Week 03: Shaders in WebGL

CS-537: Interactive Computer Graphics
Dr. Chloe LeGendre
Department of Computer Science

Some materials from the companion slides of Angel and Shreiner, “Interactive Computer Graphics, A Top-Down Approach with WebGL.”

Objectives

- Describe Simple Shaders
  - Vertex Shader
  - Fragment Shader
- Describe programming shaders with GLSL
- Finish discussing the “Hello World” WebGL programs
- Describe how to couple shaders to applications and pass data from application to shaders

What to compute in a Vertex Shader

- Anything you want to compute for every vertex
- The movement / transformation of vertices
  - Morphing the vertices in some way
    - translation, rotation, scaling, shearing
  - Projection from 3D coordinates to 2D image coordinates
  - These will be covered in detail in Weeks 05 and 06
- Lighting or shading calculations (sometimes)
  - Some shading models – how a light source interacts with the object
  - This will be covered in detail in Week 08 of this course

What to compute in a Fragment Shader

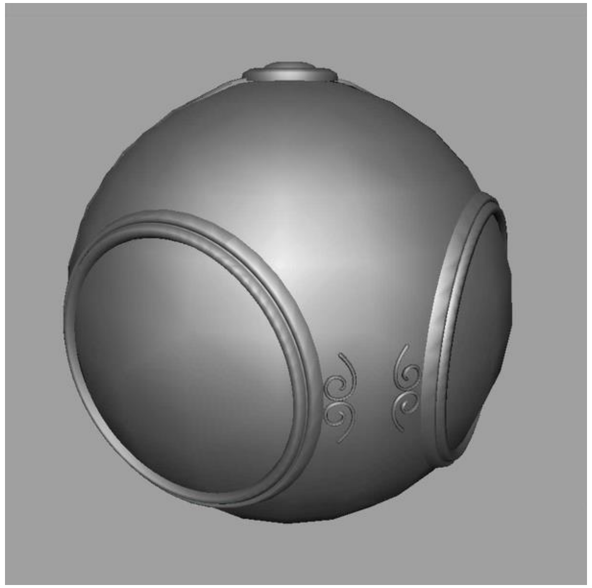
- More accurate lighting or shading calculations
  - This will be covered in detail in Week 08 of this course

per vertex lighting

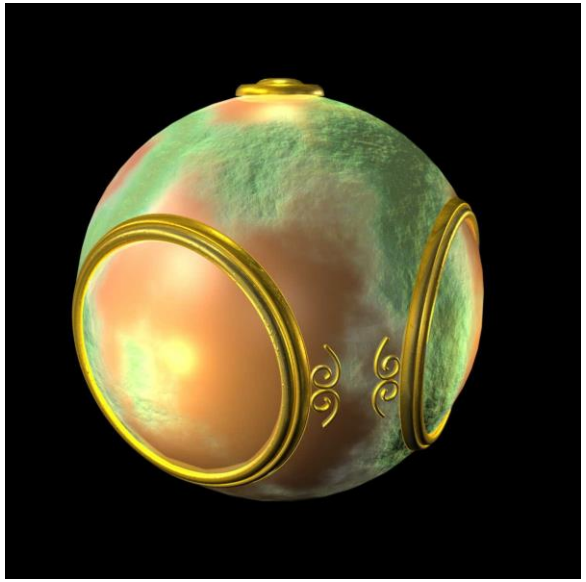
per fragment lighting

What to compute in a Fragment Shader (II)

- Application of texture to geometry
  - This will be covered in detail in Week 09 of this course


smooth shading


environment mapping


bump mapping

Writing Shaders

- WebGL uses the GLSL language for shaders
- GLSL:
  - High level, C-like language
  - New data types: matrices, vectors, samplers (used for textures)
- Every WebGL program that you write needs to have shaders provided, one each of vertex and fragment type (a pair)
- You can write programs that use more than one shader (Assignment 3 will cover this) to apply different effects to different objects.

A Simple Vertex Shader (in GLSL)

Attribute keyword tells you: this is input to shader from application

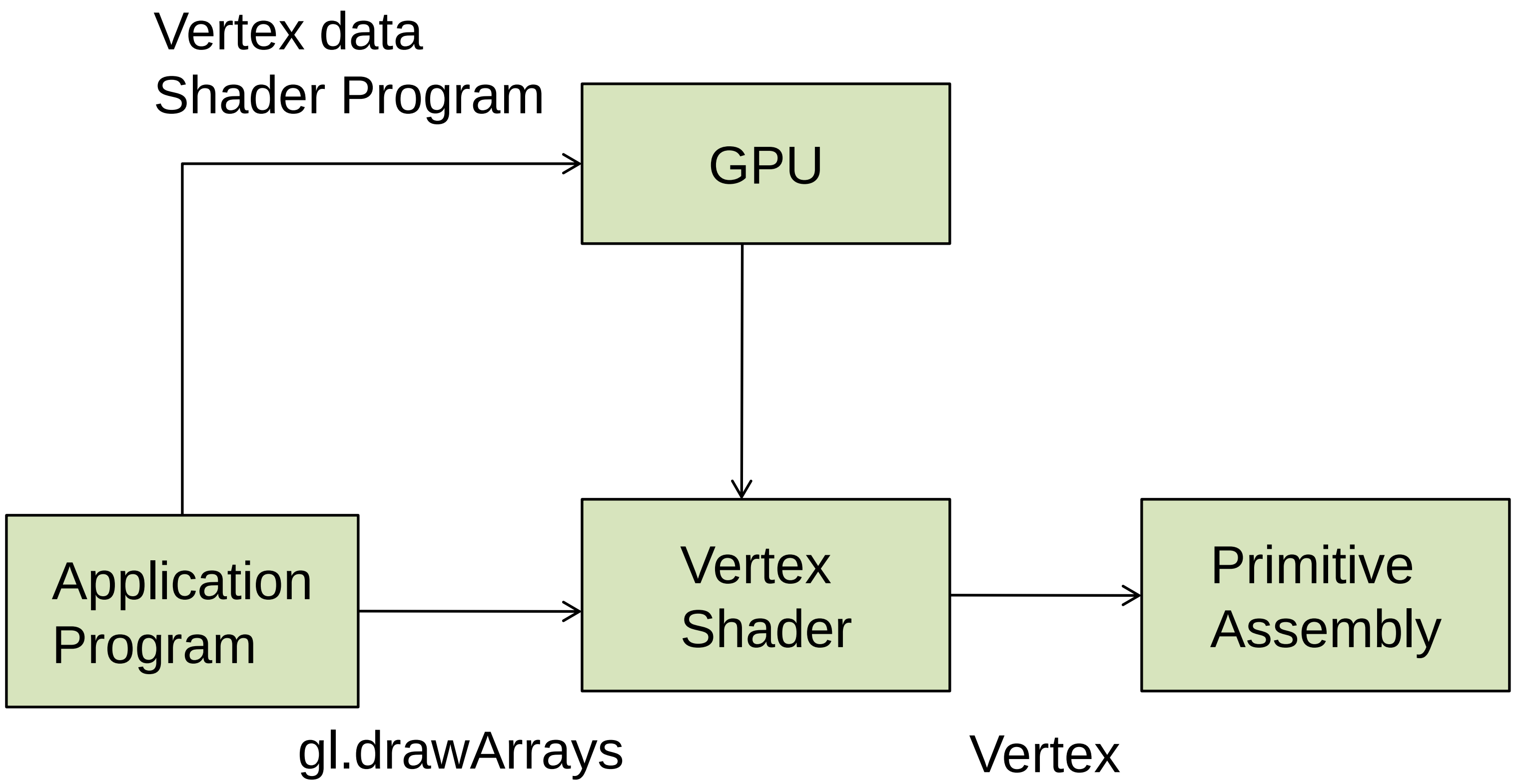
```
attribute vec4 vPosition;

void main()

{

    gl_Position = vPosition;

}
```

must link to variable in application
we use “v” at the front to indicate it’s a vertex attribute here

built-in variable (clip coordinates)

This is a simple “pass-through” vertex shader that assigns the input vertex coordinates to the output clip coordinates (gl_Position).

Execution Model (Vertex)



A Simple Fragment Shader (in GLSL)

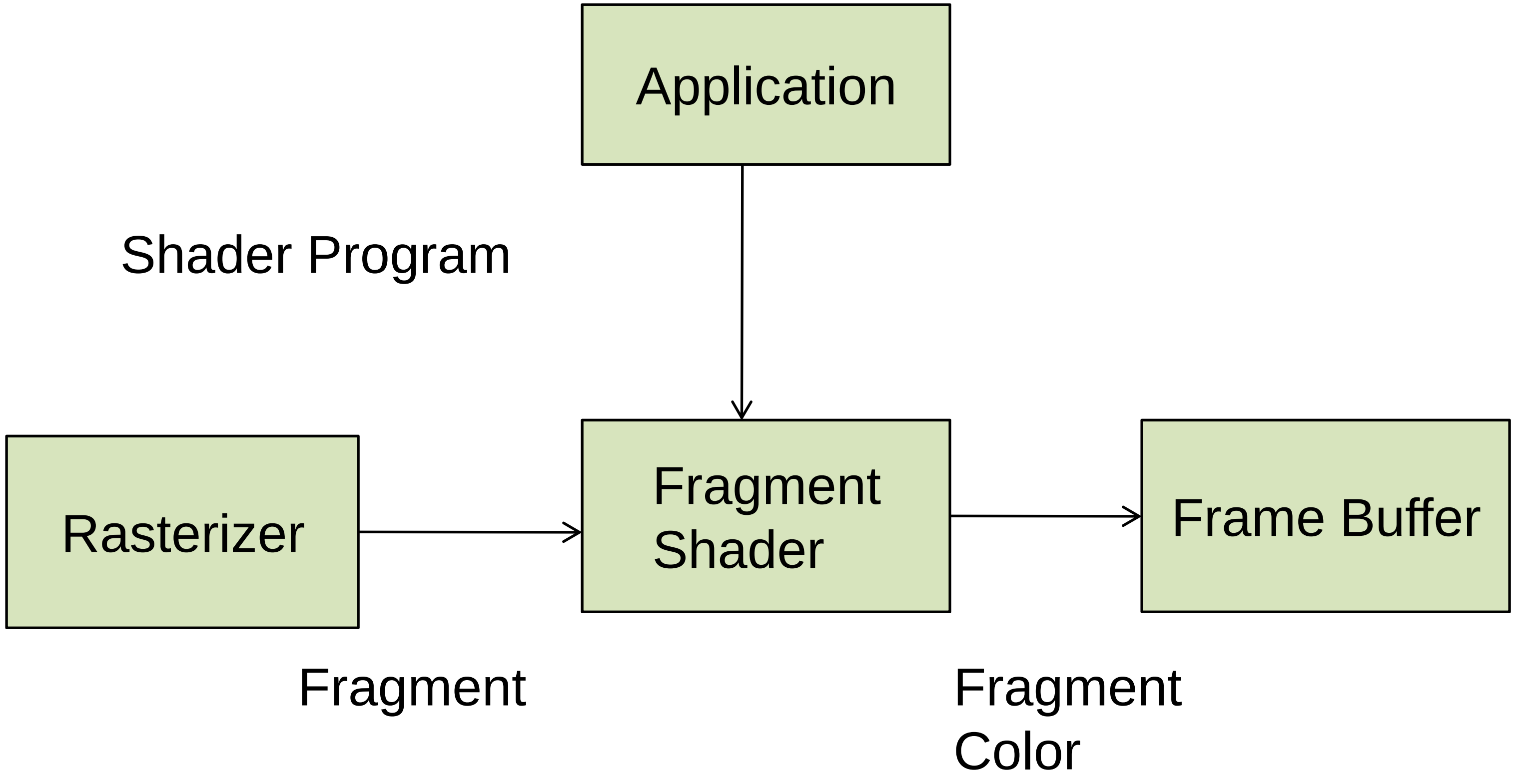
```
precision mediump float;
void main()
{
   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

must define how much precision the GPU uses for floating point math (low/medium/high)

built-in variable (final color for fragment)

This is a simple fragment shader that assigns “red” for each fragment

Execution Model (Fragment)



CS-537: Interactive Computer Graphics

Data Types in GLSL

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Stored by **columns** (“column major”)
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a = vec3(1.0, 2.0, 3.0);
  - vec2 b = vec2(a);

No Pointers in GLSL

- There are no pointers in GLSL
- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.:
  - mat3 func(mat3 a) { …}
- Variables passed by copying

Qualifiers

- GLSL has many of the same qualifiers such as `const` as C/C++
- Need others due to the nature of the execution model
- Differently qualified variables can change:
  - Once per primitive
  - Once per vertex
  - Once per fragment OR
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

The “Attribute” Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as gl_Position
  - OpenGL used to have more, but most have been deprecated
- They can also be user-defined (in application program)
  - **`attribute float temperature`**
  - **`attribute vec3 velocity`**
  - recent versions of GLSL use **`in`** and **`out`** qualifiers to get to and from shaders

The “Uniform” Qualifier

- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- **Cannot be changed in shader**
  - As far as the GPU is concerned, these are constants,
  - But the application code (JS) may change their value.
- Used to pass information to shader such as the time or a bounding box of a primitive or transformation matrices
- Assignment 1 has an example uniform qualified variable

The “Varying” Qualifier

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- With WebGL, GLSL uses the varying qualifier in both shaders

```
varying vec4 color;
```

- More recent versions of WebGL use out in vertex shader and in in the fragment shader

```
out vec4 color;    //vertex shader
in vec4 color;     //fragment shader
```

Our Naming Convention

- Attributes passed to vertex shader have names beginning with v (vPosition, vColor) in both the application (JS) and the shader
- Note that these are different entities with the same name
- varying variables begin with f (fColor) in both shaders
  - They must have same name in both vertex and fragment shader
- Uniform variables are unadorned and can have the same name in application and shaders

Example: Vertex Shader

```
attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 fColor;
void main()
{
  gl_Position = vPosition;
  fColor = vColor;
}
```

attribute keyword tells you: this is input to shader from application

varying keyword tells you: this needs to be set in the vertex shader, to be passed to the fragment shader

Example: Corresponding Fragment Shader

```
precision mediump float;
varying vec3 fColor;
void main()
{
  gl_FragColor = fColor;
}
```

varying keyword tells you: this was set by the vertex shader, and is being passed into this fragment shader

Sending colors from the application to the shader (JS)

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors),
                              gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```

Make a new buffer with identifier cBuffer

Tell GL context which buffer you're going to start modifying

Pass JS typed array data to GPU

Locate the vertex attribute variable called vColor

binds the buffer currently bound to gl.ARRAY_BUFFER to vColor and specifies its layout

"Turns on" or enables use of the variable in the shader

Sending a uniform variable value from the application to the shader

```
// in application
vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
colorLoc = gl.getUniformLocation( program, ”color" );
gl.uniform4f(colorLoc, color);


// in fragment shader (similar in vertex shader)
uniform vec4 color;
void main()
{
    gl_FragColor = color;
}
```

Operators and Functions in GLSL

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

"Swizzling" and Selection in GLSL

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - **`a[2], a.b, a.z, a.p`** are the same
- **Swizzling** operator lets us manipulate components

```
vec4 a, b;
a.yz = vec2(1.0, 2.0);
b = a.yxzw;
```

Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes (set in application, sent to shader)
  - Uniform variables (set in application, sent to shader)

Program Object

- Container for shaders
- Can contain multiple shaders
- In our assignments, most of this will happen in the common file “initShaders.js”

```
var program = gl.createProgram();

gl.attachShader( program, vertShdr );
gl.attachShader( program, fragShdr );
gl.linkProgram( program );
```

Reading a Shader

- Shaders are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function

```
gl.shaderSource( fragShdr, fragElem.text );
```

- If shader is in HTML file, we can get it into application by getElementById method
- If the shader is in a file, we can write a reader to convert the file to a string

Adding a Vertex Shader

```
var vertShdr;
var vertElem = document.getElementById( vertexShaderId );

vertShdr = gl.createShader( gl.VERTEX_SHADER );

gl.shaderSource( vertShdr, vertElem.text );
gl.compileShader( vertShdr );

// after program object created
gl.attachShader( program, vertShdr );
```

Reading a Shader (II)

- Following code may be a security issue with some browsers if you try to run it locally
  - Cross origin request

```
function getShader(gl, shaderName, type) {
        var shader = gl.createShader(type);
        shaderScript = loadFileAJAX(shaderName);
        if (!shaderScript) {
            alert("Could not find shader source:
                    "+shaderName);
        }
}
```

Precision Declaration

- In GLSL for WebGL we must specify desired precision in fragment shaders
  - artifact inherited from OpenGL ES
- ES must run on very simple embedded devices that may not support 32-bit floating point
- All implementations must support mediump
- No default for float in fragment shader
- Can use preprocessor directives (#ifdef) to check if highp supported and, if not, default to mediump