SIGGRAPH2008

OpenGL: What's Coming Down the Graphics Pipeline

SIGGRAPH2008

Syllabus

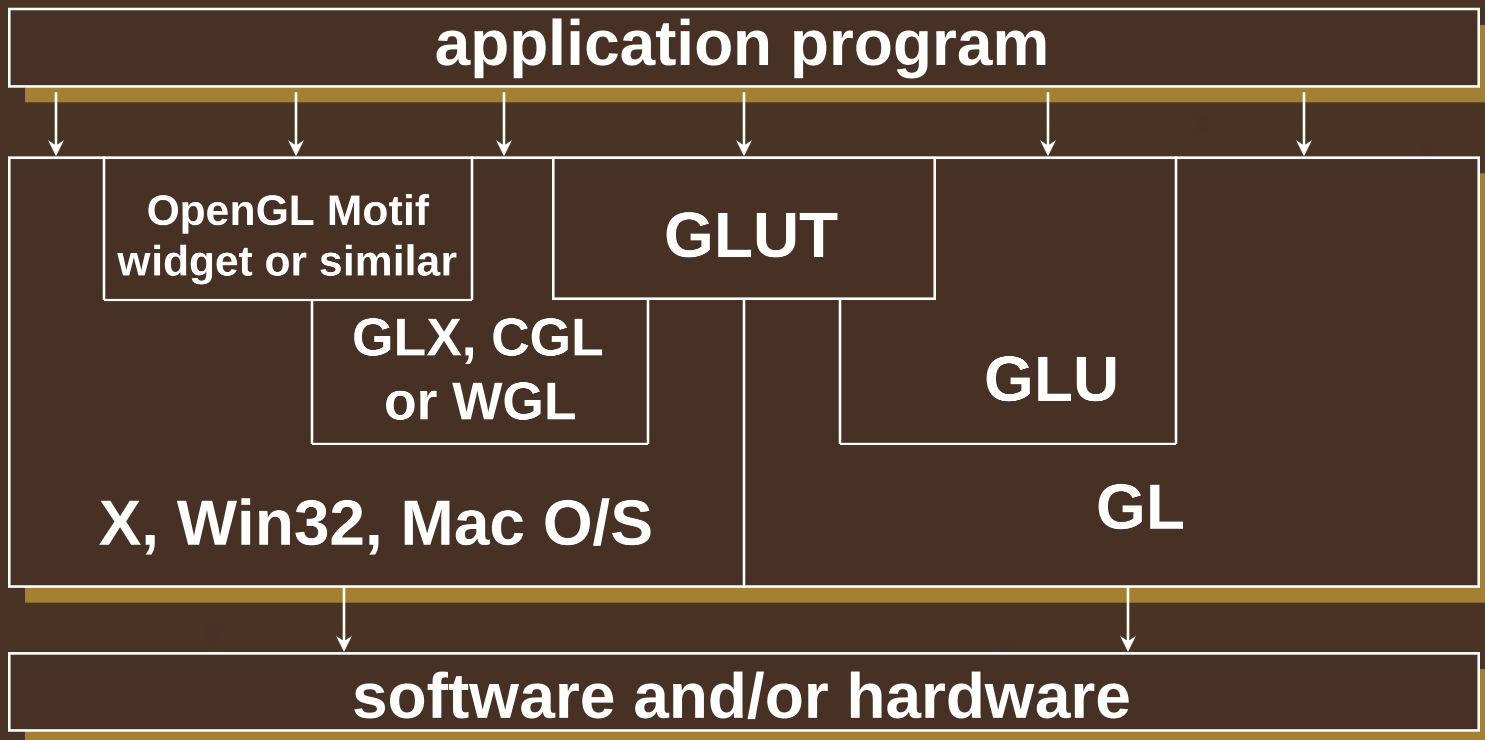
- Introduction/Rendering [Shreiner]
- Shader Overview [Licea-Kane]
- « Break »
- Fundamental Techniques [Hart]
- Applications [Angel]
- OpenGL 3.0 Overview [Shreiner]

OpenGL™ SIGGRAPH2008

What Is OpenGL, and What Can It Do for Me?

- OpenGL is a computer graphics *rendering* API
  - Generate high-quality color images by rendering with geometric and image primitives
  - Create interactive applications with 3D graphics
  - OpenGL is a cross-platform API
    - operating system independent
    - window system independent

OpenGL and Its Related APIs

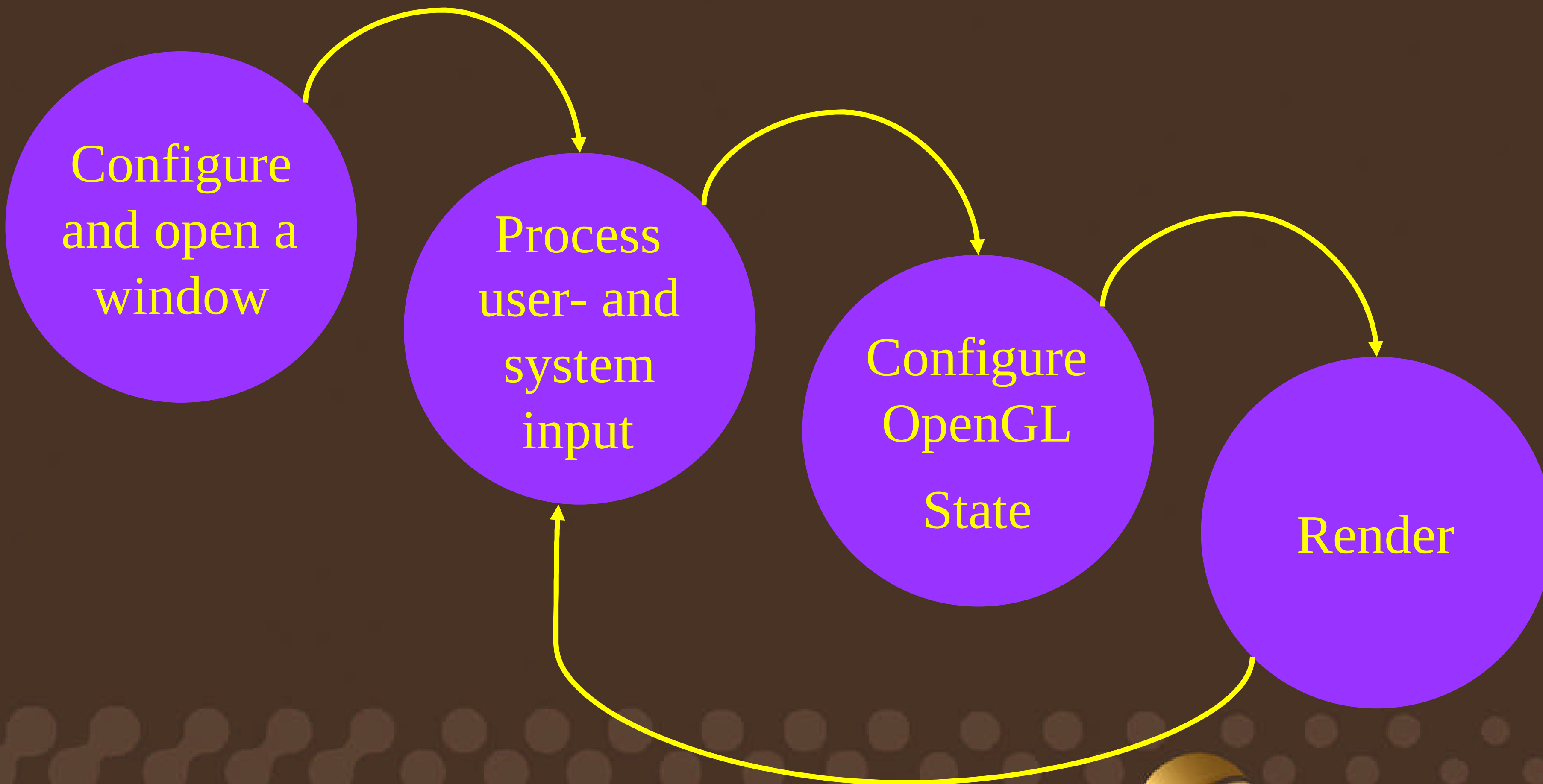


Today's Course's Ground-rules

- We assume you're familiar with basic graphics concepts
- A new version of OpenGL is coming
  - Emphasize the new way to program with OpenGL
    - using shaders – we won't discuss the fixed-function pipeline
- Updated notes and demo programs available at:

  http://www.opengl-redbook.com/SIGGRAPH/08

General Structure of an OpenGL Program

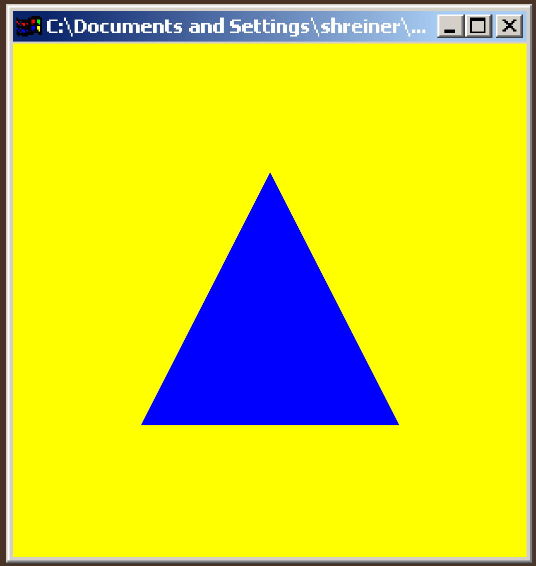


SIGGRAPH2008

An OpenGL Program

```
#include <GL/glut.h>
```

```
void main( int argc, char *argv[] )
{
  glutInit( &argc, argv );
  glutInitDisplayMode( GLUT_RGBA |
                       GLUT_DEPTH |
                       GLUT_DOUBLE );
  glutCreateWindow( argv[0] );

  init();

  glutDisplayFunc( display );
  glutReshapeFunc( reshape );

  glutMainLoop();
}
```



The main part of the program. *GLUT* is used to open the OpenGL window, and handle input from the user.

An OpenGL Program (cont'd.)

```
void init()
{
    glClearColor( 1.0, 1.0, 0.0, 1.0 );
    glEnable( GL_DEPTH_TEST );
    glEnableVertexAttribArray( 0 );

    // Load Shaders and other
    // initialization operations
}
```

Set up some initial OpenGL *state*

```
void reshape( int width, int height )
{
    glViewport( 0, 0, width, height );

    // Update other transformation state
}
```

Handle when the user resizes the window

An OpenGL Program (cont'd.)

```
void display( void )
{
  const GLfloat pos[][] = {
    { -0.5, -0.5 },
    {  0.0,  0.5 },
    {  0.5, -0.5 }
  };

  glClear( GL_COLOR_BUFFER_BIT |
           GL_DEPTH_BUFFER_BIT );

  glVertexAttribPointer( 0, 2, GL_FLOAT,
                         GL_FALSE, 0, pos );
  glDrawArrays( GL_TRIANGLES, 0, 3 );

  glutSwapBuffers();
}
```

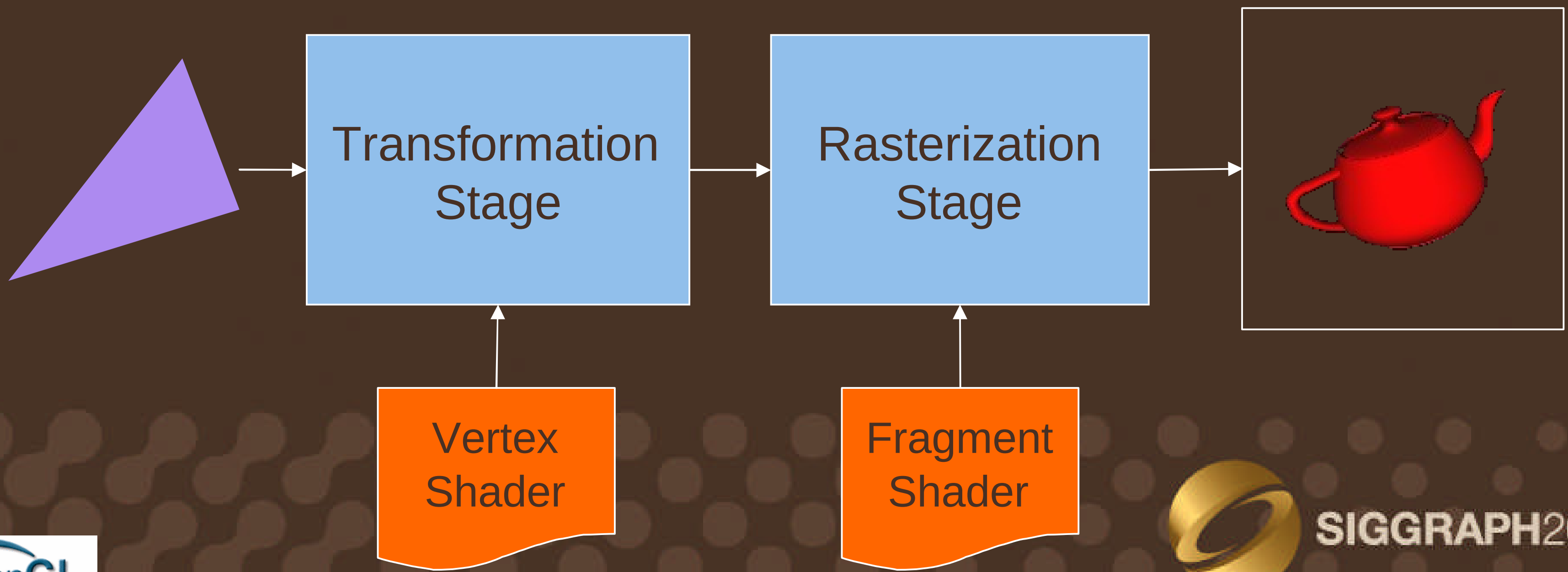
Have OpenGL draw a triangle from 2D points (*vertices)*

Graphic Pipeline Varieties

- *Fixed-function* version
  - think of a big machine with lots of knobs and switches
    - can only modify parameters and disable operations
  - order of operations is fixed
  - limited to what's implemented in the pipeline
- *Programmable* version
  - interesting parts of pipeline are under your control
    - write *shaders* to implement those operations
  - boring stuff is still "hard coded"
    - rasterization & fragment testing

The Graphics Pipeline

- *Transformation stage* converts 3D models into pixel locations
- *Rasterization stage* fills the associated pixels



(Perhaps) The Simplest Useful Vertex Shader

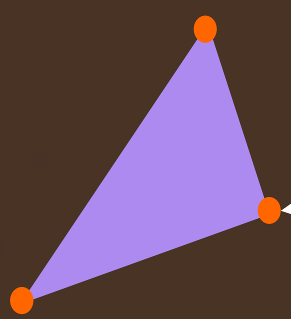
```
attribute vec4 vertex;

void main()
{
    gl_Position = vertex;
}
```

(Likewise, Perhaps) The Simplest Fragment Shader

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

Representing Geometry

- We represent geometric primitives by their *vertices*

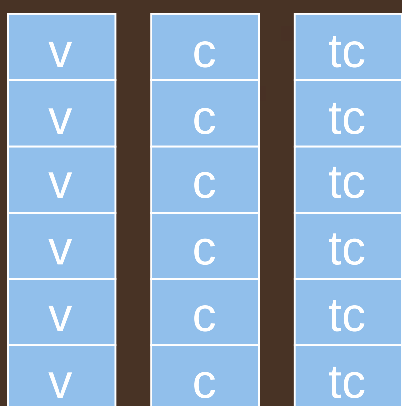


$$\vec{v} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

- Vertices are specified as *homogenous coordinates*
  - 4-tuple of reals
  - most "vertex data" are homogenous coordinates
    - makes the math easier

Storing Vertex Attributes

- Vertex arrays are very flexible
  - store data contiguously as an array, or

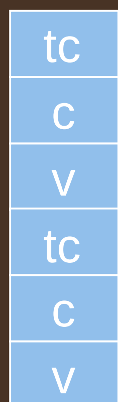



```
glVertexAttribPointer( vIndex, 3,
  GL_FLOAT, GL_FALSE, 0, v );

glVertexAttribPointer( cIndex, 4,
  GL_UNSIGNED_BYTE, GL_TRUE,
  0, c );

glVertexAttribPointer( tcIndex, 2,
  GL_FLOAT, GL_FALSE, 0, tc );
```

Storing Vertex Attributes (cont'd)

- As "offsets" into a contiguous array of structures

```
struct VertexData {
  GLfloat tc[2];
  GLubyte  c[4];
  GLfloat  v[3];
};

VertexData  verts;
```




```
glVertexAttribPointer( vIndex,
  3, GL_FLOAT, GL_FALSE,
  sizeof(VertexData), verts[0].v );

glVertexAttribPointer( cIndex,
  4, GL_UNSIGNED_BYTE, GL_TRUE,
  sizeof(VertexData), verts[0].c );

glVertexAttribPointer( tcIndex,
  2, GL_FLOAT, GL_FALSE,
  sizeof(VertexData), verts[0].tc );
```

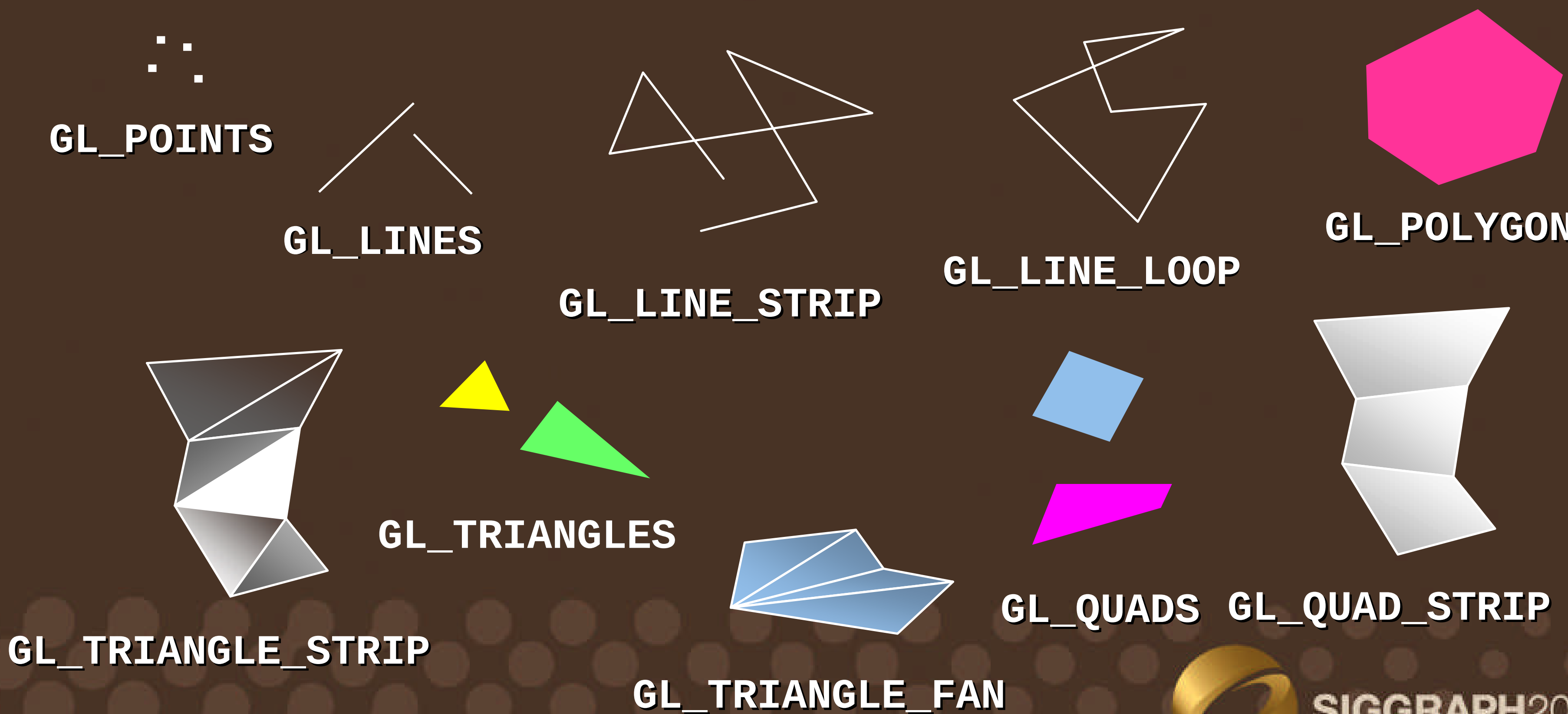
“Turning on” Vertex Arrays

- Need to let OpenGL ES know which vertex arrays you’re going to use

```
glEnableVertexAttribArray( vIndex );
glEnableVertexAttribArray( cIndex );
glEnableVertexAttribArray( tcIndex );
```

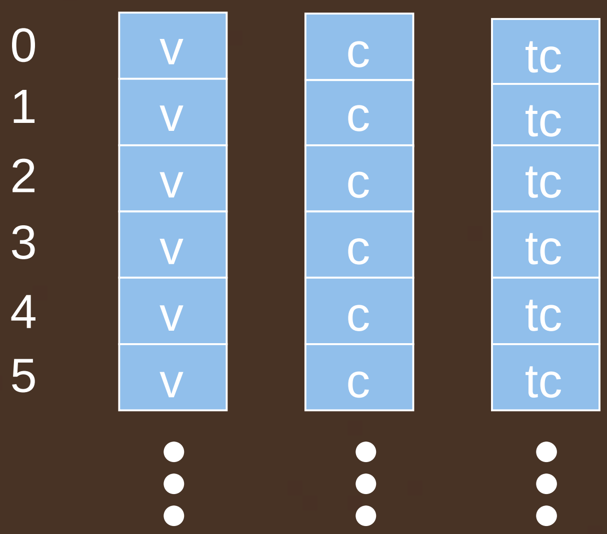
OpenGL's Geometric Primitives

- All primitives are specified by *vertices*

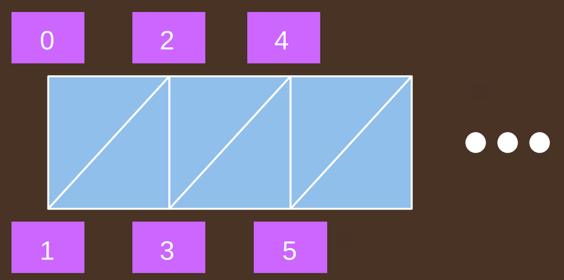


Drawing Geometric Primitives

- For contiguous groups of vertices

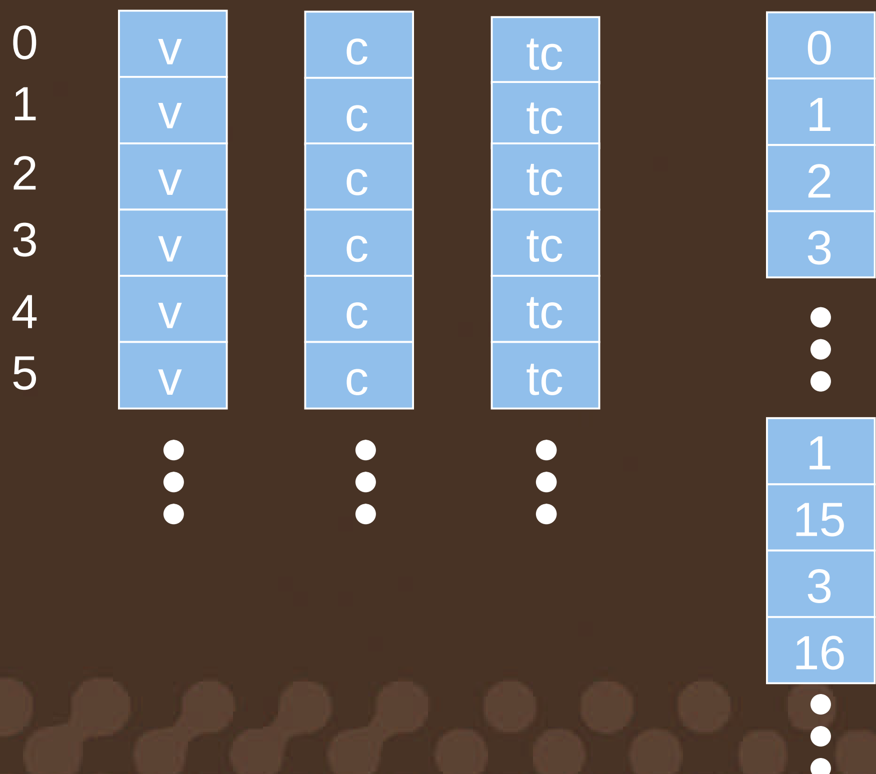



```
glDrawArrays( GL_TRIANGLES, 0, n );
```

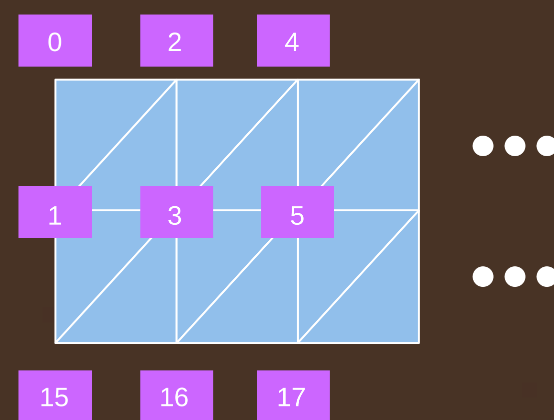


Drawing Geometric Primitives

- For indexed groups of vertices



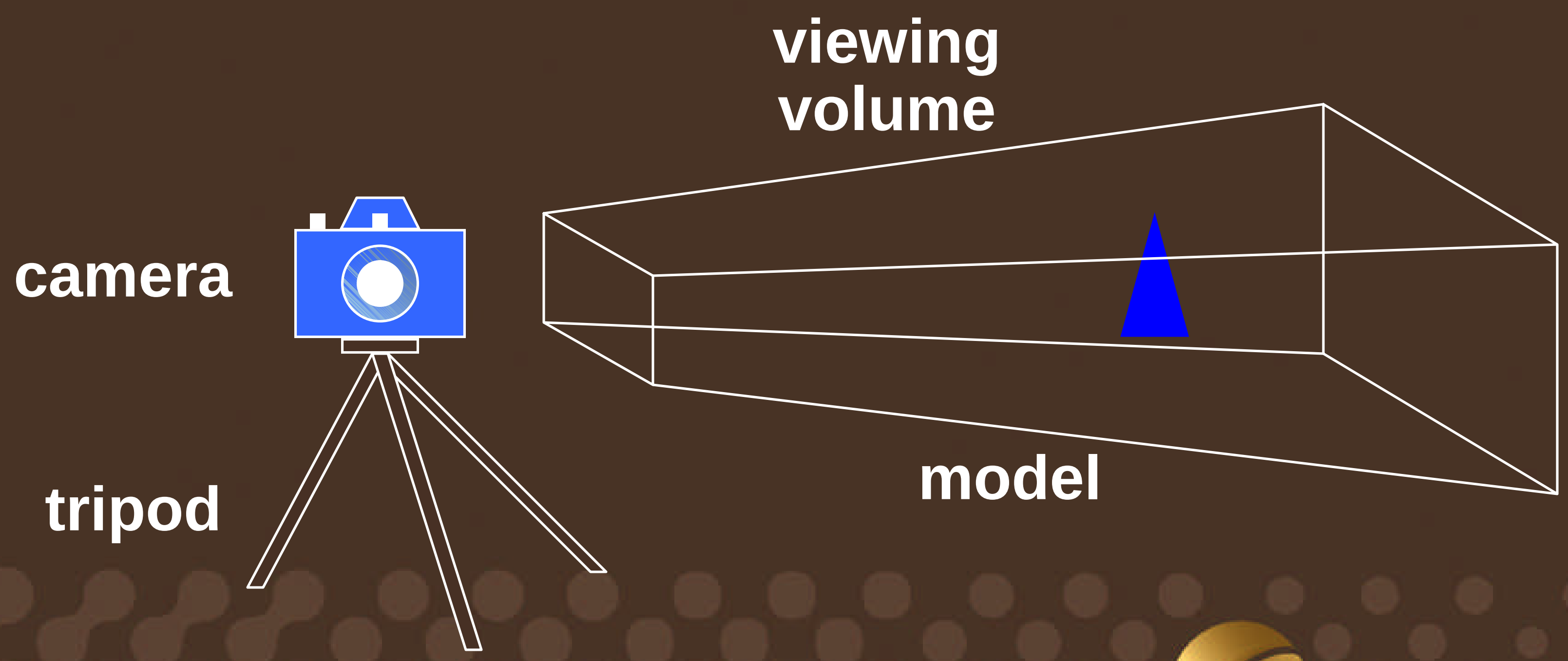

```
glDrawElements( GL_TRIANGLES,
  0, n, indices );
```



Transformations

Camera Analogy

- Rendering a 3D scene is just like taking a photograph

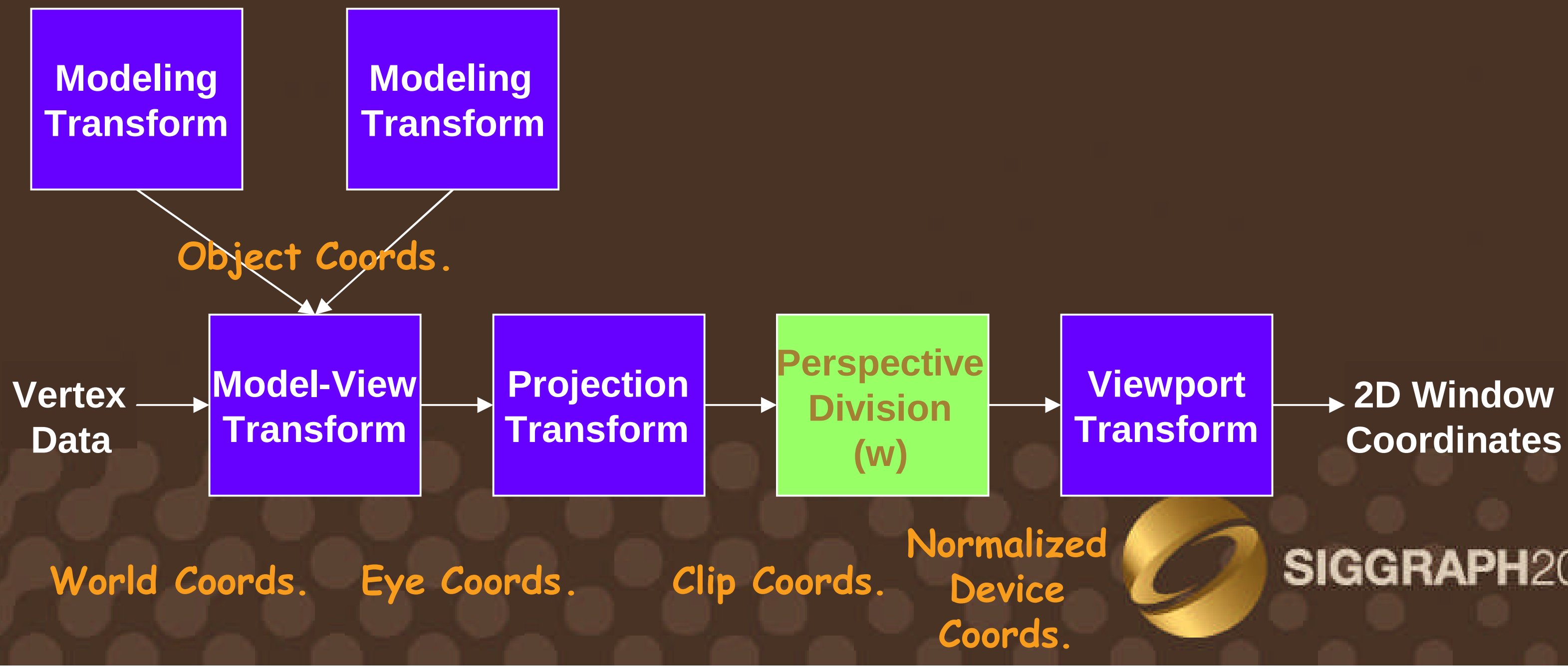


Camera Analogy and Transformations

- Projection transformation
  - adjust the lens of the camera
    - defines the size of the viewing volume, and how much of the world you can see
- Viewing transformation
  - set up the tripod
    - defines the position and orientation of the viewing volume
- Modeling transformations
  - move the model
- Viewport transformation
  - enlarge or reduce the physical photograph

The Transformation Pipeline

- Transformations take us from one "space" to another
  - All of our transforms are 4×4 matrices



3D Transformations

- A vertex is transformed by 4 x 4 matrices
  - product of matrix and vector is $\mathbf{M}\vec{v}$
  - matrices should always be post-multiplied
    - remember this when writing shaders
  - all matrices are stored column-major in OpenGL
    - this is opposite of what "C" programmers expect

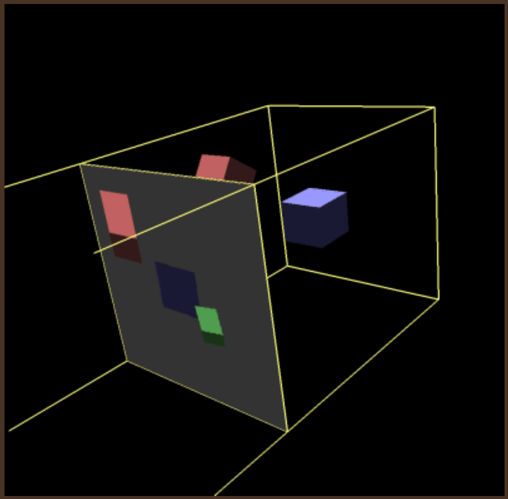
$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

“Configuring” a Viewing Frustum

- OpenGL projection model uses *eye coordinates*
  – the “eye” is located at the origin
  – looking down the –z axis
- Projection matrices use a six-plane model:
  – near (image) plane
  – far (infinite) plane
    - both are distances from the eye (positive values) for perspective projections
  – enclosing planes
    - top & bottom, left & right

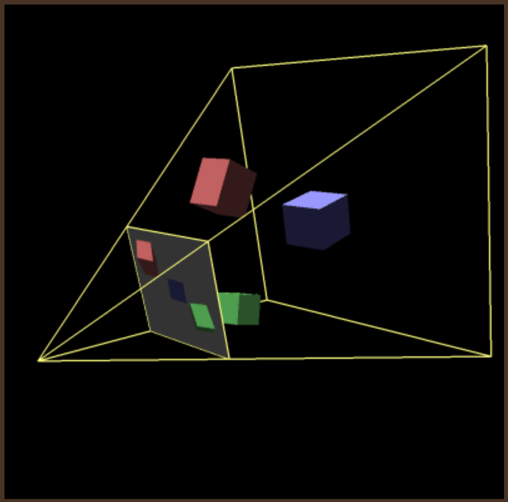
Types of Viewing Frusta

Orthographic View



$$O = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective View



$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

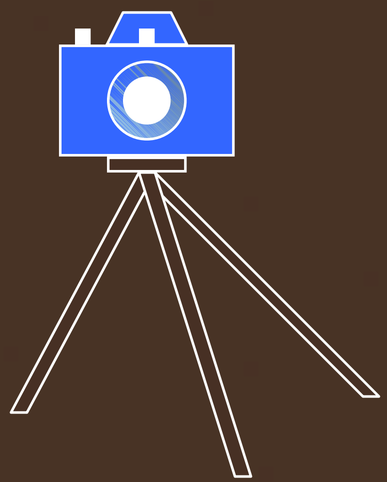
Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To "fly through" a scene
  - change viewing transformation and redraw scene

```
LookAt( eyex, eyey, eyez,
        lookx, looky, lookz,
        upx, upy, upz )
```

  - up vector determines unique orientation
  - careful of degenerate positions

tripod



Creating the LookAt Matrix

$$\hat{n} = \frac{\overrightarrow{look} - \overrightarrow{eye}}{\left\| \overrightarrow{look} - \overrightarrow{eye} \right\|}$$

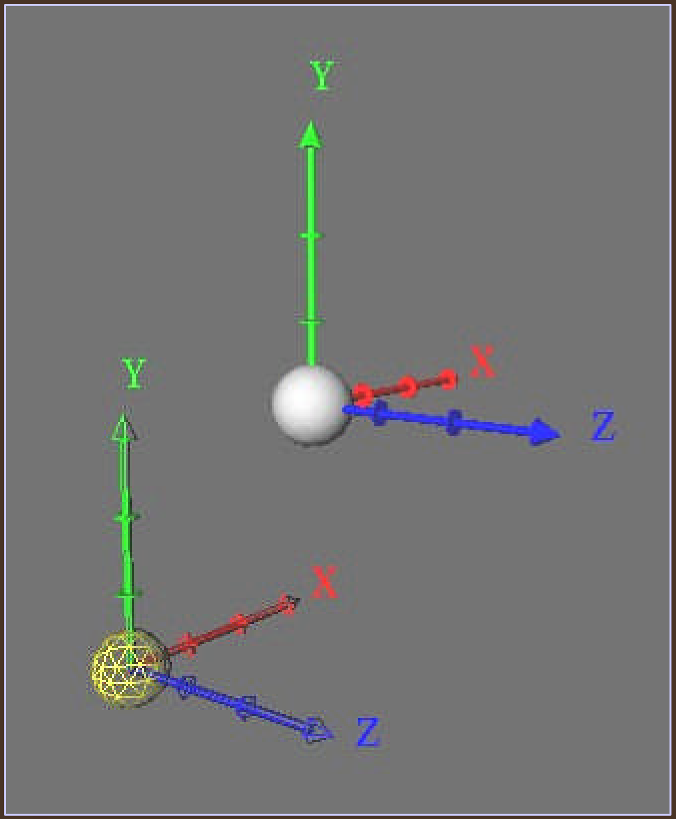
$$\hat{u} = \frac{\hat{n} \times \overrightarrow{up}}{\left\| \hat{n} \times \overrightarrow{up} \right\|} \Rightarrow \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ -n_x & -n_y & -n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\hat{v} = \hat{u} \times \hat{n}$$

Translation

- Move the origin to a new location

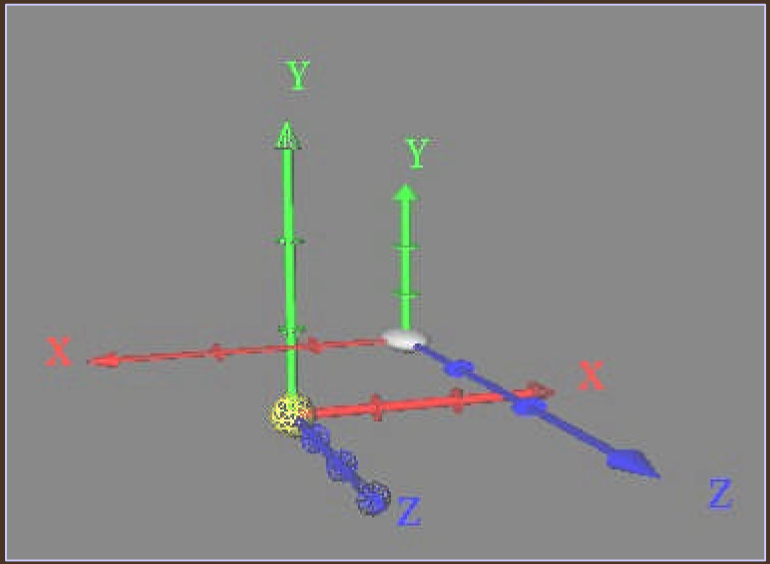
$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Scale

- Stretch, mirror or decimate a coordinate direction

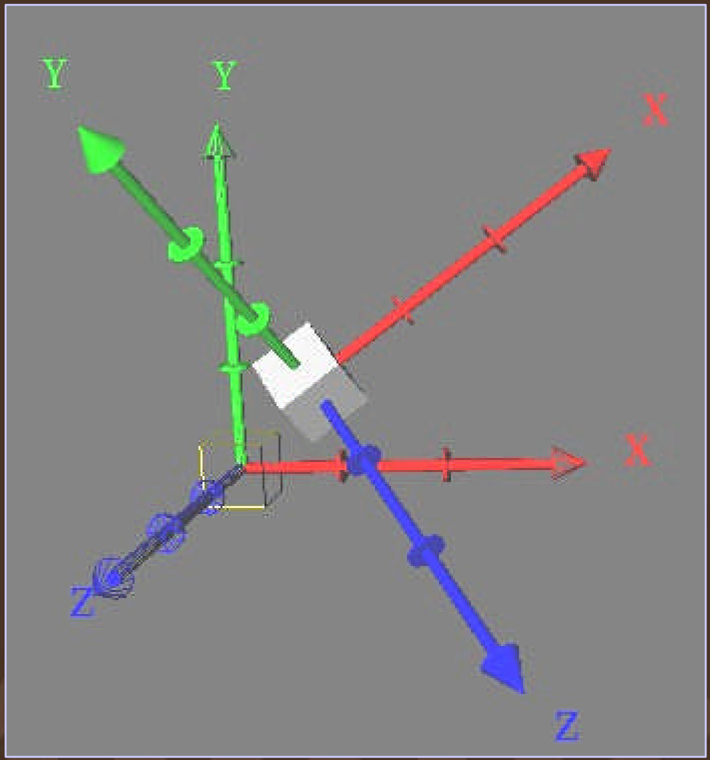
$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Note, there's a translation applied here to make things easier to see

Rotation

- Rotate coordinate system about an axis in space



Note, there's a translation applied here to make things easier to see

Rotation (cont'd)

$$\vec{v} = \begin{pmatrix} x & y & z \end{pmatrix}$$

$$\vec{u} = \frac{\vec{v}}{\|\vec{v}\|} = \begin{pmatrix} x' & y' & z' \end{pmatrix}$$

$$M = \vec{u}^t \vec{u} + \cos(\theta)(I - \vec{u}^t \vec{u}) + \sin(\theta) S$$

$$u^t u = \begin{pmatrix} x'^2 & x'y' & x'z' \\ x'y' & y'^2 & y'z' \\ x'z' & y'z' & z'^2 \end{pmatrix} \qquad S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

$$R_{\vec{v}}(\theta) = \begin{pmatrix} & & & 0 \\ & M & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

SIGGRAPH2008

OpenGL™

OpenGL Shading Language Overview

Bill Licea-Kane

AMD

OpenGL Shading Language

- Shading Language Details
- Trivial Examples
- Compiling and Using Shaders

Shading Language Details

- The OpenGL Shading Language 1.20.08
- The OpenGL Shading Language 1.30.08*
- The OpenGL ES Shading Language 1.0.14

Preprocessor

```
#
#define
#undef

#if
#ifdef
#ifndef
#else
#elif
#endif

#error
#pragma
```

```
// Comment
/* Comment */
```

Preprocessor

```
#version 110 // Shading Language 1.10 (IMPLICIT)
#version 120 // Shading Language 1.20
#version 130 // Shading Language 1.30

#extension: NAME : require|enable|warn|disable

#line LINE FILE

__LINE__
__FILE__
__VERSION__
```

Types

```
void

// Scalar
float int bool
uint                                                  // 1.30

// Vector
vec2 vec3 vec4
ivec2 ivec3 ivec4                                     // 1.30*
uvec2 uvec3 uvec4                                     // 1.30
bvec2 bvec3 bvec4

// Matrix
mat2 mat3 mat4 matCxR
```

Types

```
// Sampler
// float samplers
sampler1D sampler2D sampler3D samplerCube
sampler1DShadow sampler2DShadow
sampler1DArray sampler2DArray                    // 1.30
sampler1DShadowArray sampler2DShadowArray        // 1.30

// int samplers
isampler1D isampler2D isampler3D isamplerCube    // 1.30
isampler1DArray isampler2DArray                  // 1.30

// uint samplers
usampler1D usampler2D usampler3D usamplerCube    // 1.30
usampler1DArray usampler2DArray                  // 1.30
```

Containers

```
struct
// no qualifiers
// no bitfields
// no forward references
// no in-place definitions
// no anonymous structures

// 1D arrays                                          // 1.20*
[]
```

Scope

```
// (Outside Global)
   // Built-in functions

// Global
   // User-defined functions (Can hide Built-in)
   // Shared name space
   // Shared globals must be same type

// Local
   // RESTRICTION - No function prototypes
```

Storage Qualifiers

```
default
const

// global qualifiers
attribute                                    // 1.30*
uniform
varying                                      // 1.30*
centroid varying                             // 1.30*
in out                                       // 1.30
centroid in                                  // 1.30
centroid out                                 // 1.30
smooth flat noperspective                    // 1.30
```

Storage Qualifiers

```
// invariant qualifier
invariant

// parameter qualifiers
in out inout                                        // 1.30
```

Operators

```
()              // grouping
[]              // array and component
()              // constructor
.               // field select and swizzle

++ --           // postfix

++ --           // prefix
+ - !           // prefix
```

Operators

```
+ - * /                 // binary
< <= > >=               // relational
== !=                   // equality
&& ^^ ||                // logical

?:                      // selection

= += -= *= /=           // assignment
```

SIGGRAPH2008

Integer operators

```
~                                // prefix
%                                // binary
<< >> & ^ |                      // bitwise
%= <<= >>= &= ^= |=              // assignment
```

Constructors

```
// Scalar
float() int() bool()
uint()                                         // 1.30
// Vector
vec2() vec3() vec4()
ivec2() ivec3() ivec4()
uvec2() uvec3() uvec4()                        // 1.30
bvec2() bvec3() bvec4()

// Matrix
mat2() mat3() mat4() matCxR()

// Struct
// Array
```

SIGGRAPH2008

Components

```
// Vector
.xyzw .rgba .stpq [i]

// Matrix
[i] [i][j]
```

Flow Control

```
// expression ? TrueExpression : FalseExpression
// if, if-else
// for, while, do-while
// return, break, continue
// discard (fragment only)
```

Vector Matrix Operations

```
mat4 m4, n4;
vec4 v4;

vec4 first  = m4 * v4; // matrix * vector
vec4 second = v4 * n4; // vector * matrix
mat4 third  = m4 * n4; // matrix * matrix
```

Functions

```
// Parameter qualifiers
in out inout
const in
// Functions are call by value, copy in, copy out
// NOT exactly like C++
//
// Examples
vec4 function( const in vec3 N, const in vec3 L );
void f( inout float X, const in float Y );
```

Special Variables

```
// Vertex
vec4  gl_Position;          // must be written to
vec4  gl_ClipVertex;        // may be written to 1.30*
float gl_ClipDistance[];    // may be written to 1.30
float gl_PointSize;         // may be written to

// Fragment
vec4  gl_FragCoord;         // may be read from
bool  gl_FrontFacing;       // may be read from
vec4  gl_FragColor;         // may be written to
vec4  gl_FragData[i];       // may be written to
float gl_FragDepth;         // may be written to
```

Built-in attributes

This page intentionally blank

(Gone in ES 2.0, Marked deprecated in GLSL 1.30)

Built-in varying

This page intentionally blank

(Already gone or marked deprecated as well.)

Built-in uniforms

This page intentionally blank

(Parenthetical intentionally left blank*)

Built-in Functions

```
// angles and trigonometry
// exponential
// common
// interpolations
// geometric
// vector relational
// texture
// shadow
// noise
// vertex
vec4 ftransform( void );                                  // 1.30*
// fragment
genType dFdx( genType P );
gentype dFdy( genType P );
genType fwidth( genType P );
```

Really Simple Shaders

Smallest OpenGL Shaders

```
// Vertex Shader
//
#version 120 // Shading Language 1.20

void main( void )
{
    gl_Position = vec4( 0.0 );
}

// Fragment Shader
//
#version 120 // Shading Language 1.20

void main( void ) { }
```

Smallest OpenGL Shaders

```
// Vertex Shader
//
#version 130 // Shading Language 1.30

void main( void )
{
    gl_Position = vec4( 0.0 );
}

// Fragment Shader
//
#version 130 // Shading Language 1.30

void main( void ) { }
```

Small OpenGL Shaders

```
// Vertex Shader
//
#version 120 // Shading Language 1.20
uniform   mat4 matMVP;
attribute vec4 mPosition;
attribute vec4 mColor;
varying   vec4 fColor;
void main( void )
{
    gl_Position = matMVP * mPosition;
    fColor = mColor;
}

// Fragment Shader
//
#version 120 // Shading Language 1.20
varying   vec4 fColor;
void main( void )
{
    gl_FragData[0] = fColor;
}
```

Small OpenGL Shaders

```
// Vertex Shader
//
#version 130 // Shading Language 1.30
uniform   mat4 matMVP;
in        vec4 mPosition;              // attribute vec4 mPosition;
in        vec4 mColor;                 // attribute vec4 mColor;
out       vec4 fColor;                 // varying vec4 fColor
void main( void )
{
    gl_Position = matMVP * mPosition;
    fColor = mColor;
}

// Fragment Shader
//
#version 130 // Shading Language 1.30
in        vec4 fColor;                 // varying vec4 fColor
void main( void )
{
    gl_FragData[0] = fColor;
}
```

Compiling Shaders

SIGGRAPH2008

Creating a Shader Program

- Similar to compiling a “C” program
  - compile, and link
    - OpenGL ES supports both online and offline compilation
- Multi-step process
  - create and compile shader objects
  - attach shader objects to program
  - link objects into executable program

Shader Compilation (Part 1)

- Create and compile a Shader (with online compilation)

```
GLunit shader = glCreateShader( shaderType );
const char* str = “void main() {…}”;
glShaderSource( shader, 1, &str, 0 );
glCompileShader( shader );
```

- ***shaderType*** **is either**
  - **GL_VERTEX_SHADER**
  - **GL_FRAGMENT_SHADER**

Shader Compilation (Part 2)

- Checking to see if the shader compiled (online compilation)

```
GLint  compiled;
glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );
if ( !compiled ) {
  GLint len;
  glGetShaderiv( shader, GL_INFO_LOG_LENGTH, &len );
  std::string  msgs( ' ', len );
  glGetShaderInfoLog( shader, len, &len, &msgs[0] );
  std::cerr << msgs << std::endl;
  throw shader_compile_error;
}
```

Shader Program Linking (Part 1)

- Create an empty program object

```
GLuint  program = glCreateProgram();
```

- Associate shader objects with program

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

- Link program

```
glLinkProgram( program );
```

Shader Program Linking (Part 2)

- Making sure it worked

```
GLint  linked;
glGetProgramiv( program, GL_LINK_STATUS, &linked );
if ( !linked ) {
  GLint len;
  glGetProgramiv( program, GL_INFO_LOG_LENGTH, &len );
  std::string  msgs( ' ', len );
  glGetProgramInfoLog( program, len, &len, &msgs[0] );
  std::cerr << msgs << std::endl;
  throw shader_link_error;
}
```

Using Shaders in an Application

- Need to turn on the appropriate shader

```
glUseProgram( program );
```

Associating Shader Variables and Data

- Need to associate a shader variable with an OpenGL data source
  - vertex shader attributes → app vertex attributes
  - shader uniforms → app provided uniform values
- OpenGL relates shader variables to indices for the app to set
- Two methods for determining variable/index association
  - specify association before program linkage
  - query association after program linkage

Determining Locations After Linking

- Assumes you already know the variables' name

```
GLint  idx =
  glGetAttribLocation( program, “name” );

GLint  idx =
  glGetUniformLocation( program, “name” );
```

Initializing Uniform Variable Values

- Uniform Variables

```
glUniform4f( index, x, y, z, w );


GLboolean  transpose = GL_TRUE;  // Since we're C programmers
GLfloat  mat[3][4][4] = { … };
glUniformMatrix4fv( index, 3, transpose, mat );
```

OpenGL Shading Language

- Acknowledgements
  - John Kessenich (Intel)
  - David Baldwin
  - Randi J. Rost (Intel)
  - Robert Simpson (AMD)
  - Benj Lipchack (Apple)
  - Dave Shreiner (ARM)
  - …and ARB Contributors

Lighting and Materials

SIGGRAPH2008

Lighting Motivation

- More realistic image
- Visual cues to object shape and location

Illumination with Shaders

- Do-It-Yourself Lighting
  - No built-in illumination support in modern APIs
- Back to the basics
  - Fundamental graphics algorithms are key
- Complete flexibility
  - Any lighting model desired

Quick Refresher

- Coordinate space
- Surface normals

Coordinate Frame

- Consistent coordinate frame required for scene composition
  - Allows objects to appear in the proper place
- Common Solutions
  - World coordinates
  - Eye Coordinates

Eyespace Coordinates

- Centers coordinate frame at eye
  - Viewer position becomes (0,0,0)
- View direction is typically (0,0,±1)
  - Fixed-function OpenGL used (0,0,-1)
- Somewhat more efficient than worldspace
  - Constant direction and eye position

Surface Normal

- Vector normal to the tangent plane of the surface
- Three components - (x,y,z)
  - Homogeneous component zero - (x,y,z,0)
- Unit length
  - $X^2 + Y^2 + Z^2 == 1.0$
- Transformed by the inverse-transpose of the matrix used for positions

Illumination

Defining Illumination

- Light leaving an object and seen by a viewer
  – Direct
    - Light striking an object from a light source directly
  – Indirect
    - Light striking an object, after bouncing off other objects
  – Emitted
    - Light produced by the object

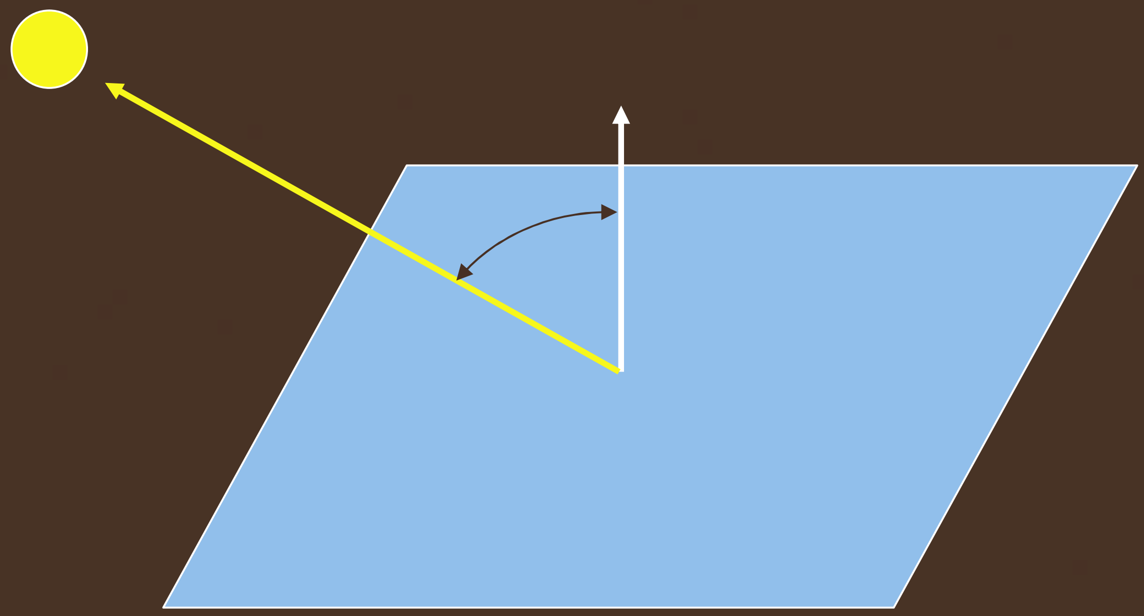
Mathematics of Illumination

- Sum of contributions from all sources of light
  - Direct
    - Summed over all light sources in the scene
    - Diffuse or view independent
    - Specular or view dependent
  - Indirect or Ambient
    - Summed either globally or computed individually for each light
  - Emissive or Emitted light
    - Accumulated once for the object

Diffuse Illumination Component

- Non-shiny direct light
- Lambertian model
  - Works well for many common materials
- Intensity derived from angle of incidence

Diffuse Illumination Diagram



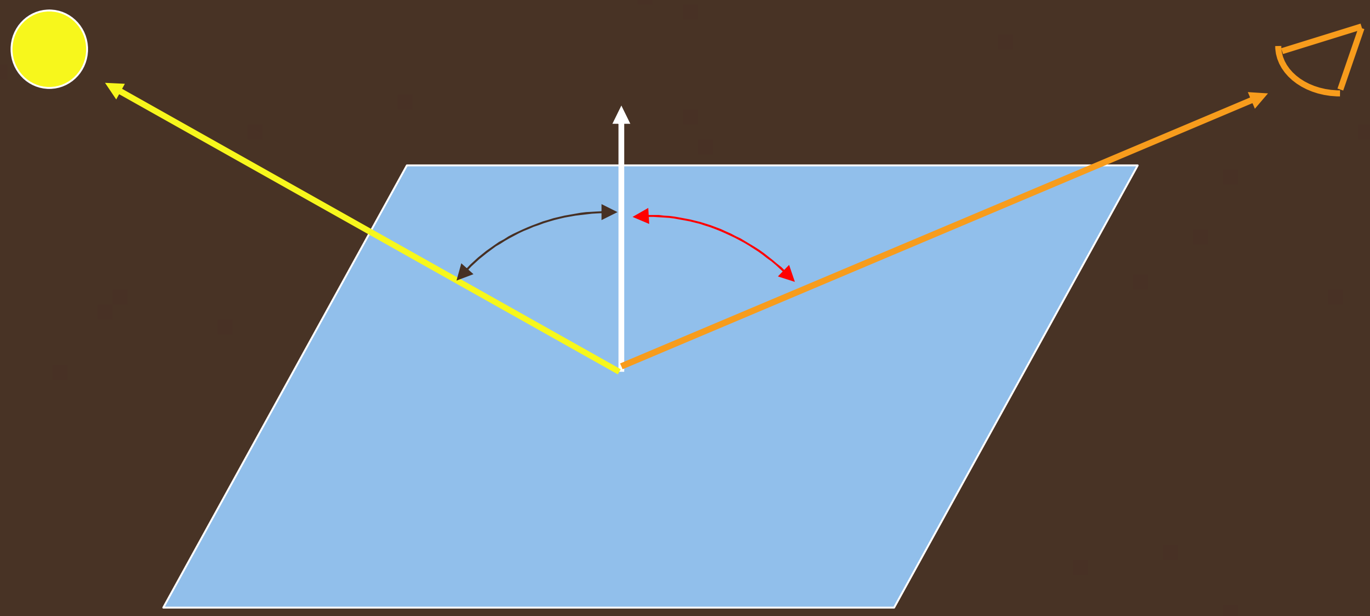
Diffuse Illumination Code

```
// Compute light direction
vec3 light_dir = normalize( light_pos - pos );

// Compute the lighting
float intensity = dot( normal, light_dir );
intensity = clamp( intensity, 0.0, 1.0 );
```

Specular Illumination Component

- Blinn-Phong model
  - Simple and efficient
  - Good for plastic (non-physical)
- Intensity derived from view vector, normal vector, and light vector

Specular Illumination Diagram



SIGGRAPH2008

Specular Illumination Code

```
// Compute the view vector
vec3 view_dir = normalize( -pos );
// Compute the half-angle vector
vec3 half = normalize( view_dir + light_dir );
// Compute the specular intensity
float spec = dot( half, normal );
spec = clamp( spec, 0.0, 1.0 );
spec = pow( spec, shininess );
```

Computational Frequency

- Per-vertex
  - Simple, fairly low-quality
- Per-fragment
  - Moderate difficulty, high-quality
- Hybrid
  - Different frequencies for different components

Interpolation

- Process of converting vertex values to fragment values
- Performed on all variables declared varying
- Interpolation mode and shaders determine shading frequency
- Not all values interpolate linearly
  - Normals must be renormalized per-fragment

OpenGL™ SIGGRAPH2008

Per-Vertex Illumination

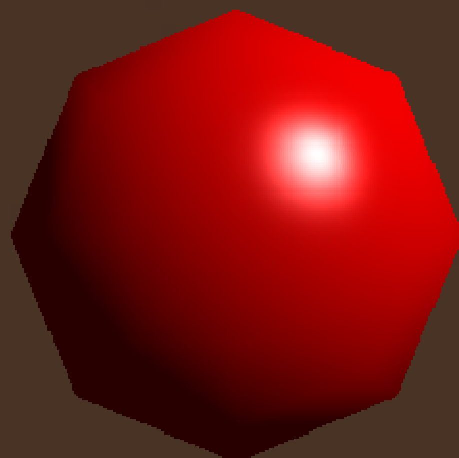
- All illumination values are computed in the vertex shader
- Illumination values are passed to the fragment shader in varyings
- May miss high-detail light effects
  - Lower quality result
- Typically faster
  - Fewer vertices than fragments

Per-Vertex Illumination

Per-Fragment Illumination

- Illumination values are calculated in the fragment shader
- Values used for illumination are computed in the vertex shader and interpolated
  - Eye space position and normal
- Several values must be derived per-pixel
  - Half-angle vector, normalized vectors

Per-Fragment Illumination



SIGGRAPH2008

Light Types

- Point light
  - Similar to a light bulb
  - Used for samples in this talk
- Directional light
  - Similar to sunlight
  - No position, all light rays are parallel
- Spot light
  - Point light with special focusing

SIGGRAPH2008

Materials

Material Properties

- Shades of gray are boring
- Materials provide the proper look
- Can include all factors feeding into lighting
  - Colors
  - Surface roughness

Material Colors

- Can provide separate colors for all illumination components
  - Specular, ambient, diffuse, etc.
- Relative combinations emulate different physical materials
  - Metals: diffuse_material == specular_material
  - Plastics: specular_material == white

Material Coloring Code

```
uniform vec3 diffuse_material;
uniform vec3 specular_material;
...

// apply the diffuse material color
vec3 diffuse_color = diffuse_material * diffuse_intensity;
// apply the specular material color
vec3 specular_color = specular_material * specular_intensity;
```

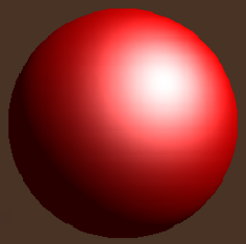
Material Color Application

- Material color application is independent of illumination
  - Can be applied per-fragment, even when illumination is per-vertex
  - Illumination components are passed as varyings to the fragment shader, then the material is applied

Shininess / Roughness

- Specular power (k)
  - Provides tightness on specular highlight
  - Often interpretted as k = 1 / roughness

k = 64



k = 8

Textures

Texturing

- Applying an image to the object surface
  - Most often 2D
- Thought of as a set of varying material properties
  - All properties mentioned so far may be provided via textures

Texture Objects

- Represented by a GLuint in the API
  - Unique names generated by glGenTextures
- Encapsulates all texture state
  - Wrap and filter
- Created by glBindTexture

Loading and Configuring Textures

- Create & Bind texture object
- Load base image
- Load mipmaps [optional]
- Specify texture object parameters
  – Filter state
  – Wrap state

Generating & Binding Names

```
GLuint tex_name;

//Generate one texture name
glGenTextures( 1, &tex_name );

//Bind the texture
glBindTexture( GL_TEXTURE_2D, tex_name );
```

Specifying a Texture Image

- glTexImage2D arguments
  - Dimensions: width and height
  - Internal format: preferred HW format
    - GL_RGBA8, GL_LUMINANCE8, etc
  - Format: format of input data
    - GL_RGB, GL_RGBA, etc
  - Type: data type of input data
    - GL_UNSIGNED_BYTE, GL_FLOAT, etc

Base Formats

Format	Interpretation
GL_RGB	(R, G, B, 1.0)
GL_RGBA	(R, G, B, A)
GL_LUMINANCE	(L, L, L, 1.0)
GL_LUMINANCE_ALPHA	(L, L, L, A)
GL_INTENSITY	(I, I, I, I)
GL_ALPHA	(0.0, 0.0, 0.0, A)

Data Types

- Unsigned byte
- Unsigned short
- Short
- Int
- Float

Applying Texture

- Sampler Variable
  - Used to select which texture object
- Shading language function
  - texture2D(sampler2D map, vec2 p)
    - Returns vec4 from point p in map
  - Others are also available

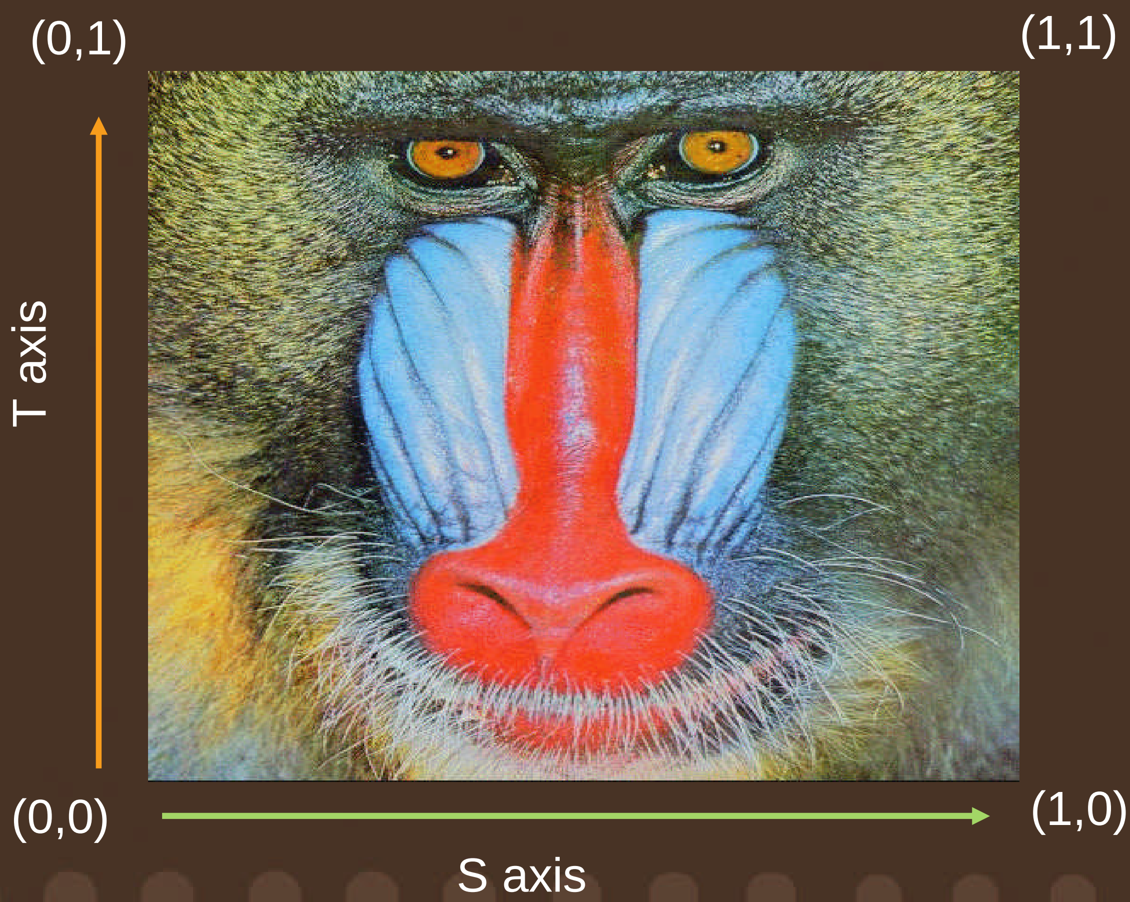
Shader Code

```
sampler2D my_texture;

//sample the lower-left of the texture
vec2 coord = vec2( 0.0, 0.0);
vec4 color = texture2D( my_texture, coord);
```

OpenGL

SIGGRAPH2008

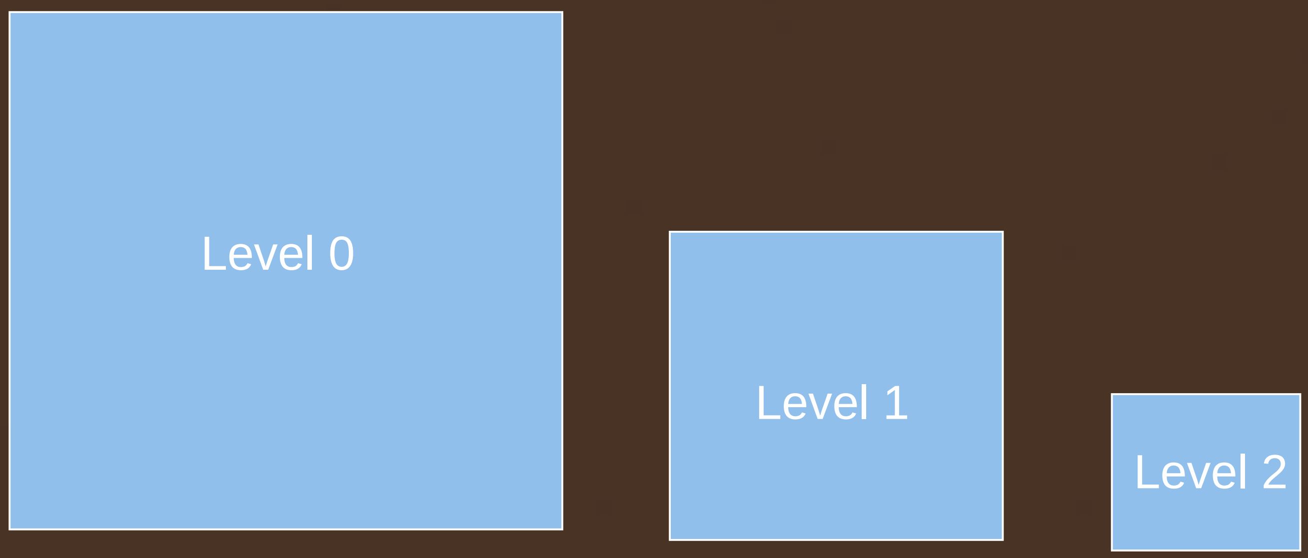
Texture Coordinates



Mipmaps

- Smaller versions of base image
- Allow for better filtering
  - Reduce aliasing
- Smaller levels are ½ size in each dimension
  - 128x128 – 64x64 – 32x32 …
- Covers all levels to 1x1

Mipmaps



Generating Mipmaps

- OpenGL allows the automatic generation of mipmaps
  - Enable prior to calling glTexImage

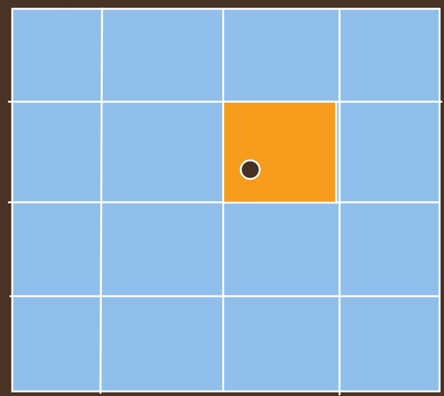
```
glTexParameteri( GL_TEXTURE_2D,
  GL_GENERATE_MIPMAP, GL_TRUE);
```

Filtering

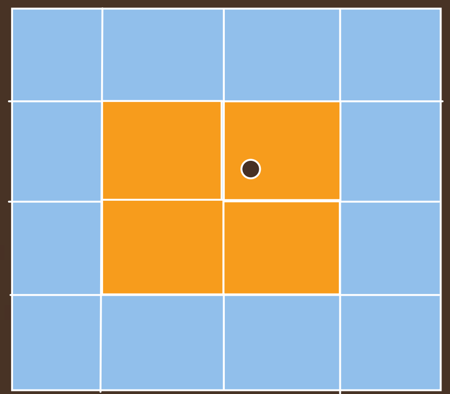
- Specified through glTexParameter
- Controls the manner of fetching a texel
  - GL_NEAREST – point sampling (lowest quality)
  - GL_LINEAR – blend between 4 nearest texels
  - GL_LINEAR_MIPMAP_NEAREST – Select one mipmap and blend the 4 nearest texels
  - GL_LINEAR_MIPMAP_LINEAR – Select two mipmaps and blend the 4 nearest texels from each

Filtering

Nearest Filtering



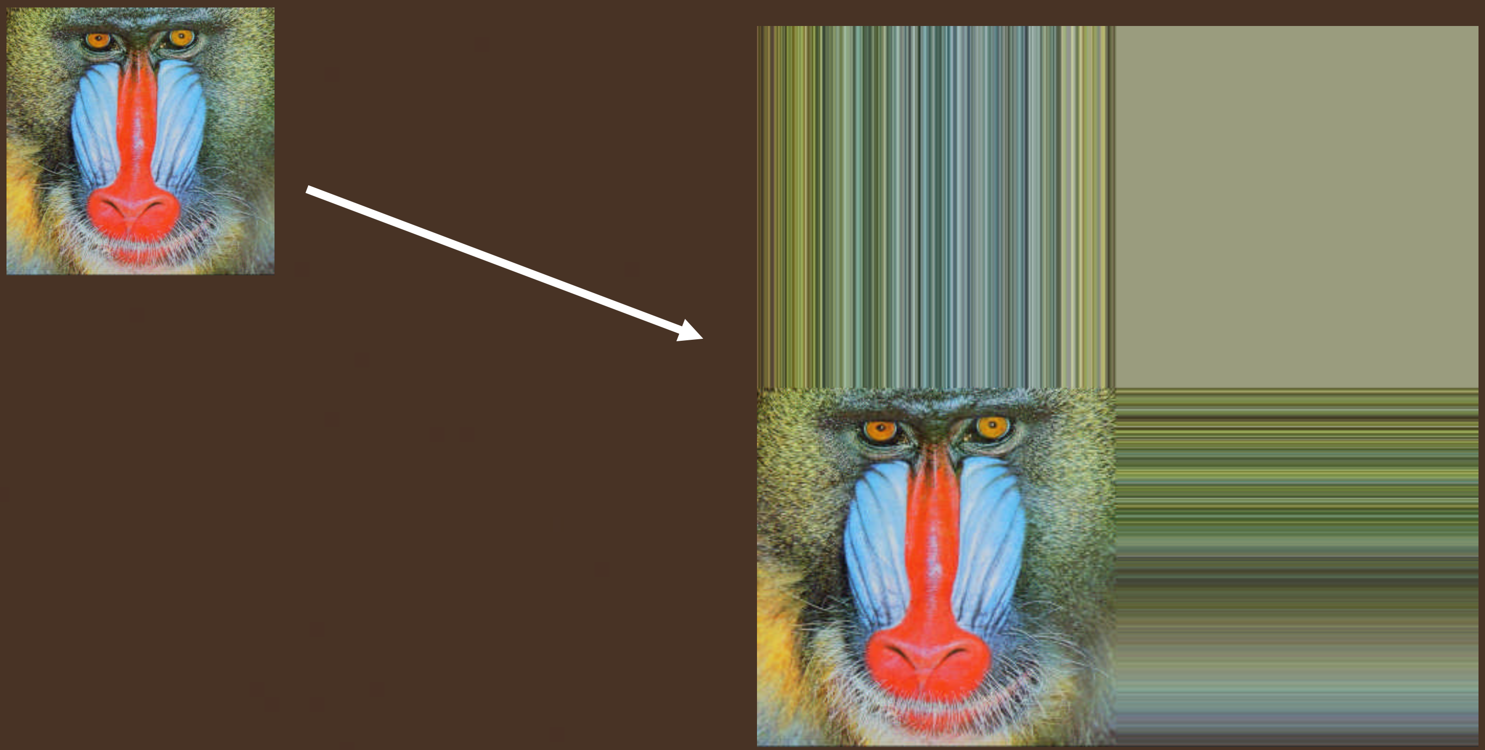
Linear Filtering



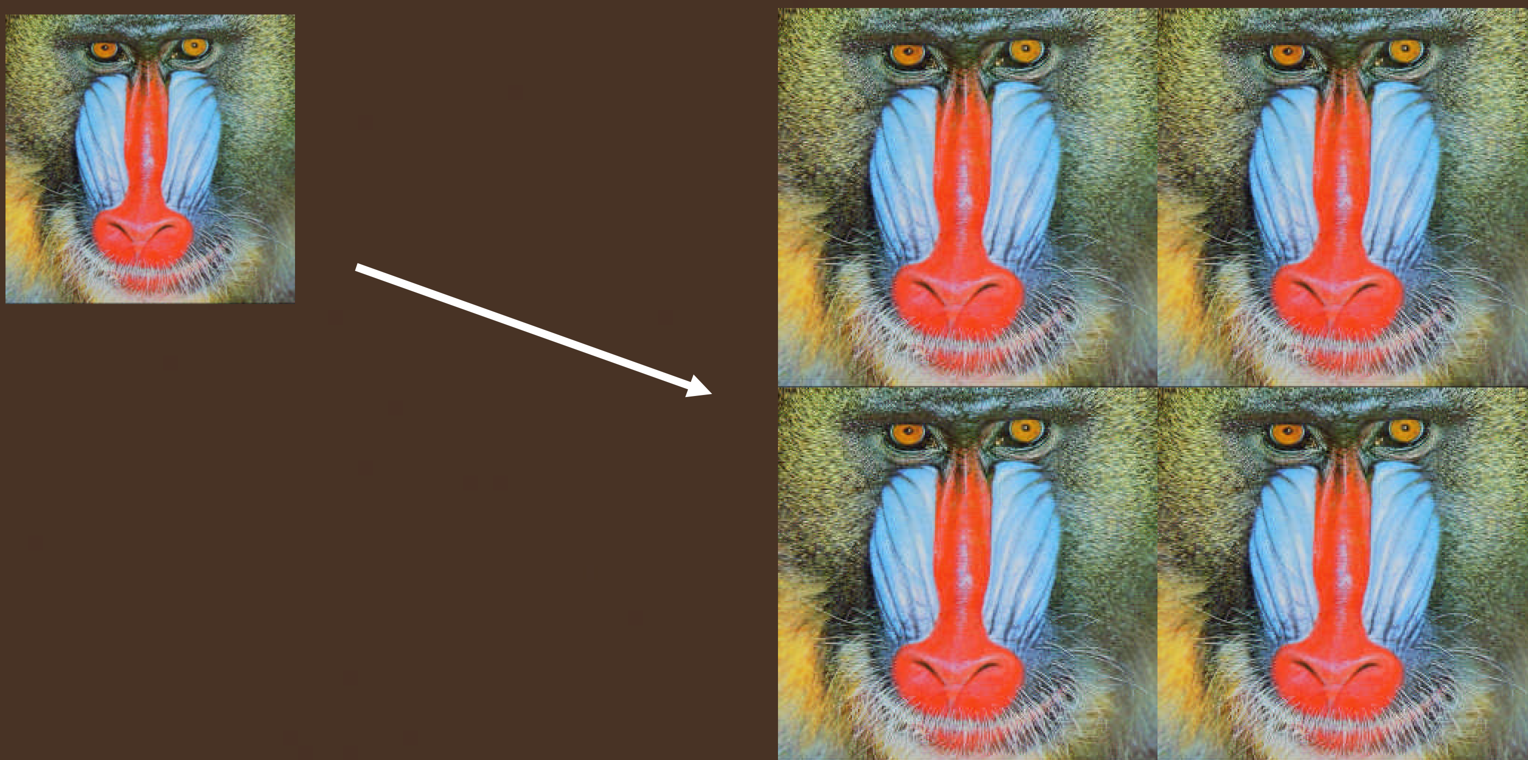
Wrap State

- Specified through glTexParameter
- Controls boundary behavior for texture filtering
  - GL_CLAMP_TO_EDGE
  - GL_REPEAT
  - GL_MIRRORRED_REPEAT

Clamp Texture Wrapping



Repeat Texture Wrapping



Mirror Texture Wrapping

Applying the Texture to the Shader

```
// Declare the sampler
uniform sampler2D diffuse_mat;


// Apply the material color
vec3 diffuse = intensity * texture2D(
  diffuse_mat, coord ).rgb;
```

Full Fragment Shader

```
varying vec3 eye_norm;
varying vec4 eye_pos;
varying vec2 tex_coord;

uniform vec3 light_dir;
uniform sampler2D my_texture;

void main() {
 vec3 norm = normalize( eye_norm );  //Normalize the normal
 vec3 view = normalize( -eyePos.xyz );
 float diffuse = clamp( dot( norm, light_dir ), 0.0, 1.0 );
 vec3 hvec = normalize( light_dir + view );    // Compute the half angle vector
 float specular = clamp( dot( hvec, norm ), 0.0, 1.0 );
 specular = pow( specular, 64.0 );
 vec3 tex_color = texture2D( my_texture, tex_coord ).rgb;  //fetch the texture
 vec3 color = clamp( ( ( diffuse + 0.2 ) * tex_color + specular ), 0.0, 1.0 );
 gl_FragColor = vec4( color, 1.0 );
}
```

Thanks

- Fellow presenters
- NVIDIA DevTech Team
- NVIDIA OpenGL Driver Team

Shader Examples

- Vertex Shaders
  - Moving vertices: height fields
  - Per vertex lighting: height fields
  - Per vertex lighting: cartoon shading
- Fragment Shaders
  - Per vertex vs per fragment lighting: cartoon shader
  - Samplers: reflection Map
  - Bump mapping

Height Fields

- A **height field** is a function $y = f(x,z)$ where the *y* value represents a quantity such as the height above a point in the *x-z* plane.
- Height fields are usually rendered by sampling the function to form a rectangular mesh of triangles or rectangles from the samples $y_{ij} = f(x_i, z_j)$ and then rendering the polygons

Displaying a Height Field

- Defining a rectangular mesh

```
for(i=0;i<N;i++) for(j=0;j<N;j++) data[i][j]=f( i, j, time);
```

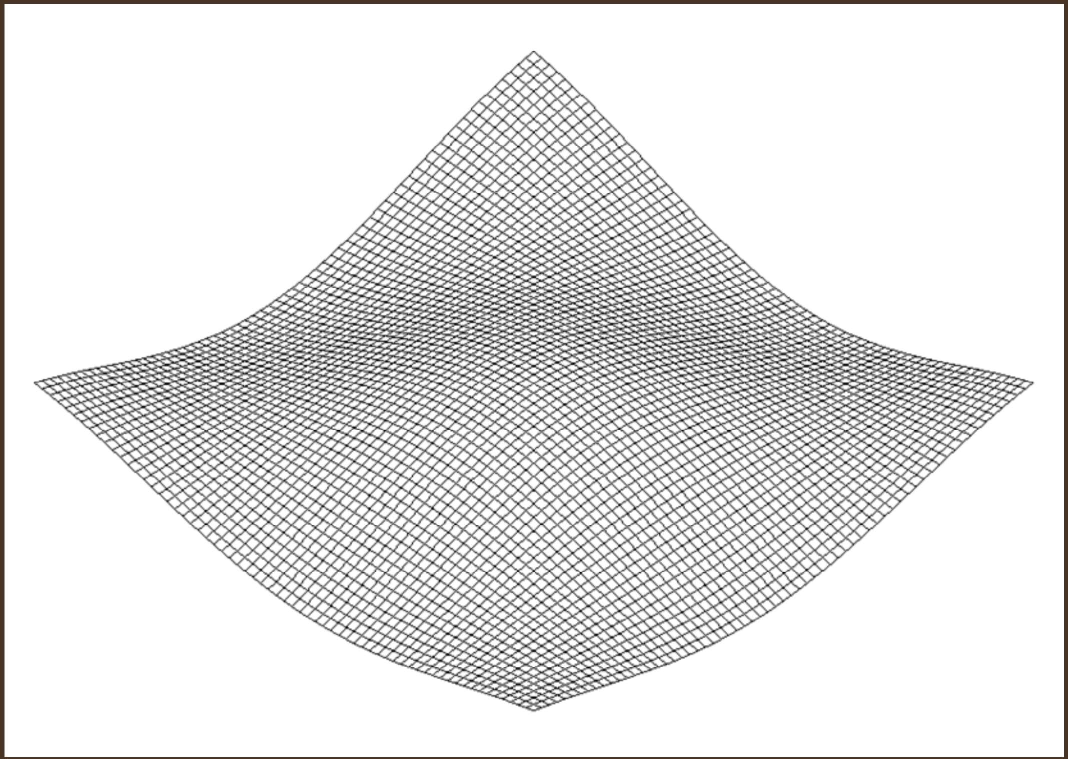
- Displaying a mesh

```
glBegin(GL_LINE_LOOP);
    glVertex3f((float)i/N, data[i][j], (float)j/N);
    glVertex3f((float)i/N, data[i][j+1], (float)(j+1)/N);
    glVertex3f((float)(i+1)/N, data[i+1][j+1], (float)(j+1)/N);
    glVertex3f((float)(i+1)/N, data[i+1][j], (float)(j)/N);
glEnd();
```

Time varying vertex shader

```
uniform float time; /* in milliseconds */

void main()
{
   vec4  t = gl_Vertex;
   t.y = 0.1*sin(0.001*time + 5.0*gl_Vertex.x)*
                      sin(0.001*time+5.0*gl_Vertex.z);
   gl_Position = gl_ModelViewProjectionMatrix * t;
   gl_FrontColor = gl_Color;
}
```

Mesh Display



Adding Lighting

- Solid Mesh:

  glBegin(GL_POLYGON);

- We must add lighting
- Must do per-vertex lighting in the shader if we use a vertex shader for time-varying mesh

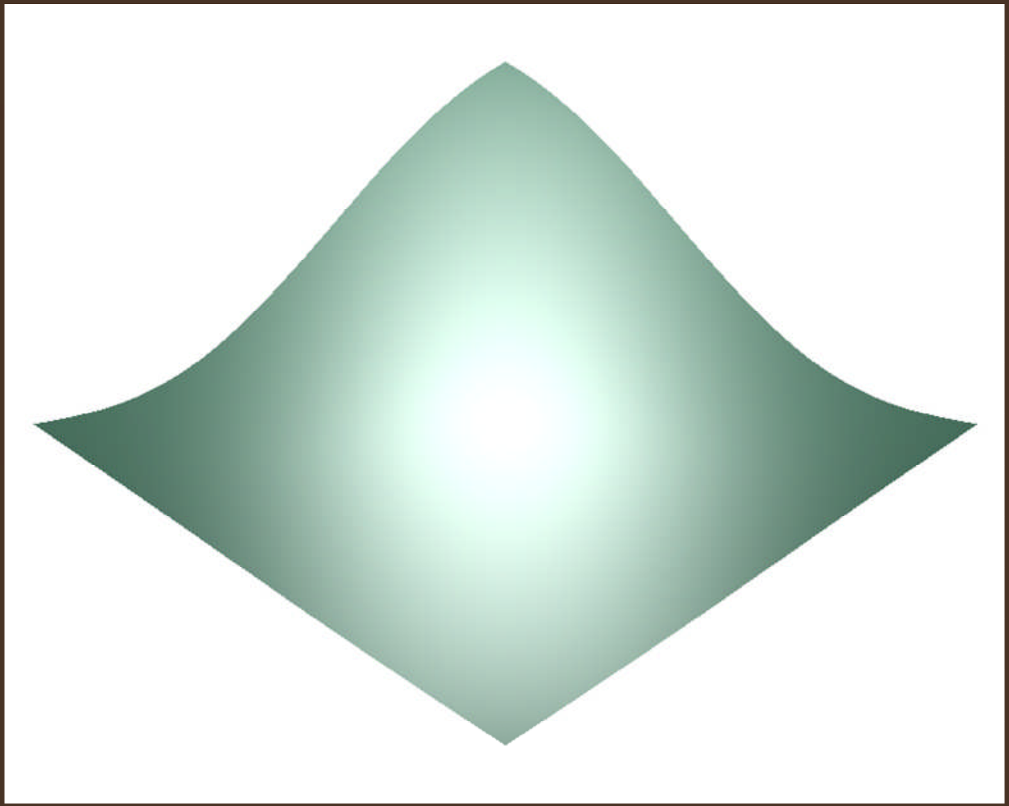
Mesh Shader

```
uniform float time;
void main()
{
   vec4  t = gl_Vertex;
    t.y = 0.1*sin(0.001*time+5.0*gl_Vertex.x)
             *sin(0.001*time+5.0*gl_Vertex.z);
    gl_Position = gl_ModelViewProjectionMatrix * t;

    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;
    vec4 eyeLightPos = gl_LightSource[0].position;
```

Mesh Shader (cont)

```
vec3 N = normalize(gl_NormalMatrix * gl_Normal);
vec3 L = normalize(eyeLightPos.xyz - eyePosition.xyz);
vec3 E = -normalize(eyePosition.xyz);
vec3 H = normalize(L + E);
float Kd = max(dot(L, N), 0.0);
float Ks = pow(max(dot(N, H), 0.0), gl_FrontMaterial.shininess);
ambient = gl_FrontLightProduct[0].ambient;
diffuse = Kd*gl_FrontLightProduct[0].diffuse;
specular = Ks*gl_FrontLightProduct[0].specular;
gl_FrontColor = ambient+diffuse+specular;
}
```

Shaded Mesh



Cartoon Shader

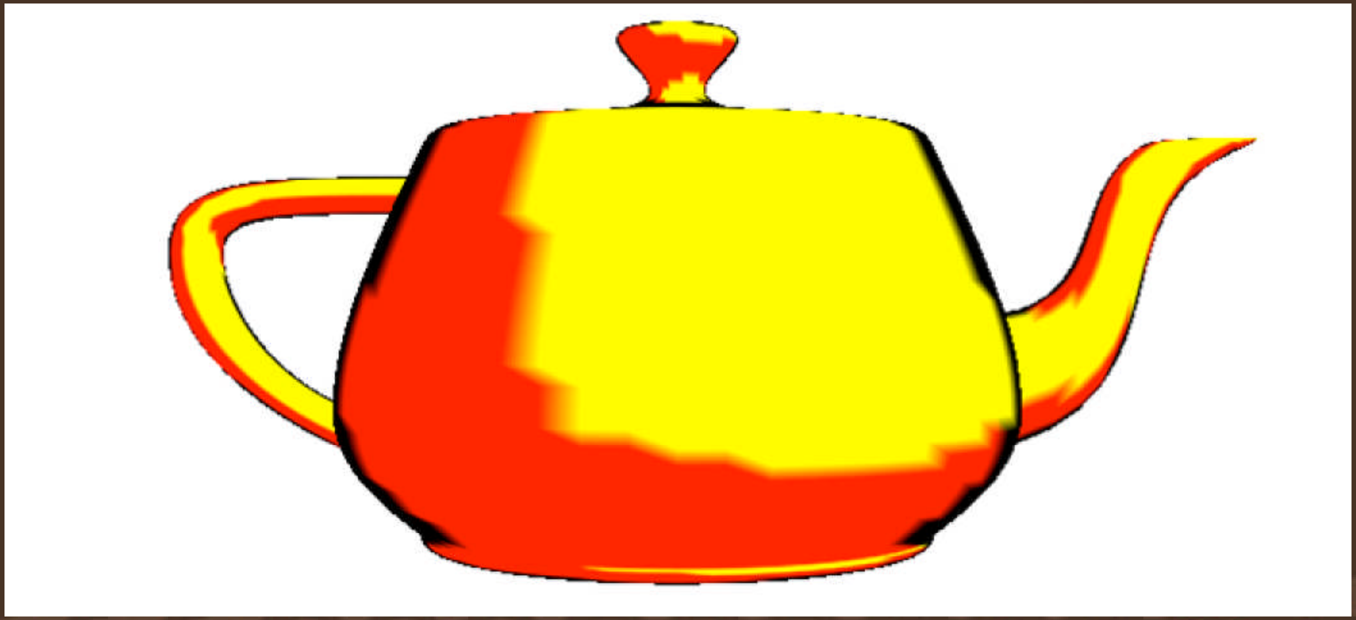
- This vertex shader uses only two colors but the color used is based on the orientation of the surface with respect to the light source
- Normal vector provided by the application through glNormal function
- A third color (black) is used for a silhouette edge

Cartoon Shader Code

```
void main()
{
   const vec4 yellow = vec4(1.0, 1.0, 0.0, 1.0);
   const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

   vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;
   vec4 eyeLightPos = gl_LightSource[0].position;
   vec3 N = normalize(gl_NormalMatrix * gl_Normal);
   vec3 L = normalize(eyeLightPos.xyz - eyePosition.xyz);
   float Kd = max(dot(L, N), 0.0);
   if(Kd > 0.6) gl_FrontColor = yellow;
     else gl_FrontColor = red;
}
```

Adding a Silhouette Edge

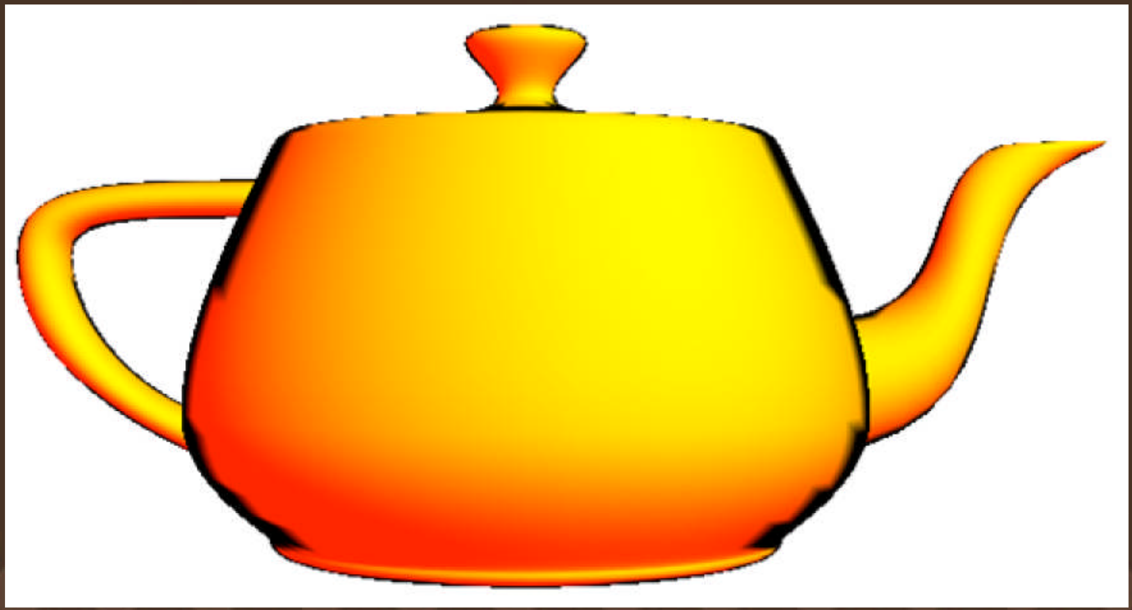
```
const vec4 black = vec4(0.0, 0.0, 0.0, 1.0);

vec3 E = -normalize(eyePosition.xyz);

if(abs(dot(E,N))<0.25)      gl_FrontColor  = black;
```



Smoothing

- We can get rid of some of the jaggedness using the mix function in the shader

```
gl_FrontColor = mix(yellow, red, Kd);
```



Fragment Shader Examples

- Per fragment lighting: Cartoon shader
- Texture Mapping: Reflection Map
- Bump Mapping

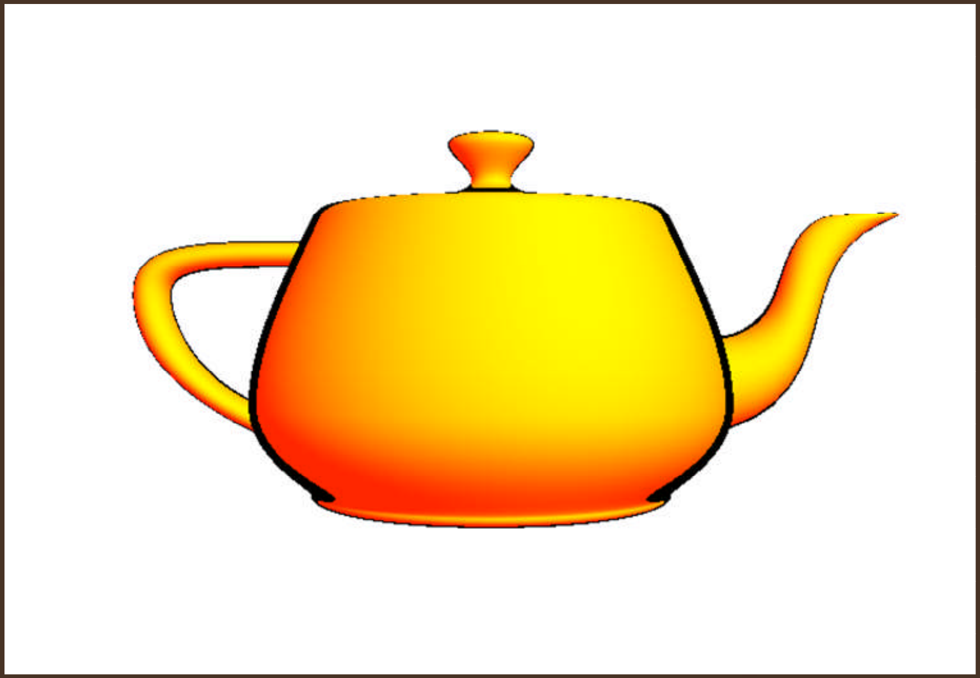
Per Fragment Cartoon Vertex Shader

```
varying vec3 N;
varying vec3 L;
varying vec3 E;

void main()
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

   vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;
   vec4 eyeLightPos = gl_LightSource[0].position;

   N = normalize(gl_NormalMatrix * gl_Normal);
   L = normalize(eyeLightPos.xyz - eyePosition.xyz);
   E = -normalize(eyePosition.xyz);
}
```

Cartoon Fragment Shader

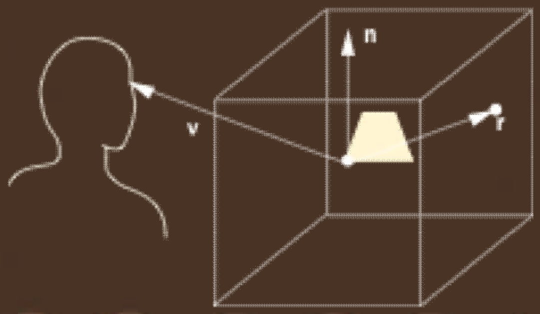
```
varying vec3 N;
varying vec3 L;
varying vec3 E;

void main()
{
   const vec4 yellow = vec4(1.0, 1.0, 0.0, 1.0);
   const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
   const vec4 black = vec4(0.0, 0.0, 0.0, 1.0);

   float Kd = max(dot(L, N), 0.0);
   gl_FragColor = mix(red, yellow, Kd);
   if(abs(dot(E,N))<0.25) gl_FragColor = black;
}
```

Cartoon Fragment Shader Result



Reflection Map

- Specify a cube map in application
- Use reflect function in vertex shader to compute view direction
- Apply texture in fragment shader



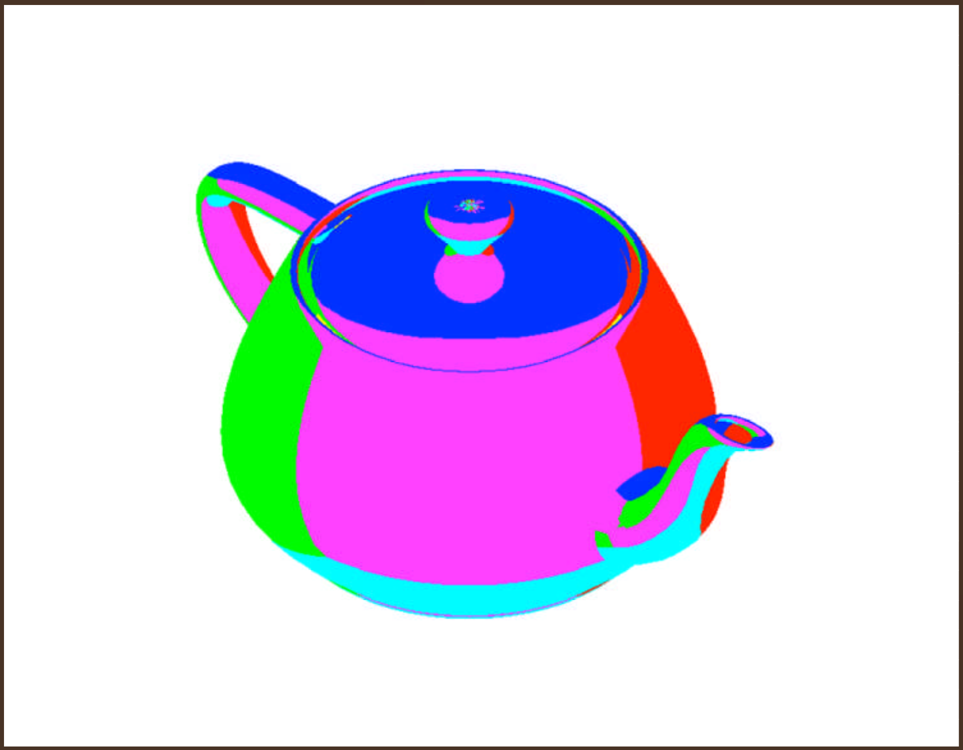
Reflection Map Vertex Shader

```
varying vec3 R;

void main()
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

   vec3 N = normalize(gl_NormalMatrix*gl_Normal);
   vec4 eyePos = gl_ModelViewMatrix*gl_Vertex;

   R = reflect(eyePos.xyz, N);
}
```

Reflection Map Fragment Shader

```
varying vec3 R;
uniform samplerCube texMap;

void main()
{
    vec4 texColor = textureCube(texMap, R);

    gl_FragColor = texColor;
}
```

Reflection mapped teapot

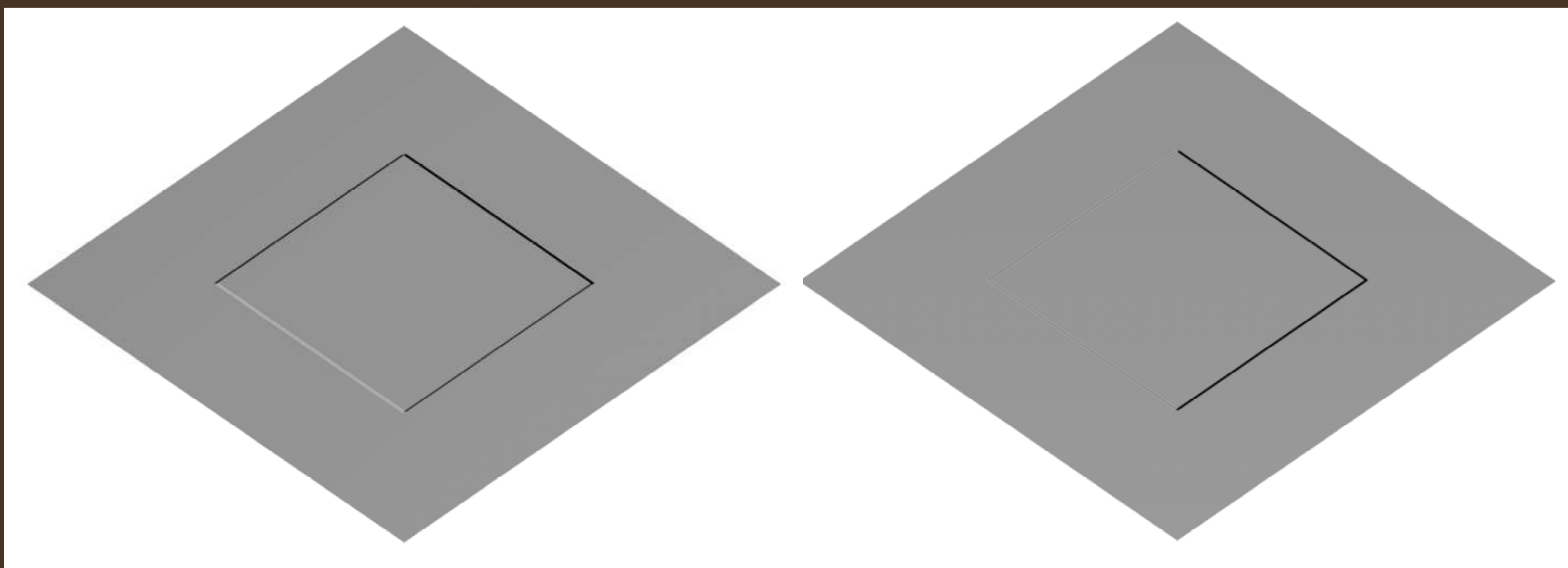


SIGGRAPH2008

Bump Mapping

- Vary normal in fragment shader so that lighting changes for each fragment
- Application: specify texture maps that describe surface variations
- Vertex Shader: calculates vertex lighting vectors and transforms to texture space
- Fragment Shader: calculate normals from texture map and shade each fragment

Bump Map Example



OpenGL Moving Forward

SIGGRAPH2008

OpenGL 3.0

- Completely backwards compatible with OpenGL 2.1
- New features to enable the latest hardware
  - most are promoted extensions
- Provides greater control of data in the graphics hardware
- Adds a mechanism for removing obsolete methods

Enhancements for Data Control

- Keeping data local to the GPU is the secret of performance
  - framebuffer objects
  - data recirculation using transform feedback
  - conditional rendering using occlusion queries
  - finer-grain control over buffer data management

Some New Core Features

- Floating-point rendering
  - texture, color, and depth buffer formats
- Enhanced texturing support
  - integer and half-float formats in shaders
- sRGB color space rendering support

Deprecation

- No features removed from OpenGL 3.0
- New context creation paradigm
  - “current” context reflects all current OpenGL features
  - “future” context indicates features that will (eventually) be deprecated out of OpenGL

Get All the Details!

OpenGL BOF
6 – 8 PM Tonight!
Wilshire Room
Wilshire Grand Hotel

Thanks for Attending!

Any Questions?

SIGGRAPH2008

Bill, Dave, Ed & Evan

On-Line Resources

- http://www.opengl.org & http://www.khronos.org
  - start here; up to date specification and lots of sample code
  - online “man pages” for all OpenGL functions
- http://www.mesa3d.org/
  - Brian Paul’s Mesa 3D
- http://www.cs.utah.edu/~narobins/opengl.html
  - very special thanks to Nate Robins for the OpenGL Tutors
  - source code for tutors available here!

Books

- *OpenGL Programming Guide*, 6th Edition
- *The OpenGL Shading Language*, 2nd Edition
- *Interactive Computer Graphics: A top-down approach with OpenGL*, 5th Edition
- *OpenGL Programming for the X Window System*
- *OpenGL: A Primer*, 3rd Edition
- *OpenGL Distilled*
- *OpenGL Programming on Mac OS® X*

SIGGRAPH2008