WebGL and GLSL Basics

CS559 – Spring 2018
Lecture 17
March 13th 2018
Review ...
Hardware Rasterization

For each point:
Compute barycentric coords
Decide if in or out
1988: The Personal Iris
A Pipeline

1

transf light project raster shade write
The pipeline (1988)

Application Program

Graphics Driver

Command Buffer (Triangle Queue)

Vertex Queue

Vertex Processing (TCL)

Vertex Cache

Assembly

Triangle Processing

Rasterize

Pixel Queue

Pixel Processing

Pixel Tests

Texture Memory

Frame Buffer

Vertex shading

Geometry Shading

Pixel Shading

Render to texture
The pipeline (2006-current)
The parts you **have** to program

1988-2014

Now (in addition to above)
A Triangle’s Journey
A Program to Draw a Triangle

The complete WebGL thing we need

Doing each necessary steps

Just one triangle...

http://jsbin.com/fowoku/edit
Just a Triangle
HTML like you are used to

```
<!DOCTYPE html>
<html>
<head>
<meta name="description" content="One Triangle">
<meta charset="utf-8">
<title>JS Bin</title>
</head>
<body onload="start()">
<canvas id="mycanvas" width="400" height="400"></canvas>
</body>
</html>
```
A Lot of Code

function start() {
  "new strict"
  // first we need to get the canvas and make an OpenGL context
  // In practice, you need to do error-checking
  var canvas = document.getElementById("canvas")
  var gl = canvas.getContext("experimental-webgl")
}

// now we have to program the hardware
// we need to have our GLSL code somewhere
// putting it in strings is bad - but it’s easy on I’ll
// do it for now
var vertexSource = 
  "attribute vec3 pos; 
  "
  "varying vec2 vTextureCoord; 
  "
  "void main() { 
  "
  "  pos = vec3(pos, 1.0); 
  "
  "  vTextureCoord = pos; 
  "
  "  gl_Position = vec4(pos, 0.0, 1.0); 
  "
  "}"

var fragmentSource = 
  "varying vec2 vTextureCoord; 
  "
  "void main() { 
  "
  "  gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0); 
  "
  "}"

// now we need to make those programs into
// Shader Objects - by running the compiler
// check for errors
// First compile the vertex shader
var vertexShader = gl.createShader(gl.VERTEX_SHADER)
gl.shaderSource(vertexShader, vertexSource)
gl.compileShader(vertexShader)

if (gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)) {
  alert(gl.getShaderInfoLog(vertexShader))
} else {
  null
}

// now compile the fragment shader
var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER)
gl.shaderSource(fragmentShader, fragmentSource)
gl.compileShader(fragmentShader)

if (gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
  alert(gl.getShaderInfoLog(fragmentShader))
} else {
  null
}

// OK, we have a pair of shaders, we need to put them together
// to be a "shader program" object
var shaderProgram = gl.createProgram()
gl.attachShader(shaderProgram, vertexShader)
gl.attachShader(shaderProgram, fragmentShader)
gl.linkProgram(shaderProgram)

if (gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
  alert("Could not link the shader")
} else {
  with the vertex shader, we need to pass it positions
  as an attribute - so set up that communication
  shaderProgram.vertexAttribPointer = gl.getAttribLocation(shaderProgram, "pos")
  gl.enableVertexAttribArray(shaderProgram.vertexAttribPointer)

  // now that we have programs to run on the hardware, we can
  // make our triangle

  // let's define the vertex positions
  var vertices = 
    [-1.0, -1.0, 0.0, 
    -1.0, 1.0, 0.0, 
    1.0, -1.0, 0.0, 
    1.0, 1.0, 0.0];

  // we need to put the vertices into a buffer so we can
  // block transfer them to the graphics hardware
  var triangleVertexBuffer = gl.createBuffer()
  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer)
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW)
  triangleVertexBuffer.numItems = 4

  // this is the "draw scene" function, but since this
  // is execute once...
  // First, let's clear the screen
  gl.clearColor(0.0, 0.0, 0.0, 1.0)
  gl.enable(gl.DEPTH_TEST)

  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)

  // now we draw the triangle
  // we tell GL what program to use, and what vertex block
  // we use for the data, and that the data goes to the pos
  // attribute
  gl.useProgram(shaderProgram)
  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer)
  gl.vertexAttribPointer(shaderProgram.vertexAttribPointer, triangleVertexBuffer.itemSize, gl.FLOAT, false, 0, 0)
  gl.drawArrays(gl.TRIANGLES, 0, 3)
Look at the process inside-out

We’ll start with the end of the pipeline

And work backwards...
The parts you *have* to program

1988-2014

Now (in addition to above)
I warned you:
This step will get to be pretty exciting
We have to program this step!

The program is called the:  
**Fragment Shader**
Pixel in $\rightarrow$ Pixel out, each independent

All we do is change its values
Coming attractions...

This step will get to be pretty exciting
The Pixel Shader

Given information about the pixel
Compute color

Optionally, compute other things
How to program it

Use a Shading Language

We’ll use a language called GLSL

Compiler built into WebGL

Language specifics as we go...
A (boring) Fragment Shader

```c
void main(void)
{
    gl_FragColor = vec4(0.0, 1.0, 1.0, 1.0);
}
```
A (boring) Fragment Shader

```c
void main(void)
{
    gl_FragColor = vec4(0.0, 1.0, 1.0, 1.0);
}
```

Shaders define a main function that take no arguments return no values.
A (boring) Fragment Shader

```glsl
void main(void)
{
    gl_FragColor = vec4(0.0, 1.0, 1.0, 1.0);
}
```

GLSL Shaders operate by side effects on special variables (they look like globals)
A (boring) Fragment Shader

```glsl
void main(void)
{
    gl_FragColor = vec4(0.0, 1.0, 1.0, 1.0);
}
```

GLSL has types useful in graphics
Like 4 vectors

This is opaque yellow
(even colors are 4-vectors)
Vertex Processing

We have to program this part too
Also in GLSL
Process each vertex independently

Transform – compute $x'$ and $n'$

Clip

Light – compute $c'$
What data about vertices?

Inputs:
  Position
  Other Stuff

Outputs:
  Position
  Other Stuff

Vertex Attributes from application

varying properties to fragment Shaders (remember interpolation)
The simplest vertex shader

attribute vec3 pos;

void main(void) {
    gl_Position = vec4(pos, 1.0);
}
}
The simplest vertex shader

```glsl
attribute vec3 pos;

void main(void) {
    gl_Position = vec4(pos, 1.0);
}
```

Shaders define a **main** function that take no arguments return no values.
The simplest vertex shader

attribute vec3 pos;

void main(void) {
    gl_Position = vec4(pos, 1.0);
}

Shaders output by side effects: setting special variables
The simplest vertex shader

```cpp
attribute vec3 pos;

void main(void) {
  gl_Position = vec4(pos, 1.0);
}
```

Shaders get input by reading special variables
Special Variables

Built in (magic)

- `gl_Position` – output of vertex shader
- `gl_FragColor` – output of frag shader

User Defined

- attributes – inputs to vertex shader
- varying – output from vertex to fragment
- uniform – “constant” over triangle group
The simplest vertex shader

```cpp
attribute vec3 pos;

void main(void) {
  gl_Position = vec4(pos, 1.0);
}
```

We are defining our own special variable
The simplest vertex shader

```glsl
attribute vec3 pos;

void main(void) {
    gl_Position = vec4(pos, 1.0);
}
```

Cool GLSL feature: type conversions
No Transformation?

I will assume the position is already in the right coordinate system.

The rasterizer (and everything else) works in Normalized Device Coordinates (NDC)

-1 to 1 in each dimension
Start here
Setup modes (window, ...)
Setup transform, lights
Draw a triangle
Position, color, normal
In JavaScript using WebGL...
The beginning

6 function start() {
7     "use strict";
8
9     // first we need to get the canvas and make an OpenGL context
10    // in practice, you need to do error checking
11    var canvas = document.getElementById("mycanvas");
12    var gl = canvas.getContext("webgl");
The beginning

```javascript
6 function start() {
7     "use strict";
8
9     // first we need to get the canvas and make an OpenGL context
10    // in practice, you need to do error checking
11    var canvas = document.getElementById("mycanvas");
12    var gl = canvas.getContext("webgl");
```
Now about those shaders...

```
// now we have to program the hardware
// we need to have our GLSL code somewhere
// putting it in strings is bad - but it's easy so I'll
// do it for now

var vertexSource = ""
   "attribute vec3 pos;
   "void main(void) {
   "   gl_Position = vec4(pos, 1.0);
   "}
";

var fragmentSource = ""
   "void main(void) {
   "   gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);
   "}
";
```

Get them into strings

Use a library to read them from resources
Run the compiler!

```javascript
// now we need to make those programs into "Shader Objects" - by running the compiler
// watch the steps:
// create an object
// attach the source code
// run the compiler
// check for errors

// first compile the vertex shader
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexSource);
gl.compileShader(vertexShader);

if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(vertexShader));
    return null;
}
```
Error Checking

Here I checked for errors
(since I often have syntax errors)

You should check for errors everywhere
Run the compiler again!

```javascript
// now compile the fragment shader
var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentSource);
gl.compileShader(fragmentShader);

if (!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(fragmentShader));
    return null;
}
```

Need to compile both shaders
Link the shaders together...

```javascript
46 // now compile the fragment shader
47 var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
48 gl.shaderSource(fragmentShader, fragmentSource);
49 gl.compileShader(fragmentShader);

50 if (!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
51   alert(gl.getShaderInfoLog(fragmentShader));
52   return null;
53 }
```

Shaders always work in pairs
Need to connect them
Setup the special variables

// with the vertex shader, we need to pass it positions
// as an attribute - so set up that communication
shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "pos");
gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

Important to communicate with shaders
The simplest vertex shader

attribute vec3 pos;

void main(void) {
    gl_Position = vec4(pos, 1.0);
}

Javascript needs to connect to the “pos” variable
Communicating an attribute

```
67 // with the vertex shader, we need to pass it positions
68 // as an attribute - so set up that communication
69 shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "pos");
70 gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
```

We give it an array of attributes
Assign it to a position
We have to ask which position
OK, Now for our triangle

// now that we have programs to run on the hardware, we can
// make our triangle

// let's define the vertex positions
var vertexPos = [
  0.0, 1.0, 0.0,
  -1.0, -1.0, 0.0,
  1.0, -1.0, 0.0
];

How do we get this data to the hardware?
Need to do a block transfer
Need to get the vertices to the hardware fast!

(normally more than 3)
Key Idea: Buffer

Create a buffer

buffer = a block of memory on the GPU

Copy the data into the buffer

Must be a special JavaScript object:
Float32Array (array of fixed types)
Now to draw

// this is the "draw scene" function, but since this
// is execute once...

// first, let's clear the screen
gl.clearColor(0.0, 0.0, 0.0, 1.0);
gl.enable(gl.DEPTH_TEST);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

First we have to clear the screen
Notice that color is a 4-vector

I don't really need the z-buffer
Now we actually draw the triangle

```cpp
// now we draw the triangle
// we tell GL what program to use, and what memory block
// to use for the data, and that the data goes to the pos
// attribute

// using the program

gl.useProgram(shaderProgram);

// bind the buffer

gl.bindBuffer(gl.ARRAY_BUFFER, trianglePosBuffer);

// bind the attribute

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, trianglePosBuffer.itemSize, gl.FLOAT, false, 0, 0);

// draw the triangles

gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Notice that we use the shaders and the buffer
All that for a triangle!
Is it really 100 lines of code?

Not really – lots of comments

Build wrappers to be more concise
    you do the same thing over and over

But there are lots of steps
    and you should understand them
Two triangles...

```javascript
var vertexPos = [
    0.0, 1.0, 0.0,
    -1.0, 0.0, 0.0,
    0.5, 0.0, 0.0,
    0.0, -1.0, 0.0,
    -0.5, 0.0, 0.0,
    1.0, 0.0, 0.0,
];
```

Can you see where these triangles will go? (remember they are in NDC)
Change the array sizes

```javascript
86 // we need to put the vertices into a buffer so we can
87 // block transfer them to the graphics hardware
88 var trianglePosBuffer = gl.createBuffer();
89 gl.bindBuffer(gl.ARRAY_BUFFER, trianglePosBuffer);
90 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexPos), gl.STATIC_DRAW);
91 trianglePosBuffer.itemSize = 3;
92 trianglePosBuffer.numItems = 6;

104 // now we draw the triangle(s)
105 // we tell GL what program to use, and what memory block
106 // to use for the data, and that the data goes to the pos
107 // attribute
108 gl.useProgram(shaderProgram);
109 gl.bindBuffer(gl.ARRAY_BUFFER, trianglePosBuffer);
110 gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
111     trianglePosBuffer.itemSize, gl.FLOAT, false, 0, 0);
112 gl.drawArrays(gl.TRIANGLES, 0, trianglePosBuffer.numItems);
```
Two triangles
How do we color them differently?
Color per vertex

Add an attribute for each vertex so we can pass a color for each

Have the vertex shader output the color varying variable for fragment shader

Have the fragment shader input the color
A (boring) Fragment Shader

```cpp
void main(void)
{
    gl_FragColor = vec4(0.0, 1.0, 1.0, 1.0);
}
```
A (less boring) Fragment Shader

precision highp float;
varying vec3 outColor;

void main(void)
{
    gl_FragColor = vec4(outColor, 1.0);
}
A (less boring) Fragment Shader

precision highp float;

varying vec3 outColor;

void main(void)
{
    gl_FragColor = vec4(outColor, 1.0);
}
Connecting Shaders

`varying` variables connect shaders

the output of a vertex shader becomes the input to a fragment shader

The 3 vertices of a triangle are interpolated
The simplest vertex shader

attribute vec3 pos;

void main(void) {
    gl_Position = vec4(pos, 1.0);
}
The (almost) simplest vertex shader

attribute vec3 pos;

varying vec3 outColor;

void main(void) {
  gl_Position = vec4(pos, 1.0);
  outColor = vec3(1.0, 0.0, 1.0);
}
Two purple triangles

http://jsbin.com/wecaci/edit?js,output
Make color an input as well

attribute vec3 pos;
attribute vec3 inColor;
varying vec3 outColor;

void main(void) {
  gl_Position = vec4(pos, 1.0);
  outColor = inColor;
}
Remember...

We can’t pass values directly to a fragment we don’t even know what they will be!

We pass attributes of vertices which can then pass them to fragments
Now to connect to JavaScript...

```javascript
shaderProgram.inColor = gl.getAttribLocation(shaderProgram, "inColor");
gl.enableVertexAttribArray(shaderProgram.inColor);
```
Colors per vertex

```javascript
var vertexColors = [
  1.0, 1.0, 0.0,
  1.0, 1.0, 0.0,
  1.0, 1.0, 0.0,
  1.0, 0.0, 1.0,
  1.0, 0.0, 1.0,
  1.0, 0.0, 1.0
];
```
Put them in a buffer

110  // a buffer for colors
111  var colorBuffer = gl.createBuffer();
112  gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
113  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexColors), gl.STATIC_DRAW);
114  colorBuffer.itemSize = 3;
115  colorBuffer.numItems = 6;
When we draw, use 2 buffers

```javascript
132 gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
133 gl.vertexAttribPointer(shaderProgram.inColor, colorBuffer.itemSize,
   gl.FLOAT, false, 0, 0);
134 gl.bindBuffer(gl.ARRAY_BUFFER, trianglePosBuffer);
135 gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
   trianglePosBuffer.itemSize, gl.FLOAT, false, 0, 0);
136 gl.drawArrays(gl.TRIANGLES, 0, trianglePosBuffer.numItems);
```
Two triangles...

http://jsbin.com/digupi/edit?js,output
Apply a transformation

One transformation for the triangle group

It is constant over the “drawArrays” call

This is a uniform variable

http://jsbin.com/tirapu/19/edit?js,output
Simplifying the Code

There is stuff you do over and over and ...

Write it once and use it often
Or let someone else write it once...

This is where twgl comes in
Compile two vertex programs

For each…
  run the compiler
  check for errors
Link them together
Attach to the attributes
Set up to specify the uniforms
Do it by hand...

```javascript
// first compile the vertex shader
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexSource);
if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)) {
  alert(gl.getShaderInfoLog(vertexShader));
  return null;
}

// now compile the fragment shader
var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
if (!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
  alert(gl.getShaderInfoLog(fragmentShader));
  return null;
}

// OK, we have a pair of shaders, we need to put them together
// into a "shader program" object
var shaderProgram = gl.createProgram();
if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
  alert("Could not initialise shaders");
}

// with the vertex shader, we need to pass it positions
// as an attribute - so set up that communication
shaderProgram.vertexAttribPointerAttribute = gl.getAttribLocation(shaderProgram, "pos");
gl.enableVertexAttribArray(shaderProgram.vertexAttribPointerAttribute);

// this gives us access to the matrix uniform
shaderProgram.transf = gl.getUniformLocation(shaderProgram, "transf");
```

Do it with `twgl`

```javascript
var shaders =
twgl.createProgramInfo(gl, ["vs", "fs"]);
```

Yes, one line...

And it grabs the string from script tags so they are separate from your JS program.

But the documentation is terrible.
How about those shaders...

They do very specific things
you need to understand the pipeline
They have 3 kinds of weird variables
you need to understand the model
They are written in a cool language
you’ll pick it up quickly
The language has a bunch of useful stuff
look at the quick reference card
Learning Shader Programming

Connecting your program to shaders is hard

So, don’t bother… (yet)

Use a Shader IDE that lets you focus on shaders

Gives you an object, a program, …
Some things about GLSL

Very strongly typed

- `float x = 1; // error! integer and float`

Cool “sub-vector” access:

- `vec3 v;
- v.xy (a 2-vector)
- vec4(v,1) (a 4-vector)
- vec4(v.xy, v.zx)`
More cool stuff about GLSL

Lots of handy math functions
  They know it’s for graphics!

Limited control structures
  parallel execution means all the same
Conditional functions
  step, softstep, …