Computer Graphics (CS 543)
Lecture 4a: Linear Algebra for Graphics (Points, Scalars, Vectors)

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Announcements

- Sample exam 1 will be posted on class website
- Exam 1 next week in class, Sept 26
  - Exam review at the end of today’s class
- **Date change:** Project 2 out next week, due Oct 10
Standard Unit Vectors

Define

\[ \mathbf{i} = (1,0,0) \]

\[ \mathbf{j} = (0,1,0) \]

\[ \mathbf{k} = (0,0,1) \]

So that any vector,

\[ \mathbf{v} = (a, b, c) = a\mathbf{i} + b\mathbf{j} + c\mathbf{k} \]
Cross Product (Vector product)

If

\[ \mathbf{a} = (a_x, a_y, a_z) \quad \text{and} \quad \mathbf{b} = (b_x, b_y, b_z) \]

Then

\[ \mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y) \mathbf{i} - (a_x b_z - a_z b_x) \mathbf{j} + (a_x b_y - a_y b_x) \mathbf{k} \]

Remember using determinant

\[
\begin{vmatrix}
  i & j & k \\
  a_x & a_y & a_z \\
  b_x & b_y & b_z \\
\end{vmatrix}
\]

**Note:** \( \mathbf{a} \times \mathbf{b} \) is perpendicular to \( \mathbf{a} \) and \( \mathbf{b} \)
Cross Product

**Note:** $\mathbf{a} \times \mathbf{b}$ is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$
Cross Product (Vector product)

Calculate \( \mathbf{a} \times \mathbf{b} \) if \( \mathbf{a} = (3,0,2) \) and \( \mathbf{b} = (4,1,8) \)

\[
\mathbf{a} = (3,0,2) \quad \quad \quad \quad \mathbf{b} = (4,1,8)
\]

Using determinant

\[
\begin{vmatrix}
  i & j & k \\
  3 & 0 & 2 \\
  4 & 1 & 8 \\
\end{vmatrix}
\]

Then

\[
\mathbf{a} \times \mathbf{b} = (0 - 2)i - (24 - 8)j + (3 - 0)k
\]

\[
= -2i - 16j + 3k
\]
Normal for Triangle using Cross Product Method

plane \( \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0 \)

\[ \mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0) \]

normalize \( \mathbf{n} \leftarrow \mathbf{n}/|\mathbf{n}| \)

Note that right-hand rule determines outward face
Newell Method for Normal Vectors

- Problems with cross product method:
  - Calculation difficult by hand, tedious
  - If 2 vectors almost parallel, cross product is small
  - Numerical inaccuracy may result

- Proposed by Martin Newell at Utah (teapot guy)
  - Uses formulae, suitable for computer
  - Compute during mesh generation
  - Robust!
Newell Method Example

- Example: Find normal of polygon with vertices
  \( P_0 = (6,1,4) \), \( P_1=(7,0,9) \) and \( P_2 = (1,1,2) \)

- Using simple cross product:
  \[
  ((7,0,9)-(6,1,4)) \times ((1,1,2)-(6,1,4)) = (2,-23,-5)
  \]
Newell Method for Normal Vectors

- Formulae: Normal $\mathbf{N} = (m_x, m_y, m_z)$

\[
m_x = \sum_{i=0}^{N-1} \left( y_i - y_{next(i)} \right) \left( z_i + z_{next(i)} \right)
\]

\[
m_y = \sum_{i=0}^{N-1} \left( z_i - z_{next(i)} \right) \left( x_i + x_{next(i)} \right)
\]

\[
m_z = \sum_{i=0}^{N-1} \left( x_i - x_{next(i)} \right) \left( y_i + y_{next(i)} \right)
\]
Newell Method for Normal Vectors

- Calculate x component of normal

\[ m_x = \sum_{i=0}^{N-1} (y_i - y_{next(i)}) (z_i + z_{next(i)}) \]

\[ m_x = (1)(13) + (-1)(11) + (0)(6) \]
\[ m_x = 13 - 11 + 0 \]
\[ m_x = 2 \]
Newell Method for Normal Vectors

- Calculate $y$ component of normal

\[
m_y = \sum_{i=0}^{N-1} \left( z_i - z_{next(i)} \right) \left( x_i + x_{next(i)} \right)
\]

\[
m_y = (-5)(13) + (7)(8) + (-2)(7)
\]

\[
m_y = -65 + 56 - 14
\]

\[
m_y = -23
\]
Newell Method for Normal Vectors

- Calculate z component of normal

\[ m_z = \sum_{i=0}^{N-1} (x_i - x_{next(i)})(y_i + y_{next(i)}) \]

\[ m_z = (-1)(1) + (6)(1) + (-5)(2) \]
\[ m_z = -1 + 6 - 10 \]
\[ m_z = -5 \]

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P1</td>
<td>7</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P0</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

**Note:** Using Newell method yields same result as Cross product method (2, -23, -5)
Finding Vector Reflected From a Surface

- $a =$ original vector
- $n =$ normal vector
- $r =$ reflected vector
- $m =$ projection of $a$ along $n$
- $e =$ projection of $a$ orthogonal to $n$

Note: $\Theta_1 = \Theta_2$

$e = a - m$
$r = e - m$
$\Rightarrow r = a - 2m$
Forms of Equation of a Line

- Two-dimensional forms of a line
  - **Explicit:** $y = mx + h$
  - **Implicit:** $ax + by + c = 0$
  - **Parametric:**
    \[x(\alpha) = \alpha x_0 + (1-\alpha)x_1\]
    \[y(\alpha) = \alpha y_0 + (1-\alpha)y_1\]

- Parametric form of line
  - More robust and general than other forms
  - Extends to curves and surfaces
Convexity

- An object is *convex* iff for any two points in the object all points on the straight line between these points are also in the object.
References

3D Applications

- **2D points**: (x, y) coordinates
- **3D points**: have (x, y, z) coordinates
Setting up 3D Applications: Main Steps

- Programming 3D similar to 2D
  1. Load representation of 3D object into data structure
  2. Draw 3D object
  3. **Set up Hidden surface removal:** Correctly determine order in which primitives (triangles, faces) are rendered (e.g. Blocked faces **NOT** drawn)

Each vertex has (x,y,z) coordinates. Store as \texttt{vec3 NOT vec2}
3D Coordinate Systems

- Vertex \((x, y, z)\) positions specified on coordinate system
- OpenGL uses **right hand coordinate system**

**Right hand coordinate system**
- Tip: sweep fingers \(x-y\): thumb is \(z\)

**Left hand coordinate system**
- Not used in OpenGL
Generating 3D Models: GLUT Models

- Make GLUT 3D calls in **OpenGL program** to generate vertices describing different shapes (Restrictive?)

- Two types of GLUT models:
  - Wireframe Models
  - Solid Models
3D Modeling: GLUT Models

- **Basic Shapes**
  - **Cone:** `glutWireCone()`, `glutSolidCone()`
  - **Sphere:** `glutWireSphere()`, `glutSolidSphere()`
  - **Cube:** `glutWireCube()`, `glutSolidCube()`

- **More advanced shapes:**
  - **Newell Teapot:** (symbolic)
  - Dodecahedron, Torus
3D Modeling: GLUT Models

- Glut functions under the hood
  - generate sequence of points that define a shape
  - Generated vertices and faces passed to OpenGL for rendering
- **Example:** `glutWireCone` generates sequence of vertices, and faces defining **cone** and connectivity
Polygonal Meshes

- Modeling with GLUT shapes (cube, sphere, etc) too restrictive
- Difficult to approach realism. E.g. model a horse
- Preferred way is using polygonal meshes:
  - Collection of polygons, or faces, that form “skin” of object
  - More flexible, represents complex surfaces better
  - Examples:
    - Human face
    - Animal structures
    - Furniture, etc

Each face of mesh is a polygon
Polygonal Meshes

- Mesh = sequence of polygons forming thin skin around object
- OpenGL good at drawing polygons, triangles
- Meshes now standard in graphics
- Simple meshes exact. (e.g. barn)
- Complex meshes approximate (e.g. human face)
Same Mesh at Different Resolutions

Original: 424,000 triangles

60,000 triangles (14%).

1000 triangles (0.2%).

(courtesy of Michael Garland and Data courtesy of Iris Development.)
Representing a Mesh

- Consider a mesh

- There are 8 vertices and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges (shown in orange)
- Each vertex has a location $v_i = (x_i, y_i, z_i)$
Simple Representation

- Define each polygon by (x,y,z) locations of its vertices
- OpenGL code

```c
vertex[i]   = vec3(x1, y1, z1);
vertex[i+1] = vec3(x6, y6, z6);
vertex[i+2] = vec3(x7, y7, z7);
i+=3;
```
Issues with Simple Representation

- **Declaring face f1**
  
  ```
  vertex[i] = vec3(x1, y1, z1);
  vertex[i+1] = vec3(x7, y7, z7);
  vertex[i+2] = vec3(x8, y8, z8);
  vertex[i+3] = vec3(x6, y6, z6);
  ```

- **Declaring face f2**
  
  ```
  vertex[i] = vec3(x1, y1, z1);
  vertex[i+1] = vec3(x2, y2, z2);
  vertex[i+2] = vec3(x7, y7, z7);
  ```

- **Inefficient and unstructured**
  
  - **Repeats**: vertices v1 and v7 repeated while declaring f1 and f2
  - Shared vertices shared declared multiple times
  - Delete vertex? Move vertex? Search for all occurrences of vertex
Geometry vs Topology

- **Geometry**: $(x, y, z)$ locations of the vertices
- **Topology**: How vertices and edges are connected

Good data structures separate **geometry** from **topology**

- **Example**:
  - A polygon is **ordered list** of vertices
  - An edge connects successive pairs of vertices
- Topology holds even if geometry changes (vertex moves)

Example: even if we move $(x, y, z)$ location of $v_1$, $v_1$ still connected to $v_6$, $v_7$ and $v_2$
Polygon Traversal Convention

- **Convention**: traverse vertices **counter-clockwise** around normal
- Focus on direction of traversal
  - Orders \( \{ v_1, v_0, v_3 \} \) and \( \{ v_3, v_2, v_1 \} \) are same (ccw)
  - Order \( \{ v_1, v_2, v_3 \} \) is different (clockwise)
- **Normal vector**: Direction each polygon is facing
**Vertex Lists**

- **Vertex list**: \((x,y,z)\) of vertices (its geometry) are put in array
- Use pointers from vertices into vertex list
- **Polygon list**: vertices connected to each polygon (face)

**Topology** example: Polygon P1 of mesh is connected to vertices \((v1,v7,v6)\)

**Geometry** example: Vertex v7 coordinates are \((x7,y7,z7)\). Note: If v7 moves, changed once in vertex list
Vertex List Issue: Shared Edges

- Vertex lists draw filled polygons correctly
- If each polygon is drawn by its edges, shared edges are drawn twice

**Alternatively:** Can store mesh by *edge list*
Edge List

Simply draw each edges once

**E.g.** e1 connects v1 and v6

<table>
<thead>
<tr>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
<th>e6</th>
<th>e7</th>
<th>e8</th>
<th>e9</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x1 y1 z1
x2 y2 z2
x3 y3 z3
x4 y4 z4
x5 y5 z5
x6 y6 z6
x7 y7 z7
x8 y8 z8

**Note** polygons are not represented
Vertex Attributes

- Vertices can have attributes
  - Position \((x, y, z)\) E.g. \((20, 12, 18)\)
  - Color \((R,G,B)\) E.g. \((1,0,0)\) or (red)
  - Normal \((x,y,z)\)
  - Texture coordinates
Vertex Attributes

- Store vertex attributes in **single** Array (array of structures)
- **Later:** pass array to OpenGL, specify attributes, order, position using `glVertexAttribAttribPointer`

### Vertex 1 Attributes
- \((12, 6, 15)\)
- \((18, 34, 6)\)
- \((20, 12, 18)\)

### Vertex 2 Attributes
- Position
- Color
- Tex0
- Tex1
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>Position</td>
<td>Color</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>z</td>
<td>r</td>
</tr>
<tr>
<td>r</td>
<td>g</td>
</tr>
<tr>
<td>g</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>s</td>
</tr>
<tr>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>s</td>
</tr>
<tr>
<td>s</td>
<td>t</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

```c
#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

```c
#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

```
#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

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

- **do same for normal, tex0 and tex1**

#### Data

<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>s</td>
<td>t</td>
<td>s</td>
<td>t</td>
</tr>
</tbody>
</table>

- Data should not be normalized
- Data is floats
- Stride: distance between consecutive vertices
- Pointer to data
Full Example: Rotating Cube in 3D

- **Desired Program behaviour:**
  - Draw colored cube
  - Continuous rotation about X, Y or Z axis
    - Idle function called repeatedly when nothing to do
    - Increment angle of rotation in idle function
  - Use 3-button mouse to change direction of rotation
    - Click left button -> rotate cube around X axis
    - Click middle button -> rotate cube around Y axis
    - Click right button -> rotate cube around Z axis

- **Use 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

Declare array of \((x,y,z,w)\) vertex positions for a unit cube centered at origin (Sides aligned with axes)

```plaintext
point4 vertices[8] = {
  0 point4( -0.5, -0.5,  0.5, 1.0 ),
  1 point4( -0.5,  0.5,  0.5, 1.0 ),
  2 point4(  0.5,  0.5,  0.5, 1.0 ),
  3 point4(  0.5, -0.5,  0.5, 1.0 ),
  4 point4( -0.5, -0.5, -0.5, 1.0 ),
  5 point4( -0.5,  0.5, -0.5, 1.0 ),
  6 point4(  0.5,  0.5, -0.5, 1.0 ),
  7 point4(  0.5, -0.5, -0.5, 1.0 )
};
```

Declare array of vertex colors (set of RGBA colors vertex can have)

```plaintext
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
};
```
// generate 6 quads,
// sides of cube

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 );
}

point4 vertices[8] = {
    0 point4( -0.5, -0.5,  0.5, 1.0 ),
    1 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 ),
    4 point4( -0.5, -0.5, -0.5, 1.0 ),
    5 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 )
};

Function quad is
Passed vertex indices
Quad Function

```c
int Index = 0;  // Index goes 0 to 5, one per vertex of 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++;
}
```

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

Points[ ] array to be
Sent to GPU

Read from appropriate index
of unique positions declared
Initialization I

```c
void init()
{
    colorcube(); // Generates cube data in application using quads

    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays ( 1, &vao );
    glBindVertexArray ( vao );

    // Create a buffer object and move 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 );
}
```

Points[ ] array of vertex positions sent to GPU

colors[ ] array of vertex colors sent to GPU
Initialization II

Send `points[ ]` and `colors[ ]` data to GPU separately using `glBufferSubData`

\[
\text{glBufferSubData( GL\_ARRAY\_BUFFER, 0, sizeof(points), points );}
\]
\[
\text{glBufferSubData( GL\_ARRAY\_BUFFER, sizeof(points), sizeof(colors), colors );}
\]

// Load vertex and fragment shaders and use the resulting shader program
\[
\text{GLuint program = InitShader( "vshader36.glsl", "fshader36.glsl" );}
\]
\[
\text{glUseProgram( program );}
\]
// 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)));

Want to Connect rotation variable theta in program to variable in shader

theta = glGetUniformLocation(program, "theta");
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );

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

    glutSwapBuffers();
}
enum { Xaxis = 0, Yaxis = 1, Zaxis = 2, NumAxes = 3 };
Idle Callback

void idle( void )
{
    theta[axis] += 0.01;

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

    glutPostRedisplay();
}

void main( void ){
    ........

    glutIdleFunc( idle );
    ........
}

The idle() function is called whenever nothing to do
Use it to increment rotation angle in steps of theta = 0.01 around currently selected axis

Note: still need to:
• Apply rotation by (theta) in shader
References