Shaders
Overview

- GPU
- Shaders
  - Types of Shaders
- Code
- Projects
Why is the GPU important? Mad Computation Capabilities, Brah.

The compute capabilities of GPUs has exploded.

GPUs are great if it executes the same code on different input, needs lots of math, does not share data, has lots of work to do without CPU intervention.
SHIT JUST GOT REAL

https://www.youtube.com/watch?v=-P28LKW7HrI
Shaders Make Shit Look Good
(and more)
The GPU and the shaders are what allows us to add a high level of detail.

OLD METHOD: Vertex and fragment operations follow the linearity of it’s code (Project 1). It is hard coded for the GPU -- no flexibility. Need to provide geometry and attributes in the CPU.

NEW METHOD: No longer hard coded, pieces of your code lie within the GPU and all you need to provide is the geometry from the CPU.
Types of Shaders

Vertex Shader: The purpose is to transform each vertex's 3D position in world space to the 2D coordinate at which it appears on the screen. Can now change location and move vertices around. Cannot add vertices.

Fragment Shader: Takes the data from the vertices, but can also add its own data. Bumps, color, shading, gloss, etc. Can add much more detail on a per-pixel level as compared to methods using interpolation alone (old method).

Geometry Shader: Much like vertex shader, but you can add vertices.

Tessellation Shader: Allows simple meshes to be subdivided into more complex meshes. Can also refine very complex meshes.
A Trip Down the Pipeline

- CPU sends geometry to GPU
- Vertex Shader: geometry is transformed
  - Geometry shader
  - Tessellation Shader
- Geometry is transformed into triangles
- Fragment shader: pixels are transformed
- Render the image
Amped Up Sophistication

● OLD METHOD:
  ○ Transform vertices with matrices (ex: view matrices -- world to device space in 1F)
  ○ Use Phong/flat Shading

● NEW METHOD:
  ○ Custom vertex transformation
  ○ Custom lighting models
  ○ More Complicated Visual Effects:
    ■ Shadows
    ■ Fog
    ■ Reflection and Refractions
    ■ Bumps
    ■ Movement
Vertex Shader

Replaces the fixed functionality of the vertex processor in the graphics pipeline.

Transform vertices, normals and texture coordinates.

Calculate lighting per vertex.

Set values to be used for interpolation in the fragment shader.

Knows nothing of view, primitives, frustum.
Fragment Shader

Replaces the fixed functionality of the fragment processor in the graphics pipeline.

Set color, get color from textures.

Make fog, bumps and other color calculations.

Use any interpolated data from the vertices.

Cannot: change coordinates, write into textures, or affect depth.
Geometry Shader

Governs the processing of primitives.

Is between the Vertex and Tessellation shaders in the pipeline.
(optional shader)

Creates new geometry.

Good example, fur
Tessellation Shader

After geometry shader in the vertex processing stage of the pipeline.

Involves subdividing an area between vertices by adding vertices and computes the values at these vertices.
How do we do any of this?

Shading languages:

- **GLSL**
  - For OpenGL, similar to C
- **Cg**
  - For OpenGL and Directx
- **HLSL**
  - For Microsoft Direct3D
- **RSL**
  - Like C, Pixar

But these are all a pain so we aren’t going to use them on the projects.
**Types [4.1]**

A shader can aggregate these using arrays and structures to build more complex types. There are no pointer types.

<table>
<thead>
<tr>
<th>Basic Types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>no function return value or empty parameter list</td>
</tr>
<tr>
<td>bool</td>
<td>Boolean</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
</tr>
<tr>
<td>float</td>
<td>floating scalar</td>
</tr>
<tr>
<td>vec2, vec3, vec4</td>
<td>n-component floating point vector</td>
</tr>
<tr>
<td>bvec2, bvec3, bvec4</td>
<td>Boolean vector</td>
</tr>
<tr>
<td>ivec2, ivec3, ivec4</td>
<td>signed integer vector</td>
</tr>
<tr>
<td>mat2, mat3, mat4</td>
<td>2x2, 3x3, 4x4 float matrix</td>
</tr>
<tr>
<td>sampler2D</td>
<td>access a 2D texture</td>
</tr>
<tr>
<td>samplerCube</td>
<td>access cube mapped texture</td>
</tr>
</tbody>
</table>

**Vector Components [5.5]**

In addition to array subscript syntax, names of vector components are denoted by a single letter. Components can be swizzled and replicated, e.g., pos.xx, pos.zy.

- `{x, y, z, w}` Use when accessing vectors that represent points or normals
- `{r, g, b, a}` Use when accessing vectors that represent colors
- `{s, t, p, q}` Use when accessing vectors that represent texture coordinates

**Fragment Shader Special Variables [7.2]**

Fragment shaders may write to `gl_FragColor` or to one or more elements of `gl_FragData[]`, but not both. The size of the `gl_FragData` array is given by the built-in constant `gl_MaxDrawBuffers`.

**Inputs:**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Units or coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>mediump vec4</td>
<td><code>gl_FragCoord;</code> fragment position within frame buffer</td>
<td>window coordinates</td>
</tr>
<tr>
<td>bool</td>
<td><code>gl_FrontFacing;</code> fragment belongs to a front-facing primitive</td>
<td>Boolean</td>
</tr>
<tr>
<td>mediump vec2</td>
<td><code>gl_PointCoord;</code> fragment position within a point (point rasterization only)</td>
<td>0.0 to 1.0 for each component</td>
</tr>
</tbody>
</table>

**Outputs:**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Units or coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>mediump vec4</td>
<td><code>gl_FragColor;</code> fragment color</td>
<td>RGBA color</td>
</tr>
<tr>
<td>mediump vec4</td>
<td><code>gl_FragData[n];</code> fragment color for color attachment n</td>
<td>RGBA color</td>
</tr>
</tbody>
</table>

**Vertex Shader Special Variables [7.1]**

**Outputs:**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Units or coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>highp vec4</td>
<td><code>gl_Position;</code> transformed vertex position</td>
<td></td>
</tr>
<tr>
<td>mediump float</td>
<td><code>gl_PointSize;</code> transformed point size (point rasterization only)</td>
<td></td>
</tr>
</tbody>
</table>
Example using VTK and Shaders

```c++
const char* frag = "void propFuncFS(void){gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);}");
vtkSmartPointer<vtkShader2> fragment=vtkShader2::New();
fragment->SetType(VTK_SHADER_TYPE_FRAGMENT);
fragment->SetSourceCode(frag);
fragment->SetContext(pgm->GetContext());
pgm->GetShaders()->AddItem(fragment);

((vtkOpenGLProperty*)win1Actor->GetProperty())->SetPropProgram(pgm);
win1Actor->GetProperty()->ShadingOn();
```
Syntax is similar to what we’ve seen.

```c
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    fragColor = vec4(.5, .6, 0.0, 1.0);
}
```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 xy = fragCoord.xy;  // We obtain our coordinates for the current pixel
    xy.x = xy.x / iResolution.x;  // We divide the coordinates by the screen size
    xy.y = xy.y / iResolution.y;
    // Now x is 0 for the leftmost pixel, and 1 for the rightmost pixel
    vec4 color = vec4(0, 0, 0, 0, 0, 1.0);  // This is actually black right now
    color.r = xy.x;  // Set its red component to the normalized x value..access as rgb
    color.z = xy.x;  // Set its green component to the normalized x value..access as xyz
    
    fragColor = color;
}
Project 1 Part 1

You’ll be provided the geometry and you need to provide a shader that mimics waves.

Hint hint: periodic functions are great for this.
Project 1 Part 2

Given a geometry, apply a fragment shader. The fragment shader will read in the given color and apply a combination of flat shading and color gradient reduction.

Color gradient reduction is taking similar colors and applying assigning a single color.
Project 2

Ray march a sphere and apply diffuse shading.

- Render a circle
- Calculate normals needed for shading via ray marching.
Ray Marching: SDF

Uses Signed Distance Functions. Consider each point, find the shortest distance between that point and the surface.

- If 0, on shape
- If < 0, inside shape
- If > 0, outside shape
  - SDF = f(v) = dist(v) - r = distance from the point to sphere
    - f is the cartesian coordinates for a sphere
Ray Marching

Ray tracing is used when you have a geometry (ex: triangles) and you use intersection equations. (Where does it intersect my triangle, if at all? What about this one? Is this one in front of this one?)

In Ray marching the scene is defined by on the SDF. For each ray, we slowly increment it asking if we’re on, in, or not in our surface.

We increment based on the distance to the surface.
Ray Marching Alg

```cpp
Float distanceToSurface(vec3 rayOrigin, vec3 rayDirection, float start, float end){
    float depth = start;
    for (int i = 0; i < MAX_MARCHING_STEPS; i++) {
        float dist = SDF(rayOrigin + depth * rayDirection);
        if (dist < EPSILON) {
            // We're inside the scene surface!
            return depth;
        }
    }
    // Move along the view ray
    depth += dist;
    if (depth >= end) {
        // Gone too far; give up
        return end;
    }
    return end;
}
```
Normals Needed for Shading

Our normal vector for each point is the line projecting from the surface. This is the gradient of our distance function at that point.