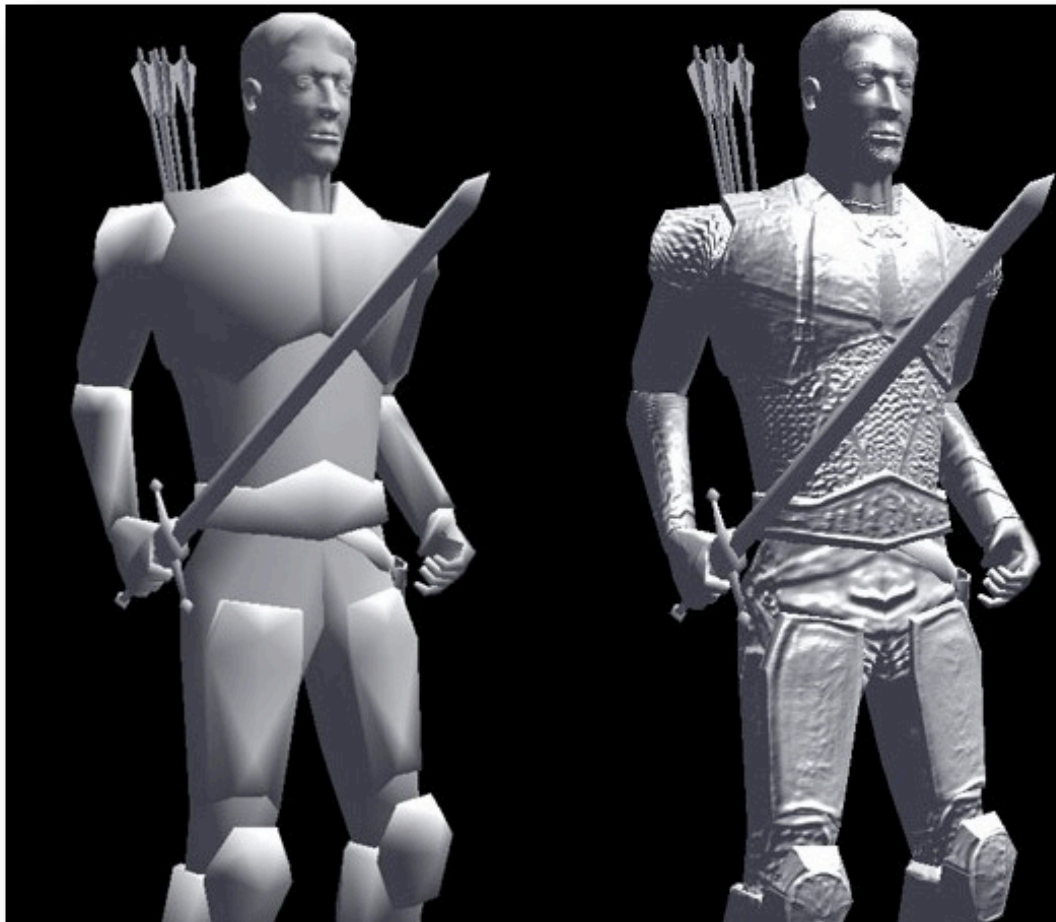
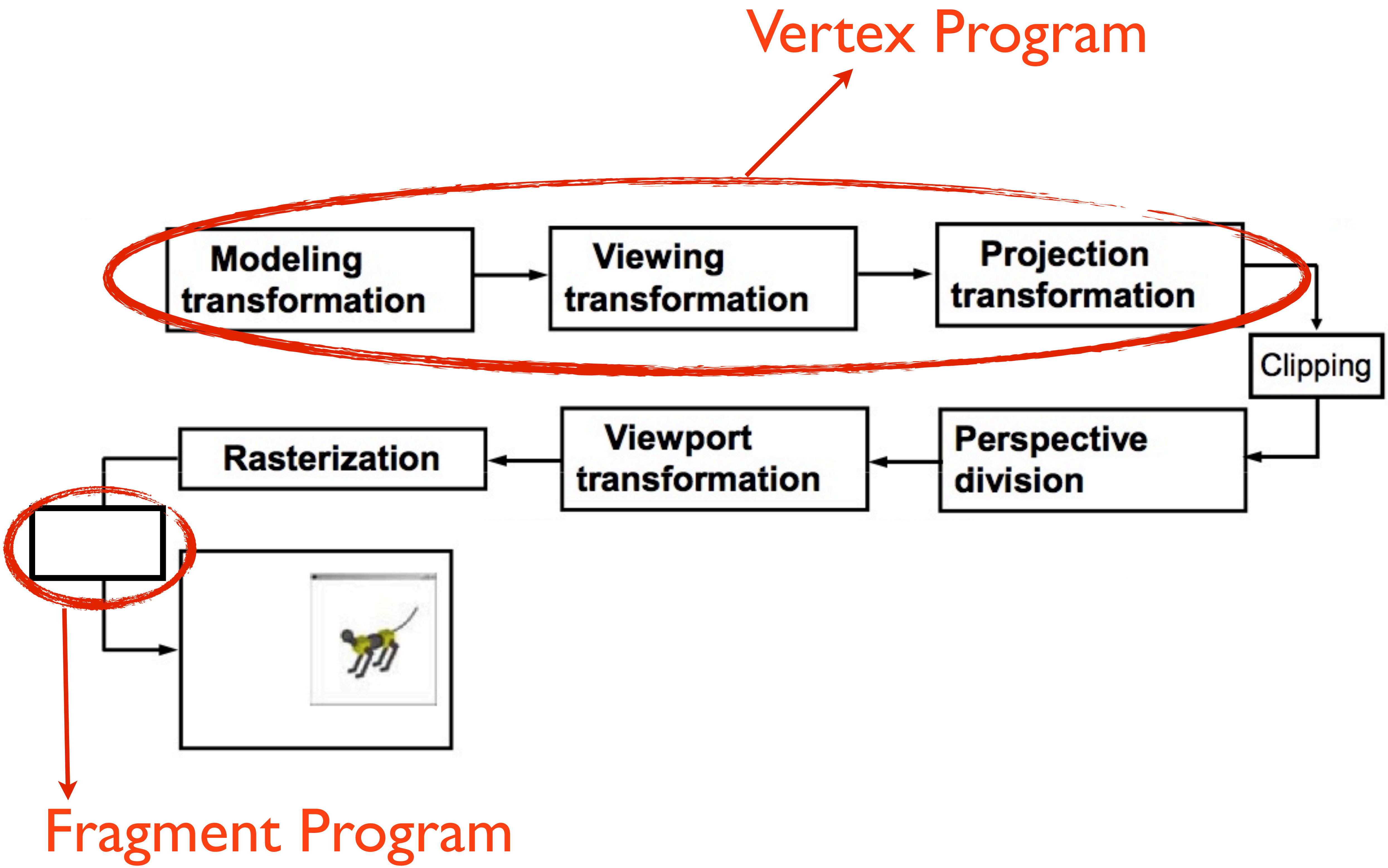
Programmable Shaders



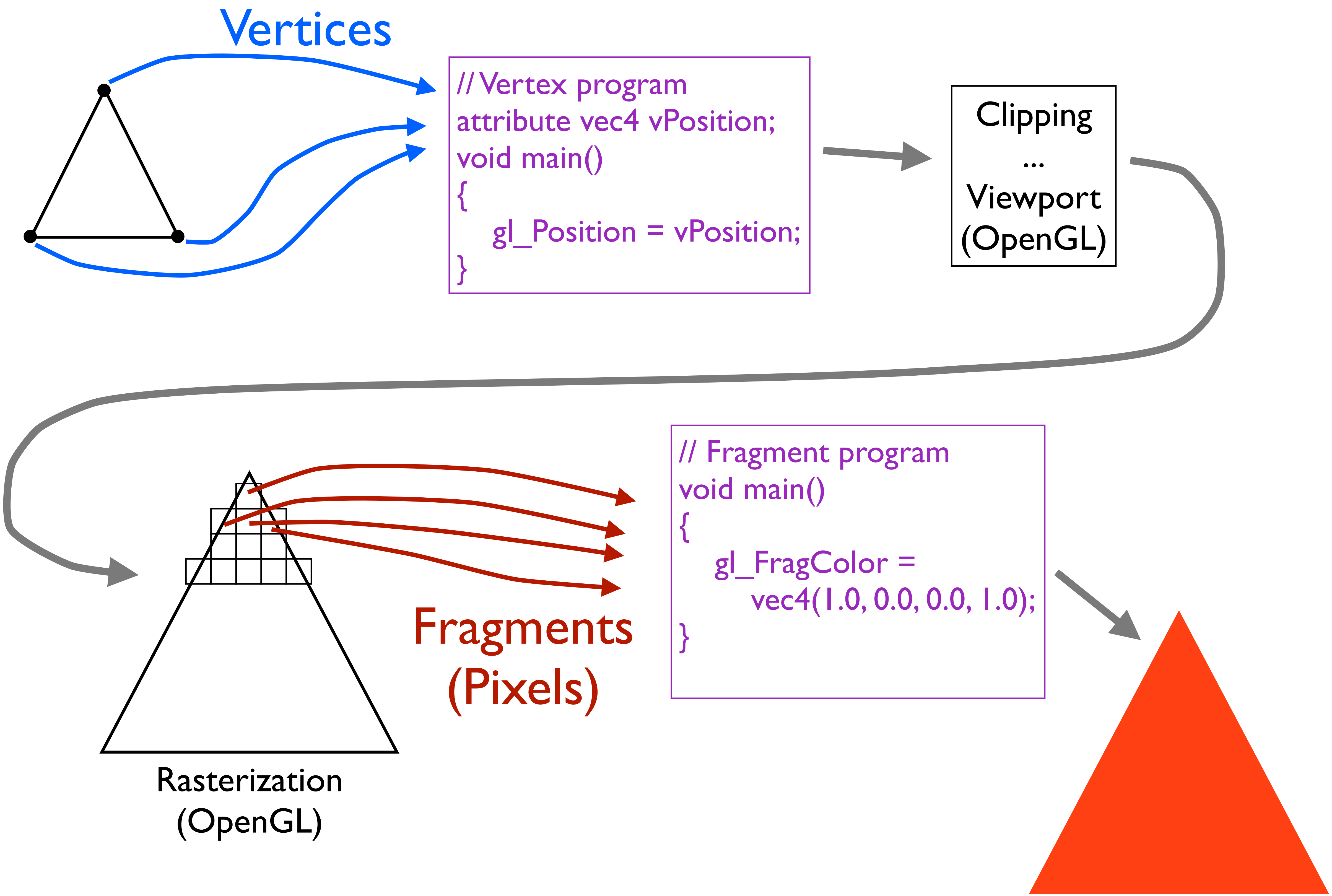
Programmable Shaders

- **Fixed-function pipeline:** OpenGL is programmed to transform your vertices and paint your pixels. You can only set parameters (input to these programs), like matrices, lights, colors.
- **Programmable pipeline:** you write programs to transform your vertices and paint your pixels!
- Programmable pipeline is more flexible. The programmer can figure out and implement all kinds of tricks.

Programmable Shaders



In a nutshell:



The simplest programs

```
// Vertex program
attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
```

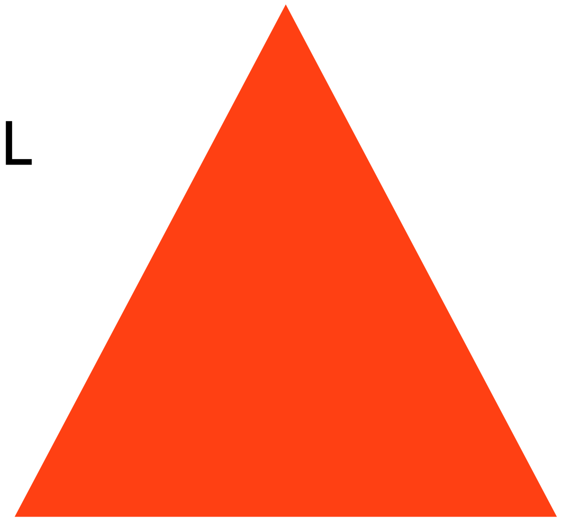
```
// Fragment program
void main()
{
    gl_FragColor =
        vec4(1.0, 0.0, 0.0, 1.0);
}
```

Pass-through:
vertices are copied
to the output.

gl_Position: standard GLSL variable. It's the vertex program output slot.

Constant color:
all pixels are painted
red.

gl_FragColor: standard GLSL variable. It's the fragment program output slot.



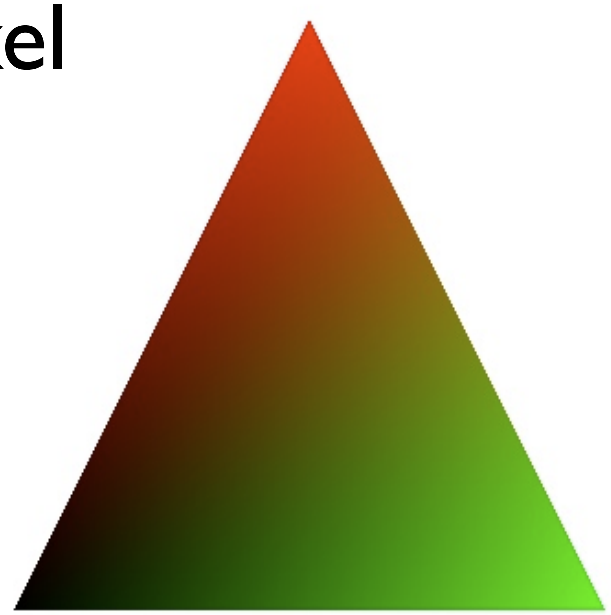
More interesting example

```
// Vertex program

attribute  vec3 vPosition;
attribute  vec3 vColor;
varying vec4 color;

void main()
{
    gl_Position = vec4(vPosition, 1.0);
    color = vec4( vColor, 1.0 );
}
```

We now have a per-vertex color attribute.

```
// Fragment program

varying  vec4 color;

void main()
{
    gl_FragColor = color;
}
```

OpenGL interpolates the color for each pixel (varying).



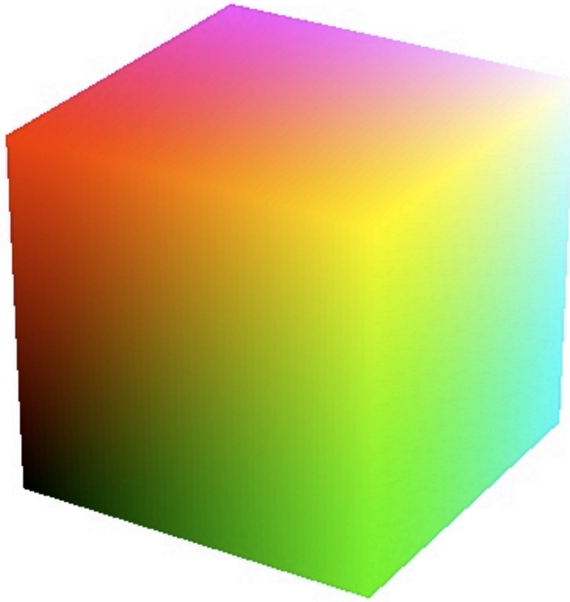
Even more interesting example

```
// Vertex program

uniform mat4 ModelView;
uniform mat4 Projection;

attribute  vec4 vPosition;
attribute  vec3 vColor;
varying vec4 color;

void main()
{
    gl_Position = Projection * ModelView * vPosition;
    color = vec4( vColor, 1.0 );
}
```

```
// Fragment program

varying  vec4 color;

void main()
{
    gl_FragColor = color;
}
```

Still painting it according to interpolated colors.

Now we finally transform the vertex!



Angel book, chapter 3, example 1.
http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SIXTH_EDITION/

Keywords

- **attribute** - variables passed from your C program to GLSL on a per-vertex basis. - attribute vec4 vPosition;
- **uniform** - variables whose values are the same for the whole primitive being processed. - uniform mat4 ModelView;
- **varying** - variables passed from the vertex program to the fragment program. Interpolated by Opengl.

  \- varying vec4 color;

http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf

The C / C++ side

Steps:

- **Load the shader**
  - Read shader source from disk and pass it to OpenGL. The graphics driver compiles it at runtime.
- **Set up vertex buffer** (and other attributes)
  - Vertex positions and other per-vertex attributes are stored in an array and passed to OpenGL. E.g.: all positions of vertices of a cube, arranged in triangles, go in the vertex array.
- **Set up uniforms** (including ModelView and Projection matrices)
  - For example, when you change the ModelView matrix, you need to call an OpenGL command to pass it to the shader.
- **Draw your model**
  - glDrawArrays(...);

The C / C++ side

1. Load the shader.

- Boilerplate code...

2x

```
GLuint program = glCreateProgram();
GLuint shader = glCreateShader( s.type );
glShaderSource( shader, 1, (const GLchar**) &s.source, NULL );
glCompileShader( shader );
glAttachShader( program, shader );
glLinkProgram( program );
glUseProgram( program );
```

- Don’t worry about the details.
- It’s done for you in InitShader.cpp.
- All you need is the variable *program* which is a handle to your shader.

The C / C++ side

2. Set up vertex buffers.

Important. Check OpenGL docs.

- More boilerplate code...

```
point4 points[3] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),     // This is where your
    point4( -0.5,  0.5,  0.5, 1.0 ),     // triangles go.
    point4(  0.5,  0.5,  0.5, 1.0 ) };   // Here's one triangle.
cubeData.numVertices = 3;
glGenVertexArrays( 1, &cubeData.vao );   // Generate one VA handle.
glBindVertexArray( cubeData.vao );       // Make it current.
GLuint buffer;
glGenBuffers( 1, &buffer );              // Generate one buffer handle.
glBindBuffer( GL_ARRAY_BUFFER, buffer ); // Make it current.

// Copy buffer to OpenGL.
glBufferData( GL_ARRAY_BUFFER, cubeData.numVertices, points,
              GL_STATIC_DRAW );

// Set the vPosition attribute.
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
                       BUFFER_OFFSET(0) );
```

- Done in Shapes.cpp.

The C / C++ side

Same code, viewed differently:

```
glGenVertexArrays( 1, &cubeData.vao );
glBindVertexArray( cubeData.vao );
```

```
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );

// Copy buffer to OpenGL.
glBufferData( GL_ARRAY_BUFFER, cubeData.numVertices,
              points, GL_STATIC_DRAW );
```

```
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
                       BUFFER_OFFSET(0) );
```

The C / C++ side

3. Set up uniforms.

- Prepare to use them: store references in global variables.

```
GLint uModelView, uProjection;
uModelView  = glGetUniformLocation( program, "ModelView"  );
uProjection = glGetUniformLocation( program, "Projection" );
```

- Change them as needed.

```
// mat4 class defined in Angel/mat.h.
// This is NOT OpenGL! It’s under control of the programmer.
mat4 model_view = mat4(1.0f);                     // Create identity.
model_view *= Translate(0.0f, 0.0f, -15.0f); // Multiply by a translation.

// This is OpenGL.
glUniformMatrix4fv( uModelView, 1, GL_TRUE, model_view );
```

The C / C++ side

4. Draw your model.

```
// Make your vertex array object become current.
// cubeData.vao is the previously stored handle to it.
glBindVertexArray( cubeData.vao );

// Draw the arrays. Tell OpenGL that they represent triangles.
glDrawArrays( GL_TRIANGLES, 0, cubeData.numVertices );
```

Summary

- Obtain handles to variables declared in your shader programs by using glGetAttribLocation, glGetUniformLocation.
- Assign values to your shader's variables through the handles obtained above by using glUniform3fv, glUniform4fv, glUniformMatrix4fv.

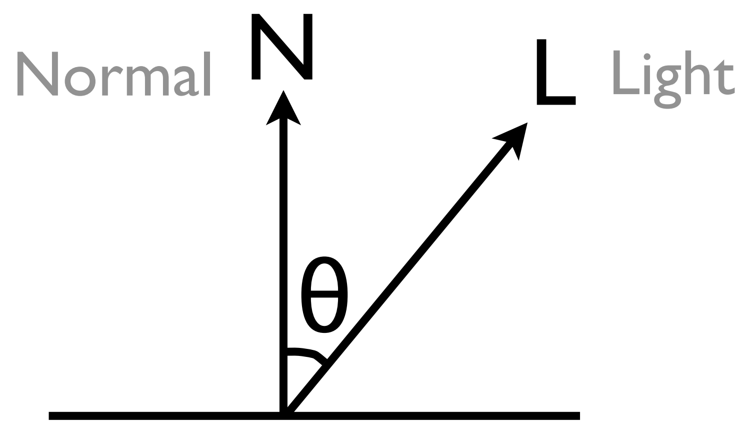
  (Remember: the ModelView matrix is now one of your shader variables; use glUniformMatrix4fv.)
- Call glBindVertexArray, glDrawArrays in sequence to draw one of your models.

More shaders

Now that we know how to write vertex / fragment shaders, what can we do with it?

Lighting

Diffuse



$$K_d = \cos\theta = N \cdot L$$

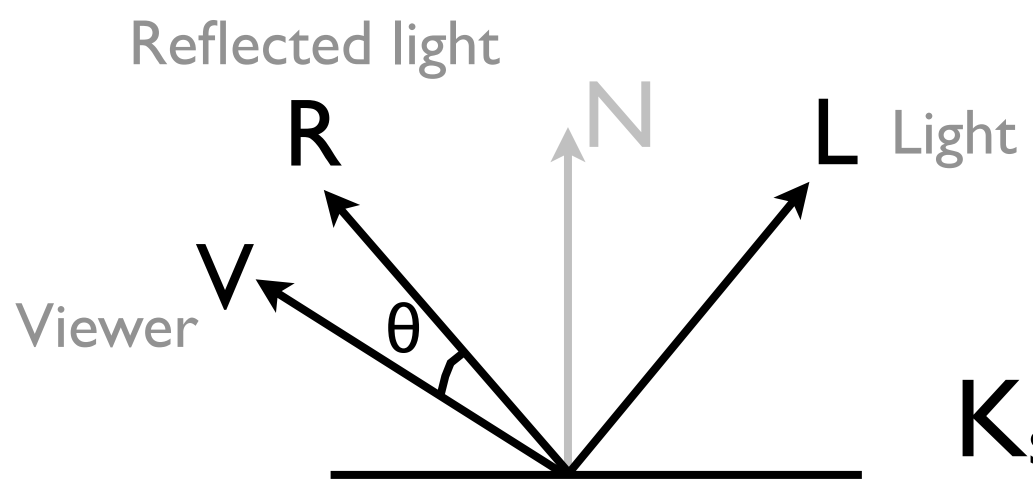
```
float Kd = max(dot(L, N), 0.0);
vec4 diffuse = Kd * DiffuseProduct;
```

Need to add:

- Attribute: normals (N);
- Uniform: light position (L);
- Uniform: DiffuseProduct (color and intensity of diffuse component).
    - Notice: there is a different notation with relation to Demetri's slides. Kd (in Demetri's slides) is DiffuseProduct (here).

Lighting

Specular



$$K_s = (\cos\theta)^s = (R \cdot V)^s$$

```
vec3 R = normalize(reflect(L, N));
float Ks = pow(max(dot(R, V), 0.0), Shininess);
vec4 specular = Ks * SpecularProduct;
```

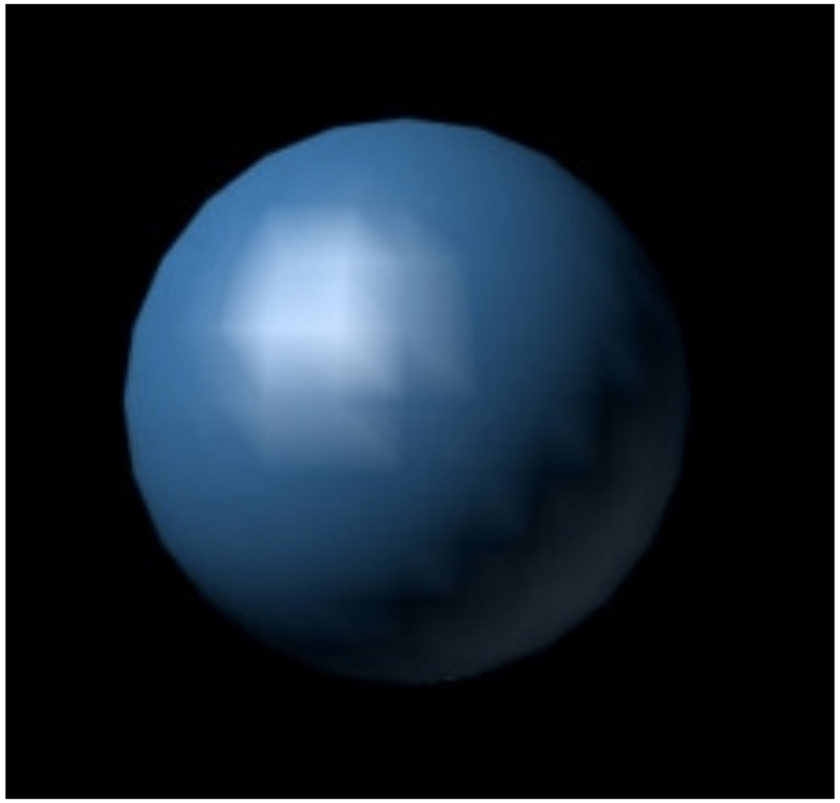
Need to add:

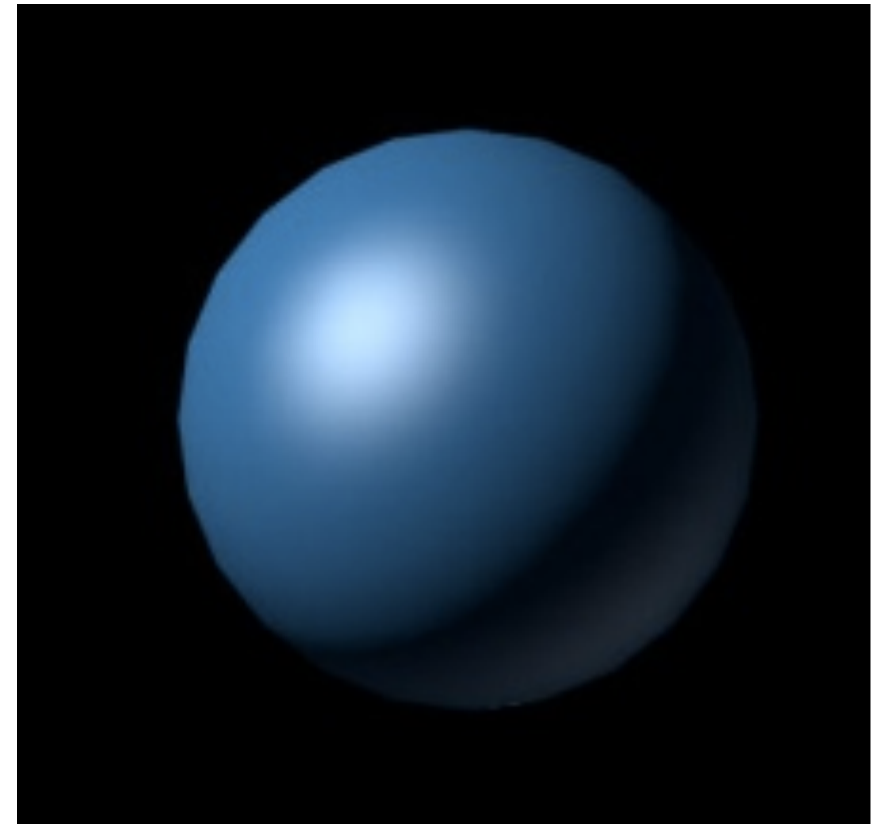
- Uniform: viewer position (V);
- Uniform: shininess (S) (also called specular exponent);
- Uniform: SpecularProduct (color and intensity of specular).
    - Notice: Ks (in Demetri's slides) is SpecularProduct (here).

Lighting

- Lighting can be done per-vertex or per-pixel.
- That means the lighting computation can go on the vertex or fragment shader.
- Per-vertex lighting: more efficient.
- Per-pixel lighting: more realistic.

Lighting



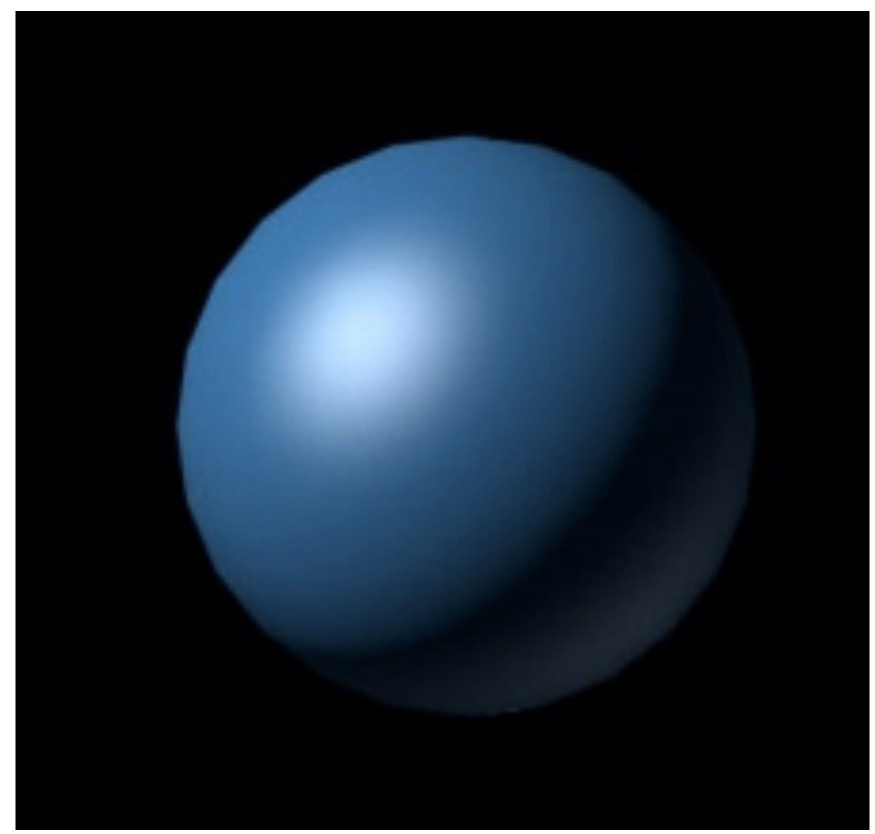


Per-vertex (Gouraud):
Light equation is
computed at vertices and
interpolated across face.

Per-pixel (Phong):
Light equation is
computed at every pixel.

Lighting

- Fixed-function pipeline used to be per-vertex.
- We'll go with per-pixel!
- What does the code look like?



Lighting

```
// Vertex program

attribute vec4 vPosition;
attribute vec3 vNormal;

varying vec3 fN; varying vec3 fE; varying vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;

void main()
{
    vec4 N = vec4(vNormal, 0.0f);
    fN =  (ModelView * N).xyz;
    fE = -(ModelView * vPosition).xyz;
    fL =  (LightPosition).xyz;

    if( LightPosition.w != 0.0 ) {
        fL = LightPosition.xyz - vPosition.xyz;
    }

    gl_Position = Projection * ModelView * vPosition;
}
```

Lighting

```
// Fragment program

varying vec3 fN; varying vec3 fL; varying vec3 fE;

uniform vec4  AmbientProduct, DiffuseProduct, SpecularProduct;
uniform float Shininess;

void main()
{
    vec3 N = normalize(fN); // Normalize the input lighting vectors
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);
    vec3 R = normalize(reflect(L, N));

    vec4 ambient = AmbientProduct;

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

    float Ks = pow(max(dot(R, E), 0.0), Shininess);
    vec4 specular = Ks * SpecularProduct;

    // discard the specular highlight if the light's behind the vertex
    if( dot(L, N) < 0.0 ) {
       specular = vec4(0.0, 0.0, 0.0, 1.0);
    }

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

Texturing

Steps for texturing:

- Make your model have texture coordinates;
- Load a texture into memory; pass it to OpenGL;
- Modify the vertex / fragment programs to use it.
    - We will multiply the diffuse color by the texture.

Texturing

- A quad with vertex positions and texture coordinates:

- Refer to template4 for details on texture coordinates, loading textures and passing them to the shader.

- Next two slides show the changes to the shaders.

Texturing

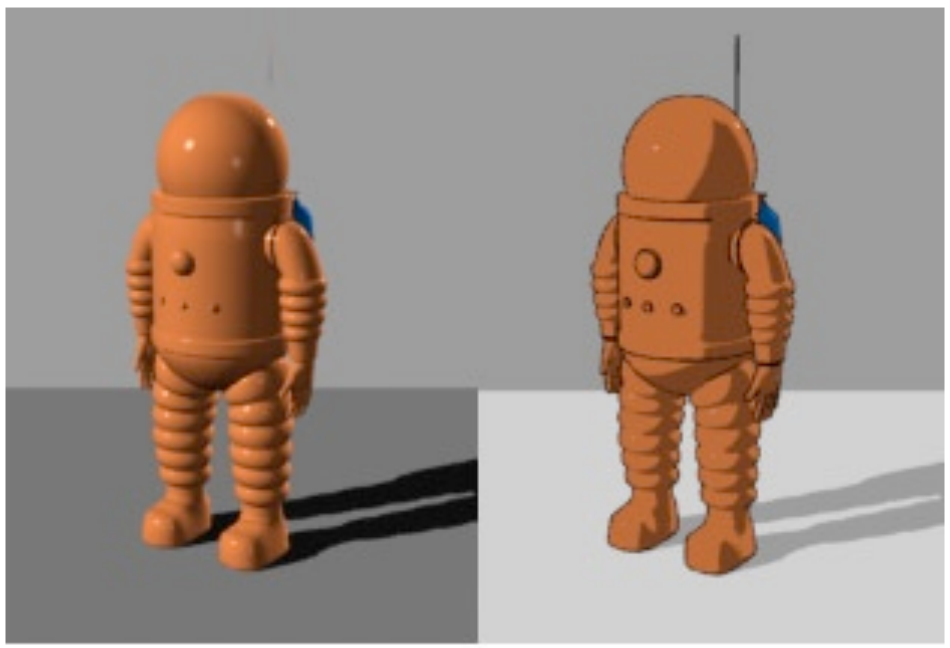
```
// Vertex program

attribute vec4 vPosition;
attribute vec3 vNormal;
attribute vec2 vTexCoords;

varying vec3 fN; varying vec3 fE; varying vec3 fL;

varying vec2 texCoord;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;

void main()
{
    vec4 N = vec4(vNormal, 0.0f);
    fN =  (ModelView * N).xyz;
    fE = -(ModelView * vPosition).xyz;
    fL =  (LightPosition).xyz;

    if( LightPosition.w != 0.0 ) {
        fL = LightPosition.xyz - vPosition.xyz;
    }

    gl_Position = Projection * ModelView * vPosition;
    texCoord = vTexCoords;
}
```

Texturing

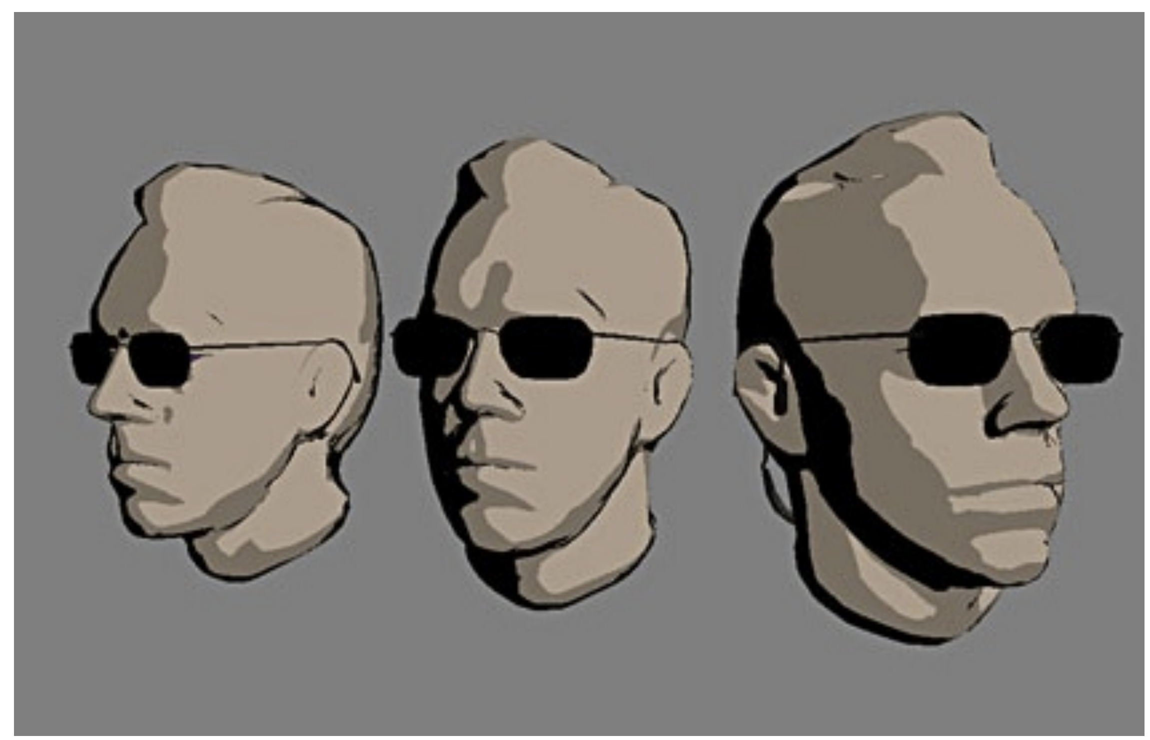
```
// Fragment program
varying vec3 fN; varying vec3 fL; varying vec3 fE;
varying vec2 texCoord;

uniform vec4  AmbientProduct, DiffuseProduct, SpecularProduct;
uniform float Shininess;
uniform sampler2D Tex;

void main()
{
    vec3 N = normalize(fN); // Normalize the input lighting vectors
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);
    vec3 R = normalize(reflect(L, N));

    vec4 ambient = AmbientProduct;

    float Kd = max(dot(L, N), 0.0);
    vec4 diffuse = Kd * DiffuseProduct;
    diffuse *= texture2D(Tex, texCoord);

    float Ks = pow(max(dot(R, E), 0.0), Shininess);
    vec4 specular = Ks * SpecularProduct;

    // discard the specular highlight if the light's behind the vertex
    if( dot(L, N) < 0.0 ) {
       specular = vec4(0.0, 0.0, 0.0, 1.0);
    }

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

Cartoon Shading

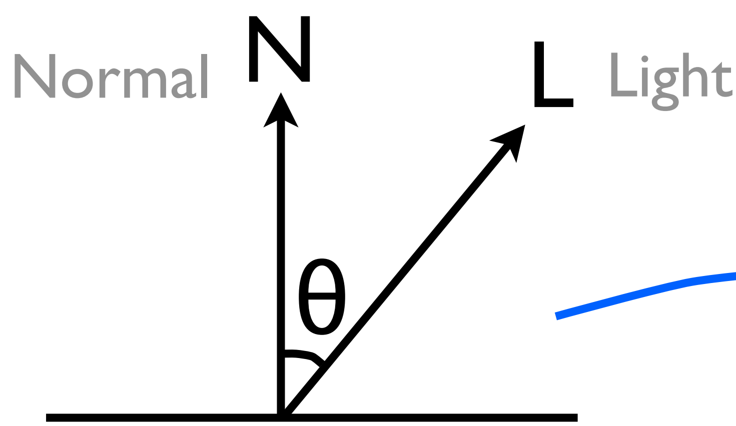
- Also called Cel-Shading



plastic shader toon shader



Cartoon Shading



$$K_d = \cos\theta = N \cdot L$$

Instead of using K_d for illumination, use it as an index into a one-dimensional texture.