Cameras / Views

Given glm::vec3 eye, at, up
ViewMatrices can be set with glm::lookAt(eye, at, up)

Cameras can be static or dynamic (animated / moved)

Dynamic cameras can provide first and third person views, animated tours of a scene. Create View Matrix from current camera.

Given a NPC class that has a position, orientation, and "eye"
As NPC moves its first person and third person cameras also move...

First person can use
NPC's OM at and up vectors
and set eye to NPC's eye

Third person needs to move the camera

```
translate(0, aboveValue, behindValue)
translate(0, 200, -400)
```

// NPC eye is 200 above NPC's feet
Multiple Viewpoints

class Camera3D : public Obj3D {
    ...
    glm::mat4 makeView() {
        glm::mat4 View;
        View = glm::lookAt(....);
        return View;
    }
    ...
};

// in application file

Camera3D camera[nCameras]; // collection of views
int currentCamera = ?;      // set in glutKeyboardFuncs
void display() {              // glutIdleFunc
    ...
    glm::mat4 View = camera[currentCamera]->makeView();
    ...
    // compute MVP and drawArrays ...
}
Setting the View – glmLookAt && OM

Given the camera's orientation matrix

\[
\begin{pmatrix}
X & U_x & A_x & T_x \\
Y & U_y & A_y & T_y \\
Z & U_z & A_z & T_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

create a view along it’s “looking at” vector with the following:

Looking at, \textbf{lat}, vector is - at

\[
\text{glm}::\text{lookAt}( \\
\quad Tx, \quad Ty, \quad Tz, \quad // \text{camera pos, "eye"} \\
\quad Tx - Ax, Ty - Ay, Tz - Az, \quad // \text{looking down Z, "look at"} \\
\quad Ux, \quad Uy, \quad Uz) \quad // \text{up vector}
\]

Sometimes the looking along – at (\textbf{lat}) should have more length

\[
Tx - Ax * 10, Ty - Ay * 10, Tz - Az * 10
\]

If object is “looking” along at vector

\[
Tx + Ax * 10, Ty + Ay * 10, Tz + Az * 10
\]
Consider a movable 6DF object as a Rigid body with rotation and translations vectors (forces) that affect its next movement.

- **@**: current position (position vector, pos, 4th column of OM)
- **OM**: matrix of orientation unit vectors: up, right, at
  - initially a 4x4 identity matrix

- **rotation vectors** (yaw, pitch, roll) are applied to OM
  - sets "local" or "frame" unit vectors

- **dv**: direction vector, length is the distance to move along the lookingAt vector lat (-at)

- other translations vectors (direction and length forces, for example:
  - **gv**: gravity vector,
  - **sv**: speed vector

\[
\begin{align*}
RM &= (RM) \cdot (\text{yaw}) \cdot (\text{pitch}) \cdot (\text{roll}) \quad // \text{set orientation} \\
dv &= gv + sv \quad // \text{translation vector} \\
@ &= @ + dv \quad // \text{set position}
\end{align*}
\]
Kinetics

Physical modeling: velocity, acceleration, inertia, gravity, friction,... requires continuous time to be "packetized".

Time deltas (dt) are timer ticks or elapsed time per render frame

Motion -- object might have:
  position (@) center of bounding sphere or box
  direction vector (DV) where length is speed
  for each dt @ += DV
  normalized DV (nDV or lat) for collision reflection, viewing

Error (e) dt ± e collision point ± e collision time ± e determine acceptable range of accuracy (speed / time unit)
accuracy Vs performance tradeoff

\[
\begin{array}{c}
\text{Movement} \\
\text{Kinetics} \\
\text{Physical modeling: velocity, acceleration, inertia, gravity, friction,... requires continuous time to be "packetized".} \\
\text{Time deltas (dt) are timer ticks or elapsed time per render frame} \\
\text{Motion -- object might have:} \\
\phantom{\text{pos}} \text{position (@) center of bounding sphere or box} \\
\phantom{\text{dir}} \text{direction vector (DV) where length is speed} \\
\phantom{\text{dir}} \text{for each dt @ += DV} \\
\phantom{\text{dir}} \text{normalized DV (nDV or lat) for collision reflection, viewing} \\
\phantom{\text{err}} \text{Error (e) dt ± e collision point ± e collision time ± e determine acceptable range of accuracy (speed / time unit)} \\
\phantom{\text{acc}} \text{accuracy Vs performance tradeoff}
\end{array}
\]
star gravity and ship thrust

Space ship, sun, and gravity

Ship's position after move

@  += dv + gv

Star's gravity = $120/distance^2$
ship's velocity = 12 initially on –Z
All forces in X Z plane
ship's location (100, 0, 0)

Test programs can use glm w/o having OpenGL.

modelViewProject.cpp

Spreadsheets can graph results.
Chase / evade

C, a "predator" chases a non-evading, E, "prey"

C Orient towards E
// Rotate in the plane containing lat and vector(E – C)
// Rotate on Normal to the plane
// cross product undefined for parallel vectors! test for this.
Axis to rotate = cross(lat, vector(E – C))
// Angle to rotate about normal
// acos ( dot (...) value is in radians not degrees
// adjusted acos valid for acute angles, obtuse use 360° - acos
rotation angle =
    adjusted acos (dot(lat, E – C) )

C_i position after move

E_i moves
Moving “connected” entities

Connected (tethered, anchored) entities: one entity’s placement is dependent on another’s e.g., Primus's position is dependent on Duo's e.g., following cameras, moons, “gold”.

Use an update() then display () architecture in application file

All movable objects are updated. Controlling entities are updated before connected entities.

```java
for all controllingEntities
    controllingEntity[i]  update();
```

Connected entities update() requires knowledge of controlling entity’s placement (OM: rotation and translation)

```java
for all connectedEntities
    connectedEntity[i]  update();
```

After application's update() all entities can be displayed
Unmovable and Movable

Rendering can involve drawing unmovable (terrain) and movable objects (avatar exploring the terrain). Draw unmovables first to facilitate collision testing.

Assume collections of UnMovables & Movables

```cpp
void update() { // glutTimerFunc, respect dependencies
    ...
    for all Movables movableObject[i]->update();
}

void display() { // glutIdleFunc
    ...
    View = camera[currentCamera]->makeView
    ...
    for all UnMovables // draw all static objects
        MVP = Project * View * unMovableObject->getModel()
    ...
    for all Movables // draw all dynamic objects
        MVP = Project * View * movableObject->getModel()
    ...
    glutSwapBuffers()
}
```
User controlled Player
Using an Player (avatar, ship) to explore a scene with following camera

```cpp
class Player : public MovableObj3D
    // has RM rotation and TM translation matrices
    // either computes OM an OrientationMatrix on update
    // or has a getModelMatrix() that returns the OM

float step // -1, 0, 1 values
float pitch, roll, yaw // -1, 0, 1
float radians // to rotate on axis,
    // fixed small value, radians(0.02)
vec3 forward // vector to move forward
...
void setMove(int i) // step = i
void setPitch(int i) // pitch = i
void setRoll(int i) // roll = i
void setYaw(int i) // yaw = i
...```
void update()
    if (pitch != 0) RM =
        glm::rotate(RM, pitch * radians, glm::vec3(1,0,0));
    else if (yaw != 0) // like pitch above but for yaw
    else if (roll != 0) // like pitch above but for roll
        forward = getIn(OM) * step * stepDistance
    TM = translate(TM, forward)
    // OM = TM * RM set here or in object's getModelMatrix()
    step = pitch = yaw = roll = 0  // reset
Separate RM and TM w/ updates

Representing an Obj3D with 3 transformation matrices

**TM** translation matrix  movement only, set w/ glm::translate(....)

\[
\begin{bmatrix}
1 & 0 & 0 & X \\
0 & 1 & 0 & Y \\
0 & 0 & 1 & Z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

all translation values are coordinate values

**RM** rotation matrix, rotations only, set w/ glm::rotate(....)

\[
\begin{bmatrix}
r_x & u_x & a_x & 0 \\
r_y & u_y & a_y & 0 \\
r_z & u_z & a_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

all rotation values are sin, cos values

**OM** OM = RM * TM

RM holds the history of applied rotations, separate from translations

TM holds the history of applied translations, separate from rotations
Process users input to control player

// In application file
// glutSpecialFunc method

void specialKeyEvent(int key, int x, int y) {
    if (key == GLUT_KEY_UP && glutGetModifiers() != GLUT_ACTIVE_CTRL)
        player->setMove(1);
    else if (key == GLUT_KEY_DOWN && glutGetModifiers() != GLUT_ACTIVE_CTRL)
        player->setMove(-1);
    ...
    else if (key == GLUT_KEY_UP && glutGetModifiers() == GLUT_ACTIVE_CTRL)
        player->setPitch(1);
    else if (key == GLUT_KEY_DOWN && glutGetModifiers() == GLUT_ACTIVE_CTRL)
        player->setPitch(-1);
    ...
}
Move towards

Sometimes a NPC needs to move towards a target on successive frames.

Need to know which way to turn > amount to turn

NPC's orientation matrix has "right" and "up" vectors. "left" and "down" are simple negations.

Assume on a plane (2D as in terrain navigation) the shortest distance between the target and "right" or "left" vectors determines turn direction. May need to use cross product to know axis (plane) to rotate on (in)

Blue triangle NPC would turn "right" on next frame to moving towards blue target.
Dot && Cross revisited

Consider 2 unit vectors, v1 and v2

\[ v1 = (1, 0, 0) \quad // \text{red} \]
\[ v2 = (0, 0, 1) \quad // \text{blue} \]

Two vectors define a plane

Their cross product is the normal to the plane.  // green

If the vectors are co-linear (same or opposite directions)
the cross product is [0, 0, 0] not a valid axis of rotation.
the normalized cross product is [-nan(ind), -nan(ind), -nan(ind)]
The dot product is 1 or -1.

The dot product returns the cosine of the angle between the vectors.
The arcCosine converts the dot product to a radian angle value

\[
\begin{align*}
\text{Vector3.dot}(v1, v2) > 0 & \quad \text{angle} < 90^\circ \quad 1.57 \\
\text{Vector3.dot}(v1, v2) < 0 & \quad \text{angle} > 90^\circ \quad 1.57 \\
\text{Vector3.dot}(v1, v2) == 0 & \quad \text{angle} == 90^\circ \quad 1.57 \\
\text{Vector3.dot}(v1, v2) == 1 & \quad \text{angle} == 0^\circ \quad 0.0 \\
\text{Vector3.dot}(v1, v2) == -1 & \quad \text{angle} == 180^\circ \quad 3.14
\end{align*}
\]
test colinear vectors

If the difference between two vector's absolute normalized values == 0 they are colinear.

The following function is in file glmUtils465.hpp in the includes465 directory.

```cpp
// v1 and v2 are colinear within epsilon (rounding) range
// epsilon of 0.0 has no range
// suggest epsilon of 0.1 for orient towards

bool colinear(glm::vec3 v1, glm::vec3 v2, double epsilon) {
    glm::vec3 v1t, v2t;  // local "temp" vectors
    v1t = glm::abs(glm::normalize(v1));
    v2t = glm::abs(glm::normalize(v2));
    if (glm::distance(v1t, v2t) <= epsilon) return true;
    else return false;
}
```
Colinerarity and chase / evade

Consider the chaser's looking at vector (LAT) and the vector from the chaser to the evader (toTarget).

If they are collinear they are on the same line, but they can be in opposite directions and have an in-front, behind relationship.

There are 2 lengths:
- **C** + C.LAT to **E**
- **C** to **E**

When **C** > **E**, chaser will not catch evader

rotate chaser Pi on Up
dot(v1, v2) // vec3 vectors

Consider the angle between two vectors

A (0.00, 0.00, -1.00) // Forward
B (0.50, 0.00, 0.87) or 12 B's equally rotated on Y

Note dot(A, B) is valid for 0 to π values when converted to arcCosine (converts to radian values).

For angles π to 2π you need to adjust by 2π - arcCosine

<table>
<thead>
<tr>
<th>radian</th>
<th>aCos(dotProduct)</th>
<th>dotProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.52</td>
<td>0.52</td>
<td>0.87</td>
</tr>
<tr>
<td>1.05</td>
<td>1.05</td>
<td>0.50</td>
</tr>
<tr>
<td>1.57</td>
<td>1.57</td>
<td>0.00</td>
</tr>
<tr>
<td>2.09</td>
<td>2.09</td>
<td>-0.50</td>
</tr>
<tr>
<td>2.62</td>
<td>2.62</td>
<td>-0.87</td>
</tr>
<tr>
<td>3.14</td>
<td>3.14</td>
<td>-1.00</td>
</tr>
<tr>
<td>3.67</td>
<td>2.62</td>
<td>-0.87</td>
</tr>
<tr>
<td>4.19</td>
<td>2.09</td>
<td>-0.50</td>
</tr>
<tr>
<td>4.71</td>
<td>1.57</td>
<td>0.00</td>
</tr>
<tr>
<td>5.24</td>
<td>1.05</td>
<td>0.50</td>
</tr>
<tr>
<td>5.76</td>
<td>0.52</td>
<td>0.87</td>
</tr>
<tr>
<td>6.28</td>
<td>0.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Orient towards in 3D
Assume the NPC wants to turn to face a target object.

**NPC vector** is the NPC's "looking at" vector, its direction of movement.

Target vector = target's position – NPC's position,
the desired new "looking at" for the NPC.

**AOR** is the axis of rotation.

All 3 vectors must be normalized {NPC, target, axisOR}

\[
\text{AOR} = \frac{\text{crossProduct} (\text{Target}, \text{NPC})}{\|\text{axisOfRotation}\|}
\]

\[
\text{AORdirection} = \text{AOR}.x + \text{AOR}.y + \text{AOR}.z
\]

\[
\text{aCos} = \frac{\text{dotProduct} (\text{Target}, \text{NPC})}{\text{rotation amount}}
\]

// rotations \(\leq \pi\) radians (0 to 180°) crossProduct in + direction

// rotations > \(\pi\) radians crossProduct in - (negative) direction

if (AORdirection) \(\geq 0\) // adjust rotational value

\[
\text{radian} = \text{aCos}
\]

else

\[
\text{radian} = 2 \pi - \text{aCos}
\]

rotationMatrix = glm::rotate(rotationMatrix, radian, AOR);
The order of your arguments for the cross and dot products determines how you adjust your arcCosine value. For example the opposite order than on the previous slide results in the following approach.

```plaintext
AOR = crossProduct (NPC, target) // axisOfRotation
AORdirection = AOR.x + AOR.y + AOR.z
aCos = dotProduct (NPC, target) // rotation amount

if (AORdirection <= 0) // adjust the rotational value
    radian = -aCos;
else
    radian = 2 π + aCos // aCos
```

Note the changes for the adjustment of the rotational value
Collision Detection: computationally intense

2 or more objects are moving - do they collide?
\[ O(n^2) \quad n = \text{faces} / \text{object} \]

How often is collision tested (collision culling)
Test when predictive collision is possible

Multi-phase approach (simple example)

Cull all objects pairs that exceed collision thresholds (distance from centers)

Test spherical bounding volumes
Test axis-aligned box bounding volumes
Test axis-oriented box bounding volumes
Test each surface for collision

Spherical test \[ (r_1 + r_2 > \text{distance between centers}) \]
often sufficient.
Error

No error – step size (distance / tick) is multiple of distance between objects.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

error – step size not multiple of distance between objects. At time $t_{i+2}$ collision is detected late ($\varepsilon$).
Multiple Bounding Spheres

Multiple bounding spheres can be used to cover non-spherical shapes with better approximation.

Collision test is against a collection of object's bounding spheres.
Colliding moving objects (Sphere == simplicity)

- distance between spheres <= sum radii
- axis aligned BB tests coordinate values
- axis oriented BB tests vector – plane intersection wrt dt

Reflected direction by collision between object and stationary surface (plane) is determined by reversing the direction (sign) of DV and a \( \pi \) radian, \( 180^\circ \), rotation of colliding vector with the surface normal.

Error, \( \varepsilon \), @ when collision detected is not on the plane.

Collision between two moving objects is a \( \pi \) radian rotation about the vector between their centers (w/ equal inertia) at their centers.
Collision with plane’s axis aligned bounding box

Collision with plane’s axis oriented bounding box.

Error possible both in direction and distance.

Collision with plane’s bounding sphere

Error with distance test for collision rather than DV axis intersection test.
Obstacle avoidance

Obstacles avoidance requires "see ahead" information or knowledge about other objects (size and location).

Objects could have "sensors" ahead of them. These sensors detect potential collisions with obstacles. The detection determines avoidance steering forces. *(extend to 3D)*

```plaintext
sensor collision state       : action
// positive rotation to left
none                          : move
far right                     : rotate(x), move
far right, right              : rotate(2x), move
far right, right, middle      : rotate(3x), move
right, middle                 : rotate(4x), move
middle                        : rotate(5x), move
far left,                     : rotate(-x), move
far left, left                : rotate(-2x), move
far left, left, middle        : rotate(-3x), move
left, middle                  : rotate(-4x), move
```