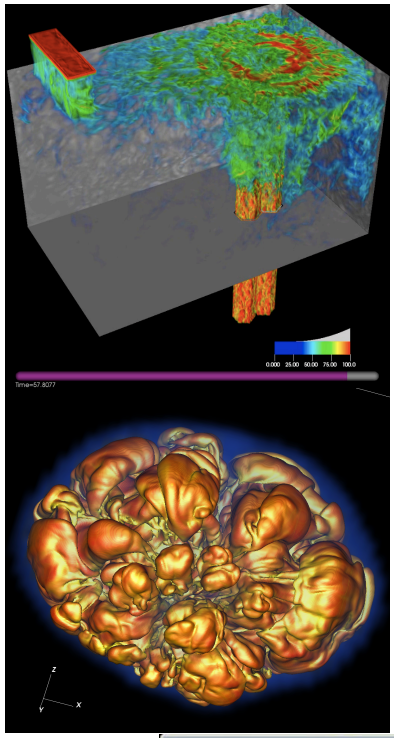
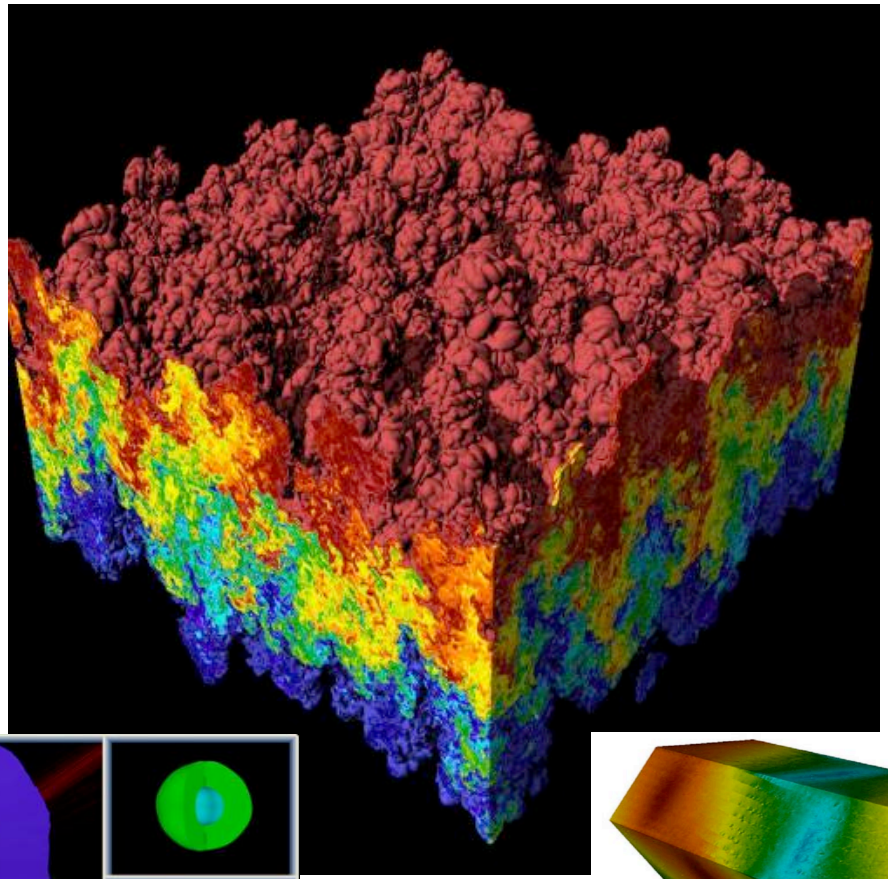
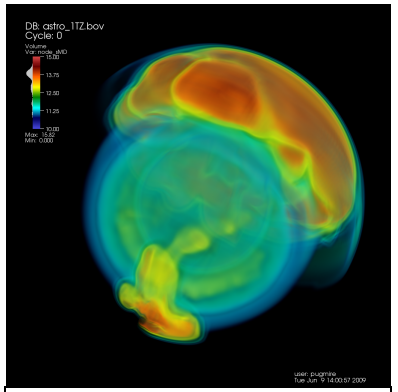
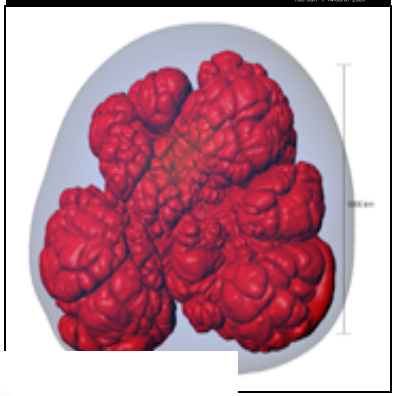
CIS 441/541: Intro to Computer Graphics

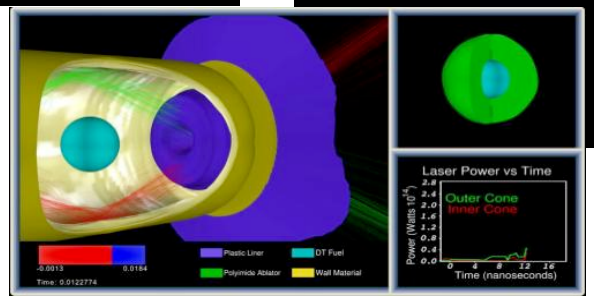
Lecture 8: Finish Shading, Intro OpenGL

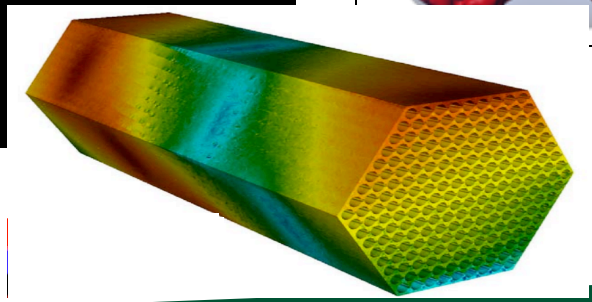












April 29, 2021

Hank Childs, University of Oregon

Class Plan

- Abhishek and I are working hard on preparing Project 2 (OpenGL)
  - ~~Quite frankly not going as well as we thought~~
  - Things are going better!
  - Grading has slowed down
- Projects will start coming faster
  - Want there to time to do great final projects
- 1E, 1F: simpler coding, harder concepts

Class Plan (New Slide)

- Option A:
  - Slow class down, more time on OpenGL, less time for final project
- Option B:
  - Keep up the pace, less time on OpenGL (but make sure we do OpenGL!), more time for final project

Current Plan (1/2)

Week	Sun	Mon	Tues	Weds	Thurs	Fri	Sat
5			Lec 7 (shading), ~~1F assigned,~~ 1E due		Lec 8 (finish shading, GL), ~~2A assigned~~ 1F assigned		
6		~~1F due~~	Lec 9 (GL), ~~2B assigned~~ 2A assigned	1F due	2B assigned Discussion of final projects / Quiz 3		~~2A due~~
7		2A due	Lec 11 – ray tracing		More discussion of final projects (?)	~~2B due~~	

Current Plan (2/2)

- Weeks 8-10 → you work on final projects
- Lectures will be on misc. topics in graphics, esp. in support of final projects
- Quiz 3 (Week 6): likely on matrices
- Quiz 4 (Week 8): likely on GL
- Quiz 5 (Week 10): likely on topics in final weeks

Office Hours

Published | Edit

How to access Office Hours

Hank Childs

Apr 4 at 2:02pm

All Sections

Hi Everyone,

We currently have an asymmetry for accessing Hank and Abhishek's Office Hours.

As of now, Abhishek's are always at: COVERED UP (THIS IS POSTED ONLINE)

And Hank's are accessible via the Zoom Meetings area in Canvas.

Let's chat on Tuesday about the most standard way to do this.

Finally, here is the OH schedule again:

Monday (Abhishek): 10am-11am
Tuesday (Abhishek): 945am-1045am
Wednesday (Hank): 230pm-330pm
Thursday (Abhishek): 945am-1045am

Best,
Hank

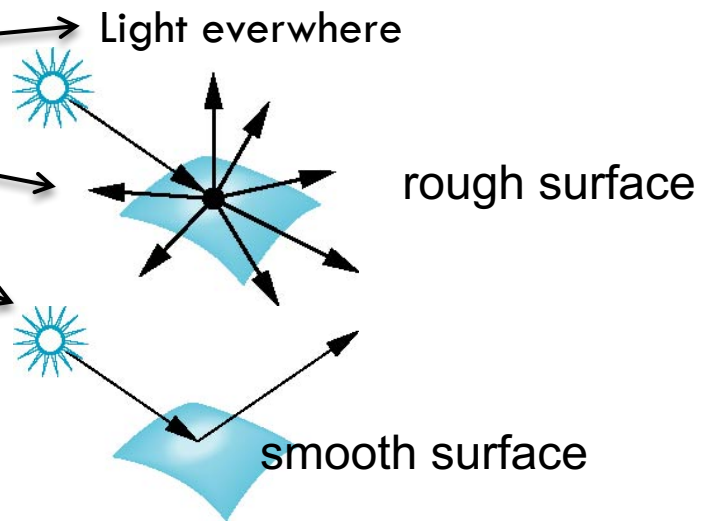
Lecture Plan

- Today:
  - Finish shading, 2F
  - Talk some about OpenGL V1
  - Talk about shaders (conceptual)
- Tuesday: get into actual OpenGL calls

- Summary: today is about concepts, Tuesday is about practical stuff

Shading

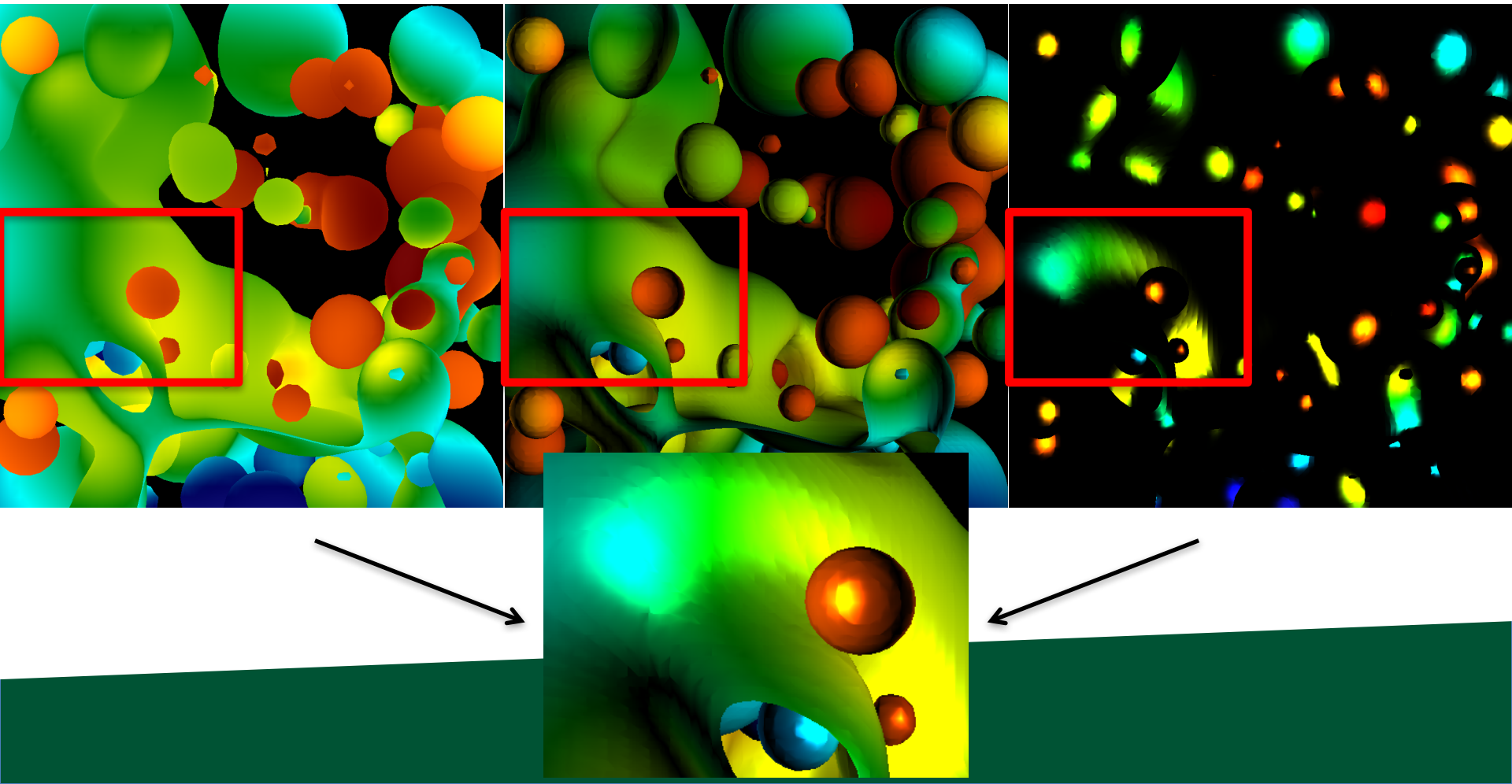
- Our goal:
  - For each pixel, calculate a shading factor
  - Shading factor typically between 0 and 1, but sometimes >1
    - Shading >1 makes a surface more white
- 3 types of lighting to consider:
  - Ambient
  - Diffuse
  - Specular



Our game plan:
Calculate all 3 and combine them.

Phong Model

- Combine three lighting effects: ambient, diffuse, specular



Phong Model

- Simple version: 1 light, with “full intensity” (i.e., don’t add an intensity term)
- Phong model
  - Shading_Amount = K_a + K_d*Diffuse + K_s*Specular
- Signature:
  - double CalculatePhongShading(LightingParameters &, double *viewDirection, double *normal)
  - Will have to calculate viewDirection for each pixel!

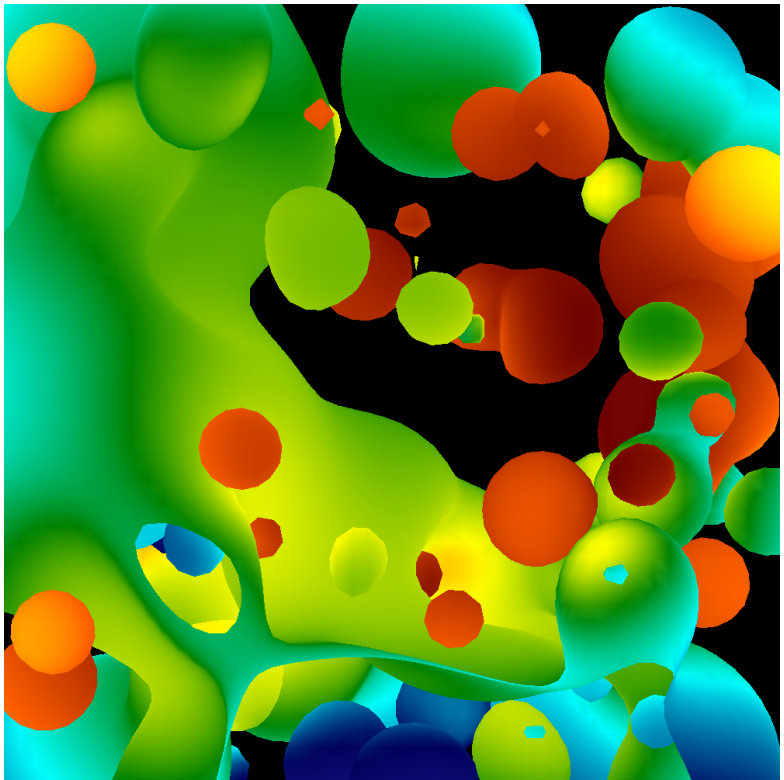
How to handle shading values greater than 1?

- Color at pixel = (1.0, 0.4, 0.8)
- Shading value = 0.5
  - Easy!
  - Color = (0.5, 0.2, 0.4)→ (128, 52, 103)
- Shading value = 2.0
  - Color = (1.0, 0.8, 1.0) → (255, 204, 255)
- Color_R = 255*min(1, R*shading_value)
- This is how bright lights makes things whiter and whiter.
  - But it won't put in colors that aren't there.

Ambient Lighting

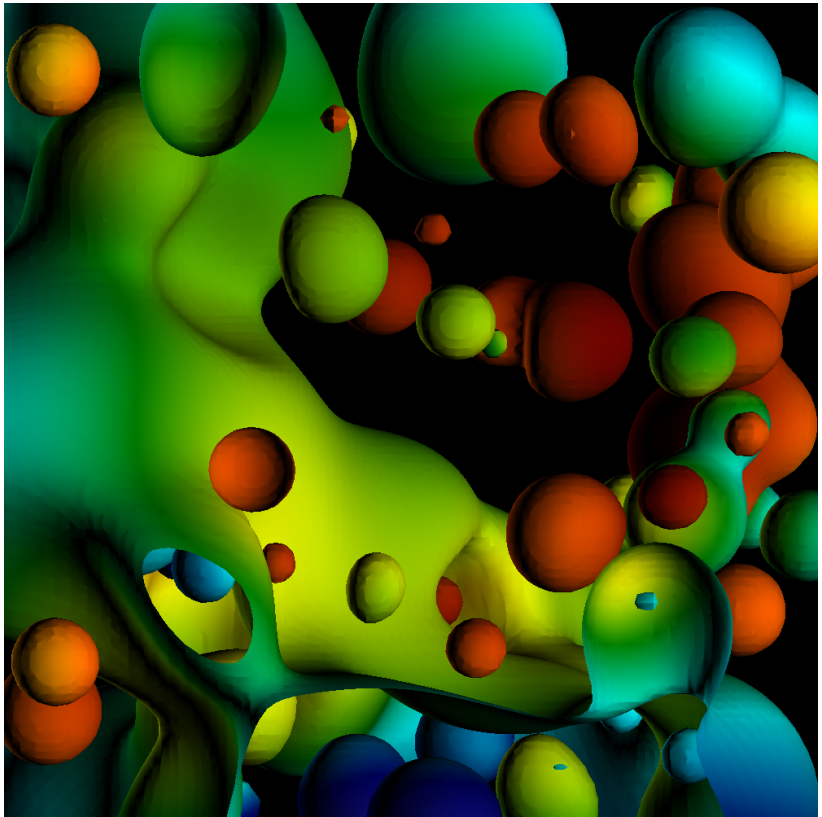
- Ambient light
  - Same amount of light everywhere in scene
  - Can model contribution of many sources and reflecting surfaces

Surface lit with ambient lighting only



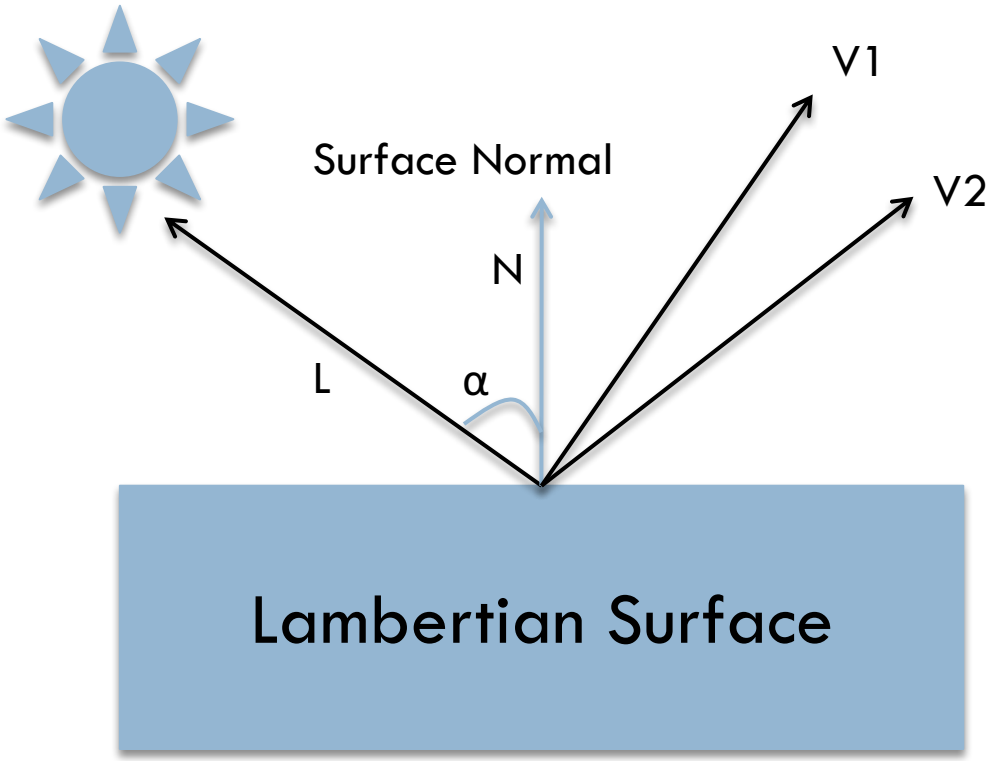
Diffuse Lighting

- Diffuse light
  - Light distributed evenly in all directions, but amount of light depends on orientation of triangles with respect to light source.
  - Different for each triangle



Surface lit with diffuse lighting only

Diffuse Lighting

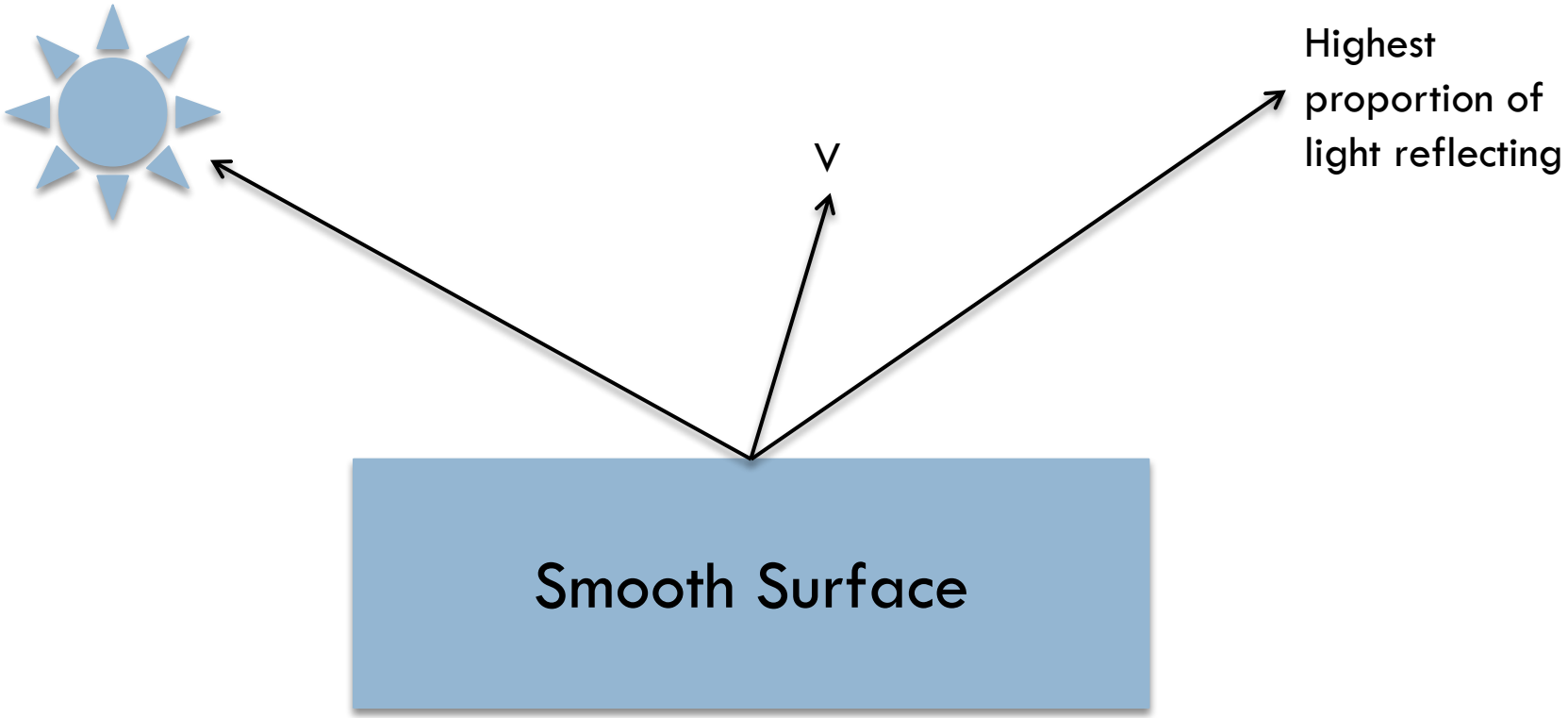


You can calculate the diffuse contribution by taking the dot product of L and N,

Since $L \cdot N = \cos(\alpha)$

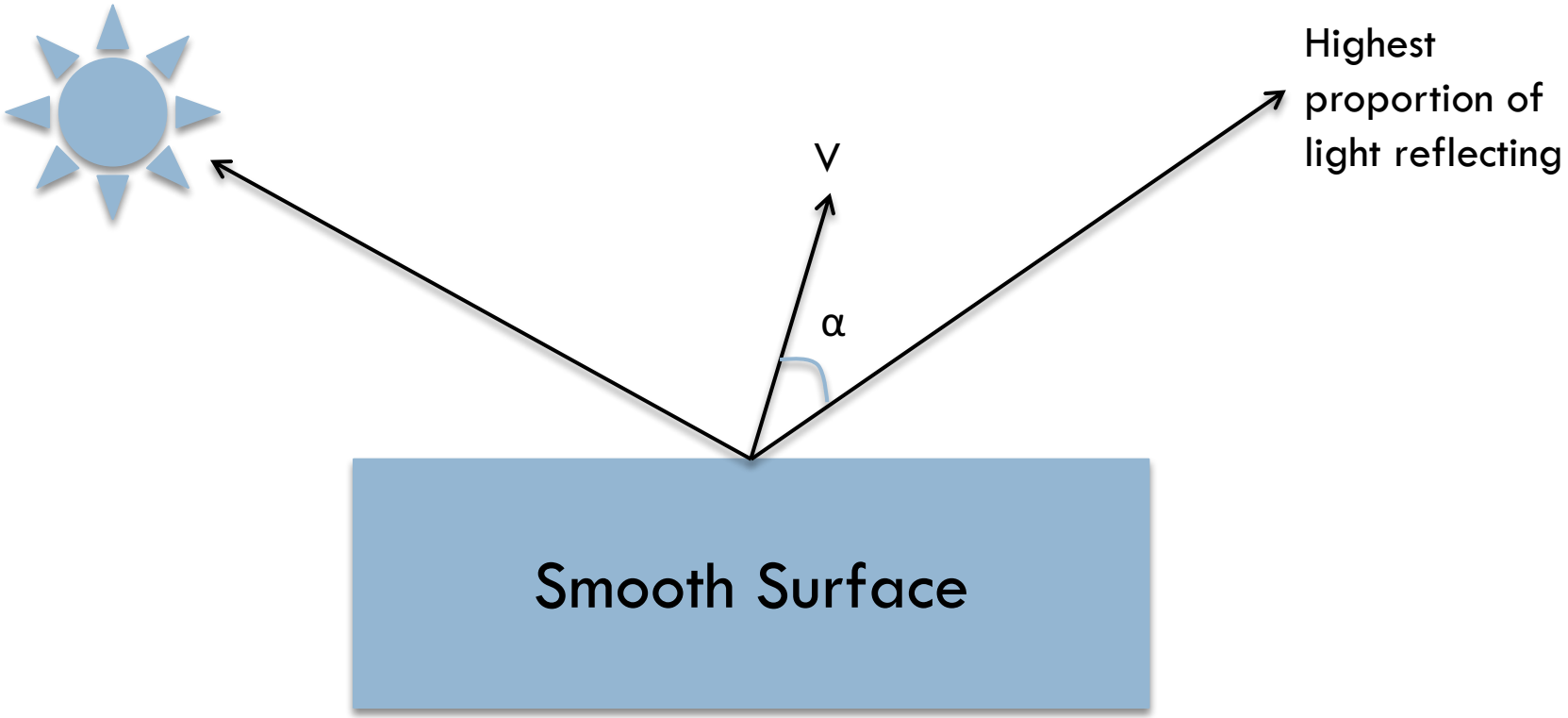
(assuming L and N are normalized)

How much light reflects with specular lighting?



How much light gets to point V?

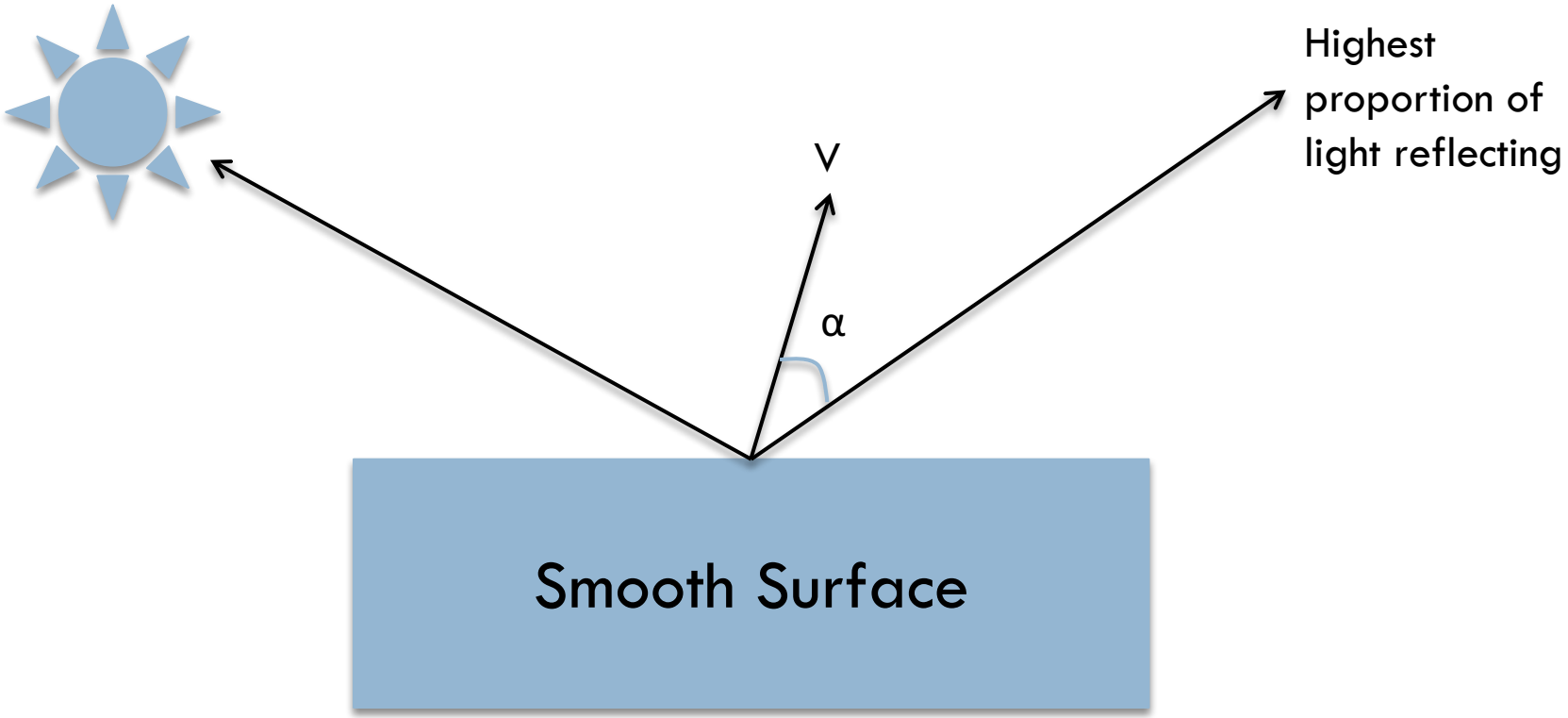
How much light reflects with specular lighting?



How much light gets to point V?

A: proportional to cos(α)

How much light reflects with specular lighting?

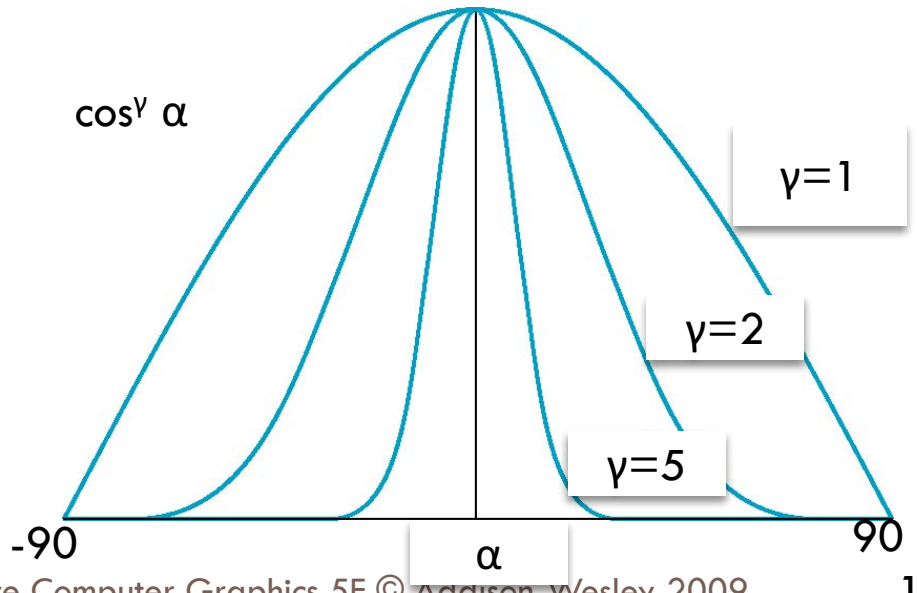


How much light gets to point V?

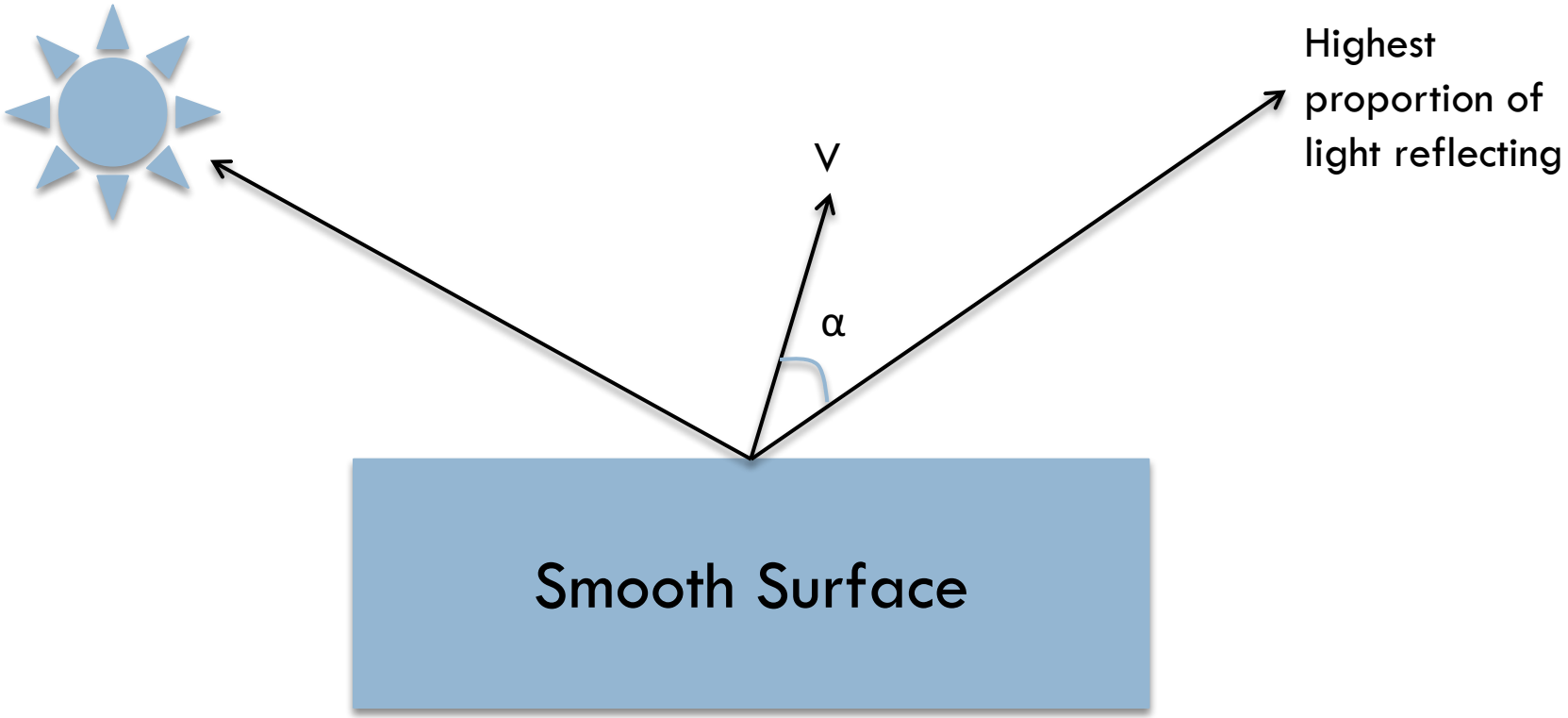
A: proportional to cos(α)
(Shininess strength) * cos(α) ^ (shininess coefficient)

γ: The Shininess Coefficient

- Values of γ between 100 and 200 correspond to metals
- Values between 5 and 10 give surface that look like plastic



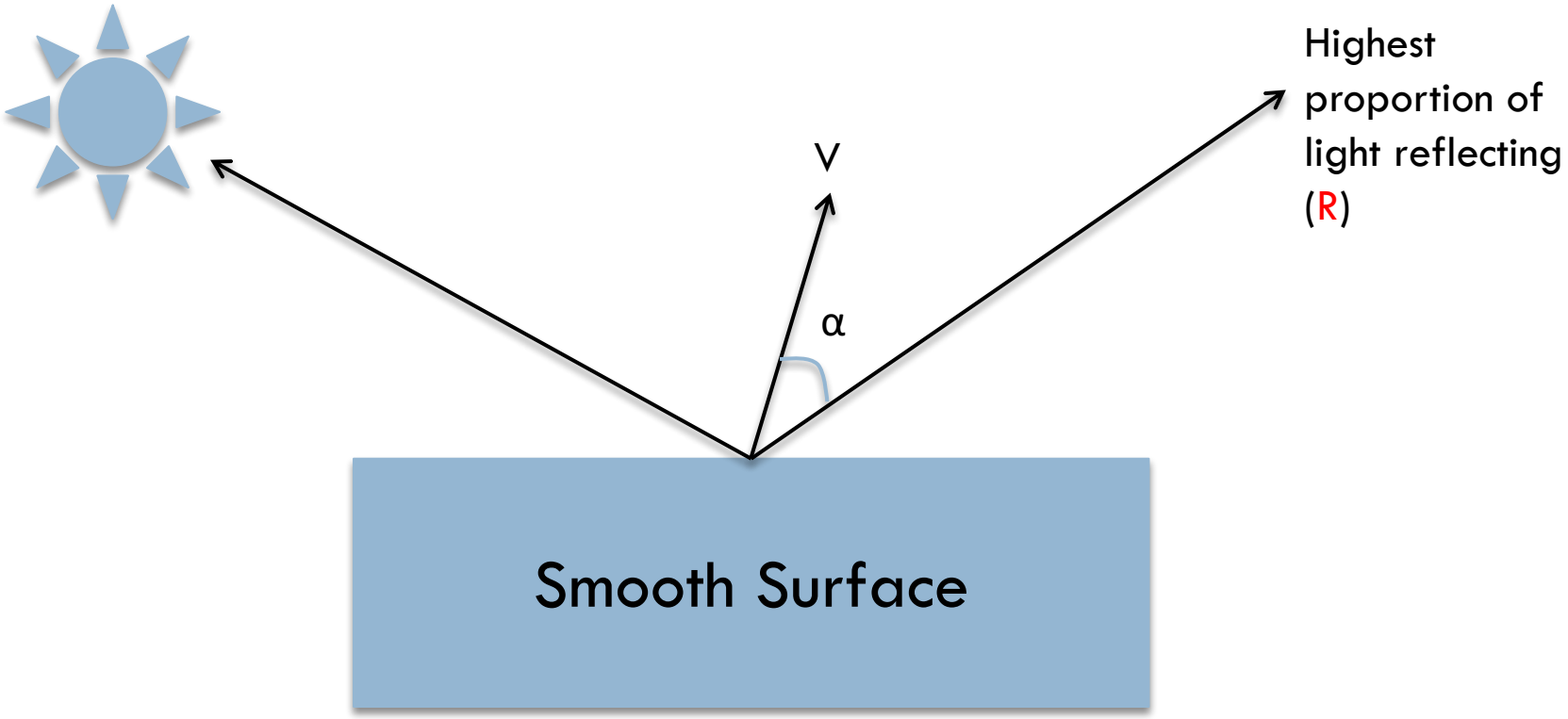
How much light reflects with specular lighting?



How much light gets to point V?

A: proportional to cos(α)
(Shininess strength) * cos(α) ^ (shininess coefficient)

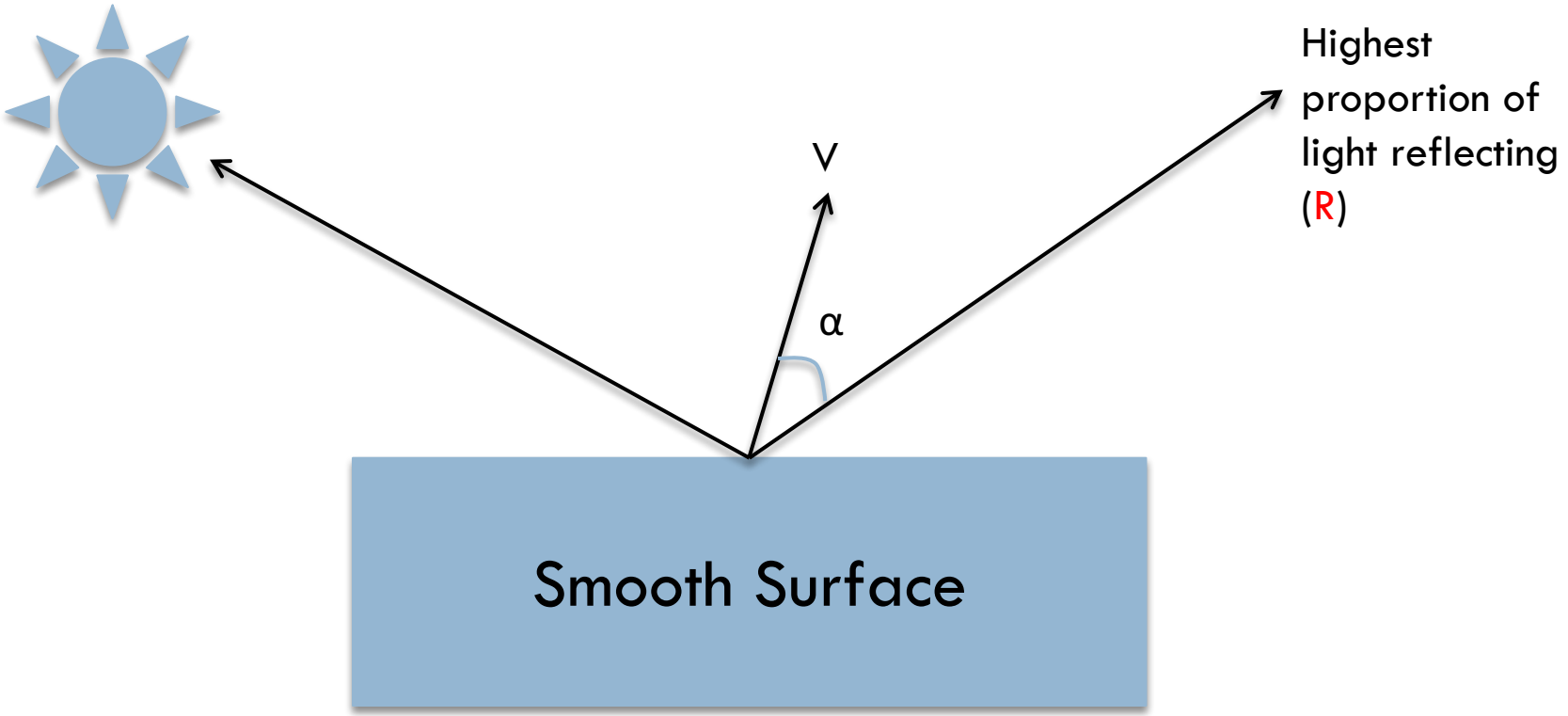
How much light reflects with specular lighting?



Great!

We know that cos(α) is V·R (provided V & R are normalized).

How much light reflects with specular lighting?



Great!

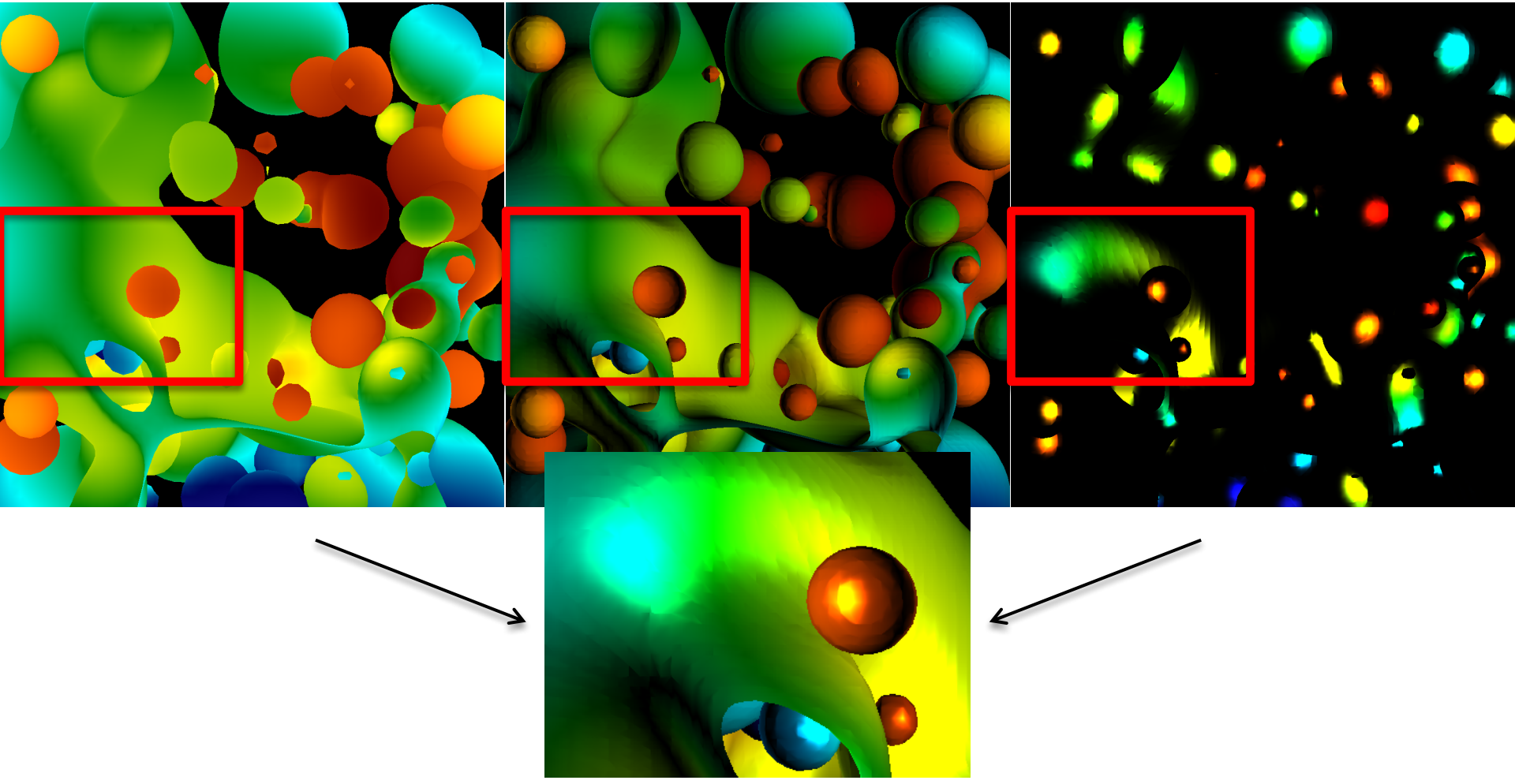
We know that cos(α) is V·R (provided V & R are normalized).

But what is R?

It is a formula: R = 2*(L·N)*N - L

Phong Model

- Combine three lighting effects: ambient, diffuse, specular



Phong Model

- Simple version: 1 light, with "full intensity" (i.e., don't add an intensity term)
- Phong model
  - Shading_Amount = K_a + K_d*Diffuse + K_s*Specular
- Signature:
  - double CalculatePhongShading(LightingParameters &, double *viewDirection, double *normal)
  - Will have to calculate viewDirection for each pixel!

Specular Term of Phong Model

- Specular part of Phong: K_s*Specular
- and Specular is: (Shininess strength) * cos(α) ^ (shininess coefficient)
- Putting it all together would be:
  - K_s * (Shininess strength) * cos(α) ^ (shininess coefficient)
- But now we have two multipliers, K_s and (Shininess Strength). Not needed.
- So: just use one. Drop Shininess Strength and only use K_s
  - K_s * cos(α) ^ (shininess coefficient)

Lighting parameters

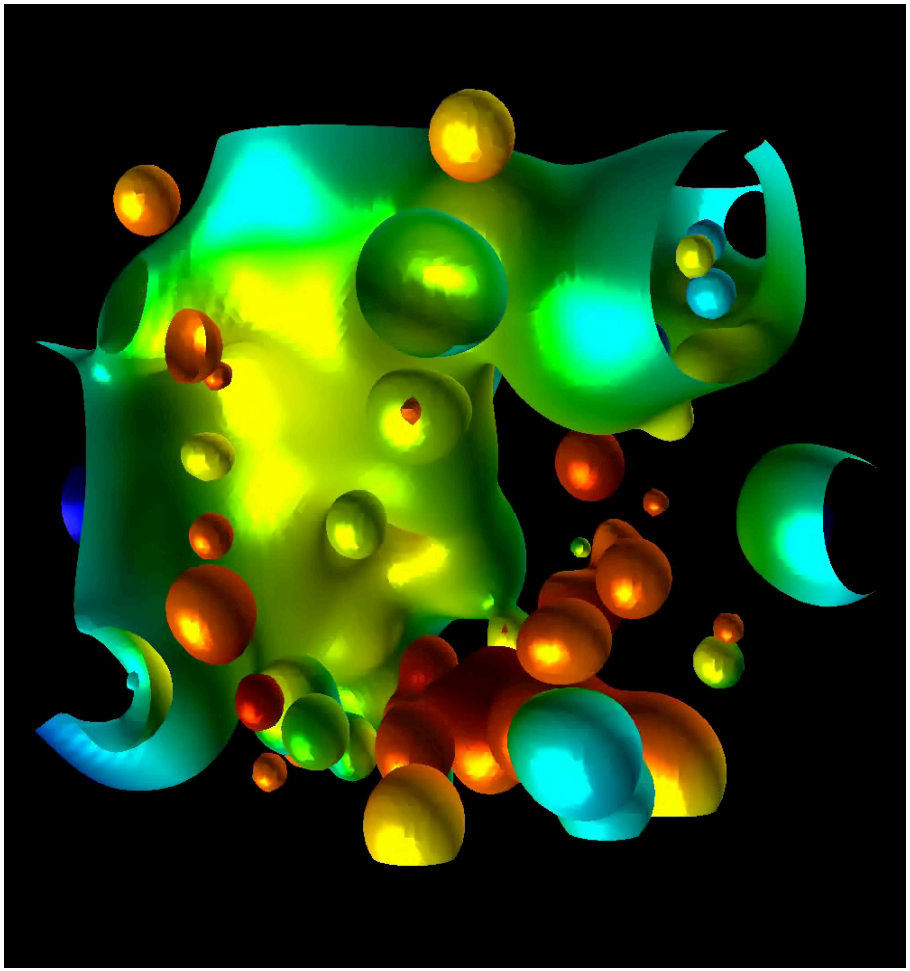
```
struct LightingParameters
{
    LightingParameters(void)
    {
        lightDir[0] = -0.6;
        lightDir[1] = 0;
        lightDir[2] = -0.8;
        Ka = 0.3;
        Kd = 0.7;
        Ks = 2.3;
        alpha = 2.5;
    };


    double lightDir[3]; // The direction of the light source
    double Ka;          // The coefficient for ambient lighting.
    double Kd;          // The coefficient for diffuse lighting.
    double Ks;          // The coefficient for specular lighting.
    double alpha;       // The exponent term for specular lighting.
};

LightingParameters lp;
```

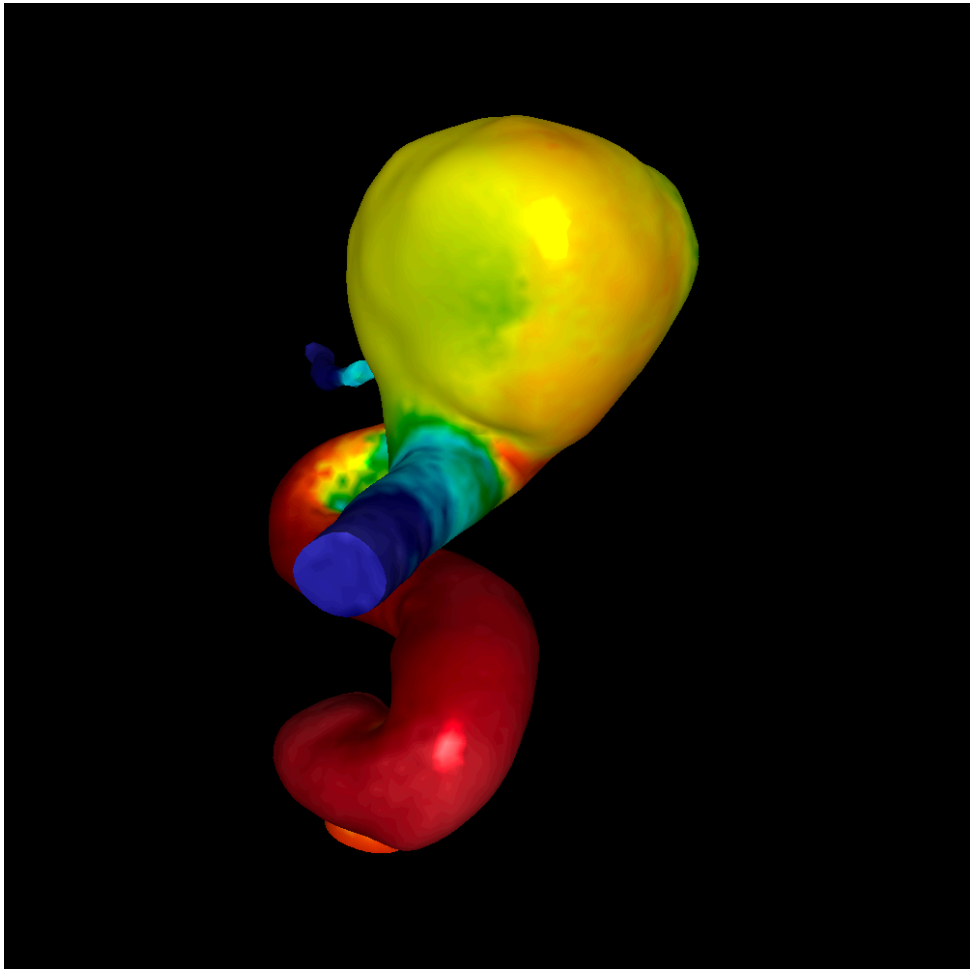
Project #1F (8%), Wed May 5th

- Goal: add shading, movie
- Extend your project1E code
- Important:
- add #define NORMALS
- Download new file, update to new file



Project #1F (8%), Wed May 5th

- Goal: add shading, movie
- Extend your project1E code
- Important:
- add #define NORMALS
- Download new file, update to new file



Changes to data structures

```
class Triangle
{
  public:
      double X[3], Y[3], Z[3];
      double colors[3][3];
      double normals[3][3];
};
```

→reader1e.cxx will not compile (with #define NORMALS) until you make these changes

→reader1e.cxx will initialize normals at each vertex

More comments (1/3)

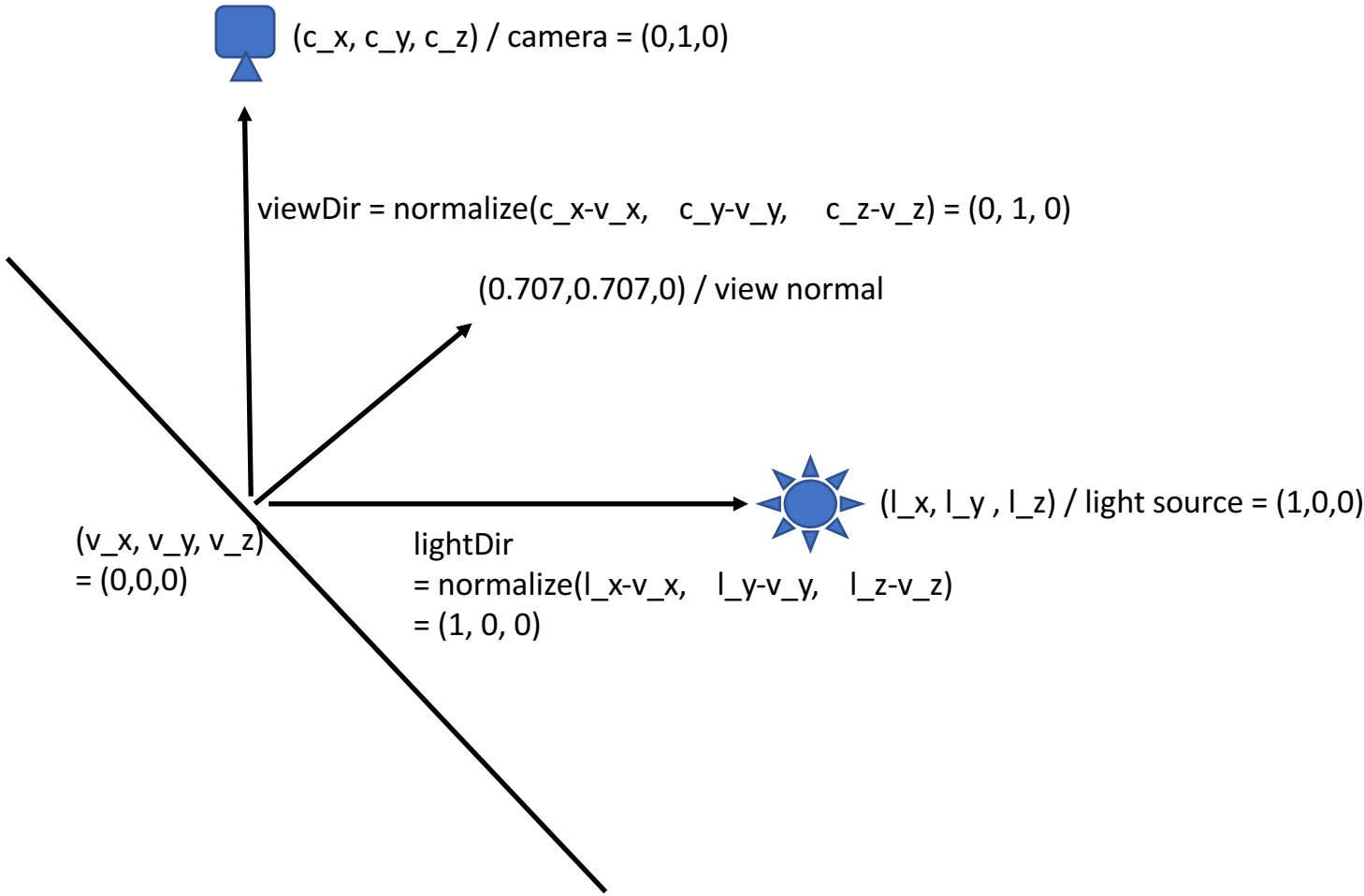
- This project in a nutshell:
  - Add method called “CalculateShading”
    - My version of CalculateShading is about ten lines of code.
  - Call CalculateShading for each vertex
  - This is a new field, which you will LERP
  - Modify RGB calculation to use shading

More comments (2/3)

- New: more data to help debug
  - I will make the shading value for each pixel available
  - I will also make it available for ambient, diffuse, specular
- ~~Don't forget to do two-sided lighting~~
- REVERSAL: do one-sided lighting

This example has a triangle vertex, v, at the origin, the camera one unit along the Y-axis and the light source one unit along the X-axis.

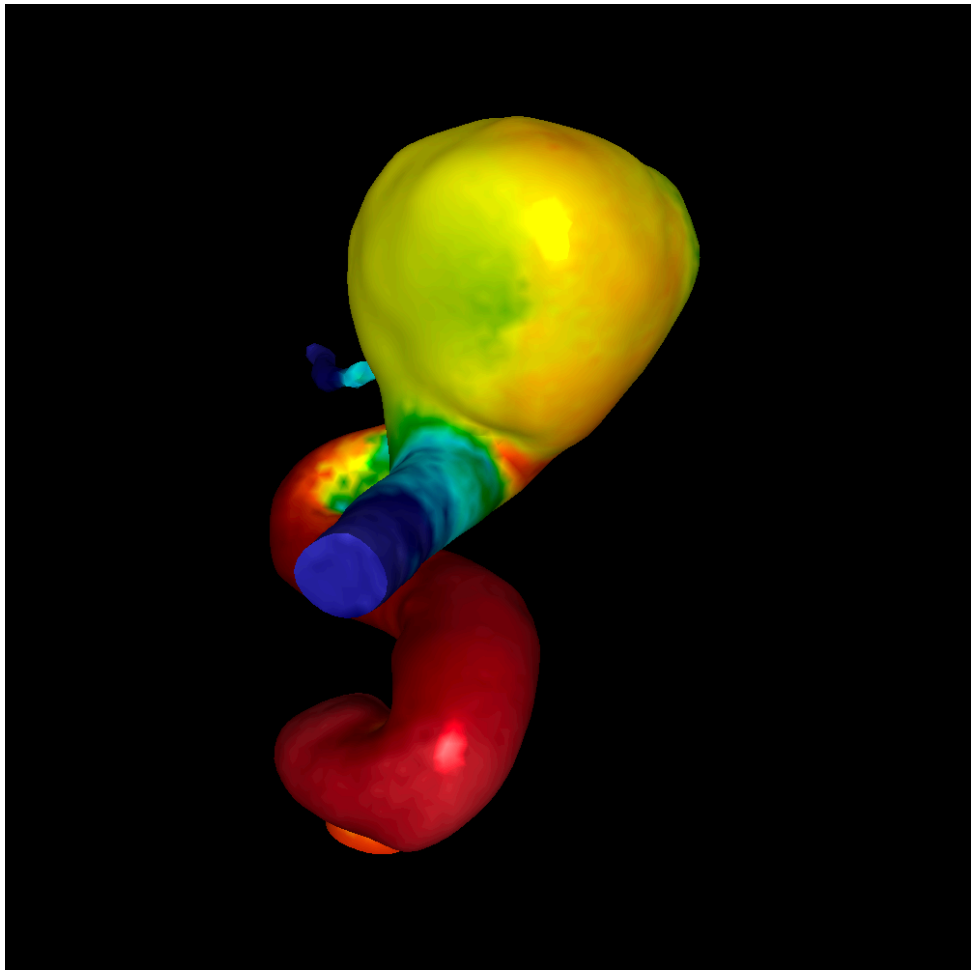
The lightDir and viewDir formulas show the conventions we should use for direction for general positions.



More comments (3/3)

- I haven't said anything about movie encoders

Project #1F (8%), Due Weds May 5th



- Goal: add shading, movie

Lecture Plan

- Today:
  - Finish shading, 2F
  - Talk some about OpenGL V1
  - Talk about shaders (conceptual)
- Tuesday: get into actual OpenGL calls

- Summary: today is about concepts, Tuesday is about practical stuff

Some notes about OpenGL

- OpenGL has evolved a lot over 25+ years
- The slides that follow ~~and the homeworks~~ will detail an early version of OpenGL (OpenGL V1.0)
- This is the easiest version to understand and implement
  - It is also inefficient
- Since efficiency is important, newer versions are more complex and also faster
  - Lecture will conclude with conceptual overview of newer OpenGL. Tuesday's lecture will have specific details.

Models and Architectures

Ed Angel

Professor of Computer Science, Electrical and Computer Engineering, and Media Arts

University of New Mexico

Objectives

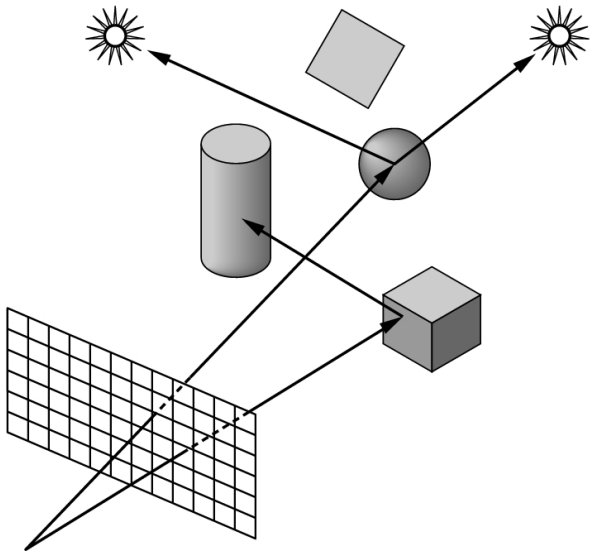
- Learn the basic design of a graphics system
- Introduce pipeline architecture
- Examine software components for an interactive graphics system

Image Formation Revisited

- Can we mimic the synthetic camera model to design graphics hardware software?
- Application Programmer Interface (API)
  - Need only specify
    - Objects
    - Materials
    - Viewer
    - Lights
- But how is the API implemented?

Physical Approaches

- **Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
  - Can handle global effects
    - Multiple reflections
    - Translucent objects
  - Slow
  - Must have whole data base available at all times



- **Radiosity**: Energy based approach
  - Very slow

Practical Approach

- Process objects one at a time in the order they are generated by the application
  - Can consider only local lighting
- Pipeline architecture

application program

display

- All steps can be implemented in hardware on the graphics card

The University of New Mexico

Vertex Processing

- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates
  - Camera (eye) coordinates
  - Screen coordinates
- Every change of coordinates is equivalent to a matrix transformation
- Vertex processor also computes vertex colors

The University of New Mexico

Projection

- *Projection* is the process that combines the 3D viewer with the 3D objects to produce the 2D image
  - Perspective projections: all projectors meet at the center of projection
  - Parallel projection: projectors are parallel, center of projection is replaced by a direction of projection

Primitive Assembly

Vertices must be collected into geometric objects before clipping and rasterization can take place

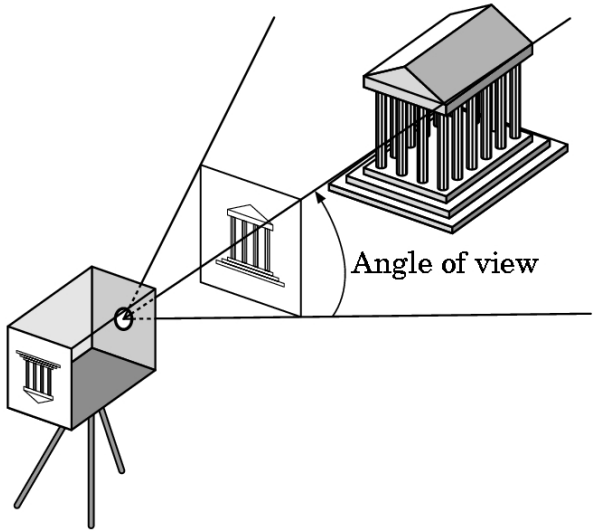
- Line segments
- Polygons
- Curves and surfaces

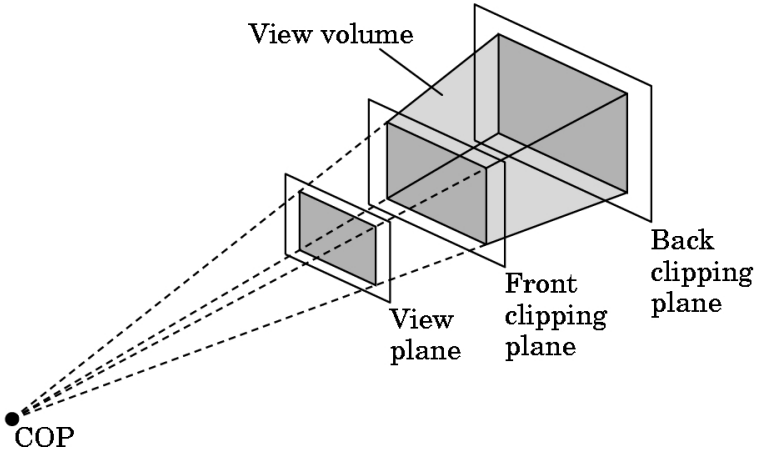
The University of New Mexico

Clipping

Just as a real camera cannot "see" the whole world, the virtual camera can only see part of the world or object space

- Objects that are not within this volume are said to be *clipped* out of the scene





Rasterization

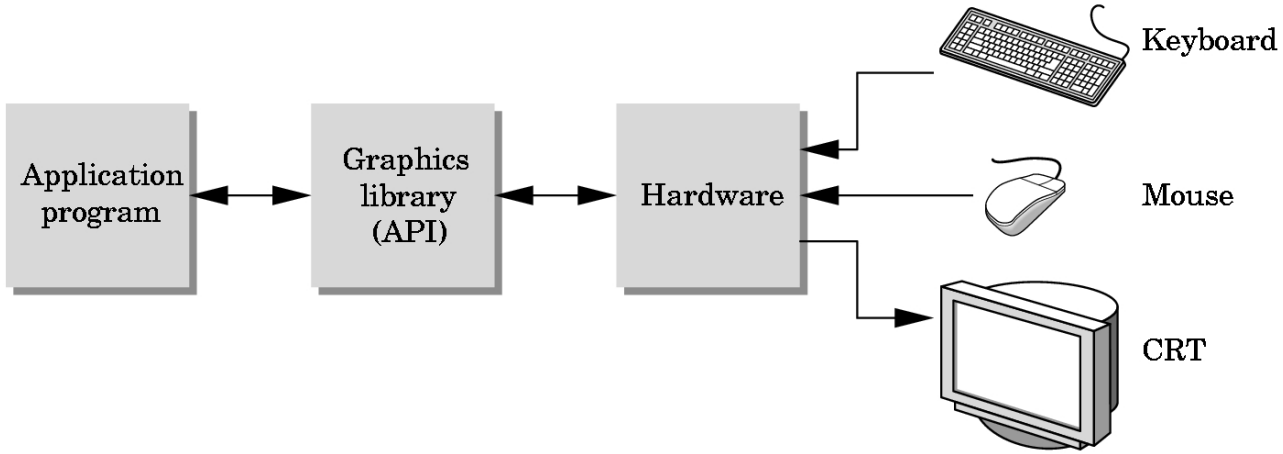
- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each object
- Fragments are “potential pixels”
  - Have a location in frame bufffer
  - Color and depth attributes
- Vertex attributes are interpolated over objects by the rasterizer

Fragment Processing

- Fragments are processed to determine the color of the corresponding pixel in the frame buffer
- Colors can be determined by *texture mapping* or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera
  - Hidden-surface removal

The Programmer's Interface

- Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)



API Contents

- Functions that specify what we need to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Materials
- Other information
  - Input from devices such as mouse and keyboard
  - Capabilities of system

Object Specification

- Most APIs support a limited set of primitives including
  - Points (0D object)
  - Line segments (1D objects)
  - Polygons (2D objects)
  - Some curves and surfaces
    - Quadrics
    - Parametric polynomials
- All are defined through locations in space or *vertices*

Example (OLD!!)

type of object

location of vertex

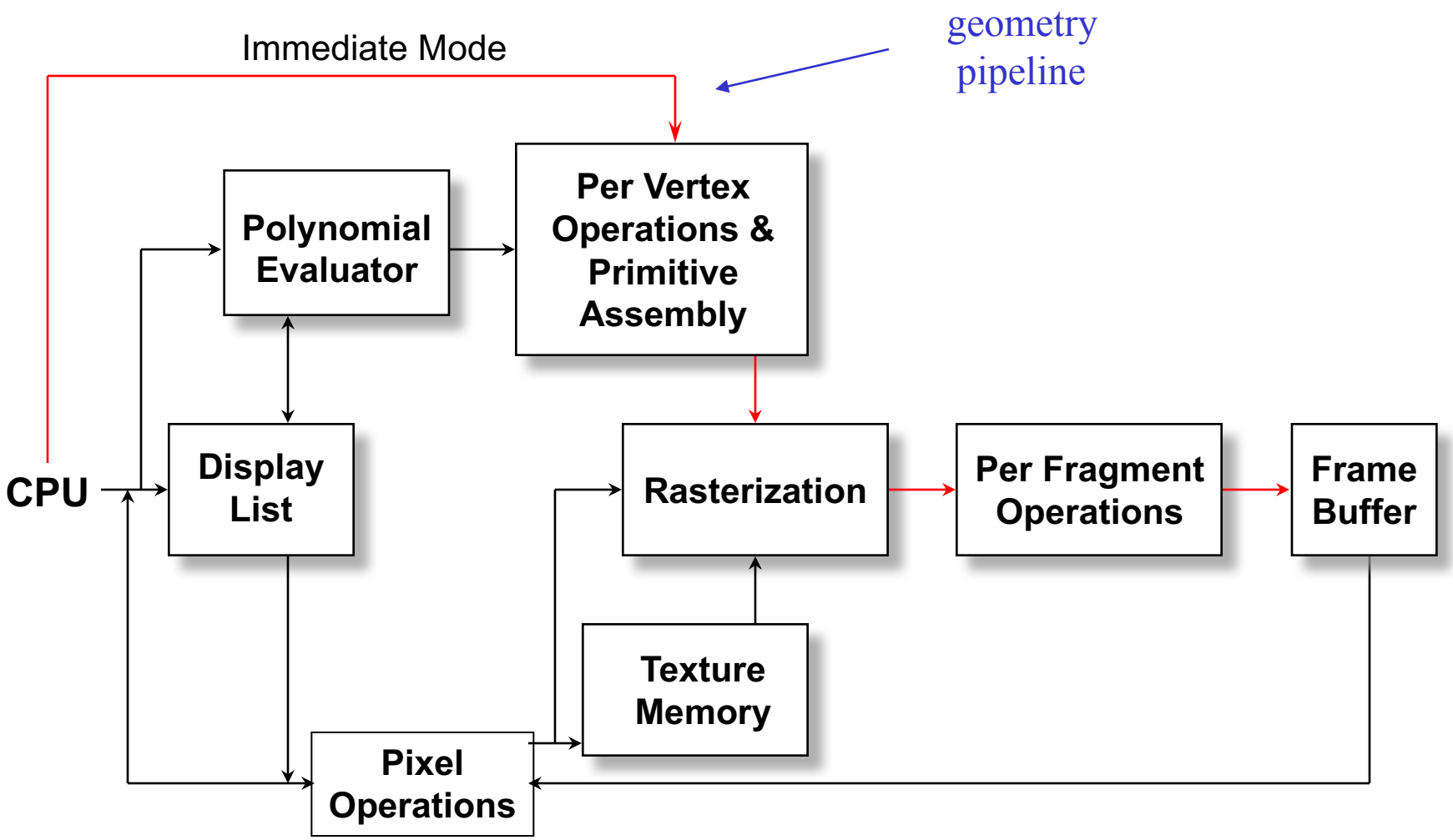
```
glBegin(GL_POLYGON);
 glVertex3f(0.0, 0.0, 0.0);
 glVertex3f(0.0, 1.0, 0.0);
 glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

end of object definition

Lights and Materials (OLD!!)

- Types of lights
  - Point sources vs distributed sources
  - Spot lights
  - Near and far sources
  - Color properties
- Material properties
  - Absorption: color properties
  - Scattering
    - Diffuse
    - Specular

OpenGL Architecture (OLD!!)



Lecture Plan

- Today:
  - Finish shading, 2F
  - Talk some about OpenGL V1
  - Talk about shaders (conceptual)
- Tuesday: get into actual OpenGL calls

- Summary: today is about concepts, Tuesday is about practical stuff

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

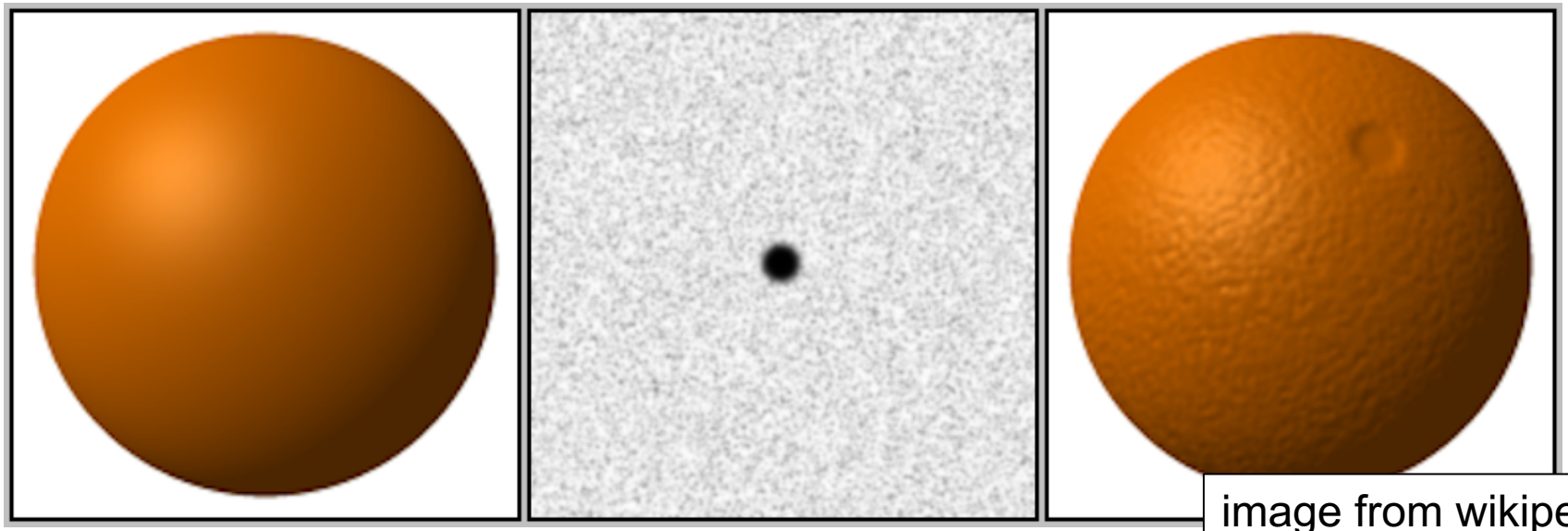
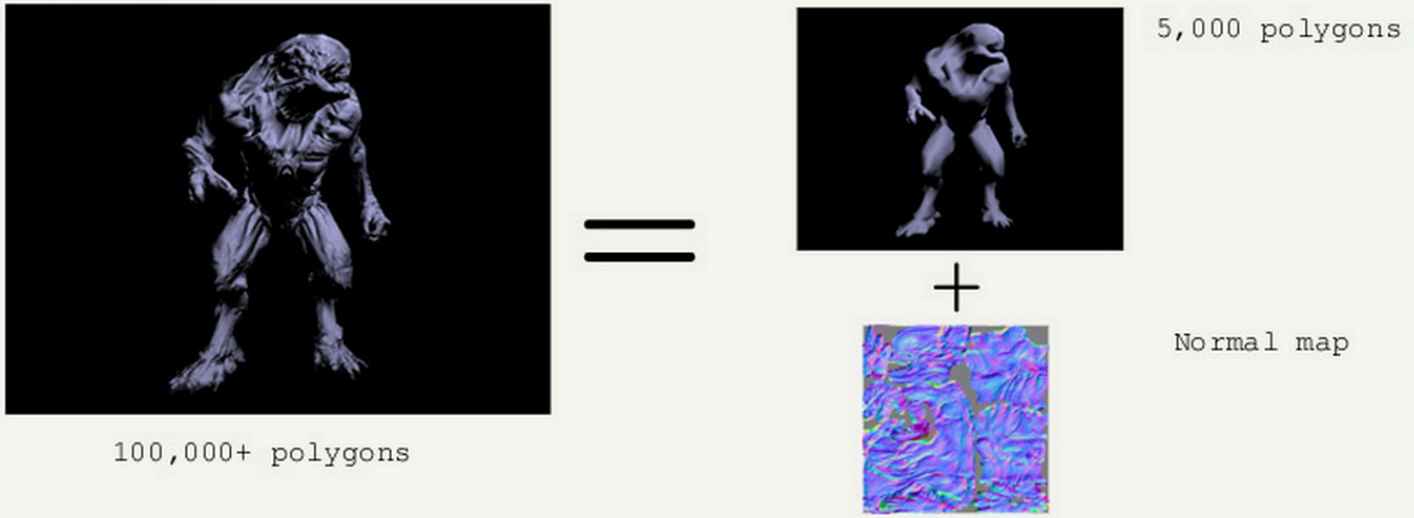


image from wikipedia

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.



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 easy in V1 of the GL interface
- Much more possible with shaders

Shading Languages

- shading language: programming language for graphics, specifically “shader” effects
- Benefits: increased flexibility with rendering
- OpenGL V1: 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

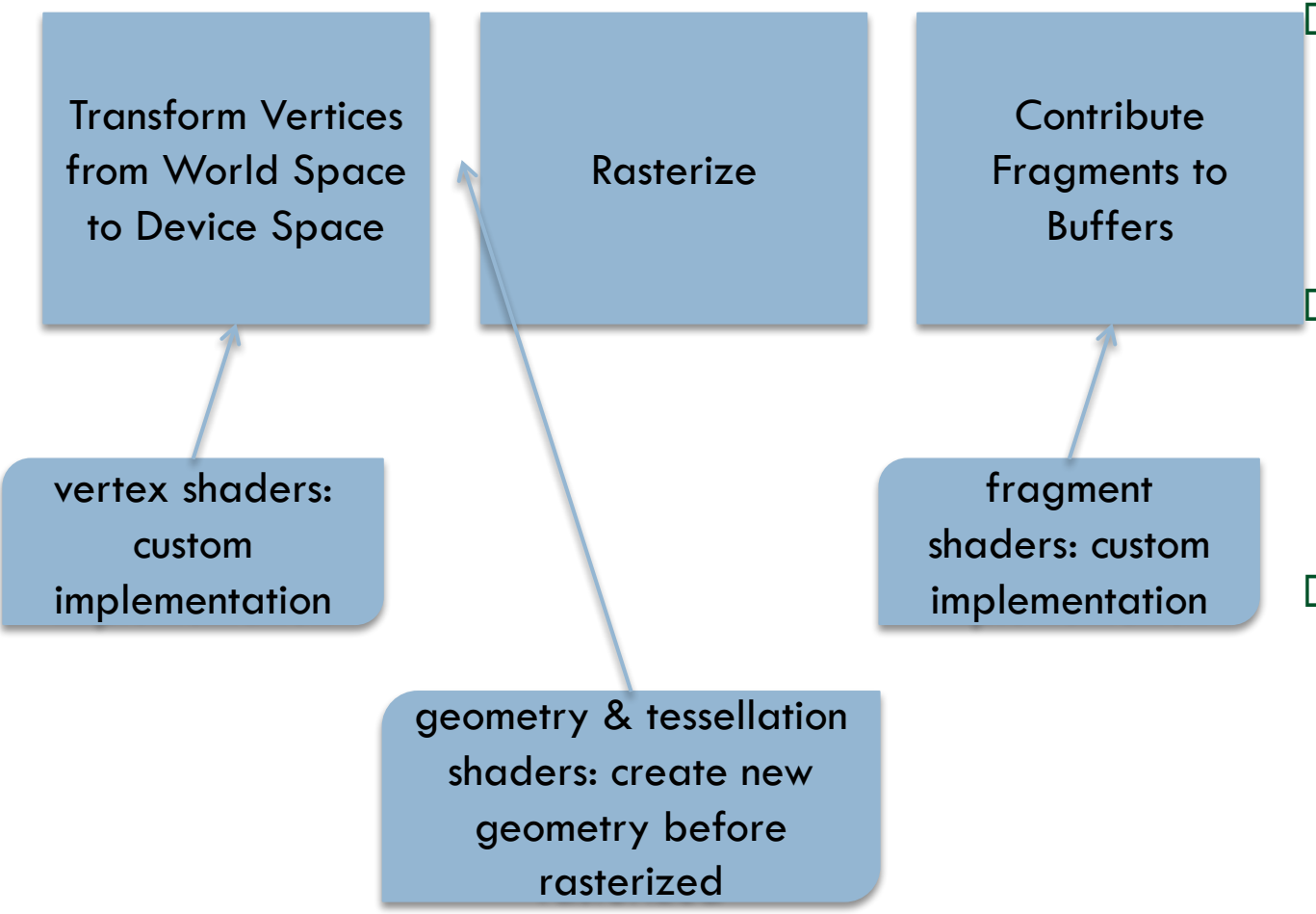
Source: wikipedia

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

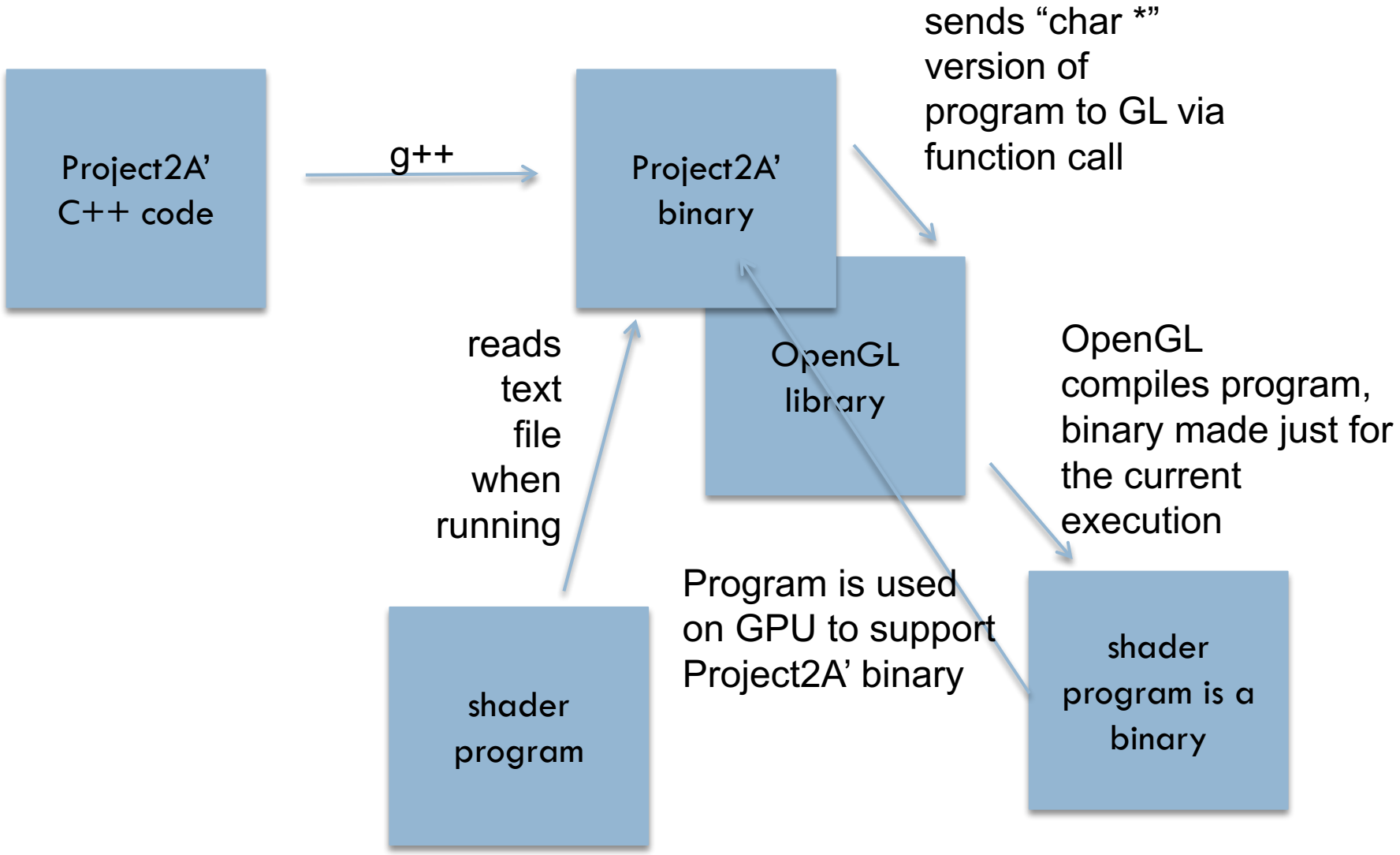
Fragment Shader

- Run once for each fragment
- Purpose: replaces the fixed capabilities in OpenGLV1 (texturing, color sum and fog)
- 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);
}
```

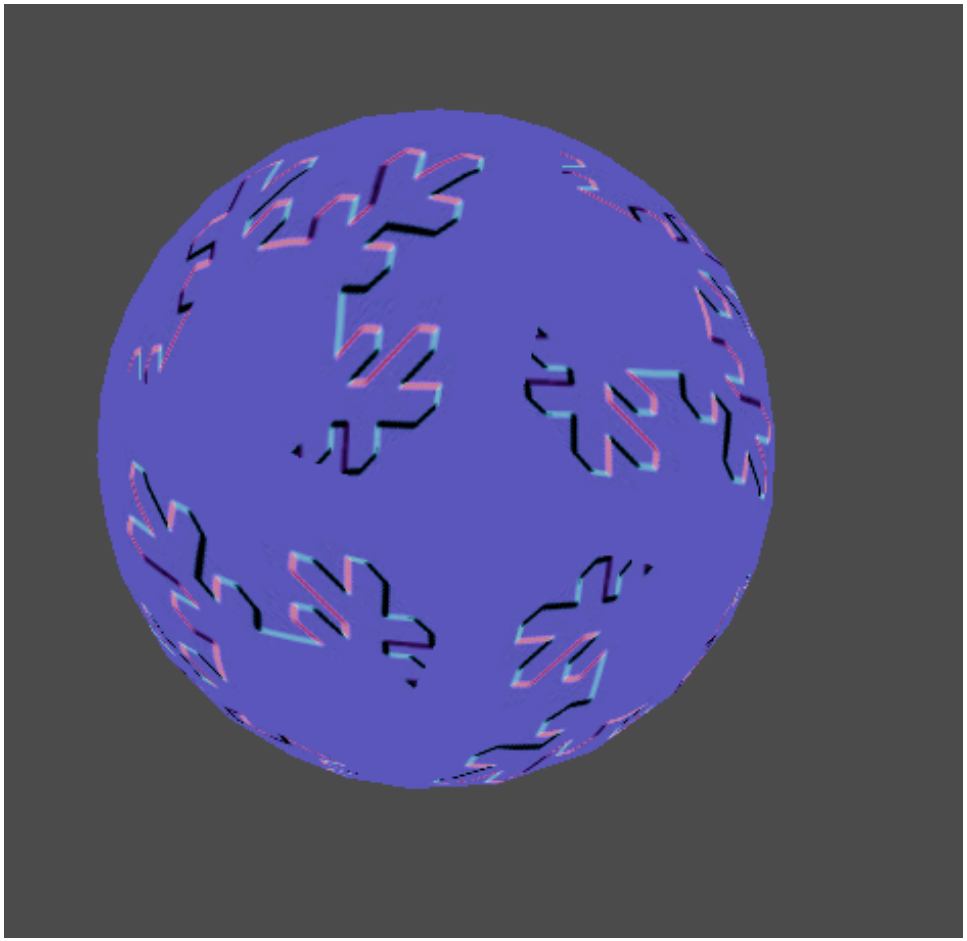
Simplest Vertex Shader

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

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

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);
}
```