FACULTY OF COMPUTER AND INFORMATION SCIENCES, AIN SHAMS UNIVERSITY

Lab 5

First Camera Position

OBJECTIVES:

1. **Be able to implement a flexible Camera class that can react to user input**
2. **Run and Test a Simple program that implements a First Camera Position**

1. BUILD A CAMERA CLASS

The function gluLookAt() is particularly useful for positioning and aiming the Camera in a fixed position, but its user interface is not so useful for a moving camera that reacts to the user input. We need to build our Camera class that gives a better control for Camera.

- ## Camera Design

We would define the position and orientation of the camera relative to the world coordinate system using four *camera vectors*: a *right vector*, *up vector*, *look vector*, and *position vector*, as Figure1 illustrates. These vectors essentially define a local coordinate system for the camera described relative to the world coordinate system.

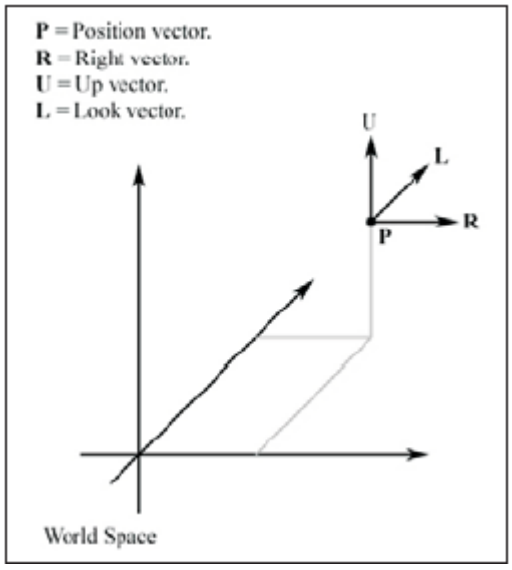


Figure 1: The camera vectors defining the position and orientation of the camera relative to the world

With these four vectors describing the camera, we would like our camera to be able to perform the following six operations:

- Rotate around the right vector (pitch)
- Rotate around the up vector (yaw)
- Rotate around the look vector (roll)
- Strafe along the right vector
- Fly along the up vector

- Move along the look vector

Through these six operations, we are able to move along three axes and rotate around three axes, giving us a flexible six degrees of freedom. The following Camera class definition reflects our description of data and desired methods:

```
class Camera
      {
            /// <summary>
            /// The position of the eye.
            /// </summary>
            public Vector3 P;

            /// <summary>
            /// The right direction. x
            /// </summary>
            public Vector3 R;

            /// <summary>
            /// The up direction. y
            /// </summary>

            public Vector3 U;

            /// <summary>
            /// The -ve of the look direction. z
            /// </summary>
            public Vector3 D;

void Yaw(float angleDegrees);

void Pitch(float angleDegrees);

void Roll(float angleDegrees);

void MoveForward(float dist);

void MoveBackward(float dist);

void MoveRight(float dist);

void MoveLeft(float dist);

void MoveUpward(float dist);

void MoveDownward(float dist);
        };
```

- **Camera Implementation**

1. Computing the View Matrix

We want a transformation matrix **V** such that:

- **pV** = (0, 0, 0)—The matrix **V** transforms the camera to the origin.
- **rV** = (1, 0, 0)—The matrix **V** aligns the right vector with the world x-axis.
- **uV** = (0, 1, 0)—The matrix **V** aligns the up vector with the world y-axis.
- **dV** = (0, 0, 1)—The matrix **V** aligns the look vector with the world z-axis.

We can divide the task of finding such a matrix into two parts:
1) a translation part that takes the camera's position to the origin and
2) a rotation part that aligns the camera vectors with the world's axes.

Having the vectors $\mathbf{p} = (p_x, p_y, p_z)$, $\mathbf{r} = (r_x, r_y, r_z)$, $\mathbf{u} = (u_x, u_y, u_z)$, and $\mathbf{d} = (d_x, d_y, d_z)$ be the position, right, up, and look vectors, respectively, the view transformation matrix **V** should be:

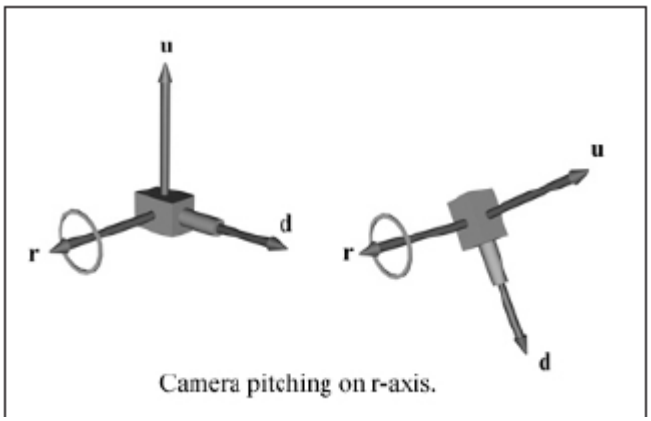
$$\begin{bmatrix} r_x & u_x & d_x & 0 \\ r_y & u_y & d_y & 0 \\ r_z & u_z & d_z & 0 \\ -\mathbf{p}\cdot\mathbf{r} & -\mathbf{p}\cdot\mathbf{u} & -\mathbf{p}\cdot\mathbf{d} & 1 \end{bmatrix}$$

We can build this in the Camera `Tellgl()` Function like that:

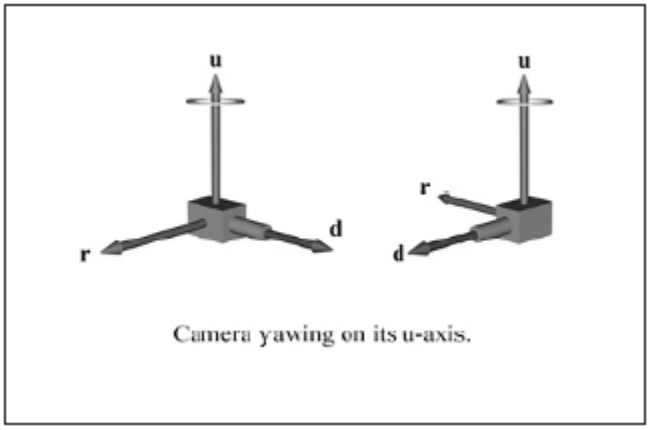
```
public void Tellgl()
{

	float m [16];

	m[0] = R.x; m[4] = R.y; m[8] = R.z; m[12] = -Vector3::DotProduct(P, R);
	m[1] = U.x; m[5] = U.y; m[9] = U.z; m[13] = -Vector3::DotProduct(P, U);
	m[2] = D.x; m[6] = D.y; m[10] = D.z; m[14] = -Vector3::DotProduct(P, D);
	m[3] = 0; m[7] = 0; m[11] = 0; m[15] = 1;
	//post-multiply the current matrix by m: ct = ct * M
	glMultMatrixf(m);
}
```

2. Pitch, Yaw and Roll

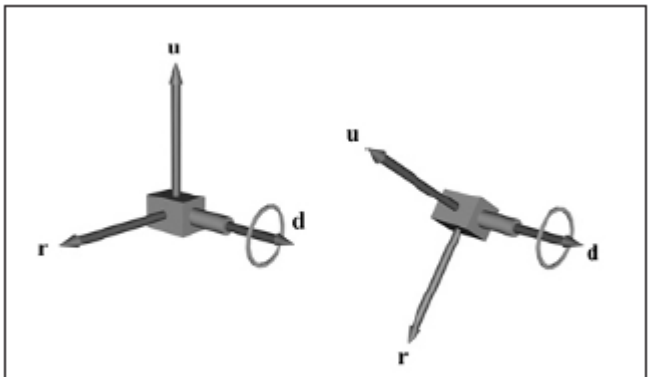
In order to **Pitch**, we need to rotate the up and look vectors around the right vector by the specified rotation angle. Similarly, we see that when we **Yaw**, we need to rotate the look and right vectors around the up vector by the specified rotation angle. Finally, we see that when we **Roll**, we need to rotate the up and right vectors around the look vector by the specified rotation angle.



Camera pitching on r-axis.



Camera yawing on its u-axis.



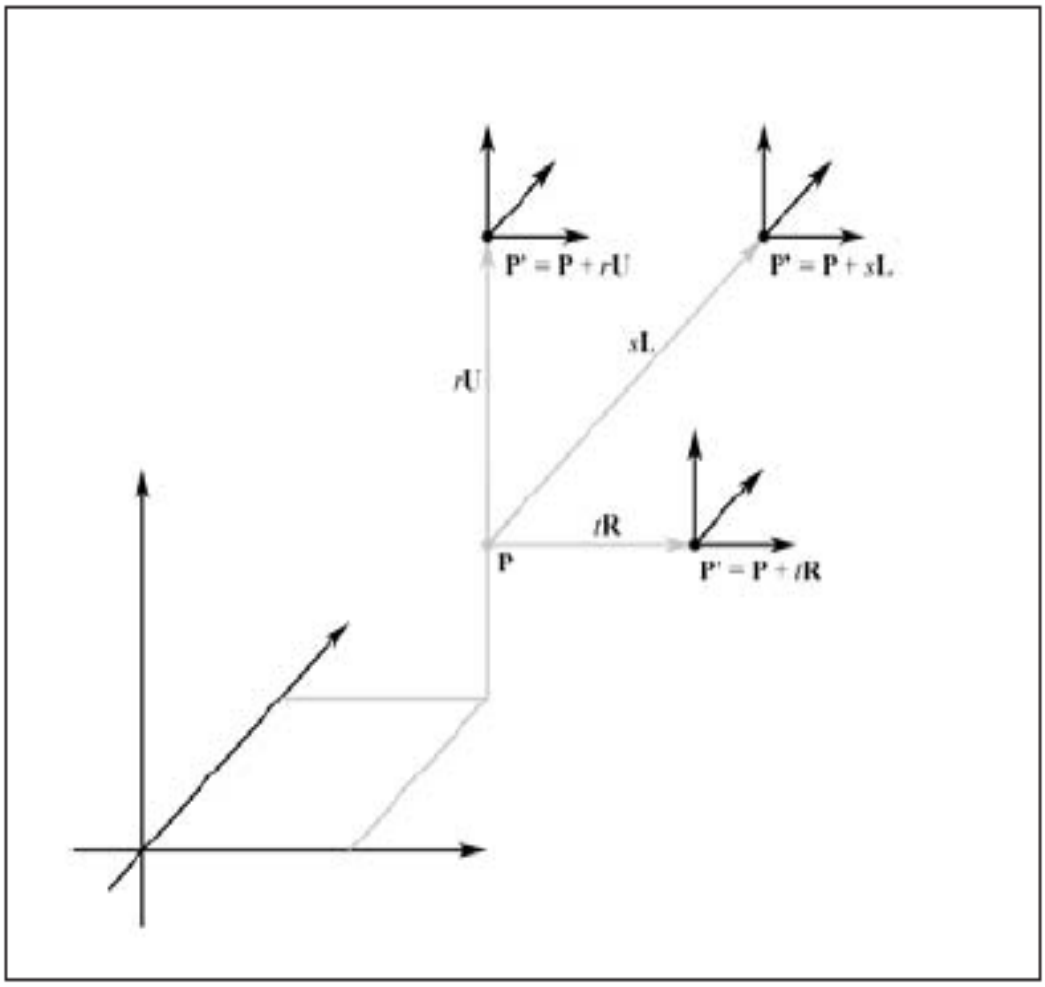
Camera rolling on its d-axis.

The implementation of the Pitch, Yaw and Roll methods follow the next code:

```
/// <summary>
/// Rotates the axes (D, R) about the U-axis with the specified angle.
            /// </summary>//y--->z--->x
            public void Yaw(float angleDegrees)
            {
                  float angleRad = ToRadians(angleDegrees);
                  float c = cos(angleRad);
                  float s = sin(angleRad);
                  Vector3 Dnew = c * D + s * R;
                  Vector3 Rnew = -s * D + c * R;
                  //Replace the old vectors with the new ones
                  D = Dnew;
                  R = Rnew;
            }

/// <summary>
/// Rotates the axes (U, D) about the R-axis with the specified angle.
/// </summary>//x--->y--->z
            public void Pitch(float angleDegrees)
            {
                  float angleRad = Math2.ToRadians(angleDegrees);
                  float c = (float)Math.Cos(angleRad);
                  float s = (float)Math.Sin(angleRad);
                  Vector3 Unew = c * U + s * D;
                  Vector3 Dnew = -s * U + c * D;
                  //Replace the old vectors with the new ones
                  U = Unew;
                  D = Dnew;
            }

/// <summary>
/// Rotates the axes (R, U) about the D-axis with the specified angle.
/// </summary>//z--->x--->y
            public void Roll(float angleDegrees)
            {
                  float angleRad = Math2.ToRadians(angleDegrees);
                  float c = (float)Math.Cos(angleRad);
                  float s = (float)Math.Sin(angleRad);
                  Vector3 Rnew = c * R + s * U;
                  Vector3 Unew = -s * R + c * U;
                  //Replace the old vectors with the new ones
                  R = Rnew;
                  U = Unew;
            }
```

3. Walking, Strafing and Flying

When we refer to walking, we mean moving in the direction that we are looking (that is, along the look vector). Strafing is moving side to side from the direction we are looking, which is of course moving along the right vector. Finally, we say that flying is moving along the up vector. To move along any of these axes, we simply add a vector that points in the same direction as the axis that we want to move along to our position vector



The implementation of the Walk, Strafe and Fly methods follow the next code:

```
/// <summary>
/// Moves the eye point a distance dist forward == -dist * D
/// Walk
/// </summary>
            void MoveForward(float dist)
            {
                  P -= dist * D;
            }

/// <summary>
/// Moves the eye point a distance dist backward == +dist * D
/// Walk
/// </summary>
            void MoveBackward(float dist)
            {
                  P += dist * D;
            }

/// <summary>
/// Moves the eye point a distance dist to the right == +dist * R
/// Strafe
/// </summary>
      void MoveRight(float dist)
        {
            P += dist * R;
        }

/// <summary>
/// Moves the eye point a distance dist to the left == -dist * R
/// Strafe
/// </summary>
        public void MoveLeft(float dist)
        {
            P -= dist * R;
        }

/// <summary>
/// Moves the eye point a distance dist upward == +dist * U
/// Fly
/// </summary>
        public void MoveUpward(float dist)
        {
            P += dist * U;
        }

/// <summary>
/// Moves the eye point a distance dist downward == -dist * U
/// Fly
/// </summary>
        public void MoveDownward(float dist)
        {
            P -= dist * U;
        }
```

2. IMPLEMENT AND RUN A SIMPLE PROGRAM FOR FIRST CAMERA POSITION

Insert an object form your camera class inside the form and inside the function `InitializeGraphics()` you should create a new instance from your own camera with the initial parameters for the eye, center and up vectors

```
Camera camera;
.
.
.
void InitGraphics()
{

      .
      .
      .
      camera = Camera(0, 0, 90, 0, 0, 0, 0, 1, 0);

}
```

Inside function OnDisplay() you should create your veiwing matrix for the camera by seting the MODELVIEW matrix with the required camera parametes using the function Tellgl()

```
void OnDisplay(){
    glClearColor(1, 1, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      .
      .
      .
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      camera.Tellgl();

      //your Drawing goes here
      .
      .
      .
}
```

Regarding key handling, the key down event will be as follows:

```
float Yawangle = 0, Rollangle = 0, Pitchangle = 0;
void OnKeyDown(unsigned char key, int, int){
     switch (key)
     {
      case 'y':
            Yawangle += 0.1f;
            camera.Yaw(Yawangle);
            break;
      case 'p':
            Pitchangle += 0.1f;
            camera.Pitch(Pitchangle);
            break;
      case 'r':
            Rollangle += 0.1f;
            camera.Roll(Rollangle);
            break;
      case 'f':
            camera.MoveForward(1);
            break;
      case 'v':
            camera.MoveBackward(1);
            break;
      case 'b':
            camera.MoveRight(1);
            break;
      case 'c':
            camera.MoveLeft(1);
            break;
      case 'u':
            camera.MoveUpward(1);
            break;
      case 'd':
            camera.MoveDownward(1);
            break;
      case 'q':
            // Restore the original position
            camera.Reset(0, 0, 90, 0, 0, 0, 0, 1, 0);
            break;
     }
}
```

You can now compile and run the program.