# EVEN MORE OPENGL

Pramook Khungurn

CS 4621, Fall 2011

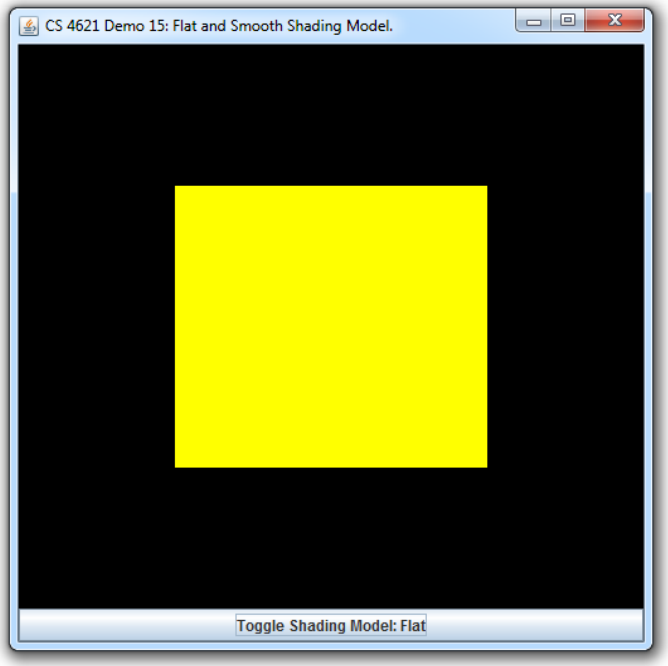
# SHADING & LIGHTING

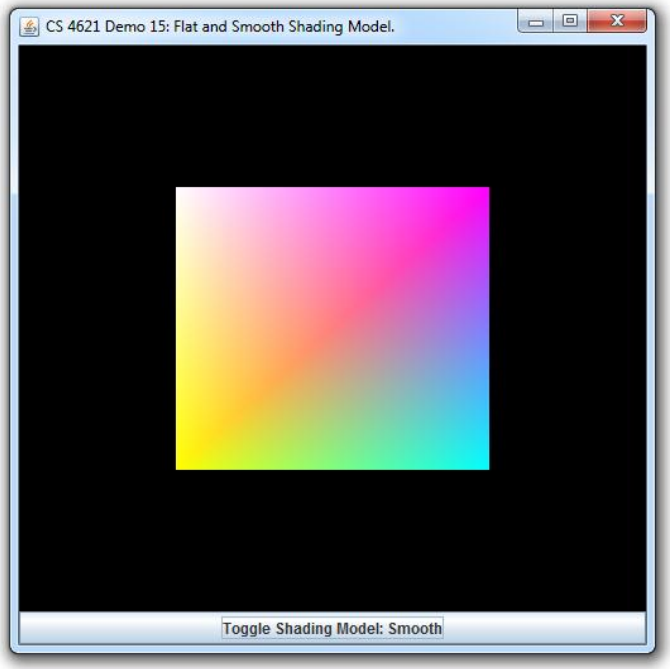
# OpenGL Shading Models

- So far:
  - Specify colors of vertices.
  - Vertices of a triangle
    - Same color → triangle have same color
    - Different colors → depends on shading model
- Flat shading
  - Use the color of the first vertex for the triangle.
- Smooth shading
  - Interpolate color using barycentric coordinate.

# Specifying Shading Models

- gl.glShadeModel(GL2.GL_FLAT);
- gl.glShadeModel(GL2.GL_SMOOTH);



# “Lighting”

- OpenGL implements Phong reflectance model.
  - Similar to the one you implemented in PA1.
- This is (somehow) called “lighting” in OpenGL.
- “Lighting” is done per vertex.
  - Input: lighting info, material info, vertex position and normal
  - Output: one color per pixel
  - Vertex colors are copied/interpolated based on shading models.

# Using “Lighting”

- Enable it.
  - gl.glEnable(GL2.GL_LIGHTING);
  - After enabling, glColor3f no longer works.

- Specify light sources.
- Specify “material” to use.
- Specify a normal for each vertex of polygons.

- Disable it.
  - gl.glDisable(GL2.GL_LIGHTING);

# Specifying Light Sources

- OpenGL keeps track of 8 light sources.
  - Referred to by GL_LIGHT0, GL_LIGHT1, …, GL_LIGHT7.
- To use one, enable it.
  - gl.glEnable(GL2.GL_LIGHT0);
- Then specify its parameters.
  - Ambient intensity
  - Diffuse intensity
  - Specular intensity
  - Position
  - Spotlight parameters (won't cover)
  - Attenuation parameters (won't cover)

# Example

```
float[] lightAmbient = new float[] { 1.0f, 1.0f, 1.0f, 1.0f };
float[] lightDiffuse = new float[] { 1.0f, 1.0f, 1.0f, 1.0f };
float[] lightSpecular = new float[] { 1.0f, 1.0f, 1.0f, 1.0f };
float[] lightPosition = new float[] { 5, 5, 5, 1.0f};

gl.glEnable(GL2.GL_LIGHTING);
gl.glEnable(GL2.GL_LIGHT0);

gl.glLightfv(GL2.GL_LIGHT0, GL2.GL_AMBIENT, lightAmbient, 0 );
gl.glLightfv(GL2.GL_LIGHT0, GL2.GL_DIFFUSE, lightDiffuse, 0 );
gl.glLightfv(GL2.GL_LIGHT0, GL2.GL_SPECULAR, lightSpecular, 0 );
gl.glLightfv(GL2.GL_LIGHT0, GL2.GL_POSITION, lightPosition, 0);
```

# glLightfv

- Specify a light's parameters.

- Arguments
  - 1st = light's ID
  - 2nd = the parameter you want to set
    - GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION
  - 3rd = the actual data as an array
  - 4th = the start offset of the array

- All parameters above expect array of size at least 4.
  - RGBA for colors.
  - Homogeneous coordinate for the light's position.

# Point Light Source

- Position's w = 1.
- Similar to the light source implemented in PA1.
- Intensity can be attenuated by distance from light source.
- Can be made a spotlight by specifying more parameters.

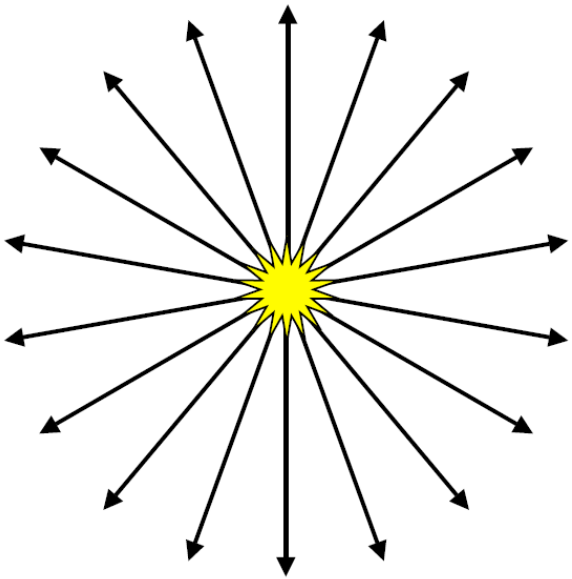


Image stolen from Jerry Talton's slides for UCSC's CMP 160: Intro to Computer Graphics, 2006

# Directional Light Source

- Position's w = 0 → becomes light direction.
- Light direction constant for every pixel.
- Intensity constant for every pixel.
- Used to model far away light sources, i.e. the sun.

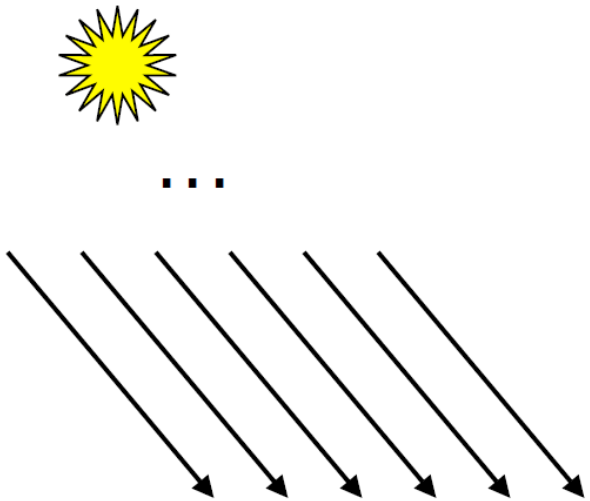


Image stolen from Jerry Talton's slides for UCSC's CMP 160: Intro to Computer Graphics, 2006

# Light Position and Transforms

- Light position gets transformed like a normal vertex.

- Many choices for where to specify:
  - After modeling transform → can specify in light's object space.
  - After view transform → can specify position in the scene.
  - After projection transform → light moves with the camera.

# Specifying Material

- OpenGL keeps track of two Phong materials.
  - One for front face.
  - One for back face.
  - Need a different parameter → respecify the material for that face
  - Must specify materials outside glBegin(…), glEnd() block.

- Parameters
  - Ambient color
  - Diffuse color
  - Specular color
  - Shininess

# Example

```
float[] ambient = new float[] {0.05f, 0.05f, 0.05f, 0.05f};
float[] diffuse = new float[] {1.0f, 0.0f, 0.0f, 0.0f};
float[] specular = new float[] {1.0f, 1.0f, 1.0f, 1.0f};
float   shininess = 50.0f;

gl.glMaterialfv(GL2.GL_FRONT, GL2.GL_AMBIENT, ambient, 0);
gl.glMaterialfv(GL2.GL_FRONT, GL2.GL_DIFFUSE, iffuse, 0);
gl.glMaterialfv(GL2.GL_FRONT, GL2.GL_SPECULAR, specular, 0);
gl.glMaterialf(GL2.GL_FRONT, GL2.GL_SHININESS, shininess);
```

# glMaterialfv and glMaterialf

- Specify a material's parameter.

- Arguments
  - 1st = the face
    - GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
  - 2nd = the parameter
    - GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_SHININESS
    - First three → glMaterialfv. Last → glMaterialf
  - 3rd = the data
    - glMaterialfv: an array of floats
    - glMaterialf: just a float
  - 4th = array offset (only for glMaterialfv)

# Specifying Normal for Each Vertex

- Use glNormal3f/glNormal3d.
- Works the same way as glColor3f.
  - Specified normal is remembered until next glNormal command.
- Normals are always specified in **object space**.
  - Gets multiplied by inverse transpose of transformation matrix.
- OpenGL don’t normalize transformed normals by default.
  - Might result in weird color when do Phong lighting.
  - To get it to normalize, use gl.glEnable(GL2.GL_NORMALIZE);

# Example

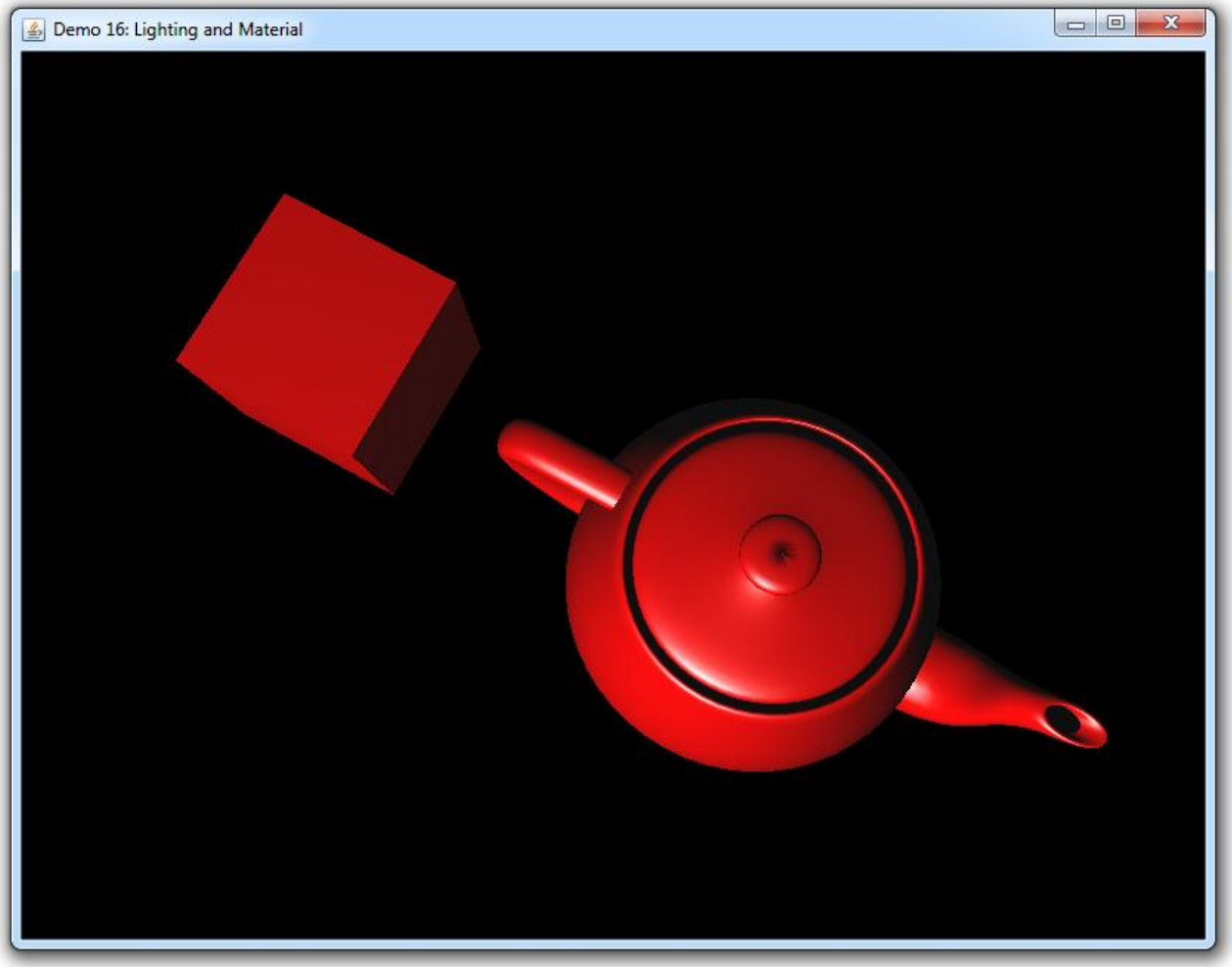
```
protected void drawCube(GL2 gl) {
      gl.glBegin(GL2.GL_QUADS);
            // Front Face
          gl.glNormal3f( 0.0f, 0.0f, 1.0f);
          gl.glVertex3f(-1.0f, -1.0f,  1.0f);
          gl.glVertex3f( 1.0f, -1.0f,  1.0f);
          gl.glVertex3f( 1.0f,  1.0f,  1.0f);
          gl.glVertex3f(-1.0f,  1.0f,  1.0f);

            // Back Face
          gl.glNormal3f( 0.0f, 0.0f,-1.0f);
          gl.glVertex3f(-1.0f, -1.0f, -1.0f);
          gl.glVertex3f(-1.0f,  1.0f, -1.0f);
          gl.glVertex3f( 1.0f,  1.0f, -1.0f);
          gl.glVertex3f( 1.0f, -1.0f, -1.0f);

            :
            :
            :

      gl.glEnd();
}
```

# Demo 16



# There is no shadow!

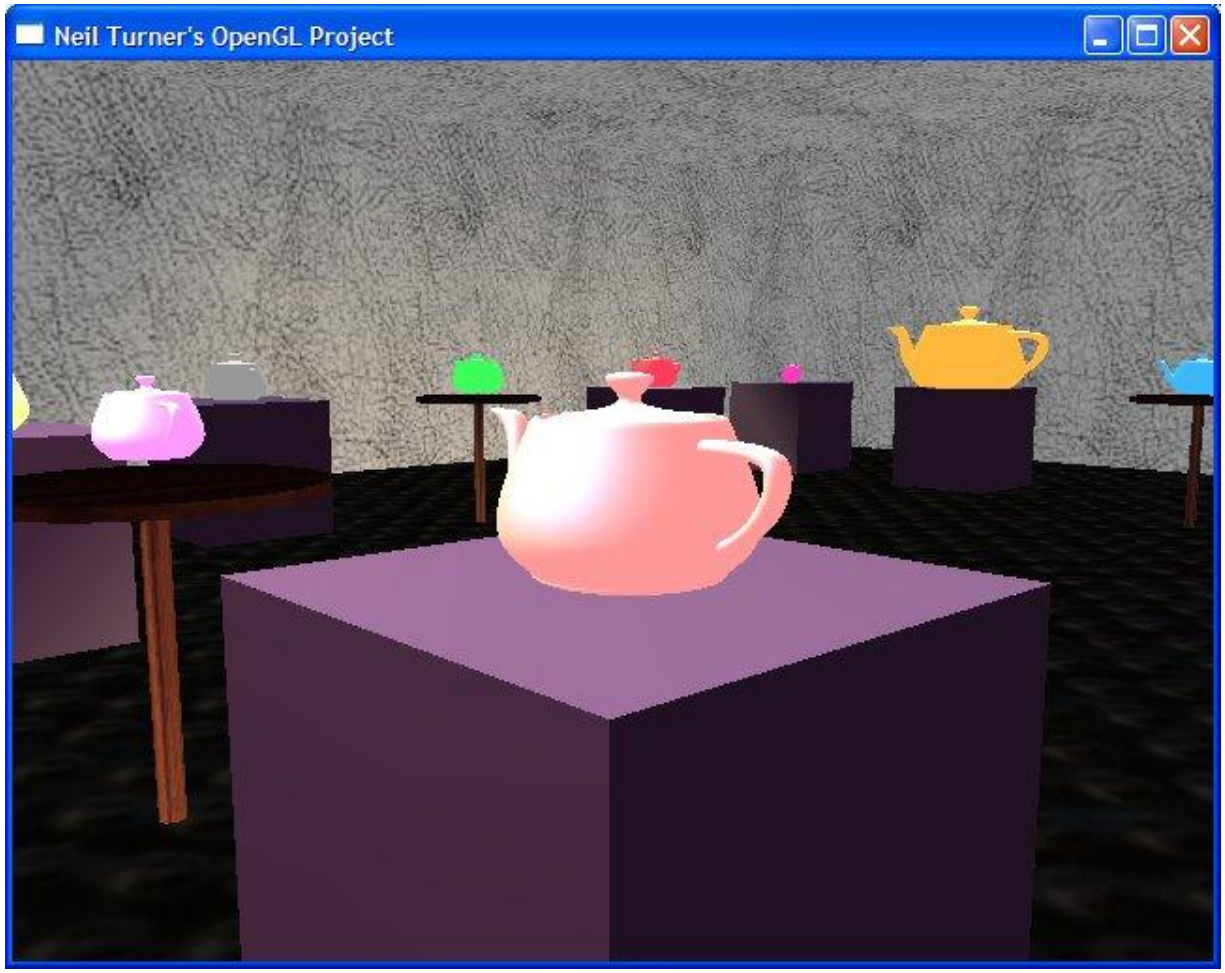
- OpenGL does not trace shadow rays.
- Light sources are assumed to always be visible.

- Simulating shadows in graphics pipeline uses very different techniques from ray tracking
  - Interested? Take CS 5625: Interactive Graphics

# SHADER PROGRAMMING

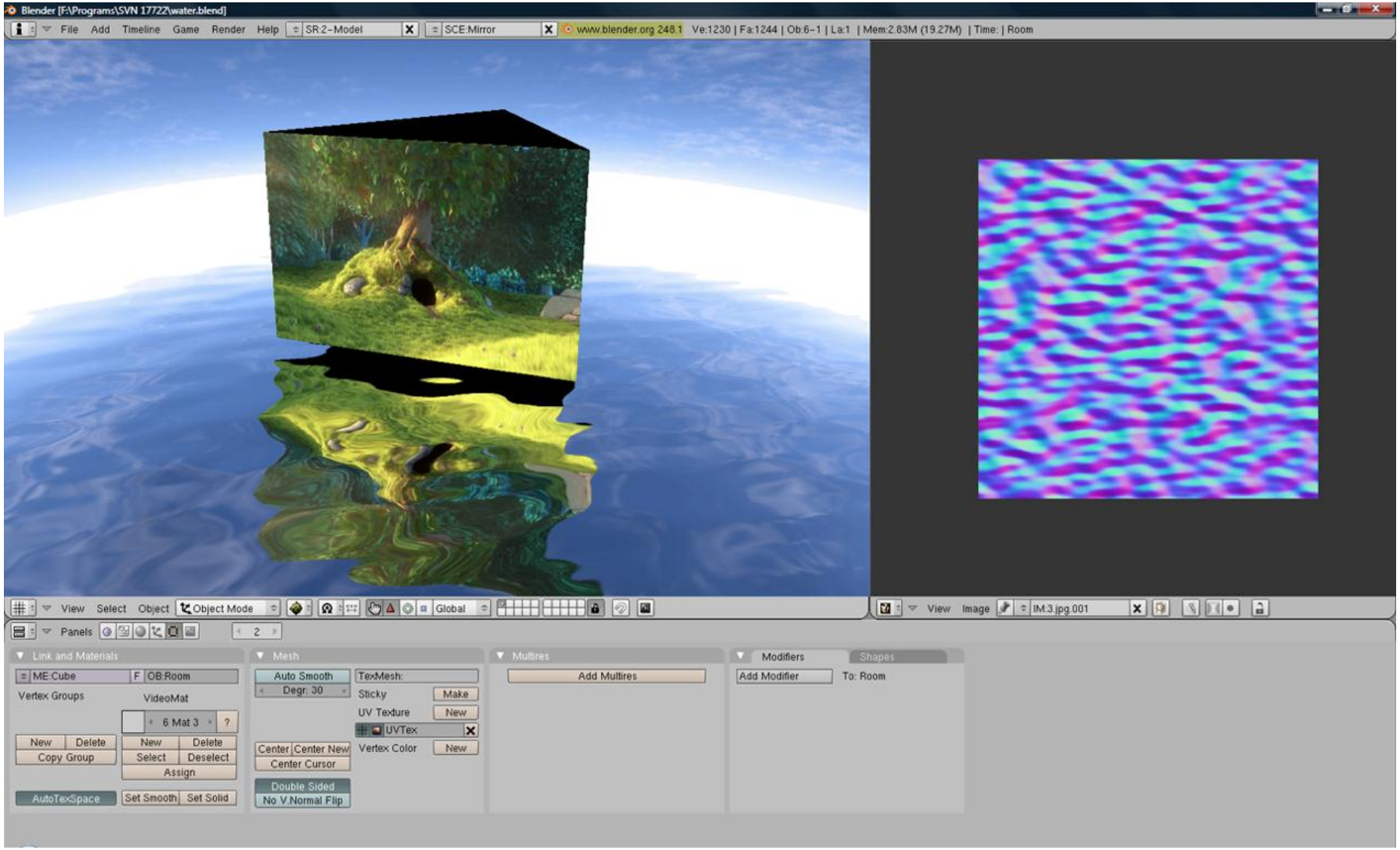
Based on Jian Huang's lecture on Shader Programming

# What OpenGL 15 years ago can do…




http://www.neilturner.me.uk/shots/opengl-big.jpg

# What OpenGL can do now…



# What's Changed?

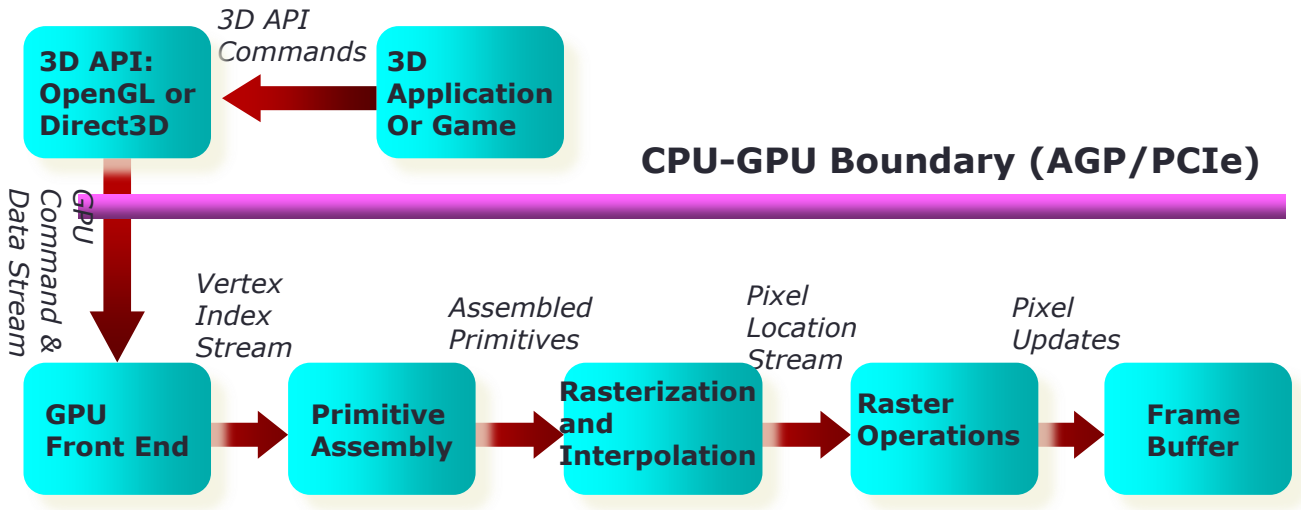
- 15 years ago:
  - Transform vertices with modelview/projection matrices.
  - Shade fragments with Phong lighting model only.

- Now:
  - Custom vertex transformation.
  - Custom lighting model.
  - More complicated visual effects.
    - Shadows
    - Displaced and detailed surfaces.
    - Simple reflections and refractions,
    - Etc.

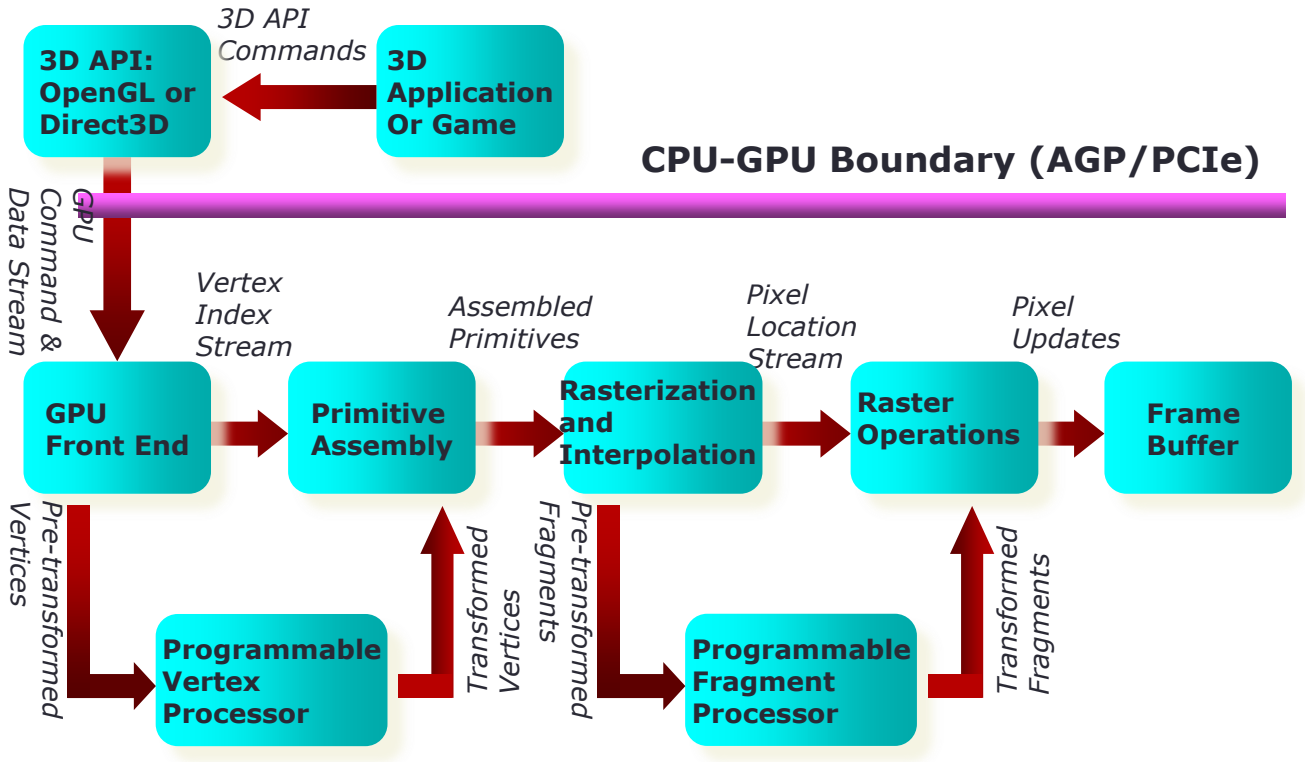
# What's Changed?

- 15 years ago:
  - Vertex transformation/fragment shading hardcoded into GPUs.
- Now:
  - More parts of the GPU are programmable (but not all).

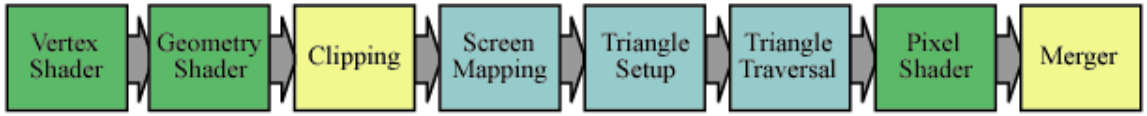
# GPU Pipeline in 2000



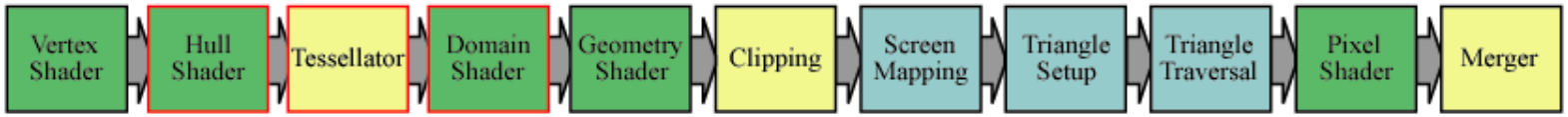
# GPU Pipeline in 2004



# GPU Pipeline Nowadays




หรือ



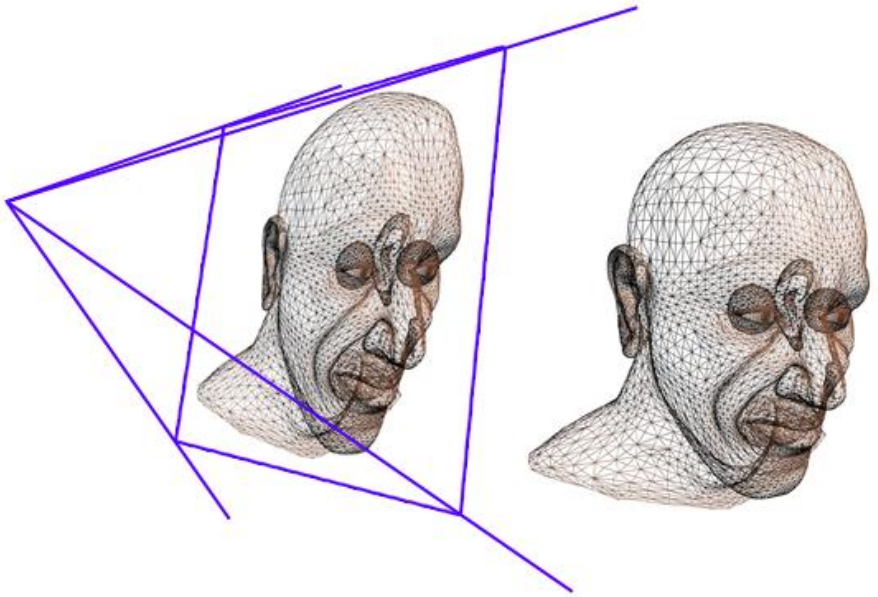

ภาพจาก http://www.realtimerendering.com/blog/direct3d-11-details-part-ii-tessellation/

# Shader

- A small program to control a part of the graphics pipeline
  - Vertex shader controls vertex transformation.
  - Fragment shader controls fragment shading.

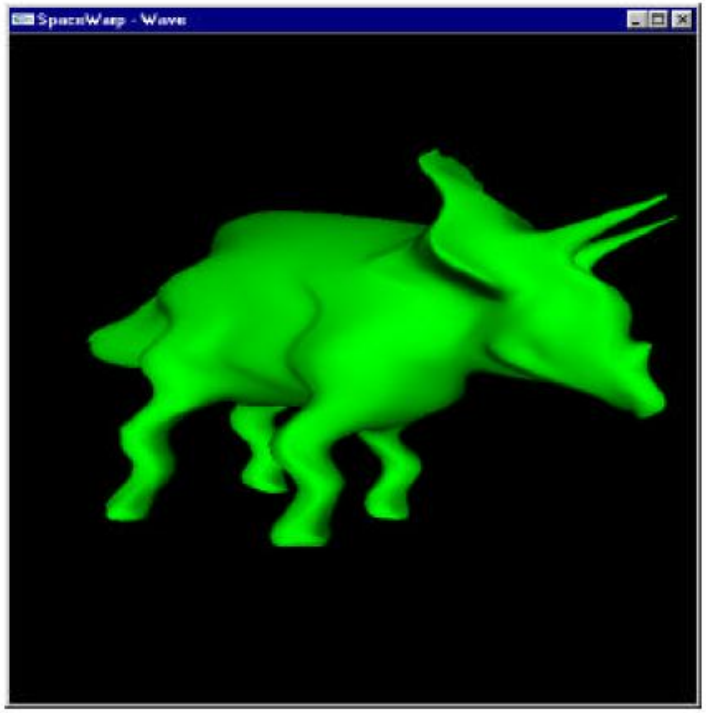
# Vertex Shader

- Transform vertices from object space to clip space.
  - Doesn’t have to be modelview followed by projection.
- Compute other data that are interpolated with vertices.
  - Color
  - Normals
  - Texture coordinates
  - Etc.



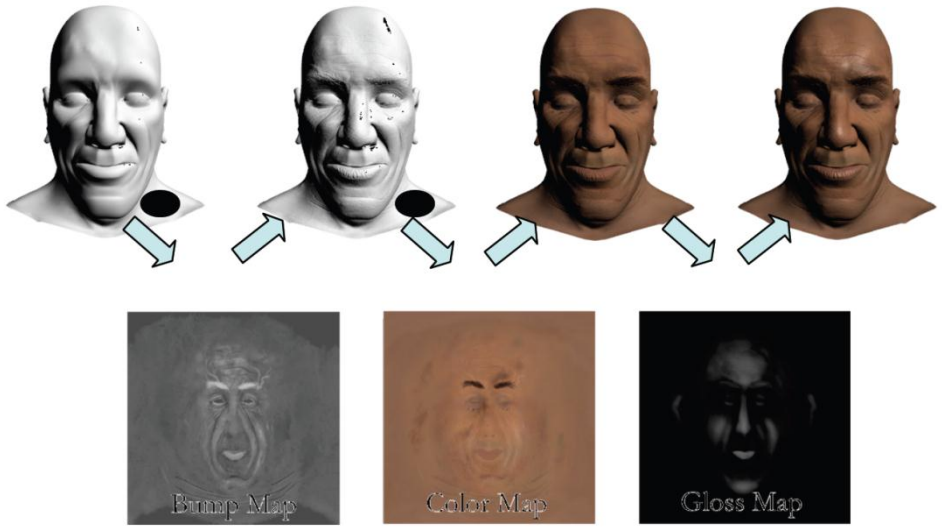
# What can we do with a vertex shader?

- Displaced/distorted surfaces



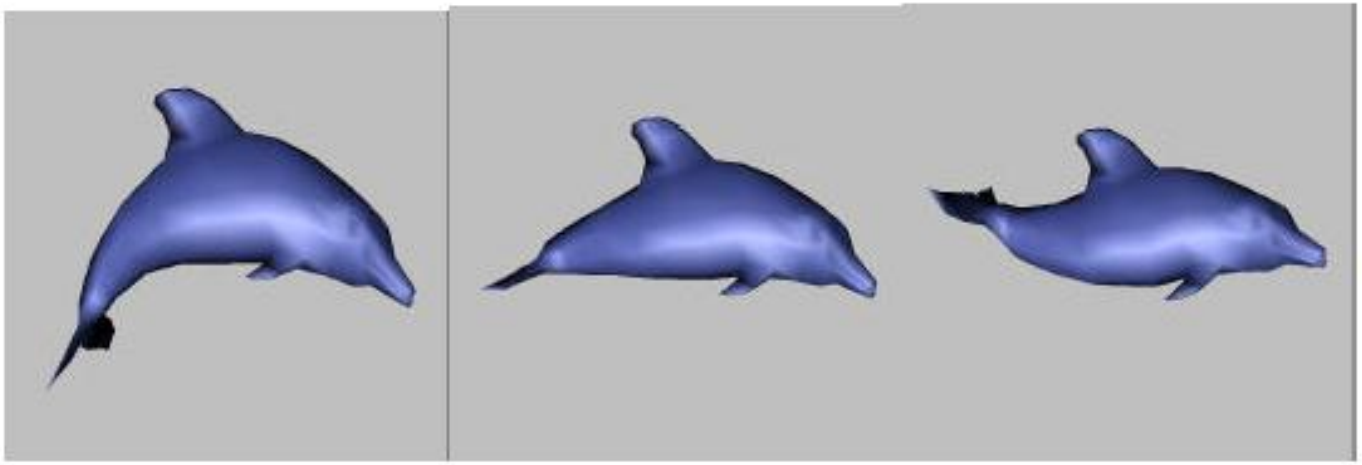

# Fragment Shader

- Compute the color of a fragment.
- Take interpolated data from vertex shaders.
- Can read more data from:
  - Textures
  - User specified values.



# What can we do with a vertex shader?

- Skinning

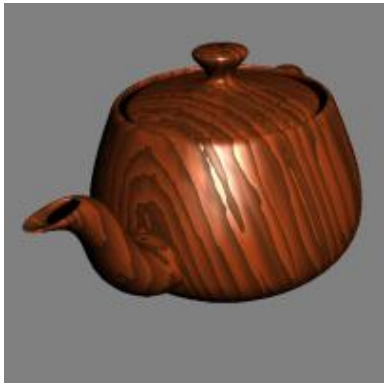


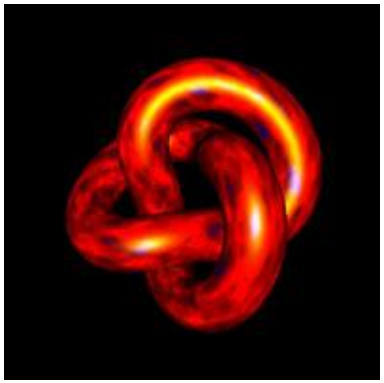
Dolphin #1 Morphed Dolphin Dolphin #2

Images courtesy of Microsoft

# What can we do with a fragment shader?

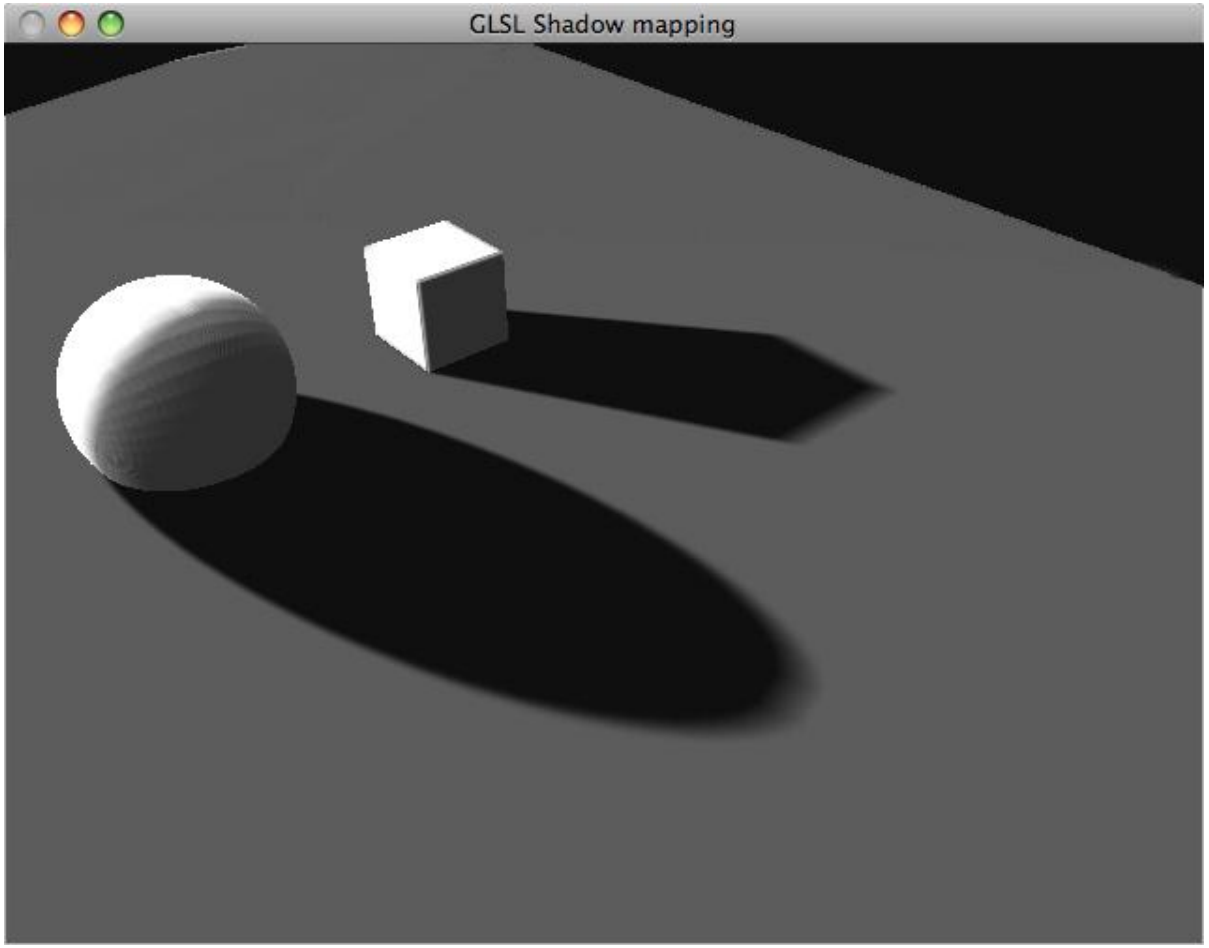
- More complicated materials:
  - Glossy
  - Reflective, refractive
  - Rough, bumpy, lots of nooks and crannies
  - Wooden





# What can we do with a fragment shader?

- Shadow




http://www.fabiensanglard.net/shadowmappingPCF/8x8kernel_nVidia.jpg

# GLSL

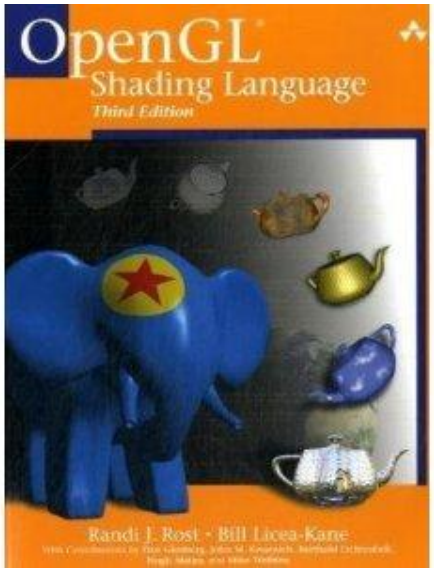
# GLSL

- Similar to C/C++

- Used to write shaders
  - Vertex, fragment, geometry
  - We only cover vertex and fragment here.

- Based on OpenGL
- First available in OpenGL 2.0 (2004)

- Competitors:
  - Nvidia Cg
  - Microsoft HLSL



# GLSL Program

- A collection of shaders that run together.
  - At least one vertex shader or one fragment shader.
  - Should have both so we know its behavior completely.

- At any time, the GPU runs only one program.
  - Initially, OpenGL uses its own default program.
  - We can tell OpenGL to use our program.
  - We can also tell it to go back and use its initial program.

# Shader

- Shader source code resamples C/C++ source code.
  - Similar data types, expressions, and control statements.
  - Functions are written in the same way.
- Entry point = “void main( )”
  - Not “int main(int argc, char **argv)” as in normal C.
- A shader’s source code can reside in many files.
  - However, only one of them must have the main function.
- Two main functions when writing a vertex shader and a fragment shader together.

# Shader Structure

```
/*
Multiple-lined comment
*/

// Single-lined comment

//
// Global variable definitions
//

void main()
{
    //
    // Function body
    //
}
```

# GREEN SHADER

# Vertex Shader

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

# Fragment Shader

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

# OpenGL/GLSL Plumbling

- Supposed we have already created the program.
- We tell OpenGL to use it.
- We then instruct OpenGL to draw a triangle:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

# OpenGL/GLSL Plumbling

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

GLSL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

OpenGL Code

# OpenGL/GLSL Plumbling

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

GLSL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

OpenGL

gl_ModelViewProjectionMatrix is a special variable that is equal to the product of projection matrix and modelview matrix

# OpenGL/GLSL Plumbling

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

GLSL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

OpenGL

A vertex shader is executed each time a vertex is specified.

So, since there are three calls to glVertex3f, the above vertex shader is executed 3 times.

1st time: gl_Vertex = (-0.5, -0.5, 0, 1)
2nd time: gl_Vertex = (0.5, -0.5, 0, 1)
3rd time: gl_Vertex = (0, 0.5, 0, 1)

# OpenGL/GLSL Plumbling

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

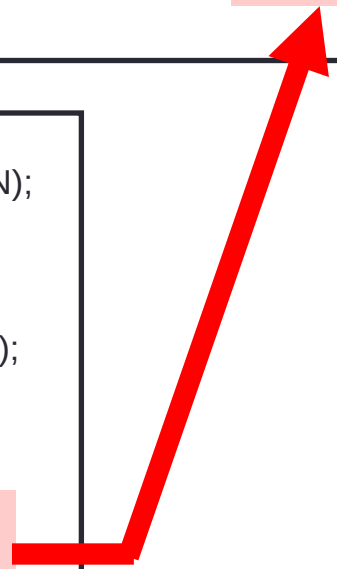
GLSL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

OpenGL



gl_Vertex
is another special variable that holds the value we specify with glVertex

# OpenGL/GLSL Plumbling

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

GLSL

gl_Position is a special variable that holds the position of the vertex in clip space.

Since a vertex shader's main output is the position in clip space. It must always set gl_Position.

So, the vertex shader above just does what OpenGL normally does.

# OpenGL/GLSL Plumbling

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

GLSL

# OpenGL/GLSL Plumbling

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

โค้ด GLSL

gl_FragColor is a special variable that stores the colof of the output fragment

Since a fragment shader computes the color of a fragment. It must always set gl_FragColor.

# OpenGL/GLSL Plumbling

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

โค้ด GLSL

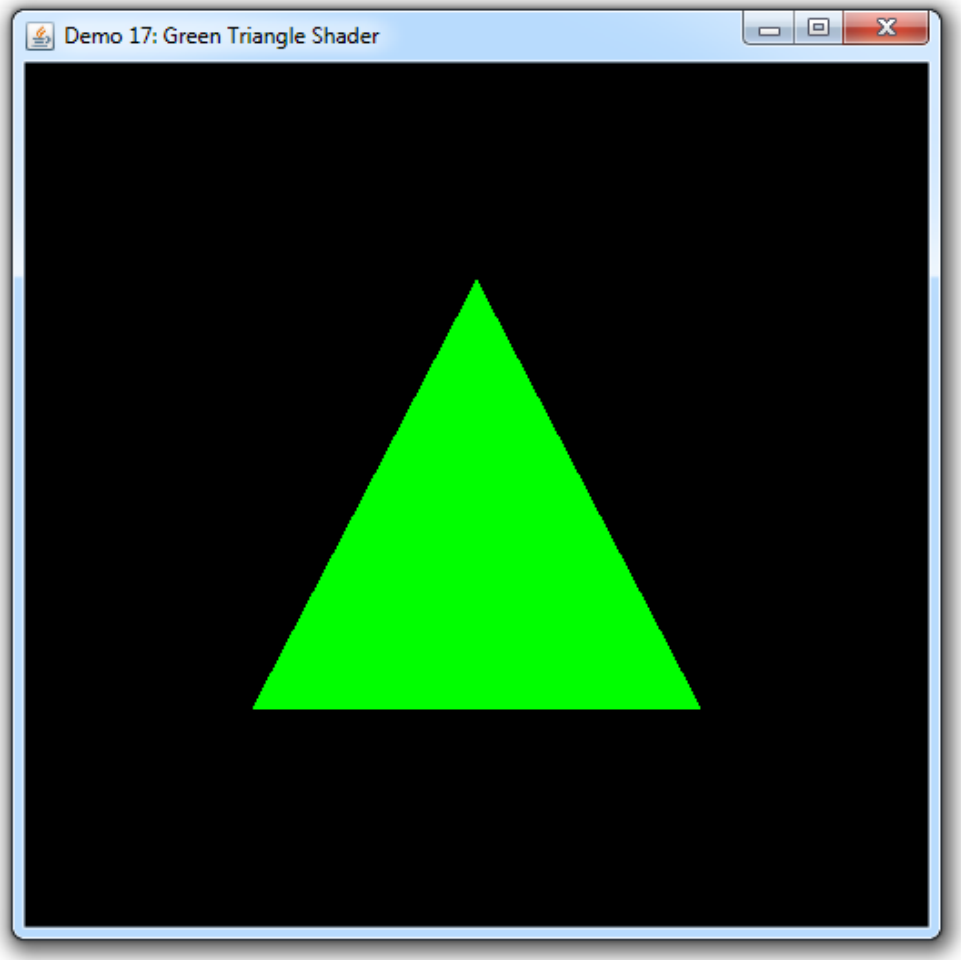
vec4 is a data type of 4D vector.

Can be used to store

- homogeneous coordinate
- สี RGBA

vec4(0,1,0,1) constructs an RGBA tuple with R=0, G=1, B=0, A=1, which is green.

# Demo 17



# USING GLSL IN JAVA

# To use a GLSL program…

- Follow the following 7 steps.

1. Create shader objects.
2. Read source code from files and feed them to the shader objects just created.
3. Compile the shader.
4. Create a program object.
5. Attach the shaders to the program.
6. Link the program.
7. Tell OpenGL to use you shaders instead of the default ones.

# To use a GLSL program…

- Follow the following 7 steps.

1. Create shader objects.
2. Read source code from files and feed them to the shader objects just created.
3. Compile the shaders.
4. Create a program object.
5. Attach the shaders to the program.
6. Link the program.
7. Tell OpenGL to use you shaders instead of the default ones.

WHAT A PAIN!

# CS 4621 Framework

- A collection of classes that take away some pain from OpenGL programming.

- Features:
  - GUI
  - Camera control
  - Picking
  - Creating and using textures
  - Creating and using GLSL programs
  - More to come…

- A version is distributed with this lecture's sample code.

# Now, to use a GLSL program…

- Create a Program object.

```
private Program greenShaderProgram = null;

public void init(GLAutoDrawable drawable) {
        final GL2 gl = drawable.getGL().getGL2();

        try {

                // Load, compile and link the shaders
                this.greenShaderProgram = new Program(
                        gl,
                        "src/shaders/green_shader.vert",
                        "src/shaders/green_shader.frag");

        } catch (GlslException e) {
                System.err.println(e.getMessage());
                System.exit(1);
        }
}
```

# Now, to use a GLSL program…

- Use it. Draw stuffs. Unuse it (if need be).
- Always change GLSL program outside glBegin(…) glEnd() block.

```
greenShaderProgram.use();

gl.glBegin(GL2.GL_TRIANGLES);
gl.glVertex3f(-0.5f, -0.5f, 0.0f);
gl.glVertex3f( 0.5f, -0.5f, 0.0f);
gl.glVertex3f( 0.0f,  0.5f, 0.0f);
gl.glEnd();

greenShaderProgram.unuse();
```