Finding Vector Reflected From a Surface

- \( \mathbf{a} = \) original vector
- \( \mathbf{n} = \) normal vector
- \( \mathbf{r} = \) reflected vector
- \( \mathbf{m} = \) projection of \( \mathbf{a} \) along \( \mathbf{n} \)
- \( \mathbf{e} = \) projection of \( \mathbf{a} \) orthogonal to \( \mathbf{n} \)

\[
\Theta_1 = \Theta_2 \\
e = \mathbf{a} - \mathbf{m} \\
r = \mathbf{e} - \mathbf{m} \\
\Rightarrow r = \mathbf{a} - 2\mathbf{m}
\]
Lines

- Consider all points of the form
  - $P(\alpha) = P_0 + \alpha \mathbf{d}$
  - **Line**: Set of all points that pass through $P_0$ in direction of vector $\mathbf{d}$
Parametric Form

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

- 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 line segment between these points are also in the object.

![Convex](convex.png) ![Not Convex](not_convex.png)
Curves and Surfaces

- **Curves**: 1-parameter non-linear functions of the form \( P(\alpha) \)
- **Surfaces**: two-parameter functions \( P(\alpha, \beta) \)
  - Linear functions give planes and polygons
3D Viewing?

- Objects **inside** view volume show up on screen
- Objects outside view volume **clipped!**

1. Set camera position

2. Set view volume
   (3D region of interest)
Different View Volume Shapes

- Different view volume => different look
- **Foreshortening?** Near objects bigger
  - Perspective projection has **foreshortening**
  - Orthogonal projection: no foreshortening
The World Frame

- Objects/scene initially defined in **world frame**
- Objects positioned, transformations (translate, scale, rotate) applied to objects in **world frame**
Camera Frame

- More natural to describe object positions relative to camera (eye)
- Think about
  - Our view of the world
  - First person shooter games
Camera Frame

- **Viewing**: After user sets camera (eye) position, represent objects in *camera frame* (origin at eye position)
- **Viewing transformation**: Changes object positions from world frame to positions in camera frame using **model-view matrix**
Default OpenGL Camera

- Initially Camera at origin: object and camera frames same
- Camera located at origin and points in negative z direction
- Default view volume is cube with sides of length 2

![Diagram showing default view volume, clipped out, and projection plane at z=0]
Moving Camera Frame

default frames

Same relative distance after
Same result/look

Translate objects +5 away from camera

Translate camera -5 away from objects

(a)
Moving the Camera

- We can move camera using sequence of rotations and translations
- Example: side view
  - Rotate the camera
  - Move it away from origin
  - Model-view matrix \( C = TR \)

```c
// Using mat.h

mat4 t = Translate (0.0, 0.0, -d);
mat4 ry = RotateY(90.0);
mat4 m = t*ry;
```
Moving the Camera Frame

- Object distances relative to camera determined by the model-view matrix
  - Transforms (scale, translate, rotate) go into modelview matrix
  - Camera transforms also go in modelview matrix (CTM)
The LookAt Function

- Previously, command `gluLookAt` to position camera
- `gluLookAt` deprecated!!
- Homegrown mat4 method LookAt() in mat.h
  - Can concatenate with modeling transformations

```cpp
void display() {
    ..........

    mat4 mv = LookAt(vec4 eye, vec4 at, vec4 up);
    ..........
}
```
LookAt

LookAt(eye, at, up)

Programmer defines:
• eye position
• LookAt point \((at)\) and
• Up vector \((Up)\) direction usually \((0,1,0)\)

But Why do we set Up direction?
Nate Robbins LookAt Demo

```
glTranslatef(0.00, 0.00, 0.00);
glRotatef(0.0, 0.00, 1.00, 0.00);
glScalef(1.00, 1.00, 1.00);
gBegin( . . . );
```

Click on the arguments and move the mouse to modify values.

```
GLfloat pos[4] = { 1.50, 1.00, 1.00, 0.00 };
glLookAtf(0.00, 0.00, 2.00, <- eye
          0.00, 0.00, 0.00, <- center
          0.00, 1.00, 0.00 ); <- up

glLightf(GL_LIGHT0, GL_POSITION, pos);
```

Click on the arguments and move the mouse to modify values.
What does LookAt do?

- Programmer defines eye, lookAt and Up
- **LookAt method:**
  - Form new axes \((u, v, n)\) at camera
  - Transform objects from world to eye camera frame
Camera with Arbitrary Orientation and Position

- Define new axes \((u, v, n)\) at eye
  - \(v\) points vertically upward,
  - \(n\) away from the view volume,
  - \(u\) at right angles to both \(n\) and \(v\).
  - The camera looks toward \(-n\).
  - All vectors are normalized.
LookAt: Effect of Changing Eye Position or LookAt Point

- Programmer sets \texttt{LookAt(eye, at, up)}
- If \texttt{eye, lookAt} point changes => \texttt{u,v,n} changes
Viewing Transformation Steps

1. Form camera \((u,v,n)\) frame
2. Transform objects from world frame (Composes matrix for coordinate transformation)

- Next, let’s form camera \((u,v,n)\) frame
Constructing U,V,N Camera Frame

- Lookat arguments: \texttt{LookAt(eye, at, up)}
- **Known:** eye position, LookAt Point, up vector
- **Derive:** new origin and three basis (u,v,n) vectors
Eye Coordinate Frame

- **New Origin:** eye position (that was easy)
- 3 basis vectors:
  - one is the normal vector ($\mathbf{n}$) of the viewing plane,
  - other two ($\mathbf{u}$ and $\mathbf{v}$) span the viewing plane

$\mathbf{n}$ is pointing away from the world because we use left hand coordinate system

$$\mathbf{N} = \text{eye} - \text{Lookat Point}$$

$$\mathbf{n} = \frac{\mathbf{N}}{||\mathbf{N}||}$$

Remember $\mathbf{u,v,n}$ should be all unit vectors

(u,v,n should all be orthogonal)
Eye Coordinate Frame

- How about $u$ and $v$?

  - We can get $u$ first -
    - $u$ is a vector that is perpendicular to the plane spanned by $N$ and view up vector ($V_{up}$)

\[
U = V_{up} \times n
\]

\[
u = U / |U|
\]
Eye Coordinate Frame

- How about $v$?

Knowing $n$ and $u$, getting $v$ is easy:

$$v = n \times u$$

$v$ is already normalized.
Eye Coordinate Frame

- Put it all together

Eye space **origin**: \((\text{Eye.x} , \text{Eye.y}, \text{Eye.z})\)

Basis vectors:

\[
\begin{align*}
\mathbf{n} &= \frac{\text{eye} - \text{Lookat}}{\| \text{eye} - \text{Lookat} \|} \\
\mathbf{u} &= \frac{\mathbf{V}_\text{up} \times \mathbf{n}}{\| \mathbf{V}_\text{up} \times \mathbf{n} \|} \\
\mathbf{v} &= \mathbf{n} \times \mathbf{u}
\end{align*}
\]
Step 2: World to Eye Transformation

- Next, use $u$, $v$, $n$ to compose LookAt matrix
- Transformation matrix ($M_{w2e}$)?

$$P' = M_{w2e} \times P$$

1. Come up with transformation sequence that lines up eye frame with world frame
2. Apply this transform sequence to point $P$ in reverse order
World to Eye Transformation

1. Rotate eye frame to “align” it with world frame
2. Translate (-ex, -ey, -ez) to align origin with eye

Rotation:

\[
\begin{bmatrix}
u_x & u_y & u_z & 0 \\
v_x & v_y & v_z & 0 \\
n_x & n_y & n_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Translation:

\[
\begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
World to Eye Transformation

- Transformation order: apply the transformation to the object in reverse order - translation first, and then rotate

\[ M_{w2e} = \begin{pmatrix}
ux & uy & ux & 0 \\
vx & vy & vz & 0 \\
x & ny & nz & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{pmatrix} \]

Note: \( e.u = ex.ux + ey.uy + ez.uz \)
lookAt Implementation (from mat.h)

Eye space **origin**: \((\text{Eye} . x, \text{Eye} . y, \text{Eye} . z)\)

Basis vectors:

\[
\begin{align*}
\mathbf{n} &= \frac{\text{eye} - \text{Lookat}}{|\text{eye} - \text{Lookat}|} \\
\mathbf{u} &= \frac{(\text{V}_\text{up} \times \mathbf{n})}{|\text{V}_\text{up} \times \mathbf{n}|} \\
\mathbf{v} &= \mathbf{n} \times \mathbf{u}
\end{align*}
\]

| \text{ux} & \text{uy} & \text{uz} & -\mathbf{e} \cdot \mathbf{u} | \\
| \text{vx} & \text{vy} & \text{vz} & -\mathbf{e} \cdot \mathbf{v} | \\
| \text{nx} & \text{ny} & \text{nz} & -\mathbf{e} \cdot \mathbf{n} | \\
| 0 & 0 & 0 & 1 |

\begin{verbatim}
mat4 LookAt( const vec4& eye, const vec4& at, const vec4& up )
{
    vec4 n = normalize(eye - at);
    vec4 u = normalize(cross(up,n));
    vec4 v = normalize(cross(n,u));
    vec4 t = vec4(0.0, 0.0, 0.0, 1.0);
    mat4 c = mat4(u, v, n, t);
    return c * Translate( -eye );
}
\end{verbatim}
References

- Interactive Computer Graphics, Angel and Shreiner, Chapter 4