EECS 487: Interactive Computer Graphics

Lecture 12:

- Vertex passing: triangle strip/fan, vertex array, drawing modes
- From 3D to 2D Overview
- Viewing transform

OpenGL: Drawing Triangles

You can draw multiple triangles between
\( \text{glBegin(GL_TRIANGLES)} \) and \( \text{glEnd();} \):

```c
float v1[3], v2[3], v3[3], v4[3];
...
glBegin(GL_TRIANGLES);
  glVertex3fv(v1);
  glVertex3fv(v2);
  glVertex3fv(v3);
  glVertex3fv(v1);
  glVertex3fv(v3);
  glVertex3fv(v4);
glEnd();
```

Each triangle sent to the rendering pipeline as 3 vertices at a time ⇒ vertex duplication, not efficient
- each must be transformed and lit
- fewer vertices = faster drawing

OpenGL: Triangle Strips

An OpenGL triangle strip primitive reduces this redundancy by sharing vertices:

```c
glBegin(GL_TRIANGLE_STRIP);
  glVertex3fv(v1);
  glVertex3fv(v2);
  glVertex3fv(v3); // T1
  glVertex3fv(v4);
  glVertex3fv(v5); // T2
  glVertex3fv(v6); // T3
  glEnd();
```

- triangle 1 is \( v_1, v_2, v_3 \)
- triangle 2 is \( v_3, v_2, v_4 \)
  (why not \( v_2, v_3, v_4 \)?)
- triangle 3 is \( v_3, v_4, v_5 \)
- triangle 4 is \( v_5, v_4, v_6 \)

When looking at the front side, the vertices go counterclockwise (right-hand rule)
- \( n \) odd ⇒ \( T_n; n, n+1, n+2 \)
- \( n \) even ⇒ \( T_n; n+1, n, n+2 \)
- \( n \) starts at 1

Triangle Strips

Without strips:
- 8 triangles * 3 vertices = 24 vertices

With strips:
- use 1 vertex per triangle instead of 3
  - startup cost \( v_1, v_2 \), then
    \( v_3(T_1), v_4(T_2), v_5(T_3), v_6(T_4), v_7(T_5), v_8(T_6), v_9(T_7), v_{10}(T_8) \)
- total 10 vertices instead of 24
  - \( 10/8 = 1.125 \) vertices/triangle
  - \( 100\times10/24 = 37.5\% \) less data

We can expect the geometry stage to run almost 3 times faster!
How to Create Triangle Strips from a 3D Model?

Manually
• only doable for small models, and not fun...

Or write your own program:
• need to know triangle’s neighbors
• it’s quite simple to make a working strip creator
• to make a really good one is more work

Or use nvidia’s NVTriStrip (on the web)

OpenGL: Triangle Fan

The GL_TRIANGLE_FAN primitive is another way to eliminate vertex duplication (also 1 vertex per triangle):

```c
glBegin(GL_TRIANGLE_FAN);
glVertex3fv(v0); // start with central point
glVertex3fv(v1); // build triangles around it
glVertex3fv(v2); // T_0
glVertex3fv(v3); // T_1
glVertex3fv(v4); // T_2
glVertex3fv(v5); // T_3
glEnd();
```

Vertex Arrays

Even with triangle strips, passing each vertex to OpenGL requires a separate function call

Vertex arrays allow for passing an array of vertices to OpenGL with a constant number of function calls
• store vertex data in triangle strip sequentially in application/client-side memory
• pass pointer to this memory to the API
• the API copies the data from memory to GPU/server

Vertex Array

```
GL_VERTEX_ARRAY vertices[]
GL_COLOR_ARRAY color[]

App/Client Memory

system bus/network

Graphics Card/Server Memory

Vertex attribute data read from client memory to server memory whenever glDraw*() is called
```
**Vertex Arrays**

Enable vertex array in app memory:
- `glEnableClientState(GL_VERTEX_ARRAY)`

Next, pass array to OpenGL, specifying data format:
- `glVertexPointer()`

Array can be accessed in three ways (array must be in scope when calling these):
- randomly: `glArrayElement()`
- sequentially: `glDrawArrays(...)`, with multiple triangle strips, e.g., in a sphere, no vertex sharing across strips, **duplicated vertices** must be enumerated in array
- indexed: `glDrawElements(...)`, **duplicated vertices** listed once in array

**Vertex Arrays Example**

```c
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
```

Using the Array

With random access (in display list):
```c
glBegin(GL_TRIANGLES);
glArrayElement(1);
glArrayElement(0);
glArrayElement(2);
glEnd();
```

With indexed access (not between `glBegin`/`glEnd`):
```c
unsigned char indices[] = { 1, 0, 2 };
```
```c
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE, indices);
```

**Other Vertex Attributes**

Aside from specifying vertex coordinates, you can also specify the following attribute arrays, with 1-to-1 mapping to the vertex array:
- `glNormalPointer()`: vertex normal array
- `glColorPointer()`: vertex RGB color array
- `glIndexPointer()`: indexed vertex color array
- `glTexCoordPointer()`: texture coordinates array
- `glEdgeFlagPointer()`: array indicating boundary vertices

Note: corresponding client-state must be enabled:
- `GL_NORMAL_ARRAY`, `GL_COLOR_ARRAY`, etc.
Drawing Modes

Immediate drawing mode: graphics system does not store drawn primitives
• `glVertex()`: vertices streamed through to display as soon as specified
• vertex array: vertices copied from client state to graphics card/server as needed (`glDrawElements()` may cache vertices)

Often we want to draw the same object several times, perhaps transformed...inefficient to copy the same vertices multiple times, use retained drawing mode (display list, OpenGL 2.1) or vertex buffer object (OpenGL 3.0+)

Getting from 3D to 2D

Given a 3D scene, how do we get it onto a 2D screen?
• specify 3D location of all points on the object
• map these points to locations on 2D device monitor

Various coordinate systems used to accomplish this

Coordinate System Relationships

EECS 487: Interactive Computer Graphics

Lecture 12:
• Vertex passing: triangle strip/fan, vertex array, drawing modes
• From 3D to 2D Overview
• Viewing transform
Device Coordinate

Image in clipping window mapped onto viewport area

Location of a pixel in a viewport is expressed in device coordinates \((x, y)\)

Device coordinates are:
- integers
- resolution dependent

![Device Coordinate Diagram](image)

Normalized Device Coordinate

Normalized device coordinate (NDC):
- location of a pixel expressed in terms of percentages of image size
- resolution independent
- \((z\)-values retained\)

![Normalized Device Coordinate Diagram](image)

Projection Transforms

**View volume**: part of 3D scene we want to draw, constrained by front, back, and side clipping planes
- drawing window is of finite size:
  - we can only store a finite number of pixels
  - and a discrete, finite range of depths
  - like color, only have a fixed number of depth bits at each pixel
  - points too close or too far away will not be drawn
- for parallel projection: rectangular box
- for perspective projection: truncated pyramid (**frustum**)

Projection transforms project the view volume onto the viewing plane

![Projection Transforms Diagram](image)

Canonical View Volume

A cube centered at the origin, aligned with the axes, spanning \((-1, -1, -1)\) to \((1,1,1)\)

A 3D version of NDC
- decouples projection from window/screen sizes
- parallel sides and equal dimensions of the cvv make many operations, e.g., clipping, easier (cvv is thus a.k.a. clip coordinates)

![Canonical View Volume Diagram](image)
From View Frustum to Screen Space

Perspective transform:
1. perspective project view frustum to orthographic space
2. orthographic/parallel project to canonical view volume
3. map canonical view volume to NDC for display

Viewport transform/screen mapping:
• map NDC to viewport

Illustrated in the next slide ...

OpenGL States:
CTMs

Modified with:
glLoadIdentity()
gluLookAt()
gluScale() etc.

Model/Transform matrix stack (32 x 4 matrices)

Modified with:
glLoadIdentity()
gluOrtho2D()

Where in the Pipeline?

3D Geometric Pipeline

Where in the Pipeline?
**EECS 487: Interactive Computer Graphics**

Lecture 12:
- Vertex passing: triangle strip/fan, vertex array, drawing modes
- From 3D to 2D Overview
- Viewing transform

**Viewing Transform**

Transform the scene such that the camera is at the origin; simplifies projection, visibility and clipping determination, lighting

**Camera Coordinate Frame**

Given, in world coordinates:
- camera/eye location (e)
- lookat point (c), where camera is pointed to (centered at)
  - or view/gaze direction (g): line from eye to lookat point (c – e)
- and an up-vector (t): a vector pointing up from the camera/eye

How do we construct a camera coordinate frame?

**Viewing Transform**

First construct a **camera coordinate frame**

What is a coordinate frame?

A set of 3 vectors \((u, v, w)\) and an origin \(o\) such that:

\[
\begin{align*}
\|u\| = \|v\| = \|w\| &= 1 \\
u \cdot v &= v \cdot w = u \cdot w = 0 \\
w &= u \times v
\end{align*}
\]

and for any point \(p\):

\[
p = o + (p \cdot u)u + (p \cdot v)v + (p \cdot w)w
\]
**Camera Coordinate Frame**

Given \( e, c, \) and \( t \) of the camera in world coordinates,
- we can use \( e \) as the origin \((o)\), and
- would like to use \( g \) as \( w \) and \( v \) as \( t \)
- but \( g \) and \( t \) are neither orthogonal nor of unit length!
- furthermore, we need a \( u \)

(Called Gram-Schmidt Orthonormalization)

\[
\begin{align*}
\mathbf{w} &= -(\mathbf{c} - \mathbf{e})/||\mathbf{c} - \mathbf{e}|| \\
\mathbf{u} &= (\mathbf{t} \times \mathbf{w})/||\mathbf{t} \times \mathbf{w}|| \\
\mathbf{v} &= \mathbf{w} \times \mathbf{u}
\end{align*}
\]

**Viewing Transform Implementation**

Given \( e, c, t, \) and \( u, v, w \) as computed, how do we transform from world to eye coords?

1. Translate by \(-e\), bring camera’s viewpoint to origin
2. Rotate \( w \) to the \( z \)-axis \((z = [0,0,1])\):
   - what are the rotation axis and angle?
   - axis: \( a = (w \times z) / ||w \times z|| \)
   - angle: \( \cos \theta = \mathbf{w} \cdot \mathbf{z} \) and \( \sin \theta = ||w \times z|| \)
   - \( \text{glRotate}(\theta, ax, ay, az) \)
   - \( u \) and \( v \) are now on the \( x-y \) plane
3. Rotate \( v \) by \( \alpha \) about the \( z \)-axis, to get \( v \) parallel to the \( y \)-axis and \( u \parallel \) to \( x \)
   \[
p' = \mathbf{Vp} = R(\alpha, z)R(\theta, a)T(-e)p
\]
Change of Orthonormal Basis

Given coordinate frames $\text{xyz}$ (world) and $\text{uvw}$ (eye) and point $p = (p_x, p_y, p_z)$

Find: $p = (p_u, p_v, p_w)$ by transforming the coordinate frame

(Easier to visualize in 2D!)

Change of Orthonormal Basis

Inverse transformation from $x, y, z$ to $u, v, w$

column space: $\begin{bmatrix} u & v & w \end{bmatrix}$

row space: $\begin{bmatrix} s & t & r \end{bmatrix}$

Expressing the $x, y, z$ (world) bases in terms of $u, v, w$ (eye) (coordinates are length of projections!):

$x = (x \cdot u) u + (x \cdot v) v + (x \cdot w) w$

$y = (y \cdot u) u + (y \cdot v) v + (y \cdot w) w$

$z = (z \cdot u) u + (z \cdot v) v + (z \cdot w) w$
Change of Orthonormal Basis

\[ x = (x \cdot u) u + (x \cdot v) v + (x \cdot w) w \]
\[ y = (y \cdot u) u + (y \cdot v) v + (y \cdot w) w \]
\[ z = (z \cdot u) u + (z \cdot v) v + (z \cdot w) w \]

Substitute into equation for \( p \):
\[ p = (p_x, p_y, p_z) = p_x x + p_y y + p_z z \]

Rewrite:
\[ p = \begin{bmatrix} p_x (x \cdot u) + p_y (y \cdot u) + p_z (z \cdot u) \\ p_x (x \cdot v) + p_y (y \cdot v) + p_z (z \cdot v) \\ p_x (x \cdot w) + p_y (y \cdot w) + p_z (z \cdot w) \end{bmatrix} \]

Change of Basis to Eye Coordinate Frame

Inverse transform from \( x, y, z \) to \( u, v, w \)

\[ \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} \]

\[ \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \]

Translation to eye and transformed to \( u, v, w \) basis:
\[ M_{\text{world} \rightarrow \text{eye}} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \end{bmatrix} \]
\[ = \begin{bmatrix} u_x & u_y & u_z & -u_x e_x - u_y e_y - u_z e_z \\ v_x & v_y & v_z & -v_x e_x - v_y e_y - v_z e_z \\ w_x & w_y & w_z & -w_x e_x - w_y e_y - w_z e_z \end{bmatrix} \]

And to go all the way from world to cvv:
\[ p_{\text{canonical}} = M_{\text{world} \rightarrow \text{canonical}} D_{\text{world}} \]
**Viewing Transform in OpenGL 2.1**

Position the camera/eye in the scene

```c
    gluLookAt(eyeX,eyeY,eyeZ,
              centerX,centerY,centerZ,
              upX,upY,upZ);
```

To “fly through” a scene, change viewing transform and redraw scene

- moving camera is equivalent to moving every object in the world relative to a stationary camera

`gluLookAt()` operates on the ModelView matrix just like any modeling transform:

- must come “before” in code, after in action” to other transforms

**Viewing Transforms**

**Example:**

```c
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // viewing transform, comes before
    // model transform, applied last
    gluLookAt(0.0,0.0,5.0, // camera at (0,0,5)
              0.0,0.0,0.0, // gazing at (0,0,0)
              0.0,1.0,0.0); // up is y-axis

    // model transform
    glRotated(-20.0, 0.0,1.0,0.0);

    // then draw
    glutSolidTeapot(1.0);
```