OpenGL Application Development
A Simplified Pipeline Model

Application → GPU Data Flow → Framebuffer

- Vertices
  - Vertex Processing
  - Vertex Shader

- Vertices
  - Rasterizer

- Fragments
  - Fragment Processing
  - Fragment Shader

- Pixels

Vertices → Rasterizer → Fragments → Pixels
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
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
Our First Program

- 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
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)

```cpp
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

```cpp
typedef vec4 point4;
typedef vec4 color4;
```
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

```c
point4 points[NumVertices];
color4 colors[NumVertices];
```
// 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
};
// 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++;
}
// 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 );
}
VAOs store the data of a geometric object

Steps in using a VAO

1. generate VAO names by calling `glGenVertexArrays()`
2. bind a specific VAO for initialization by calling `glBindVertexArray()`
3. update VBOs associated with this VAO
4. bind VAO for use in rendering

This approach allows a single function call to specify all the data for an object previously, you might have needed to make many calls to make all the data current
// 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
  1. generate VBO names by calling `glGenBuffers()`
  2. bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, ...)`
  3. load data into VBO using `glBufferData(GL_ARRAY_BUFFER, ...)`
  4. bind VAO for use in rendering `glBindVertexArray()`
// 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 );
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()`
// set up vertex arrays (after shaders are loaded)

GLuint vPosition = glGetUniformLocation(program, "vPosition");

glEnableVertexAttribArray(vPosition);

glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

GLuint vColor = glGetUniformLocation(program, "vColor");

glEnableVertexAttribArray(vColor);

glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(points)));
Drawing Geometric Primitives

- For contiguous groups of vertices
  
  \[
  \text{glDrawArrays(GL_TRIANGLES, } 0, \text{ NumVertices);}
  \]

- Usually invoked in display callback

- Initiates vertex shader
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
The 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
How OpenGL Simulates Lights

- 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
- Define the surface properties of a primitive

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuse</td>
<td>Base object color</td>
</tr>
<tr>
<td>Specular</td>
<td>Highlight color</td>
</tr>
<tr>
<td>Ambient</td>
<td>Low-light color</td>
</tr>
<tr>
<td>Emission</td>
<td>Glow color</td>
</tr>
<tr>
<td>Shininess</td>
<td>Surface smoothness</td>
</tr>
</tbody>
</table>

- 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;
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)

g1_Position = Projection * ModelView * vPosition;

color = ambient + diffuse + specular;
color.a = 1.0;
Shader Examples
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
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 $y_{ij} = f(x_i, z_j)$.
Form a quadrilateral mesh

```cpp
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

```cpp
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;
}
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
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;
}
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