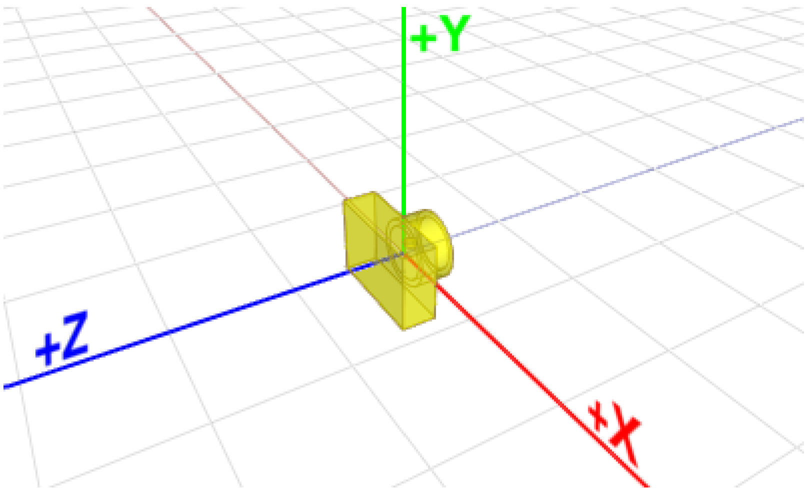
OpenGL Camera

Overview



OpenGL camera is always at origin and facing to -Z in eye space

OpenGL doesn't explicitly define neither camera object nor a specific matrix for camera transformation. Instead, OpenGL transforms the entire scene (*including the camera*) inversely to a space, where a fixed camera is at the origin (0,0,0) and always looking along -Z axis. This space is called **eye space**.

Because of this, OpenGL uses a single GL_MODELVIEW matrix for both object transformation to world space and camera (view) transformation to eye space.

You may break it down into 2 logical sub matrices:

$$M_{\text{modelView}} = M_{\text{view}} \cdot M_{\text{model}}$$

That is, each object in a scene is transformed with its own $\mathbf{M}_{\text{model}}$ first, then the entire scene is transformed reversely with $\mathbf{M}_{\text{view}}$. In this page, we will discuss only $\mathbf{M}_{\text{view}}$ for camera transformation in OpenGL.

LookAt

gluLookAt() is used to construct a viewing matrix where a camera is located at the eye position (x_e, y_e, z_e) and looking at (or rotating to) the target point (x_t, y_t, z_t) . The eye position and target are defined in world space. This section describes how to implement the viewing matrix equivalent to **gluLookAt()**.

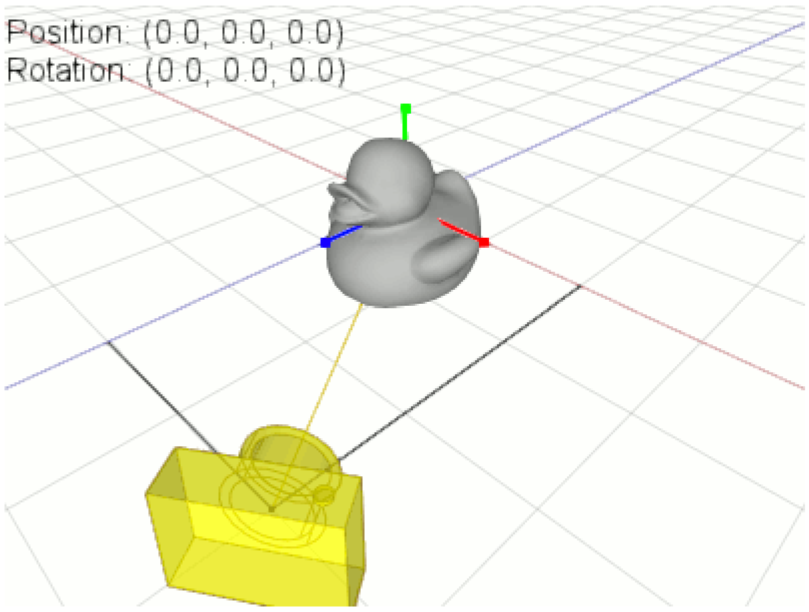
Camera's **lookAt** transformation consists of 2 transformations; translating the whole scene inversely from the eye position to the origin ($\mathbf{M}_\mathbf{T}$), and then rotating the scene with reverse orientation ($\mathbf{M}_\mathbf{R}$), so the camera is positioned at the origin and facing to the -Z axis.

$$M_{\text{view}} = M_{\text{R}} M_{\text{T}} = \begin{pmatrix} r_0 & r_4 & r_8 & 0 \\ r_1 & r_5 & r_9 & 0 \\ r_2 & r_6 & r_{10} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} r_0 & r_4 & r_8 & r_0 t_x + r_4 t_y + r_8 t_z \\ r_1 & r_5 & r_9 & r_1 t_x + r_5 t_y + r_9 t_z \\ r_2 & r_6 & r_{10} & r_2 t_x + r_6 t_y + r_{10} t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

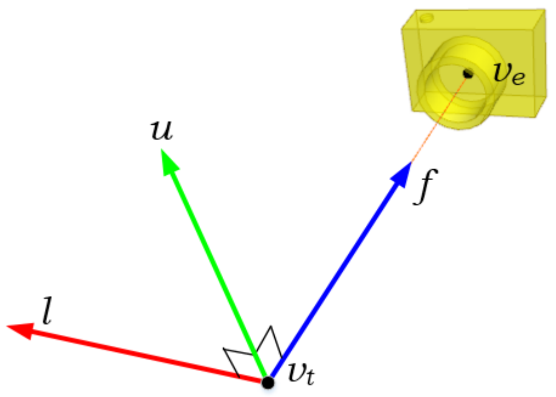
Suppose a camera is located at (2, 0, 3) and looking at (0, 0, 0) in world space. In order to construct the viewing matrix for this case, we need to translate the world to (-2, 0, -3) and rotate it about -33.7 degree along Y-axis. As a result, the virtual camera becomes facing to -Z axis at the origin.

The translation part of lookAt is easy. You simply move the camera position to the origin. The translation matrix $\mathbf{M_T}$ would be (replacing the 4th column with the negated eye position):

$$M_{\text{T}} = \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



OpenGL camera's lookAt() transformation



Left, Up and Forward vectors from target to eye

The rotation part $\mathbf{M_R}$ of lookAt is much harder than translation because you have to calculate 1st, 2nd and 3rd columns of the rotation matrix all together.

First, compute the normalized forward vector f from the target position $\mathbf{v_t}$ to the eye position $\mathbf{v_e}$ of the rotation matrix. Note that the forward vector is from the target to the eye position, not eye to target because the scene is actually rotated, not the camera is.

$$v_e - v_t = (x_e - x_t,\ y_e - y_t,\ z_e - z_t)$$
$$f = \frac{v_e - v_t}{\|v_e - v_t\|} \qquad \text{(forward vector)}$$

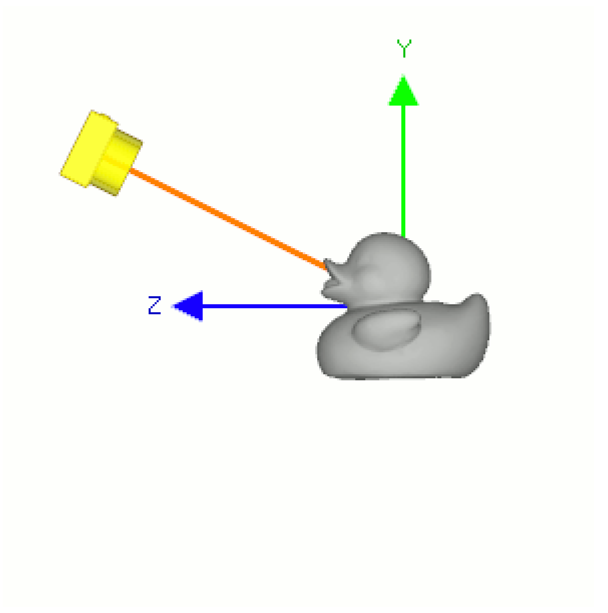
Second, compute the normalized left vector l by performing cross product with a given camera's up vector. If the up vector is not provided, you may use (0, 1, 0) by assuming the camera is straight up to +Y axis.

$$left = up \times f$$
$$l = \frac{left}{\|left\|}$$

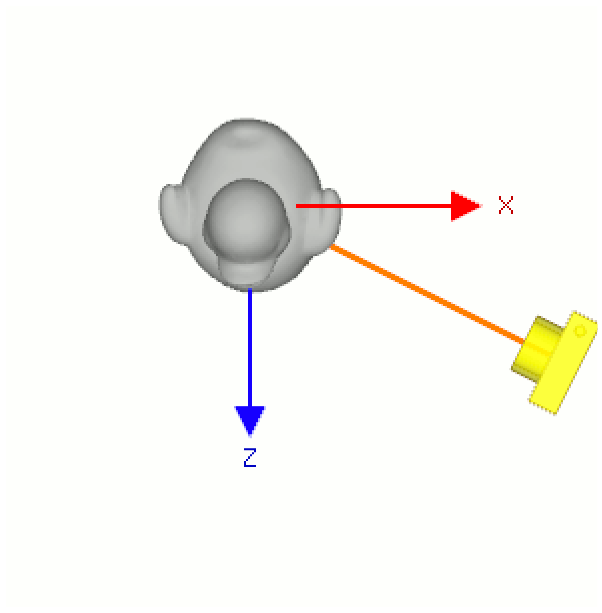
Finally, re-calculate the normalized up vector u by doing cross product the forward and left vectors, so all 3 vectors are orthonormal (perpendicular and unit length). Note we do not normalize the up vector because the cross product of 2 perpendicular unit vectors also produces a unit vector.

$$u = f \times l$$

These 3 basis vectors, l, u and f are used to construct the rotation matrix $\mathbf{M_R}$ of lookAt, however, the rotation matrix must be inverted. Suppose a camera is located above a scene. The whole scene must rotate downward inversely, so the camera is facing to -Z axis. In a similar way, if the camera is located at the left of the scene, the scene should rotate to right in order to align the camera to -Z axis. The following diagrams show why the rotation matrix must be inverted.



The scene rotates downward if the camera is above



The scene rotates to right if the camera is at left

Therefore, the rotation part **M_R** of lookAt is finding the rotation matrix from the target to the eye position, then invert it. And, the inverse matrix is equal to its transpose matrix because it is orthogonal which each column has unit length and perpendicular to the other column.

$$M_{\mathrm{R}} = \begin{pmatrix} l_x & u_x & f_x & 0 \\ l_y & u_y & f_y & 0 \\ l_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} l_x & u_x & f_x & 0 \\ l_y & u_y & f_y & 0 \\ l_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Finally, the view matrix for camera's lookAt transform is multiplying $\mathbf{M_T}$ and $\mathbf{M_R}$ together:

$$M_{\mathrm{view}} = M_{\mathrm{R}} M_{\mathrm{T}} = \begin{pmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} l_x & l_y & l_z & -l_x x_e - l_y y_e - l_z z_e \\ u_x & u_y & u_z & -u_x x_e - u_y y_e - u_z z_e \\ f_x & f_y & f_z & -f_x x_e - f_y y_e - f_z z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Here is C++ snippet to construct the view matrix for camera's lookAt transformation. Please see the details in OrbitCamera.cpp.

```
// dependency: Vector3 and Matrix4
struct Vector3
{
    float x;
    float y;
    float z;
};

class Matrix4
{
    float m[16];
}

//////////////////////////////////////////////////////////////////////////////
////
// equivalent to gluLookAt()
// It returns 4x4 matrix
//////////////////////////////////////////////////////////////////////////////
////
Matrix4 lookAt(Vector3& eye, Vector3& target, Vector3& upDir)
{
    // compute the forward vector from target to eye
    Vector3 forward = eye - target;
    forward.normalize();                 // make unit length

    // compute the left vector
    Vector3 left = upDir.cross(forward); // cross product
    left.normalize();

    // recompute the orthonormal up vector
    Vector3 up = forward.cross(left);    // cross product

    // init 4x4 matrix
    Matrix4 matrix;
    matrix.identity();

    // set rotation part, inverse rotation matrix: M^-1 = M^T for Euclidean
transform
    matrix[0] = left.x;
    matrix[4] = left.y;
    matrix[8] = left.z;
    matrix[1] = up.x;
    matrix[5] = up.y;
    matrix[9] = up.z;
    matrix[2] = forward.x;
    matrix[6] = forward.y;
    matrix[10]= forward.z;

    // set translation part
    matrix[12]= -left.x * eye.x - left.y * eye.y - left.z * eye.z;
    matrix[13]= -up.x * eye.x - up.y * eye.y - up.z * eye.z;
    matrix[14]= -forward.x * eye.x - forward.y * eye.y - forward.z * eye.z;

    return matrix;
}
```

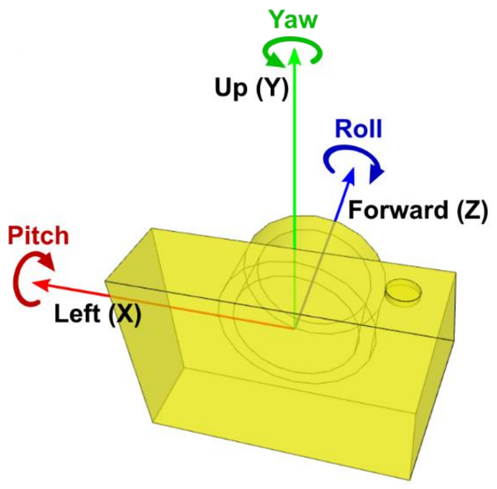
Rotation (Pitch, Yaw, Roll)

Other types of camera's rotations are **pitch**, **yaw** and **roll** rotating at the position of the camera. **Pitch** is rotating the camera up and down around the camera's local left axis (+X axis). **Yaw** is rotating left and right around the camera's local up axis (+Y axis). And, **roll** is rotating it around the camera's local forward axis (+Z axis).

First, you need to move the whole scene inversely from the camera's position to the origin (0,0,0), same as lookAt() function. Then, rotate the scene along the rotation axis; pitch (X), yaw (Y) and roll (Z) independently.

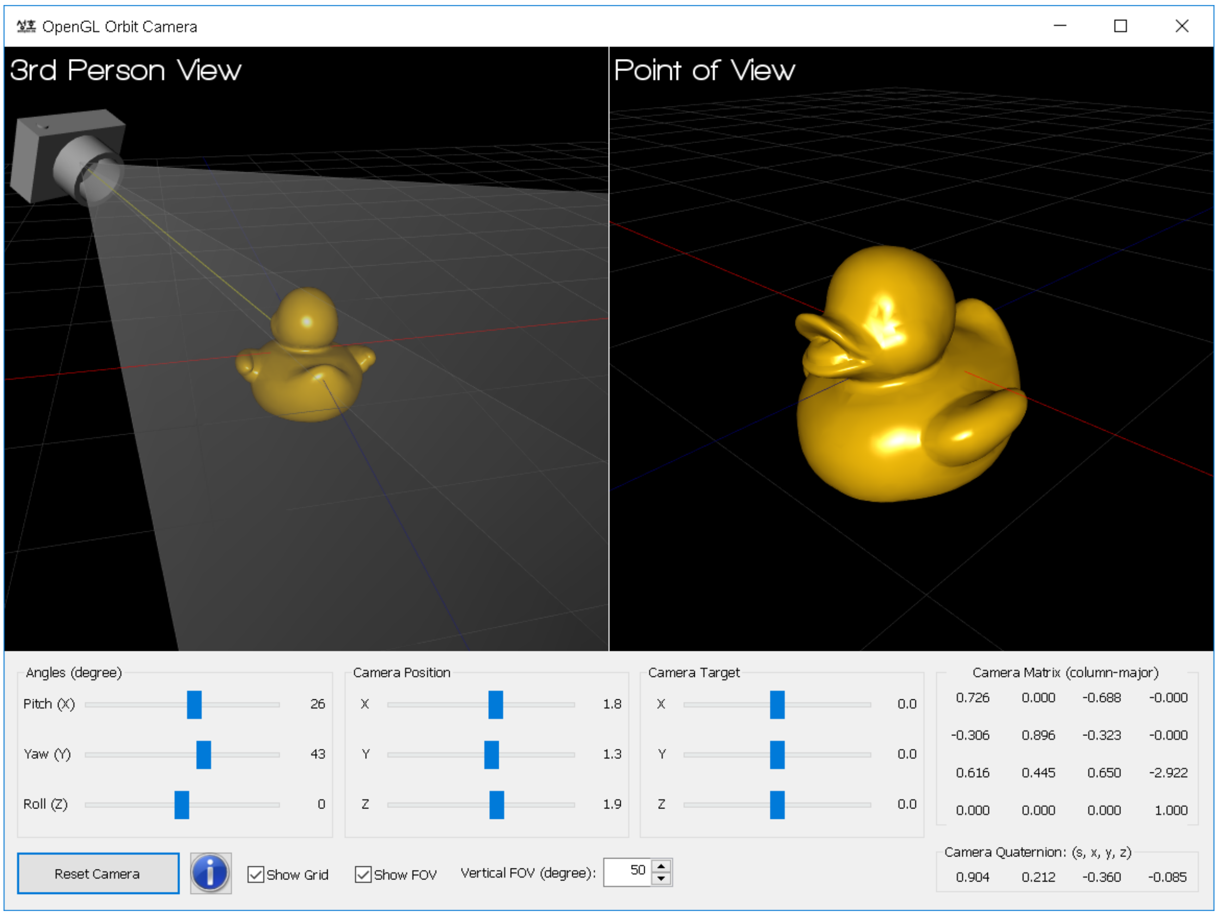
If multiple rotations are applied, combine them sequentially. Note that the different order of rotations produces a different result. A commonly used rotation order is **roll -> yaw -> pitch**. It produces a free-look camera or first-person shooter camera interface.

Keep in mind that the directions of the rotations (right-hand rule); a positive pitch angle is rotating the camera downward, a positive yaw angle means rotating the camera left, and a positive roll angle is rotating the camera clockwise.



Camera Rotations; Pitch, Yaw and Roll

Example: OpenGL Orbit Camera



Angles (degree)

Pitch (X)	26
Yaw (Y)	43
Roll (Z)	0

Camera Position

X	1.8
Y	1.3
Z	1.9

Camera Target

X	0.0
Y	0.0
Z	0.0

Camera Matrix (column-major)

0.726	0.000	-0.688	-0.000
-0.306	0.896	-0.323	-0.000
0.616	0.445	0.650	-2.922
0.000	0.000	0.000	1.000

Camera Quaternion: (s, x, y, z)

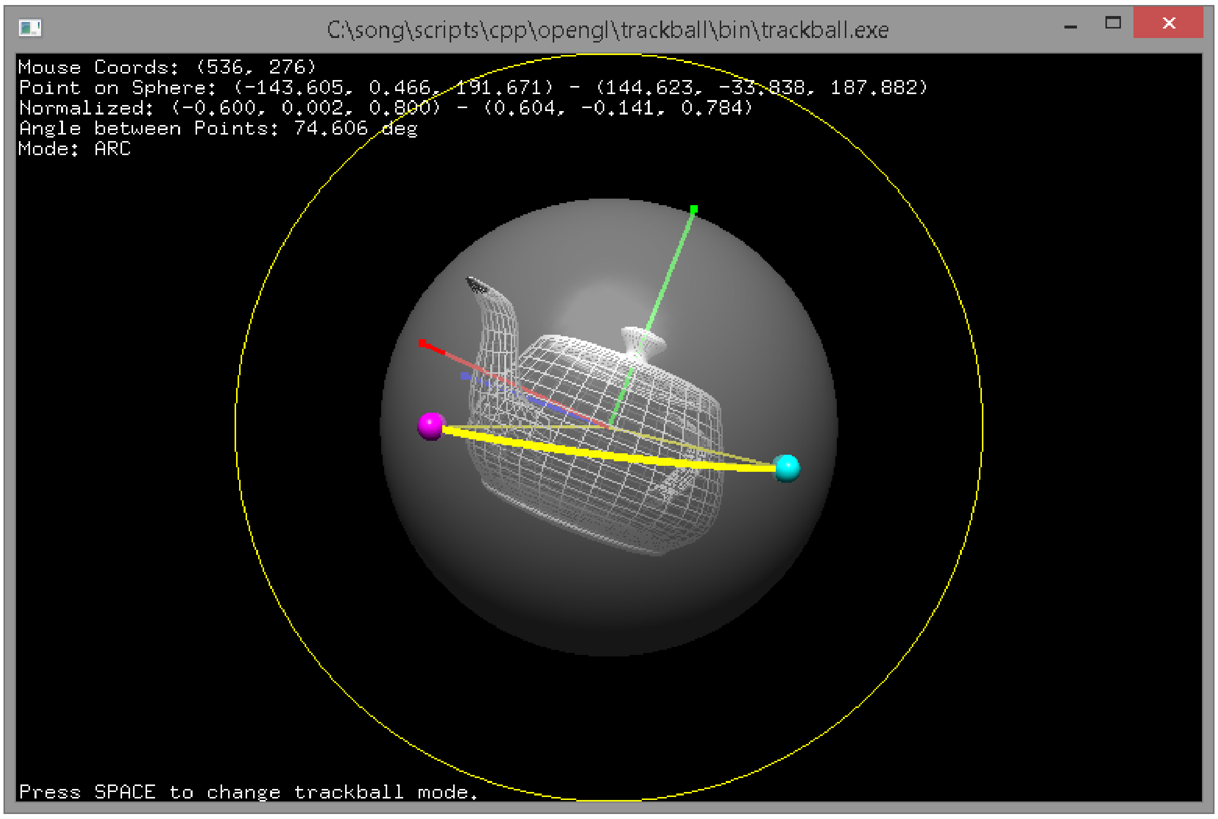
0.904	0.212	-0.360	-0.085

[✓] Show Grid [✓] Show FOV Vertical FOV (degree): 50

Download source and binary:
(Updated: 2021-03-10)

OrbitCamera.zip *(include VS 2015 project)*

Example: Trackball



```
Mouse Coords: (536, 276)
Point on Sphere: (-143.605, 0.466, 191.671) - (144.623, -33.838, 187.882)
Normalized: (-0.600, 0.002, 0.800) - (0.604, -0.141, 0.784)
Angle between Points: 74.606 deg
Mode: ARC
```

```
Press SPACE to change trackball mode.
```

Download: trackball.zip