertex
  - Fragment shader: Determines color of a pixel

Review

- We also saw geometry shaders
  - Take 1-6 vertices as input, output arbitrary number of vertices
  - Limited in how much *vertex amplification* is feasible

Idea

- Modern GPU bottleneck: Memory access
  - Often faster to do computations for one frame, use results, throw them away, recompute on next frame
- Tessellation shaders are helpful here

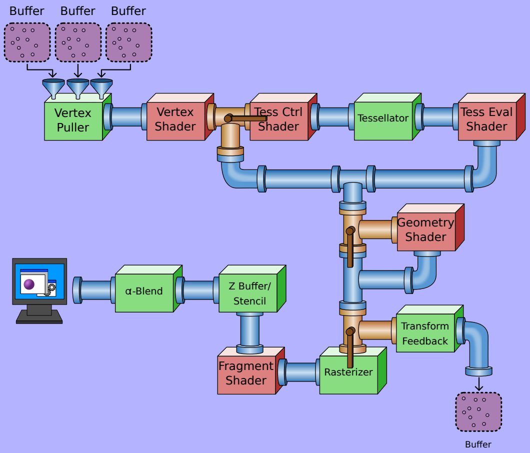
Pipeline

- Recall the GPU pipeline we saw in 2801/2802:

```
p = []
for each triangle T:
    for each vertex v of T:
        run vertex shader on v to get 2D position p[i]
for each triple of 2D points in p:
    compute which pixels are covered by this triangle
    for each pixel
        run fragment shader to compute color
        set pixel to that color
```

Pipeline

- The actual GPU pipeline:



Tessellation Shader

- Conventional rendering: Input = triangle mesh
  - Vertices (positions, texture coordinates, normals)
  - Indices specify which vertices make up triangles
- Ordinarily, vertices are neither created nor destroyed when drawing meshes
  - Ignoring the geometry shader...
- Things are different when using tessellation shader

Input

- Tessellation shader takes *patch* as input
  - Patch = Some (constant) number of vertices
  - Anywhere from 1…32 vertices
- Example:
  glPatchParameteri(GL_PATCH_VERTICES, numVerts);
  glDrawElements(GL_PATCHES, count, GL_UNSIGNED_INT, 0);

Tessellation

- GPU doesn't attach any geometric meaning to a patch
  - Just a blob of vertices
- GPU doesn't consider them "connected" or part of a mesh or anything else
- Shaders must tell where things end up

VS

- When using tessellation, VS usually has little to do
  - Just takes inputs (from buffers) and sends down the pipeline

TCS

- Tessellation control shader: Runs anywhere from 1...32 times per patch
- Can see all vertices of the patch at once
- Produces one output per invocation

Tessellator

- Tessellator is fixed-function
- Produces a bunch of vertices
- Connects together in predetermined pattern to form faces
- Feeds these to next stage (tessellation evaluation shader)

TES

- Tessellation evaluation shader: Runs once for each vertex produced by the tessellator
- Can read all the outputs from the TCS invocations
- Writes value to gl_Position
  - If no geometry shader, this will be screen space location
  - If there is GS, we can decide what meaning gl_Position will have

GS

- Recall: GS is run once for each primitive from earlier in pipeline (typically triangles)
- Can output zero or more triangle strips

FS

- Recall: Fragment shader: Runs once per pixel
- Determines what color pixel should have

Example

- Let's look at a motivating example: Fur/hair generation
- Idea: We have a mesh; we want to create a shaft of hair at each vertex
  - Each shaft made up of a line segment

C++ Code

- Our draw code is pretty simple:

```
void draw(){
    cam->setUniforms();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    Program::setUniform("worldMatrix", mat4::identity() );
    meshprog->use();
    M->draw();
    tessprog->use();
    glPatchParameteri(GL_PATCH_VERTICES,1);
    M->draw(GL_PATCHES);
    fpsText->draw();
}
```

VS

- Vertex shader is very simple: Just passes through input data

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

out vec3 v_position;
out vec3 v_normal;

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

TCS

- New stage: Tessellation control shader

```
layout(vertices=1) out;

in vec3 v_position[];
in vec3 v_normal[];

out vec3 tcs_position[];
out vec3 tcs_normal[];

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

Explanation

- layout line: Says “For each patch, run TCS this many times”
- Inputs (’in’) are from VS output
  - Array because we can see all vertices of the patch
- Outputs will be forwarded to next shader stage
  - It’s an array, but this is a bit of a misnomer
  - TCS must only write to tcs_position[gl_InvocationID] and tcs_normal[gl_InvocationID]
    - Writing to any other slot of an output is an error
- gl_TessLevelOuter: Governs amount of tessellation
  - [0] says how many lines will be generated by tessellator
  - [1] says how many segments there are per line

TES

- Next we have tessellation evaluation shader:

```
layout(isolines) in;
in vec3 tcs_position[];
in vec3 tcs_normal[];
out float tes_shaftPct;

void main(){
    tes_shaftPct = gl_TessCoord[0];
    vec3 p1 = tcs_position[0] + tes_shaftPct * tcs_normal[0];
    vec4 p = vec4(p1,1.0);
    p = p * worldMatrix;
    p = p * viewMatrix;
    p = p * projMatrix;
    gl_Position = p;
}
```

Explanation

- layout(isolines) in
  - Tells tessellator we want it to generate line segments
  - It's a bit confusing: The TCS and TES work in tandem to direct the tessellator
    - TCS tells *how much* tessellation to do
    - TES tells *what format* to use (lines, triangles, quads)
- Inputs
  - Arrays because TES can see entire patch's TCS outputs
- Outputs go to next shader stage (FS in this case)
- gl_TessCoord[0] = How far we are along this line (0...1)
- gl_TessCoord[1] = Which shaft we're doing (0...1)
  - Since we only create one shaft, this would always be 0
  - Yes, this seems reversed compared to the tessellation levels' order...

FS

- FS doesn't show any new concepts:

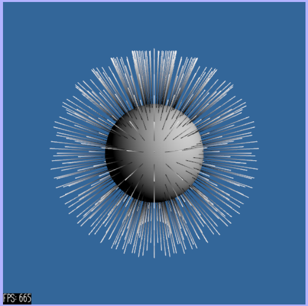
```
in float tes_shaftPct;

out vec4 color;

void main(){
    color.a = 1.0;
    color.rgb = vec3(0) + tes_shaftPct * vec3(1);
}
```

Results

- What we get:



Results

- It looks kind of...artificial
- We'd like the shafts to bend downward like real fur
- Idea: We do something of a particle simulation
- Physics:
  - $s = \int v\, dt$ (where s=distance, v=velocity, t=time)
  - $v = \int a\, dt$ (where a=acceleration)
- Since acceleration (due to gravity) is a constant, $v = at$ and $s = \frac{1}{2}at^2$(ignoring constants)
- We want to use a quadratic curve

TCS

- The new TCS: Here, we do constant amount of subdivision:

```
layout(vertices=1) out;
in vec3 v_position[];
in vec3 v_normal[];
out vec3 tcs_position[];
out vec3 tcs_normal[];
void main(){
    gl_TessLevelOuter[0] = 1;
    gl_TessLevelOuter[1] = 8;  //<----- new
    tcs_position[gl_InvocationID] = v_position[gl_InvocationID];
    tcs_normal[gl_InvocationID] = v_normal[gl_InvocationID];
}
```

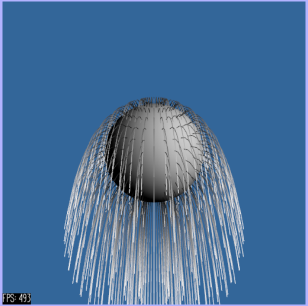
TES

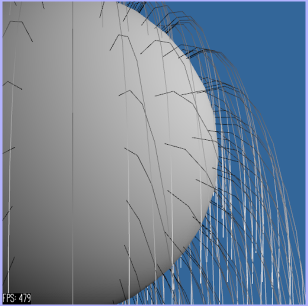
- We also alter the TES: The constants are chosen from experimentation to give desired appearance

```
layout(isolines) in;
in vec3 tcs_position[];
in vec3 tcs_normal[];
out float tes_shaftPct;
void main(){
    tes_shaftPct = gl_TessCoord[0];
    float t = (tes_shaftPct * 20);
    vec3 p1 = tcs_position[0] +
              tes_shaftPct * tcs_normal[0] +
              vec3(0,-0.005,0)*t*t;
    vec4 p = vec4(p1,1.0);
    p = p * worldMatrix; p = p * viewMatrix; p = p * projMatrix;
    gl_Position = p;
}
```

Results

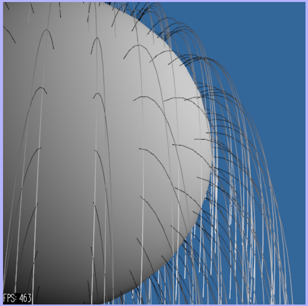
- Gives better appearance than before...But closeups don't look so good





Results

- If we do more subdivision, hair shafts are smoother (this uses 16 instead of 8)



Idea

- In practice, we'd measure distance from viewer and subdivide more if object is closer to camera
- But: Our hair shafts are still pretty far apart
- Problem: We are requiring each shaft to emanate from a vertex
- We could get more shafts by adding more vertices
  - But that's a bad idea (slows rendering)

Procedure

- First, we'll use entire triangle instead of just one point as input to TCS
- In draw(): glPatchParameteri(GL_PATCH_VERTICES,3);
- Now, TCS will run once per triangle
- This would make results sparser

TCS

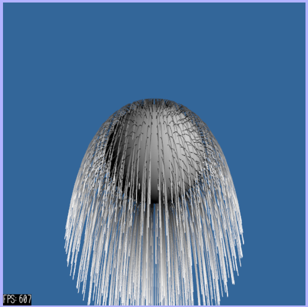
- Suppose we change the TCS like so:

```
layout(vertices=1) out;
in vec3 v_position[];
in vec3 v_normal[];
out vec3 tcs_position[];
out vec3 tcs_normal[];
void main(){
    gl_TessLevelOuter[0] = 1;
    gl_TessLevelOuter[1] = 8;
    tcs_position[gl_InvocationID] = (v_position[0]+v_position[1]+
        v_position[2])/3.0;
    tcs_normal[gl_InvocationID] = (v_normal[0]+v_normal[1]+
        v_normal[2])/3.0;
}
```

- Now, each triangle gets one shaft at its centroid

Results

- Screenshot



Multiple Hairs

- How to create more hairs?
- Want number of hairs to be proportional to triangle area
  - Can get this via cross product
- Once we have number of hairs, how can we distribute randomly?

Barycentric

- Recall: Barycentric coordinates: For any u,v,w The point v[0]*u + v[1]*v + v[2]*w will be within the triangle if:
  - All three u,v,w are between 0 and 1
  - u+v+w = 1
- u, v, w are barycentric coordinates of triangle

Randomness

- Next issue: How can we obtain random numbers on GPU?
- On CPU, we use rand()
- How does rand() work?

LCG

- Most systems use *linear congruential generator* for rand()
  - Input: A seed
  - Compute: seed = (a * seed) mod b
  - Return seed as random number
- Notice: Generator continually updates seed

Problem

- What do we choose for a and b?
  - Lots of mathematical theory
  - Recommended values: a=0x7fffffff, b=16807
- Problem: Computing a*seed may overflow
  - We can address this by some careful coding
  - But: Can we avoid the problem entirely?

Xorshift

- Another RNG that does not have this problem is Xorshift
- Basic structure: We have some global variable named "state"
  - state = state ^ (state << 13);
  - state = state ^ (state >> 17);
  - state = state ^ (state << 15);
- Often we want a value between 0...1, so we convert this uint to a float:
  randomValue = float(state & 0x7fffffff) / float(0x7fffffff);

Code

- We can package this into a function:

```
uint state;
float xorshift(){
    state = state ^ (state << 13);
    state = state ^ (state >> 17);
    state = state ^ (state << 15);
    return float(state & 0x7fffffff) / float(0x7fffffff);
}
```

- Our only problem...How to initialize state?

Initialize

- Shader invocations don't share global variables
- So we need to initialize state to different value in each shader
- We have a primitive number (patch ID)
- And we know our shaft number:
uint shaftNumber = uint(gl_TessCoord[1] * gl_TessLevelOuter[0]);
  - gl_TessCoord[1] is value in range [0,1): 0 for first shaft
  - gl_TessLevelOuter[0] tells how many shafts we have total
  - So we multiply to get integer value 0,1,2,...

Initialization

- Now we can write an initialization function: Stir primitive ID and shaft number together

```
void xorshiftInit(uint v){
    state = gl_PrimitiveID ^ (v<<16);
}
```

- Call it as:
  xorshiftInit(shaftNumber);

Putting it Together

- C code doesn't change:

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

Putting it Together

- TCS:

```
layout(vertices=3) out;
in vec3 v_position[];
in vec3 v_normal[];
out vec3 tcs_position[];
out vec3 tcs_normal[];
void main(){
    gl_TessLevelOuter[0] = 16; //16 shafts per triangle
    gl_TessLevelOuter[1] = 8;  //8 segments per shaft
    tcs_position[gl_InvocationID] = v_position[gl_InvocationID];
    tcs_normal[gl_InvocationID] = v_normal[gl_InvocationID];
}
```

Note

- We should scale number of shafts per triangle based on triangle area...
  - We'll overlook that for now
- Notice TCS runs 3 times per patch
- We've set patch size to 3 (C code)

Putting it Together

- TES:

```
layout(isolines) in;
in vec3 tcs_position[];
in vec3 tcs_normal[];
out float tes_shaftPct;

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

TES

- What does TES need to do?
- First: Initialize RNG

```
tes_shaftPct = gl_TessCoord[0];
uint shaftNumber = uint(
    gl_TessCoord[1] * gl_TessLevelOuter[0] );
xorshiftInit(shaftNumber);
```

TES

- Next, generate random barycentric coordinates

```
float u = xorshift();
float v = xorshift();
if( u+v > 1.0 ){
    u = 1-u;
    v = 1-v;
}
float w = 1.0-u-v;
```

- Now, u+v+w=1 and all of u,v,w are between 0 and 1

TES

- We can write a small barycentric interpolation routine:

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

- If this is given triangle vertices or normals, it does the required interpolation

TES

- Compute position and normal where the hair shaft is:

```
vec3 basePos = interpolate(
    u,v,w,
    tcs_position[0],tcs_position[1],tcs_position[2]);

vec3 N = interpolate(
    u,v,w,
    tcs_normal[0],tcs_normal[1],tcs_normal[2]);
```

TES

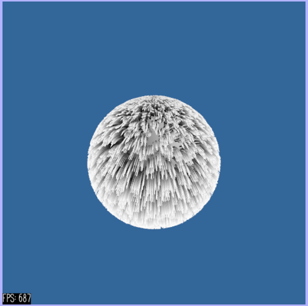
- Last thing: Compute output for rasterizer:

```
float t = (tes_shaftPct * 20);
vec3 p1 = basePos + tes_shaftPct * 0.35 * N + vec3(0,-0.0005,0)*t
    *t;
vec4 p = vec4(p1,1.0);
p = p * worldMatrix;
p = p * viewMatrix;
p = p * projMatrix;
gl_Position = p;
```

- The 0.35 governs how long the shaft is
- The -0.0005 governs the amount of vertical bend to the shaft

Result

- Our result:



- Hmmm... It looks kind of...patchy

Problem

- Random number generators (LCG's or Xorshift) are good for sequences of numbers
- But if the seeds are not random, they tend to produce correlated sequences
- One solution: We can hash the input seed for the generator

Hash

- One good hash: A Wang hash (developed by Thomas Wang)

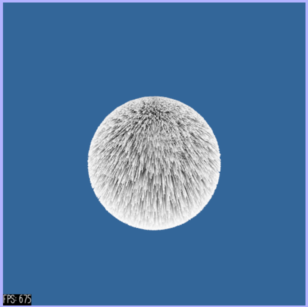
```
uint hash(uint v){
    v = 9*( (v >> 16 ) ^ (v ^ 61));
    v ^= (v>>4);
    v = v * 668265261;
    return v ^ (v>>15);
}
```

- Use it:

```
void xorshiftInit(uint v){
    state = gl_PrimitiveID ^ (v<<16);
    state = hash(state)
}
```

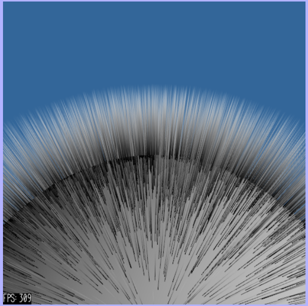
Results

- Much better



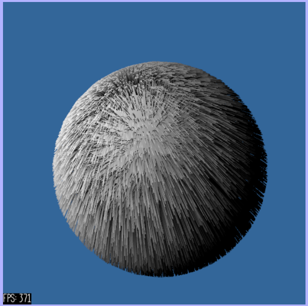
Note

- If we are simulating scene where we mostly look at shafts edge-on (ex: grass), we can use transparency to help appearance
  - When drawing: glDepthMask(0) before rendering hair; glDepthMask(1) afterwards
  - In FS: out.a = 1.0 - tes_shaftPos



Final Render

- With lighting



Sources

- https://www.khronos.org/opengl/wiki/Tessellation
- https://www.khronos.org/opengl/wiki/ Tessellation_Control_Shader
- https://www.khronos.org/opengl/wiki/ Tessellation#Tessellation_primitive_generation
- Kostas Anagnostou. Rendering Fur Using Tessellation. https://interplayoflight.wordpress.com/2014/12/31/rendering-fur-using-tessellation/
- Nathan Reed. Quick and Easy GPU Random Numbers in D3D11. http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/
- Integer Hashing. http://www.burtleburtle.net/bob/hash/integer.html
- Adam Swaab. Random Point in a Triangle - Barycentric Coordinates. https://adamswaab.wordpress.com/2009/12/11/random-point-in-a-triangle-barycentric-coordinates/
- Random Numbers. http://www.eternallyconfuzzled.com/tuts/algorithms/jsw_tut_rand.aspx
- George Marsaglia. Xorshift RNGs. http://www.jstatsoft.org/v08/i14/paper
- Humus. Phone-Wire AA. http://www.humus.name/index.php?page=3D&ID=89

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