OpenGL

COMP 575/770
Spring 2013
What is OpenGL?

- An API for 2D and 3D rendering
  - Cross-platform (Windows, Mac, consoles, mobile...)
  - Multi-language (C/C++, Java, Python...)

- Presents a unified interface to all kinds of graphics hardware
  - Scales from low-power phones to high-end desktops

- Originally released in 1992
  - Now at version 4.3
OpenGL Applications

Rage (id Software, 2011)
Similar APIs

- **OpenGL ES**
  - Variant of OpenGL for use in Embedded Systems
  - Highly popular due to iOS, Android
  - Some high-end OpenGL features missing

- **Direct3D**
  - Microsoft’s rendering API
  - Driven by Xbox 360
  - Feature set nearly identical to OpenGL
OpenGL API Family

- OpenGL
  - The main rendering API

- GLU
  - 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
OpenGL Pipeline

- OpenGL implements the standard graphics pipeline
  - i.e., designed for rasterization

- Most GPUs today are **programmable**
  - Vertex shaders, fragment shaders, etc.

- Many are not, though

- For these, OpenGL has **fixed-function** mode
OpenGL Pipeline

- **Fixed-Function**
  - Overall pipeline is fixed, with some configurability
  - Can specify matrices, configure depth buffer tests, etc.
  - Can perform per-vertex lighting

- **Programmable**
  - Can specify shaders for different stages of the pipeline
  - Shaders written in **GLSL** (OpenGL Shading Language)
  - Preferred way to write OpenGL code
A Minimal GLUT Program

```c
int main(int argc, char** argv)
{
    // Initialize GLUT.
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutCreateWindow("OpenGL");

    // Set up GLUT callbacks.
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);

    // Start rendering.
    glutMainLoop();
    return 0;
}
```
A Minimal GLUT Program

```c
void reshape(int w, int h)
{
    glutPostRedisplay();
}

void display()
{
    glutSwapBuffers();
}
```
A Minimal GLUT Program

```c
void reshape(int w, int h)
{
    glutPostRedisplay();
}

void display()
{
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);

    glutSwapBuffers();
}
```
Rasterization

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

Draws a single triangle
Function Names

```c
glVertex2f(0, 1);
```

- All OpenGL functions start with `gl`
- `Vertex` is the “name” of the function
- Specifies a vertex using 2 numbers
- Each of these numbers is a `float`
Rasterization

glBegin(GL_TRIANGLES);
.glColor3f(0, 1, 0);
 glVertex2f(0, 0);
 glVertex2f(1, 0);
 glVertex2f(0, 1);
 glEnd();  }

Draws a single green triangle
Rasterization

```c
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();
```

Draws a single triangle with interpolated colors
Primitive Assembly

- All OpenGL sees is a sequence of vertices

- What to make of them?
  - Specify using `glBegin`

- Tells OpenGL how to assemble a primitive from a sequence of vertices

- Multiple possible values can be passed
(0, 5, 0)  (10, 5, 0)  (20, 5, 0)

(-5, -5, 0)  (5, -5, 0)  (15, -5, 0)

GL_POINTS
GL_LINES

(-5, -5, 0) → (0, 5, 0)
(5, -5, 0) → (10, 5, 0)
(15, -5, 0) → (20, 5, 0)
Transform Pipeline

- Our version:

  \[ p = M_{vp} M_{orth} M_{persp} M_{cam} M_{model} p_0 \]

- OpenGL’s version:

  \[ p = M_{viewport} M_{projection} M_{modelview} p_0 \]

- Viewport matrix not stored explicitly
Transform Pipeline

```c
# Transform Pipeline

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

# Projection Transform

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**
  - Opposite to the usual convention in many languages!
  - More relevant when using programmable pipeline

- Matrices aren’t just calculated, they’re **multiplied** into the “current” matrix
  - Multiplication occurs from the **right**

- Near and far depths are **positive** by convention
Back–Face Culling

- To enable back–face culling:

  ```
  glEnable(GL_CULL_FACE);
  ```

- To specify which face is the front face:
  - In terms of vertex order in window coordinates

  ```
  glFrontFace(GL_CCW);
  ```

- To specify which side to cull:

  ```
  glCullFace(GL_BACK);
  ```
Depth Buffering

- 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);
```
// Clear depth buffer.

// Set up viewport and projection matrices.

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);
Depth Buffering

// Clear depth buffer.

// Set up viewport and projection matrices.

glEnable(GL_DEPTH_TEST);

glMatrixMode(GL_MODELVIEW);

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

glLoadIdentity();

setPosition(2, 0, -10);
setColor(1, 0, 0);
glutSolidSphere(2, 32, 16);

setPosition(2, 0, -10);
setColor(1, 0, 0);
glutSolidSphere(2, 32, 16);
Shading

- Blinn–Phong shading model:
  
  \[ L = k_a I_a + k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^p \]

- Need to specify:
  - Material properties \( k_a, k_d, k_s, p \)
  - Light parameters \( I_a, I \)

- Vertex properties:
  - \texttt{glVertex3f} specifies positions
  - \texttt{glNormal3f} specifies normals
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

```c
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);
```

NOTE: Model–View matrix is applied to the light position, so need to compensate
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

- No way to do per-pixel shading
  - Once vertices are specified, OpenGL takes over

- No way to do deferred shading
  - How to redirect output to an offscreen buffer?

- There’s still a bottleneck!
  - We’re specifying vertices over and over again
  - Data travels from CPU to GPU unnecessarily
  - How to store vertex data in GPU memory?
Programmable Pipeline

- Arbitrary code executed during:
  - Vertex processing
  - Fragment processing
  - Other pipeline stages

- Some pipeline stages remain fixed
  - Perspective divide, clipping, rasterization, depth buffer…

- Shader programs written using **GLSL**
  - Compiled on CPU, uploaded to GPU
The Simplest Vertex Shader

```cpp
#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 just passes the vertex through as-is
#version 330

out vec4 outColor;

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

- `out` defines an output value
- This shader just outputs the same color for all fragments
Compiling Shaders

// 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);
Shader Programs

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

glUseProgram(program);
in vec4 position;

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

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

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

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

- **Uniform variables** are inputs to vertex shaders
- They have the same value for all vertices
  - They are not vertex attributes
Per-Vertex Shading (VS)

```
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);
}
```
Per-Vertex Shading (FS)

```glsl
in vec4 color;
out vec4 outColor;

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

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

void main()
{
    outColor = color;
}
```
Per-Pixel Shading (VS)

```glsl
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;
}
```
Per-Pixel Shading (FS)

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

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

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

```c
// "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` called

- `GL_STATIC_DRAW` indicates that the buffer won’t be modified
  - Lets the driver optimize where it allocates the buffer
Binding in OpenGL

- `glBufferData` isn’t passed the buffer itself

- `GL_ARRAY_BUFFER` is a binding point

- Like a “global variable” can be set to any buffer
  - Using `glBindBuffer`
  - Buffer `bound` to `GL_ARRAY_BUFFER`

- Many other binding points and objects that can be bound
Vertex Array Objects

- Need to specify how the data in a buffer should be interpreted
  - Positions? Normals? Positions followed by normals?

- Described using **vertex array objects**
  - Defines the semantics of buffers
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

- Store each vertex exactly once
  - Describe the triangle using 3 indices into the vertex array

- The indices may also be stored in GPU memory
  - In an element buffer

- Bind a buffer to `GL_ELEMENT_ARRAY_BUFFER` to use as an element buffer
Drawing From Buffers

- Without an index buffer:

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

- With an index 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

- Many other uses for GPU memory buffers, including

- Textures
  - Storing images to paint on objects

- Uniform shader variables
  - e.g., an array of light source positions

- Frame buffers
  - Destination for rendering operations
  - Useful for deferred shading, shadows, etc.
Other Shaders

- Vertex and fragment shaders are not the only kinds of 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 Specifications
  - [http://www.khronos.org/opengl](http://www.khronos.org/opengl)

- Microsoft/Apple documentation

- Tutorials