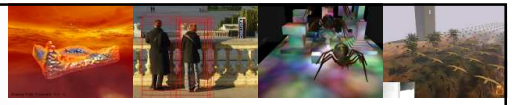


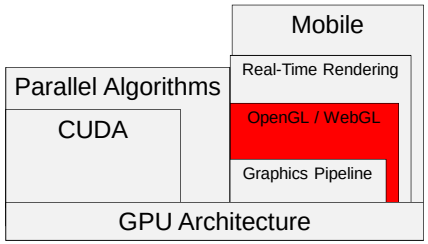
Introduction to GLSL

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2012

Announcements

- Initial project blog post due today
  - Email a link to cis565-s2012@googlegroups.com
- Homework 4 will be released 03/19

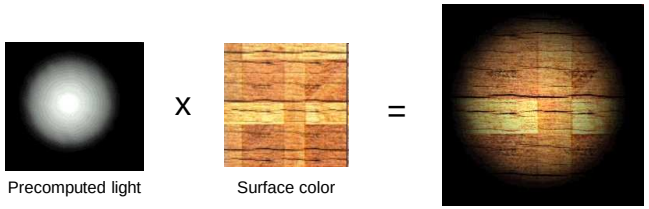
Course Contents



Agenda

- Fixed vs. Programmable Pipeline Example
- GLSL

Light Map



- Multiple two textures component-wise

Images from: http://zanir.wz.cz/?p=56&lang=en

Light Map: Fixed Function

```
GLuint lightMap;
GLuint surfaceMap;
// ...

glEnable(GL_TEXTURE_2D);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, lightMap);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, surfaceMap);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

glDraw*(...);
```

Tell fixed function we are using texture mapping

Tell fixed function how to combine textures

Light Map: Fixed Function

- In general, the fixed function
  - is *configurable*
  - is limited
  - leads to a bloated API
  - Is a pain to use
  - Isn’t as cool as writing shaders
    - True – but not a valid answer on the homework/exam

Light Map: Programmable

- Write a *fragment shader*:

```
uniform sampler2D lightMap;
uniform sampler2D surfaceMap;

varying vec2 fs_txCoord;

void main(void)
{
    float intensity = texture2D(lightMap, fs_txCoord).r;
    vec3 color = texture2D(surfaceMap, fs_txCoord).rgb;
    gl_FragColor = vec4(intensity * color, 1.0);
}
```

Textures (input)

Per-fragment input

one channel intensity

Three channel color

modulate

Light Map: Programmable

Recall the fixed function light map:

```
GLuint lightMap;
GLuint surfaceMap;
// ...

glEnable(GL_TEXTURE_2D);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, lightMap);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, surfaceMap);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

glDraw*(...);
```

Light Map: Programmable

```
GLuint lightMap;
GLuint surfaceMap;
GLuint program;
// ...

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, lightMap);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, surfaceMap);

glUseProgram(program);  // Later: pass uniform variables
glDraw*(...);
```

Programmable Shading

- In general:
  - Write a *shader*: a small program that runs on the GPU
  - Tell OpenGL to execute your shader
  - Write less CPU code / API calls
  - Forget that the equivalent fixed function API ever existed

Programmable Shading

- In general:

Say no to drugs too, please.

Fixed function shading

Programmable shading

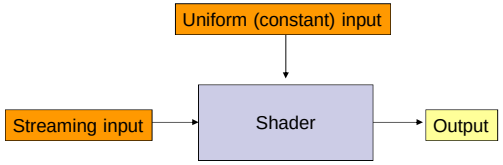
Image from: http://upgifting.com/tmnt-pizza-poster

Programmable Shading

- Software engineering question:
  - If different GPUs have different levels of shader support, what capabilities do we target?

Shader Execution Model

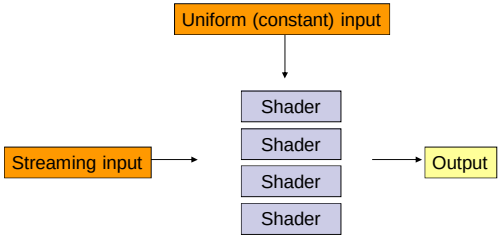
- For any shader type:



• Streaming input and output examples: vertices, primitives, fragments, …
• Uniform input examples: matrices, textures, time, …

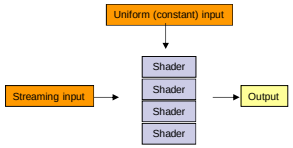
Shader Execution Model

- Shaders run in *parallel* on the GPU



Shader Execution Model

- Each shader
  - Shares the same read-only *uniform* inputs
  - Has different read-only input from a stream
  - Writes it own output
  - Has no side effects*
  - Executes independently without communicating with other shaders*

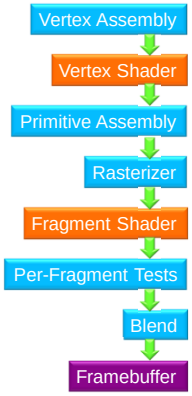


* Except in OpenGL 4.2+

Shader Execution Model

- Parallelism is implicit
  - Calling `glDraw*` invokes a parallel processor – the GPU
  - The driver/hardware takes care of scheduling and synchronizing
  - Users write parallel applications without even knowing it!

Shaders in the Pipeline



- Classic OpenGL 2 / OpenGL ES 2 / WebGL / Direct3D 9 Pipeline
- Newer pipelines also have programmable geometry and tessellation shaders

Vertex Shaders in the Pipeline

- A simple *vertex shader*:

```
uniform mat4 u_modelViewProjection;
attribute vec4 position;

void main(void)
{
  gl_Position = u_modelViewProjection * vec4(position, 1.0);
}
```

The same model-view transform is used for each vertex in a particular glDraw* call.

Each vertex shader executes in a different thread with a different `position`.

`gl_Position` is the GLSL built-in vertex shader position output. We must write to it.

4x4 matrix times a 4-element vector; transform from model to clip coordinates.

Vertex Shaders in the Pipeline

- A *vertex shader* with two input attributes:

```
uniform mat4 u_modelViewProjection;
attribute vec3 position;
attribute vec3 color;
varying vec3 fs_color;

void main(void)
{
  fs_color = color;
  gl_Position = u_modelViewProjection * vec4(position, 1.0);
}
```

Each vertex shader executes in a different thread with a different `position` and `color`.

This vertex shader outputs a `vec3` color in addition to `gl_Position`.

Fragment Shaders in the Pipeline

- Recall:
  - Input
    - Fragment position in screen space: `gl_FragCoord.xy`
    - Fragment depth: `gl_FragCoord.z`
    - Interpolated vertex shader outputs
    - Uniforms
  - Output
    - Fragment color
    - Optional: fragment depth: `gl_FragDepth`*
    - Optional: multiple "colors" to multiple textures*
    - `discard`
    - Can't change `gl_FragCoord.xy`. Why?

* When supported

Fragment Shaders in the Pipeline

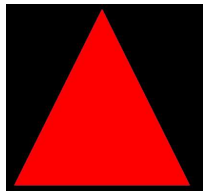
- A simple *fragment shader*:

Each fragment shader executes in a different thread and outputs the color for a different fragment.

```
void main(void)
{
  gl_FragColor = vec3(1.0, 0.0, 0.0, 1.0);
}
```

Shade solid red.

Result:



Fragment Shaders in the Pipeline

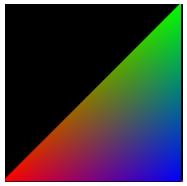
- A slightly less simple *fragment shader*:

Fragment shader input from vertex shader output after rasterization.

```
varying vec3 fs_color;

void main(void)
{
  gl_FragColor = fs_color;
}
```

Pass color through.

Result:



How?

GLSL Syntax

- GLSL is like C without
  - pointers
  - recursion
  - dynamic memory allocation
- GLSL is like C with
  - Built-in vector, matrix, and sampler types
  - Constructors
  - A great math library

Language features allow us to write concise, efficient shaders.

GLSL Syntax

- My advice: If you know C, just do it.

Image from: http://nouvellemode.wordpress.com/2009/11/25/just-do-it/

GLSL Syntax

- GLSL has a preprocessor

```
#ifdef FAST_EXACT_METHOD
  FastExact();
#else
  SlowApproximate();
#endif

#line 0

// ... many others
```

- All shaders have `main()`

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

GLSL Syntax: Vectors

- Scalar types: `float`, `int`, `uint`, and `bool`
- Vectors are also built-in types:
  - `vec2`, `vec3`, and `vec4`
  - Also `ivec*`, `uvec*`, and `bvec*`
- Access components three ways:
  - `.x`, `.y`, `.z`, `.w` ← Position or direction
  - `.r`, `.g`, `.b`, `.a` ← Color
  - `.s`, `.t`, `.p`, `.q` ← Texture coordinate

GLSL Syntax: Vectors

- Vectors have constructors

```
vec3 xyz = vec3(1.0, 2.0, 3.0);

vec3 xyz = vec3(1.0); // [1.0, 1.0, 1.0]

vec3 xyz = vec3(vec2(1.0, 2.0), 3.0);
```

GLSL Syntax: Swizzling

- *Swizzle*: select or rearrange components

```
vec4 c = vec4(0.5, 1.0, 0.8, 1.0);

vec3 rgb = c.rgb;  // [0.5, 1.0, 0.8]

vec3 bgr = c.bgr;  // [0.8, 1.0, 0.5]

vec3 rrr = c.rrr;  // [0.5, 0.5, 0.5]

c.a = 0.5;         // [0.5, 1.0, 0.8, 0.5]
c.rb = 0.0;        // [0.0, 1.0, 0.0, 0.5]

float g = rgb[1];  // 0.5, indexing, not swizzling
```

- Try it – you'll love it.

GLSL Syntax: Matrices

- Matrices are built-in types:
  - Square: `mat2`, `mat3`, and `mat4`
  - Rectangular: `matmxn`. *m* columns, *n* rows
- Stored *column major*.

GLSL Syntax: Matrices

- Matrix Constructors

```
mat3 i = mat3(1.0); // 3x3 identity matrix

mat2 m = mat2(1.0, 2.0,  // [1.0 3.0]
              3.0, 4.0); // [2.0 4.0]
```

- Accessing Elements

```
float f = m[column][row];

float x = m[0].x; // x component of first column

vec2 yz = m[1].yz; // yz components of second column
```

Treat matrix as array of column vectors

Can swizzle too!

GLSL Syntax: Vectors and Matrices

- Matrix and vector operations are easy and fast:

```
vec3 xyz = // ...

vec3 v0 = 2.0 * xyz; // scale
vec3 v1 = v0 + xyz;  // component-wise
vec3 v2 = v0 * xyz;  // component-wise

mat3 m = // ...
mat3 v = // ...

mat3 mv = v * m;      // matrix * matrix
mat3 xyz2 = mv * xyz; // matrix * vector
mat3 xyz3 = xyz * mv; // vector * matrix
```

GLSL Syntax: `attribute` / `varying` / `uniform`

- Recall:

```
uniform mat4 u_modelViewProjection;
attribute vec3 position;
attribute vec3 color;
varying vec3 fs_color;

void main(void)
{
  fs_color = color;
  gl_Position = u_modelViewProjection * vec4(position, 1.0);
}
```

`uniform`: shader input constant across `glDraw*`

`attribute`: shader input varies per vertex attribute

`varying`: shader output

GLSL Syntax: Samplers

- *Opaque* types for accessing textures

```
uniform sampler2D diffuseMap; // 2D texture

vec3 color = texture2D(diffuseMap, vec2(0.5, 0.5)).rgb;

// Also samplerCube.
```

GLSL Syntax: Samplers

- *Opaque* types for accessing textures

```
uniform sampler2D diffuseMap; // 2D texture

vec3 color = texture2D(diffuseMap, vec2(0.5, 0.5)).rgb;

// Also samplerCube.
```

Samplers must be uniforms

GLSL Syntax: Samplers

- *Opaque* types for accessing textures

```
uniform sampler2D diffuseMap; // 2D texture

vec3 color = texture2D(diffuseMap, vec2(0.5, 0.5)).rgb;

// Also samplerCube.
```

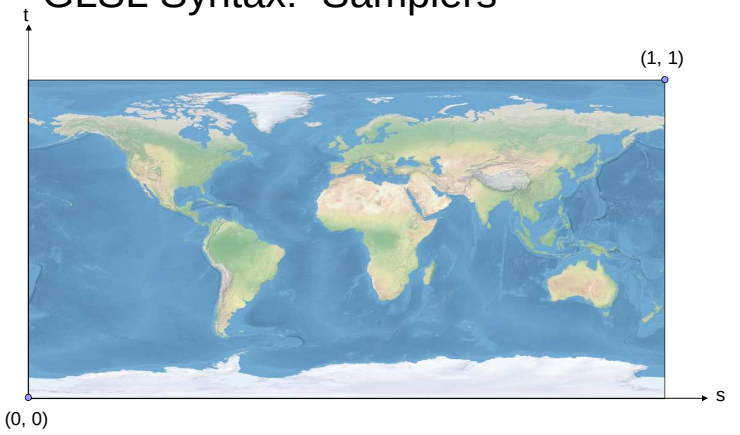
`texture()` returns a `vec4`; extract the components you need

2D texture uses 2D texture coordinates for lookup

GLSL Syntax: Samplers

- Textures
  - Usually, but not always:
    - Textures are square, e.g., 256x256
    - Dimensions are a power of two
  - Coordinates are usually normalized, i.e., in the range [0, 1]
  - *Texel*: a pixel in a texture
  - `texture2D()` does filtering using fixed function hardware

GLSL Syntax: Samplers



Images from: http://www.naturalearthdata.com/

GLSL Built-in Functions

- Selected Trigonometry Functions

```
float s = sin(theta);
float c = cos(theta);
float t = tan(theta);

float as = asin(theta);
// ...

vec3 angles = vec3(/* ... */);
vec3 vs = sin(angles);
```

Works on vectors component-wise.

GLSL Built-in Functions

- Exponential Functions

```
float xToTheY = pow(x, y);
float eToTheX = exp(x);
float twoToTheX = exp2(x);

float l = log(x);   // ln
float l2 = log2(x); // log2

float s = sqrt(x);
float is = inversesqrt(x);
```

One GPU instruction!

GLSL Built-in Functions

- Selected Common Functions

```
float ax = abs(x);  // absolute value
float sx = sign(x); // -1.0, 0.0, 1.0

float m0 = min(x, y); // minimum value
float m1 = max(x, y); // maximum value
float c = clamp(x, 0.0, 1.0);

// many others:  floor(), ceil(),
// step(), smoothstep(), ...
```

GLSL Built-in Functions

- Rewrite with one function call

```
float minimum = // ...
float maximum = // ...
float x = // ...

float f = min(max(x, minimum), maximum);
```

GLSL Built-in Functions

- Rewrite this without the `if` statement

```
float x = // ...
float f;

if (x > 0.0)
{
  f = 2.0;
}
else
{
  f = -2.0;
}
```

GLSL Built-in Functions

- Rewrite this without the `if` statement

```
float root1 = // ...
float root2 = // ...

if (root1 < root2)
{
  return vec3(0.0, 0.0, root1);
}
else
{
  return vec3(0.0, 0.0, root2);
}
```

GLSL Built-in Functions

- Rewrite this without the `if` statement

```
bool b = // ...
vec3 color;

if (b)
{
  color = vec3(1.0, 0.0, 0.0);
}
else
{
  color = vec3(0.0, 1.0, 0.0);
}
```

Hint: no built-in functions required for this one.

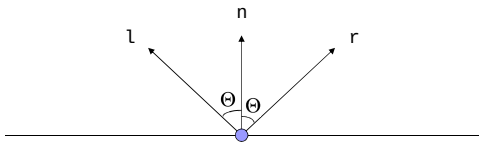
GLSL Built-in Functions

- Selected Geometric Functions

```
vec3 l = // ...
vec3 n = // ...
vec3 p = // ...
vec3 q = // ...

float f = length(l);      // vector length
float d = distance(p, q); // distance between points

float d2 = dot(l, n);     // dot product
vec3 v2 = cross(l, n);    // cross product
vec3 v3 = normalize(l);   // normalize

vec3 v3 = reflect(l, n);  // reflect

// also:  faceforward() and refract()
```

GLSL Built-in Functions

- `reflect(-l, n)`
  - Given `l` and `n`, find `r`. Angle in equals angle out



GLSL Built-in Functions

- Rewrite without `length`.

```
vec3 p = // ...
vec3 q = // ...

vec3 v = length(p – q);
```

GLSL Built-in Functions

- What is wrong with this code?

```
vec3 n = // ...

normalize(n);
```

GLSL Built-in Functions

- Selected Vector Relational Functions

```
vec3 p = vec3(1.0, 2.0, 3.0);
vec3 q = vec3(3.0, 2.0, 1.0);

bvec3 b = equal(p, q);        // (false, true, false)
bvec3 b2 = lessThan(p, q);    // (true, false, false)
bvec3 b3 = greaterThan(p, q); // (false, false, true)

bvec3 b4 = any(b);            // true
bvec3 b5 = all(b);            // false
```

GLSL Built-in Functions

- Rewrite this in one line of code

```
bool foo(vec3 p, vec3 q)
{
  if (p.x < q.x)
  {
    return true;
  }
  else if (p.y < q.y)
  {
    return true;
  }
  else if (p.z < q.z)
  {
    return true;
  }
  return false;
}
```

GLSL Syntax and Built-in Functions

- We didn't cover:
  - Arrays
  - Structs
  - Function calls
  - `const`
  - `if` / `while` / `for`
  - `dFdX`, `dFdy`, `fwidth`
  - …

GLSL Resources

- OpenGL ES/GLSL Quick Reference Card
  - http://www.khronos.org/opengles/sdk/2.0/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf
- GLSL Man Pages
  - http://www.opengl.org/sdk/docs/manglsl/
- NShader: Visual Studio GLSL syntax highlighting
  - http://nshader.codeplex.com/