OpenGL

Comp 575/770
Spring 2016

What is OpenGL?

- Cross platform API for 2D and 3D rendering
  - only provides rendering APIs, no windowing/input/sound APIs
- Developers need an abstraction between graphics application and graphics hardware drivers
  - OpenGL provides a unified interface to all kinds of graphics hardware.
- Has libraries for many languages
- Originally released in 1992
  - At version 4.5 currently

Similar APIs

- OpenGL ES
  - Variant of OpenGL for Embedded Systems
  - Highly popular due to iOS, Android
  - Some high-end OpenGL features missing
- Direct 3D
  - Microsoft’s rendering API
  - Xbox and Windows
  - Feature set nearly identical to OpenGL
- Vulkan
  - Spiritual successor to OpenGL, was initially called the “next generation OpenGL initiative”
  - Lower level api with less driver overhead
  - Can distribute workloads over multiple CPU cores
  - SPIR-V

OpenGL API Family

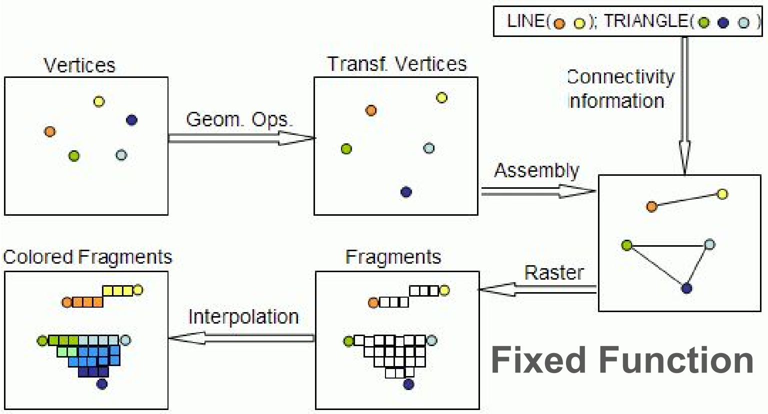
- OpenGL
  - **only provides rendering API**
- GLU (OpenGL Utility Functions)
  - OpenGL Utility functions
  - Various helper functions for matrices, surfaces, etc.
  - Packaged with OpenGL
- GLUT (OpenGL Utility Toolkit)
  - Manages window creation, keyboard/mouse input, etc.
  - Modern implementation: FreeGLUT
- GLFW (OpenGL Framework)
  - Similar to GLUT, but modern and gives finer control over the event/game loop
  - Better for games and applications that need control over loop timing

OpenGL Pipeline

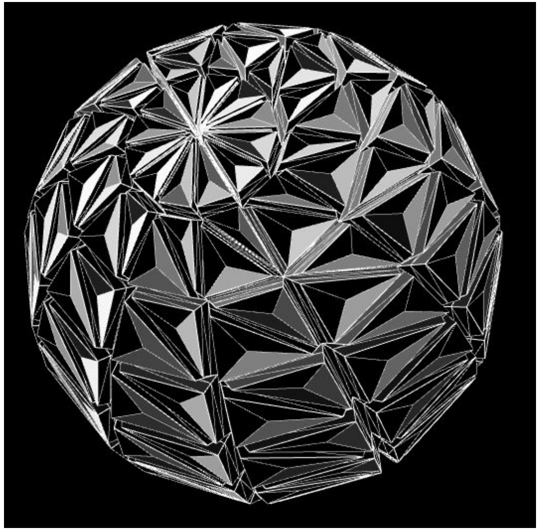
- implements a standard graphics pipeline (for rasterization)
- Most GPUs today are **programmable**
- Many are not though
    - OpenGL provides fixed-function mode for these

OpenGL Pipeline

- Fixed-Function
  - Overall pipeline is fixed, with some configurability
  - Can specify matrices, configure depth buffer, etc.
  - Once data is specified, OpenGL takes over
  - Can perform per-vertex lighting
- Programmable
  - Can specify shaders for different stages of the pipeline
  - Vertex shaders, fragment shaders, geometry shaders, etc.
  - Shaders written in GLSL (OpenGL Shading Language)
  - Preferred way to write OpenGL code



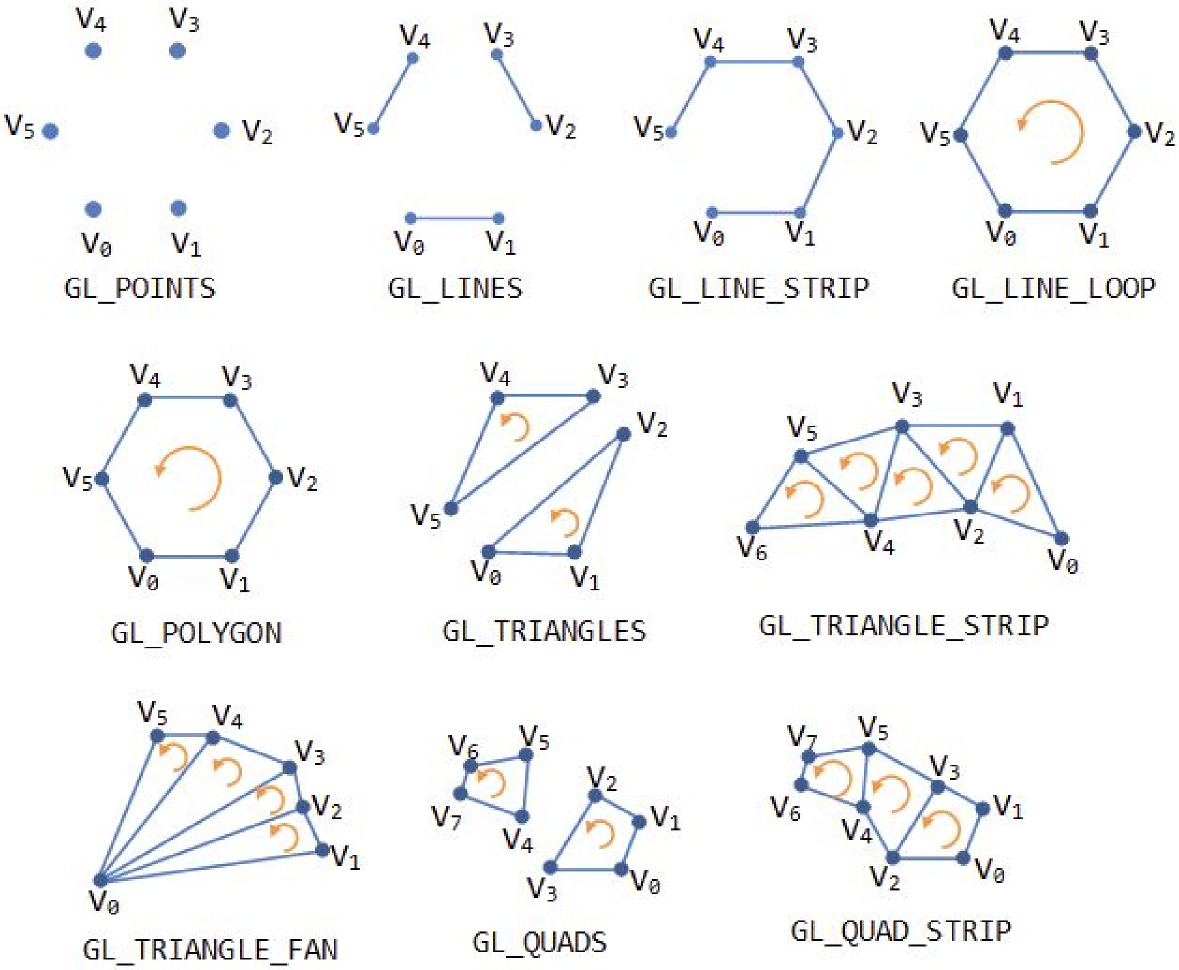
Fixed Function



Geometry Shader

Primitives

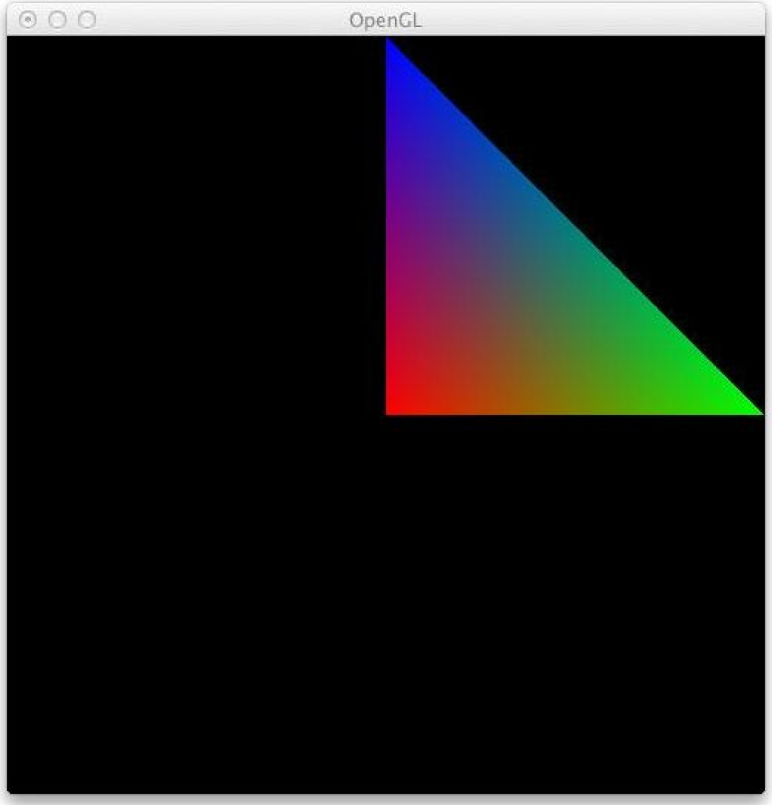
- Input to the fixed-function pipeline are primitives
- Primitives are a sequence of vertices
- OpenGL interprets them differently depending on arguments to *glBegin*
- Triangle List
  - list of triangles, every 3 vertices is interpreted as a triangle
- Triangle Strip
  - first 3 vertices are interpreted as a triangle
  - every vertex after the first 3 make a new triangle by reusing the preceding 2 vertices
- Triangle Fan
  - first 3 vertices are interpreted as a triangle
  - every vertex after the first 3 make a new triangle by reusing the preceding vertex and the first vertex



OpenGL Primitives

Primitive Example

```
glBegin(GL_TRIANGLES);
glColor3f(1, 0, 0);
glVertex2f(0, 0);
glColor3f(0, 1, 0);
glVertex2f(1, 0);
glColor3f(0, 0, 1);
glVertex2f(0, 1);
glEnd();
```



Primitive Example

```
glBegin(GL_TRIANGLES);
glColor3f(1, 0, 0);
glVertex2f(0, 0);
glColor3f(0, 1, 0);
glVertex2f(1, 0);
glColor3f(0, 0, 1);
glVertex2f(0, 1);
glEnd();
```

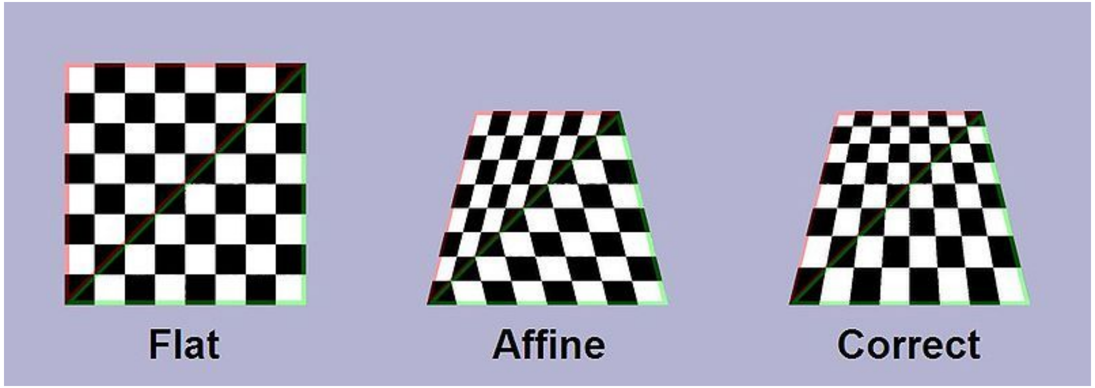
- All OpenGL functions begin with *gl*
- Vertex is the “name” of the function
- The following number (Vertex**2**) specifies how many components to use
- The last letter (Vertex2**f**) specifies the components are floats

Color Interpolation

- Colors are interpolated between the vertices of a triangle
- Naive (affine): linear interpolation
- Perspective correct: linear interpolate colors divided by depth and then use interpolated reciprocal of depth to recover the color

$$u_\alpha = (1-\alpha)u_0 + \alpha u_1$$

$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}}$$

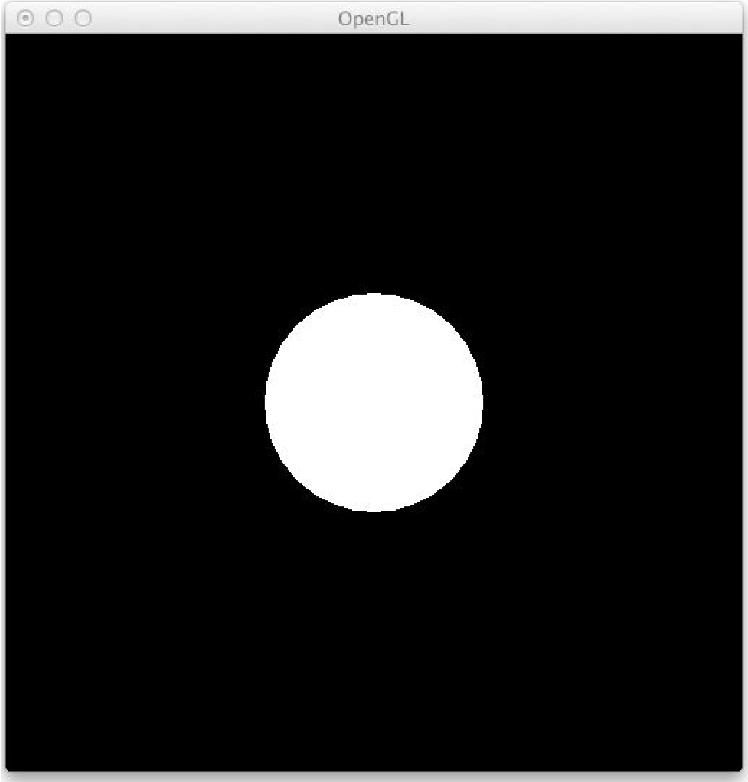


Transform Pipeline

- Our version:
  - $\mathbf{p} = \mathbf{M}_{vp}\mathbf{M}_{orth}\mathbf{M}_{persp}\mathbf{M}_{cam}\mathbf{M}_{model}\mathbf{p}_0$
- OpenGL:
  - $\mathbf{p} = \mathbf{M}_{viewport}\mathbf{M}_{projection}\mathbf{M}_{modelview}\mathbf{p}_0$
- Viewport matrix not stored explicitly

Transform Pipeline

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0, 0, 0, 0, 0, -1, 0, 1, 0);
glTranslatef(0, 0, -7);
glScalef(2, 2, 2);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-0.1, 0.1, -0.1, 0.1, 0.1, 1000);

glViewport(0, 0, 512, 512);

glutSolidSphere(1, 32, 16);
```



Transform Pipeline

- OpenGL stores matrices in **column-major** order
- Matrices are multiplied into the current matrix (multiplication mutates the current matrix instead of producing a new one)
  - matrix multiplication occurs from the right
- Near and far depths are **positive** by convention

Back-Face Culling

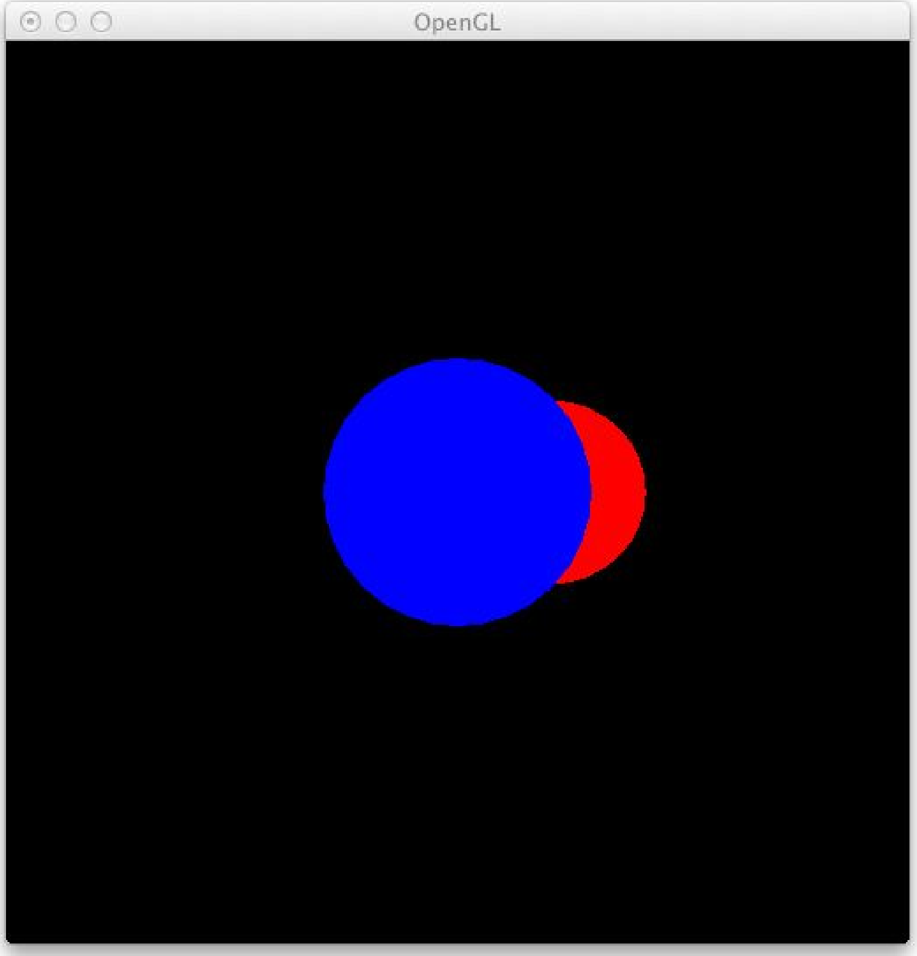
- Some faces will only have one face ever visible (perhaps the other face is occluded by other faces), so culling these faces reduces the number of faces to rasterize.
- To enable back-face culling: *glEnable(GL_CULL_FACE);*
- To specify which face is the front face: *glFrontFace(GL_CCW);*
    - GL_CCW and GL_CW are accepted (counterclockwise and clockwise)
    - these are in terms of vertex order in window coordinates
    - default: GL_CCW
- To specify which side to cull: *glCullFace(GL_BACK);*

Depth Buffering

- Need a way to determine which face is visible when faces occlude each other
  - keep track of depth values per pixel while rasterizing
  - only render when the depth value is smaller than the previous value
- Depth buffer must be initialized when window is created
  - glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
- Must be cleared before rendering:
  - glClearDepth(1000);
    glClear(GL_DEPTH_BUFFER_BIT);

Depth Buffering

```
// Clear depth buffer.

// Set up viewport and projection matrices.

glEnable(GL_DEPTH_TEST);

glMatrixMode(GL_MODELVIEW);

glLoadIdentity();
glTranslatef(0, 0, -7);
glColor3f(0, 0, 1);
glutSolidSphere(2, 32, 16);

glLoadIdentity();
glTranslatef(2, 0, -10);
glColor3f(1, 0, 0);
glutSolidSphere(2, 32, 16);
```



Shading

- Blinn-Phong shading model:
  - $L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p$
- Need to specify:
  - Material properties: k_a, k_d, k_s, p
  - Light parameters: I_a, I
- Vertex Properties:
  - glVertex3f specifies vertex positions
  - glNormal3f specifies normals
- Same information needed in our raytracer

Material Properties

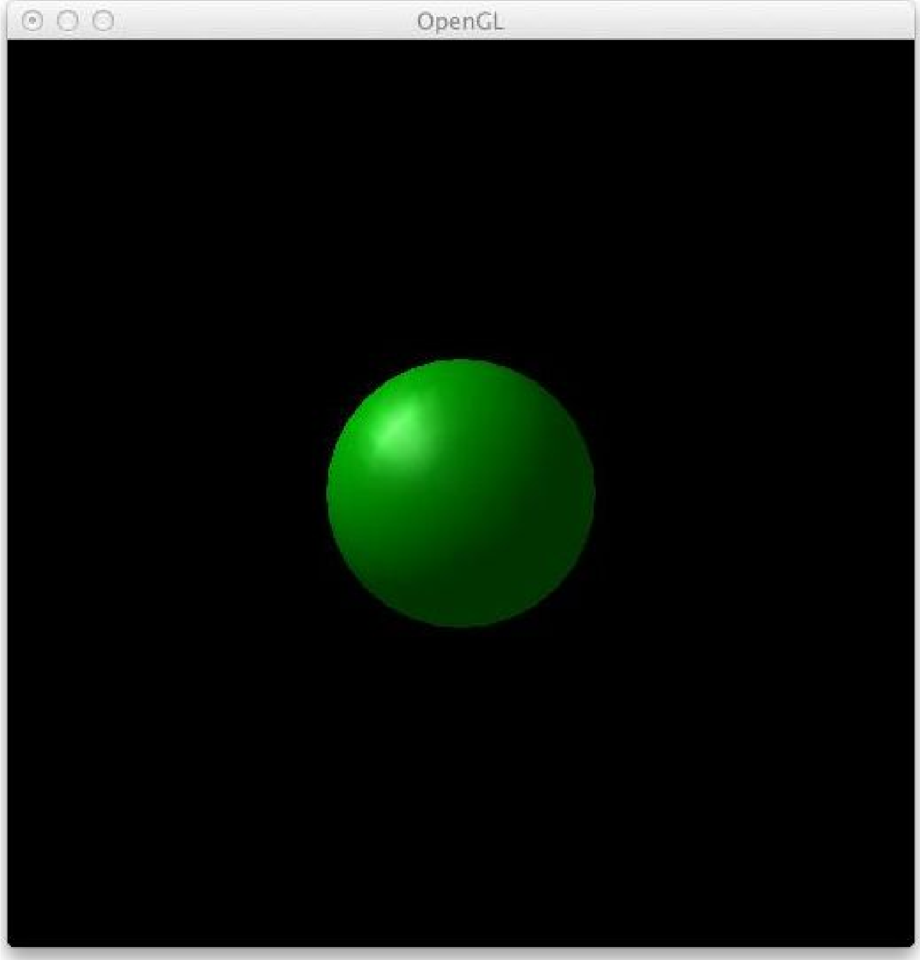
```
float ka[] = {0, 1, 0, 0};
float kd[] = {0, 0.5, 0, 0};
float ks[] = {0.5, 0.5, 0.5, 0};
float p = 32;

glMaterialfv(GL_FRONT, GL_AMBIENT, ka);
glMaterialfv(GL_FRONT, GL_DIFFUSE, kd);
glMaterialfv(GL_FRONT, GL_SPECULAR, ks);
glMaterialf(GL_FRONT, GL_SHININESS, p);
```

Light Parameters

```
float Ia[] = {0.2, 0.2, 0.2, 0};
float l[] = {-4, 4, 4, 1};
float la[] = {0, 0, 0, 0};
float ld[] = {1, 1, 1, 0};
float ls[] = {1, 1, 1, 0};

glLightModelfv(GL_LIGHT_MODEL_AMBIENT, Ia);

glLightfv(GL_LIGHT0, GL_POSITION, l);
glLightfv(GL_LIGHT0, GL_AMBIENT, la);
glLightfv(GL_LIGHT0, GL_DIFFUSE, ld);
glLightfv(GL_LIGHT0, GL_SPECULAR, ls);
```

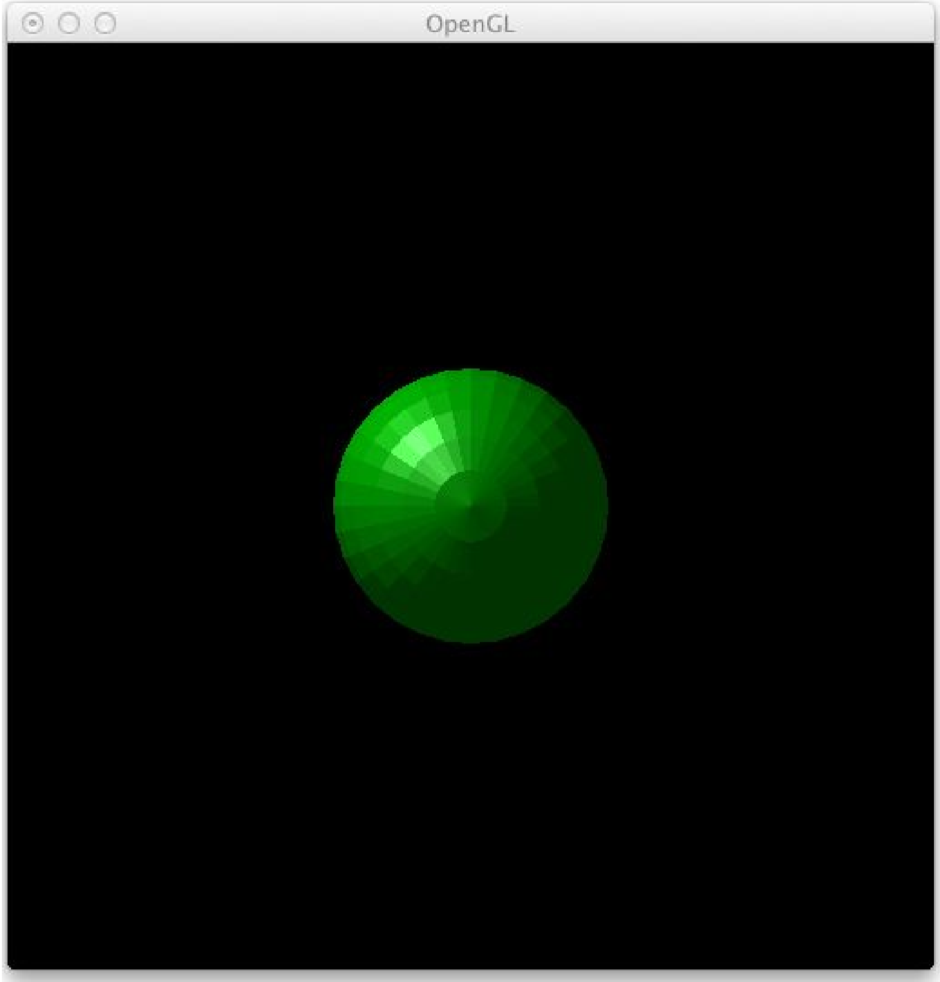
Shading

```
// Clear depth buffer, enable depth test.

// Set up transform pipeline.

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Configure material properties and light parameters.

glutSolidSphere(2, 32, 16);
```



Flat Shading

```
// Clear depth buffer, enable depth test.

// Set up transform pipeline.

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Configure material properties and light parameters.

glShadeModel(GL_FLAT);

glutSolidSphere(2, 32, 16);
```



Fixed Function Limitations

- Can’t do per-pixel shading. Once vertices are specified, OpenGL takes over.
- Can’t do deferred shading (can’t redirect output to an off screen buffer)
- Communication Bottleneck
  - Specify vertices over and over again
  - Data travels from CPU to GPU unnecessarily
  - Can we store this data in the GPU memory?

GPU Buffers

- Can allocate buffers (arrays) in GPU memory to store vertices, indices and other data

```
// "data" is a pointer to an array containing "size" bytes
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
```

- Buffer not allocated until glBufferData is called
- GL_STATIC_DRAW declares the buffer as read only, allowing the driver to optimize where it allocates the buffer.

Binding in OpenGL

- *glBufferData* is not passed the buffer itself
- GL_ARRAY_BUFFER is a **binding point**
- It can be set to any buffer using *glBindBuffer*
- OpenGL has many bind points and objects that can be bound

Vertex Array Objects

- Buffer data needs to be interpreted a certain way. We haven't told OpenGL what is semantically contained in the GPU buffer.
  - Is the buffer full of vertices? normals? vertices followed by normals?
- Described using **vertex array objects**
  - Defines the semantics of buffers

Vertex Array Objects

```
GLuint vertexArray;
glGenVertexArrays(1, &vertexArray);
glBindVertexArray(vertexArray);

// Positions and normals stored in the buffer
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

// Positions and normals 4 floats each, interleaved
// p0 n0 p1 n1 p2 n2...
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4*sizeof(float), 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 4*sizeof(float),
                      4*sizeof(float));
```

Element Buffers

- Storing each vertex for each triangle is costly and prone to repetition
- Solution: store each vertex only once and refer to them with indices
    - Thus, a triangle would be three indices corresponding to three vertices
- These indices may be stored on the GPU in an **element buffer**
- Bind a buffer to *GL_ELEMENT_ARRAY_BUFFER* to use it as an element buffer

Drawing with Buffers

- Without an element buffer

```
// Draw two triangles.
glDrawArrays(GL_TRIANGLES, 0, 6);
```

- With an element buffer

```
// Draw two triangles.
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

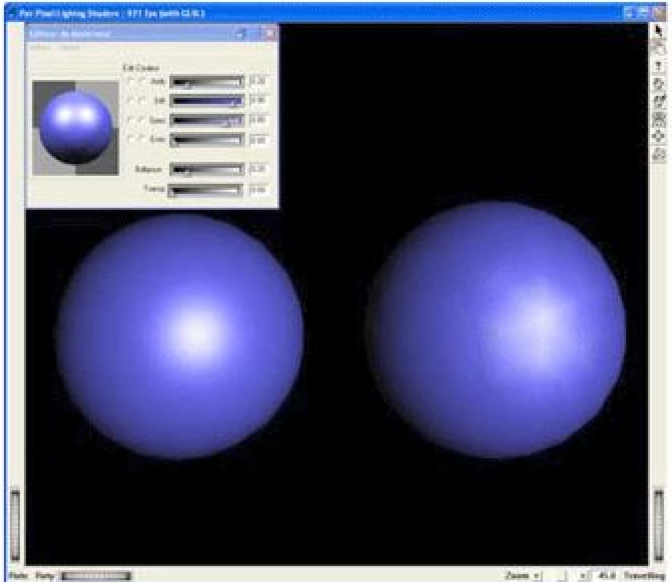
- The appropriate vertex array object and element buffer **must** be bound

Other Buffers

- Textures: storing images to map onto objects
- Uniform shader variables
  - e.g., an array of light source positions
- Frame Buffers
  - destination for rendering operations
  - useful for deferred shading, shadows, etc.

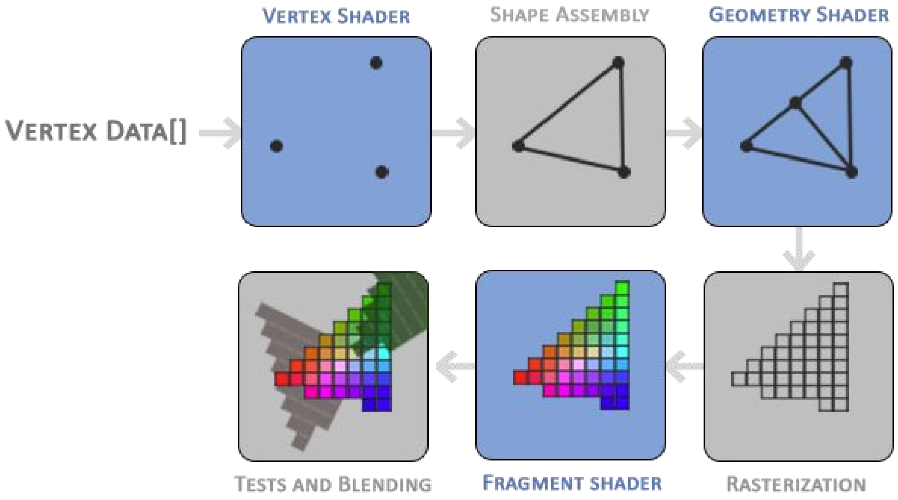
Fixed Function Limitations

- Can't do per-pixel shading. Once vertices are specified, OpenGL takes over.
- Without per-pixel shading, color is calculated for the vertices and then interpolated inside the triangle. Our shading model is not linear!
- With per-pixel shading, color is computed for each pixel



Programmable Pipeline

- Arbitrary code execution during certain phases of the pipeline:
  - Vertex processing, fragment processing, and other pipeline stages
- Some pipeline stages remain fixed: perspective divide, clipping, rasterization, depth buffer, …
- Shader programs written in **GLSL**
  - compiled on the CPU (by graphics drivers) and uploaded to the GPU



Simplest Vertex Shader

```
#version 330

in vec4 position;

void main()
{
  gl_Position = position;
}
```

- *in* defines a vertex attribute
- *gl_Position* is the position in canonical view volume
- This shader passes through positions it's given

Simplest Fragment Shader

```
#version 330

out vec4 outColor;

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

- *out* defines an output value
- This shader outputs the same color for all fragments

Compiling Shader Programs

```
// vsSource is a string containing the vertex shader source
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vsSource, NULL);
glCompileShader(vertexShader);

// fsSource is a string containing the fragment shader source
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fsSource, NULL);
glCompileShader(fragmentShader);

GLuint program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);
glLinkProgram(program);

glUseProgram(program);
```

Transform Vertex Shader

```
in vec4 position;

uniform mat4 modeling;
uniform mat4 camera;
uniform mat4 projection;

void main()
{
  gl_Position = projection * camera * modeling * position;
}
```

- uniform variables are inputs to vertex shaders
- they have the same value for all vertices (each vertex shares the same value)

Binding Uniform Variables

```
// Assumes modelingMatrix is a row-major 4x4 matrix.
GLint modelingUniform = glGetUniformLocation("modeling");
glUniformMatrix4fv(modelingUniform, 1, GL_TRUE, modelingMatrix);

// Repeat for "camera" and "projection".
```

Per-Vertex Shader

Vertex Shader

```
in vec4 position;
in vec4 normal;
uniform mat4 modeling;
uniform mat4 modeling_inv_tr;
uniform mat4 camera;
uniform mat4 projection;
out vec4 color;

vec4 shade(vec4 wp, vec4 wn) { // Shading code goes here. }

void main()
{
  gl_Position = projection * camera * modeling * position;
  vec4 wPos = modeling * position;
  vec4 wNormal = modeling_inv_tr * normal;
  color = shade(wPos, wNormal);
}
```

Fragment Shader

```
in vec4 color;
out vec4 outColor;

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

Flat Shading (Vertex Shader)

```
flat in vec4 color;
out vec4 outColor;

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

Per-Pixel Shading

Vertex Shader

```
in vec4 position;
in vec4 normal;
uniform mat4 modeling;
uniform mat4 modeling_inv_tr;
uniform mat4 camera;
uniform mat4 projection;
out vec4 wPosition;
out vec4 wNormal;

void main()
{
  gl_Position = projection * camera * modeling * position;
  wPosition = modeling * position;
  wNormal = modeling_inv_tr * normal;
}
```

Fragment Shader

```
in vec4 wPosition;
in vec4 wNormal;
out vec4 outColor;

void main()
{
  outColor = shade(wPosition, wNormal);
}
```

Other Shaders

- Vertex and fragment shaders are not the only shaders
- Geometry shaders
  - Runs on each primitive, outputs one or more primitives
  - Useful for cube map rendering
- Tessellation shaders
  - Useful for rendering curved surfaces
- Compute shaders
  - Essentially GPGPU code

Further Information

- OpenGL and GLSL Specs
  - http://www.khronos.org/opengl
- Microsoft Docs
  - https://msdn.microsoft.com/en-us/library/windows/desktop/dd374278(v=vs.85).aspx
- Tutorials
  - http://learnopengl.com/