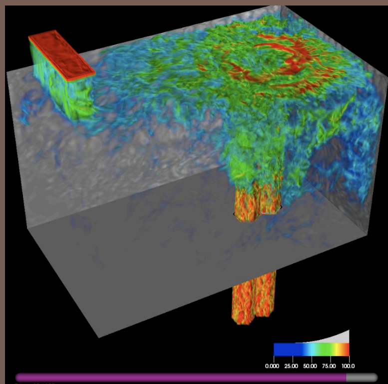
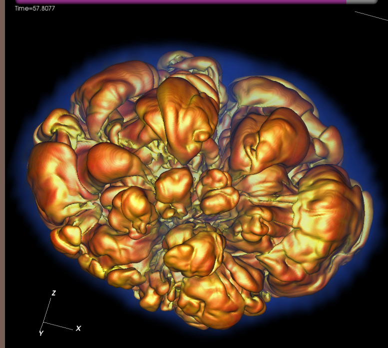
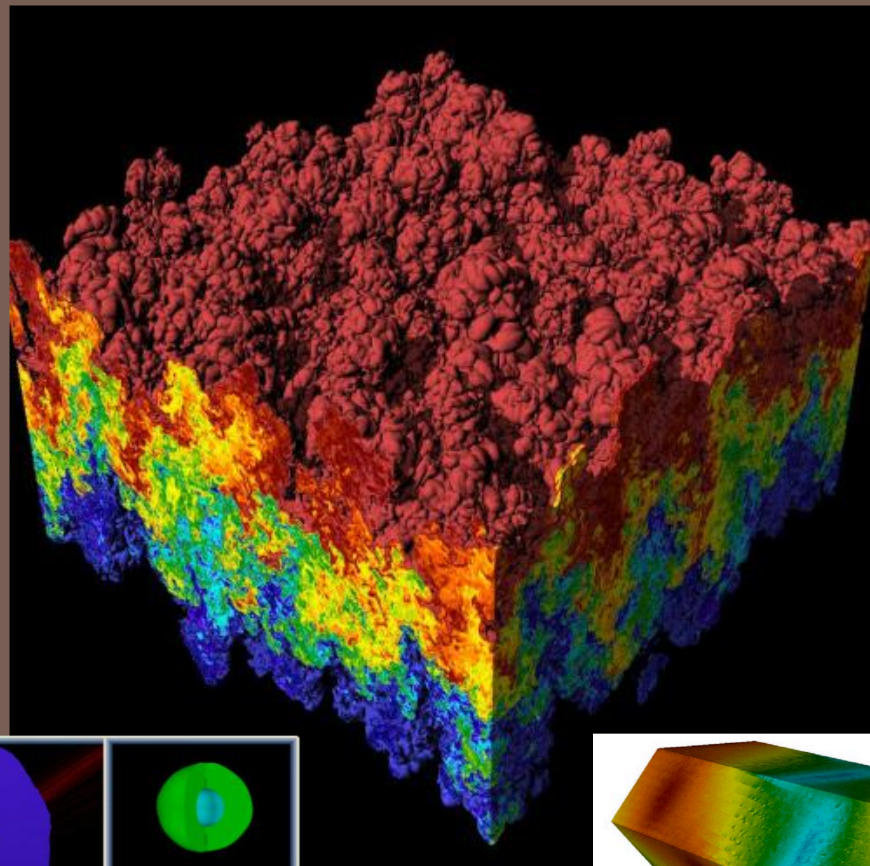
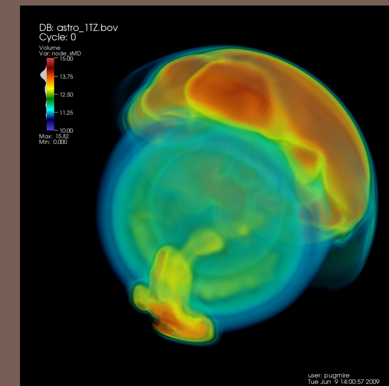
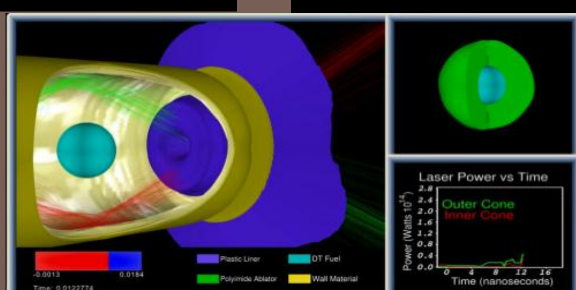
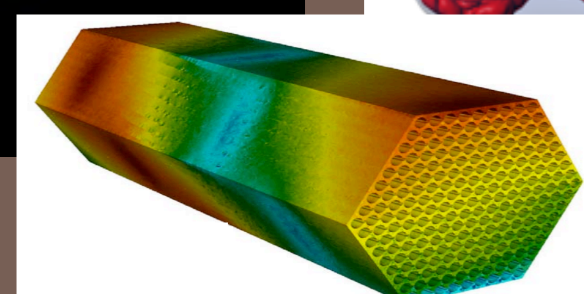
CIS 441/541: Introduction to Computer Graphics

Lecture 15: shaders



March 12th, 2019

Hank Childs, University of Oregon

Talk _more_ about the test

Project G

- Blender
- WebGL #1
- WebGL #2
- CUDA
- Bezier
- Computer Vision
- Shaders

Schedule

- Upcoming:
  - Today: shaders, live code
  - Thursday: test re-take

Late Passes

- Will bring forms to final

Hank OH

- Primary purpose of Hank's OH is now to help with self-defined projects
- Also can help with 1A-1F,2A,2B
- Friday OH: 11am-12noon

Final Presentations

- 3 minutes each
- Make sure to make it clear what you did
- Try to impress judges
  - What is cool about what you did?
- Format
  - PowerPoint
  - Demo
  - PowerPoint + Demo
- Connect A/V to Rm 220 project before 9am as test

Shaders

Shaders

- Shader: computer program used to do “shading”
- “Shading”: general term that covers more than just shading/lighting
  - Used for many special effects
- Increased control over:
  - position, hue, saturation, brightness, contrast
- For:
  - pixels, vertices, textures

Motivation: Bump Mapping

- Idea:
  - typical rasterization, calculate fragments
  - fragments have normals (as per usual)
  - also interpolate texture on geometry & fragments
    - use texture for "bumps"
    - take normal for fragment and displace it by "bump" from texture

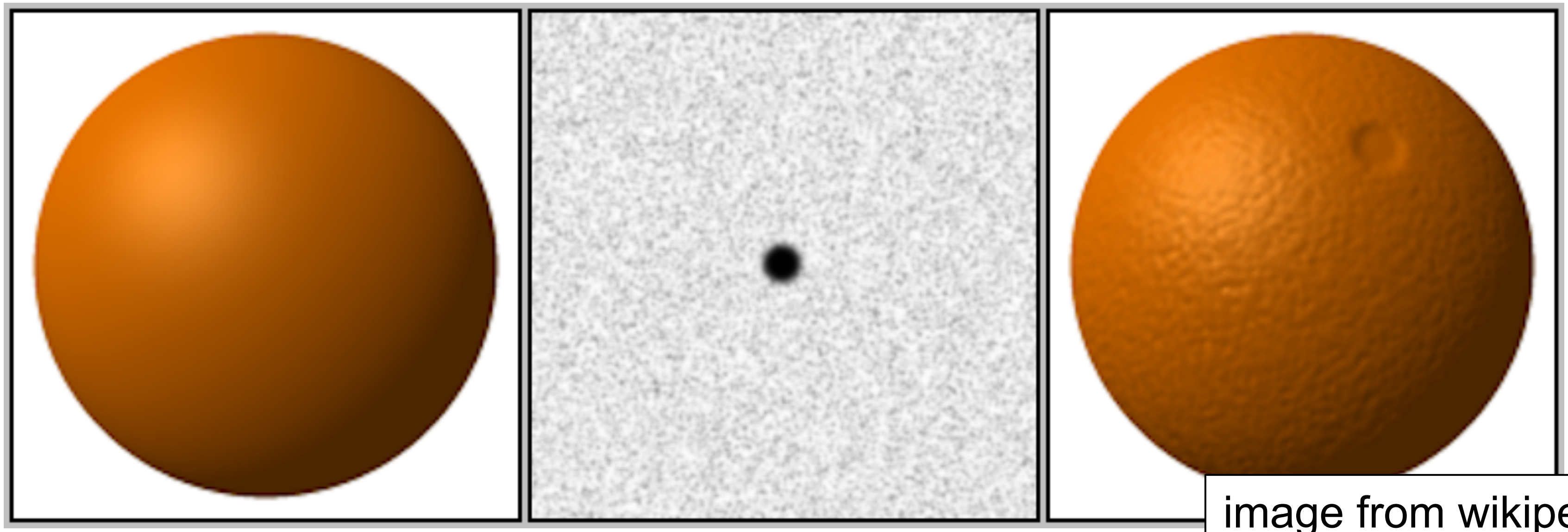
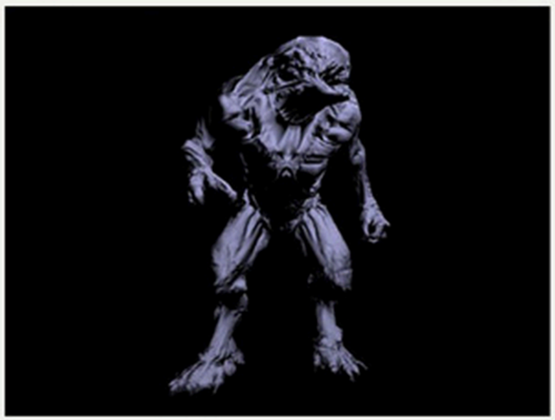


image from wikipedia

Bump Mapping Example

Concept

BumpMapping allows designers to express their creativity through a 100,000+ polygons creature. Once art is done, a low poly model (5000 polygons) is automatically generated along with a normal map.

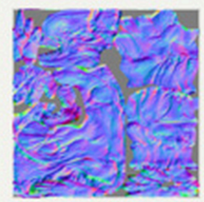


100,000+ polygons

=

5,000 polygons

+

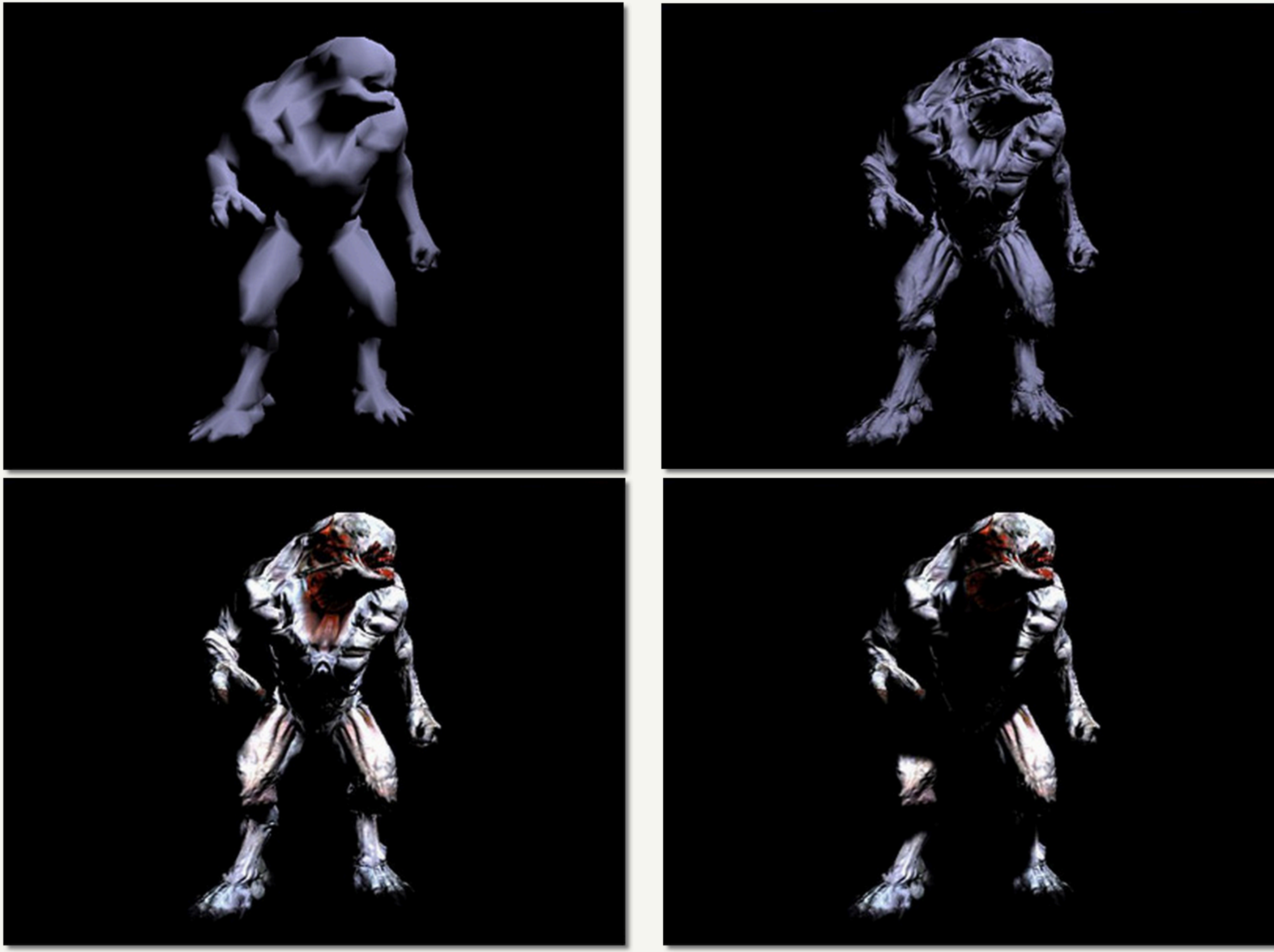


Normal map

At runtime, details are added back by combining the low model with the normal map.

credit: http://www.fabiensanglard.net/bumpMapping/

Results



How to do Bump Mapping?

- Answer: easy to imagine doing it in your Project 1A-1F infrastructure
  - You have total control
- But what OpenGL commands would do this?
  - Not possible in V1 of the GL interface, which is what we have learned
- It is possible with various extensions to OpenGL
  - We will learn to do this with shaders

Shading Languages

- shading language: programming language for graphics, specifically shader effects
- Benefits: increased flexibility with rendering
- OpenGL (as we know it so far): fixed transformations for color, position, of pixels, vertices, and textures.
- Shader languages: custom programs, custom effects for color, position of pixels, vertices, and textures.

ARB assembly language

- ARB: low-level shading language
  - at same level as assembly language
- Created by OpenGL Architecture Review Board (ARB)
- Goal: standardize instructions for controlling GPU
- Implemented as a series of extensions to OpenGL
- You don't want to work at this level, but it was an important development in terms of establishing foundation for today's technology

GLSL: OpenGL Shading Language

- GLSL: high-level shading language
  - also called GLSLang
  - syntax similar to C
- Purpose: increased control of graphics pipeline for developers, but easier than assembly
  - This is layer where developers do things like "bump mapping"
- Benefits:
  - Benefits of GL (cross platform: Windows, Mac, Linux)
  - Support over GPUs (NVIDIA, ATI)
  - HW vendors support GLSL very well

Other high-level shading languages

- Cg (C for Graphics)
  - based on C programming language
  - outputs DirectX or OpenGL shader programs
  - deprecated in 2012
- HLSL (high-level shading language)
  - used with MicroSoft Direct3D
  - analogous to GLSL
  - similar to CG
- RSL (Renderman Shading Language)
  - C-like syntax
  - for use with Renderman: Pixar's rendering engine

Relationship between GLSL and OpenGL

Versions [edit]

GLSL versions have evolved alongside specific versions of the OpenGL API. It is only with OpenGL versions 3.3 and above that the GLSL and OpenGL major and minor version numbers match. These versions for GLSL and OpenGL are related in the following table:

GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59 [1]	2.0	April 2004	#version 110
1.20.8[2]	2.1	September 2006	#version 120
1.30.10[3]	3.0	August 2008	#version 130
1.40.08[4]	3.1	March 2009	#version 140
1.50.11[5]	3.2	August 2009	#version 150
3.30.6[6]	3.3	February 2010	#version 330
4.00.9[7]	4.0	March 2010	#version 400
4.10.6[8]	4.1	July 2010	#version 410
4.20.11[9]	4.2	August 2011	#version 420
4.30.8[10]	4.3	August 2012	#version 430
4.40[11]	4.4	July 2013	#version 440
4.50[12]	4.5	August 2014	#version 450

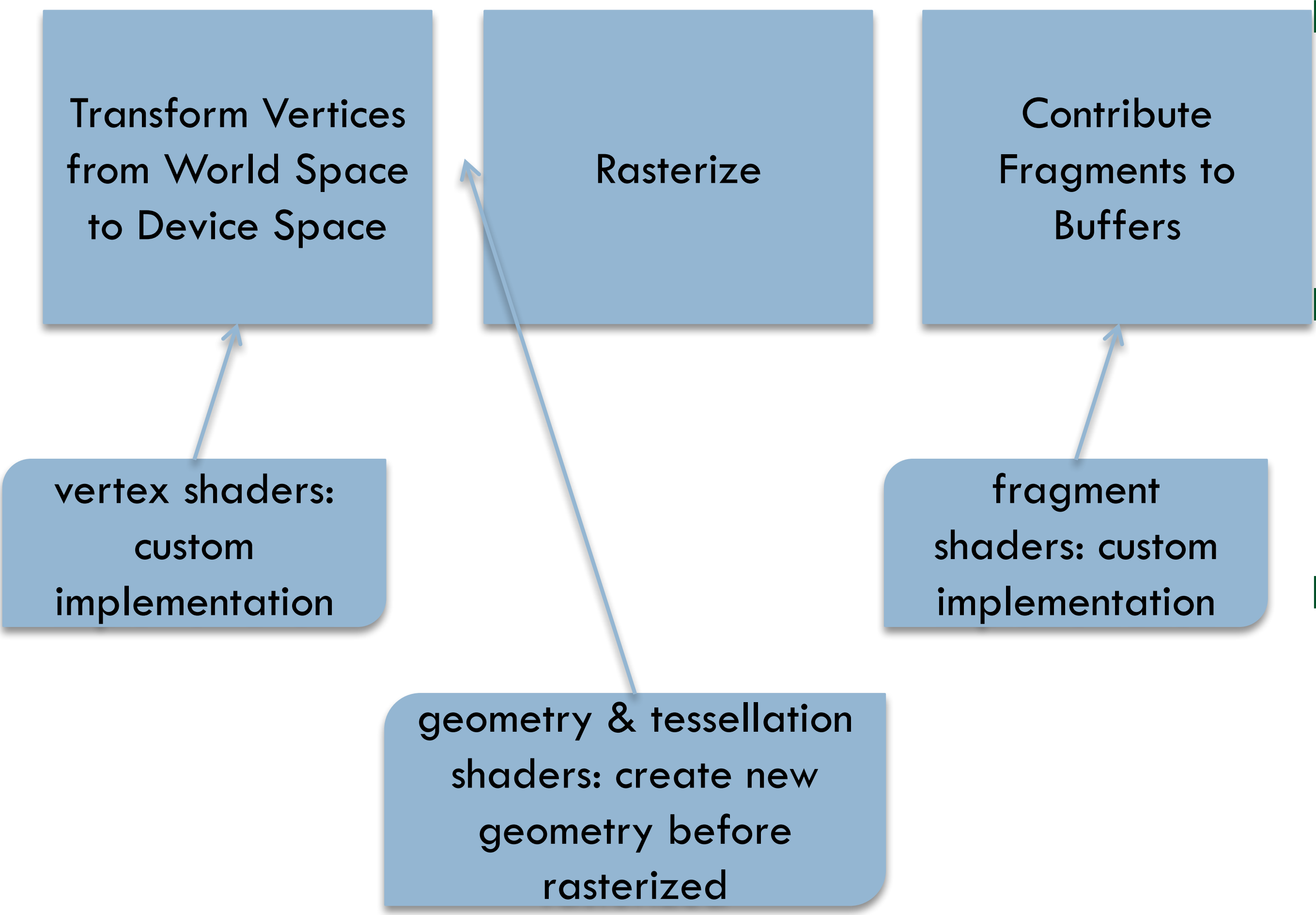
Source: wikipedia

4 Types of Shaders

- Vertex Shaders
- Fragment Shaders
- Geometry Shaders
- Tessellation Shaders

- It is common to use multiple types of shaders in a program and have them interact.

How Shaders Fit Into the Graphics Pipeline



- You can have 0 or 1 of each shader type
- Vertex & fragment: very common
- Geometry & tessellation: less common
  - adaptive meshing

Vertex Shader

- Run once for each vertex
- Can: manipulate position, color, texture
- Cannot: create new vertices
- Primary purpose: transform from world-space to device-space (+ depth for z-buffer).
  - However: A vertex shader replaces the transformation, texture coordinate generation and lighting parts of OpenGL, and it also adds texture access at the vertex level
- Output goes to geometry shader or rasterizer

Geometry Shader

- Run once for each geometry primitive
- Purpose: create new geometry from existing geometry.
- Output goes to rasterizer
- Examples: glyphing, mesh complexity modification
- Formally available in GL 3.2, but previously available in 2.0+ with extensions

- Tessellation Shader: doing some of the same things
- Available in GL 4.0

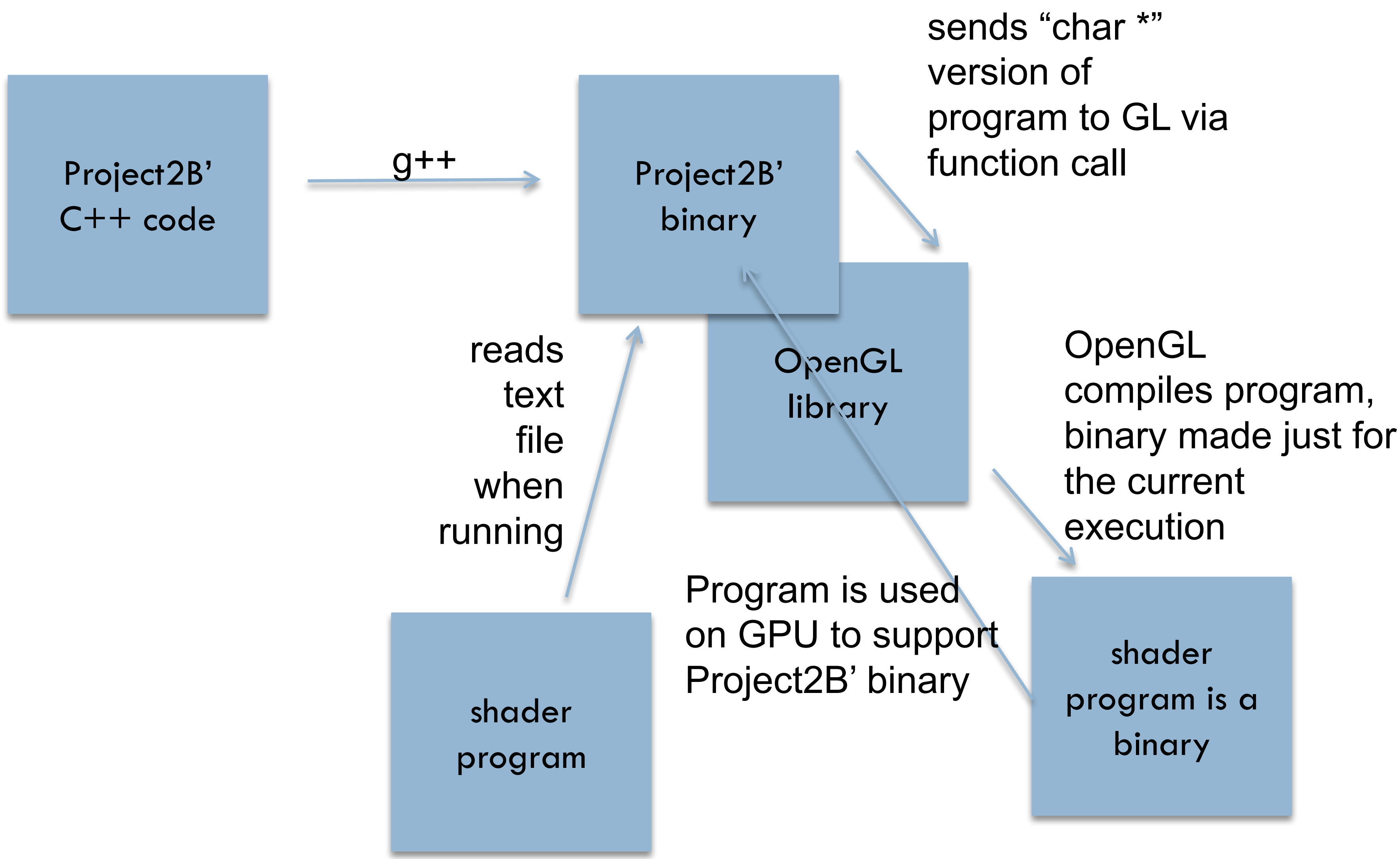
Fragment Shader

- Run once for each fragment
- Purpose: replaces the OpenGL 1.4 fixed-function texturing, color sum and fog stages
- Output goes to buffers
- Example usages: bump mapping, shadows, specular highlights
- Can be very complicated: can sample surrounding pixels and use their values (blur, edge detection)
- Also called pixel shaders

How to Use Shaders

- You write a shader program: a tiny C-like program
- You write C/C++ code for your application
- Your application loads the shader program from a text file
- Your application sends the shader program to the OpenGL library and directs the OpenGL library to compile the shader program
- If successful, the resulting GPU code can be attached to your (running) application and used
- It will then supplant the built-in GL operations

How to Use Shaders: Visual Version



Compiling Shader

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);
```

Compiling Shader: inspect if it works

```
if(isCompiledVS == GL_FALSE)
{
   cerr << "Did not compile VS" << endl;

   GLint maxLength = 0;
   glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

   // The maxLength includes the NULL character
   std::vector<GLchar> errorLog(maxLength);
   glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
   cerr << "Vertex shader log says " << &(errorLog[0]) << endl;
   exit(EXIT_FAILURE);
}
```

Compiling Multiple Shaders

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);

if(isCompiledVS == GL_FALSE)
{
   cerr << "Did not compile VS" << endl;

   GLint maxLength = 0;
   glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

   // The maxLength includes the NULL character
   std::vector<GLchar> errorLog(maxLength);
   glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
   cerr << "Vertex shader log says " << &(errorLog[0]) << endl;
   exit(EXIT_FAILURE);
}

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
std::string fragmentProgram = loadFileToString("fs.glsl");
const char *fragment_shader_source = fragmentProgram.c_str();
GLint const fragment_shader_length = strlen(fragment_shader_source);
glShaderSource(fragmentShader, 1, &fragment_shader_source, &fragment_shader_length);
glCompileShader(fragmentShader);
GLint isCompiledFS = 0;
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &isCompiledFS);
```

Attaching Shaders to a Program

```
GLuint program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);

glLinkProgram(program);

glDetachShader(program, vertexShader);
glDetachShader(program, fragmentShader);
```

Inspecting if program link worked…

```
GLint isLinked = 0;
glGetProgramiv(program, GL_LINK_STATUS, (int *)&isLinked);
if(isLinked == GL_FALSE)
{
   GLint maxLength = 0;
   glGetProgramiv(program, GL_INFO_LOG_LENGTH, &maxLength);

   //The maxLength includes the NULL character
   std::vector<GLchar> infoLog(maxLength);
   glGetProgramInfoLog(program, maxLength, &maxLength, &infoLog[0]);
   cerr << "Couldn't link" << endl;
   cerr << "Log says " << &(infoLog[0]) << endl;

   exit(EXIT_FAILURE);
}
```

BUT: this doesn't work in VTK…

- VTK has its own shader handling, and it doesn't play well with the GL calls above…

```
vtkSmartPointer<vtkShaderProgram2> pgm = vtkShaderProgram2::New();
pgm->SetContext(renWin);

vtkSmartPointer<vtkShader2> vertexShader=vtkShader2::New();
vertexShader->SetType(VTK_SHADER_TYPE_VERTEX);
std::string vertexProgram = loadFileToString("v_vs.glsl");
vertexShader->SetSourceCode(vertexProgram.c_str());
vertexShader->SetContext(pgm->GetContext());

pgm->GetShaders()->AddItem(vertexShader);

vtkSmartPointer<vtkShader2> fragmentShader=vtkShader2::New();
fragmentShader->SetType(VTK_SHADER_TYPE_FRAGMENT);
std::string fragmentProgram = loadFileToString("v_fs.glsl");
fragmentShader->SetSourceCode(fragmentProgram.c_str());
fragmentShader->SetContext(pgm->GetContext());

pgm->GetShaders()->AddItem(fragmentShader);

((vtkOpenGLProperty*)win3Actor->GetProperty())->SetPropProgram(pgm);
```

note: VTK6.1 much better for shaders than 6.0

Simplest Vertex Shader

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

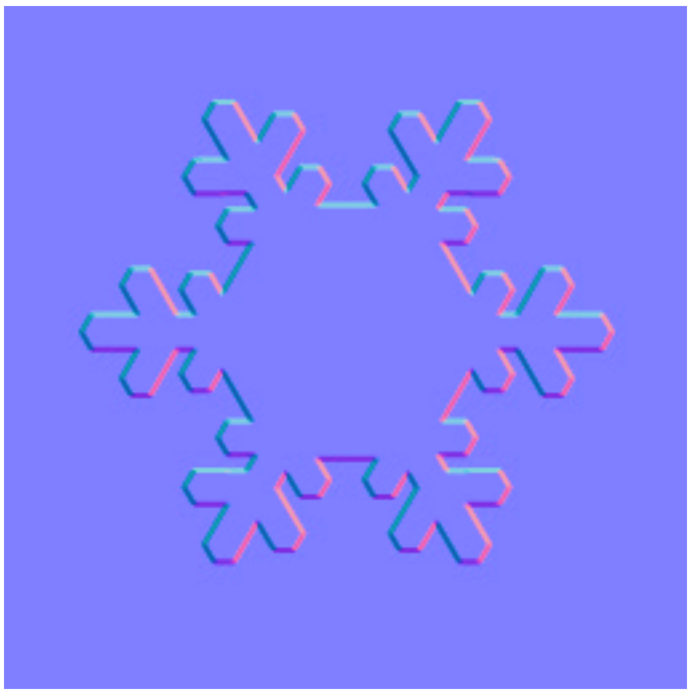
Many built-in variables.
Some are input.
Some are required output (gl_Position).

Simplest Vertex Shader (VTK version)

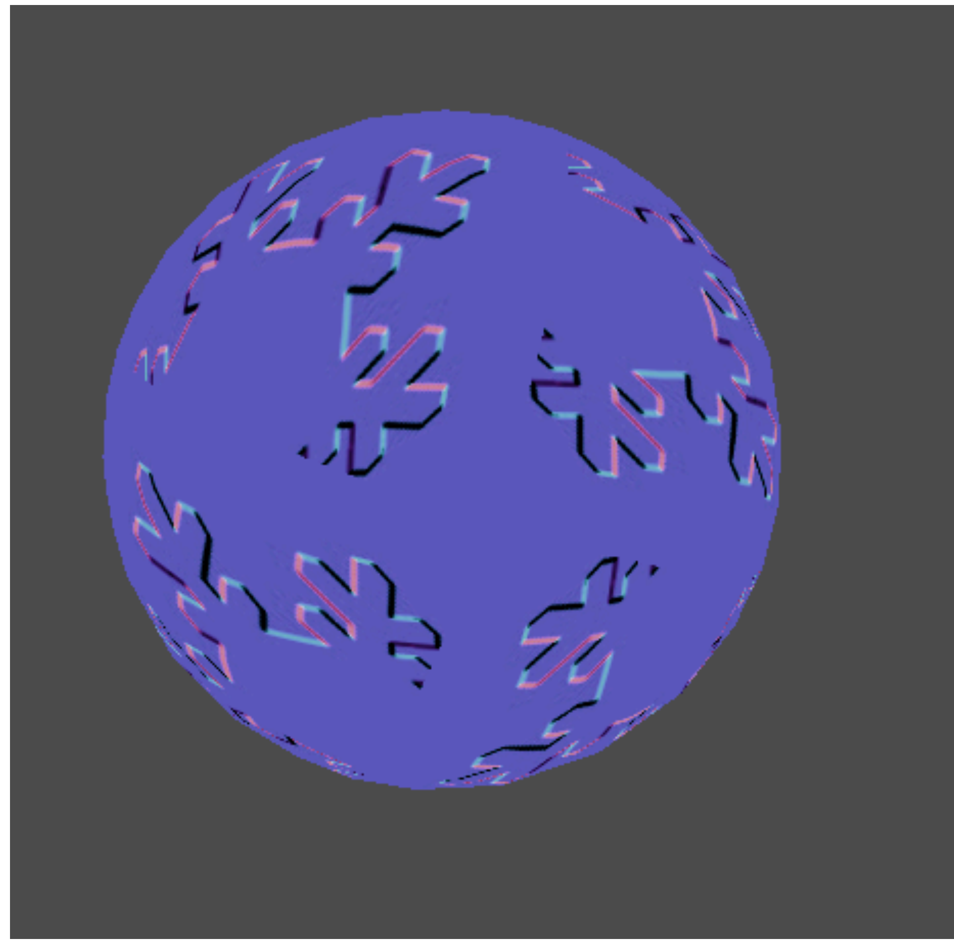
```
void propFuncVS(void)
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

VTK uses special names
    propFuncVS: vertex shader
    propFuncFS: fragment shader
somehow it changes these into “main” just in time…

Bump-mapping with GLSL



bump map texture



output

Will need to load a texture…

```
// from swiftless.com
GLuint LoadTexture( const char * filename, int width, int height )
{
    GLuint texture;
    unsigned char * data;
    FILE * file;

    //The following code will read in our RAW file
    file = fopen( filename, "rb" );

    if ( file == NULL ) return 0;
    data = (unsigned char *)malloc( width * height * 3 );
    fread( data, width * height * 3, 1, file );

    fclose( file );

    glGenTextures( 1, &texture ); //generate the texture with the loaded data
    glBindTexture( GL_TEXTURE_2D, texture ); //bind the texture to it's array

    glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE ); //set texture environment parameters

    //And if you go and use extensions, you can use Anisotropic filtering textures which are of an
    //even better quality, but this will do for now.
    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    //Here we are setting the parameter to repeat the texture instead of clamping the texture
    //to the edge of our shape.
    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );

    //Generate the texture
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);

    free( data ); //free the texture

    return texture; //return whether it was successfull
}
```

Need to put 2D textures on our triangles…

```
class Triangle
{
  public:
      double       X[3];
      double       Y[3];
      double       Z[3];
      double       Tu[3];
      double       Tv[3];
};
```

```
void DrawSphere()
{
    int recursionLevel = 3;
    Triangle t;
    t.X[0] = 1;
    t.Y[0] = 0;
    t.Z[0] = 0;
    t.Tu[0] = 0;
    t.Tv[0] = 0;
    t.X[1] = 0;
    t.Y[1] = 1;
    t.Z[1] = 0;
    t.Tu[1] = 1;
    t.Tv[1] = 0;
    t.X[2] = 0;
    t.Y[2] = 0;
    t.Z[2] = 1;
    t.Tu[2] = 1;
    t.Tv[2] = 1;
    std::vector<Triangle> list;
    list.push_back(t);
    for (int r = 0 ; r < recursionLevel ; r++)
    {
        list = SplitTriangle(list);
    }
```

```
std::vector<Triangle> SplitTriangle(std::vector<Triangle> &list)
{
    std::vector<Triangle> output(4*list.size());
    for (unsigned int i = 0 ; i < list.size() ; i++)
    {
        double mid1[5], mid2[5], mid3[5];
        mid1[0] = (list[i].X[0]+list[i].X[1])/2;
        mid1[1] = (list[i].Y[0]+list[i].Y[1])/2;
        mid1[2] = (list[i].Z[0]+list[i].Z[1])/2;
        mid1[3] = (list[i].Tu[0]+list[i].Tu[1])/2;
        mid1[4] = (list[i].Tv[0]+list[i].Tv[1])/2;
        mid2[0] = (list[i].X[1]+list[i].X[2])/2;
        mid2[1] = (list[i].Y[1]+list[i].Y[2])/2;
        mid2[2] = (list[i].Z[1]+list[i].Z[2])/2;
        mid2[3] = (list[i].Tu[1]+list[i].Tu[2])/2;
        mid2[4] = (list[i].Tv[1]+list[i].Tv[2])/2;
        mid3[0] = (list[i].X[0]+list[i].X[2])/2;
        mid3[1] = (list[i].Y[0]+list[i].Y[2])/2;
        mid3[2] = (list[i].Z[0]+list[i].Z[2])/2;
        mid3[3] = (list[i].Tu[0]+list[i].Tu[2])/2;
        mid3[4] = (list[i].Tv[0]+list[i].Tv[2])/2;
        output[4*i+0].X[0] = list[i].X[0];
        output[4*i+0].Y[0] = list[i].Y[0];
        output[4*i+0].Z[0] = list[i].Z[0];
        output[4*i+0].Tu[0] = list[i].Tu[0];
        output[4*i+0].Tv[0] = list[i].Tv[0];
        output[4*i+0].X[1] = mid1[0];
        output[4*i+0].Y[1] = mid1[1];
        output[4*i+0].Z[1] = mid1[2];
        output[4*i+0].Tu[1] = mid1[3];
        output[4*i+0].Tv[1] = mid1[4];
        output[4*i+0].X[2] = mid3[0];
        output[4*i+0].Y[2] = mid3[1];
        output[4*i+0].Z[2] = mid3[2];
        output[4*i+0].Tu[2] = mid3[3];
        output[4*i+0].Tv[2] = mid3[4];
        output[4*i+1].X[0] = list[i].X[1];
        output[4*i+1].Y[0] = list[i].Y[1];
        output[4*i+1].Z[0] = list[i].Z[1];
        output[4*i+1].Tu[0] = list[i].Tu[1];
        output[4*i+1].Tv[0] = list[i].Tv[1];
```

Need to set up shaders and textures…

```
vtkSmartPointer<vtkShaderProgram2> pgm = vtkShaderProgram2::New();
pgm->SetContext(renWin);

vtkSmartPointer<vtkShader2> vertexShader=vtkShader2::New();
vertexShader->SetType(VTK_SHADER_TYPE_VERTEX);
//std::string vertexProgram = loadFileToString("vs.glsl");
std::string vertexProgram = loadFileToString("v_vs.glsl");
vertexShader->SetSourceCode(vertexProgram.c_str());
vertexShader->SetContext(pgm->GetContext());

pgm->GetShaders()->AddItem(vertexShader);

vtkSmartPointer<vtkShader2> fragmentShader=vtkShader2::New();
fragmentShader->SetType(VTK_SHADER_TYPE_FRAGMENT);
//std::string fragmentProgram = loadFileToString("light_fs.glsl");
std::string fragmentProgram = loadFileToString("v_fs.glsl");
fragmentShader->SetSourceCode(fragmentProgram.c_str());
fragmentShader->SetContext(pgm->GetContext());

pgm->GetShaders()->AddItem(fragmentShader);

((vtkOpenGLProperty*)win3Actor->GetProperty())->SetPropProgram(pgm);
win3Actor->GetProperty()->ShadingOn();

GLuint texture = LoadTexture("normal_map.raw", 256, 256);
glEnable(GL_TEXTURE_2D);
int texture_location = glGetUniformLocation(fragmentShader->GetId(), "color_texture");
glUniform1i(texture_location, 0);
glBindTexture(GL_TEXTURE_2D, texture);
```

So what is the vertex shader program?...

```
void propFuncVS(void)
{
   gl_TexCoord[0] = gl_MultiTexCoord0;

   // Set the position of the current vertex
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

And what is the fragment shader program?...

```
uniform sampler2D color_texture;
uniform sampler2D normal_texture;

void propFuncFS(void)
{
   // Extract the normal from the normal map
   vec3 normal = normalize(texture2D(normal_texture, gl_TexCoord[0].st).rgb * 2.0 - 1.0);

   // Determine where the light is positioned (this can be set however you like)
   vec3 light_pos = normalize(vec3(1.0, 1.0, 1.5));

   // Calculate the lighting diffuse value
   float diffuse = max(dot(normal, light_pos), 0.0);

   vec3 color = diffuse * texture2D(color_texture, gl_TexCoord[0].st).rgb;

   // Set the output color of our current pixel
   gl_FragColor = vec4(color, 1.0);
}
```