

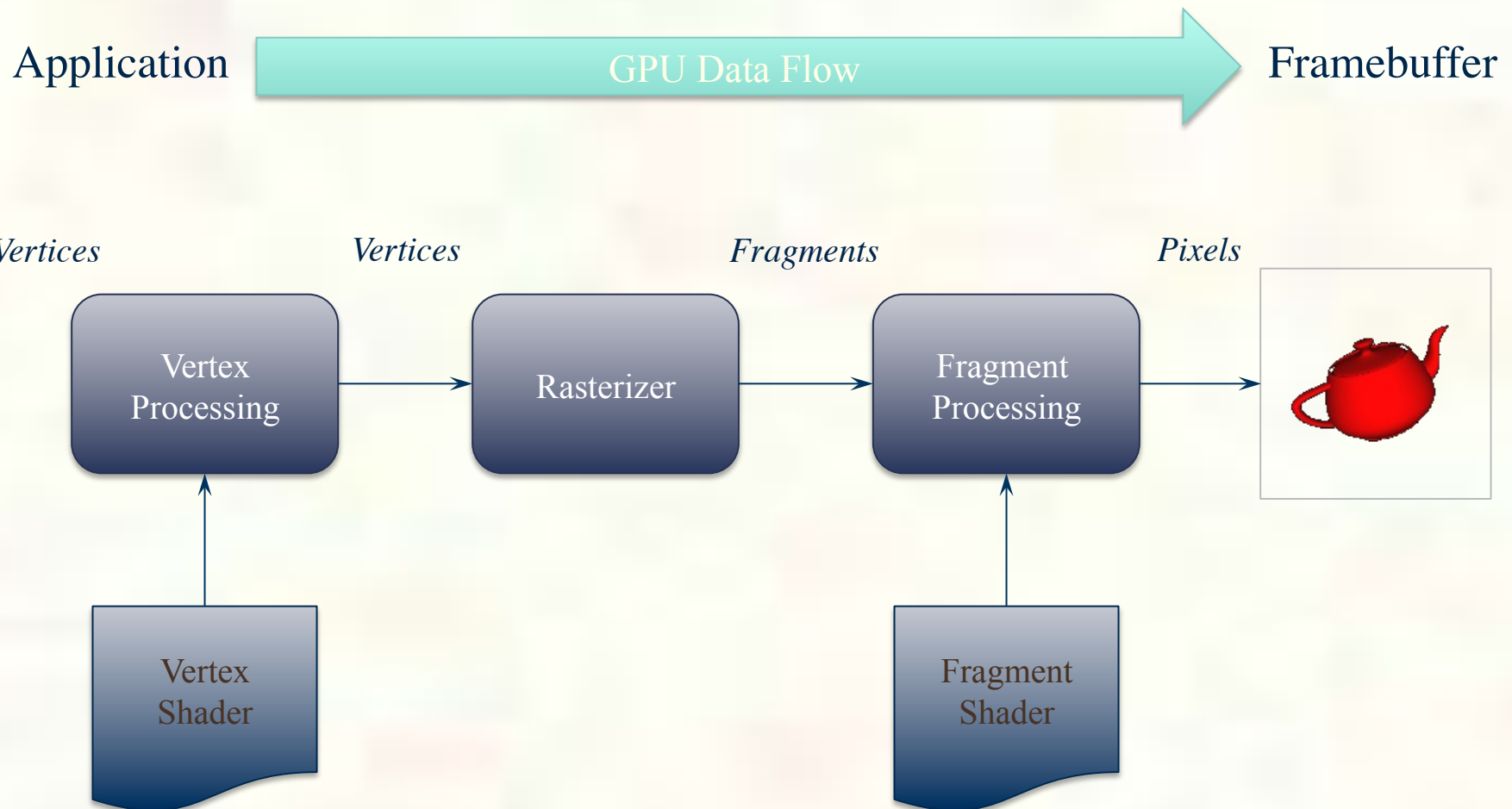


OpenGL Application Development





A Simplified Pipeline Model





OpenGL Programming in a Nutshell

- Modern OpenGL programs essentially do the following steps:
 1. Create shader programs
 2. Create buffer objects and load data into them
 3. “Connect” data locations with shader variables
 4. Render



Application Framework Requirements

- OpenGL applications need a place to render into
 - usually an on-screen window
- Need to communicate with native windowing system
- Each windowing system interface is different
- We use GLUT (more specifically, freeglut)
 - simple, open-source library that works everywhere
 - handles all windowing operations:
 - opening windows
 - input processing



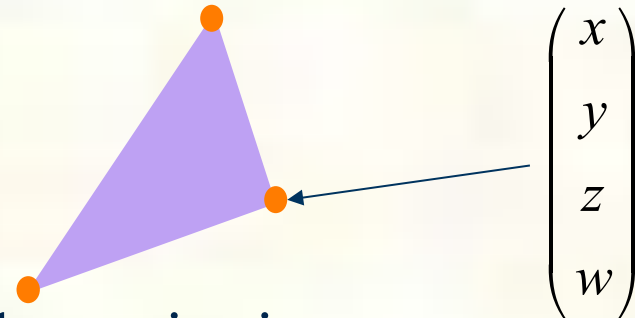
Simplifying Working with OpenGL

- Operating systems deal with library functions differently
 - compiler linkage and runtime libraries may expose different functions
- Additionally, OpenGL has many versions and profiles which expose different sets of functions
 - managing function access is cumbersome, and window-system dependent
- We use another open-source library, GLEW, to hide those details



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)
- VBOs must be stored in *vertex array objects* (VAOs)



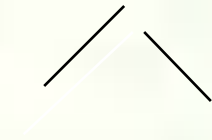


OpenGL's Geometric Primitives

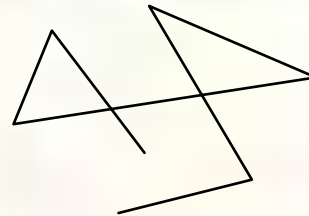
- All primitives are specified by vertices



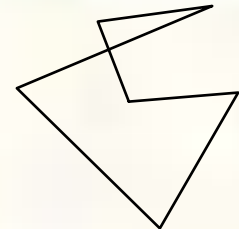
GL_POINTS



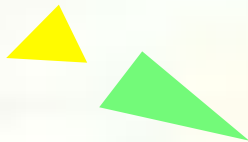
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN



A First Program





Rendering a Cube

- We'll render a cube with colors at each vertex
- Our example demonstrates:
 - initializing vertex data
 - organizing data for rendering
 - simple object modeling
 - building up 3D objects from geometric primitives
 - building geometric primitives from vertices



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)
`const int NumVertices = 36;`
- To simplify communicating with GLSL, we'll use a `vec4` class (implemented in C++) similar to GLSL's `vec4` type
 - we'll also typedef it to add logical meaning
`typedef vec4 point4;`
`typedef vec4 color4;`



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

```
point4  points[NumVertices];  
color4  colors[NumVertices];
```



Cube Data

```
// Vertices of a unit cube centered at origin, sides aligned  
with axes
```

```
point4 vertex_positions[8] = {  
    point4( -0.5, -0.5,  0.5, 1.0 ),  
    point4( -0.5,  0.5,  0.5, 1.0 ),  
    point4(  0.5,  0.5,  0.5, 1.0 ),  
    point4(  0.5, -0.5,  0.5, 1.0 ),  
    point4( -0.5, -0.5, -0.5, 1.0 ),  
    point4( -0.5,  0.5, -0.5, 1.0 ),  
    point4(  0.5,  0.5, -0.5, 1.0 ),  
    point4(  0.5, -0.5, -0.5, 1.0 )  
};
```



Cube Data

```
// RGBA colors
color4 vertex_colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ), // black
    color4( 1.0, 0.0, 0.0, 1.0 ), // red
    color4( 1.0, 1.0, 0.0, 1.0 ), // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ), // green
    color4( 0.0, 0.0, 1.0, 1.0 ), // blue
    color4( 1.0, 0.0, 1.0, 1.0 ), // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ), // white
    color4( 0.0, 1.0, 1.0, 1.0 ) // cyan
};
```



Generating a Cube Face from Vertices

```
// quad() generates two triangles for each face and assigns
// colors to the vertices
int Index = 0; // global variable indexing into VBO arrays

void quad(int a, int b, int c, int d) {
    colors[Index] = vertex_colors[a]; points[Index] =
    vertex_positions[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] =
    vertex_positions[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] =
    vertex_positions[c]; Index++;
    colors[Index] = vertex_colors[a]; points[Index] =
    vertex_positions[a]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] =
    vertex_positions[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] =
    vertex_positions[d]; Index++;
}
```



Generating the Cube from Faces

```
// generate 12 triangles: 36 vertices and 36
  colors
void
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 );
}
```



Vertex Array Objects (VAOs)

- VAOs store the data of a geometric object
- Steps in using a VAO
 - generate VAO names by calling `glGenVertexArrays()`
 - bind a specific VAO for initialization by calling `glBindVertexArray()`
 - update VBOs associated with this VAO
 - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
 - previously, you might have needed to make many calls to make all the data current



VAOs in Code

```
// Create a vertex array object  
GLuint vao;  
glGenVertexArrays(1, &vao);  
glBindVertexArray(vao);
```



Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
 - generate VBO names by calling `glGenBuffers()`
 - bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, ...)`
 - load data into VBO using `glBufferData(GL_ARRAY_BUFFER, ...)`
 - bind VAO for use in rendering `glBindVertexArray()`



VBOs in Code

```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
             sizeof(colors), NULL,
             GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0,
               sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
               sizeof(colors), colors);
```



Connecting Vertex Shaders with Geometry

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
 - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`



Vertex Array Code

```
// set up vertex arrays (after shaders are loaded)
GLuint vPosition = glGetAttribLocation(program,
    "vPosition");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(0));
GLuint vColor = glGetAttribLocation(program,
    "vColor");
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(sizeof(points)));
```



Drawing Geometric Primitives

- For contiguous groups of vertices
`glDrawArrays(GL_TRIANGLES, 0, NumVertices);`
- Usually invoked in display callback
- Initiates vertex shader

Shaders and GLSL



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
- Operators overloaded for matrix and vector operations

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

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



Components and Swizzling

For vectors can use [], xyzw, rgba or stpq

Example:

```
vec3 v;
```

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

Swizzling:

```
vec3 a, b;
```

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



Qualifiers

- **in, out**
 - Copy vertex attributes and other variables to/from shaders
 - `in vec2 tex_coord;`
 - `out vec4 color;`
- **Uniform: variable from application**
 - `uniform float time;`
 - `uniform vec4 rotation;`



Flow Control

- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for



Functions

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



Built-in Variables

- **gl_Position**: output position from vertex shader
- **gl_FragColor**: output color from fragment shader
 - Only for ES, WebGL and older versions of GLSL
 - Present version use an out variable



Simple Vertex Shader for Cube

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

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



The Simplest Fragment Shader

```
in vec4 color;  
out vec4 FragColor;  
  
void main() {  
    FragColor = color;  
}
```




Getting Shaders into OpenGL

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

Create Program

`glCreateProgram()`

Create Shader

`glCreateShader()`

Load Shader Source

`glShaderSource()`

Compile Shader

`glCompileShader()`

Attach Shader to Program

`glAttachShader()`

Link Program

`glLinkProgram()`

Use Program

`glUseProgram()`

These steps need to be repeated for each type of shader in the shader 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 =  
    glGetUniformLocation(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);
```



Finishing the Cube Program

```
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE |
        GLUT_DEPTH);
    glutInitWindowSize(512, 512);
    glutCreateWindow("Color Cube");
    glewInit();
    init();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```



Cube Program GLUT Callbacks

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);  
    glutSwapBuffers();  
}
```

```
void keyboard(unsigned char key, int x, int y) {  
    switch( key ) {  
        case 033: case 'q': case 'Q':  
            exit( EXIT_SUCCESS );  
            break;  
    }  
}
```



Vertex Shader Examples

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



Vertex Shader for Cube Rotation

```
in vec4 vPosition;
in vec4 vColor;
out 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 Cube Rotation

// 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 Cube Rotation

```
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

```
// compute angles using mouse and idle callbacks
GLuint theta; // theta uniform location
vec3 Theta; // Axis angles

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUniform3fv(theta, 1, Theta);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);

    glutSwapBuffers();
}
```

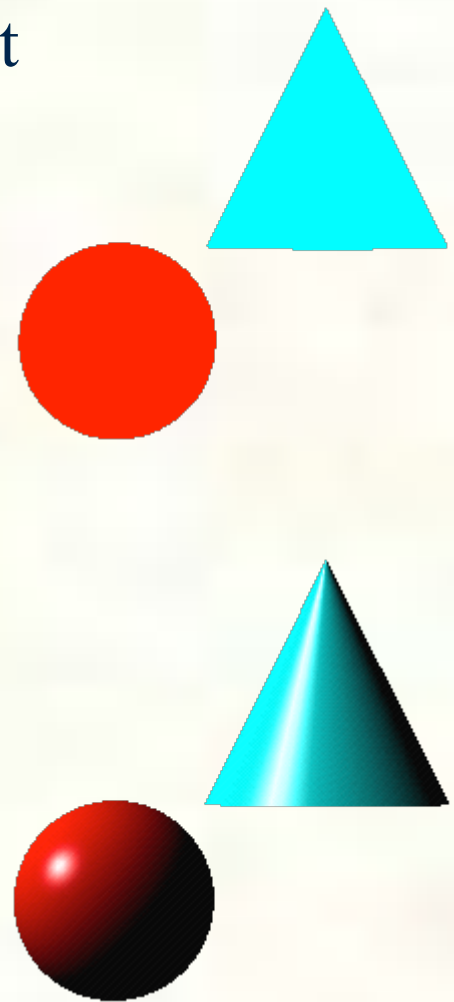
Vertex Lighting





Lighting Principles

- Lighting simulates how objects reflect light
 - material composition of object
 - light's color and position
 - global lighting parameters
- Lighting functions deprecated in 3.1
- Can implement in
 - Application (per vertex)
 - Vertex or fragment shaders





Modified Phong Model

- Computes a color or shade for each vertex using a lighting model (the modified Phong model) that takes into account
 - Diffuse reflections
 - Specular reflections
 - Ambient light
 - Emission
- Vertex shades are interpolated across polygons by the rasterizer



Modified Phong Model

- The model is a balance between simple computation and physical realism
- The model uses
 - Light positions and intensities
 - Surface orientation (normals)
 - Material properties (reflectivity)
 - Viewer location
- Computed for each source and each color component



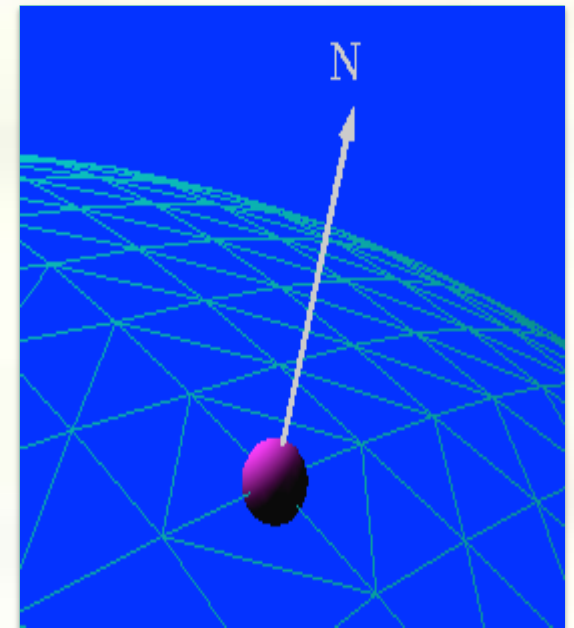
OpenGL Lighting

- Modified Phong lighting model
 - Computed at vertices
- Lighting contributors
 - Surface material properties
 - Light properties
 - Lighting model properties



Surface Normals

- Normals define how a surface reflects light
 - Application usually provides normals as a vertex attribute
 - 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



Adding Lighting to Cube

```
// vertex shader
```

```
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

```
void main() {  
    // Transform vertex position into eye coordinates  
    vec3 pos = (ModelView * vPosition).xyz;  
  
    vec3 L = normalize(LightPosition.xyz - pos);  
    vec3 E = normalize(-pos);  
    vec3 H = normalize(L + E);  
  
    // Transform vertex normal into eye coordinates  
    vec3 N = normalize(ModelView * vec4(vNormal, 0.0)).xyz;
```



Adding Lighting to Cube

```
// 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;
}
```



Shader Examples





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
 - Bump Mapping
 - Environment (Reflection) Maps



Per Fragment Lighting

- Compute lighting using same model as for per vertex lighting but for each fragment
- Normals and other attributes are sent to vertex shader and output to rasterizer
- Rasterizer interpolates and provides inputs for fragment shader



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.
- Heights fields are usually rendered by sampling the function to form a rectangular mesh of triangles or rectangles from the samples $y_{ij} = f(x_i, z_j)$



Displaying a Height Field

- Form a quadrilateral mesh

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

```
vertex[Index++] = vec3((float)i/N, data[i][j], (float)j/N);
```

```
vertex[Index++] = vec3((float)i/N, data[i][j], (float)(j+1)/N);
```

```
vertex[Index++] = vec3((float)(i+1)/N, data[i][j], (float)(j+1)/N);
```

```
vertex[Index++] = vec3((float)(i+1)/N, data[i][j], (float)j/N);
```

- Display each quad using

```
for(i=0;i<NumVertices ;i+=4) glDrawArrays(GL_LINE_LOOP, 4*i, 4);
```



Time Varying Vertex Shader

```
in vec4 vPosition;
in vec4 vColor;

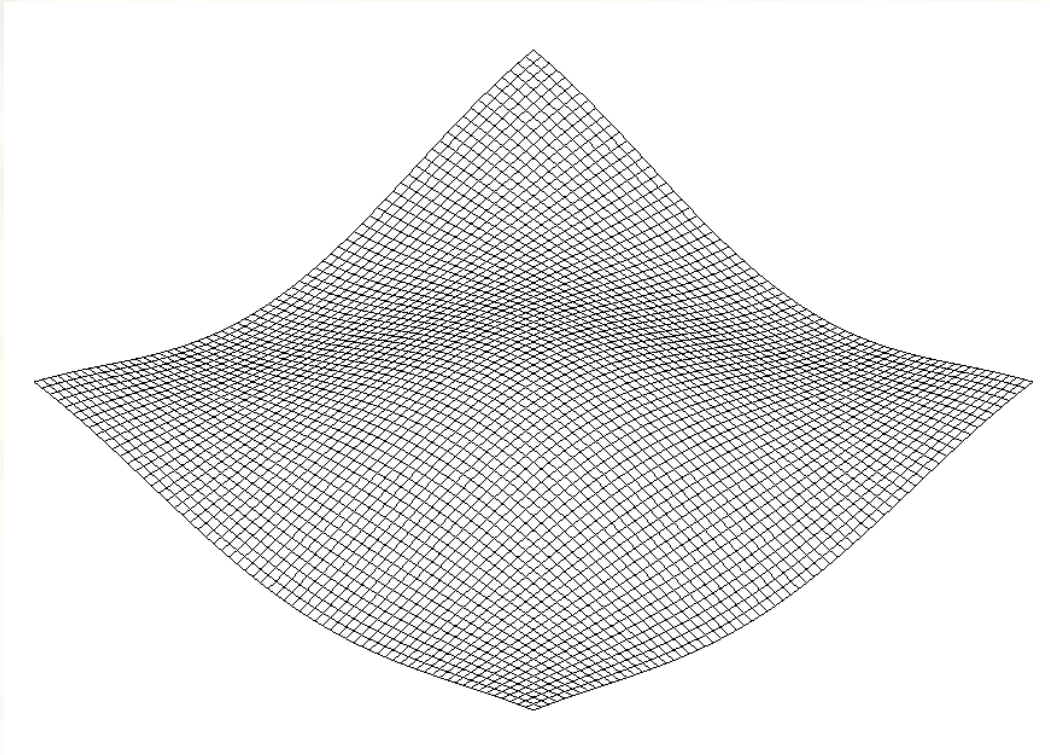
uniform float time; /* in milliseconds */
uniform mat4 ModelView, ProjectionMatrix;

void main() {
    vec4 v = vPosition;
    vec4 t = sin(0.001*time + 5.0*v);
    v.y = 0.1*t.x*t.z;

    gl_Position = ModelViewProjectionMatrix * t;
}
```



Mesh Display





Adding Lighting

- Solid Mesh: create two triangles for each quad
- Display with

```
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
```
- For better looking results, we'll add lighting
- We'll do per-vertex lighting
 - leverage the vertex shader since we'll also use it to vary the mesh in a time-varying way



Mesh Shader

```
uniform float time, shininess;
uniform vec4 vPosition, light_position diffuse_light,
specular_light;
uniform mat4 ModelViewMatrix, ModelViewProjectionMatrix,
NormalMatrix;

void main() {
    vec4 v = vPosition;
    vec4 t = sin(0.001*time + 5.0*v);
    v.y = 0.1*t.x*t.z;

    gl_Position = ModelViewProjectionMatrix * v;

    vec4 diffuse, specular;
    vec4 eyePosition = ModelViewMatrix * vPosition;
    vec4 eyeLightPos = light_position;
```



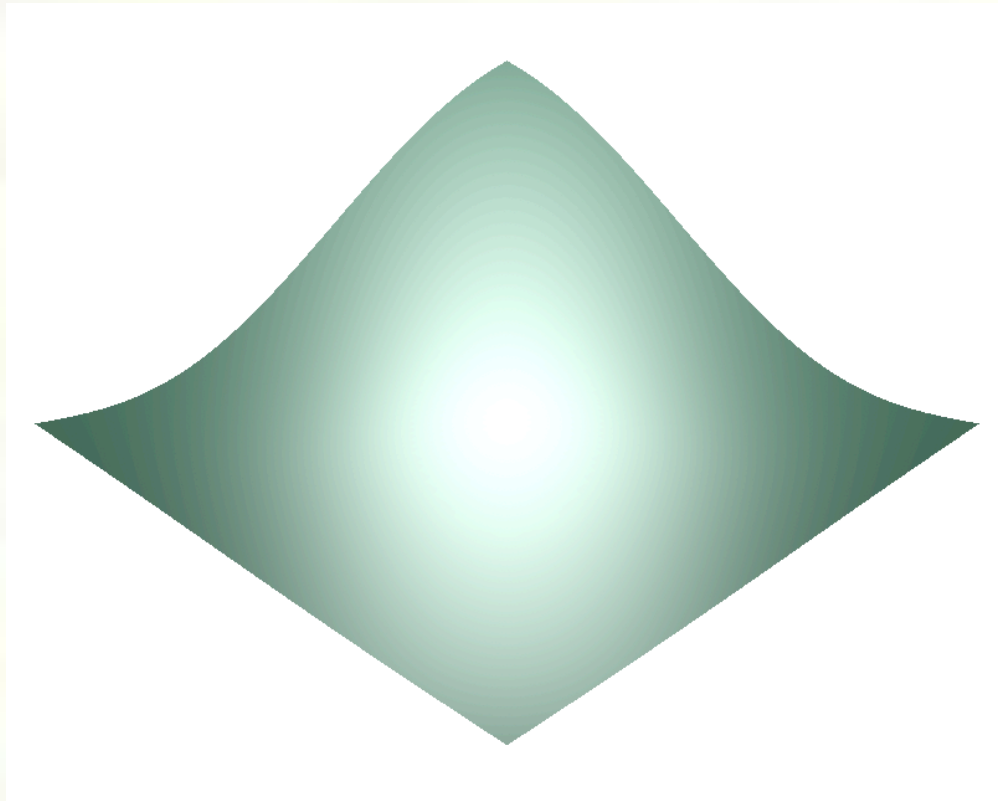
Mesh Shader (cont'd)

```
vec3 N = normalize(NormalMatrix * 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), shininess);
diffuse  = Kd*diffuse_light;
specular = Ks*specular_light;
color    = diffuse + specular;
}
```



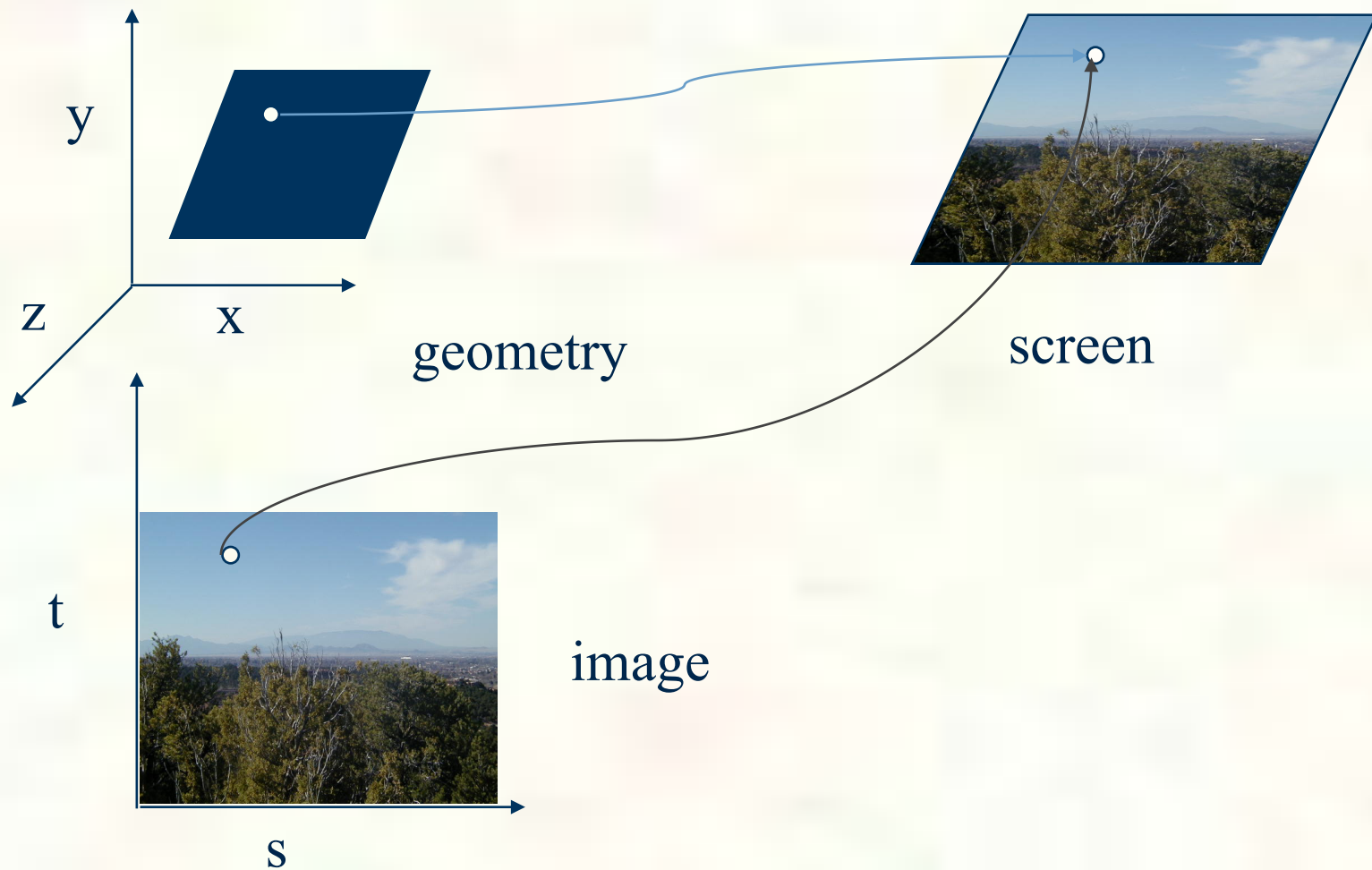

Shaded Mesh



Texture Mapping



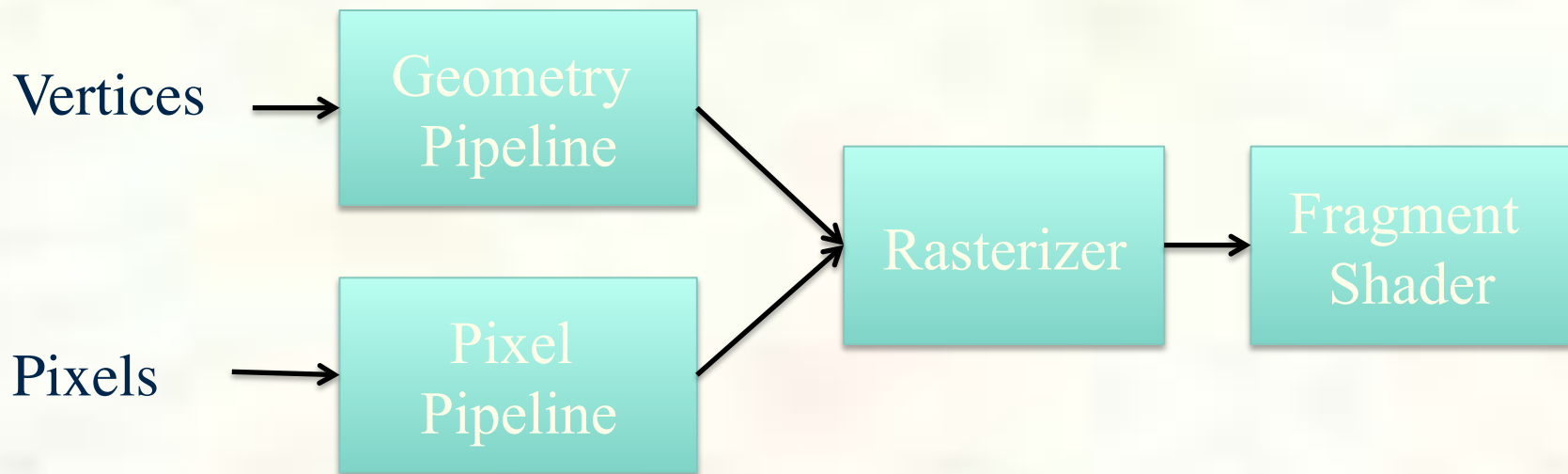
Texture Mapping





Texture Mapping in OpenGL

- 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



Applying Textures

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. set optional perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex



Texture Objects

- Have OpenGL store your images
 - one image per texture object
 - may be shared by several graphics contexts
- Generate texture names

```
glGenTextures(n, *texIds);
```



Texture Objects (cont'd.)

- Create texture objects with texture data and state
 - **glBindTexture(*target*, *id*);**
- Bind textures before using
 - **glBindTexture(*target*, *id*);**



Specifying a Texture Image

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

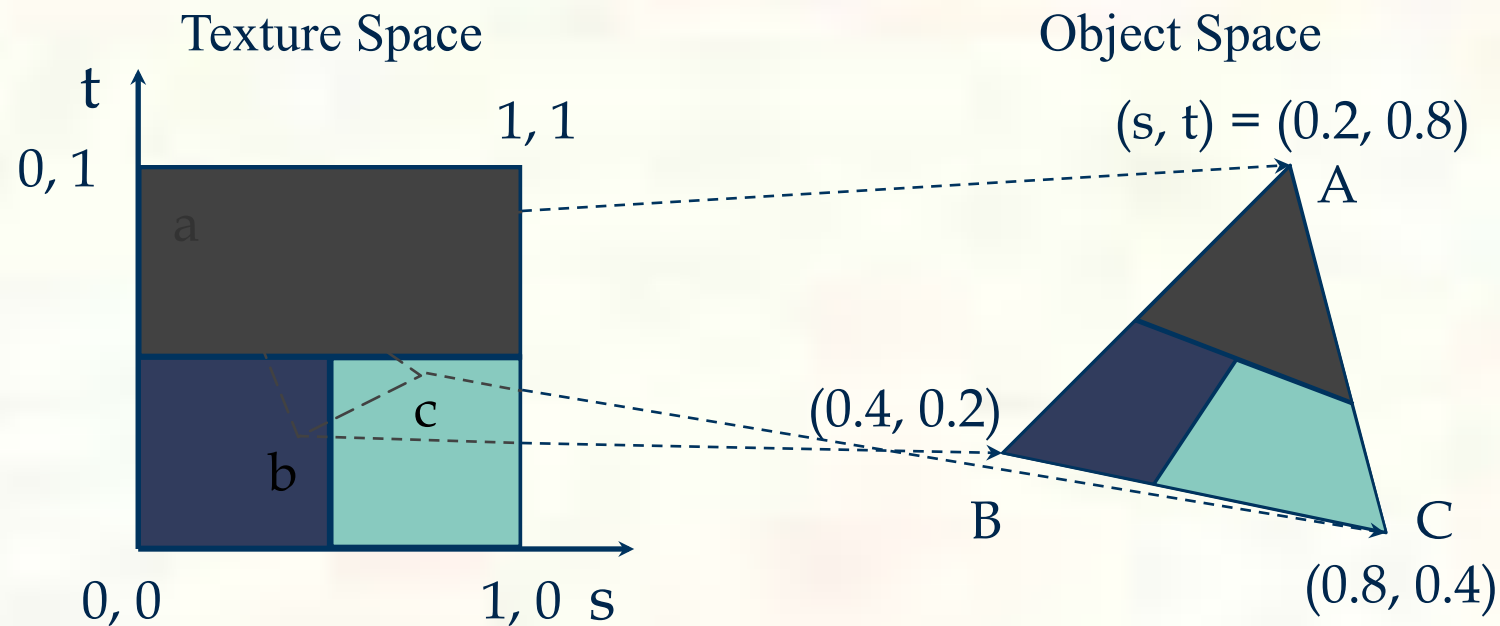
```
glTexImage2D(target, level, components,  
             w, h, border, format, type, *texels);
```

- Texel colors are processed by pixel pipeline
 - pixel scales, biases and lookups can be done



Mapping a Texture

- Based on parametric texture coordinates
- Coordinates need to be specified at each vertex





Applying the Texture in the Shader

```
// Declare the sampler
uniform sampler2D diffuse_mat;
// GLSL 3.30 has overloaded texture();
// Apply the material color
vec3 diffuse = intensity *
    texture2D(diffuse_mat, coord).rgb;
```



Texturing the Cube

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

```
quad(int a, int b, int c, int d) {  
    quad_colors[Index] = vertex_colors[a];  
    points[Index] = vertex_positions[a];  
    tex_coords[Index] = vec2(0.0, 0.0);  
    Index++;  
    ... // rest of vertices  
}
```



Creating a Texture Image

```
// Create a checkerboard pattern
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        GLubyte c;
        c = (((i & 0x8) == 0) ^ ((j & 0x8) == 0)) * 255;
        image[i][j][0] = c;
        image[i][j][1] = c;
        image[i][j][2] = c;
        image2[i][j][0] = c;
        image2[i][j][1] = 0;
        image2[i][j][2] = c;
    }
}
```



Texture Object

```
GLuint textures[1];
glGenTextures(1, textures);

glBindTexture(GL_TEXTURE_2D, textures[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, TextureSize,
             TextureSize, GL_RGB, GL_UNSIGNED_BYTE, image);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glActiveTexture(GL_TEXTURE0);
```



Vertex Shader

```
in vec4 vPosition;  
in vec4 vColor;  
in vec2 vTexCoord;  
  
out vec4 color;  
out vec2 texCoord;  
  
void main() {  
    color          = vColor;  
    texCoord       = vTexCoord;  
    gl_Position   = vPosition;  
}
```



Fragment Shader

```
in vec4 color;  
in vec2 texCoord;  
out vec4 FragColor;  
  
uniform sampler texture;  
  
void main() {  
    FragColor = color * texture(texture, texCoord);  
}
```