Announcement

Homework 1 has been posted in dropbox and course website

Due: 1:15 pm, Monday, September 12
Today’s Agenda

Primitives

Programming with OpenGL
OpenGL Primitives

GL_POINTS

GL_LINES

GL_LINE_STRIP

GL_LINE_LOOP

GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN
## OpenGL Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_POINTS</td>
<td>Each vertex is a single point on the screen.</td>
</tr>
<tr>
<td>GL_LINES</td>
<td>Each pair of vertices defines a line segment.</td>
</tr>
<tr>
<td>GL_LINE_STRIP</td>
<td>A line segment is drawn from the first vertex to each successive vertex.</td>
</tr>
<tr>
<td>GL_LINE_LOOP</td>
<td>Same as GL_LINE_STRIP, but the last and first vertex are connected.</td>
</tr>
<tr>
<td>GL_TRIANGLES</td>
<td>Every three vertices define a new triangle.</td>
</tr>
<tr>
<td>GL_TRIANGLE_STRIP</td>
<td>Triangles share vertices along a strip.</td>
</tr>
<tr>
<td>GL_TRIANGLE_FAN</td>
<td>Triangles fan out from an origin, sharing adjacent vertices.</td>
</tr>
</tbody>
</table>
Primitive #1: Points

Points are either 2- or 3-dimensional
  • by convention, represent them as column vectors

\[
\mathbf{v} = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad \mathbf{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}
\]

A 2D point, a special case of a 3D point, can be represented as
  • A 2D vector (e.g., vec2(0, 1)),
  • A 3D vector (e.g., vec3(0, 1, 0)),
  • and more general a 4D vector (e.g., vec4(0, 1, 0, 1)),

\[
glDrawArrays(GL_POINTS, 0, N);
\]
Primitive #2: Line Segments

2-D lines are the set of all points satisfying

\[ ax + by + c = 0 \quad \text{or} \quad \mathbf{n} \cdot \mathbf{p} + c = 0 \]

- vector \([a \ b]\) is perpendicular to segment
- it is a normal vector of the segment
- (almost) always want unit normals!

\[ \mathbf{n} \cdot \mathbf{n} = a^2 + b^2 = 1 \]

Can also use a vector-valued function:

\[ \mathbf{p}(t) = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1) \quad \text{for} \ 0 \leq t \leq 1 \]
Drawing Piecewise-Linear 2-D Curves

The 3 types of polyline objects:
- GL_LINE_STRIP — open curve
- GL_LINE_LOOP — closed curve
- GL_LINES — separate segments
Triangle

Triangles define a unique plane in 3-D
• set of all points satisfying equation
  \[ ax + by + cz + d = 0 \]
• vector \([ a \ b \ c ]\) is the plane normal
• hence perpendicular to the triangle
• typically use unit normal vector
  \[ a^2 + b^2 + c^2 = 1 \]

Normals will show up again and again
• especially in rendering
Polygons

OpenGL will only display triangles

- **Simple**: edges cannot cross, i.e., only meet at the end points
- **Convex**: All points on line segment between two points in a polygon are also in the polygon
- **Flat**: all vertices are in the same plane

Display triangles in three ways:

- Points (GL_POINT)
- Edges (GL_LINE)
- Filled (GL_FILL)

Determined by glPolygonMode
Polygon Issues

OpenGL will only display triangles

- **Simple**: edges cannot cross, i.e., only meet at the end points
- **Convex**: All points on line segment between two points in a polygon are also in the polygon
- **Flat**: all vertices are in the same plane

Application program must tessellate a polygon into triangles (triangulation)

nonsimple polygon

nonconvex polygon
Polygon Testing

Conceptually simple to test for simplicity and convexity
Time consuming
Earlier versions left testing to the application
Present version only renders triangles

Need algorithm to triangulate an arbitrary polygon
• trivial if polygon is convex: connect all vertices to a point of interior
• requires more sophisticated algorithms for general polygons
Optimizing Drawing: Triangle Strips

Emitting vertices costs something
- each must be transformed fewer vertices = faster drawing

Take advantage of prior vertices
- first 3 specify triangle
- for each subsequent vertex
  - take previous 2 vertices
  - this will define the next triangle

Up to a factor of 3 improvement
- for sufficiently long strips
- requires only 1 vertex/triangle
Triangle Fans

start with a central point
build triangles around it

Also 1 vertex per triangle
• if the loop is sufficiently large
• but it usually won’t be

```
glBegin(GL_TRIANGLE_FAN);
glVertex3fv(v1);
glVertex3fv(v2);
glVertex3fv(v3); // Triangle A
glVertex3fv(v4); // Triangle B
glVertex3fv(v5); // Triangle C
glVertex3fv(v6); // Triangle D
glEnd();
```
Good and Bad Triangles

Long thin triangles render badly

Equilateral triangles render well

Maximize minimum angle

Delaunay triangulation for unstructured points

A Simple Program (?)

Generate two triangles on a solid background
OpenGL Camera

OpenGL places a camera at the origin in object space pointing in the negative $z$ direction.

The default viewing volume is a box centered at the origin with sides of length 2.
Orthographic Viewing

In the default orthographic view, points are projected forward along the $z$ axis onto the plane $z=0$. 

![Diagram of orthographic viewing](image)

Orthographic Viewing

OpenGL coordinates

(0,0)

Clipping

(a)

(b)

Clipping rectangle

FIGURE 2.35 Two-dimensional viewing. (a) Objects before clipping. (b) Image after clipping.

From Vertex to Screen

Mapping from vertex coordinates to screen coordinates

Aspect ratio mismatched

Flexible Way to Treat it

Solution: Do not have use the entire window for the image

```c
void glViewport (GLint x, GLint y, GLsizei w, GLsizei h)
• (x,y): Lower-left corner of the view port in pixels
• (w,h): width and height in pixels
```
Now, Let’s Start the First Program

Build a complete first program
  • Introduce shaders
  • Introduce a standard program structure

Initialization steps and program structure
Program Structure

Most OpenGL programs have a similar structure that consists of the following functions

- **main:**
  - specifies the callback functions
  - opens one or more windows with the required properties
  - enters event loop (last executable statement)

- **init():** sets the state variables
  - Viewing
  - Attributes

- **initShader:** read, compile and link shaders

- **callbacks**
  - Display function
  - Input and window functions
triangle.c

enum VAO_IDs { Triangles, NumVAOs };  
enum Buffer_IDs { ArrayBuffer, NumBuffers };  
enum Attrib_IDs { vPosition = 0 };  

GLuint VAOs[NumVAOs];  
GLuint Buffers[NumBuffers];  

const GLuint NumVertices = 6;
#include <GL/glew.h>
#include <GL/freeglut.h>

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA);
    glutInitWindowSize(512, 512);
    glutInitContextVersion(4, 3);
    glutInitContextProfile(GLUT_CORE_PROFILE);
    glutCreateWindow(argv[0]);
    if (glewInit()) {
        cerr << "Unable to initialize GLEW ... exiting" << endl;
        exit(EXIT_FAILURE);
    }
    init();
    glutDisplayFunc(display);
    glutMainLoop();
}
GLUT functions

- `glutInit` initializes GLUT library, processes command line arguments and setups data structures.

- `glutInitDisplayMode` requests properties for the window (the rendering context),
  - RGB color
  - Single buffering
  - Other options such as depth buffers, or animation

- `glutInitWindowSize` specifies the size of windows in pixels

- `glutInitContextVersion` and `glutInitContextProfile` specify the type of OpenGL context, i.e., the internal data structure
GLUT functions

`glutCreateWindow` creates window with title as arg[0]

`glewInit` initializes the GLEW library

`Init` initializes OpenGL states and initializes the shader

`glutDisplayFunc` sets up display callback

`glutMainLoop` enter infinite event loop to process user input
Initialization

Initialize the vertex array

Vertex array objects and buffer objects can be set up on init()

Also set up shaders as part of initialization
  • Read
  • Compile
  • Link
void init(void)
{
    glGenVertexArrays(NumVAOs, VAOs);

    glBindVertexArray(VAOs[Triangles]);

    GLfloat vertices[NumVertices][2] = {
        { -0.90, -0.90 },  // Triangle 1
        {  0.85, -0.90 },
        { -0.90,  0.85 },
        {  0.90, -0.85 },  // Triangle 2
        {  0.90,  0.90 },
        { -0.85,  0.90 }
    };

    GLuint VAOs[NumVAOs];
    Vertex-Array object: Bundles all vertex data (positions, colors, ...)
    Initialize the VAO and get name for buffer

    Create a new VAO with the assigned name
    or activate a VAO if binding to an existing VAO

    A vertex array can hold many attributes of vertices, such as position, color, texture, coordinates, etc.
GLuint Buffers[NumBuffers];
Buffer objects store data to be used
Create a BO and return a name

Specify the type of BO

GLint Target, e.g., vertex
attribute data, index
data, pixel data, etc

Usage, how the data will be read
and written, e.g.,
GL_STREAM_DRAW
init()

ShaderInfo shaders[] = {
    { GL_VERTEX_SHADER, "triangles.vert" },
    { GL_FRAGMENT_SHADER, "triangles.frag" },
    { GL_NONE, NULL }
};

GLuint program = LoadShaders(shaders);
glUseProgram(program);

glVertexAttribPointer(vPosition, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
glEnableVertexAttribArray(vPosition);
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(r,g,b, α=0);
    glBindVertexArray(VAOs[Triangles]);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}