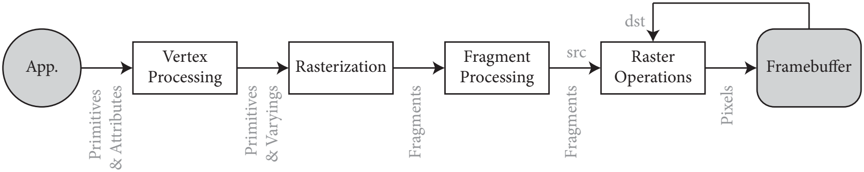
CSC 4356 / ME 4573 Interactive Computer Graphics

The Programmable Pipeline

The 3D Graphics Pipeline



We now have the opportunity to control what happens during Vertex Processing and Fragment Processing.

GPU functions are called *shaders*. We have two programmable pipeline stages* so we have two types of shaders:

- *Vertex* shaders
- *Fragment* shaders

One vertex shader and one fragment shader are *linked* to form one *program*.

* For now. There are two more optional, advanced stages.

The OpenGL Shading Language

Shaders are written in GLSL.

- The syntax is essentially C.
- Matrix and vector types are added.
- GPU-specific functions are provided.
- Storage qualifiers indicate pipeline relationships.

The OpenGL Shading Language

The official GLSL language specification may be found at the OpenGL web site.

In the interest of near-universal hardware compatibility, this document uses **GLSL version 1.20**.

Scalar Types

Familiar C-like data types are supported:

`float` `int` `bool`

But automatic type promotion is *not*. If you type `2` when you mean `2.0`, you will get a compilation error.

Vector Types

```
vec2 p;                             // Declaration
vec3 a;                             // Declaration
vec3 b = vec3(0.0, 0.5, 1.0);      // Initialization
vec4 v = vec4(0.0);                 // Initialization

a.x   = 2.0;                        // Element access
a.yz  = b.xy;                       // Subset access
a     = a + b;                      // Arithmetic
a.xyz = b.zyx                       // Swizzling

float k = dot(a, b);                // Dot product
```

Matrix types

```
mat3 N;
mat4 M, A, B;                          // Declaration
vec4 u, v;

M[3][3] = 1.0;                         // Element access
M[0]    = u;                           // Column assignment
M       = A * B;                       // Multiplication
u       = M * v;                       // Vector transformation
```

Structural Types

Familiar C-like structures and arrays:

```
struct light
{
    vec3 color;
    vec3 position;
};

struct light my_lights[4];
```

GLSL Storage Qualifiers

Variables declared at file scope are used for communication between stages. They may be…

Constant ……	Never changing
Uniform ……	Set per-frame
Attribute ……	Set per-vertex
Varying ……	Set per-fragment

Uniforms

Uniform variables are set by the application, and remain constant through the entire rendering of an object. Built-in uniforms include:

```
mat4 gl_ProjectionMatrix;
mat4 gl_ModelViewMatrix;
vec4 gl_FrontMaterial.diffuse;
```

(etc)

Attributes

Attributes are the per-vertex values supplied to the GPU via vertex buffer objects. They are the *input* to the vertex processer.

```
vec4 gl_Vertex;
vec3 gl_Normal;
vec4 gl_MultiTexCoord0;
```

(etc)

Varyings

Varying variables are the values interpolated during rasterization. They are the *output* from the vertex processor and the *input* to the fragment processor.

```
vec4 gl_Position
vec4 gl_FrontColor
vec4 gl_TexCoord[0]
```

(etc)

Simplest Shader Demo

Simplest Vertex Shader

```
void main()
{
    gl_FrontColor = gl_FrontMaterial.diffuse;
    gl_Position   = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Copy the current *diffuse material* built-in uniform to the *front color* built-in varying. Transform the *vertex position* attribute using the *MVP* uniform, giving the *position* built-in varying.

Simplest Fragment Shader

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

Copy the current *color* built-in varying to the *fragment color* built-in "special" variable.

Note, the color varyings don't match.

In the Vertex Shader:

- `gl_FrontColor`
- `gl_BackColor`

In the Fragment Shader:

- `gl_Color`

The front or back color varying is selected depending on the current face orientation.

Note, the code listing changed color

For added clarity in these course notes, background color indicates the *language* of listed code…

```
// This is application code in C.
// It runs on the CPU.
```

```
// This is vertex shader code in GLSL.
// It runs on the vertex processor of the GPU.
```

```
// This is fragment shader code in GLSL.
// It runs on the fragment processor of the GPU.
```

OpenGL Shader API

The are several things an application must do to use the programmable pipeline:

1. Loading shader source
2. Compiling shaders
3. Linking the program
4. Binding the program

1. Loading Shader Source

Each GLSL shader file must be loaded into memory as a C-style null-terminated ASCII string.

The application does not know the size of the incoming source text, so the file must be measured and storage for the contents dynamically allocated…

```c
char *load(const char *name)
{
    FILE *fp = 0;
    void  *p = 0;
    size_t n = 0;

    if ((fp = fopen(name, "rb")))                       // Open the file.
    {
        if (fseek(fp, 0, SEEK_END) == 0)                // Seek the end.
            if ((n = (size_t) ftell(fp)))               // Tell the length.
                if (fseek(fp, 0, SEEK_SET) == 0)        // Seek the beginning.
                    if ((p = calloc(n + 1, 1)))         // Allocate a buffer.
                        fread(p, 1, n, fp);             // Read the data.

        fclose(fp);                                     // Close the file.
    }
    return p;
}
```

2. Compiling a Vertex Shader

As with all OpenGL objects, a compiled vertex shader object is represented by a `GLuint`.

```
GLuint vert_shader = glCreateShader(GL_VERTEX_SHADER);

GLchar *vert_text = load(vert_filename);

glShaderSource (vert_shader, 1, (const GLchar **) &vert_text, 0);
glCompileShader(vert_shader);

free(vert_text);
```

2. Compiling a Fragment Shader

A fragment shader is compiled in the same fashion, but with a different create parameter.

```
GLuint frag_shader = glCreateShader(GL_FRAGMENT_SHADER);

GLchar *frag_text = load(frag_filename);

glShaderSource (frag_shader, 1, (const GLchar **) &frag_text, 0);
glCompileShader(frag_shader);

free(frag_text);
```

2.1. Check for compilation errors

```
GLchar  *p;
GLint s, n;

glGetShaderiv(shader, GL_COMPILE_STATUS,  &s);
glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &n);

if ((s == 0) && (p = (GLchar *) calloc(n + 1, 1)))
{
    glGetShaderInfoLog(shader, n, NULL, p);

    fprintf(stderr, "OpenGL Shader Error:\n%s", p);
    free(p);
}
```

This is optional, but if you don’t do it, you’ll regret it.

3. Linking a Program

One vertex shader and one fragment shader are linked into a program.

```
GLuint program = glCreateProgram();

glAttachShader(program, vert_shader);
glAttachShader(program, frag_shader);

glLinkProgram(program);
```

3.1 Check for linking errors

```
GLchar  *p;
GLint s, n;

glGetProgramiv(program, GL_LINK_STATUS,     &s);
glGetProgramiv(program, GL_INFO_LOG_LENGTH, &n);

if ((s == 0) && (p = (GLchar *) calloc(n + 1, 1)))
{
    glGetProgramInfoLog(program, n, NULL, p);

    fprintf(stderr, "OpenGL Program Error:\n%s", p);
    free(p);
}
```

I dare you to not do this. Do you feel lucky?

4. Binding a Program

As with all other OpenGL objects, a program must be bound to be used. Breaking with tradition, it's called "use."

- `glUseProgram(program);`

With a valid program in use, the programmable pipeline is now active.

4.1. Unbinding a Program

If program zero is bound, the fixed function pipeline falls back into place.

- `glUseProgram(0);`

Uniform API

Uniforms allow the application to influence the execution of vertex and fragment shaders.

```
GLuint uniform_time = glGetUniformLocation(program, "time");

glUniform1f(uniform_time, get_current_time());
```

Access comes through yet another `GLuint` object called a "uniform location."

Uniform API

Different uniform functions set values for different uniform types.

```
glUniform1f(GLuint location, GLfloat x);
glUniform2f(GLuint location, GLfloat x, GLfloat y);
glUniform3f(GLuint location, GLfloat x, GLfloat y, GLfloat z);
glUniform4f(GLuint location, GLfloat x, GLfloat y, GLfloat z, GLfloat w);

glUniformMatrix4fv(GLuint location, GLsizei count,
                   GLboolean transpose, const GLfloat *value)
```

A Uniform-using Vertex Shader

```
uniform float time;

void main()
{
    vec4 P = gl_Vertex;

    P.y += sin(P.x + time * 3.0) + sin(P.z + time * 2.0);

    gl_FrontColor = gl_FrontMaterial.diffuse;          // Vertex color
    gl_Position   = gl_ModelViewProjectionMatrix * P; // Vertex position
}
```

Per-vertex Wave Demo

Whither the Fixed Function?

When the programmable pipeline is active, the fixed function pipeline is not. It is all-or-nothing.

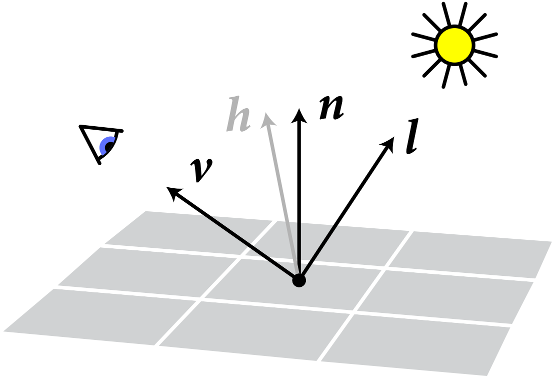
If we wish to continue to use the familiar illumination model, we must implement it ourselves.

Fortunately, it's easy, and we can immediately step beyond it to something more powerful.

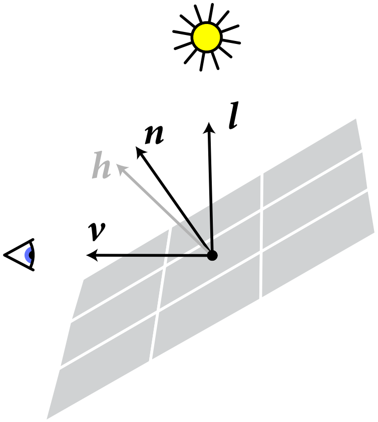
Vertex Lighting

Recall the inputs to the lighting equation:

1. The unit vector $\boldsymbol{n}$ normal to the surface.
2. The unit vector $\boldsymbol{l}$ toward the light.
3. The unit vector $\boldsymbol{v}$ toward the viewer.
4. The diffuse $\boldsymbol{m}_d$ and specular $\boldsymbol{m}_s$ material properties.



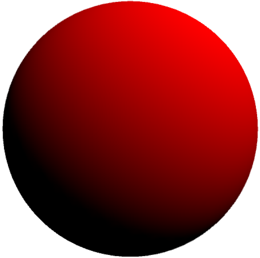
Remember this?



Think of it now in *eye* space.

Diffuse Lighting

Diffuse lighting models matte surfaces.



$$\boldsymbol{c}_d = \boldsymbol{m}_d\,(\boldsymbol{n} \cdot \boldsymbol{l})$$

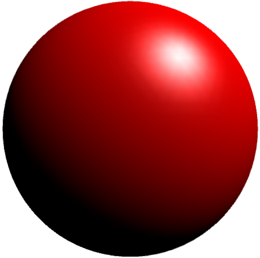
Diffuse Per-Vertex GLSL

```
void main()
{
    vec3  V   = vec3(0.0, 0.0, 1.0);                     // View vector
    vec3  L   = normalize(gl_LightSource[0].position.xyz); // Light vector
    vec3  N   = normalize(gl_NormalMatrix * gl_Normal);     // Normal vector

    vec4  D   = gl_FrontMaterial.diffuse;                 // Diffuse color

    float kd  = max(dot(N, L), 0.0);                      // Diffuse intensity

    vec3  rgb = D.rgb * kd;                               // RGB channels
    float a   = D.a;                                      // Alpha channel

    gl_FrontColor = vec4(rgb, a);                         // Vertex color
    gl_Position   = ftransform();                         // Vertex position
}
```

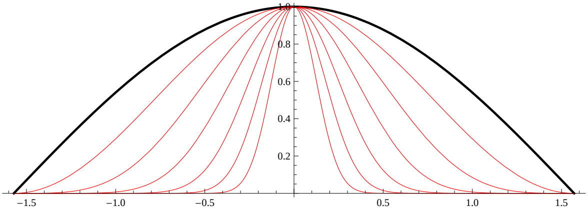
Per-vertex Diffuse Lighting Demo

Diffuse + Specular

Most materials exhibit both diffuse *and* specular properties, so one usually uses both at the same time.



$$\boldsymbol{c} = \boldsymbol{m}_d\,(\boldsymbol{n}\cdot\boldsymbol{l}) + \boldsymbol{m}_s\,(\boldsymbol{n}\cdot\boldsymbol{h})^{\alpha}$$



Diffuse + Specular Per-Vertex Lighting

```
void main()
{
    vec3  V   = vec3(0.0, 0.0, 1.0);                       // View vector
    vec3  L   = normalize(gl_LightSource[0].position.xyz); // Light vector
    vec3  N   = normalize(gl_NormalMatrix * gl_Normal);    // Normal vector
    vec3  H   = normalize(L + V);                          // Half-angle vector

    vec4  D   = gl_FrontMaterial.diffuse;                  // Diffuse color
    vec4  S   = gl_FrontMaterial.specular;                 // Specular color
    float n   = gl_FrontMaterial.shininess;                // Specular exponent

    float kd  =     max(dot(N, L), 0.0);                   // Diffuse intensity
    float ks  = pow(max(dot(N, H), 0.0), n);               // Specular intensity

    vec3  rgb = D.rgb * kd + S.rgb * ks;                   // RGB channels
    float a   = D.a;                                       // Alpha channel

    gl_FrontColor = vec4(rgb, a);                          // Vertex color
    gl_Position   = ftransform();                          // Vertex position
}
```

Per-vertex Diffuse + Specular Lighting Demo

We're *still* using that trivial fragment shader.

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

Where do we go from here? Obviously, toward better fragment shaders.

User-defined varyings

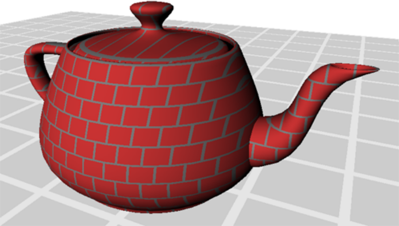
```
varying vec3 var_L;
varying vec3 var_N;

void main()
{
    var_L = gl_LightSource[0].position.xyz;  // Light vector
    var_N = gl_NormalMatrix * gl_Normal;     // Normal vector

    gl_Position = ftransform();              // Vertex position
}
```

Instead of using L and N in the vertex shader, we'll let them vary, and compute the lighting in the fragment shader…

```glsl
varying vec3 var_L;
varying vec3 var_N;

void main()
{
    vec3  V   = vec3(0.0, 0.0, 1.0);                             // View vector
    vec3  L   = normalize(var_L);                                // Light vector
    vec3  N   = normalize(var_N);                                // Normal vector
    vec3  H   = normalize(L + V);                                // Half-angle vector

    vec4  D   = gl_FrontMaterial.diffuse;                        // Diffuse color
    vec4  S   = gl_FrontMaterial.specular;                       // Specular color
    float n   = gl_FrontMaterial.shininess;                      // Specular exponent

    float kd  =     max(dot(N, L), 0.0);                         // Diffuse intensity
    float ks  = pow(max(dot(N, H), 0.0), n);                     // Specular intensity

    vec3  rgb = D.rgb * kd + S.rgb * ks;                         // RGB channels
    float a   = D.a;                                             // Alpha channel

    gl_FragColor = vec4(rgb, a);                                 // Fragment color
}
```

Per-fragment Diffuse + Specular Lighting Demo

Procedural Materials



Given programmable shading, it becomes possible to control material properties on a per-pixel basis.

Brick Vertex Shader

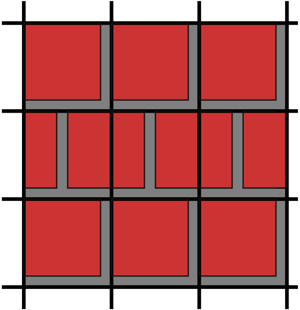
```
varying vec3 var_L;
varying vec3 var_N;
varying vec2 var_P;

void main()
{
    var_L = gl_LightSource[0].position.xyz;  // Light vector
    var_N = gl_NormalMatrix * gl_Normal;     // Normal vector
    var_P = gl_Vertex.xy;                    // Position vector

    gl_Position = ftransform();              // Vertex position
}
```

Brick shading uses an off-the-shelf vertex shader, but with the *object-space* position varying.

Brick Fragment Shader

```
varying vec3 var_L;
varying vec3 var_N;
varying vec2 var_P;

void main()
{
    vec3  L   = normalize(var_L);                         // Light vector
    vec3  N   = normalize(var_N);                         // Normal vector

    float kd  = max(dot(N, L), 0.0);                      // Diffuse intensity

    gl_FragColor = vec4(material_color(var_P) * kd, 1.0);   // Fragment color
}
```

The fragment shader is also standard, though the color is a function of the object-space position.



```
uniform vec3 mortar_color;
uniform vec3 brick_color;
uniform vec2 brick_size;
uniform vec2 brick_frac;

vec3 material_color(vec2 position)
{
    vec2 p = position / brick_size;

    if (fract(p.y * 0.5) > 0.5)
        p.x += 0.5;

    p = fract(p);

    vec2 b = step(p, brick_frac);

    return mix(mortar_color, brick_color, b.x * b.y);
}
```

And here's the definition of that function, with its user-defined uniforms.

Brick Demo

Programmable Texture Mapping

```
varying vec3 var_L;
varying vec3 var_N;

void main()
{
    var_L = gl_LightSource[0].position.xyz;  // Light vector
    var_N = gl_NormalMatrix * gl_Normal;     // Normal vector

    gl_TexCoord[0] = gl_MultiTexCoord0;      // Texture coordinate
    gl_Position    = ftransform();           // Vertex position
}
```

Note here the use of the `gl_MultiTexCoord0` built-in attribute and the `gl_TexCoord` built-in varying.

```
uniform sampler2D diffuse;
varying vec3 var_L;
varying vec3 var_N;

void main()
{
    vec3  V   = vec3(0.0, 0.0, 1.0);                     // View vector
    vec3  L   = normalize(var_L);                        // Light vector
    vec3  N   = normalize(var_N);                        // Normal vector
    vec3  H   = normalize(L + V);                        // Half-angle vector

    vec4  D   = texture2D(diffuse, gl_TexCoord[0].xy);  // Diffuse color
    vec4  S   = gl_FrontMaterial.specular;              // Specular color
    float n   = gl_FrontMaterial.shininess;             // Specular exponent

    float kd  =     max(dot(N, L), 0.0);                // Diffuse intensity
    float ks  = pow(max(dot(N, H), 0.0), n);            // Specular intensity

    vec3  rgb = D.rgb * kd + S.rgb * ks;                // RGB channels
    float a   = D.a;                                    // Alpha channel

    gl_FragColor = vec4(rgb, a);                        // Fragment color
}
```

Sampler Uniforms

```
GLint location = glGetUniformLocation(program, "diffuse");

glUniform1i(location, 0);
```

A *sampler* is a uniform providing access to a texture image. It’s value is an integer.

An *integer*?

Texture Image Units

There are anywhere from 4 to 64 TIUs, depending on the hardware.

Texture Image Units

The sampler uniform's value is a *texture image unit* index.

```
GLuint my_diffuse_texture;
GLuint my_specular_texture;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, my_diffuse_texture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, my_specular_texture);
glActiveTexture(GL_TEXTURE0);

glUniform1i(my_diffuse_sampler,  0);
glUniform1i(my_specular_sampler, 1);
```

Textured Fragment Lighting Demo

Programmable Texturing

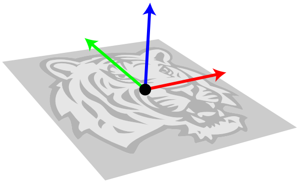
Consider for a moment a 2D texture mapped onto a surface. It has two axes $\boldsymbol{s}$ and $\boldsymbol{t}$, which are (usually) perpendicular to one another.

Normal $\boldsymbol{n}$ is perpendicular to both $\boldsymbol{s}$ and $\boldsymbol{t}$.

Together, $\boldsymbol{s}$, $\boldsymbol{t}$, and $\boldsymbol{n}$, define a *unique* 3D coordinate system for *every pixel* of a surface…

Tangent Space

A 3D vector basis giving the ultimate coordinate system in which to perform per-pixel illumination.



- ***n*** is still called the *normal*.
- ***s*** is known here as the *tangent*.
- ***t*** is called the *bi-tangent*.*

* Many people in computer graphics call ***t*** the "bi-normal." They are wrong.

At this point the lecture goes off the rails and turns into a parade of interactive demos and live code examples.