

CS 314: Lookat, Hierarchical Modeling

Robert Bridson

September 23, 2008

1 Lookat

A common problem programmers run into is rendering a black screen because a bug in their transformations from world space to camera space ended up with the camera in the wrong position or pointed the wrong way. The objects might be there, but behind the camera or otherwise off-screen unintentionally. One helpful tip is to use a large field-of-view initially to have a better chance of spotting geometry that's there. Another is to use **Lookat**.

Lookat is a popular way of constructing the world-to-camera transformation that is nearly fool-proof in terms of making sure you see objects on screen. It's not always appropriate for an application, but can be very useful in debugging. In OpenGL, this is done with a call to `gluLookAt`.

It works by specifying three things:

- the location of the camera in world space (let's call it \vec{c} for now)
- the point you want the camera to look at (say \vec{p})
- an “up-vector” that should be orthogonal to what the camera sees as horizontal (say \vec{u}); this is almost always specified as $\vec{u} = (0, 1, 0)$, the usual notion of “up” in the world.

The first part of the transformation (the rightmost matrix in the product) is a translation by $-\vec{c}$ which will translate the camera in world space to the origin in camera space. The second part is a rotation; instead of specifying it with angles and axes, however, we'll directly construct a right-handed orthonormal basis that defines a rotation matrix as we saw in earlier notes. In camera space, the camera looks along the $-z$ axis, so we want the negative of the last basis vector to point (in world space) from \vec{c} to \vec{p} . Normalizing

this vector gives the last basis vector as:

$$\vec{t} = \frac{\vec{c} - \vec{p}}{\|\vec{c} - \vec{p}\|}$$

We want the first basis vector (the camera space x -axis) to be orthogonal to \vec{t} and also horizontal, i.e. orthogonal to \vec{u} as well. The cross-product comes in handy here, with a normalization again:

$$\vec{r} = \frac{\vec{u} \times \vec{t}}{\|\vec{u} \times \vec{t}\|}$$

I was vague in class about whether \vec{r} should be this vector or the negative of it, i.e. if it should be $\vec{u} \times \vec{t}$ or $\vec{t} \times \vec{u}$: this is the correct way around, designed so that we get back the standard basis if the camera should be pointing down the world z -axis and the up-vector is the world y axis.

Finally, like any right-handed orthonormal basis the final vector is defined with a cross-product, in this case:

$$\vec{s} = \vec{t} \times \vec{r}$$

If you put \vec{r} , \vec{s} and \vec{t} as the rows of a 3×3 orthogonal matrix Q , and expand it as usual to 4×4 for homogeneous coordinates, we get the second part (leftmost matrix) of the Lookat transformation.

2 Hierarchical Modeling

We'll wrap up our initial discussion of transformations—there are only a few more technical issues to cover later on—with a look at modeling objects in a scene, and the related infrastructure a typical graphics API such as OpenGL provides.

So far we have assumed the user could specify the entire geometry of the scene in world space coordinates. However, it's pretty common for a scene to be composed of movable objects: for example, in a game there might be a static building, but characters will move around within that building as gameplay proceeds. Constantly updating the world space coordinates of every moving object is not only potentially slow and tedious, but can incur serious rounding errors from the nature of floating point: if you start off with a model and keep applying small rotations, gradually the rounding errors committed with every update will distort the shape of the model into something quite different.

2.1 Object Space

A much superior approach is to introduce more coordinate systems, and use the transformation matrices we've developed so far to go between them. In particular, we can talk storing an object—which at some

point will basically be a triangle mesh an artist built—in **object space**, a coordinate system that has no meaning other than it was convenient for the artist. Quite often the approximate centre of the object will be at the origin in object space, or some other special point on the object, and it will be oriented naturally (if the object was a human, probably the head will be along the positive y -axis in object space, and the face will point along the z -axis say.) The program can then construct a 4×4 matrix which puts the object in the world, transforming object space coordinates to world space coordinates based on where the object should be and how it should be oriented, etc. Besides translations and rotations, it's also fairly common to use scalings to allow changes to the size of the object. The object's geometry in world space remains untouched—it's read-only, unchanged from the file it was stored in on disk—and only the transformation matrix mapping it to world space changes. This transformation usually gets multiplied into the model-view transformation, so it can directly transform object geometry to camera space.

With that in place, take a moment to follow the code path of a vertex in an object. Its constant object space coordinates, that never change, are first multiplied by the model-view matrix, which is constructed as the product of the object-to-world transformation and the world-to-camera transformation. This gives camera space coordinates of the vertex, which are multiplied by the projection matrix to get normalized device coordinates. This 4D vector is now homogenized, then mapped to pixel coordinates, and finally is sent to rasterization.

One of the great things this notion of object space gives us right away is **instancing**. If we have a hundred identical enemy drones in a game, for example, instead of having to store and update the geometry in world space for all hundred of them, we can keep just one copy in object space (that remains unchanged) and only update/construct a hundred separate object-to-world 4×4 matrices. We get a hundred copies (or **instances**) of the same object almost for free. This is a huge savings in some cases.

Note that to avoid accumulated round-off problems in rotation mentioned above, it's important to store and update underlying parameters such as Euler angles for the object-to-world transformations, then compute the actual 4×4 matrix from them. (While you may accumulate rounding error in the Euler angles, at least the matrix you produce this way will be orthogonal up to machine precision and won't badly distort your geometry the same way that happens when you multiply a long sequence of incremental rotation matrices and get something far from orthogonal.)

2.2 Matrix Stacks

However, it's a bit bothersome that we need a different model-view matrix for every object in the scene, when it naturally can be thought of as a product of a particular object-to-world transformation (different for each object) and a world-to-camera transformation (the same for everything in the scene). The widely

adopted solution is to maintain a **matrix stack**. After creating the world-to-camera transformation matrix, we push it on the **model-view stack** to keep it around. When we go to render an object, we create a new model-view matrix by multiplying the top of the model-view stack (at the moment, the world-to-camera transformation) with the object-to-world transformation, and push this onto the top of the model-view stack. We then process all the vertices in the object, and when we are finished with it and need to move on, we simply pop the top off the model-view matrix—returning us to the basic world-to-camera matrix—and proceed to the next object.

It's important to keep in mind the correct order of multiplication, by the way! If the top of the model-view stack is the matrix M_{wc} (a matrix mapping world space coordinates to camera space coordinates), and we want to multiply it with the matrix M_{ow} (a matrix mapping object space coordinates to world space coordinates) to get the new model-view matrix, we need the product $M_{wc}M_{ow}$ and not the other way around. (Think of which matrix should transform a vector on the right, containing the object space coordinates, first.)

We can also define a similar matrix stack for projection as well, and OpenGL supports this as well, but it's not nearly as useful.

2.3 Hierarchies

We've now worked up enough to conveniently deal with objects moving around in the scene—but only if they are perfectly rigid objects. We can't produce a 4×4 matrix which lets a robot character bend its limbs or twist their neck, for example, or a helicopter that can spin its rotor. However, if you consider these examples more closely, even though the object as a whole isn't rigid, they can be broken up into rigid pieces.

This is the essential idea of **hierarchical modeling**: break a complicated object up into smaller pieces, stored separately, and then introduce transformation matrices to assemble them together into the current configuration you want—and to do this recursively with the individual pieces if they are still complicated.

In the helicopter example, we could store the geometry of the main rotor separately from the body of the helicopter, in its own coordinate system. To draw the helicopter at a particular instant in time, we would transform the body of the helicopter to world space as before, but we would use more transformations for the rotor: first a translation and rotation to put it in the right place in the body's coordinate system, and then apply the rest of the object-to-world space (and then to camera space etc.) transformation.

This produces a natural tree structure, or hierarchy, on the whole rendering. Every node in this tree is a coordinate system: the root is world space, its children are the object spaces of the main objects in the scene, their children if they have any are the local coordinate systems of the component parts, and so on. The edges in the tree express the transformation from the child's coordinate system to the parent's coordinate system. The actual geometry (vertices, triangles, etc.) might only be stored at the leaves.

To render any leaf node geometry, its model-view matrix must be computed by multiplying the transformations on all the edges of the path up the tree to the root. (The first edge transforms the leaf's coordinates to its parent's coordinate system, the next edge up transforms that to the grandparent's coordinate system, and so on until you get to the root's coordinate system which is world space. Finally you need to multiply in the world-to-camera space transformation to get the complete model-view matrix.

This can be done much more efficiently and elegantly with a tree traversal, using the matrix stack we introduced above. Beginning with the world-to-camera transformation in the model-view stack, and after rendering any geometry stored directly in world space, the children of the root are rendered recursively as follows:

- Take the top of the model-view stack, multiply it by this node's transformation matrix (to its parent) on the right and push the product on to the stack.
- Render any geometry directly stored at this node using the top of the stack as the model-view matrix.
- Recursively render any children of this node.
- Pop the top from the model-view stack before returning.