An Introduction to WebGL Programming

Ed Angel
University of New Mexico

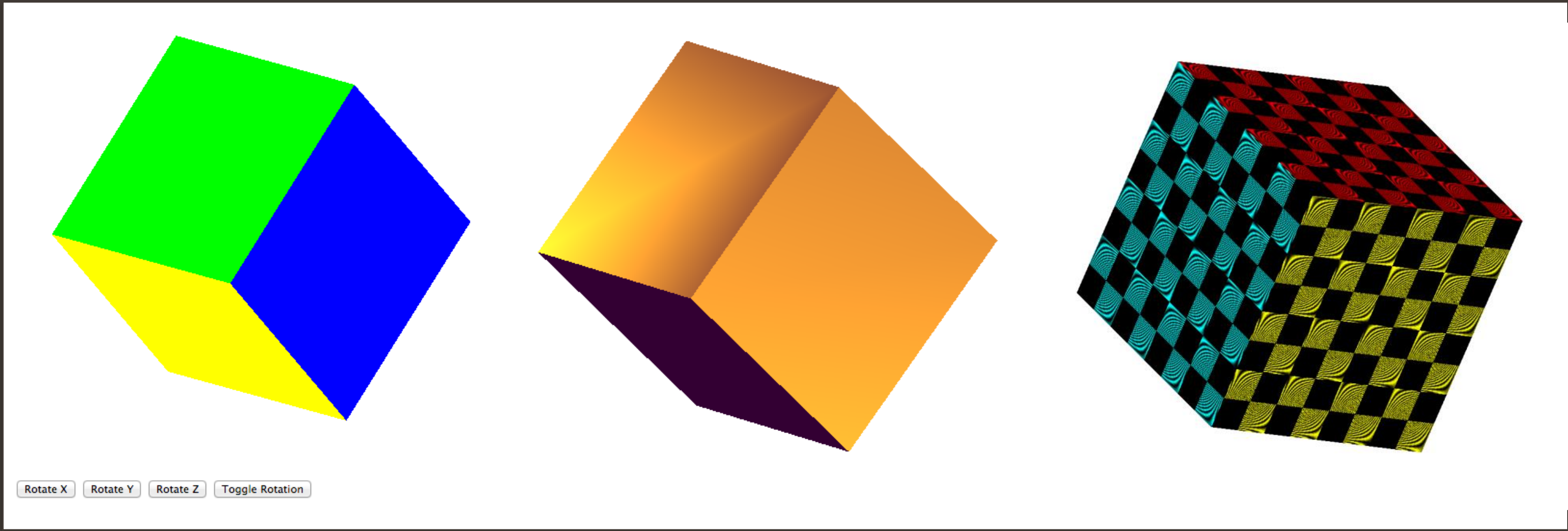
Dave Shreiner
ARM, Inc.

(+ small changes w.a. 2017)

Agenda

- Evolution of the OpenGL Pipeline
- Prototype Applications   in WebGL
- OpenGL Shading Language (GLSL)
- Vertex Shaders
- Fragment Shaders
- Examples

Examples



rotating cube with buttons

cube with lighting

texture mapped cube

What Is OpenGL?

- OpenGL is a computer graphics rendering *application programming interface*, or API (for short)
  - With it, you can generate high-quality color images by rendering with geometric and image primitives
  - It forms the basis of many interactive applications that include 3D graphics
  - By using OpenGL, the graphics part of your application can be
    - operating system independent
    - window system independent

What Is WebGL?

- WebGL: JavaScript implementation of OpenGL ES 2.0
  - runs in all recent browsers (Chrome, Firefox, IE, Safari)
  - application can be located on a remote server
  - rendering is done within browser using local hardware
  - uses HTML5 canvas element
  - integrates with standard Web packages and apps
    - CSS
    - jQuery

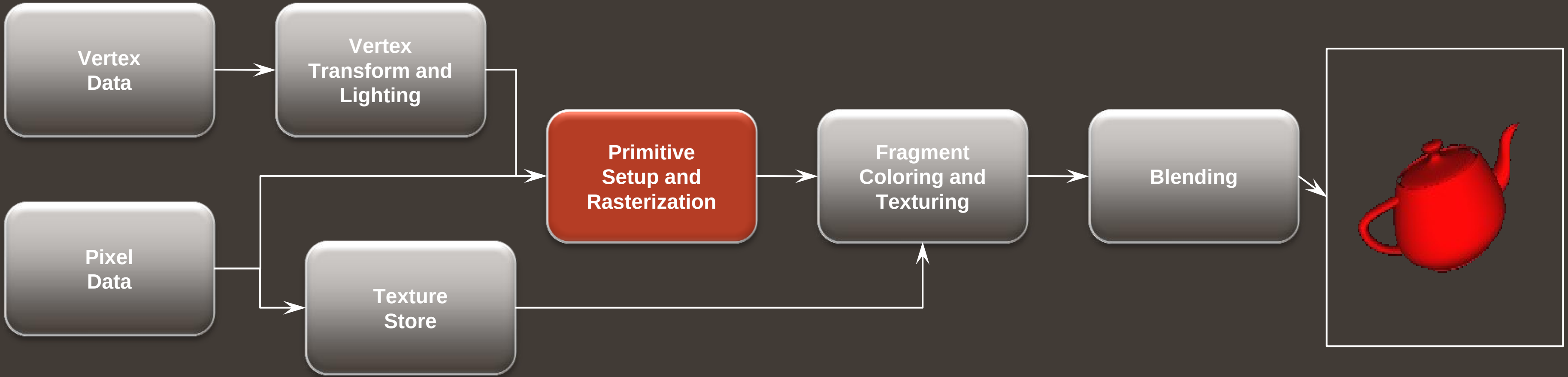
What do you need to know?

- Web environment and execution
- Modern OpenGL basics
  - pipeline architecture
  - shader based OpenGL
  - OpenGL Shading Language (GLSL)
- JavaScript

Evolution of the OpenGL Pipeline

In the Beginning …

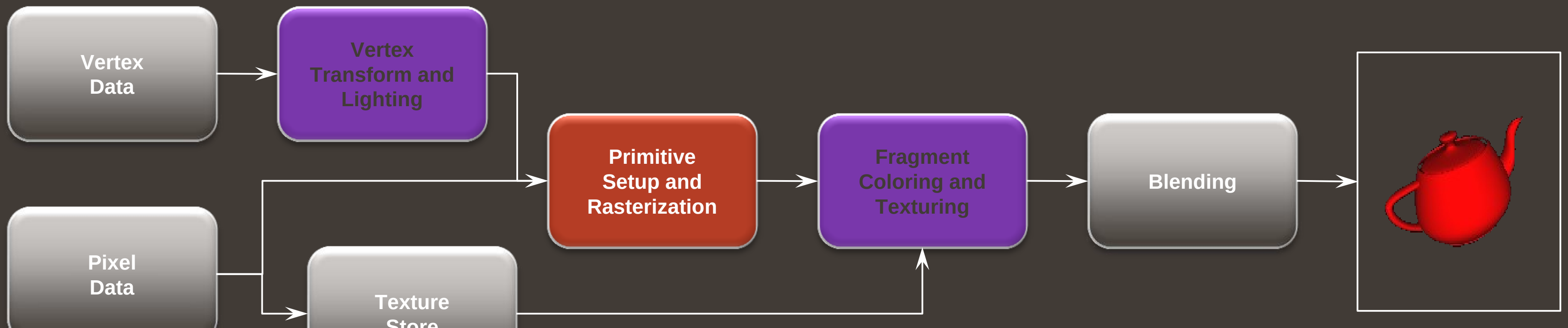
- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation



- The pipeline evolved
  - but remained based on fixed-function operation through OpenGL versions 1.1 through 2.0 (Sept. 2004)

Beginnings of The Programmable Pipeline

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available



An Evolutionary Change

- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

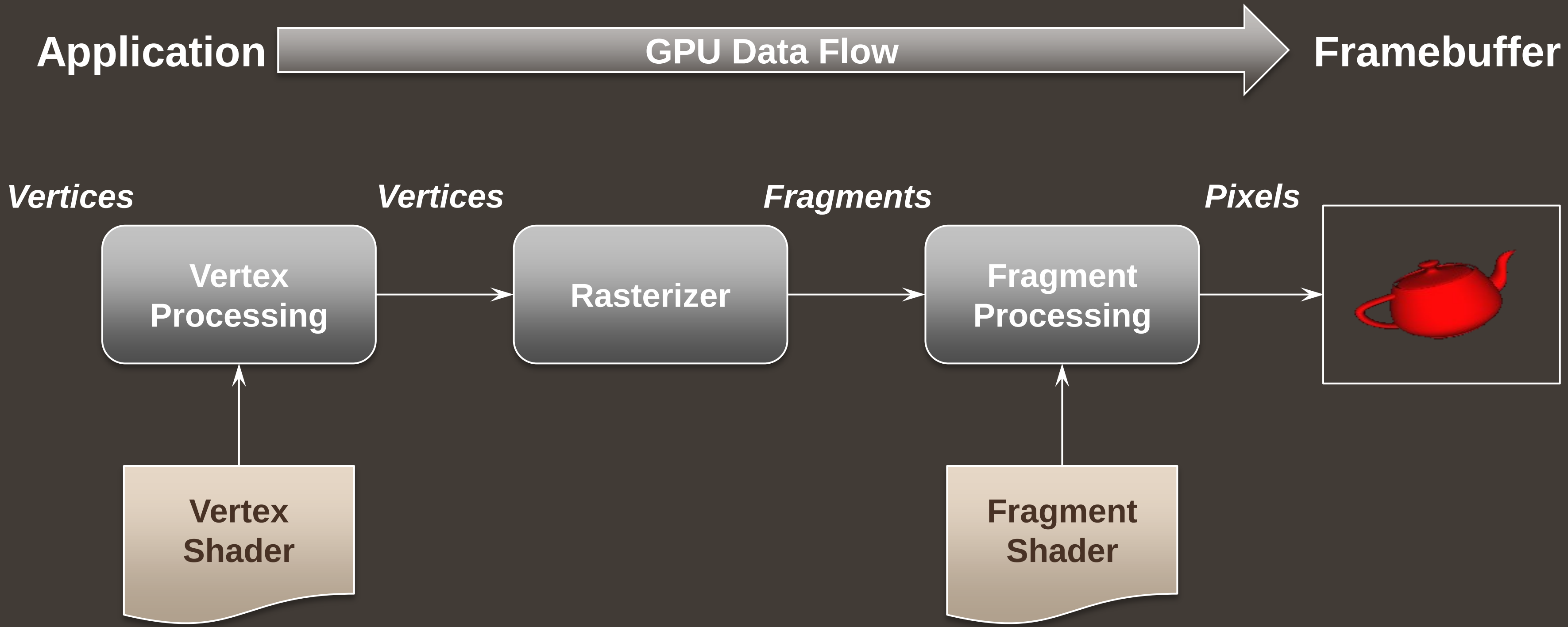
Context Type	**Description**
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

OpenGL ES and WebGL

- OpenGL ES 2.0
  - Designed for embedded and hand-held devices such as cell phones
  - Based on OpenGL 3.1
  - Shader based
- WebGL
  - JavaScript implementation of ES 2.0
  - Runs on most recent browsers

WebGL Application Development

Simplified Pipeline Model



WebGL Programming in a Nutshell

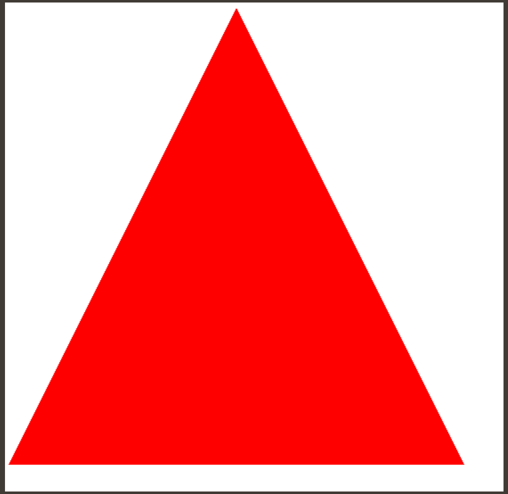
- All WebGL programs must do the following:
  - Set up canvas to render onto
  - Generate data in application
  - Create shader programs
  - Create buffer objects and load data into them
  - “Connect” data locations with shader variables
  - Render

Application Framework

- WebGL applications need a place to render into
  - HTML5 Canvas element
- We can put all code into a single HTML file
- We prefer to put setup in an HTML file and application in a separate JavaScript file
  - HTML file includes shaders
  - HTML file reads in utilities and application

A Really Simple Example

- Generate one red triangle
- Has all the elements of a more complex application
    - vertex shader
    - fragment shader
    - HTML canvas



- www.cs.unm.edu/~angel/WebGL

triangle.html

```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
</script>
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
void main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>
```

triangle.html

```
<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="triangle.js"></script>
</head>
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

triangle.js

```
var gl;
var points;

window.onload = function init()
{
    var canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" );
 }

var vertices = new Float32Array([-1, -1, 0, 1, 1, -1]);

 //  Configure WebGL

gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
```

triangle.js

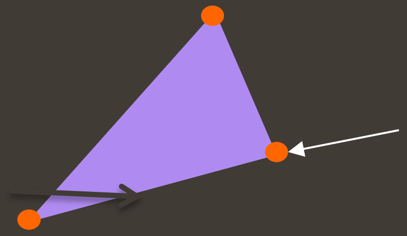
```
// Load shaders and initialize attribute buffers

var program = initShaders( gl, "vertex-shader", "fragment-shader" );

 gl.useProgram( program );

// Load the data into the GPU

 var bufferId = gl.createBuffer();
 gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
 gl.bufferData( gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW );
```

triangle.js

```
// Associate out shader variables with our data buffer

    var vPosition = gl.getAttribLocation( program, "vPosition" );
    gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
    gl.enableVertexAttribArray( vPosition );
   render();
};

function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLES, 0, 3 );
}
```

Representing Geometric Objects

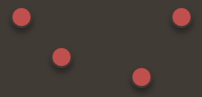
- Geometric objects are represented using *vertices*
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space
- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in vertex buffer objects (VBOs)



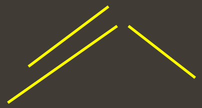
$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

OpenGL Geometric Primitives

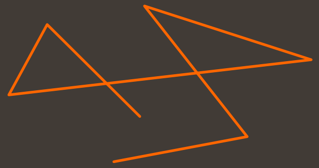
All primitives are specified by vertices



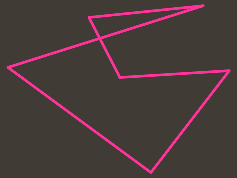
GL_POINTS
gl.POINTS



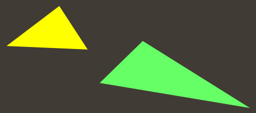
GL_LINES
gl.LINES



GL_LINE_STRIP
gl.LINE_STRIP



GL_LINE_LOOP
gl.LINE_LOOP



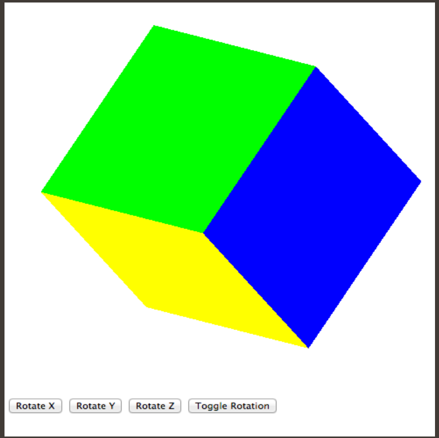
GL_TRIANGLES
gl.TRIANGLES

GL_TRIANGLE_STRIP
gl.TRIANGLE_STRIP

GL_TRIANGLE_FAN
gl.TRIANGLE_FAN

Our Second Program

- Render a cube with a different color for each face
- Our example demonstrates:
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices
  - initializing vertex data
  - organizing data for rendering
  - interactivity
  - animation



Initializing the Cube's Data

- We'll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
var numVertices = 36;
```

- To simplify communicating with GLSL, we'll use a package MV.js which contains a `vec3` object similar to GLSL's `vec3` type

Initializing the Cube's Data (cont'd)

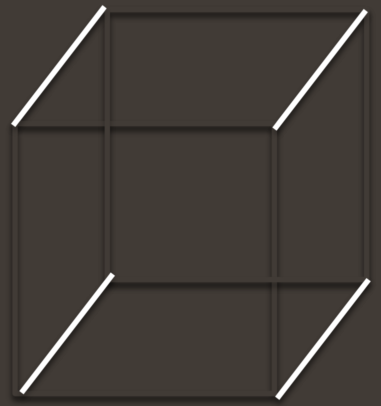
- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
var points = [];
var colors = [];
```

Cube Data

- Vertices of a unit cube centered at origin
  - sides aligned with axes

```
var vertices = [
     vec4( -0.5, -0.5,  0.5, 1.0 ),
     vec4( -0.5,  0.5,  0.5, 1.0 ),
     vec4(  0.5,  0.5,  0.5, 1.0 ),
     vec4(  0.5, -0.5,  0.5, 1.0 ),
     vec4( -0.5, -0.5, -0.5, 1.0 ),
     vec4( -0.5,  0.5, -0.5, 1.0 ),
     vec4(  0.5,  0.5, -0.5, 1.0 ),
     vec4(  0.5, -0.5, -0.5, 1.0 )
  ];
```



Cube Data (cont'd)

- We'll also set up an array of RGBA colors
- We can use vec3 or vec4 or just JS array

```
var vertexColors = [
    [ 0.0, 0.0, 0.0, 1.0 ],  // black
    [ 1.0, 0.0, 0.0, 1.0 ],  // red
    [ 1.0, 1.0, 0.0, 1.0 ],  // yellow
    [ 0.0, 1.0, 0.0, 1.0 ],  // green
    [ 0.0, 0.0, 1.0, 1.0 ],  // blue
    [ 1.0, 0.0, 1.0, 1.0 ],  // magenta
    [ 0.0, 1.0, 1.0, 1.0 ],  // cyan
    [ 1.0, 1.0, 1.0, 1.0 ]   // white
];
```

Arrays in JS

- A JS array is an object with attributes and methods such as length, push() and pop()
  - fundamentally different from C-style array
  - cannot send directly to WebGL functions
  - use flatten() function to extract data from JS array

```
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors),
   gl.STATIC_DRAW );
```

Generating a Cube Face from Vertices

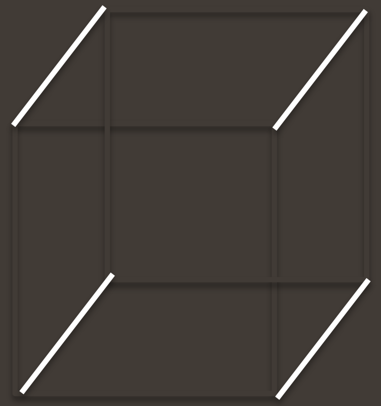
- To simplify generating the geometry, we use a convenience function `quad()`
  - create two triangles for each face and assigns colors to the vertices

```
function quad(a, b, c, d) {
var indices = [ a, b, c, a, c, d ];
    for ( var i = 0; i < indices.length; ++i ) {
        points.push( vertices[indices[i]] );

        // for vertex colors use
        //colors.push( vertexColors[indices[i]] );

        // for solid colored faces use
        colors.push(vertexColors[a]);
}
```

Generating the Cube from Faces

- Generate 12 triangles for the cube
  - 36 vertices with 36 colors

```
function colorCube() {
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```



Storing Vertex Attributes

- Vertex data must be stored in a Vertex Buffer Object (VBO)
- To set up a VBO we must
  - create an empty by calling gl.createBuffer(); ( )
  - bind a specific VBO for initialization by calling

```
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
```

  - load data into VBO using (for our points)

```
gl.bufferData( gl.ARRAY_BUFFER, flatten(points),
      gl.STATIC_DRAW );
```

Vertex Array Code

Associate shader variables with vertex arrays

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );

var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );
```

Drawing Geometric Primitives

- For contiguous groups of vertices, we can use the simple render function

```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

- gl.drawArrays initiates vertex shader
- requestAnimationFrame needed for redrawing if anything is changing
- Note we must clear both the frame buffer and the depth buffer
- Depth buffer used for hidden surface removal

  gl.enable(gl.GL_DEPTH) in init()

Vertex Shaders

- A shader that's executed for each vertex
  - Each instantiation can generate one vertex
  - Outputs are passed on to rasterizer where they are interpolated and available to fragment shaders
  - Position output in clip coordinates
- There are lots of effects we can do in vertex shaders
  - Changing coordinate systems
  - Moving vertices
  - Per vertex lighting: height fields

Fragment Shaders

- A shader that's executed for each "potential" pixel
  - fragments still need to pass several tests before making it to the framebuffer
- There are lots of effects we can do in fragment shaders
  - Per-fragment lighting
  - Texture and bump Mapping
  - Environment (Reflection) Maps

GLSL

- Open<u>GL</u> <u>S</u>hading <u>L</u>anguage
- C like language with some C++ features
- 2-4 dimensional matrix and vector types
- Both vertex and fragment shaders are written in GLSL
- Each shader has a main()

GLSL Data Types

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
  `ivec2, ivec3, ivec4`
  `bvec2, bvec3, bvec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, samplerCube`
- C++ Style Constructors

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

Operators

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Components and Swizzling

- Access vector components using either:
  - `[ ]` (C-style array indexing)
  - `xyzw`, `rgba` or `strq` (named components)

- For example:

```
vec3 v;
v[1], v.y, v.g, v.t
```
 - all refer to the same element

- Component swizzling:
-

```
vec3 a, b;
a.xy = b.yx;
```

Qualifiers

- `attribute`
- `varying`
  - copy vertex attributes and other variables from vertex shaders to fragment shaders
  - values are interpolated by rasterizer

```
varying  vec2 texCoord;
varying vec4 color;
```

- `uniform`
  - shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

Functions

- Built in
  - Arithmetic: `sqrt`, `power`, `abs`
  - Trigonometric: `sin`, `asin`
  - Graphical: `length`, `reflect`
- User defined

Built-in Variables

- `gl_Position`
  - (required) output position from vertex shader
- gl_FragColor
  - (required) output color from fragment shader
- `gl_FragCoord`
  - input fragment position
- `gl_FragDepth`
  - input depth value in fragment shader

Simple Vertex Shader for Cube Example

```
attribute  vec4 vPosition;
attribute  vec4 vColor;

varying vec4 fColor;

void main()
{
    fColor = vColor;
    gl_Position = vPosition;
}
```

Simple Fragment Shader for Cube Example

```
precision mediump float;

varying vec4 fColor;
void main()
{
    gl_FragColor = fColor;
}
```

Getting Your Shaders into WebGL

- Shaders need to be compiled and linked to form an executable shader program
- WebGL provides the compiler and linker
- A WebGL program must contain vertex and fragment shaders

Create Program	`gl.createProgram()`
Create Shader	`gl.createShader()`
Load Shader Source	`gl.shaderSource()`
Compile Shader	`gl.compileShader()`
Attach Shader to Program	`gl.attachShader()`
Link Program	`gl.linkProgram()`
Use Program	`gl.useProgram()`

A Simpler Way

- We’ve created a function for this course to make it easier to load your shaders
  - available at course website

```
initShaders(vFile, fFile );
```

- `initShaders` takes two filenames
  - `vFile` path to the vertex shader file
  - `fFile` for the fragment shader file
- Fails if shaders don’t compile, or program doesn’t link

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' names

```
loc = gl.getAttribLocation( program,“name” );

loc = gl.getUniformLocation( program,“name” );
```

Initializing Uniform Variable Values

Uniform Variables

```
gl.uniform4f( index, x, y, z, w );

var transpose = gl.GL_TRUE;



gl.uniformMatrix4fv( index, 3, transpose, mat );
```

Application Organization

- HTML file:
  - contains shaders
  - brings in utilities and application JS file
  - describes page elements: buttons, menus
  - contains canvas element
- JS file
  - init()
    - sets up VBOs
    - contains listeners for interaction
    - sets up required transformation matrices
    - reads, compiles and links shaders
  - render()

Buffering, Animation and Interaction

Double Buffering

- The processes of rendering into a frame buffer and displaying the contents of the frame buffer are independent

- To prevent displaying a partially rendered frame buffer, the browser uses double buffering
  - rendering is into the back buffer
  - display is from the front buffer
  - when rendering is complete, buffers are swapped

- However, we need more control of the display from the application

Animation

- Suppose we want to change something and render again with new values
  - We can send new values to the shaders using uniform qualified variables
- Ask application to rerender with requestAnimFrame()
  - Render function will execute next refresh cycle
  - Change render function to call itself
- We can also use the timer function setInterval(render, milliseconds) to control speed

Animation Example

Make cube bigger and smaller sinusoidally in time

```
timeLoc = gl.getUniformLocation(program, "time"); // in init()

function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.uniform3fv(thetaLoc, theta);
    time+=dt;
    gl.uniform1f(timeLoc, time);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
// in vertex shader

uniform float time;
gl_Position = (1.0+0.5*sin(time))*vPosition;
gl_Position.w = 1.0;
```

Vertex Shader Applications

- A vertex shader is initiated by each vertex output by `gl.drawArrays()`
- A vertex shader must output a position in clip coordinates to the rasterizer
- Basic uses of vertex shaders
  - Transformations
  - Lighting
  - Moving vertex positions
    - animation
    - morphing

Event Driven Input

- Browser execute code sequential and then wait for an event to occur
- Events can be of many types
  - mouse and keyboard
  - menus and buttons
  - window events
- Program responds to events through functions called listeners or callbacks

Adding Buttons

- In HTML file

```
<button id= "xButton">Rotate X</button>
<button id= "yButton">Rotate Y</button>
<button id= "zButton">Rotate Z</button>
<button id = "ButtonT">Toggle Rotation</button>
```

Event Listeners

In init()

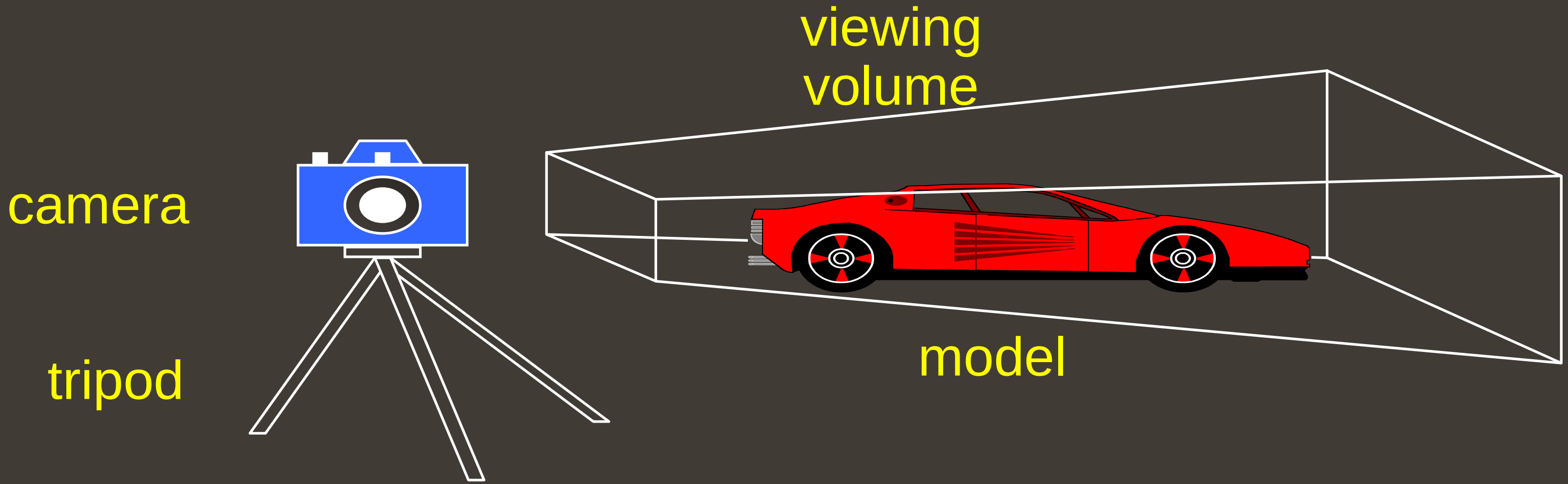
```
document.getElementById( "xButton" ).onclick =
    function () { axis = xAxis; };  document.getElementById(
"yButton" ).onclick =
    function () { axis = yAxis; };    document.getElementById(
"zButton" ).onclick =
    function () { axis = zAxis;};
document.getElementById("ButtonT").onclick =
    function(){ flag = !flag; };

 render();
```

Render Function

```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT);

    if(flag) theta[axis] += 2.0;

    gl.uniform3fv(thetaLoc, theta);

    gl.drawArrays( gl.TRIANGLES, 0, numVertices );

    requestAnimFrame( render );
}
```

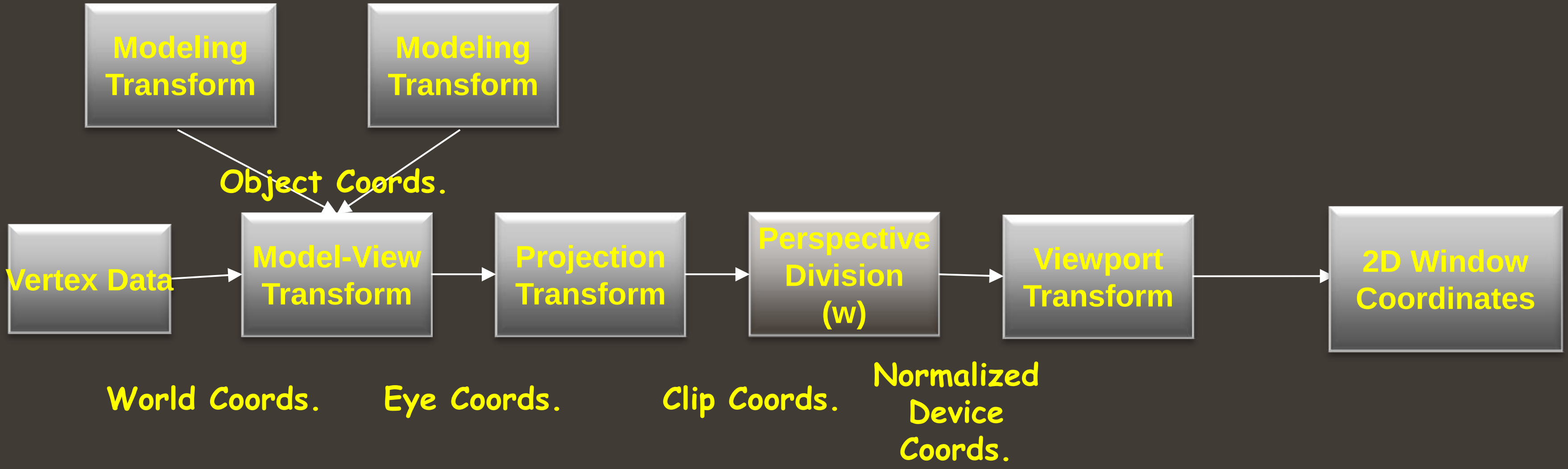
Synthetic Camera Model

- 3D is just like taking a photograph (lots of photographs!)



Transformations

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



Camera Analogy and Transformations

- Projection transformations
  - adjust the lens of the camera
- Viewing transformations
  - tripod–define position and orientation of the viewing volume in the world
- Modeling transformations
  - moving the model
- Viewport transformations
  - enlarge or reduce the physical photograph

3D Transformations

- A vertex is transformed by 4×4 matrices
  - all affine operations are matrix multiplications
- All matrices are stored column-major in OpenGL
  - this is opposite of what "C" programmers expect
- Matrices are always post-multiplied
  - product of matrix and vector is $\mathbf{M}\check{v}$

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

Specifying What You Can See

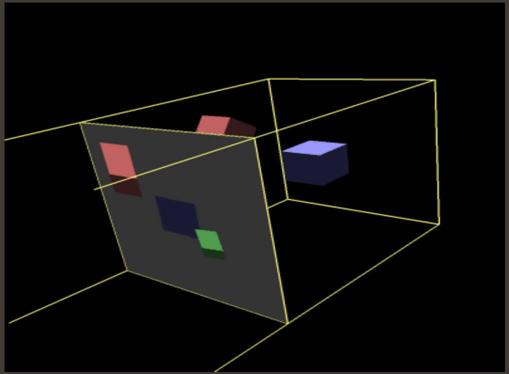
- Set up a viewing frustum to specify how much of the world we can see
- Done in two steps
  - specify the size of the frustum (projection transform)
  - specify its location in space (model-view transform)
- Anything outside of the viewing frustum is clipped
  - primitive is either modified or discarded (if entirely outside frustum)

Specifying What You Can See (cont'd)

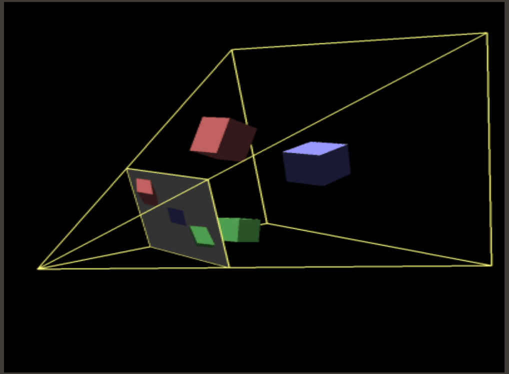
- OpenGL projection model uses eye coordinates
  - the "eye" is located at the origin
  - looking down the -z axis
- Projection matrices use a six-plane model:
  - near (image) plane and far (infinite) plane
    - both are distances from the eye (positive values)
  - enclosing planes
    - top & bottom,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

Specifying What You Can See (cont'd)

Orhographic View



Perspective View



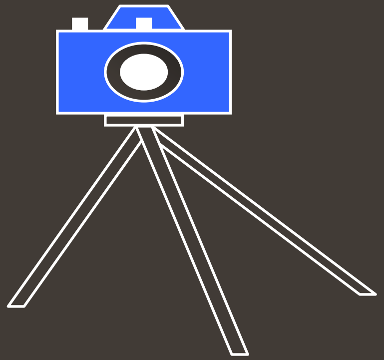
$$O = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & \frac{2}{near - far} & \frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P = \begin{bmatrix} \frac{2 \cdot near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2 \cdot near}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & \frac{2 \cdot far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

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
  - lookAt() is in MV.js and is functionally equivalent to deprecated OpenGL function

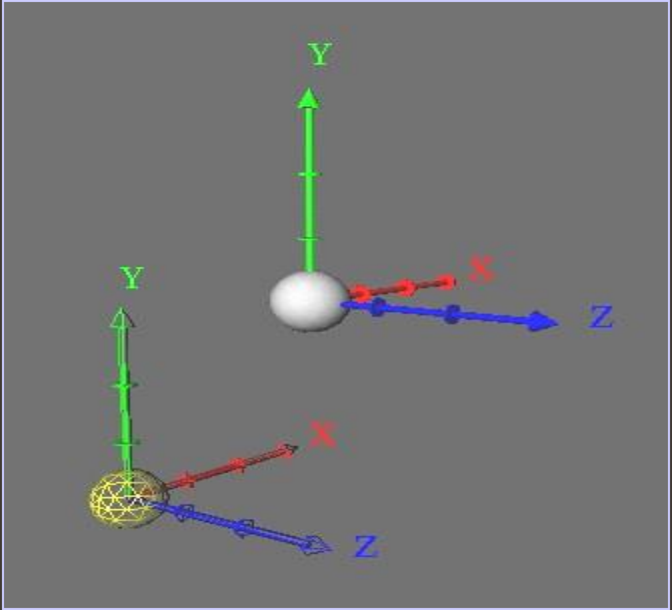
tripod



Translation

- Move the origin to a new location

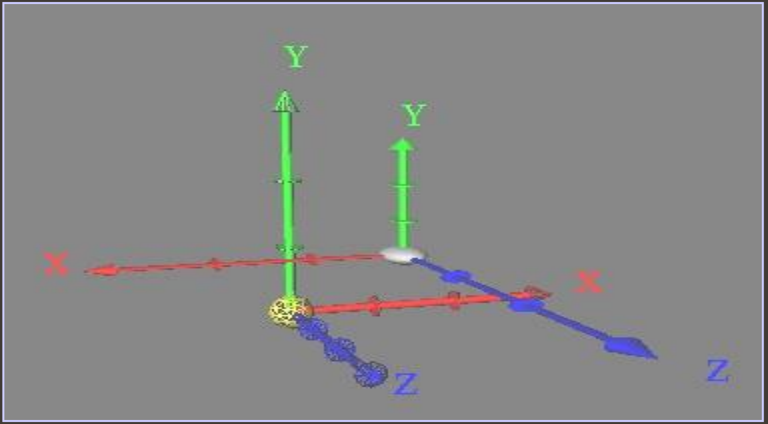
$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Scale

- Stretch, mirror or decimate a coordinate direction

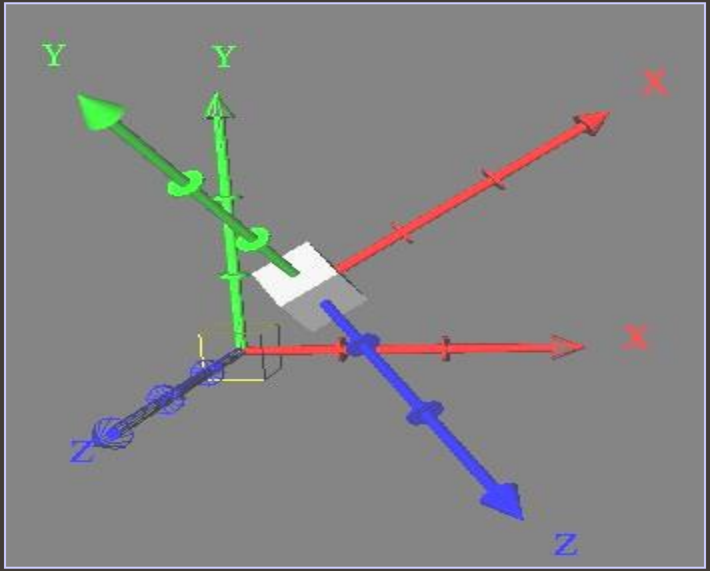
$$\mathbf{S} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



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

Vertex Shader for Rotation of Cube

```
attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```

Vertex Shader for Rotation of Cube (cont'd)

```
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```

Vertex Shader for Rotation of Cube (cont'd)

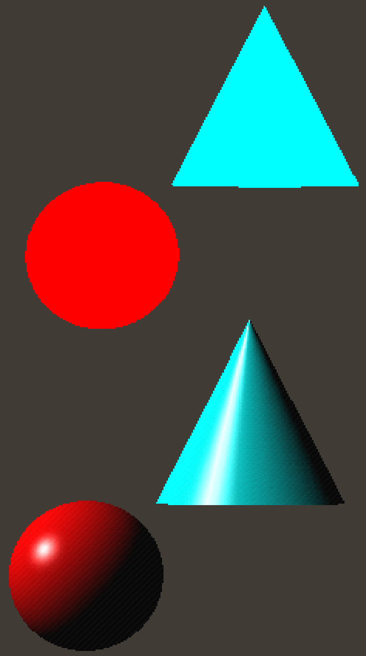
```
    mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                    s.z,  c.z, 0.0, 0.0,
                    0.0,  0.0, 1.0, 0.0,
                    0.0,  0.0, 0.0, 1.0 );

    color = vColor;
    gl_Position = rz * ry * rx * vPosition;
}
```

Sending Angles from Application

```
// in init()

var theta = [ 0, 0, 0 ];
var axis = 0;
thetaLoc = gl.getUniformLocation(program, "theta");

// set axis and flag via buttons and event listeners

// in render()

 if(flag) theta[axis] += 2.0;
 gl.uniform3fv(thetaLoc, theta);
```

Lighting Principles

- Lighting simulates how objects reflect light
  - material composition of object
  - light’s color and position
  - global lighting parameters
- Usually implemented in
  - vertex shader for faster speed
  - fragment shader for nicer shading

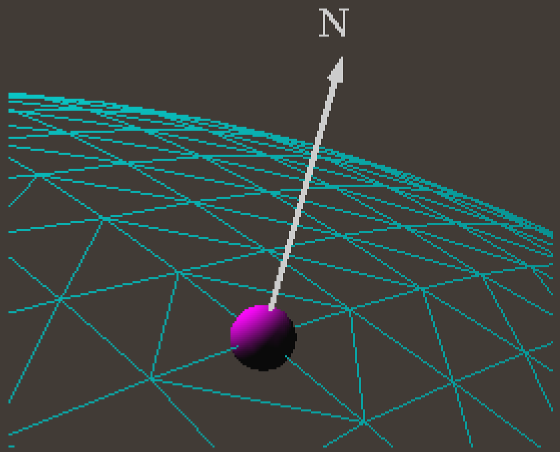


Modified Phong Model

- Computes a color for each vertex using
  - Surface normals
  - Diffuse and specular reflections
  - Viewer’s position and viewing direction
  - Ambient light
  - Emission
- Vertex colors are interpolated across polygons by the rasterizer
  - *Phong shading* does the same computation per pixel, interpolating the normal across the polygon
    - more accurate results

Surface Normals

- Normals define how a surface reflects light
  - Application usually provides normals as a vertex atttribute
  - Current normal is used to compute vertex’s color
  - Use unit normals for proper lighting
    - scaling affects a normal’s length



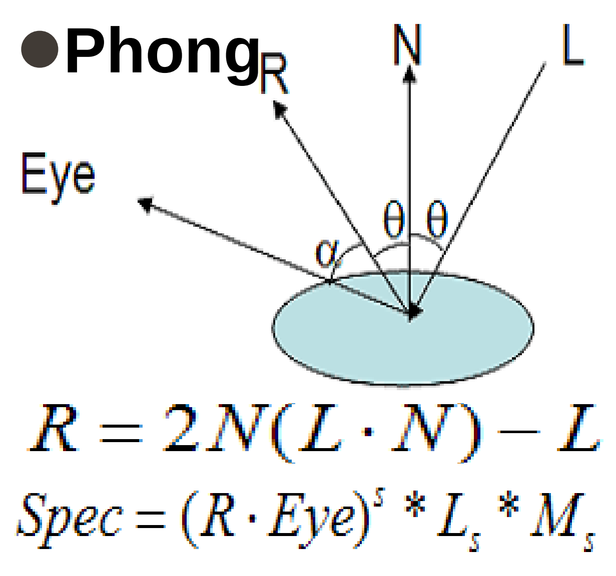
Material Properties

- Define the surface properties of a primitive

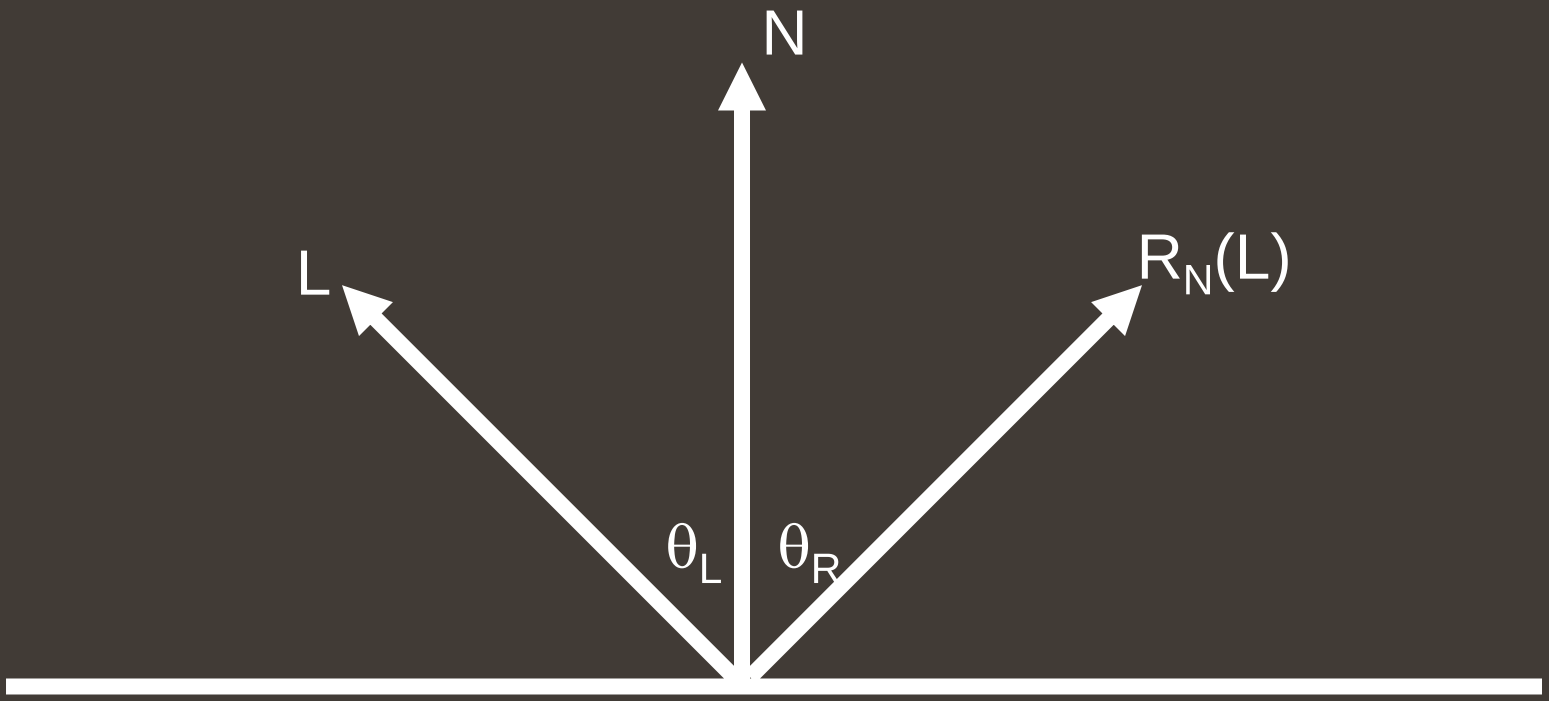
Property	Description
Diffuse	Base object color
Specular	Highlight color
Ambient	Low-light color
Emission	Glow color
Shininess	Surface smoothness

  – you can have separate materials for front and back

Phong Model

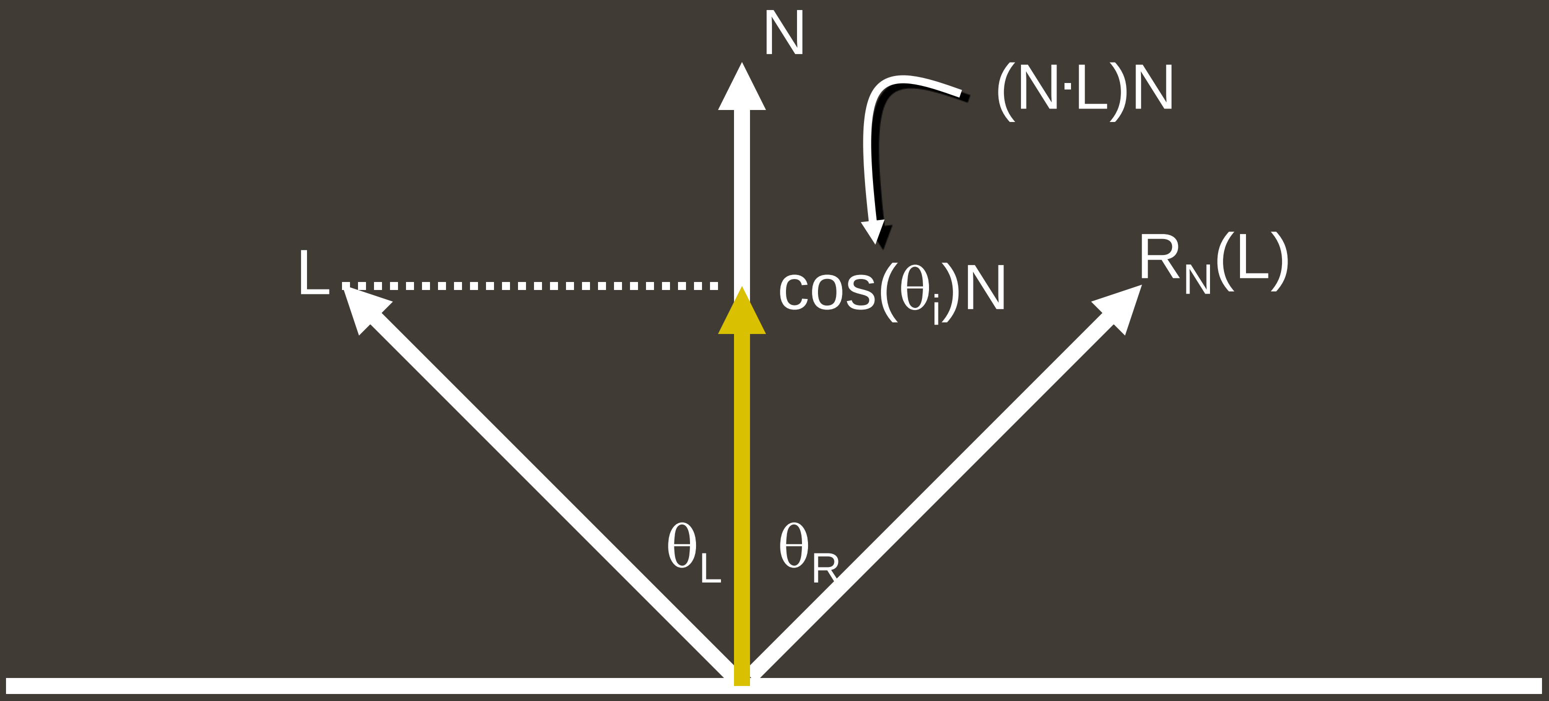


Phong formula explained



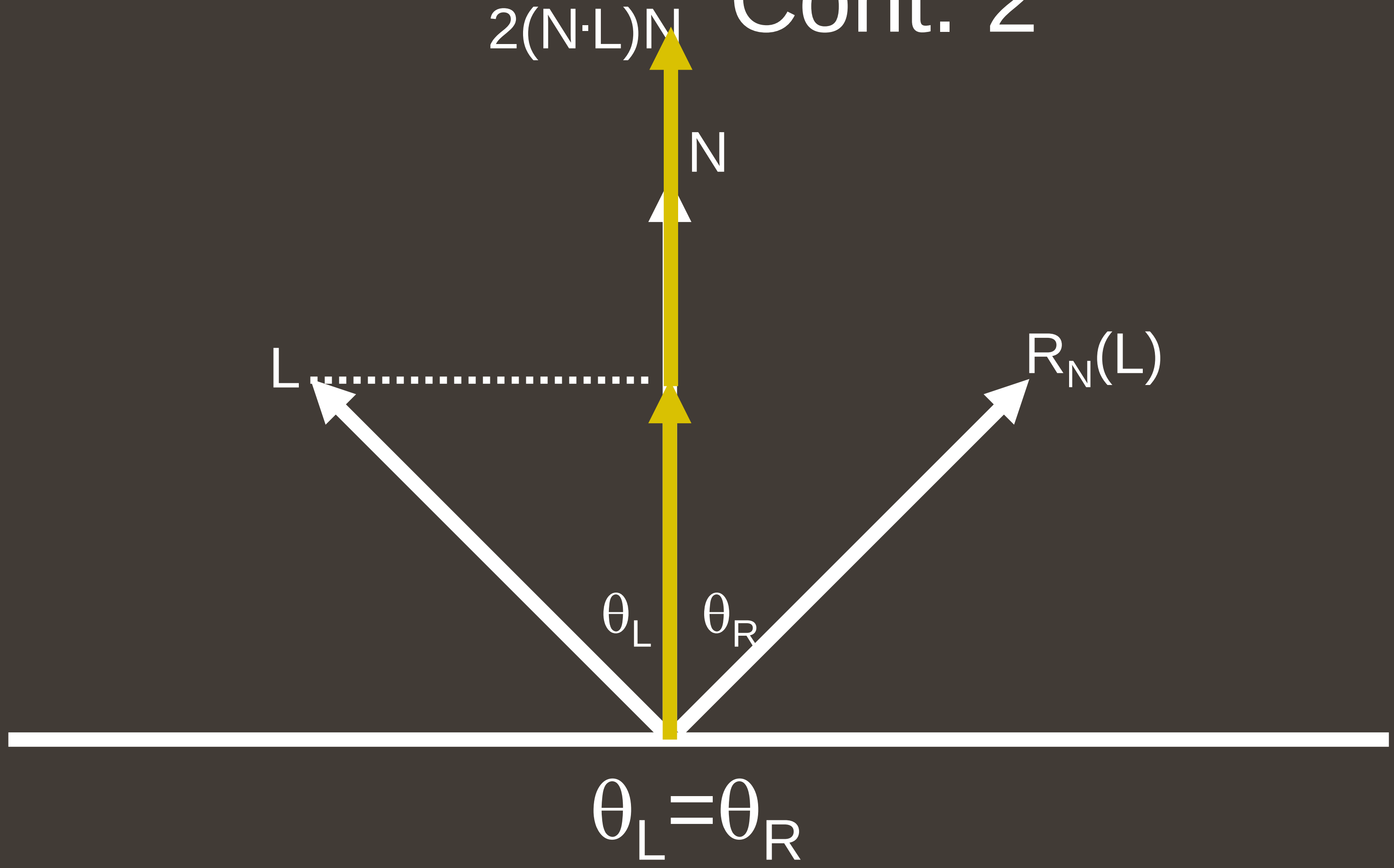
$$\theta_L = \theta_R$$

Cont. 1

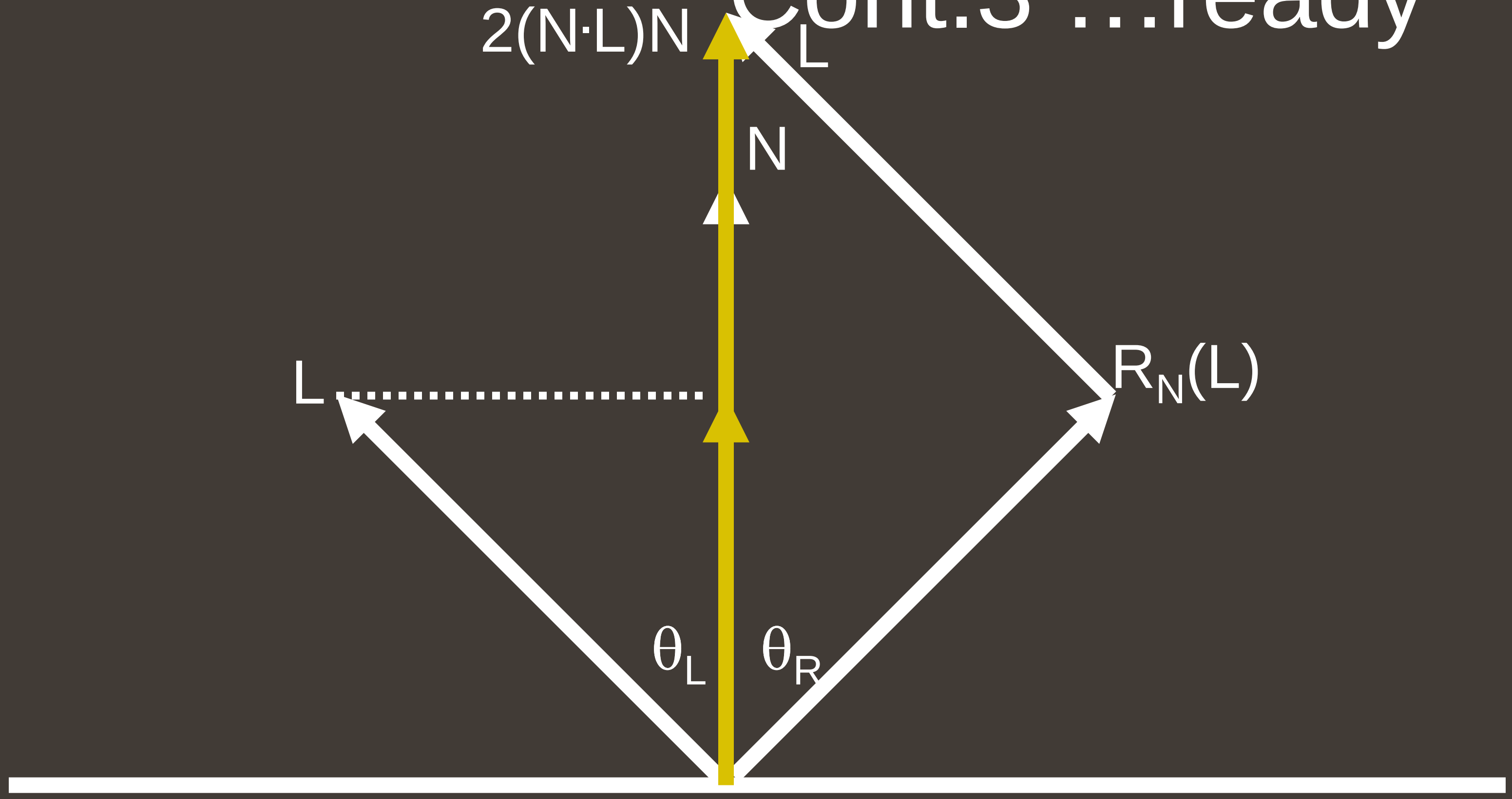


$$\theta_L = \theta_R$$

Cont. 2

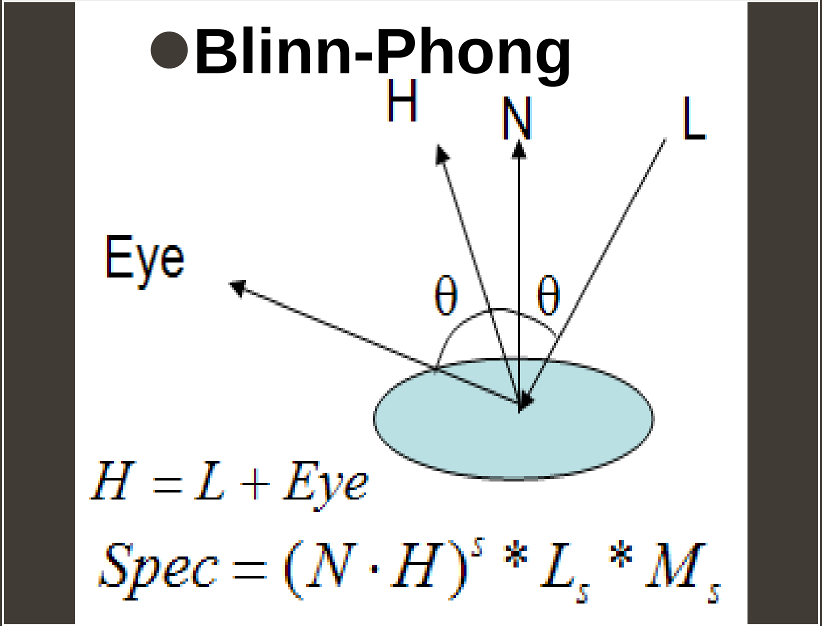


Cont.3 ...ready



$$\theta_L = \theta_R$$

Blinn-Phong formula



Blinn vs Phong

Adding Lighting to Cube

```
// vertex shader with Blinn-Phong formula

in vec4 vPosition;
in vec3 vNormal;
out vec4 color;

uniform vec4  AmbientProduct, DiffuseProduct,
      SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

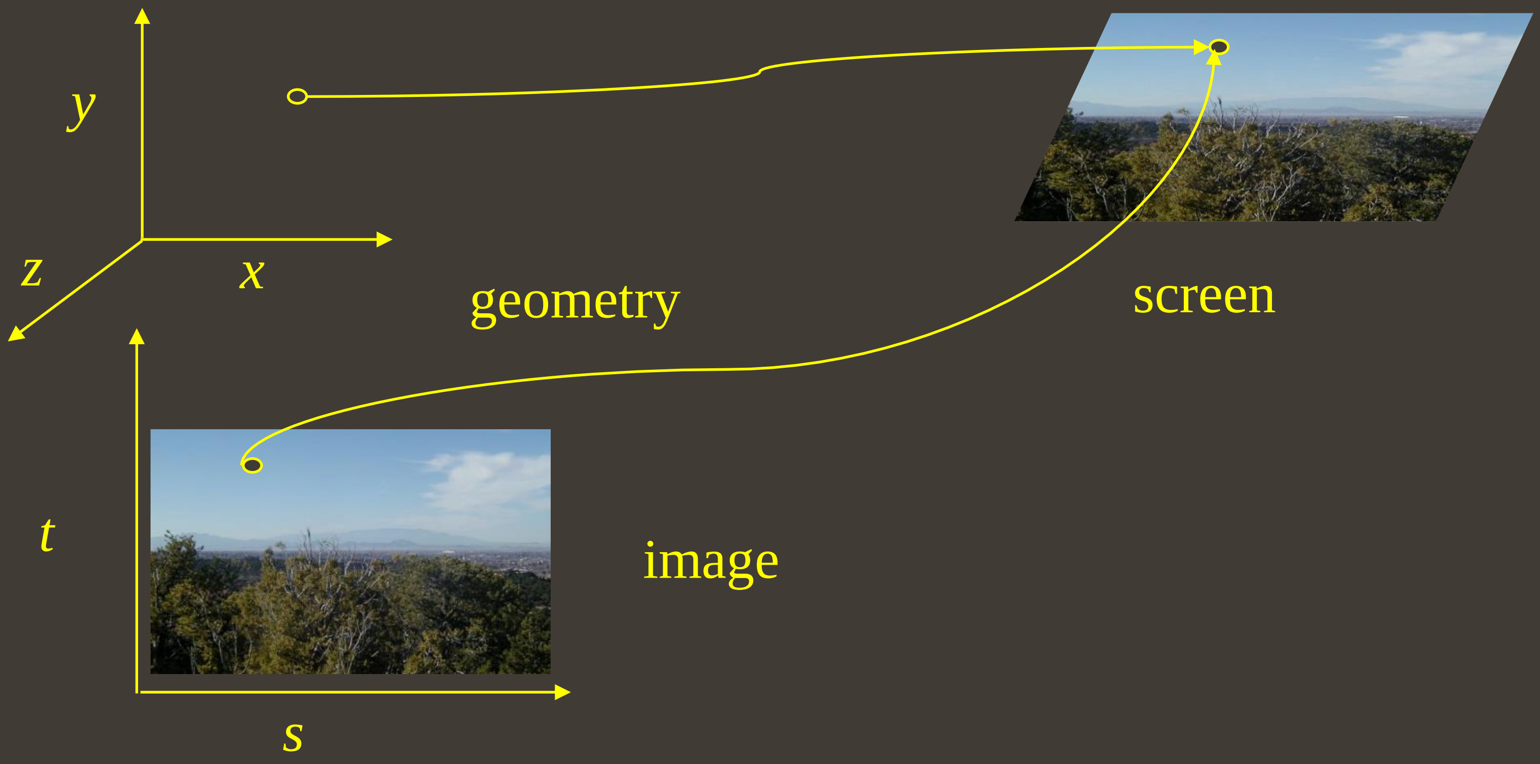
Adding Lighting to Cube (cont'd)

```
void main()
{
   // Transform vertex  position into eye coordinates
   vec3 pos = vec3(ModelView * vPosition);

   vec3 L = normalize(LightPosition.xyz - pos);
   vec3 E = normalize(-pos);
   vec3 H = normalize(L + E);

   // Transform vertex normal into eye coordinates
   vec3 N = normalize(vec3(ModelView * vNormal));
```

Adding Lighting to Cube (cont’d)

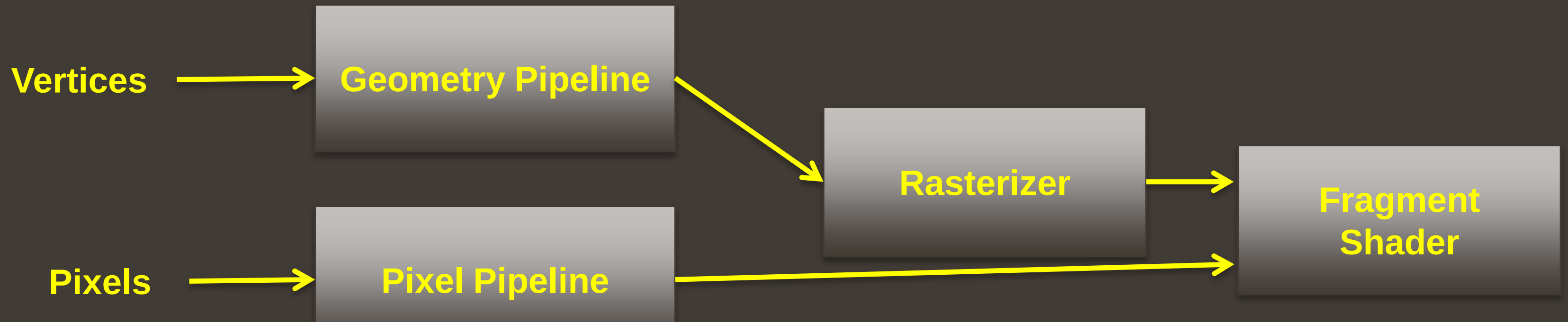
```
// Compute terms in the illumination equation
   vec4 ambient = AmbientProduct;

   float Kd = max( dot(L, N), 0.0 );
   vec4  diffuse = Kd*DiffuseProduct;

   float Ks = pow( max(dot(N, H), 0.0), Shininess );
   vec4  specular = Ks * SpecularProduct;
   if( dot(L, N) < 0.0 )
       specular = vec4(0.0, 0.0, 0.0, 1.0)

   gl_Position = Projection * ModelView * vPosition;

   color = ambient + diffuse + specular;
   color.a = 1.0;
}
```

Texture Mapping



Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer
  - "complex" textures do not affect geometric complexity



Applying Textures

- Three basic steps to applying a texture
  1. specify the texture
     - read or generate image
     - assign to texture
     - enable texturing
  2. assign texture coordinates to vertices
  3. specify texture parameters
     - wrapping, filtering

Texture Objects

- Have WebGL store your images
  - one image per texture object
- Create an empty texture object

```
var texture = gl.createTexture();
```

- Bind textures before using

```
gl.bindTexture( gl.TEXTURE_2D, texture );
```

Specifying a Texture Image

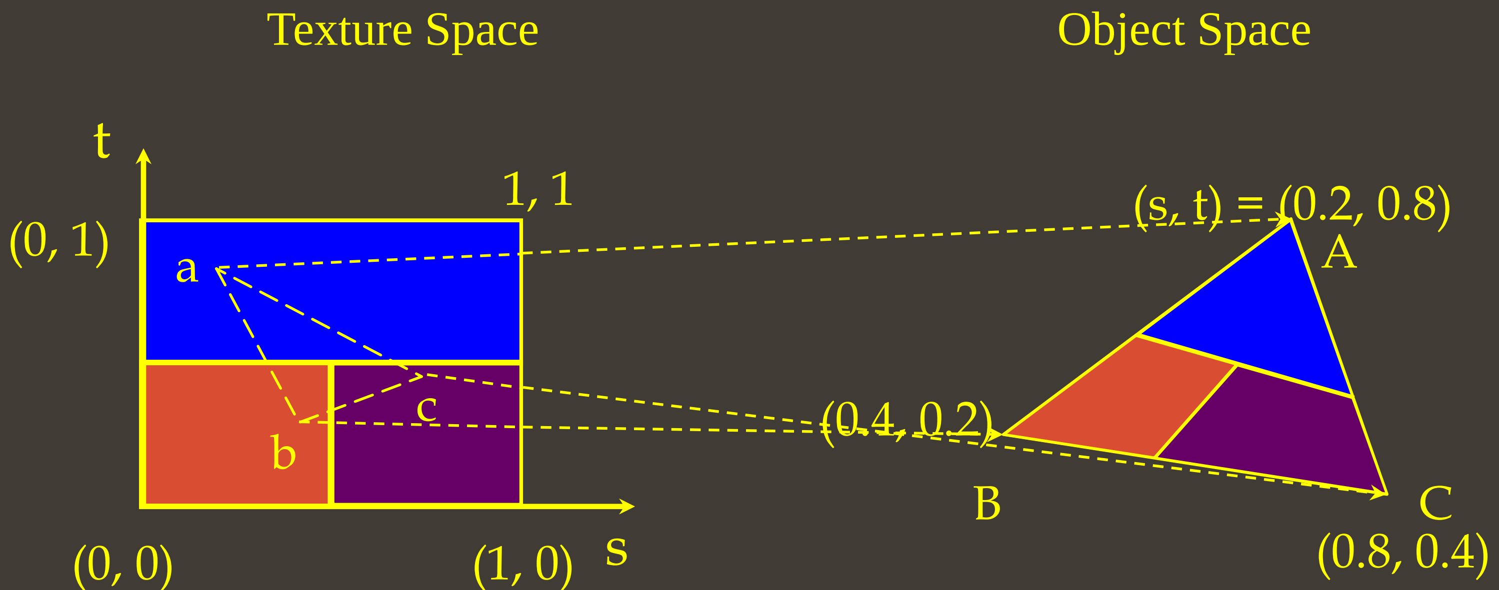
- Define a texture image from an array of *texels* in CPU memory

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize,
    texSize, 0,  gl.RGBA, gl.UNSIGNED_BYTE, image);
```

-

```
var image = document.getElementById("texImage");

gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB,
        gl.RGB, gl.UNSIGNED_BYTE, image );
```

Mapping Texture Coordinates



Applying the Texture in the Shader

```
precision mediump float;

varying vec4 fColor;
varying  vec2 fTexCoord;
uniform sampler2D texture;

void main()
{
    gl_FragColor = fColor*texture2D( texture, fTexCoord );
}
```

Applying Texture to Cube

```
// add texture coordinate attribute  to quad function

function quad(a, b, c, d)
{    pointsArray.push(vertices[a]);
     colorsArray.push(vertexColors[a]);
     texCoordsArray.push(texCoord[0]);

     pointsArray.push(vertices[b]);
     colorsArray.push(vertexColors[a]);
    texCoordsArray.push(texCoord[1]);
     .
     .
 }
```

Creating a Texture Image

```
var image1 = new Array()
    for (var i =0; i<texSize; i++)  image1[i] = new Array();
    for (var i =0; i<texSize; i++)
         for ( var j = 0; j < texSize; j++)
            image1[i][j] = new Float32Array(4);
    for (var i =0; i<texSize; i++) for (var j=0; j<texSize; j++) {
        var c = (((i & 0x8) == 0) ^ ((j & 0x8)  == 0));
        image1[i][j] = [c, c, c, 1];    }

// Convert floats to ubytes for texture
var image2 = new Uint8Array(4*texSize*texSize);
    for ( var i = 0; i < texSize; i++ )
         for ( var j = 0; j < texSize; j++ )
            for(var k =0; k<4; k++)
                image2[4*texSize*i+4*j+k] = 255*image1[i][j][k];
```

Texture Object

```
texture = gl.createTexture();

gl.activeTexture( gl.TEXTURE0 );    gl.bindTexture( gl.TEXTURE_2D, texture );
 //   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize, texSize, 0,
        gl.RGBA, gl.UNSIGNED_BYTE, image);
    gl.generateMipmap( gl.TEXTURE_2D );
gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
        gl.NEAREST_MIPMAP_LINEAR );    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
        gl.NEAREST );
```

Vertex Shader

```
attribute vec4 vPosition;
attribute vec4 vColor;
attribute vec2 vTexCoord;

varying vec4 color;
varying vec2 texCoord;

void main()
{
    color       = vColor;
    texCoord    = vTexCoord;
    gl_Position = vPosition;
}
```

Fragment Shader

```
varying vec4 color;
varying vec2 texCoord;

uniform sampler texture;

void main()
{
  gl_FragColor = color * texture( texture, texCoord );
}
```

What we haven’t talked about

- Off-screen rendering
- Compositing
- Cube maps

What’s missing in WebGL (for now)

- Other shader stages
  - geometry shaders
  - tessellation shaders
  - compute shaders
    - WebCL exists
- Vertex Array Objects

Books

- Modern OpenGL
  - The OpenGL Programming Guide, 8th Edition
  - Interactive Computer Graphics: A Top-down Approach using WebGL, 7th Edition
  - WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL
  - The OpenGL Superbible, 5th Edition
- Other resources
  - The OpenGL Shading Language Guide, 3rd Edition
  - OpenGL and the X Window System
  - OpenGL Programming for Mac OS X
  - OpenGL ES 2.0 Programming Guide

Online Resources

- The OpenGL Website: www.opengl.org
  - API specifications
  - Reference pages and developer resources
  - Downloadable OpenGL (and other APIs) reference cards
  - Discussion forums
- The Khronos Website: www.khronos.org
  - Overview of all Khronos APIs
  - Numerous presentations
- get.webgl.org
- www.cs.unm.edu/~angel/WebGL/7E
- www.chromeexperiments.com/webgl

Q & A

Thanks for Coming!

Thanks!

- Feel free to drop us any questions:

angel@cs.unm.edu
shreiner@siggraph.org

- Course notes and programs available at

www.daveshreiner.com/SIGGRAPH
www.cs.unm.edu/~angel