The Graphics Pipeline and OpenGL III: OpenGL Shading Language (GLSL 1.10)

Gordon Wetzstein
Stanford University
EE 267 Virtual Reality
Lecture 4
stanford.edu/class/ee267/
 Updates

• for 24h lab access: please email Steven Clark (with some arguments that convince him that you read the lab instructions, e.g. screenshot of last slide)

• lab computers can also be used remotely! (instructions on piazza & website to follow)

• waitlist: looks like everyone who is still on the waitlist will get in

• HW1 due Thursday at midnight!
Lecture Overview

• Review of graphics pipeline
• vertex and fragment shaders
• OpenGL Shading Language (GLSL 1.10)
• Implementing lighting & shading with GLSL vertex and fragment shaders
Reminder: The Graphics Pipeline

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
two goals:
1. determine which fragments are inside the primitives (triangles) and which ones aren’t
2. interpolate per-vertex attributes (color, texture coordinates, normals, …) to each fragment in the primitive

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Reminder: The Graphics Pipeline

vertex shader
- transforms
- (per-vertex) lighting
- …

fragment shader
- texturing
- (per-fragment) lighting
- …
Vertex Shaders

- **Input**
  - vertex position, normal, color, material, texture coordinates
  - modelview matrix, projection matrix, normal matrix
  - …

- **Vertex Shader** (executed for each vertex)

```c
void main ()
{
    // do something here
    ...
}
```

- **Output**
  - transformed vertex position (in clip coords), texture coordinates
  - …

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Fragment Shaders

input
- vertex position in window coords, texture coordinates
- ...

fragment shader (executed for each fragment)

void main ()
{
    // do something here
    ...
}

output
- fragment color
- fragment depth
- ...

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Why Do We Need Shaders?

- massively parallel computing
- single instruction multiple data (SIMD) paradigm → GPUs are designed to be parallel processors
- vertex shaders are independently executed for each vertex on GPU (in parallel)
- fragment shaders are independently executed for each fragment on GPU (in parallel)
Why Do We Need Shaders?

- most important: vertex transforms and lighting & shading calculations
- shading: how to compute color of each fragment (e.g. interpolate colors)
  1. Flat shading
  2. Gouraud shading (per-vertex shading)
  3. Phong shading (per-fragment shading)
- other: render motion blur, depth of field, physical simulation, …

courtesy: Intergraph Computer Systems
Shading Languages

- Cg (C for Graphics – NVIDIA, deprecated)
- GLSL (GL Shading Language – OpenGL)
- HLSL (High Level Shading Language - MS Direct3D)
Demo – Simple Vertex Shader

```cpp
// variable passed in from application
uniform float deformation = 1.0;

void main () // vertex shader
{
    // deform vertex position
    vec3 pos = gl_Vertex.xyz + deformation * gl_Normal;

    // convert to clip space
    gl_Position = gl_ModelViewProjectionMatrix * vec4(pos,1.0);

    // do lighting calculations here (in world space)
    ...  
}
```
// variables passed in from application
uniform sampler2D texture;
uniform float gamma = 1.0;

void main () // fragment shader
{
    // texture lookup
    vec3 textureColor = texture2D(texture,
    gl_TexCoord[0].xy).rgb;

    // set output color by applying gamma
    gl_FragColor.rgb = pow(textureColor, gamma);
}
// variable to be passed from vertex to fragment shader
varying vec4 myColor;

void main () // vertex shader – Gouraud shading
{
    // transform position to clip space
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // transform position to world space
    vec4 positionWorld = gl_ModelViewMatrix * gl_Vertex;

    // transform normal into world space
    vec3 normalWorld = gl_NormalMatrix * gl_Normal;

    // do lighting calculations here (in world space)
    ...
    myColor = ...
}

// variable to be passed from vertex to fragment shader
varying vec4 myColor;

void main () // fragment shader – Gouraud shading
{
    gl_FragColor = myColor;
}
Demo – Vertex & Fragment Shader

// variable to be passed from vertex to fragment shader
varying vec4 myPos;
varying vec3 myNormal;

void main () // vertex shader – Phong shading
{
  // transform position to clip space
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // transform position to world space
  vec4 myPos = gl_ModelViewMatrix * gl_Vertex;

  // transform normal into world space
  vec3 myNormal = gl_NormalMatrix * gl_Normal;
}

// variable to be passed from vertex to fragment shader
varying vec4 myPos;
varying vec3 myNormal;

void main () // fragment shader – Phong shading
{
  // do lighting calculations here
  ...
  gl_FragColor = ...;
}
Demo – General Purpose Computation Shader

// variables passed in from application
uniform sampler2D tex;
uniform float timestep = 1.0;

void main () // fragment shader
{
    vec2 texcoord = gl_TexCoord[0].xy;

    // texture lookups
    float u = texture2D(tex,texcoord).r;

    float u_xp1 = texture2D(tex,
                         float2(texcoord.x+1,texcoord.y)).r;
    float u_xm1 = texture2D(tex,
                         float2(texcoord.x-1,texcoord.y)).r;
    float u_yp1 = texture2D(tex,
                         float2(texcoord.x,texcoord.y+1)).r;
    float u_ym1 = texture2D(tex,
                         float2(texcoord.x,texcoord.y-1)).r;

    glFragColor.r = u +
    timestep*(u_xp1+u_xm1+u_yp1+u_ym1-4*u);
}

heat equation: \( \frac{\partial u}{\partial t} = \alpha \nabla^2 u \quad \Rightarrow \quad u^{(i+1)} = \Delta_t \alpha \nabla^2 u + u^{(i)} \)
OpenGL Shading Language (GLSL)

• high-level programming language for shaders

• syntax similar to C (i.e. has `main` function and many other similarities)

• usually very short programs that are executed in parallel on GPU

• good introduction / tutorial:

  https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/
versions of OpenGL, WebGL, GLSL can get confusing

here’s what we use:

- WebGL 1.0 - based on OpenGL ES 2.0
- GLSL 1.10 - shader preprocessor: `#version 110`

reason: three.js doesn’t support WebGL 2.0 yet
void main () // vertex shader
{
    // transform position to clip space
    // this is similar to gl_Position = ftransform();
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
void main () // fragment shader
{
    // set same color for each fragment
    gl_FragColor = vec4(1.0,0.0,0.0,1.0);
}

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>boolean (true or false)</td>
</tr>
<tr>
<td>int</td>
<td>signed integer</td>
</tr>
<tr>
<td>float</td>
<td>32 bit floating point</td>
</tr>
<tr>
<td>ivec2, ivec3, ivec4</td>
<td>integer vector with 2, 3, or 4 elements</td>
</tr>
<tr>
<td>vec2, vec3, vec4</td>
<td>floating point vector with 2, 3, or 4 elements</td>
</tr>
<tr>
<td>mat2, mat3, mat4</td>
<td>floating point matrix with 2x2, 3x3, or 4x4 elements</td>
</tr>
<tr>
<td>sampler2D</td>
<td>handle to a 2D texture</td>
</tr>
</tbody>
</table>
GLSL Data Types

**uniform type** – read-only values passed in from CPU application, e.g. `uniform float` or `uniform sampler2D`

vertex shader

```c
void main ()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

fragment shader

```c
uniform sampler2D texture;
void main ()
{
    gl_FragColor = texture2D(texture, gl_TexCoords[0].xy);
}
```
GLSL Data Types

**varying type** – variables that are passed from vertex to fragment shader (i.e. write-only in vertex shader, read-only in fragment shader)

```
vertex shader

varying float myValue;

void main ()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    myValue = 3.14159 / 10.0;
    myValue = 3.14159 / 10.0;
}

fragment shader

varying float myValue;

void main ()
{
    gl_FragColor = vec4(myValue, myValue, myValue, 1.0);
    gl_FragColor = vec4(myValue, myValue, myValue, 1.0);
}
```
<table>
<thead>
<tr>
<th>GLSL – Standard Attributes in <strong>Vertex Shader</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>built-in attributes</strong></td>
</tr>
<tr>
<td>vec4 gl_Vertex</td>
</tr>
<tr>
<td>vec3 gl_Normal</td>
</tr>
<tr>
<td>vec4 gl_Color</td>
</tr>
<tr>
<td>vec4 gl_MultiTexCoordX</td>
</tr>
<tr>
<td><strong>built-in uniforms</strong></td>
</tr>
<tr>
<td>mat4 gl_ModelViewMatrix</td>
</tr>
<tr>
<td>mat4 gl_ModelViewProjectionMatrix</td>
</tr>
<tr>
<td>mat3 gl_NormalMatrix</td>
</tr>
<tr>
<td><strong>built-in varying</strong></td>
</tr>
<tr>
<td>vec4 gl_Position</td>
</tr>
<tr>
<td>vec4 gl_FrontColor</td>
</tr>
<tr>
<td>vec4 gl_TexCoord[X]</td>
</tr>
</tbody>
</table>
Disclaimer

• modelview and projection matrices can be used via either the built-in uniforms OR as regular, user-defined built-in variables

• in the lab & homework, we will not use the built-in variables but pass in the matrices manually
GLSL – Standard Attributes in **Fragment Shader**

**built-in (varying)** input

- `vec4 gl_FragCoord` (x,y,z,1/w\text{clip}) in window space
- `vec4 gl_TexCoord[X]` interpolated texture coordinates
- `vec4 gl_Color` interpolated color from `gl_FrontColor`

**built-in output**

- `vec4 gl_FragColor` fragment color
- `float gl_FragDepth` value written to depth buffer, if not specified: `gl_FragCoord.z`
GLSL Shader

all of these values will be interpolated over the primitive by rasterizer
GLSL – built-in functions

dot

dot product between two vectors

cross
cross product between two vectors

texture2D
texture lookup (get color value of texture at some tex coords)

normalize
normalize a vector

clamp
clamp a scalar to some range (e.g., 0 to 1)

radians, degrees, sin, cos, tan, asin, acos, atan, pow, exp, log,
exp2, log2, sqrt, abs, sign, floor, ceil, mod, min, max, length, ...

good summary of OpenGL ES (WebGL) shader functions:

http://www.shaderific.com/glsl-functions/
Gouraud Shading with GLSL (only diffuse part)

uniform vec3 lightPositionWorld;
uniform vec3 lightColor;
uniform vec3 diffuseMaterial;

void main () // vertex shader
{
    // transform position to clip space
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // transform vertex position, normal, and light position to view space
    vec3 P = ...
    vec3 L = ...
    vec3 N = ...

    // compute the diffuse term here
    float diffuseFactor = ...

    // set output color
    gl_FrontColor.rgb = diffuseFactor * diffuseMaterial * lightColor;
}
void main () // fragment shader
{
    // set output color
    gl_FragColor = gl_Color;
}
varying vec3 vPosition;
varying vec3 vNormal;

void main () // vertex shader
{
    // transform position to clip space
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // transform vertex position, normal, and light position to view space
    vec3 P = ...
    vec3 N = ...

    // set output texture coordinate to vertex position in world coords
    vPosition = P;

    // set output color to vertex normal direction
    vNormal = N;
}
uniform vec3 lightColor;
uniform vec3 diffuseMaterial;
uniform vec3 lightPositionWorld;

varying vec3 vPosition;
varying vec3 vNormal;

void main () // fragment shader
{
    // incoming color is interpolated by rasterizer over primitives!
    vec3 N = vNormal;

    // vector pointing to light source
    vec3 L = …

    // compute the diffuse term
    float diffuseFactor …

    // set output color
    gl_FragColor.rgb = diffuseFactor * diffuseMaterial * lightColor;
}
GLSL - Misc

- swizzling:
  
  ```glsl
  vec4 myVector1;
  vec4 myVector2;
  vec3 myVector1.xxy + myVector2.zxy;
  ```

- matrices are column-major ordering

- initialize vectors in any of the following ways:
  
  ```glsl
  vec4 myVector    = vec4(1.0, 2.0, 3.0, 4.0);
  vec4 myVector2   = vec4(vec2(1.0, 2.0), 3.0, 4.0);
  vec4 myVector3   = vec4(vec3(1.0, 2.0, 3.0), 4.0);
  ```

- sometimes OpenGL quantizes `gl_FrontColor` (vertex shader) to `gl_Color` (fragment shader) to 8 bits per channel, despite being a float → may need to use `gl_TexCoord[X]` instead to preserve precision

- these are equivalent:    `myVector.xyzw = myVector.rgba = myVector.uvst`

- we omitted a lot of details…
goals:
• loading, compiling, and linking GLSL shaders (from a file) using JavaScript
• activating and deactivate GLSL shaders in JavaScript
• accessing uniforms from JavaScript

our approach (for labs and homeworks):
• use three.js to handle all of the above
• can do manually, but more work – we will shield this from you
Summary

• GLSL is your language for writing vertex and fragment shaders

• each shader is independently executed for each vertex/fragment on the GPU

• usually require both vertex and fragment shader, but can “pass-through” data
Further Reading

- GLSL tutorial: https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/

- summary of built-in GLSL functions: http://www.shaderific.com/glsl-functions/