Computer Graphics (CS 4731)
Lecture 4 (Part 2): Building 3D Models
(Part 2)

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Old Way: Inefficient

- Previously drew cube by its 6 faces using
  - 6 `glBegin`, 6 `glEnd`
  - 6 `glColor`
  - 24 `glVertex`
  - More commands if we use texture and lighting
  - E.g: to draw each face

```gl
glBegin(GL_QUAD)
    glVertex(x1, y1, z1);
    glVertex(x2, y2, z2);
    glVertex(x3, y3, z3);
    glVertex(x4, y4, z4);
glEnd();
```
New Way: Vertex Representation and Storage

- We have declare vertex lists, edge lists and arrays
- But OpenGL expects meshes passed to have a specific structure
- We now study that structure....
Vertex Arrays

- **Previously:** OpenGL provided a facility called *vertex arrays* for storing rendering data
- Six types of arrays were supported initially
  - Vertices
  - Colors
  - Color indices
  - Normals
  - Texture coordinates
  - Edge flags
- Now vertex arrays can be used for *any attributes*
Vertex Attributes

- Vertices can have attributes
  - Position (e.g. 20, 12, 18)
  - Color (e.g. red)
  - Normal (x,y,z)
  - Texture coordinates
Vertex Attributes

- Store vertex attributes in **single** Array (array of structures)
Declaring Array of Vertex Attributes

- Consider the following array of vertex attributes

<table>
<thead>
<tr>
<th>Vertex 1 Attributes</th>
<th>Vertex 2 Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y z r g b s t s t</td>
<td>x y z r g b s t s t</td>
</tr>
<tr>
<td>Position</td>
<td>Position</td>
</tr>
<tr>
<td>Color</td>
<td>Color</td>
</tr>
<tr>
<td>Tex0</td>
<td>Tex0</td>
</tr>
<tr>
<td>Tex1</td>
<td>Tex1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- So we can define attribute positions (per vertex)

```c
#define VERTEX_POS_INDEX 0
#define VERTEX_COLOR_INDEX 1
#define VERTEX_TEXCOORD0_INDEX 2
#define VERTEX_TEXCOORD1_INDEX 3
```
Declaring Array of Vertex Attributes

- Also define number of floats (storage) for each vertex attribute

```plaintext
#define VERTEX_POS_SIZE 3 // x, y and z
#define VERTEX_COLOR_SIZE 3 // r, g and b
#define VERTEX_TEXCOORD0_SIZE 2 // s and t
#define VERTEX_TEXCOORD1_SIZE 2 // s and t

#define VERTEX_ATTRIB_SIZE VERTEX_POS_SIZE + VERTEX_COLOR_SIZE + VERTEX_TEXCOORD0_SIZE + VERTEX_TEXCOORD1_SIZE
```
Declaring Array of Vertex Attributes

- Define offsets (# of floats) of each vertex attribute from beginning

```plaintext
#define VERTEX_POS_OFFSET 0
#define VERTEX_COLOR_OFFSET 3
#define VERTEX_TEXCOORD0_OFFSET 6
#define VERTEX_TEXCOORD1_OFFSET 8
```
Allocating Array of Vertex Attributes

- Allocate memory for entire array of vertex attributes

```c
#define VERTEX_ATTRIB_SIZE VERTEX_POS_SIZE + VERTEX_COLOR_SIZE + \
                           VERTEX_TEXCOORD0_SIZE + \
                           VERTEX_TEXCOORD1_SIZE

float *p = malloc(numVertices * VERTEX_ATTRIB_SIZE * sizeof(float));
```

Allocate memory for all vertices
Specifying Array of Vertex Attributes

- `glVertexAttribPointer` used to specify vertex attributes
- Example: to specify vertex position attribute

```
glVertexAttribPointer(VERTEX_POS_INDEX, VERTEX_POS_SIZE, GL_FLOAT, GL_FALSE, VERTEX_ATTRIB_SIZE * sizeof(float), p);
```

- `glEnableVertexAttribArray(0);`
- do same for normal, tex0 and tex1

<table>
<thead>
<tr>
<th>Position</th>
<th>Color</th>
<th>Tex0</th>
<th>Tex1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
<td>r</td>
</tr>
<tr>
<td>g</td>
<td>b</td>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>s</td>
<td>t</td>
<td>s</td>
<td>t</td>
</tr>
</tbody>
</table>

Data is floats

Position 0
3 floats (x, y, z)

Data should not be normalized

Stride: distance between consecutive vertices

Pointer to data
New Way: Drawing the cube

- Drawing Similar to 2D
  - Move array of 3D mesh vertices to *vertex buffer object*
  - Draw mesh using `glDrawArrays`
Full Example: Rotating Cube

- **Desired** Program behaviour:
  - Draw colored cube
  - Use 3-button mouse to change direction of rotation
  - Use idle function to increment angle of rotation
- **Note:** Default camera?
  - If we don’t set camera, we get a default camera
  - Located at origin and points in the negative z direction
Cube Vertices

// (x,y,z,w) coordinates of the
// vertices of a unit cube centered at origin
// sides aligned with axes

point4 vertices[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 )
};
Colors

// Unique set of RGBA colors that vertices can have

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 Function

// quad generates two triangles (a,b,c) and (a,c,d) for each face and
// assigns colors to the vertices

int Index = 0;  // Index goes from 1 to 6, one per face

void quad( int a, int b, int c, int d )
{
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++
    colors[Index] = vertex_colors[b]; points[Index] = vertices[b]; Index++
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++
    colors[Index] = vertex_colors[d]; points[Index] = vertices[d]; Index++
}
Color Cube

// 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 );
}
void init()
{
    colorcube(); // Generates cube data in application

    // Create a vertex array object (allows us switch between VBOs)
    GLuint vao;
    glGenVertexArrays ( 1, &vao );
    glBindVertexArray ( vao );
// Create and initialize a buffer object and move points
data to GPU

GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER, sizeof(points) +
sizeof(colors), NULL, GL_STATIC_DRAW );
Transfer \texttt{points[]} and \texttt{colors[]} data
Separately using \texttt{glBufferSubData}

\begin{verbatim}
glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
                 sizeof(colors), colors );
\end{verbatim}

// Load shaders and use the resulting shader program
GLuint program = InitShader( "vshader36.glsl", "fshader36.glsl" );
glUseProgram( program );

Initialize vertex and fragment shaders
Initialization IV

// set up vertex arrays

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)) );

theta = glGetUniformLocation( program, "theta" );

Specify vertex data
Connect variable theta in program
To variable in shader
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );

    glUniform3fv( theta, 1, theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );

    glutSwapBuffers();
}

Draw series of triangles forming cube
Mouse Callback

```c
void mouse( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN ) {
        switch( button ) {
        case GLUT_LEFT_BUTTON:    axis = Xaxis;  break;
        case GLUT_MIDDLE_BUTTON:  axis = Yaxis;  break;
        case GLUT_RIGHT_BUTTON:   axis = Zaxis;  break;
        }
    }
}
```

Select axis (x,y,z) to rotate around
Using mouse click
void idle( void )
{
    theta[axis] += 0.01;

    if ( theta[axis] > 360.0 ) {
        theta[axis] -= 360.0;
    }

    glutPostRedisplay();
}

The idle( ) function is called Whenever nothing to do Rotate by theta = 0.01 around axes.
Hidden-Surface Removal

- We want to see only surfaces in front of other surfaces
- OpenGL uses hidden-surface technique called the z-buffer algorithm
- Z-buffer uses distance from viewer (depth) to determine closer objects
- Objects rendered so that only front objects appear in image

Draw face A (front face)
Do not draw faces B and C
Using OpenGL’s z-buffer algorithm

- Z-buffer uses an extra buffer, (the z-buffer), to store depth information as geometry travels down the pipeline
- 3 steps to set up Z-buffer:
  1. In `main.c`
     ```
     glutInitDisplayMode
     (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
     ```
  2. Enabled in `init.c`
     ```
     glEnable(GL_DEPTH_TEST)
     ```
  3. Cleared in the display callback
     ```
     glClear(GL_COLOR_BUFFER_BIT | DEPTH_BUFFER_BIT)
     ```
3D Mesh file formats

- 3D meshes usually stored in 3D file format
- Format defines how vertices, edges, and faces are declared
- Over 400 different file format
- **Polygon File Format (PLY)** used a lot in graphics
- Originally PLY was used to store 3D files from 3D scanner
- We can get PLY models from web to work with
- We will use PLY files in this class
Georgia Tech Large Models Archive

Models

- Stanford Bunny
- Turbine Blade
- Skeleton Hand
- Dragon
- Happy Buddha
- Horse
- Visible Man Skin
- Visible Man Bone
- Grand Canyon
- Puget Sound
- Angel
Stanford 3D Scanning Repository

Lucy: 28 million faces

Happy Buddha: 9 million faces
Sample PLY File

ply
format ascii 1.0
comment this is a simple file
obj_info any data, in one line of free form text
element vertex 3
property float x
property float y
property float z
element face 1
property list uchar int vertex_indices
end_header
-1 0 0
0 1 0
1 0 0
1 0 0
3 0 1 2
References