Objectives

- Expanding primitive set
- Adding color
- Vertex attributes
- Uniform variables

OpenGL Primitives

- GL_POINTS
- GL_LINES
- GL_LINE_LOOP
- GL_LINE_STRIP
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
- GL_LINES
- GL_LINE_LOOP
- GL_LINE_STRIP
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN

Polygon Issues

- OpenGL will correctly display polygons that are:
  - Simple: edges cannot cross
  - 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
- Your application can check if the above hold
- Triangles satisfy all three conditions!
- Other polygons will be displayed, but correct?
- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator

Polygon Testing

- Conceptually simple to test for simplicity and convexity
- Time consuming
- Earlier versions assuming both and left testing to the application
- Present version only renders triangles
- Need algorithm to triangulate an arbitrary polygon
**Good and Bad Triangles**

- Long thin triangles render badly
- Equilateral triangles render well
- Maximize minimum angle
- Delaunay triangulation for unstructured points

**Triangularization**

- Convex polygon

  ```
  b
   
  c
   
  d
  ```

- Start with abc, remove b, then acd, ....

**Non-convex (concave)**

```xml
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd" xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape" width="470" height="460" viewBox="0 0 470 460" style="enable-background:new 0 0 470 460;" xml:space="preserve">
  <polygon points="100 80, 180 120, 200 80, 280 120, 300 80, 380 120, 400 80, 420 120, 440 80, 100 80" style="fill:rgb(255,255,255); stroke:rgb(0,0,0); stroke-width:1;"/>
  <polygon points="100 80, 180 120, 200 80, 280 120, 300 80, 380 120, 400 80, 420 120, 440 80, 100 80" style="fill:rgb(0,0,0); stroke:rgb(0,0,0); stroke-width:1;"/>
  <path d="M100 80 L180 120 L200 80 L280 120 L300 80 L380 120 L400 80 L420 120 L440 80 L100 80" style="fill:rgb(0,0,0); stroke:rgb(0,0,0); stroke-width:1;"/>
</svg>
```

**Recursive Division**

- Find leftmost vertex and split

```xml
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd" xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape" width="470" height="460" viewBox="0 0 470 460" style="enable-background:new 0 0 470 460;" xml:space="preserve">
  <polygon points="100 80, 180 120, 200 80, 280 120, 300 80, 380 120, 400 80, 420 120, 440 80, 100 80" style="fill:rgb(255,255,255); stroke:rgb(0,0,0); stroke-width:1;"/>
  <polygon points="100 80, 180 120, 200 80, 280 120, 300 80, 380 120, 400 80, 420 120, 440 80, 100 80" style="fill:rgb(0,0,0); stroke:rgb(0,0,0); stroke-width:1;"/>
  <path d="M100 80 L180 120 L200 80 L280 120 L300 80 L380 120 L400 80 L420 120 L440 80 L100 80" style="fill:rgb(0,0,0); stroke:rgb(0,0,0); stroke-width:1;"/>
</svg>
```

**Attributes**

- Attributes determine the appearance of objects
- Color (points, lines, polygons)
- Size and width (points, lines)
- Stipple pattern (lines, polygons)
- Polygon mode
  - Display as filled: solid color or stipple pattern
  - Display edges
  - Display vertices
- Only a few (glPointSize) are supported by OpenGL functions

**RGB color**

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes
Indexed Color

- Colors are indices into tables of RGB values
- Requires less memory
  - indices usually 8 bits
  - Memory inexpensive
  - Need more colors for shading

Smooth Color

- Default is smooth shading
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is flat shading
  - Color of first vertex determines fill color
  - Handle in shader

Setting Colors

- Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
- Application color: pass to vertex shader as a uniform variable (next lecture) or as a vertex attribute
- Vertex shader color: pass to fragment shader as varying variable (next lecture)
- Fragment color: can alter via shader code

Objectives

- Coupling shaders to applications
  - Reading
  - Compiling
  - Linking
- Vertex Attributes
- Setting up uniform variables
- Example applications

GLSL II

Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables
Program Object

- Container for shaders
- Can contain multiple shaders
- Other GLSL functions

```c
GLuint myProgObj;
myProgObj = glCreateProgram();
/* define shader objects here */
glUseProgram(myProgObj);
glLinkProgram(myProgObj);
```

Reading a Shader

- Shaders are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function `glShaderSource`
- If the shader is in a file, we can write a reader to convert the file to a string

```c
static char* readShaderSource(const char* shaderFile)
{
    FILE* fp = fopen(shaderFile, "r");
    if (fp == NULL) { return NULL; }
    fseek(fp, GL SEEK_END);
    long size = ftell(fp);
    fseek(fp, GL SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);
    buf[size] = '\0';
    fclose(fp);
    return buf;
}
```

Adding a Vertex Shader

```c
GLuint vShader;
GLuint myVertexObj;
GLchar vShaderfile[] = "my_vertex_shader";
GLchar* vSource = readShaderSource(vShaderFile);
glShaderSource(myVertexObj, 1, &vShaderfile, NULL);
myVertexObj = glCreateShader(GL_VERTEX_SHADER);
glCompileShader(myVertexObj);
glAttachObject(myProgObj, myVertexObj);
```

Vertex Attributes

- Vertex attributes are named in the shaders
- Linker forms a table
- Application can get index from table and tie it to an application variable
- Similar process for uniform variables

```c
#define BUFFER_OFFSET( offset )
((GLvoid*)(offset))

GLuint loc =
glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( loc );
glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
glVertexAttribPointer(loc, index, size, type, normalized, stride, pointer);
```
Uniform Variable Example

```c
GLint angleParam;
angleParam = glGetUniformLocation(myProgObj, "angle");
/* angle defined in shader */
/* my_angle set in application */
GLfloat my_angle;
my_angle = 5.0 /* or some other value */
glUniform1f(angleParam, my_angle);
```

Double Buffering

- Updating the value of a uniform variable opens the door to animating an application
- Execute glUniform in display callback
- Force a redraw through glutPostRedisplay()
- Need to prevent a partially redrawn frame buffer from being displayed
- Draw into back buffer
- Display front buffer
- Swap buffers after updating finished

Adding Double Buffering

- Request a double buffer
  - glutInitDisplayMode(GLUT_DOUBLE)
- Swap buffers

```c
void myDisplay()
{
    glClear(……);
    glDrawArrays();
    glutSwapBuffers();
}
```

Idle Callback

- Idle callback specifies function to be executed when no other actions pending
- glutIdleFunc(myIdle);

```c
void myIdle()
{
    // recompute display
    glutPostRedisplay();
}
```

Attribute and Varying Qualifiers

- Starting with GLSL 1.5 attribute and varying qualifiers have been replaced by in and out qualifiers
- No changes needed in application
- Vertex shader example:

```c
#version 1.4
attribute vec3 vPosition;
varying vec3 color;
```

Adding Color

- If we set a color in the application, we can send it to the shaders as a vertex attribute or as a uniform variable depending on how often it changes
- Let’s associate a color with each vertex
- Set up an array of same size as positions
- Send to GPU as a vertex buffer object

```c
#version 1.5
in vec3 vPosition;
out vec3 color;
```
Setting Colors

typedef vec3 color3;

color3 base_colors[4] = {color3(1.0, 0.0, 0.0), ...}

vec3 colors[NumVertices];

//in loop setting positions

colors[i] = base_colors[color_index]
position[i] = ...;

Setting Up Buffer Object

//need larger buffer

glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
sizeof(colors), NULL, GL_STATIC_DRAW);

//load data separately

glBufferSubData(GL_ARRAY_BUFFER, 0,
sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
sizeof(colors), colors);

Second Vertex Array

// vPosition and vColor identifiers in vertex shader

loc = glGetUniformLocation(program, "vPosition");
gEnableVertexAttribArray(loc);
gVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));

loc2 = glGetUniformLocation(program, "vColor");
gEnableVertexAttribArray(loc2);
gVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(sizeof(points)));

Wave Motion Vertex Shader

in vec4 vPosition;
uniform float xs, zs; // frequencies
uniform float h; // height scale
void main()
{
vec4 t = vPosition;
t.y = vPosition.y
+ h*sin(time + xs*vPosition.x)
+ h*sin(time + zs*vPosition.z);
gl_Position = t;
}

Vertex Shader Applications

• Moving vertices
• Morphing
• Wave motion
• Fractals
• Lighting
• More realistic models
• Cartoon shaders

Particle System

in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 init_vel;
uniform float g, m, t;
void main()
{
vec3 object_pos;
object_pos.x = vPosition.x + vel.x*t;
object_pos.y = vPosition.y + vel.y*t
+ g/(2.0*m)*t*t;
object_pos.z = vPosition.z + vel.z*t;
gl_Position =
ModelViewProjectionMatrix*vec4(object_pos, 1);
}
Pass Through Fragment Shader

/* pass-through fragment shader */
in vec4 color;
void main(void)
{
    gl_FragColor = color;
}

Vertex vs Fragment Lighting

per vertex lighting

per fragment lighting

Fragment Shader Applications

Texture mapping

smooth shading

environment mapping

bump mapping

THREE DIMENSIONS

Objectives

• Develop a more sophisticated three-dimensional example
• Sierpinski gasket: a fractal
• Introduce hidden-surface removal

Three-dimensional Applications

• In OpenGL, two-dimensional applications are a special case of three-dimensional graphics
• Going to 3D
  • Not much changes
  • Use vec3, glGetUniformLocation
  • Have to worry about the order in which primitives are rendered or use hidden-surface removal
Sierpinski Gasket (2D)

- Start with a triangle
- Connect bisectors of sides and remove central triangle

Example

- Five subdivisions

The gasket as a fractal

- Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)
- As we continue subdividing
  - the area goes to zero
  - but the perimeter goes to infinity
- This is not an ordinary geometric object
  - It is neither two- nor three-dimensional
  - It is a fractal (fractional dimension) object

Gasket Program

```c
#include <GL/glut.h>
/* initial triangle */

void draw_triangle(point2 a, point2 b, point2 c) {
    /* display one triangle */
    {
        static int i = 0;
        points[i] = a;
        points[i] = b;
        points[i] = c;
        i += 3;
    }
}
```

Triangle Subdivision

```c
void divide_triangle(point2 a, point2 b, point2 c, int m) {
    /* triangle subdivision using vertex numbers */
    point2 ab, ac, bc;
    if(m>0) {
        ab = (a + b)/2;
        ac = (a + c)/2;
        bc = (b + c)/2;
        divide_triangle(a, ab, ac, m-1);
        divide_triangle(c, ac, bc, m-1);
        divide_triangle(b, bc, ac, m-1);
    } else (triangle(a,b,c));
    /* draw triangle at end of recursion */
}
```
display and init Functions

```c
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}

void myinit()
{
    vec2 v[3] = (point2(...
        .
        divide_triangles(v[0], v[1], v[2], n);
        .
}
```

main Function

```c
int main(int argc, char **argv)
{
    n=4;
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("2D Gasket");
    glutDisplayFunc(display);
    myinit();
    glutMainLoop();
}
```

Moving to 3D

- We can easily make the program three-dimensional by using
  - point3 v[3]
and we start with a tetrahedron

3D Gasket

- We can subdivide each of the four faces
- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra
- Code almost identical to 2D example

Almost Correct

- Because the triangles are drawn in the order they are specified in the program, the front triangles are not always rendered in front of triangles behind them

Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image
Using the z-buffer algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- It must be:
  - Requested in `main.c`
  - `glutInitDisplayMode` (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
  - Enabled in `init.c`
    - `glEnable(GL_DEPTH_TEST)`
  - Cleared in the display callback
    - `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`

Surface vs Volume Subdivision

- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a volume in the middle
- See text for code

Volume Subdivision