DM 2231
GAMES DEVELOPMENT TECHNIQUES

2015/16 SEMESTER 1

Week 4 – Camera and GUI #1

MODULE SCHEDULE

Week	Dates	Topic	Remarks	Public Holidays
1	20-Apr-2015 to 24-Apr-2015	Module Introduction / 3D Game Programming	Issue Assignment 1	
2	27-Apr-2015 to 1-May-2015	Game Application		1 May. Labour Day
3	4-May-2015 to 8-May-2015	User Input		
4	11-May-2015 to 15-May-2015	Camera and GUI #1		
5	18-May-2015 to 22-May-2015	Camera and GUI #2		
6	25-May-2015 to 29-May-2015	Basic Game Physics		
7	1-Jun-2015 to 5-Jun-2015	Implementing Game Audio (E-learning)	Submit Assignment 1	1 Jun. Vesak Day
8	8-Jun-2015 to 12-Jun-2015	Mid-Sem Break		
9	15-Jun-2015 to 19-Jun-2015	Mid-Sem Break		
10	22-Jun-2015 to 26-Jun-2015	2D Game Programming #1	Issue Assignment 2	
11	29-Jun-2015 to 3-Jul-2015	2D Game Programming #2		
12	6-Jul-2015 to 10-Jul-2015	2D Game Programming #3		
13	13-Jul-2015 to 17-Jul-2015	Game Data		17 Jul. Hari Raya Puasa
14	20-Jul-2015 to 24-Jul-2015	Design Pattern #1		
15	27-Jul-2015 to 31-Jul-2015	Design Pattern #2		
16	3-Aug-2015 to 7-Aug-2015	Basic Artificial Intelligence (E-learning)		7 Aug. SG50 Public Holiday
17	10-Aug-2015 to 14-Aug-2015	Good Programming Practices	Submit Assignment 2	10 Aug. National Day

RECAP ON LAST WEEK'S LECTURE

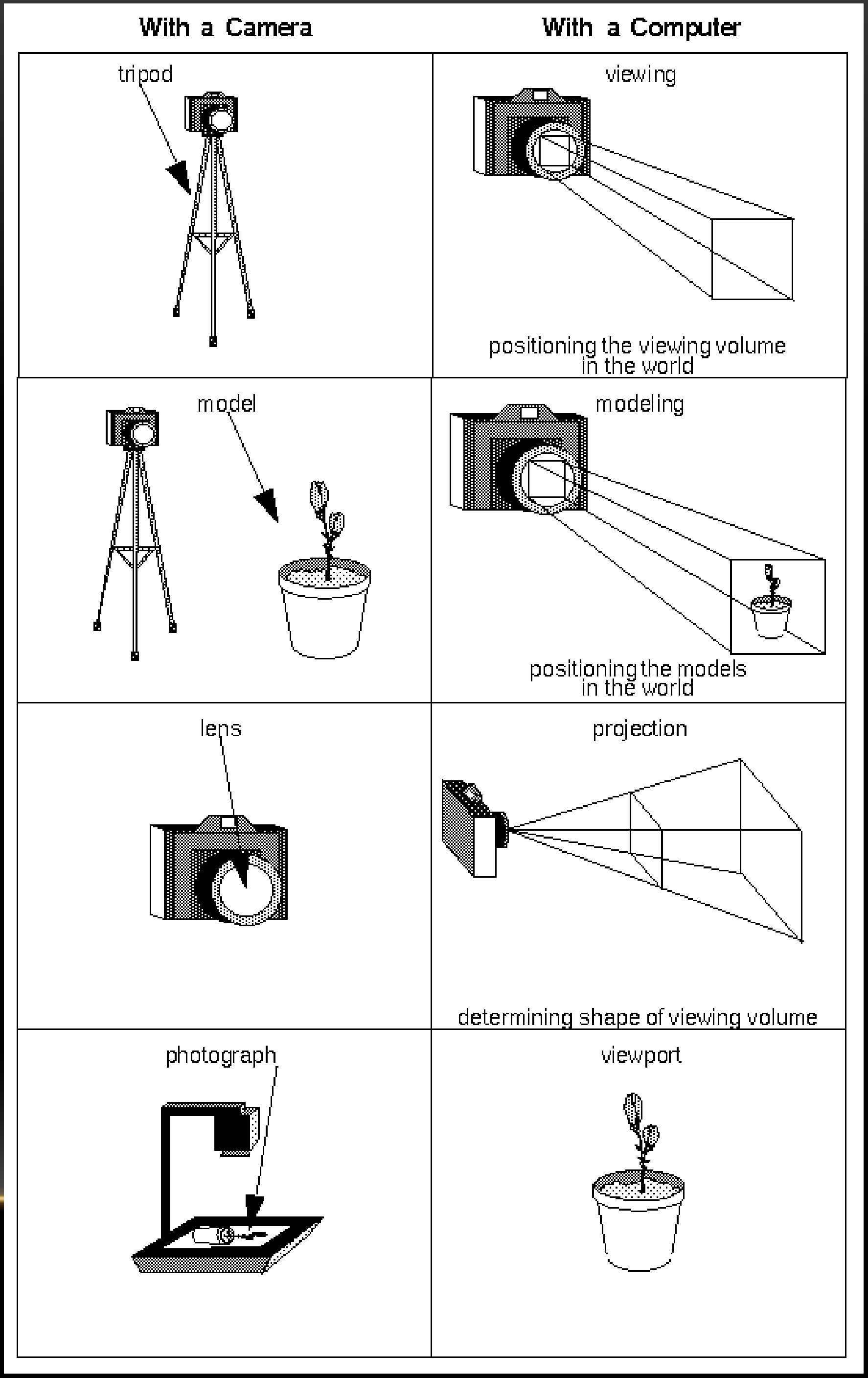
- We have discussed about the main issues with User Input
  - Techniques to get input from the keyboard and mouse
  - Using Hardware Abstraction to create codes for generic input
  - Using Frame-Independent Movements to run with the same gameplay speed on both fast and slow hardwares
  - Use Firing and Weapons Control to make the 3D FPS games more realistic

TABLE OF CONTENT

- Camera and GUI #1
  - The role of cameras in video games
  - Camera Class
  - First-Person Shooters
  - Camera Inertia

CAMERA

- The process to display a scene is like taking a photograph with a camera.
- The steps with a camera (or a computer):
  - Set up your tripod and point the camera at the scene (viewing transformation).
  - Arrange the desired composition (modeling transformation).
  - Choose the camera lens or adjust the zoom (projection transformation).
  - Determine how large you want the final photograph to be - for example, you might want it enlarged (viewport transformation).
  - Snap the photo, or draw the scene.



CAMERA

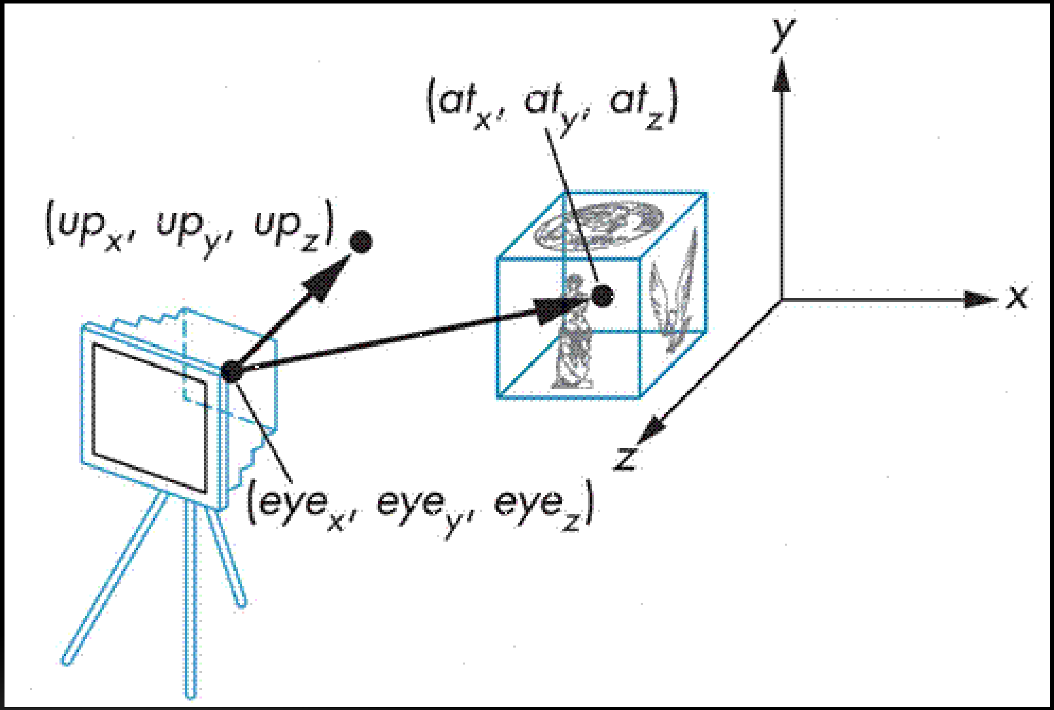
This is the typical OpenGL codes to a first person shooter camera.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
Mtx44 perspective;
perspective.SetToPerspective(45.0f, 4.0f / 3.0f, 0.1f, 10000.0f);
projectionStack.LoadMatrix(perspective);

// Camera matrix
viewStack.LoadIdentity();
viewStack.LookAt( camera.position.x, camera.position.y, camera.position.z,
                  camera.target.x, camera.target.y, camera.target.z,
                  camera.up.x, camera.up.y, camera.up.z );
// Model matrix : an identity matrix (model will be at the origin)
modelStack.LoadIdentity();
```

These values determine the type of camera which you have!

PARAMETERS OF THE CAMERA

- How the camera works



CAMERA CLASS

- Existing available camera tool - gluLookAt()
  - Basic utility which includes a series of rotate and translate commands <u>inside</u>
  - Allows viewing along an arbitrary line of sight with all 3 parameters defined.
- What if you wish to do extra transformations?
  - For greater flexibility of camera use?
    - Broadcast, dynamic, pro, co-op?
- Why is a Camera Class needed?
  - Encapsulate more commands for greater ease of use
  - Easier to add modes and actions

PROPOSED CAMERA CLASS FEATURES

- The camera class should manage:
  - Motion along the view vectors as well as arbitrary axes (in some cases)
  - Rotation about the view vectors as well as arbitrary axes (in some cases)
  - Camera's own orientation by keeping the viewing vectors orthogonal to each other
- It should define motion for at least two possible types of camera:
  - Land camera – e.g. for road vehicles simulation
  - Air camera – e.g. for flight simulation

CAMERA CLASS DECLARATION

```
#ifndef CAMERA_3_H
#define CAMERA_3_H

#include "Camera.h"

class Camera3 : public Camera
{
public:
   Vector3 defaultPosition;
   Vector3 defaultTarget;
   Vector3 defaultUp;

   Camera3();
   ~Camera3();
   virtual void Init(const Vector3& pos,
const Vector3& target, const Vector3& up);
   virtual void Update(double dt);
   // Update Camera status
   virtual void UpdateStatus(const unsigned
char key);
   virtual void Reset();

   virtual void MoveForward(const double
dt);
   virtual void MoveBackward(const double
dt);
   virtual void MoveLeft(const double dt);
   virtual void MoveRight(const double dt);

private:
   bool myKeys[255];
};

#endif
```

- The Camera class given to you is shown on the left.
- Good practise to use Vector3D class
  - Provides storage, and
  - Common operations (e.g. dot product, cross product etc.).
- Use enumerated type to distinguish between the types of camera,

```
enum CAM_TYPE { LAND_CAM,
AIR_CAM };
```

CAMERA CLASS DECLARATION

- We can set the camera type during usage
  - Automatically activate pre-defined features.
    - Bar the usage of certain features
      - Aeroplane's camera can do strafing?!

```
virtual void SetCameraType(CAM_TYPE sCameraType);
virtual CAM_TYPE GetCameraType(void);
```

CAMERA CLASS DECLARATION

- Previously, we had created these basic methods to help us move the camera

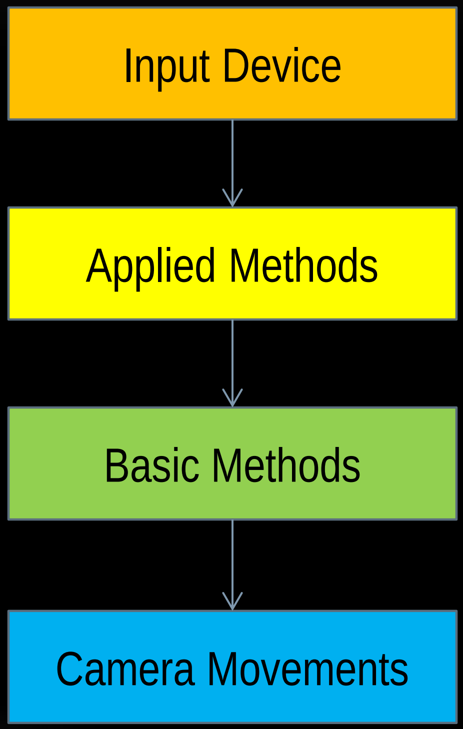
```
virtual void MoveForward(const double dt);
virtual void MoveBackward(const double dt);
virtual void MoveLeft(const double dt);
virtual void MoveRight(const double dt);
```

- We add applied methods to perform specific actions using the basic methods.

```
class Camera {
    ...
    public:
    ...
    virtual void Pitch(const double dt);
    virtual void Yaw(const double dt);
    virtual void Roll(const double dt);
    virtual void Walk(const double dt);
    virtual void Strafe(const double dt);
    virtual void Jump(const double dt);
};
```

CAMERA CLASS DECLARATION

- We use the applied methods to call the basic methods.
  - A form of Abstraction
  - Easy to add new camera features
    - Combine basic methods to have new camera features
  - Easy to add new forms of input device
    - Input device calls directly related to camera's desired movement



CAMERA CLASS DECLARATION

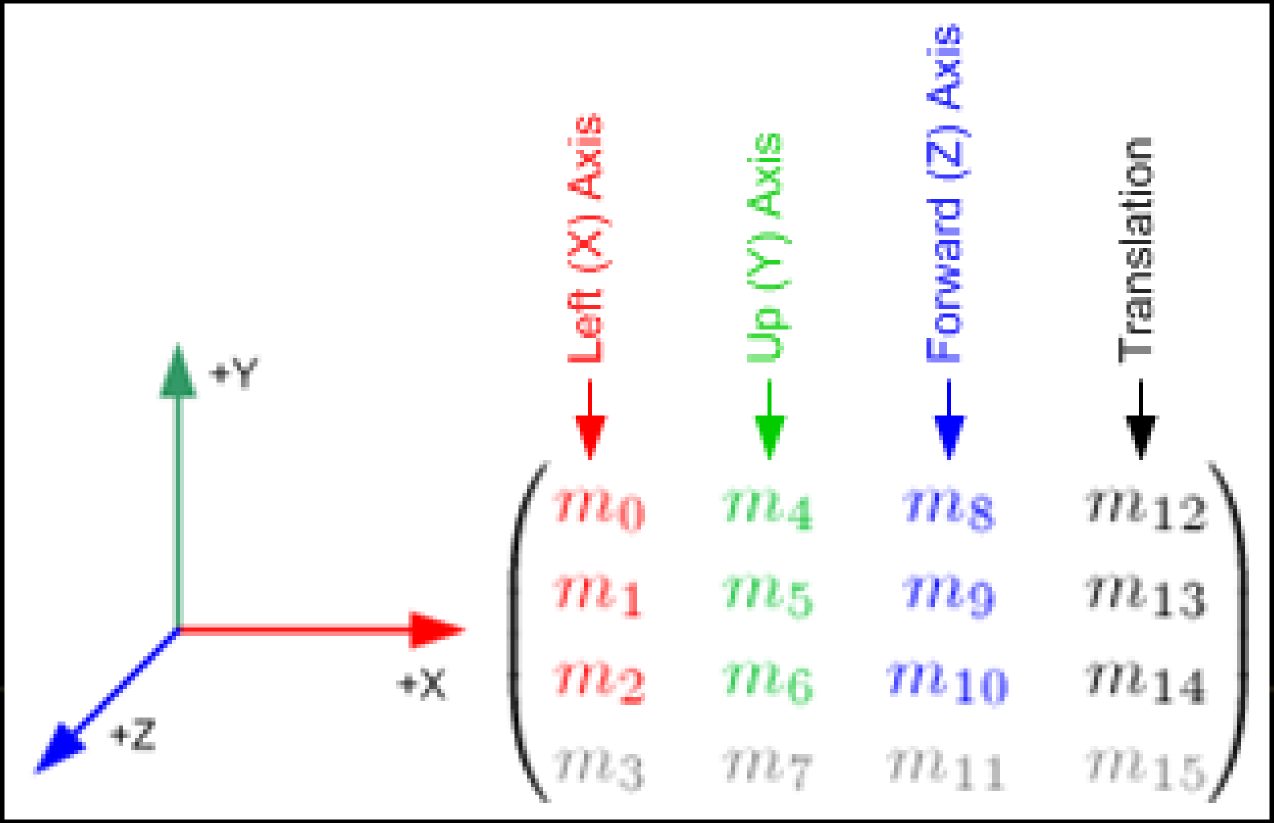
- Additional basic methods which you can add in

```
        virtual void TurnLeft(const double dt);
        virtual void TurnRight(const double dt);
        virtual void LookUp(const double dt);
        virtual void LookDown(const double dt);
        virtual void SpinClockWise(const double dt);
        virtual void SpinCounterClockWise(const double 
dt);
```

CAMERA CLASS: BUILDING THE VIEW MATRIX

Refresher on how to translate

http://www.songho.ca/opengl/gl_transform.html

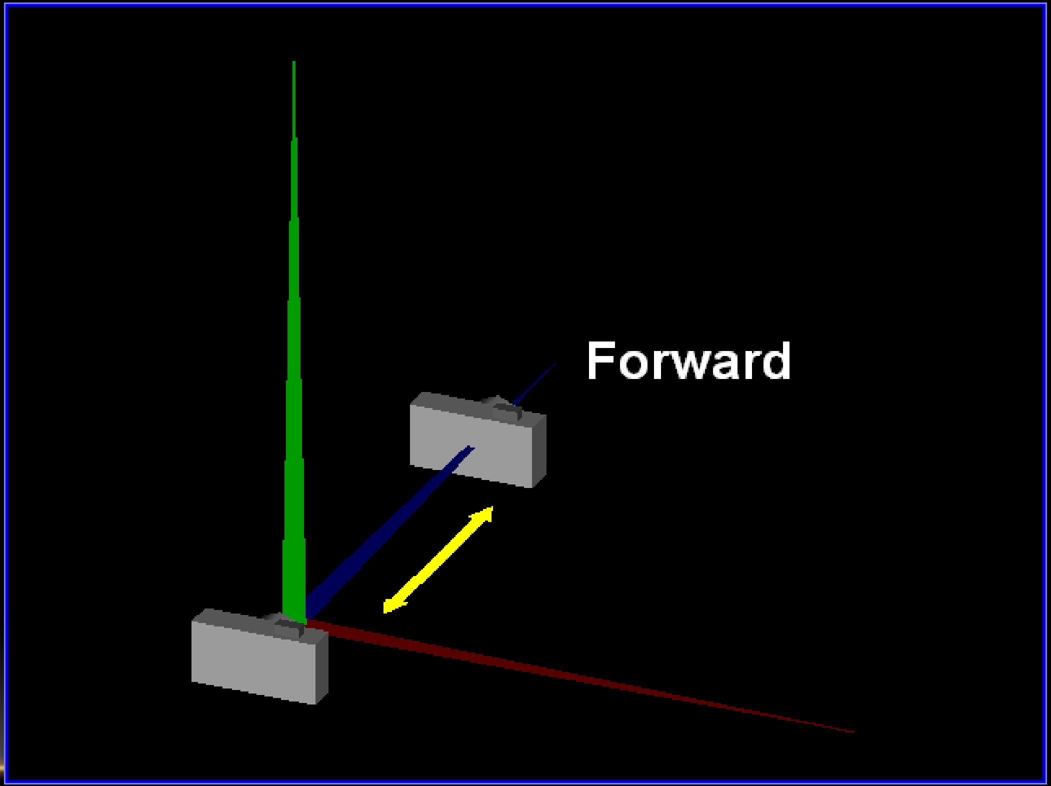


CAMERA:
FIRST-PERSON CAMERA

- First Person Shooters uses the first-person camera.
- Defined by at least four degrees of freedom (X,Y,Z and yaw), with pitch sometimes added to the mix.
- Usually, the keyboard is used to control the movement of the camera
  - Camera moves forward with W or S key, or the Up and Down arrow keys.
  - Camera moves sideways with A or D key, or the Left and Right arrow keys
  - Yaw and pitch is done with the mouse movements

CAMERA MOTION

- Walking
  - This is motion along the Forward vector (or Z-axis):



CAMERA:
FIRST-PERSON CAMERA

```
void Camera3::MoveForward(const double dt)
{
  // Calculate the direction vector of the camera
  Vector3 view = (target - position).Normalized();

  // Constrain the movement to the ground if the camera type
is land based
  if (sCameraType == LAND_CAM)
  {
    view.y = 0.0f;
    view = view.Normalized();
  }

  // Update the camera and target position
  position += view * CAMERA_SPEED * (float)dt;
  target += view * CAMERA_SPEED * (float)dt;
}
```

CAMERA_SPEED
is the movement
speed of the player

$$Velocity = \frac{Distance}{Time}$$

$$\therefore Distance = Velocity * Time$$

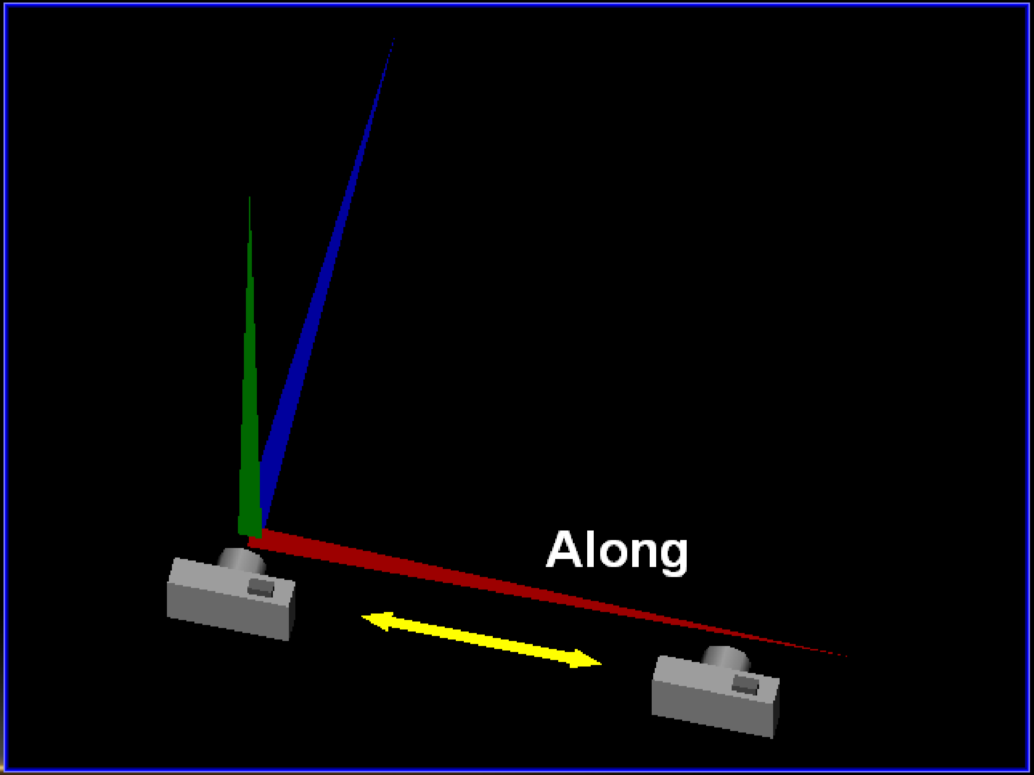
CAMERA: FIRST-PERSON CAMERA

```
void Camera3::Walk(const
double dt)
{
  if (dt > 0)
    MoveForward(dt);
  else if (dt < 0)
    MoveBackward(abs(dt));
}
```

```
void Camera3::Update(double
dt)
{
  if ( myKeys['w'] == true)
  {
    Walk( dt );
    myKeys['w'] = false;
  }
  if (myKeys['s'] == true)
  {
    Walk( -dt );
    myKeys['s']      = false;
  }
}
```

CAMERA MOTION

- Strafing
  - This is side to side motion on the Along vector (or X-axis):



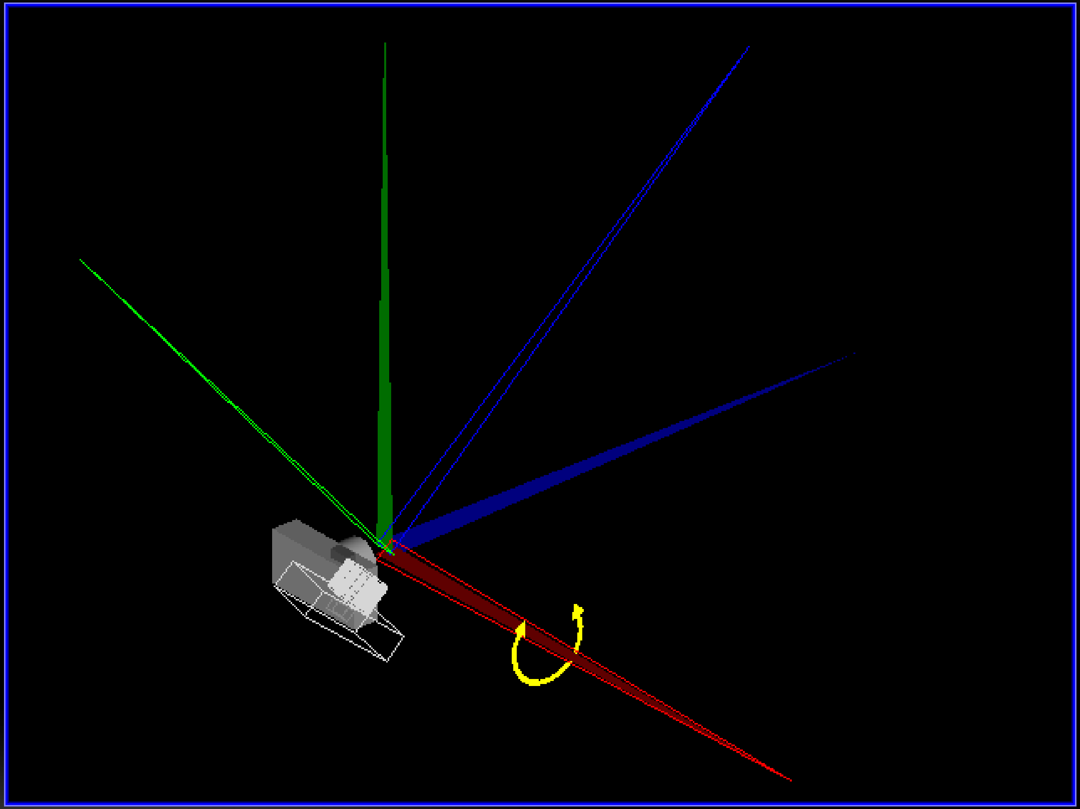
CAMERA:
FIRST-PERSON CAMERA

```
void
Camera3::Strafe(const
double dt)
{
  if (dt > 0)
    MoveRight(dt);
  else if (dt < 0)
    MoveLeft(abs(dt));
}
```

```
void Camera3::Update(double
dt)
{
  if (myKeys['a'] == true)
  {
    Strafe( -dt );
    myKeys['a']      = false;
  }
  if (myKeys['d'] == true)
  {
    Strafe( dt );
    myKeys['d']      = false;
  }
}
```

CAMERA ROTATION

- Pitching
  - This is rotation about the Along vector – looking up and down



CAMERA:
FIRST-PERSON CAMERA

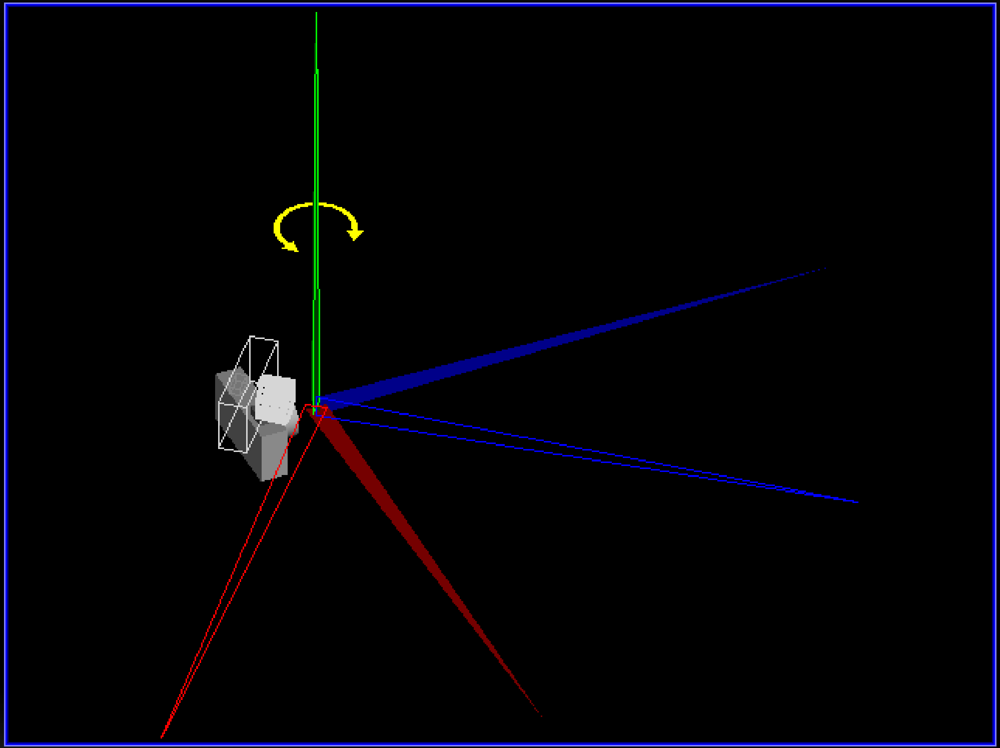
```
void Camera3::Update(const double dt)
{
  //Update the camera direction based on mouse move
  // left-right rotate
  if ( Application::camera_yaw != 0 )
    Yaw( dt );
  if ( Application::camera_pitch != 0 )
    Pitch( dt );
}

void Camera3::Pitch(const double dt)
{
  if ( Application::camera_pitch > 0.0 )
    LookUp( dt );
  else if ( Application::camera_pitch < 0.0 )
    LookDown( dt );
}
```

CAMERA:
FIRST-PERSON CAMERA

```
void Camera3::LookUp(const double dt)
{
  float pitch = (float)(-CAMERA_SPEED *
                      Application::camera_pitch * (float)dt);
  Vector3 view = (target - position).Normalized();
  Vector3 right = view.Cross(up);
  right.y = 0;
  right.Normalize();
  up = right.Cross(view).Normalized();
  Mtx44 rotation;
  rotation.SetToRotation(pitch, right.x, right.y, right.z);
  view = rotation * view;
  target = position + view;
}
```

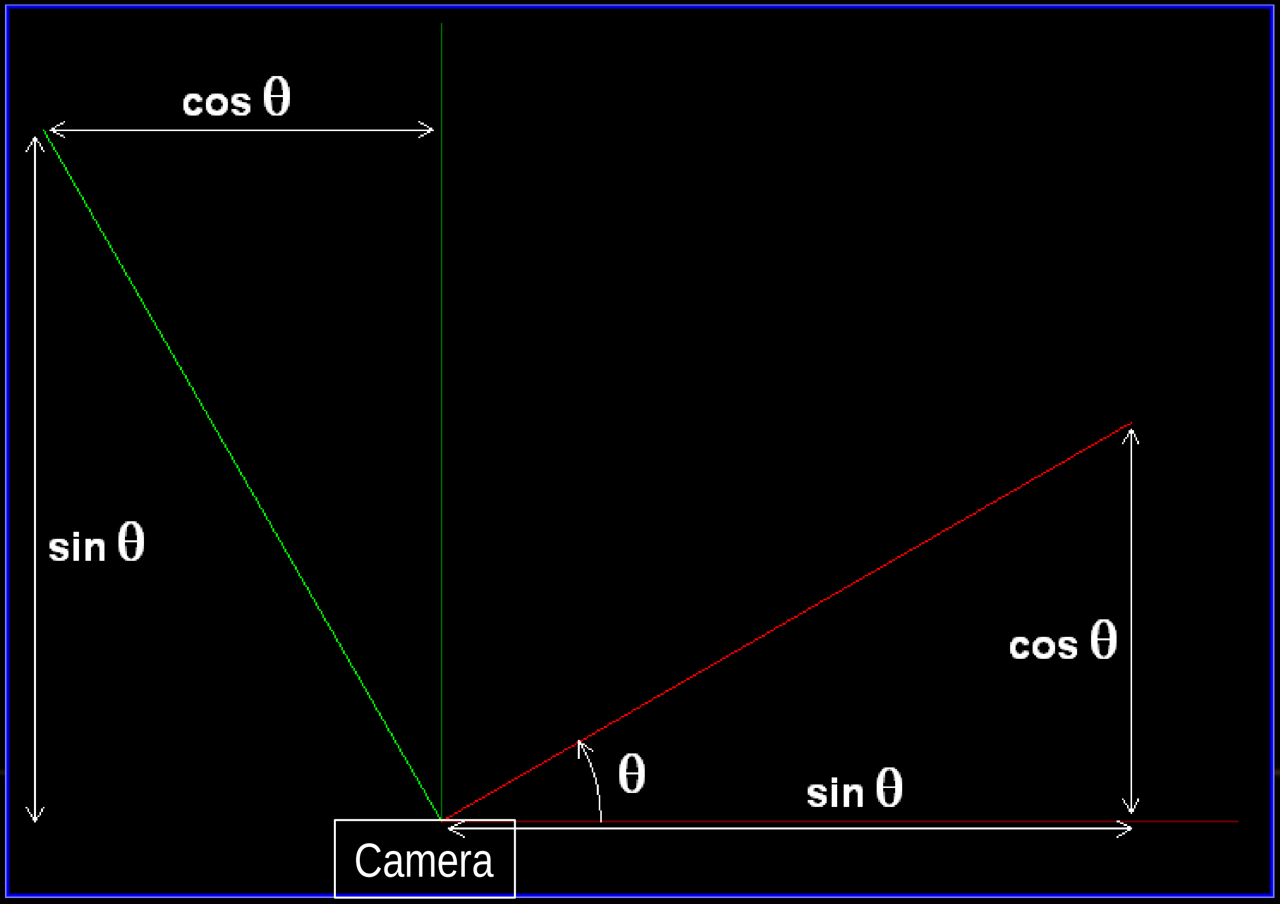
CAMERA ROTATION

- Yaw
  - This is rotation about the Up vector – looking left and right



CAMERA ROTATION FUNCTIONS

- If we move the mouse left on our screen, then what it looks like from <u>ABOVE</u>.



CAMERA:
FIRST-PERSON CAMERA

```
// Yaw with mouse
Yaw += ROTSPEED * elapsed * dx;

playerpos.x= CAMERA_SPEED * deltaTime * dz * cos(yaw);
playerpos.z= CAMERA_SPEED * deltaTime * dz * sin(yaw);
```

CAMERA:
FIRST-PERSON CAMERA

```
void Camera3::Update(const double dt)
{
  //Update the camera direction based on mouse move
  // left-right rotate
  if ( Application::camera_yaw != 0 )
    Yaw( dt );
  if ( Application::camera_pitch != 0 )
    Pitch( dt );
}

void Camera3::Yaw(const double dt)
{
  if ( Application::camera_yaw > 0.0 )
    TurnRight( dt );
  else if ( Application::camera_yaw < 0.0 )
    TurnLeft( dt );
}
```

CAMERA:
FIRST-PERSON CAMERA

```
void Camera3::TurnRight(const double dt)
{
  Vector3 view = (target - position).Normalized();
  float yaw = (float)(-CAMERA_SPEED * Application::camera_yaw
* (float)dt);
  Mtx44 rotation;
  rotation.SetToRotation(yaw, 0, 1, 0);
  view = rotation * view;
  target = position + view;
  Vector3 right = view.Cross(up);
  right.y = 0;
  right.Normalize();
  up = right.Cross(view).Normalized();
}
```

CAMERA:
CAMERA INERTIA

- Most first-person shooters (FPSs) implement inertia on their camera controllers for increased realism.
- Our character accelerates progressively and also stops moving in an inertial fashion.
  - This makes movement smoother at almost no coding cost.
  - To add inertia, we need to use these physics equations:

$$Acceleration = \frac{Velocity}{Time}$$

$$\therefore Velocity = Acceleration * \text{Time}$$

$$Distance = Velocity * \text{Time}$$

CAMERA:
CAMERA INERTIA

- Rotation of camera

```
yawvel+=ROTACCEL*elapsed*(input.right-input.left);
if (yawvel>ROTSPEED)
      yawvel=ROTSPEED;
if (yawvel<-ROTSPEED)
      yawvel=-ROTSPEED;
if (input.right-input.left==0)
      yawvel=yawvel*BRAKINGFACTOR;
yaw+=yawvel*elapsed*(input.right-input.left);
```

Calculate velocity of rotation

Clamp to the maximum rotation speed

Calculate distance to rotate

CAMERA:
CAMERA INERTIA

- Movement of camera

```
dz=(input.up-input.down);
vel+=ACCEL*elapsed*dz;
if (vel>SPEED) vel=SPEED;
if (vel<-SPEED) vel=-SPEED;
if (dz==0) vel=vel*BRAKINGFACTOR;
playerpos.x+=vel*elapsed*dz*cos(yaw);
playerpos.z+=vel*elapsed*dz*sin(yaw);
```

Calculate velocity of movement

Clamp to the maximum move speed

Calculate distance to move

SUMMARY

- We have discussed about the main issues with Camera Control
  - The role of cameras in video games
  - Using a Camera Class to encapsulate camera movement methods and View Matrix
  - First-Person Shooter camera
  - Camera Inertia