OpenGL: The Camera, and User Controls

C. Godsalve
email: seagods@hotmail.com

February 21, 2020

Contents

1 Introduction

This article is an explanation of how the controls for some graphics programmes that I have written work. The programmes were written using OpenGL.

OpenGL stands for "Open Graphics Library". The "Open" bit means that it is open source. You can see the source code, and modify it. The details of what exactly you legally can and cannot do with the software are explained in the public license.

If the reader also has access to the OpenGL Red book (www.opengl.org), my own web-page articles on OpenGL, especially the link from there to NeHe, the reader should be able to follow the code for my applications, and start to produce high quality graphics themselves. However, the main purpose here is to enable anyone to use my code more easily.

All of my programmes can be quit at any time by pressing <Esc>, and <F1> will bring

up a help screen. The help screen shall vary from programme to programme because the "natural" defaults will depend on the application. For instance in some applications we may wish to examine an object, in others we may view a world from a fixed location (as you would in a planetarium) and in others we may move about as in a first person computer game, or a flight simulator.

All this is achieved through a non-existent entity: the OpenGL camera. In fact, OpenGL does its 3D modelling using matrix transformations. Even though the "OpenGL Camera" doesn't exist as such, the notion of a camera is a useful, as long as we don't forget about the transformation stuff.

We shall take a quick look at what these transformations are like before moving on, because there is an oddity in that the transformations are "four dimensional". The position vector of any point is the four vector (x, y, z, w). In fact, usually $w = 1$. If not the 3D position is actually $(x/w, y/w, z/w)$. Why on Earth do they do that?

Well, take a look at

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} X \\ Y \\ Z \\ a_{41}x + a_{42}y + a_{43}z \end{pmatrix} + \begin{pmatrix} a_{14} \\ a_{24} \\ a_{34} \\ a_{44} \end{pmatrix}. \tag{1}$$

Here, we have split the right hand side up, and (X, Y, Z) is just the three vector that we would get if we multiplied (x, y, z) by a 3 by 3 matrix consisting of the "top left" part of the matrix in eqn.1. If we have $a_{41} = a_{42} = a_{43} = 0$ and $a_{44} = 1$, then eqn.1 is equivalent to some arbitrary transformation without translation, plus a translation by (a_{14}, a_{24}, a_{34}).

So, rotation, reflection, stretching, scaling *and translation* are encoded in the 4×4 OpenGL matrix. It may seem a peculiar way to do things, but it makes sense in that translations are on an even par with any other kind of transform.

So now we know roughly what OpenGL is. It is a free open source 3D graphics library. It does pretty well all its 3D stuff using transformations. The transformation matrices and vectors are 4D, and are structured so that the transformation matrices include translation.

Remember that the programmes can be quit by pressing <Esc>, and that <F1> bring up a help screen. So, it is time to think about our 3D world.

2 The Camera

The camera is there so we can explore some 3D world the user has defined, or has data for. In order to do this, we must be able to move the camera around, and we must be able to look in different directions from a given camera position. How all this shall work shall differ for different applications. Nonetheless, the basics will be the same.

Whatever happens, there will be a call to an OpenGL function called "gluPerspective". This defines a "view" volume in the shape of a frustum. This shape can be thought of as follows. Suppose you have a pyramid with a rectangular base, and then chop off the point (the slice is parallel to the base). This is a frustum.

Only objects within (or partially within) the frustum can be seen. The camera is defined by an angle (the field of view), an aspect ratio (that of the rectangular base) and a near and a far value (near being the distance to the slice through the pyramid, and far being the distance to the base). The base is always at right angles to the camera view direction, and the camera looks toward the centre of the base.

So, our camera looks into the frustum from the smaller end, and this frustum translates and rotates with the camera. We have chosen a view angle of 45°, and aspect ratio of 2:3, and near and far values of 100 to 150 000.

So, we start off with the basic camera in Fig.1.

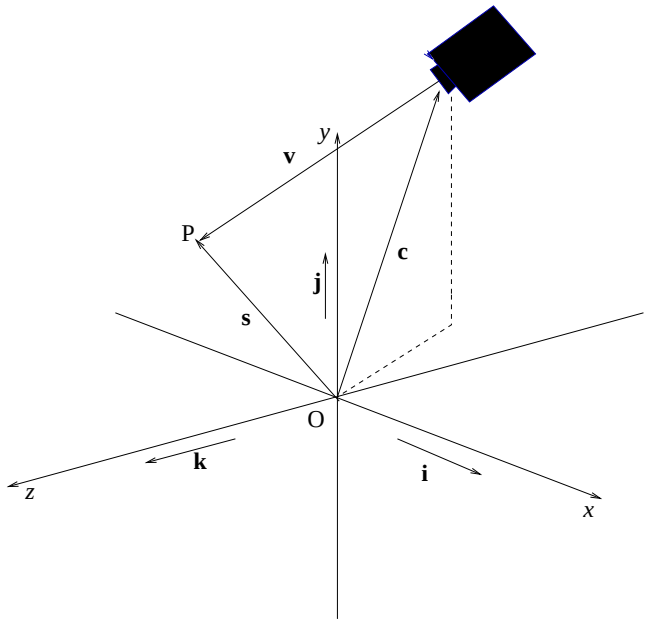

Figure 1: Relative to the origin O, we need a position vector for the camera ($\mathbf{c}$) and a position vector for the "stare at" point P ($\mathbf{s}$)

The first thing to notice in Fig.1, is that the coordinate system looks a little unusual. It *is* a right handed system (despite what some explanatory website material might tell you).

If the reader is not used to 3D coordinates, there is no confusion. However most 3D coordinates (and any data files you may have available) are oriented as follows.

Suppose your right arm is outstretched to your right with the thumb pointing up, the

index finger is pointing away to your right, and the second finger pointing ahead of you (at right angles to the index finger and thumb). Then, the x direction is given by the index finger, the y direction is given by the second finger, and the upward pointing thumb is the z direction. In the OpenGL system, we have rotate our hand so that the thumb is pointing behind us, the second finger is pointing up, and the index finger is still pointing away to the right. (The fingers and thumb are still labelled x, y, and z the same way as before.)

Going back to Fig.1, P is the point that the camera is "staring at" or "focused on". We assign this point P the position vector $\mathbf{s}$ (s for "stare"). The camera is at position vector $\mathbf{c}$, and it looks in the direction $\mathbf{v} = \mathbf{s} - \mathbf{c}$. We remark that in a right handed system, the unit vectors along the axes $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ have the following cross products. These are $\mathbf{i} \times \mathbf{j} = \mathbf{k}$, $\mathbf{j} \times \mathbf{k} = \mathbf{i}$, and $\mathbf{k} \times \mathbf{i} = \mathbf{j}$. Swapping the order changes the sign so $\mathbf{k} \times \mathbf{i} = -\mathbf{i} \times \mathbf{k}$.

Using OpenGL coordinates, camera is initialised with $\mathbf{c} = (0, 0, 10000)$, $\mathbf{s} = (0, 0, 0)$ and $\mathbf{u} = (0, 1, 0)$. So, the above choices in gluPerspective means that an object in the OpenGL (x, y) plane ($z = 0$) that goes from x=-5000 to x=+5000 just occupies the entire screen.

We shall need a local coordinate system at the camera. There are three obvious choices. We shall look at these in turn. The first type we shall call xyz coordinates. The local y axis will be parallel to the OpenGL y axis, and the local xz plane is parallel to the OpenGL xz plane. The local x and z axes are not parallel to the OpenGL axes though. This is seen in Fig.2. In the local xyz camera we form the vector

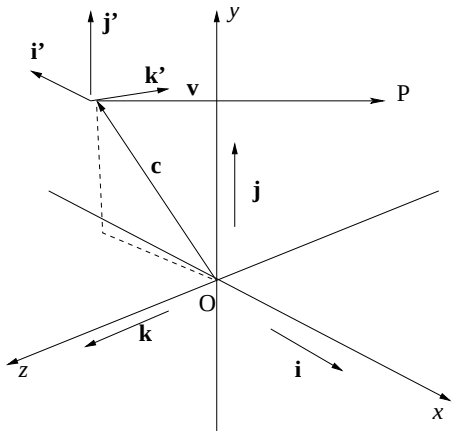

Figure 2: The local xyz type coordinates at the camera.

$$\mathbf{i}' = \frac{\mathbf{j} \times \mathbf{v}}{|\mathbf{j} \times \mathbf{v}|}. \tag{2}$$

then we have the basis vector $\mathbf{j}' = \mathbf{j}$, so $\mathbf{k}' = \mathbf{i}' \times \mathbf{j}'$. So, $\mathbf{v}$ is in the $\mathbf{j}'\mathbf{k}'$ plane.

The next type of coordinates we shall call "boom" coordinates (the boom being a long stick on which the camera is mounted, the camera look direction is towards the base of the stick). This basis set is useful if we are looking at an object centred at the origin and the camera is far enough from the origin to be outside.

We form the vector

$$\mathbf{i}' = \frac{\mathbf{c} \times \mathbf{j}}{|\mathbf{c} \times \mathbf{j}|}. \tag{3}$$

Then we use the vectors $\mathbf{k}' = -\mathbf{c}/|\mathbf{c}|$ and $\mathbf{j}' = \mathbf{k}' \times \mathbf{i}'$ as the other two local basis vectors. We may wish to have the camera fixed at the origin, in which case we reverse the directions

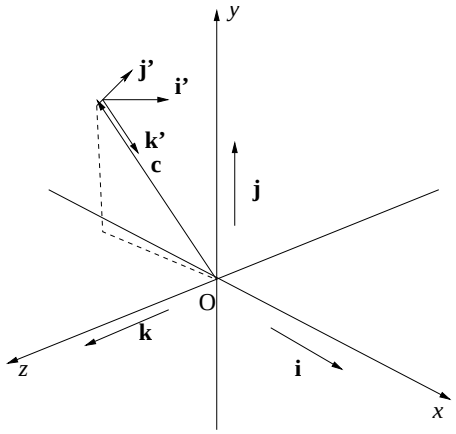

Figure 3: The local boom type coordinates, the camera is at the end of a boom.

of $\mathbf{i}'$ and $\mathbf{k}'$. We shall call this local coordinate set as "reverse boom".

Lastly, we shall call a third set of coordinates "flight" coordinates. The local coordinates are the roll pitch and yaw axes for the camera. So we use

$$\mathbf{i}' = \frac{\mathbf{j} \times \mathbf{v}}{|\mathbf{j} \times \mathbf{v}|}. \tag{4}$$

This is the pitch axis, and it is parallel to the xz plane. Then $\mathbf{k}' = \mathbf{v}/|\mathbf{v}|$ provides a roll axis. Finally, we can form the yaw axis $\mathbf{j}' = \mathbf{k}' \times \mathbf{i}'$. This local system is depicted in Fig.4. Now we have a local camera coordinates we can go on to discuss how we alter the camera position and view direction in real time as would happen in a first person computer game.

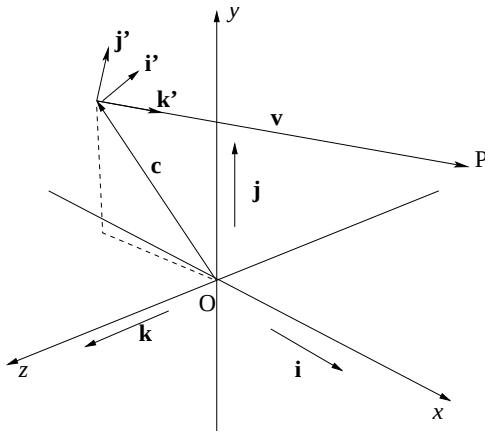

Figure 4: Local flight type coordinates give roll, pitch, and yaw axes.

3 The OpenGL Camera Modes and Controls

For an OpenGL programme to interact with the mouse, the keyboard, or say, the soundcard, it needs an interface with the computer. OpenGL itself does not have such an interface. This interface is mostly either GLUT (Graphics Library Utilities) or SDL (Simple Direct Layer). We shall use SDL.

3.1 Basic Structure

We shall refer to our OpenGL programme as "Prog". So, we shall have a file called "Prog.h", and the very first line in the C++ source file "Prog.cpp" shall be ' # include "Prog.h"'. We shall declare all our global variables at the beginning of Prog.cpp. These include, WIDTH, HEIGHT, and SCREEN_DEPTH. They are "const USHORT" (USHORT is a typedef for unsigned int), and WIDTH and HEIGHT are initialised at 800 and 600. These are to be the screen width and height in pixels. The SCREEN_DEPTH is set at 16 bits per pixel.

Surprisingly, the "main()" programme is just a few lines in another header file named Init.h. First of all main() initialises SDL, next it sets up some SDL things by calling Setup(), then a window is opened. The SizeOpenGLScreen functions sets up some OpenGL things including the "viewport", and makes that call to gluPerspective. As mentioned previously, this sets up a "View Volume" in the shape of a frustum. Any object within this volume is rendered. Here we set a view angle in the xz plane to 45 degrees, the aspect ratio to be the

same as WIDTH:HEIGHT, and then znear and zfar to be 100 and 150 000.

Next a camera object (CCAM Camera1) is created. This object is defined in Camera.h. Its member data are the camera position vector, the stare at position vector, and $\mathbf{j}$ and $\mathbf{j}'$. Next there is a call to Init(), which initialises such things as lighting. Then there is a call to EventLoop(Camera1). This is where the "action" takes place.

First of all, data can be read in here, then all the permanent objects to be drawn are defined or loaded into the program.

There is a bunch of SDL functions being used. It will suffice for the moment to know that they are "listening" to the keyboard and the mouse. There is a bool called quitit, which is initialised as false. After all is permanent data is loaded the 3D interactive stuff can start. This is done in a "while loop" which runs until a bool called quitit is set to be true. The quitit variable is set to be true as soon as an SDL function registers that the <Esc< key has been pressed. Once this is pressed the programme exits.

In this while(!quitit){} loop, there is a RenderScene function, which draws everything into a buffer (a chunk of memory). There are two buffers according to the way the above functions are set up. The contents of one buffer are displayed on the screen while the other buffer is being drawn to. Once all is drawn, this completed buffer is displayed on the screen and the other buffer will be cleared and drawn to. This is achieved by a call to SDL_GL_SwapBuffers() after the RenderScene function is finished.

To Summarise, the source for Prog is in Prog.cpp. There are various function prototypes and global data declared here and more globals declared at the start of Prog.cpp. Within Prog.h there are #includes for Init.h and Camera.h. These contain other function prototypes and function definitions. Init includes the definition of main(). The main() function's job is just to sets up various things before calling EventLoop(Camera1). This EventLoop function can read in data and set up the 3D world. Once this is done, we enter a "game loop" which makes any alterations to the 3D world on the fly, even if that is only altering the camera and view direction.

The SDL functions listen out for what's happening to the mouse and the keyboard, and various function calls are made depending on these mouse and keyboard events.

3.2 Rotations of Camera Position and View Direction

The header file Camera.h contains the functions that change the camera position and "stare at" point. We begin with rotations controled by the arrow keys. (The details may vary as to which functions are called on pressing which keys. We outline examples and explain what the functions do here, and will leave the rest to help screens specific to particular programs.)

We start off with the camera function *CamRotatePos1* and *CamRotatePos2*. The Pos1 version is called when the left or right arrow keys are down, and the Pos2 version is called if the up or down arrow keys are down. What happens is depicted in Fig.5. These rotations apply if the camera is in what we shall call *xyz mode* corresponding to Fig.2 in the previous section. In this mode, the camera's local y axis is always parallel to the OpenGL y axis, and the camera's local (x, z) plane is always parallel to the OpenGL (x, z) plane. The camera's local (x', z') directions may be rotated with respect to the OpenGL (x, z).

In Fig.5, the camera is at $\mathbf{c}$ and the stare point is at $\mathbf{s}$ so the view direction is $\mathbf{v} = \mathbf{s} - \mathbf{c}$. We could write $\mathbf{c} = \mathbf{s} - \mathbf{v}$. This is where the camera is before the rotation. The camera is to rotate *about the stare point, and not about the OpenGL origin unless that is the stare at point.* These CamRotatePos functions will keep the local basis y axis parallel to the OpenGL y axis. If A is a rotation operator, the position after the rotation will be $\mathbf{c} = \mathbf{s} - A\mathbf{v}$.

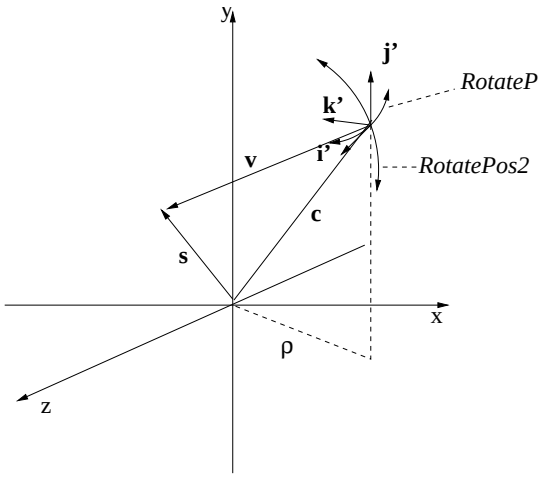

Figure 5: The Camera's RotatePos 1 and 2 functions in xyz mode. In xyz mode, RotatePos1 moves the camera around a verticle line passing through the stare at point. If in tethered mode as well, the camera rotates about a line of constant latitude.

In the CamRotatePos1, the rotation is in the (x, z) plane. This means we must rotate the local basis vectors $\mathbf{i}'$ and $\mathbf{k}'$ as well. In the CamRotatePos2 function, the rotation is in the vertical plane through the stare at point and the camera position. This means we must find the distance maked ρ in Fig.5. With this rotation *none* of the camera's local basis vectors is changed at all.

Things are rather like being in a cherry picker cradle with the stare at point at the base of the vehicle. If we press the left button, the cradle spins to the left, so the world seems to spin to the right. This changes our orientation with respect to the ground. If the up button is pressed, the cradle moves up, but it's orientation doesn't change. The person in the cradle remains upright.

In this xyz mode, we also have functions *CamRotateView1* and *CamRotateView2*. To continue the cherry picker analogy, View1 rotates the view vector from the cherry picker cradle in the (x, z) plane, as if the person in the cradle looks left or right (corresponding to the left and right arrow keys). This rotation does not change the orientation of the cradle, so the local basis vectors remain unchanged. CamRotateView2 corresponds to looking up or down (again corresponding to the up and down arrow keys). We note that cherry picker can "drive around" with the camera in the cradle.

We have another mode, which we shall call tethered. Here the "stare at" point is fixed at the origin. By analogy, the cherry picker has parked at the origin and put the handbrake on. In this mode, we can still be in *xyz* mode as well, or in *boom mode*. This corresponds to Fig.3. in the previous section. In this mode shall only rotate the camera, as the view shall always towards the base of the cherry picker. So, we have functions CamRotateBoomPos1 and CamRotateBoomPos2. The Pos1 version is called when the left or right arrow keys are down. Suppose we had a globe centred at the OpenGL origin, and we are in tethered mode.

In xyz mode the camera would move around lines of constant latitude. In boom mode, the left and right arrow kews would move the camera round in a great circle instead. That is, we rotate in the local (x', z') plane of Fig.3. instead of the OpenGL (x, z). Also, in the case of the up or down arrow keys being down, the camera will move along lines of constant longitude as in xyz mode, but now the local $\mathbf{k}'$ and $\mathbf{j}\prime$ vectors are rotated as well. In boom mode we also have a *CamBoomSpin* function using <Ctrl> and left or right arrows. This spins the camera in the local (xy) plane, and the local $\mathbf{i}'$ and $\mathbf{j}'$ vectors rotate with the camera.

In tethered mode, there is one last arrow function called *CamZoom* which moves the camera toward or away from the origin. This is called when <Ctrl> as well as the up and down arrows.

Now, we have yet more rotation functions, this time, they are controled via the mouse. These functions work as follows. SDL monitors any change in the mouse pointers' *screen coordinates*. If there is any change, the *SDLWarpMouse* function immediately "warps" the mouse back to the screen centre. However, we do get the screen coordinates of the mouse before this is called, and these two numbers are used as proxies for angles. For example, in xyz mode the x screen coordinate determines the rotation about the camera's y axis, and the y screen coordinate determines the rotation about the camera x axis. This is done by

the *MouseView* function. A similar function, that rotates the camera position instead of the viewpoint is *MouseLookat*.

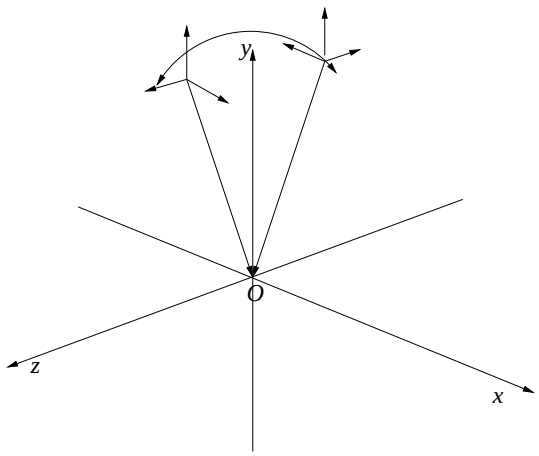

Figure 6: In both tethered and xyz mode the camera flips as we move from positive z to negative z.

There is an oddity in using xyz combined with tethered mode. This is depicted in Fig.6. We are telling OpenGL to always look at the Origin, so if we are moving in the (y, z) plane say, as we cross from -ve to positive z, OpenGL rotates us instantly through 180° in order to keep looking at the origin. If we keep pressing the up arrow key, we pass back to positive z, and the camera flips again, and so on. We can't get past the north pole! There is no such problem (if it can be called a problem) in boom mode.

3.3 Camera Translations

The camara also has free modes, where the camera roams at will. Two of these modes shall be called *wander* and *flight*. We use the name wander mode as it is similar to just "wandering about". If the user is not actively making the camera move, the camera stays still allowing the user to examine the view. In flight mode, the user is always on the move.

In the wander mode, the up and down arrows move the camera forward or backward in the direction of the view vector. The both wander and flight modes, the translation of the stare at point matches that of camera. So the stare at point constantly runs away when the camera is chasing it.

The camera keeps moving as long as the up or down arrows are down, and stops immediately the keys are released. If the left or right arrows are down, the camera rotates the stare at point around the camera position while staying still. Along with <Ctrl>, the up and down arrows let the camera look up and down. So, if the view is changed using MouseView or CamRotateView, and we then start moving again by pressing the up arrow say, we have changed direction of the camera's motion. This is because the camera always "chases" the view direction. So, unless we restrict the camera to 2D motion, we shall have complete 3D freedom.

In wander mode, we can rotate the camera about the stare at position as in tethered (x, y, z) mode. This is done using the <Shift> key along with the arrow keys. The <Ctrl> key combined with the left and right arrows translates the camera and the stare at point left and right. When combined with the up and down arrows, the camera and the stare at point translate up and down.

In *flight* mode, we may still use (x, y, z) mode, but if we do, there are differences. The camera is always in motion. The camera can accelerate or brake using the 'a' and 'z' keys. This time, the up and down arrows will rotate the camera view direction up and down. The senses are reversed, so the up arrow is like moving the joystick forward, which pitches the camera view downwards.

There are other functions, *CamPlaneRoll* and *CamPlanePitch* which are used in *aeroplane* mode. These are operated by the arrow keys, but the functionality has not been extended to mouse controls yet.