Other Camera Controls

- The LookAt function is only for positioning camera
- Other ways to specify camera position/movement
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles
Flexible Camera Control

- Sometimes, we want camera to move
- Like controlling an airplane’s orientation
- Adopt aviation terms:
  - **Pitch**: nose up-down
  - **Roll**: roll body of plane
  - **Yaw**: move nose side to side
Yaw, Pitch and Roll Applied to Camera

- Similarly, yaw, pitch, roll with a camera
Flexible Camera Control

- Create a **camera** class

```cpp
class Camera{
  private:
    Point3 eye;
    Vector3 u, v, n;.... etc
}
```

- Camera functions to specify pitch, roll, yaw. E.g

```cpp
cam.slide(1, 0, 2); // slide camera backward 2 and right 1
cam.roll(30);      // roll camera 30 degrees
cam.yaw(40);       // yaw it 40 degrees
cam.pitch(20);     // pitch it 20 degrees
```
Recall: Final LookAt Matrix

- Slide along u, v or n
- Changes eye position
- Changes these components

\[
\begin{pmatrix}
ux & uy & uz \\
vx & vy & vz \\
xn & yn & nz \\
0 & 0 & 0
\end{pmatrix}
- e \cdot u
- e \cdot v
- e \cdot n
1
\]

- Pitch, yaw, roll rotates u, v or n
- Changes u, v or n

slide

roll
Implementing Flexible Camera Control

- Camera class: maintains current \((u,v,n)\) and eye position

```cpp
class Camera
private:
    Point3 eye;
    Vector3 u, v, n;.... etc
```

- User inputs desired roll, pitch, yaw angle or slide
  1. Roll, pitch, yaw: calculate modified vector \((u', v', n')\)
  2. Slide: Calculate new eye position
  3. Update `lookAt` matrix, Load it into CTM
Example: Camera Slide

- Recall: the axes are unit vectors
- User changes eye by $\text{del}U$, $\text{del}V$ or $\text{del}N$
- $\text{eye} = \text{eye} + \text{changes} (\text{del}U, \text{del}V, \text{del}N)$
- Note: function below combines all slides into one
  
  E.g moving camera by $D$ along its u axis $= \text{eye} + Du$

```cpp
void camera::slide(float delU, float delV, float delN)
{
    eye.x += delU*u.x + delV*v.x + delN*n.x;
    eye.y += delU*u.y + delV*v.y + delN*n.y;
    eye.z += delU*u.z + delV*v.z + delN*n.z;
    setModelViewMatrix( );
}
```
Load Matrix into CTM

```cpp
void Camera::setModelViewMatrix(void)
{
    // load modelview matrix with camera values
    mat4 m;
    Vector3 eVec(eye.x, eye.y, eye.z); // eye as vector
    m[0] = u.x; m[4] = u.y; m[8] = u.z; m[12] = -dot(eVec, u);
    m[2] = n.x; m[6] = n.y; m[10] = n.z; m[14] = -dot(eVec, n);
    CTM = m; // Finally, load matrix m into CTM Matrix
}
```

- Slide changes `eVec`,
- roll, pitch, yaw, change `u, v, n`
- Call `setModelViewMatrix` after slide, roll, pitch or yaw
Example: Camera Roll

```cpp
void Camera::roll(float angle)
{
    // roll the camera through angle degrees
    float cs = cos(3.142/180 * angle);
    float sn = sin(3.142/180 * angle);
    Vector3 t = u; // remember old u
    u.set(cs*t.x - sn*v.x, cs*t.y - sn.v.y, cs*t.z - sn.v.z);
    v.set(sn*t.x + cs*v.x, sn*t.y + cs.v.y, sn*t.z + cs.v.z)
    setModelViewMatrix( );
}
```
Recall: 3D Viewing and View Volume

Previously: Lookat( ) to set camera position

Now: Set view volume
Recall: Different View Volume Shapes

- Different view volume => different look
- **Foreshortening**? Near objects bigger
View Volume Parameters

- Need to set view volume parameters
  - **Projection type:** perspective, orthographic, etc.
  - Field of view and aspect ratio
  - Near and far clipping planes
Field of View

- View volume parameter
- Determines how much of world in picture (vertically)
- Larger field of view = smaller objects drawn
Near and Far Clipping Planes

- Only objects between near and far planes drawn
Viewing Frustum

- Near plane + far plane + field of view = Viewing Frustum
- Objects outside the frustum are clipped
Setting up View Volume/Projection Type

- Previous OpenGL projection commands **deprecated**!!
  - Perspective view volume/projection:
    - `gluPerspective(fovy, aspect, near, far)` or
    - `glFrustum(left, right, bottom, top, near, far)`
  - Orthographic:
    - `glOrtho(left, right, bottom, top, near, far)`
- Useful functions, so we implement similar in `mat.h`:
  - `Perspective(fovy, aspect, near, far)` or
  - `Frustum(left, right, bottom, top, near, far)`
  - `Ortho(left, right, bottom, top, near, far)`

What are these arguments? Next!
Perspective(fovy, aspect, near, far)

- Aspect ratio used to calculate window width

Aspect = w / h
Frustum(left, right, bottom, top, near, far)

- Can use **Frustum()** in place of **Perspective()**
- Same view volume **shape**, different **arguments**

```
x  y  z
left  top
bottom  right
```

**near** and **far** measured **from** camera
Ortho(left, right, bottom, top, near, far)

- For orthographic projection

near and far measured from camera
Example Usage:
Setting View Volume/Projection Type

```c
void display()
{
    // clear screen
    glClear(GL_COLOR_BUFFER_BIT);

    // Set up camera position
    LookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);
    eye at up

    // set up perspective transformation
    Perspective(fovy, aspect, near, far);

    // draw something
    display_all();    // your display routine
}
```
Demo

- Nate Robbins demo on projection
Perspective Projection

- After setting view volume, then projection transformation
- Projection?
  - **Classic:** Converts 3D object to corresponding 2D on screen
  - How? Draw line from object to projection center
  - Calculate where each intersects projection plane
Orthographic Projection

- How? Draw parallel lines from each object vertex
- The projection center is at infinite
- In short, use (x,y) coordinates, just drop z coordinates
The Problem with Classic Projection

- Keeps \((x,y)\) coordinates for drawing, drops \(z\)
- We may need \(z\). Why?

\[
\begin{align*}
    x_p &= x \\
    y_p &= y \\
    z_p &= 0
\end{align*}
\]

Classic Projection Loses \(z\) value
Normalization: Keeps z Value

- Most graphics systems use *view normalization*
- **Normalization**: convert all other projection types to orthogonal projections with the *default view volume*
Computer Graphics (CS 4731)
Lecture 15: Projection (Part 2): Derivation

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Parallel Projection

- **normalization** ⇒ find 4x4 matrix to transform user-specified view volume to **canonical view volume** (cube)

\[
glOrtho(left, right, bottom, top, near, far)
\]
Parallel Projection: Ortho

- Parallel projection: 2 parts
  1. **Translation**: centers view volume at origin
Parallel Projection: Ortho

2. **Scaling**: reduces user-selected cuboid to canonical cube (dimension 2, centered at origin)
Parallel Projection: Ortho

- Translation lines up midpoints: E.g. midpoint of $x = (\text{right} + \text{left})/2$
- Thus translation factors:
  - $-(\text{right} + \text{left})/2, -(\text{top} + \text{bottom})/2, -(\text{far} + \text{near})/2$
- Translation matrix:

$$
\begin{bmatrix}
1 & 0 & 0 & -(\text{right} + \text{left})/2 \\
0 & 1 & 0 & -(\text{top} + \text{bottom})/2 \\
0 & 0 & 1 & -(\text{far} + \text{near})/2 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$
Parallel Projection: Ortho

- Scaling factor: ratio of ortho view volume to cube dimensions
- Scaling factors: \( \frac{2}{\text{right - left}}, \frac{2}{\text{top - bottom}}, \frac{2}{\text{far - near}} \)
- Scaling Matrix M2:

\[
\begin{bmatrix}
\frac{2}{\text{right - left}} & 0 & 0 & 0 \\
0 & \frac{2}{\text{top - bottom}} & 0 & 0 \\
0 & 0 & \frac{2}{\text{far - near}} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Parallel Projection: Ortho

Concatenating **Translation** x **Scaling**, we get Ortho Projection matrix

\[
\begin{pmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & 0 \\
0 & \frac{2}{\text{top} - \text{bottom}} & 0 & 0 \\
0 & 0 & \frac{2}{\text{far} - \text{near}} & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
1 & 0 & 0 & -(\text{right} + \text{left})/2 \\
0 & 1 & 0 & -(\text{top} + \text{bottom})/2 \\
0 & 0 & 1 & -(\text{far} + \text{near})/2 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
P = ST =
\begin{pmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} - \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bottom}} & 0 & -\frac{\text{top} + \text{bottom}}{\text{top} + \text{bottom}} \\
0 & 0 & \frac{2}{\text{far} - \text{near}} & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Final Ortho Projection

- Set $z = 0$
- Equivalent to the homogeneous coordinate transformation

\[
M_{\text{orth}} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- Hence, general orthogonal projection in 4D is

\[
P = M_{\text{orth}}ST
\]
Perspective Transformation

- We want to transform viewing frustum volume into canonical view volume

 Canonical View Volume

(1, 1, -1)

(-1, -1, 1)

Canonical View Volume
Perspective Projection Matrix

Derivation skipped!

\[
\begin{pmatrix}
\frac{2N}{x_{\text{max}} - x_{\text{min}}} & 0 & \frac{\text{right} + \text{left}}{\text{top} + \text{bottom}} & 0 \\
0 & \frac{2N}{\text{top} - \text{bottom}} & \frac{\text{right} - \text{left}}{\text{top} - \text{bottom}} & 0 \\
0 & 0 & \frac{-\left(F + N\right)}{F - N} & -2FN \\
0 & 0 & \frac{F - N}{F - N} & 0
\end{pmatrix}
\]

\(\text{glFrustum(left, right, bottom, top, N, F)}\)  \(N = \text{near plane, } F = \text{far plane}\)
Geometric Nature of Perspective Transform

a) Lines through eye map into lines parallel to z axis after transform
b) Lines perpendicular to z axis map to lines perp to z axis after transform
Implementation

- Set modelview and projection matrices in application program
- Pass matrices to shader

```c
void display(){
    ....
    model_view = LookAt(eye, at, up);
    projection = Ortho(left, right, bottom, top, near, far);

    // pass model_view and projection matrices to shader
    glUniformMatrix4fv(matrix_loc, 1, GL_TRUE, model_view);
    glUniformMatrix4fv(projection_loc, 1, GL_TRUE, projection);
    ....
}
```
Implementation

- And the corresponding shader

```glsl
in vec4 vPosition;
in vec4 vColor;
Out vec4 color;
uniform mat4 model_view;
Uniform mat4 projection;

void main()
{
    gl_Position = projection * model_view * vPosition;
    color = vColor;
}
```
References