Enhancing Computer Graphics Effects by Writing Shaders

1

Oregon State University

Mike Bailey

mjb@cs.oregonstate.edu

1

How Many Computers do you see in this Photo? One?

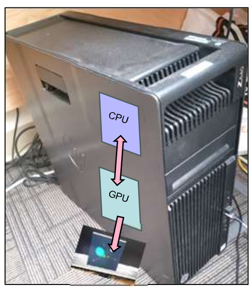
2

2

No, There Are *Two* Computers Here

3

Buried within a single chassis, we are tempted to think there is just one computer here.

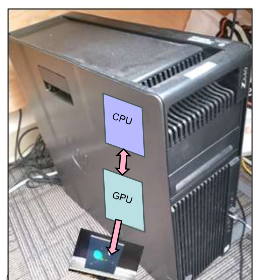


But there are really *two* computers here, a CPU and a GPU. So far, you have been "programming" the GPU by telling OpenGL how to do it for us. This is about to change!

3

No, There Are *Two* Computers Here

4



We are now going to get into a way-cool part of this class where you get to program the GPU yourself. This is called ***Shaders***.

Let's think about it. If you set out to program an external computer, here is what you would need:

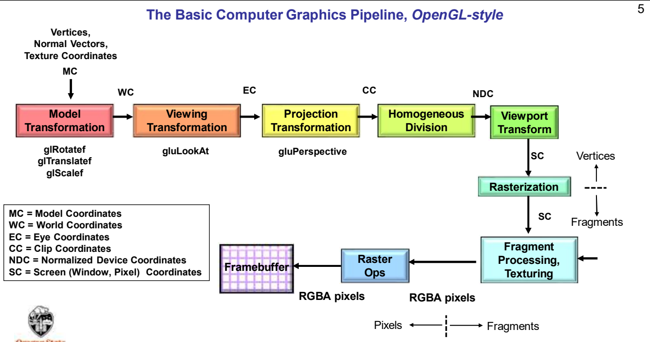
1. **A programming language**
2. **A compiler for that language to create an executable**
3. **A way to see the compiler's error messages**
4. **A way to download the executable onto the external computer**
5. **A way to run that executable on the external computer**
6. **A way to get information into the executable**

This sounds like a lot, but it won't turn out to be that big a deal. Trust me!

4

The Basic Computer Graphics Pipeline, *OpenGL-style*

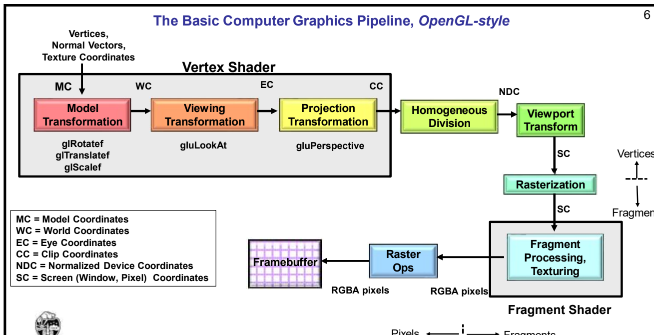
5



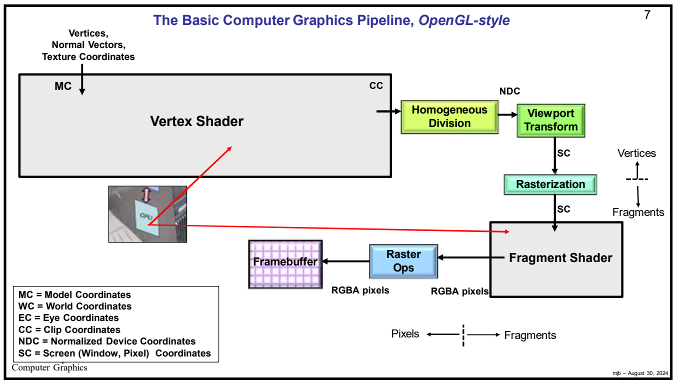
5

The Basic Computer Graphics Pipeline, *OpenGL-style*

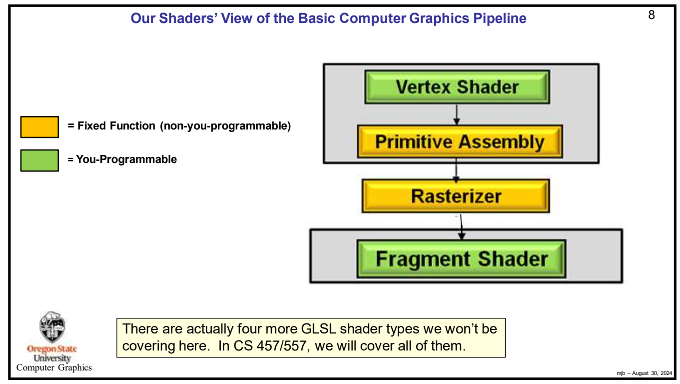
6



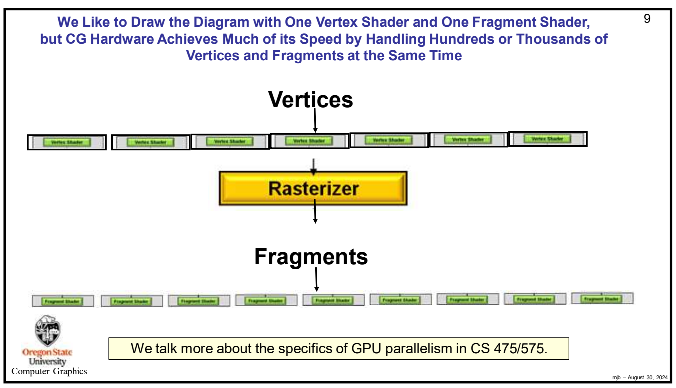
6


The Basic Computer Graphics Pipeline, *OpenGL-style*
7
Vertices,
Normal Vectors,
Texture Coordinates
MC
CC
Vertex Shader
NDC
Homogeneous Division
Viewport Transform
SC
Vertices
Rasterization
SC
Fragments
Framebuffer
Raster Ops
Fragment Shader
RGBA pixels
RGBA pixels
MC = Model Coordinates
WC = World Coordinates
EC = Eye Coordinates
CC = Clip Coordinates
NDC = Normalized Device Coordinates
SC = Screen (Window, Pixel) Coordinates
Pixels
Fragments
Computer Graphics

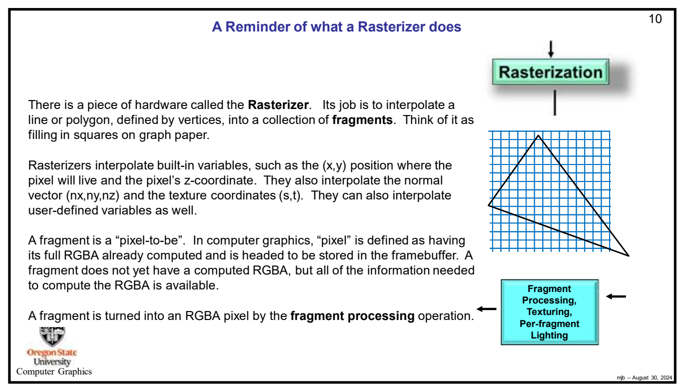
7


Our Shaders' View of the Basic Computer Graphics Pipeline
8
= Fixed Function (non-you-programmable)
= You-Programmable
Vertex Shader
Primitive Assembly
Rasterizer
Fragment Shader
There are actually four more GLSL shader types we won't be covering here. In CS 457/557, we will cover all of them.
Oregon State University
Computer Graphics

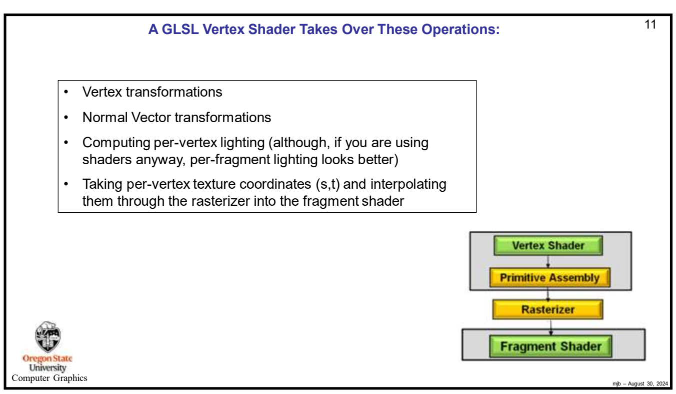
8


We Like to Draw the Diagram with One Vertex Shader and One Fragment Shader, but CG Hardware Achieves Much of its Speed by Handling Hundreds or Thousands of Vertices and Fragments at the Same Time
9
Vertices
Rasterizer
Fragments
We talk more about the specifics of GPU parallelism in CS 475/575.
Oregon State University
Computer Graphics

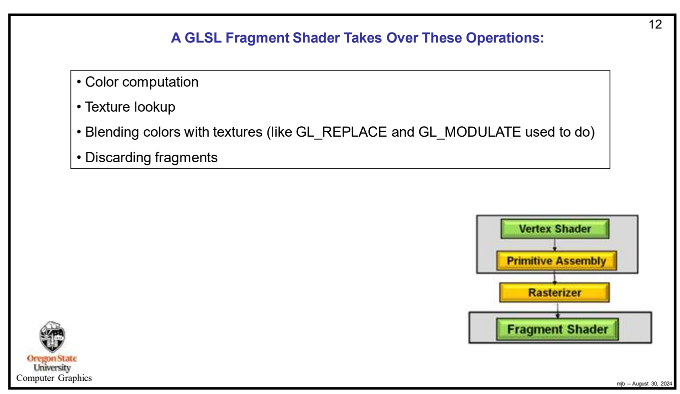
9


A Reminder of what a Rasterizer does
10
Rasterization
There is a piece of hardware called the **Rasterizer**. Its job is to interpolate a line or polygon, defined by vertices, into a collection of **fragments**. Think of it as filling in squares on graph paper.
Rasterizers interpolate built-in variables, such as the (x,y) position where the pixel will live and the pixel's z-coordinate. They also interpolate the normal vector (nx,ny,nz) and the texture coordinates (s,t). They can also interpolate user-defined variables as well.
A fragment is a "pixel-to-be". In computer graphics, "pixel" is defined as having its full RGBA already computed and is headed to be stored in the framebuffer. A fragment does not yet have a computed RGBA, but all of the information needed to compute the RGBA is available.
A fragment is turned into an RGBA pixel by the **fragment processing** operation.
Fragment Processing, Texturing, Per-fragment Lighting
Oregon State University
Computer Graphics

10


A GLSL Vertex Shader Takes Over These Operations:
11
• Vertex transformations
• Normal Vector transformations
• Computing per-vertex lighting (although, if you are using shaders anyway, per-fragment lighting looks better)
• Taking per-vertex texture coordinates (s,t) and interpolating them through the rasterizer into the fragment shader
Vertex Shader
Primitive Assembly
Rasterizer
Fragment Shader
Oregon State University
Computer Graphics

11


A GLSL Fragment Shader Takes Over These Operations:
12
• Color computation
• Texture lookup
• Blending colors with textures (like GL_REPLACE and GL_MODULATE used to do)
• Discarding fragments
Vertex Shader
Primitive Assembly
Rasterizer
Fragment Shader
Oregon State University
Computer Graphics

12

1. We Need a Programming Language

OpenGL developed a shader language called **GLSL: GL Shader Language.**

GLSL is very C-ish, so it should look familiar.

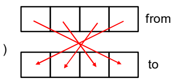
GLSL has Many C-Familiar Data Types, plus Extensions for Graphics:

- Types include int, ivec2, ivec3, ivec4
- Types include float, vec2, vec3, vec4
- Types include bool, bvec2, bvec3, bvec4

Computer Graphics uses values in groups of 2, 3, and 4

- Vector components are accessed with .rgba, .xyzw, or.stpq
- Types include mat4 } Computer Graphics uses 4x4 matrices to transform 3D vertices
- Types include sampler1D, sampler2D, sampler3D to access textures
- You can use parallel SIMD operations (doesn't necessarily get implemented in hardware):

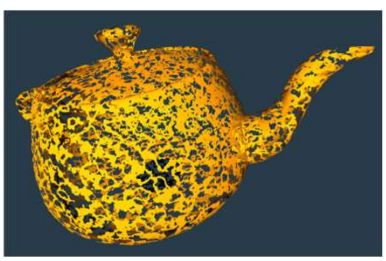
```
vec4 a = vec4( 1., 2., 3., 4. );
vec4 b = vec4( 5., 6., 7., 8. );
. . .
vec4 c = a + b;
```



- Vector components can be "swizzled" (to.abgr = from.rgba)
- Type qualifiers: const, uniform, in, out
- Variables can have "layout qualifiers" to describe how data is stored
- The *discard* operator is used in fragment shaders to get rid of the current fragment

The *discard* Operator Halts Production of the Current Fragment

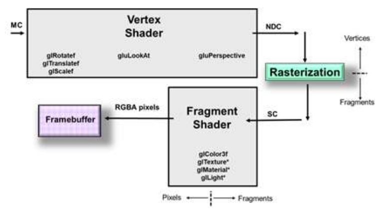

```
if( random_number < 0.5 )
        discard;
```



GLSL also has Some Different Variable Types to Pass Information Around

uniform These are "global" values, assigned into your GLSL program from your C++ program and left alone for a group of primitives. They are read-only accessible from all of your shaders. **They cannot be written to from a shader.**

out / in These are passed ***out*** from the vertex shader stage, interpolated in the rasterizer, and passed ***in*** to the fragment shader stage.



GLSL has Some *Built-in* Vertex Shader Variables :

Input built-ins:

- vec4 gl_Vertex
- vec3 gl_Normal
- vec4 gl_Color
- vec4 gl_MultiTexCoord0
- mat4 gl_ModelViewMatrix
- mat4 gl_ProjectionMatrix
- mat4 gl_ModelViewProjectionMatrix (= gl_ModelViewMatrix * gl_ProjectionMatrix)
- mat3 gl_NormalMatrix (this is the transpose of the inverse of the MV matrix)

Output built-in:

- vec4 gl_Position

Note: while this all still works, OpenGL now prefers that you pass in all the above input variables as user-defined *in* variables. We can talk about this later. For now, we are going to use the most straightforward approach possible.

GLSL has Some *Built-in* Fragment Shader Variables :

Output built-in:

- vec4 gl_FragColor = the RGBA being sent to the framebuffer

Note: while this all still works, OpenGL now prefers that you pass the RGBA out as a user-defined *out* variable. We can talk about this later. For now, we are going to use the most straightforward approach possible.

We haven't forgotten about this.
If you set out to program an external computer, here is what you would need:

1. A programming language

   GLSL

2. A compiler for that language to create an executable

   The GLSL compiler is pre-built into the OpenGL driver. You've already got it.

3. A way to see the compiler's error messages
4. A way to download the executable onto the external computer
5. A way to run that executable on the external computer
6. A way to get information into the executable

We will give you a C++ class to take care of all of this. This is coming up soon.

.
But, first, let's take a look at what vertex and fragment shader code looks like.

My Own Variable Naming Convention

With 7 different places that GLSL variables can be created from, I decided to adopt a naming convention to help me recognize what program-defined variables came from what sources:

Beginning letter(s)	Means that the variable …
a	Is a per-vertex in (attribute) from the application
u	Is a uniform variable from the application
v	Came from the vertex shader
tc	Came from the tessellation control shader
te	Came from the tessellation evaluation shader
g	Came from the geometry shader
f	Came from the fragment shader

This isn't part of "official" GLSL – it is just *my* way of handling the chaos

The Minimal Vertex and Fragment Shader

A Vertex Shader is automatically called once per vertex:

```
#version 330 compatibility

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

vec4 gl_Vertex
vec3 gl_Normal
vec4 gl_Color
vec4 gl_MultiTexCoord0
mat4 gl_ModelViewMatrix
mat4 gl_ProjectionMatrix
mat4 gl_ModelViewProjectionMatrix (= gl_ModelViewMatrix * gl_ProjectionMatrix)
mat3 gl_NormalMatrix (this is the transpose of the inverse of the MV matrix)

vec4 gl_Position

Rasterizer

A Fragment Shader is automatically called once per fragment:

```
#version 330 compatibility

void
main( )
{
        gl_FragColor = vec4( .5, 1., 0.,  1. );
}
```

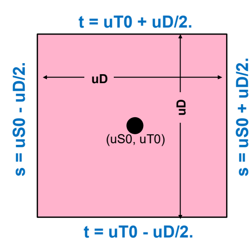
This code assigns a fixed color (r=0.5, g=1., b=0.) and alpha (=1.) to each fragment drawn

vec4 gl_FragColor

Not terribly useful yet …

A Little More Interesting, I:
What if we Want to Color in a Pattern?

This pattern example is defined by three uniform variables: uS0, uT0, and uD, all in texture coordinates (0.-1.). **(uS0,uT0)** are the center of the pattern. **uD** is the length of each edge of the pattern. The s and t boundaries of the pattern are like this:



- uS0, uT0 are the center of the pattern in (0.-1.) texture coordinates
- uD is the size of the pattern in (0.-1.) texture coordinates

A Little More Interesting, II:
Getting the Texture Coordinates from the Vertex Shader to the Fragment Shader

The vertex shader needs to pass the texture coordinates to the rasterizer so that each fragment shader gets it:

A Vertex Shader is automatically called once per vertex:

```
#version 330 compatibility

out  vec2  vST;

void
main( )
{
        vST = gl_MultiTexCoord0.st;          // a vertex's (s,t) texture coordinates

        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

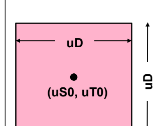
The texture coordinates need to come from the *vertex shader* because they were assigned to each *vertex* to begin with

A Little More Interesting, III:
Drawing a Pattern with the Fragment Shader

The fragment shader answers the question: *"Am I (the current fragment) inside the pattern or outside it?"*

A Fragment Shader is automatically called once per fragment:

```
#version 330 compatibility
uniform float uS0, uT0, uD;  // from your program
in vec2 vST;                 // from the vertex shader, interpolated through the rasterizer

void
main( )
{
        float s = vST.s;              // the s coordinate of where this fragment is
        float t = vST.t;              // the t coordinate of where this fragment is
        vec3 myColor = vec3( 1., 0.5, 0. );          // default color

        if(     uS0 - uD/2. <= s  &&  s <= uS0 + uD/2.  &&
                uT0 - uD/2. <= t  &&  t <= uT0 + uD/2.  )
        {
                        myColor = vec3( 1., 0., 0. );  // new pattern color
        }

        . . .
        gl_FragColor = << myColor with lighting applied >>
}
```



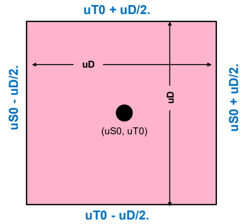
A Little More Interesting, IV: Drawing a Pattern with the Fragment Shader

25

The fragment shader answers the question: *"Am I (the current fragment) inside the pattern or outside it?"*

```
if( uS0 - uD/2. <= s && s <= uS0 + uD/2. && uT0 - uD/2. <= t && t <= uT0 + uD/2. )
{
        myColor = vec3( 1., 0., 0. );           // change the pattern color if inside the pattern boundaries
}
```

All 4 of these must be true to conclude this fragment is inside the pattern!



- uS0, uT0 are the center of the pattern in (0.-1.) texture coordinates
- uD is the size of the pattern in (0.-1.) texture coordinates

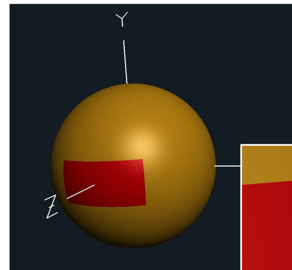
Oregon State University Computer Graphics

mjb – August 30, 2024

25

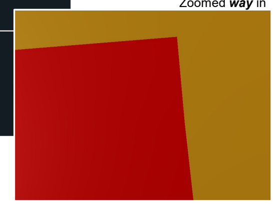
Drawing a Pattern on an Object

26



Here's the cool part: It doesn't matter (up to the limits of 32-bit floating-point precision) how far you zoom in. You still get a crisp edge. This is an advantage of procedural (equation-based) textures, as opposed to image-based textures.

Zoomed ***way*** in



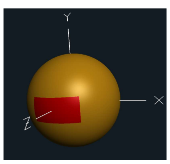
Oregon State University Computer Graphics

mjb – August 30, 2024

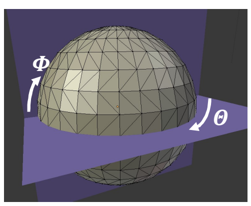
26

If the Equation Defines a *Square*, Why Does the Pattern Look Like a *Rectangle*?

27



$$s = \frac{\Theta - (-\pi)}{2\pi} \qquad t = \frac{\Phi - (-\frac{\pi}{2})}{\pi}$$



It's because, ***in a sphere***, the s coordinate encompasses twice as much angle (-180° → +180°) as the t coordinate does (-90° → +90°). So the same amount of "s" produces twice the distance as the same amount of "t". If you care, you can fix it like this:

```
float s = vST.s;
float t = vST.t;
s = 2.*s;
```

Oregon State University Computer Graphics

mjb – August 30, 2024

27

Per-Vertex Lighting vs. Per-Fragment Lighting

28

In **per-vertex lighting**, like we have done so far, we apply the lighting equation to the parameters at the vertices and then interpolate the color intensities in the rasterizer. This is what is built-in to standard OpenGL.

In **per-fragment lighting**, we will interpolate the parameters through the rasterizer first and then apply the lighting equation in the fragment shader. To do this, requires shaders.

Lighting Type	Vertex Shader	Rasterizer	Fragment Shader
Per-vertex	Apply lighting model to produce color intensities	Interpolate color intensities	Color the fragments
Per-fragment	Send parameters to rasterizer	Interpolate the parameters	Apply lighting model to color the fragments

Oregon State University Computer Graphics

mjb – August 30, 2024

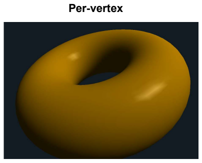
28

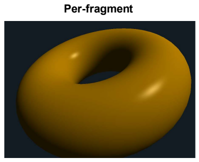
Per-Vertex Lighting vs. Per-Fragment Lighting

29

Per-vertex

Per-fragment





Oregon State University Computer Graphics

mjb – August 30, 2024

29

Applying Per-Fragment Lighting, I

30

Vertex shader:

```
#version 330 compatibility
out vec2  vST;                 // texture coords
out vec3  vN;                  // normal vector
out vec3  vL;                  // vector from point to light
out vec3  vE;                  // vector from point to eye

const vec3 LIGHTPOSITION = vec3(  5., 5., 0. );

void
main( )
{
        vST = gl_MultiTexCoord0.st;
        vec4 ECposition = gl_ModelViewMatrix * gl_Vertex;           // eye coordinate position
        vN = normalize( gl_NormalMatrix * gl_Normal );              // normal vector
        vL = LIGHTPOSITION - ECposition.xyz;                        // vector from the point to the light position
        vE = vec3( 0., 0., 0. ) - ECposition.xyz;                   // vector from the point to the eye position
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Rasterizer

Oregon State University Computer Graphics

mjb – August 30, 2024

30

Applying Per-Fragment Lighting, II

Fragment shader:

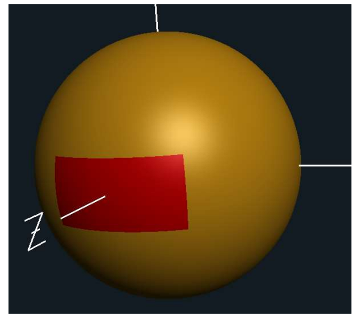
```
#version 330 compatibility
uniform float   uKa, uKd, uKs;            // coefficients of each type of lighting
uniform float   uShininess;               // specular exponent
in vec2   vST;                            // texture cords
in  vec3  vN;                                     // normal vector
in  vec3  vL;                             // vector from point to light
in  vec3  vE;                             // vector from point to eye
void main( )
{
        vec3 Normal = normalize(vN);
        vec3 Light    = normalize(vL);
        vec3 Eye      = normalize(vE);

        vec3 myColor =          vec3( 1.0, 0.5, 0.0 );        // default color
        vec3 mySpecularColor = vec3( 1.0, 1.0, 1.0 );         // specular highlight color

        << possibly change myColor >>
-----------------------------------------------------------------
        vec3 ambient = uKa * myColor;
        float d = 0.;
        float s = 0.
        if( dot(Normal,Light) > 0. )              // only do specular if the light can see the point
        {
                d = dot(Normal,Light);
                vec3 ref = normalize(  reflect( -Light, Normal )  );        // reflection vector
                s = pow( max( dot(Eye,ref),0.  ), uShininess );
        }
        vec3 diffuse    = uKd * d * myColor;
        vec3 specular = uKs * s * mySpecularColor;
        gl_FragColor = vec4( ambient + diffuse + specular,  1. );
}
```

Here's where we figure out what color this fragment will be, like before

Here's where we apply lighting to that color

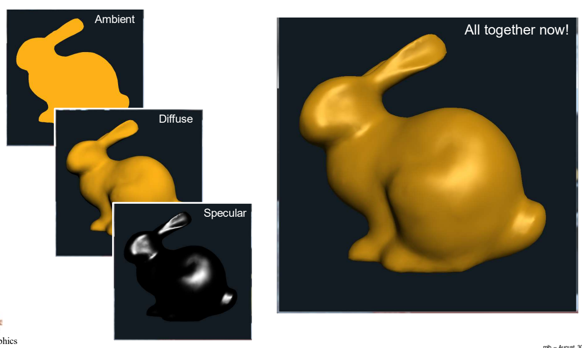
Oregon State University Computer Graphics

Applying Per-Fragment Lighting, III



Oregon State University Computer Graphics

Per-fragment Lighting is Good, Even Without a Pattern!



Oregon State University Computer Graphics

Setting up a Shader via the OpenGL API is somewhat Involved: Here is our C++ Class to Simplify the Shader Setup for You

First, follow these steps:

1. You will see two files that are already in your Sample folder: **glslprogram.h** and **glslprogram.cpp**
2. In your sample.cpp file, un-comment the line:
   #include "glslprogram.cpp"

> These two files have been reduced to have just the shader features you need for Project #6.
>
> If you are not working on Project #6, but are working on something bigger, I have more complete versions of glslprogram.h and glslprogram.cpp – just ask me.

Oregon State University Computer Graphics

Setting up a Shader via the OpenGL API is somewhat Involved: Here is our C++ Class to Simplify the Shader Setup for You

Put these in with the Global Variables:

```
GLSLProgram   Pattern;          // your VS+FS shader program name
float                Time;
#define MS_IN_THE_ANIMATION_CYCLE      10000
```

Oregon State University Computer Graphics

Setting up a Shader via the OpenGL API is somewhat Involved: Here is our C++ Class to Simplify the Shader Setup for You

Do this in Animate( ) like you've always done:

```
void
Animate( )
{
        int ms = glutGet( GLUT_ELAPSED_TIME );                    // milliseconds
        ms  %=  MS_IN_THE_ANIMATION_CYCLE;

        Time = (float)ms  /  (float)MS_IN_THE_ANIMATION_CYCLE;     // [ 0., 1. )
}
```

Oregon State University Computer Graphics

Setting up a Shader via the OpenGL API is somewhat Involved: Here is our C++ Class to Simplify the Shader Setup for You

37

Do this in InitGraphics( ) somewhere after where the window has been created and GLEW has been setup:

```
Pattern.Init( );
bool valid = Pattern.Create( "pattern.vert",  "pattern.frag" );

if( ! valid )
{
        fprintf( stderr, "Yuch!  The shader did not compile.\n" );
}
else
{
        fprintf( stderr, "Woo-Hoo!  The shader compiled.\n" );
}
```

2. A compiler for that language to create an executable
3. A way to see the compiler's error messages
4. A way to download the executable onto the external computer

In C/C++, the exclamation point (!) is pronounced "not".

This attempts to load, compile, and link the shader program. If something goes wrong, Pattern.Create( ) prints error messages into the console window and returns a value of ***valid=false***.

We cover the full GLSL API in CS 457/557

Setting up a Shader via the OpenGL API is somewhat Involved: Here is our C++ Class to Simplify the Shader Setup for You

38

Do this in Display( ):

```
float s0 = some function of Time
float t0  = some function of Time
float d   = some function of Time
        . . .
Pattern.Use( );         // turns the shader program on
                        // no more fixed-function – the shader Pattern now handles everything
                        // but the shader program just sits there idling until you draw something

Pattern.SetUniformVariable( "uS0", s0);
Pattern.SetUniformVariable( "uT0", t0 );
Pattern.SetUniformVariable( "uD",    d);

glCallList( SphereList );  // now the shader program has vertices and fragments to work on

Pattern.UnUse( );       // go back to fixed-function OpenGL
```

5. A way to run that executable on the external computer

6. A way to get information into the executable

Setting Up Texturing with Shaders

39

Graphics chips have functionality on them called ***Texture Units***. Each Texture Unit is identified by an integer number, typically 0-15, but oftentimes more.

To tell a shader how to get to a specific texture image, assign that texture into a specific **Texture Unit number** and then tell your shader what Texture Unit number to use. Your C/C++ code will look like this:

```
glActiveTexture( GL_TEXTURE5 );              // use texture unit 5
glBindTexture( GL_TEXTURE_2D, TexName );
```

The file gl.h has these lines:

```
#define GL_TEXTURE0        0x84C0
#define GL_TEXTURE1        0x84C1
#define GL_TEXTURE2        0x84C2
#define GL_TEXTURE3        0x84C3
#define GL_TEXTURE4        0x84C4
#define GL_TEXTURE5        0x84C5
#define GL_TEXTURE6        0x84C6
#define GL_TEXTURE7        0x84C7
#define GL_TEXTURE8        0x84C8
. . .
```


The Whole Process Looks Like This, I:

40

```
// globals:

unsigned char * Texture;
GLuint          TexName;
GLSLProgram     Pattern;

. . .
// In InitGraphics( ):

glGenTextures( 1, &TexName );
int nums, numt;
Texture = BmpToTexture( "filename.bmp", &nums, &numt );
glBindTexture( GL_TEXTURE_2D, TexName );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexImage2D(    GL_TEXTURE_2D, 0, 3, nums, numt, 0, 3, GL_RGB, GL_UNSIGNED_BYTE, Texture );

Pattern.Init( );
bool valid = Pattern.Create( "pattern.vert", "pattern.frag" );
If( !valid )
{
        . . .
}
```


The Whole Process Looks Like This, II:

41

This is the hardware Texture Unit Number. You can choose anything in the range 0-15.

```
. . .
// In Display( ):

Pattern.Use( );
glActiveTexture( GL_TEXTURE5 );          // your C++ program specifies that you want the texture to live on texture unit 5
glBindTexture( GL_TEXTURE_2D, TexName );
Pattern.SetUniformVariable( "uTexUnit", 5 );   // tell your shader program to find the texture on texture unit 5
        << draw something >>
Pattern.UnUse( );
```


2D Texturing within the Shaders

42

Vertex shader:

```
#version 330 compatibility
out  vec2  vST;

void
main( )
{
        vST = gl_MultiTexCoord0.st;
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

texture( ) **is a built-in texture map lookup function – it returns a vec4 (RGBA)**

Rasterizer

Fragment shader:

```
#version 330 compatibility
in  vec2  vST;
uniform sampler2D uTexUnit;

void
main( )
{
        vec3 newcolor = texture( uTexUnit, vST ).rgb;
        gl_FragColor = vec4( newcolor, 1. );
}
```

Pattern.SetUniformVariable("uTexUnit", 5);

Convert the vec4 rgba from the texture( ) call to just the vec3 rgb that we need

2D Texturing within the Shaders

43

Oregon State University
Computer Graphics

mjb – August 30, 2024

43

What if You Want to Use Two Textures in a Shader?

44

```
// In Display( ):

Pattern.Use( );
glActiveTexture( GL_TEXTURE5 );
glBindTexture( GL_TEXTURE_2D, TexName0 );

glActiveTexture( GL_TEXTURE6 );
glBindTexture( GL_TEXTURE_2D, TexName1 );

Pattern.SetUniformVariable( "uTexUnit0", 5 );
Pattern.SetUniformVariable( "uTexUnit1", 6 );

glCallList( … );

Pattern.UnUse( );
```

Fragment shader:

```
#version 330 compatibility
in  vec2  vST;
uniform sampler2D  uTexUnit0;
uniform sampler2D  uTexUnit1;

void
main( )
{
        vec3 newcolor0 = texture( uTexUnit0, vST ).rgb;
        vec3 newcolor1 = texture( uTexUnit1, vST ).rgb;
        gl_FragColor = …
}
```

Oregon State University
Computer Graphics

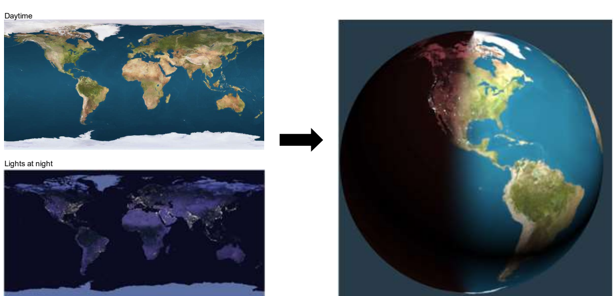
mjb – August 30, 2024

44

Why Would You Want to Use More Than One Texture in a Shader?

45

Once the RGBs have been read from a texture, they are just numbers. You can do any arithmetic you want with the texture RGBs, other colors, lighting, etc. Here is an example of blending two textures at once:



Computer Graphics

mjb – August 30, 2024

45

Why Would You Want to Use More Than One Texture in a Shader?

46

Textures used here:
- Day
- Night
- Heights (bump-mapping)
- Clouds
- Specular highlights

Visualization by Nick Gebbie

Oregon State University
Computer Graphics

mjb – August 30, 2024

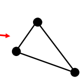
46

Something Goofy: Turning XYZs into RGBs in Model Coordinates

47

Vertex shader:

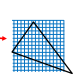
```
#version 330 compatibility
out  vec3  vColor;        per-vertex

void
main( )
{
        vec4 pos = gl_Vertex;
        vColor = pos.xyz;            // set rgb from xyz!
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```



Rasterizer

Fragment shader:

```
#version 330 compatibility
in  vec3  vColor;        per-fragment

void
main( )
{
        gl_FragColor = vec4( vColor,  1. );
}
```



Oregon State University
Computer Graphics

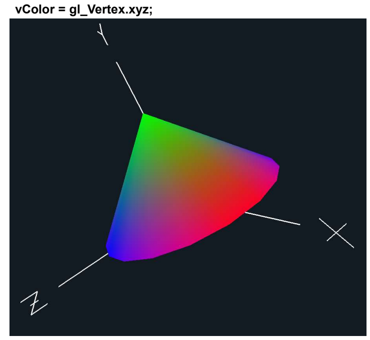
mjb – August 30, 2024

47

Setting rgb from the Untransformed xyz, I

48

vColor = gl_Vertex.xyz;



Oregon State University
Computer Graphics

mjb – August 30, 2024

48

Turning XYZs into RGBs in Eye (World) Coordinates

Vertex shader:

```
#version 330 compatibility
out vec3 vColor;

void
main( )
{
        vec4 pos = gl_ModelViewMatrix * gl_Vertex;
        vColor = pos.xyz;           // set rgb from xyz!
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Rasterizer

Fragment shader:

```
#version 330 compatibility
in vec3 vColor;

void
main( )
{
        gl_FragColor = vec4( vColor, 1. );
}
```

Oregon State University Computer Graphics

What's Different About These Two?

Set the color from the ***untransformed (MC) xyz***

```
#version 330 compatibility
out vec3 vColor;

void
main( )
{
        vec4 pos = gl_Vertex;
        vColor = pos.xyz;           // set rgb from xyz!
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Set the color from the ***transformed (WC/EC) xyz***:

```
#version 330 compatibility
out vec3 vColor;

void
main( )
{
        vec4 pos = gl_ModelViewMatrix * gl_Vertex;
        vColor = pos.xyz;           // set rgb from xyz!
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

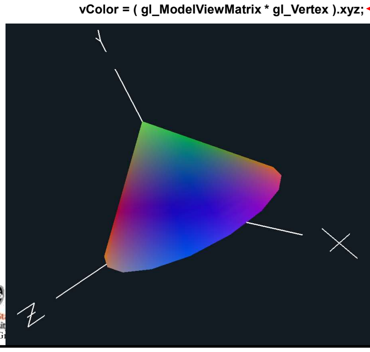
Oregon State University Computer Graphics

Setting rgb from the Transformed xyz, II

vColor = (gl_ModelViewMatrix * gl_Vertex).xyz;

Note: the phrase ".xyz" and the phrase ".rgb" mean exactly the same thing: "give me the first 3 numbers from this vec variable".

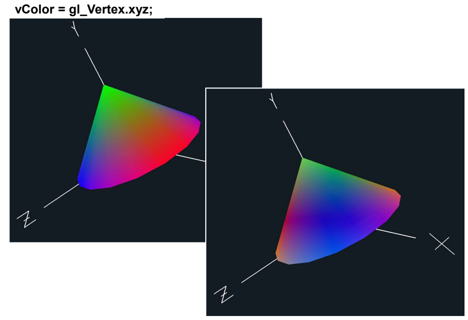
What you can't do is mix them, such as ".xgz"



Oregon State University Computer Graphics

Setting rgb From xyz

vColor = gl_Vertex.xyz;



vColor = (gl_ModelViewMatrix * gl_Vertex).xyz;

Oregon State University Computer Graphics

Hints on Running Shaders on Your Own System

- You need a graphics system that is OpenGL 2.0 or later. Basically, if you got your graphics system in the last 5 years, you should be OK, unless it came from Apple. In that case, who knows how much OpenGL support it has? (The most recent OpenGL level is 4.6)
- Update your graphics driver to the most recent version!
- Do the GLEW setup if you are on Windows. It looks like this in the sample code:

```
GLenum err = glewInit( );
if( err != GLEW_OK )
{
        fprintf( stderr, "glewInit Error\n" );
}
else
        fprintf( stderr, "GLEW initialized OK\n" );
```

  This must come ***after you've created a graphics window***. (It is this way in the sample code, but I'm saying this because I know some of you go in and "simplify" my sample code by deleting everything you don't think you need.)

- You use the GLSL C++ class you've been given ***only after a window has been created and GLEW has been setup***. Only then can you initialize your shader program:

```
Pattern.Init( );
bool valid = Pattern.Create( "pattern.vert", "pattern.frag" );
```


A Common Error to Look Out For

Here is a piece of code:

```
#version 330 compatibility
out vec3 vColor;

void
main( )
{
        vec4 pos = gl_Vertex;
        vec3 vColor = pos.xyz;           // set rgb from xyz!
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Abandon hope, all ye who do this

It looks like our example from earlier in these notes. It compiles OK. It should work, right?

Wrong! By re-declaring vColor in "vec3 vColor = pos.xyz", you are making a *local version* of vColor and writing pos.xyz into that local version, not the out variable! The out version of vColor is never getting written to, and so the vColor in the fragment shader will have no sensible value.

Don't ever re-declare in, out, or uniform variables!

Trust me, you will do this sometime. It's an easy mistake to make mindlessly. I do it every so often myself.

Oregon State University Computer Graphics

Differences if You are on a Mac

Unfortunately, Apple froze their GLSL support at version 1.20 – here is how to adapt to that:

- Your shader version number should be 120 (at the top of the .vert and .frag files):
  `#version 120 compatibility`
- Instead of the keywords **in** and **out**, use ***varying***
- Your OpenGL includes will need to look like this:
  #include <OpenGL/gl.h>
  #include <OpenGL/glu.h>
- You don't need to do anything with GLEW
- Your compile sequence will look like this:
  g++ -framework OpenGL -framework GLUT sample.cpp -o sample -Wno-deprecated

Guide to Where to Put Pieces of Your Shader Code, I

1. Declare the GLSLProgram above the main program (i.e., as a global):

```
GLSLProgram   Pattern;
```

2. At the end of InitGraphics( ), create the shader program and setup your shaders:

```
Pattern.Init( );
bool valid = Pattern.Create( "pattern.vert", "pattern.frag" );
if( ! valid ) { . . . }
```

3. Turn on the shader program in Display( ), set shader uniform variables, draw the objects, then turn off the shader program:

```
Pattern.Use( );

Pattern.SetUniformVariable( ...

glCallList( SphereList( );

Pattern.UnUse( );          // return to the fixed function pipeline
```

4. When you run your program, be sure to check the console window for shader compilation errors!

Guide to Where to Put Pieces of Your Shader Code, II

Tips on drawing the object:

- If you want to key off of s and t coordinates in your shaders, the object must *have* s and t coordinates (vt) assigned to its vertices – *not all OBJ files do*!
- If you want to use surface normals in your shaders, the object must *have* surface normals (vn) assigned to its vertices – *not all OBJ files do*!
- Be sure you explicitly assign *all* of your uniform variables – no error messages occur if you forget to do this – it just quietly screws up.
- The glutSolidTeapot( ) has been textured in patches, like a quilt – cute, but weird
- The **OsuSphere( )** function from the texturing project will give you a very good sphere. Use it, not the GLUT sphere.