Title: How can I use gluLookAt ()to specify viewer movement?

From: kindy
Date: 11/05/2002 09:05PM PST
Status: Waiting for Answer
Points: 100

I am a beginner of openGL, my question is that I am going to create a scene and initially viewer is at (0,0,0) and looks at (0,0,1), suppose Y is vertical axis, I should move viewer parallel to the XZ plane, I don't know how to make viewer move forward, backward, left , right and rotate using gluLookAt(). I am not sure how to compute the arguments used in gluLookAt(), would any one like to give me some help?

Thanks a lot

Question History

Comment from UncleSquirrel 11/07/2002 12:47AM PST
A couple of things:

1. By default, in OpenGL your viewpoint should be at (0,0,0) but looking down the -Z axis toward (0,0,-1), not (0,0,1).

2. Second, although gluLookAt() is very powerful and useful, you may find that you want to end up moving & aiming the camera yourself using calls to glTranslatef() and glRotatef() instead.

3. Make sure you're setting up your projection viewmatrix correctly. (Are you already seeing objects in 3d space okay?) You'll probably want to use gluPerspective() for this.

4. As far as gluLookAt() goes, here's the basic rundown:

```
gluLookAt(
GLdouble eyex, // x-coordinate of camera's position
GLdouble eyey, // y-coordinate of camera's position
GLdouble eyez, // z-coordinate of camera's position
GLdouble centerx, // x-coordinate of look-at point in 3d space
GLdouble centery, // y-coordinate of look-at point in 3d space
GLdouble centerz, // z-coordinate of look-at point in 3d space
GLdouble upx, // x-component of a vector pointing "up" on the screen
GLdouble upy, // y-component of a vector pointing "up" on the screen
GLdouble upz, // z-component of a vector pointing "up" on the screen
);
```

The <eye> and triples are relatively self-explanatory (though ask if you have questions about them).

The one that's probably most confusing is the <up> vector. Consider this: by providing two points in space, <eye> and , we establish (A) where the camera is at and (b) at point in space at which the camera is aimed. Now imagine a straight line drawn in 3d space going from point A to point B. We have NOT specified whether the camera is "right-side up", "upside-down", or any other weird and arbitrary rotation (roll) around this line. In order to do this, we specify an <up> vector for the camera. Essentially, we are saying, "if there were an antenna pointing out of the top of the camera, this is the direction it would be pointing".

As for your example (moving and turning on the XZ plane, with +Y as up): If you weren't able to turn around, the problem would be a simple one. Forward movement would simply mean "make the Z component of <eye> and smaller (more negative)". Backward movement means "make <eyez> & <centerz> bigger <more positive>". Strafing left means "make <eyex> and <centerx> more negative"; strafing right means "make <eyex> and <centerx> more positive". (Modifying <eyey> and <centery> would make the character rise and fall vertically.)

Last but not least, your <up> vector would ALWAYS be (0,1,0) (i.e. the camera is "right-side up", antenna pointing in the +Y direction).

However, when turning becomes involved in a Quake-style fashion (which is how I read what you're describing), using gluLookAt() actually becomes a hassle. If instead we use glTranslatef() and glRotatef(), we can get a simpler solution.

Let's say we have the following information for the player / camera. I'm not sure if you're in C or C++ so we'll stick to floats:

```
float position[ 3 ]; // eye position in 3d space (0=x, 1=y, 2=z)
float yawDegrees; // degrees of rotation, counter-clockwise around +Y.
float pitchDegrees; // degrees of bending-over ("pitch" angle)
```

...and we have a macro (or, preferably, an inline function) to convert from degrees to radians:

```
#define PI 3.14159265358979323846264338327 95
#define DEG2RAD( x ) ((x) * PI / 180.0)
```

When the camera moves forward <dist> units:

```
position[ 0 ] -= dist * cos( DEG2RAD( yawDegrees ) );
position[ 2 ] -= dist * sin( DEG2RAD( yawDegrees ) );

When the camera strafes to the left <dist> units:
position[ 0 ] -= dist * cos( DEG2RAD( yawDegrees + 90 ) );
position[ 2 ] -= dist * sin( DEG2RAD( yawDegrees + 90 ) );

When the camera moves back <dist> units:
position[ 0 ] -= dist * cos( DEG2RAD( yawDegrees + 180 ) );
position[ 2 ] -= dist * sin( DEG2RAD( yawDegrees + 180 ) );

When the camera strafes to the right <dist> units:
position[ 0 ] -= dist * cos( DEG2RAD( yawDegrees + 270 ) );
position[ 2 ] -= dist * sin( DEG2RAD( yawDegrees + 270 ) );

When the camera moves straight up <dist> units:
position[ 1 ] += dist;

When the camera moves straight down <dist> units:
position[ 1 ] -= dist;

When the camera turns to the right <turnAngle> degrees:
yawDegrees -= turnAngle;

When the camera turns to the left <turnAngle> degrees:
yawDegrees += turnAngle;

When the player looks up <pitchAngle> degrees:
pitchDegrees += pitchAngle;

When the player looks down <pitchAngle> degrees:
pitchDegrees -= pitchAngle;

Then, when it comes time to draw your scene, place the OpenGL camera like such:

/// Clear the camera's translation from the previous render frame
glLoadIdentity();

/// Set up the perspective view
gluPerspective( ..... ) // you should already have a line like this in your code

/// Move the camera to the player's position
glTranslatef( -position[0], -position[1], -position[2] );

/// Rotate the view by <yawDegrees> around the Y-axis:
glRotatef( yawDegrees, 0.0f, 1.0f, 0.0f );

/// Rotate the view by <pitchDegrees> around the new, rotated X-axis:
glRotatef( pitchDegrees, 1.0f, 0.0f, 0.0f );

/// Now draw your objects in 3d space.
```

Note that we had to use DEG2RAD to convert our angle from degrees (0-360) to radians (0-2xPI), since the cos() and sin() functions take angles in radians. However, we did not have to do this for our glRotatef() command, since it takes angles in degrees.

I'm sure this is probably confusing and/or contains errors, but it's 2:20am so I claim no responsibility whatsoever. ;-)

Feel free to ask for clarification on any points that are confusing... admittedly, this is a poor overview of the vast subject of 3d view transformations, which deserves 20 pages of text rather than 2. In a large-scale 3d game - or one involving complex cameras and/or cameras floating in free space (with no discernible "up" or "down") - there are actually much better ways of treating angles than with yaw/pitch/roll (called Euler angles), namely Quaternions and Matricies.

cheers,
-sq

p.s. Come to think of it, things might not be all that bad using gluLookAt() after all. We'd basically just have a 3d "forward" vector for the player (which we'd update whenever yaw-ing or pitch-ing occurred) and then set a view target at an arbitrary remote point along that vector. However, it's still probably conceptually simpler to think in terms of Euler angles for the time being.

Comment from Peaker 11/15/2002 06:06PM PST
As UncleSquirrel said, gluLookAt can be still be used, however not with a "forward" vector, but with a forward matrix, and a position. Basically, the forward matrix is just the 3 vectors:
1. Right
2. Up
3. Forward

And to use it with gluLookAt, you'd have to call:

```
gluLookAt(position.x,
position.y,
position.z,
position.x + forward.forward.x,
position.y + forward.forward.y,
position.z + forward.forward.z,
forward.up.x,
forward.up.y,
forward.up.z);
```

The forward matrix is basically the Identity Matrix, multiplied by whatever transformation the camera has on the modelview matrix. This means that looking forward, the camera is :

right =1,0,0
up =0,1,0
forward=0,0,1

Looking backwards, it is:

right =1,0,0
up =0,1,0
forward=0,0,-1

For games like Quake, it is not easier to use, but for games like descent or such, where the camera is completely free to rotate in any direction, and forward always means camera-forward, it is easier to maintain.

You can rotate the camera by multiplying its matrix by a rotation matrix of any direction, and it will rotate relatively to its current rotational status (useful when the player hits "rotate right", and expects it to rotate relatively to himself, and not the absoloute axis), which is impossible to (reasonably) do with angluar rotation coordinates.

Its also easy to move forward, strafe, etc, simply by adding one of the camera's vectors to the position.

Its also easy to calculate modelview transformations of points in 3d space, just by multiplying them by the handy camera matrix.

This method, however, involves loss of precision which may make the camera matrix, since it is not regenerated every time, lose precision. If you normalize the matrix every now and then (by normalizing each of the vectors, and ensuring their orthogonality), however, it will be resolved.

Comment from kindy 11/20/2002 05:36PM PST
Thank all of you for the kind reply!I need some while to fully understand those.