Billboarding Tutorial

by
António Ramires Fernandes

1. Introduction

Billboarding is a technique that adjusts an object's orientation so that it "faces" some target, usually the camera. The word *faces* is in quotes since it can have several meanings, as the tutorial will show.

This technique is quite popular in games and applications that require a large number of polygons. Regardless of how fast graphic cards get (Thanks Nvidia!), it seems that it is never enough. Games take advantage of the hardware advances to push things further to the next level in complexity and detail (In turn hardware manufacturers try to keep up with games designers in a never ending cycle). 3D graphics programmers in general, and game programmers in particular, have a constant struggle with the amount of polygons that can be displayed with a decent frame rate.

Billboarding can be used to cut back on the number of polygons required to model a scene by replacing geometry with an impostor texture. A classic example is a tree. Consider how many polygons would be required to get a decent representation of a tree (specially in the Spring!). Billboarding allows us to replace the geometry with a single texture applied on a quad (or two triangles). Billboarding guarantees that the texture is always facing the camera, therefore the user never realizes that the "tree" is in fact a flat texture quad, unless the user flies over it in which case the illusion is broken.

Beware that shading will not be correct if the same texture is used for all orientations. A 3D tree viewed from different locations will exhibit a different look assuming a fixed, or at least non-correlated light movement with the camera. For a 2D billboard, when using the same texture for all orientations, it will look as if the light is moving with the camera. It may be the case that the user won't notice this shading discrepancy, but be prepared for it. One possible solution is to have multiple textures taken from different views. However, unless a prohibitive number of textures is used, a popping effect will be noticeable when the texture changes. Blending may be a solution for this problem but this is outside the scope of this tutorial.

Billboarding can be used to give the illusion of a 3D object without compromising the polygons budget. However billboarding is not limited to saving polygons. In general, the billboarding technique can be applied to make any object face a target, where the target may be the camera, another object, or simply a 3D location.

In this latter scenario billboarding isn't used to save polygons but to achieve some goal related to the purpose of the application, for instance the enemy might always be facing you, or a football player is always facing the ball.

Two types of billboarding are presented: cylindrical and spherical. In the spherical version there is no restriction to the orientation of the object, whereas in the cylindrical approach the rotation of the object is restricted to a vector, usually the positive direction of the Y axis.

In this tutorial several techniques to implement billboarding will be covered. The first one is a very simple method but it doesn't provide a real billboard. It is a cheating approach that is very easy to implement. A faster approach for quad billboards that provides exactly the same results is provided afterwards. It is not as easy to implement but there can be a significant difference in speed when using a large number of quad billboards.

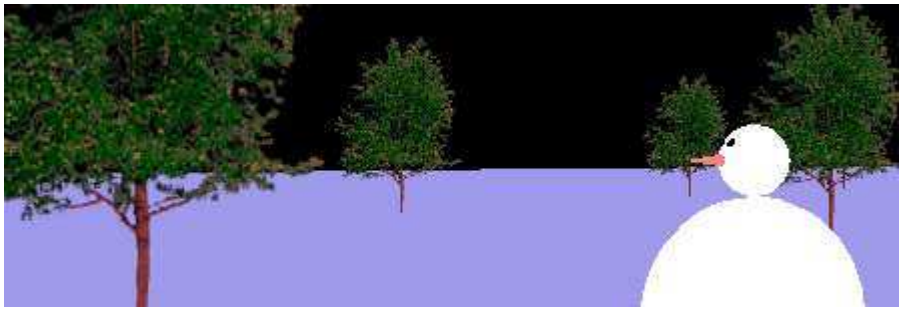
Finally true billboards are presented. These methods allow you to set a billboard to face the camera. they can also be used to have an object face another object or point in space. In this sense they are more general than the cheating versions.

2. Cheating - Fast and Easy Spherical Billboards

The technique about to be presented isn't true billboarding. Instead it provides a cheap way of achieving an approximation that may be good enough in some applications.

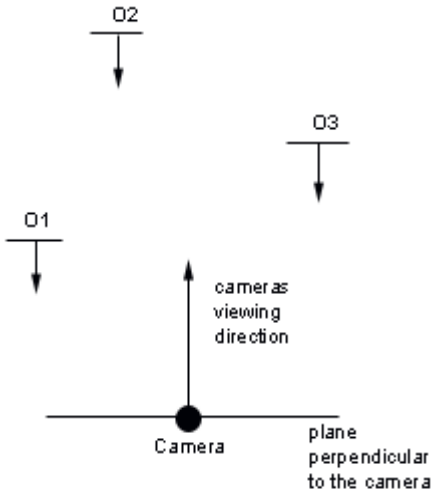
The source code for this tutorial is an extension to one of the projects on the GLUT tutorial. If you're not familiar with GLUT then maybe you should check it out to get a better grasp of the source code.

The following image is a screenshot of the demo. It shows a snowman plus some billboarded trees. Although it is hard to tell, the trees aren't facing the camera, although it is hard to tell. Instead they are facing a plane perpendicular to the cameras viewing direction.



The next diagram gives a clearer picture of what's going on. The black circle at the bottom represents the camera. The vector starting from the camera is the current viewing direction. This vector defines the orientation of the plane perpendicular to the

cameras viewing direction. The objects in the scene, O1..O3, have been rotated so that their front facing vector is pointing to this plane. It is assumed that the objects in the scene are positioned in the local origin, and that they're facing the positive Z axis.



An approach to achieve this effect requires the modelview matrix. As you're aware the modelview matrix stores the geometric transformations, rotations, scales and translations, you do. This matrix contains the required transformations to change the coordinates you input, world coordinates, into camera coordinates.

$$M1 = \begin{bmatrix} a0 & a4 & a8 & a12 \\ a1 & a5 & a9 & a13 \\ a2 & a6 & a10 & a14 \\ a3 & a7 & a11 & a15 \end{bmatrix}$$

The top 3 values of the right column provide the current position of the local origin relative to the camera's position and orientation. The top 3x3 submatrix contains the scaling and rotation operations. Setting this submatrix to the identity matrix effectively reverses these operations, so that the cameras viewing direction is aligned with the worlds Z axis. The up and right vectors of the camera are also aligned with the worlds axis, which means that this type of billboard acts like a cheat to spherical billboarding. The center of the billboard is the local origin. If you're planning to do your trees with this type of billboarding then when the user looks up or down the trees will bend forwards and backwards as well, this may be a little awkward. This is because the orientation of the billboard depends on the camera's orientation as opposed to the camera's position.

$$M1 = \begin{bmatrix} 1 & 0 & 0 & a12 \\ 0 & 1 & 0 & a13 \\ 0 & 0 & 1 & a14 \\ a3 & a7 & a11 & a15 \end{bmatrix}$$

In practice, setting the submatrix to the identity matrix means the your object won't suffer any rotation when it is rendered. Therefore it will be rendered in the same way as if the camera was looking down the negative Z axis.

Note that scaling is also reversed, so if you wanted a tall tree you're not going to get it with this method, unless you scale after you changed the modelview matrix.

The first step is to save the current modelview matrix. Afterwards get the modelview matrix. Then reset the top 3x3 submatrix to the identity matrix. Load the matrix in to the OpenGL state machine, and render your object in the local origin. Finally restore the original modelview matrix.

```
float modelview[16];
int i,j;

// save the current modelview matrix
glPushMatrix();

// get the current modelview matrix
glGetFloatv(GL_MODELVIEW_MATRIX , modelview);

// undo all rotations
// beware all scaling is lost as well
for( i=0; i<3; i++ )
        for( j=0; j<3; j++ ) {
                if ( i==j )
                        modelview[i*4+j] = 1.0;
                else
                        modelview[i*4+j] = 0.0;
        }

// set the modelview with no rotations and scaling
glLoadMatrixf(modelview);

drawObject();

// restores the modelview matrix
glPopMatrix();
```

With the above snippet of code, the object will be rotated to face the plane perpendicular to the cameras viewing direction. The centre for this rotation is the local origin.

The above code can be divided in two functions, besides the rendering function, *drawObject*. The first function will setup the modelview matrix, and the second will restore the current matrix.

```
void billboardCheatSphericalBegin() {

        float modelview[16];
        int i,j;

        // save the current modelview matrix
        glPushMatrix();
```

```
        // get the current modelview matrix
        glGetFloatv(GL_MODELVIEW_MATRIX , modelview);

        // undo all rotations
        // beware all scaling is lost as well
        for( i=0; i<3; i++ )
            for( j=0; j<3; j++ ) {
                if ( i==j )
                    modelview[i*4+j] = 1.0;
                else
                    modelview[i*4+j] = 0.0;
            }

        // set the modelview with no rotations
        glLoadMatrixf(modelview);
}



void billboardEnd() {

        // restore the previously
        // stored modelview matrix
        glPopMatrix();
}
```

The source code for rendering a billboard object becomes:

```
billboardCheatSphericalBegin();
    drawObject();
billboardEnd();
```

As mentioned before, scaling operations are lost when you set the new modelview matrix. If you want to scale your objects you can do it after setting up the billboard as in the following snippet.

```
billboardCheatSphericalBegin();
    glScalef(1,2,1);
    drawObject();
billboardEnd();
```

---

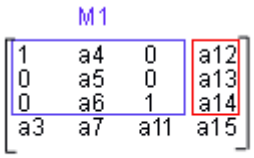
3. Cheating - Cylindrical Billboards

As mentioned in the previous section, the modelview matrix contains the transformations required to perform the change of coordinates between local coordinates and world coordinates. It has been shown that by replacing the top 3x3

submatrix with the identity matrix a cheating version of spherical billboarding can be obtained. But what about Cylindrical billboarding?

Let us analyze in depth what is actually the structure of the top 3X3 submatrix. This matrix contains 3 columns, and each has a special meaning. The first is the right vector, the second column is the up vector and the 3rd represents the lookAt vector. These vectors are relative to the cameras orientation. Setting the top 3x3 submatrix to the identity matrix effectively causes the camera's coordinate system to be aligned with the world's coordinate system.

In order to achieve a (cheating) Cylindrical Billboard, it is sufficient to align the right and lookAt vectors, leaving the up vector unchanged. The difference between this version and the spherical one, presented in the previous section, is that when the camera looks up or down the billboard won't move. The billboard will only be rotated when the camera looks right or left.



This type of billboards can be applied to trees. Trees don't bend backwards and forwards as the camera looks down or up because the up vector is fixed, so it doesn't make much sense to use a spherical billboard. Beware that the illusion is broken when you fly over the tree. In this case you'll see your tree getting thinner and thinner and all is lost. However if you're application keeps you on the terrain then cylindrical billboarding is a good option.

The code for cheating cylindrical billboards is almost identical to the spherical version.

```
float modelview[16];
int i,j;

// save the current modelview matrix
glPushMatrix();

// get the current modelview matrix
glGetFloatv(GL_MODELVIEW_MATRIX , modelview);

// The only difference now is that
// the i variable will jump over the
// up vector, 2nd column in OpenGL convention

for( i=0; i<3; i+=2 )
        for( j=0; j<3; j++ ) {
                if ( i==j )
                        modelview[i*4+j] = 1.0;
                else
                        modelview[i*4+j] = 0.0;
        }

// set the modelview matrix with the
// up vector unchanged
glLoadMatrixf(modelview);
```

```
drawObject();

// restores the modelview matrix
glPopMatrix();
```

As we did before, the above code can be divided in two functions, besides the rendering function, *drawObject*. The first function will setup the modelview matrix, and the second will restore the current matrix.

```
void billboardCheatCylindricalBegin() {

        float modelview[16];
        int i,j;

        // save the current modelview matrix
        glPushMatrix();

        // get the current modelview matrix
        glGetFloatv(GL_MODELVIEW_MATRIX , modelview);

        for( i=0; i<3; i+=2 )
            for( j=0; j<3; j++ ) {
                if ( i==j )
                    modelview[i*4+j] = 1.0;
                else
                    modelview[i*4+j] = 0.0;
            }

        // set the modelview matrix
        glLoadMatrixf(modelview);
}


void billboardEnd() {

        // restore the previously
        // stored modelview matrix
        glPopMatrix();
}
```

The source code for rendering a billboard object becomes:

```
billboardCheatCylindricalBegin();
    drawObject();
billboardEnd();
```

Again, beware with scaling operations. Scaling in the X and Y axis will be undone

using this approach, but scaling in the Z axis will persist. the best option is to scale after you call *billboardCheatCylindricalBegin()*.

4. Cheating - Faster but not so easy

As mentioned before to get a cheating version of a billboard it is enough to reverse the orientations of the top 3x3 submatrix from the modelview matrix. The previous sections achieved this by setting this submatrix such that the appropriate transformations were reversed.

In here an alternative approach is presented. This is also a popular approach, it even got mentioned in the book OpenGL Game Programming.

Instead of changing the modelview matrix, the vertices of the billboard are manually transformed. This transformation effectively reverses the orientations present in the modelview matrix. The end result is the same as the one presented in the previous sections.

The vertices of the billboard are defined using up and right vectors that reverse the orientations of the modelview matrix. These vectors can be extracted from the inverse of M1.

$$M1 = \begin{bmatrix} a0 & a4 & a8 & a12 \\ a1 & a5 & a9 & a13 \\ a2 & a6 & a10 & a14 \\ a3 & a7 & a11 & a15 \end{bmatrix}$$

Fortunately, for an orthogonal matrix, the inverse is equal to the transpose. Is M1 orthogonal? It should be. If you're only using *gluLookAt* to perform you're camera movements, or just doing translations and rotations to the camera, then M1 is orthogonal. If, on the other hand you mess around with the modelview matrix then you may end up with M1 not being orthogonal.

Just in case you're wondering what the $%*# is a transpose, a matrix A is a transpose of B, AT = B, if aij = bji, for every element.

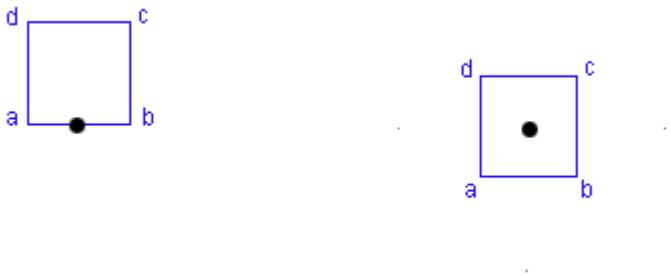
$$M^{-1} = \begin{bmatrix} a0 & a1 & a2 \\ a4 & a5 & a6 \\ a8 & a9 & a10 \end{bmatrix}$$

So what is an orthogonal matrix? As mentioned before a square matrix Q is orthogonal if Q x QT = I, where QT is the transpose of Q and I is the identity matrix. In practice this means that the multiplying any two different columns or rows returns zero. This means that the cross product is zero, and therefore that the vectors are at right angles. It also means that multiplying a vector by itself should give 1, i.e. the vectors are normalized. So now that you know you can write your own routine to see if a matrix is orthogonal.

So we can easily get the inverse of M1. The first column of the inverse of M1 is the required right vector, and the second column represents the up required vector.

```
right = [a0,a4,a8]
up =    [a1,a5,a9]
```

Two examples of a quad billboard are now presented to show how to define the vertices. Assume that the quads edges have a length *size*.



First a quad with the center at the bottom origin is presented (left figure).

```
a = center - right * (size * 0.5);
b = center + right * size * 0.5;
c = center + right * size * 0.5 + up * size;
d = center - right * size * 0.5 + up * size;
```

The right figure represents a quad with a centre on the middle of the quad. Its vertices are defined as:

```
a = center - (right + up) * size;
b = center + (right - up) * size;
c = center + (right + up) * size;
d = center - (right - up) * size;
```

When compared to the previous method you only get the modelview matrix once per frame. On the other hand you have to manually transform the vertices of the billboard. Overall this method is faster for quads, only 4 vertices, but harder to implement. If however you try to billboard an object with a larger number of vertices then the previous method could perform better.

In the source code a function is provided to extract the up and right vectors

void l3dBillboardGetUpRightVector(float *up, float *right);

Parameters:

up - an array of 3 floats. The function sets this to be the up vector
right - an array of 3 floats. The function sets this to be the right vector

---

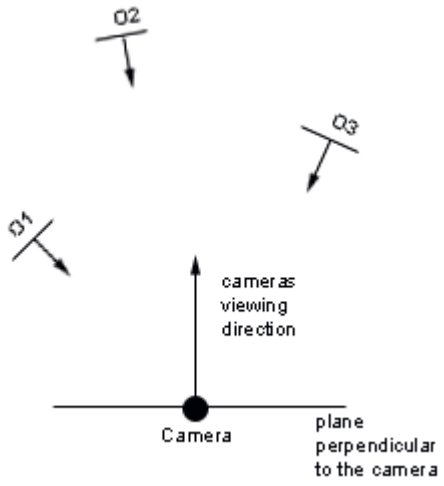
The source code is as follows:

```
void l3dBillboardGetUpRightVector(float *up,float *right) {

        float modelview[16];

        glGetFloatv(GL_MODELVIEW_MATRIX, modelview);

        right[0] = modelview[0];
        right[1] = modelview[4];
        right[2] = modelview[8];

        up[0] = modelview[1];
        up[1] = modelview[5];
        up[2] = modelview[9];
}
```

If what you're after is the cylindrical version of this method then just set the up vector to [0,1,0]. You can use the function *l3dBillboardGetRightVector*, provided in the source code, to get the right vector.

5. Cylindrical Billboards

True cylindrical billboarding constraints the objects rotation to an axis, as in the cheating version, but the look at vector from the object will be rotated in the camera's direction, restricted to the plane defined by the up vector, and the center of the local origin. The following image shows what happens.



In the previous example, a user walking in the ground, the trees would correctly face the user.

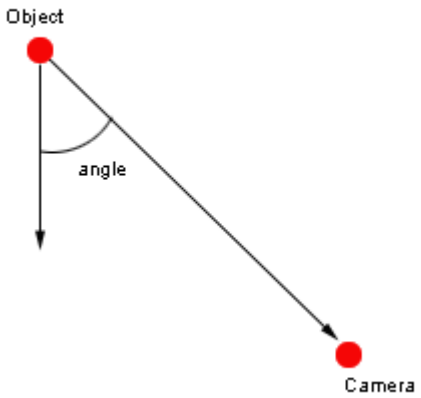
The method presented in here requires that you know where the object is in world coordinates, as well as the targets position, also in world coordinates. Usually the target is the camera's position which known. However the same may not be true for the object. If the object is positioned in the world through geometric transformations, such as translations and rotations, then keeping track of its position in world coordinates can be a hard task. We'll come back to this issue in a later section.

Assuming that both the objects and target positions are known in world coordinates, then billboarding becomes a simple geometrical operation.

For simplicity reasons it is assumed that the object has the following vectors:

- right vector = [1,0,0]
- up vector = [0,1,0]
- lookAt vector = [0,0,1] In practice this means that the object is drawn in the local origin, and it is looking along the positive Z axis.

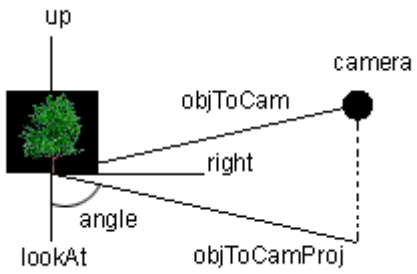
In order to orientate the object a rotation is performed around the up vector, by the angle between the lookAt vector and the vector from the object to the camera.



The vector from the object to the camera can be defined as

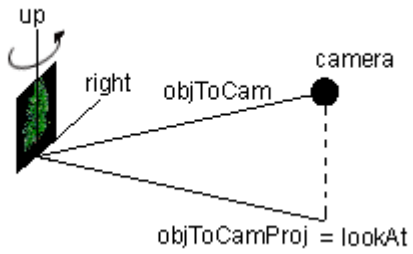
- objToCam = CamPos - ObjPos The vector *objToCamProj* is the projection of *objToCam* in the XZ plane. Therefore its Y component should be set to zero.

If *objToCamProj* is normalized then the inner product between *lookAt* and *objToCam* will allow the computation of the cosine of the angle. However knowing the cosine alone is not enough, since the cos(a) = cos(-a). Computing the cross product as well allows us to uniquely determine the angle. The cross product vector will have the same direction as the *up* vector if the angle is positive. For negative angles the *up* vector's direction will opposed to the up vector, effectively reversing the rotation.



The required steps after computing *objToCamProj* are:

- 1. normalize(objToCamProj)
- 2. angleCosine = innerProduct(lookAt,objToCamProj)
- 3. upAux = crossProduct(lookAt,objToCamProj)
- 4. glRotatef(acos(aux)*180/3.14,upAux[0], upAux[1], upAux[2]);



As in the previous billboards a function can be defined to setup the billboard. The code is as follows:

```
void billboardCylindricalBegin(
                      float camX, float camY, float camZ,
                      float objPosX, float objPosY, float objPosZ) {

        float lookAt[3],objToCamProj[3],upAux[3];
        float modelview[16],angleCosine;

        glPushMatrix();

// objToCamProj is the vector in world coordinates from the
// local origin to the camera projected in the XZ plane
        objToCamProj[0] = camX - objPosX ;
        objToCamProj[1] = 0;
        objToCamProj[2] = camZ - objPosZ ;

// This is the original lookAt vector for the object
// in world coordinates
        lookAt[0] = 0;
        lookAt[1] = 0;
        lookAt[2] = 1;
```

```
// normalize both vectors to get the cosine directly afterwards
        mathsNormalize(objToCamProj);

// easy fix to determine wether the angle is negative or positive
// for positive angles upAux will be a vector pointing in the
// positive y direction, otherwise upAux will point downwards
// effectively reversing the rotation.

        mathsCrossProduct(upAux,lookAt,objToCamProj);

// compute the angle
        angleCosine = mathsInnerProduct(lookAt,objToCamProj);

// perform the rotation. The if statement is used for stability
reasons
// if the lookAt and objToCamProj vectors are too close together then
// |angleCosine| could be bigger than 1 due to lack of precision
   if ((angleCosine < 0.99990) && (angleCosine > -0.9999))
      glRotatef(acos(angleCosine)*180/3.14,upAux[0], upAux[1],
upAux[2]);
}
```

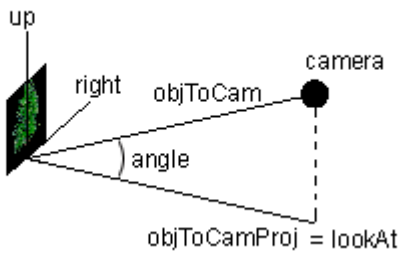
The function *billboardEnd()* should be called after rendering the object as shown in the following example.

```
billboardCylindricalBegin(cx,cy,cz,ox,oy,oz);
    drawObject();
billboardEnd();
```

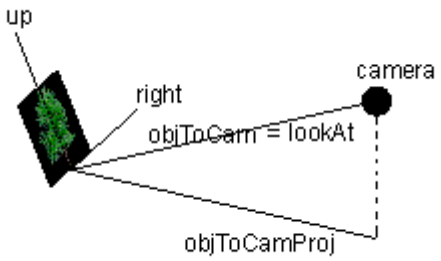
---

6. Spherical Billboards

The spherical version is a simple extension to the cylindrical case. After the object is rotated using the cylindrical approach, all which is left to do is to tilt the object until it truly faces the camera.



The axis of rotation is the right vector. As for the angle, a cosine can be obtained by the inner product between objToCamProj and objToCam.



As in the previous billboards a function can be defined to setup the billboard. The code is as follows:

```
void billboardSphericalBegin(
                      float camX, float camY, float camZ,
                      float objPosX, float objPosY, float objPosZ) {

        float lookAt[3],objToCamProj[3],upAux[3];
        float modelview[16],angleCosine;

        glPushMatrix();

// objToCamProj is the vector in world coordinates from the
// local origin to the camera projected in the XZ plane
        objToCamProj[0] = camX - objPosX ;
        objToCamProj[1] = 0;
        objToCamProj[2] = camZ - objPosZ ;

// This is the original lookAt vector for the object
// in world coordinates
        lookAt[0] = 0;
        lookAt[1] = 0;
        lookAt[2] = 1;


// normalize both vectors to get the cosine directly afterwards
        mathsNormalize(objToCamProj);

// easy fix to determine wether the angle is negative or positive
// for positive angles upAux will be a vector pointing in the
// positive y direction, otherwise upAux will point downwards
// effectively reversing the rotation.

        mathsCrossProduct(upAux,lookAt,objToCamProj);

// compute the angle
        angleCosine = mathsInnerProduct(lookAt,objToCamProj);

// perform the rotation. The if statement is used for stability
reasons
// if the lookAt and objToCamProj vectors are too close together then
// |angleCosine| could be bigger than 1 due to lack of precision
   if ((angleCosine < 0.99990) && (angleCosine > -0.9999))
      glRotatef(acos(angleCosine)*180/3.14,upAux[0], upAux[1],
upAux[2]);

// so far it is just like the cylindrical billboard. The code for the
```

```
// second rotation comes now
// The second part tilts the object so that it faces the camera

// objToCam is the vector in world coordinates from
// the local origin to the camera
        objToCam[0] = camX - objPosX;
        objToCam[1] = camY - objPosY;
        objToCam[2] = camZ - objPosZ;

// Normalize to get the cosine afterwards
        mathsNormalize(objToCam);

// Compute the angle between objToCamProj and objToCam,
//i.e. compute the required angle for the lookup vector

        angleCosine = mathsInnerProduct(objToCamProj,objToCam);


// Tilt the object. The test is done to prevent instability
// when objToCam and objToCamProj have a very small
// angle between them

        if ((angleCosine < 0.99990) && (angleCosine > -0.9999))
                if (objToCam[1] < 0)
                        glRotatef(acos(angleCosine)*180/3.14,1,0,0);
                else
                        glRotatef(acos(angleCosine)*180/3.14,-1,0,0);

}
```

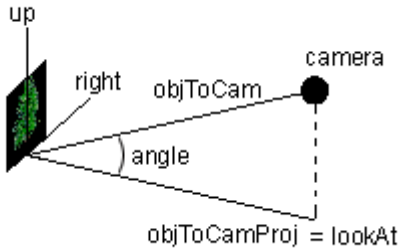
The function *billboardEnd()* should be called after rendering the object as shown in the following example.

```
billboardCylindricalBegin(cx,cy,cz,ox,oy,oz);
    drawObject();
billboardEnd();
```

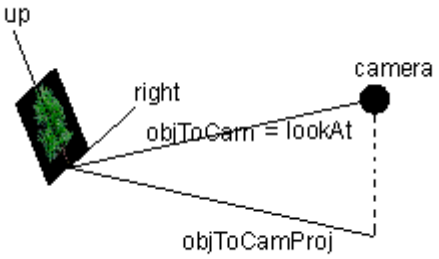
---

7. Spherical Billboards

The spherical version is a simple extension to the cylindrical case. After the object is rotated using the cylindrical approach, all which is left to do is to tilt the object until it truly faces the camera.



The axis of rotation is the right vector. As for the angle, a cosine can be obtained by the inner product between objToCamProj and objToCam.



As in the previous billboards a function can be defined to setup the billboard. The code is as follows:

```
void billboardSphericalBegin(
                      float camX, float camY, float camZ,
                      float objPosX, float objPosY, float objPosZ) {

        float lookAt[3],objToCamProj[3],upAux[3];
        float modelview[16],angleCosine;

        glPushMatrix();

// objToCamProj is the vector in world coordinates from the
// local origin to the camera projected in the XZ plane
        objToCamProj[0] = camX - objPosX ;
        objToCamProj[1] = 0;
        objToCamProj[2] = camZ - objPosZ ;

// This is the original lookAt vector for the object
// in world coordinates
        lookAt[0] = 0;
        lookAt[1] = 0;
        lookAt[2] = 1;


// normalize both vectors to get the cosine directly afterwards
        mathsNormalize(objToCamProj);

// easy fix to determine wether the angle is negative or positive
// for positive angles upAux will be a vector pointing in the
// positive y direction, otherwise upAux will point downwards
// effectively reversing the rotation.
```

```
        mathsCrossProduct(upAux,lookAt,objToCamProj);

// compute the angle
        angleCosine = mathsInnerProduct(lookAt,objToCamProj);

// perform the rotation. The if statement is used for stability
reasons
// if the lookAt and objToCamProj vectors are too close together then
// |angleCosine| could be bigger than 1 due to lack of precision
   if ((angleCosine < 0.99990) && (angleCosine > -0.9999))
      glRotatef(acos(angleCosine)*180/3.14,upAux[0], upAux[1],
upAux[2]);

// so far it is just like the cylindrical billboard. The code for the
// second rotation comes now
// The second part tilts the object so that it faces the camera

// objToCam is the vector in world coordinates from
// the local origin to the camera
        objToCam[0] = camX - objPosX;
        objToCam[1] = camY - objPosY;
        objToCam[2] = camZ - objPosZ;

// Normalize to get the cosine afterwards
        mathsNormalize(objToCam);

// Compute the angle between objToCamProj and objToCam,
//i.e. compute the required angle for the lookup vector

        angleCosine = mathsInnerProduct(objToCamProj,objToCam);


// Tilt the object. The test is done to prevent instability
// when objToCam and objToCamProj have a very small
// angle between them

        if ((angleCosine < 0.99990) && (angleCosine > -0.9999))
                if (objToCam[1] < 0)
                        glRotatef(acos(angleCosine)*180/3.14,1,0,0);
                else
                        glRotatef(acos(angleCosine)*180/3.14,-1,0,0);

}
```

The function *billboardEnd()* should be called after rendering the object as shown in the following example.

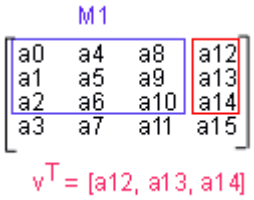
```
billboardCylindricalBegin(cx,cy,cz,ox,oy,oz);
    drawObject();
billboardEnd();
```

---

8. Where is the Object?

When an object is placed in the world using translations and rotations it becomes hard to find its world coordinates. In this section a simple technique to find the whereabouts in world coordinates of an object is presented. It is assumed that the object is rendered in the local origin. The technique tells you where the local origin is in world coordinates.

Again we have to look at the modelview matrix. The position of the local origin in camera's coordinates is know, it is the *v* vector of the figure bellow. But since the local origin's position is dependent on both the camera's position and orientation, one can't just add that position to the camera position.



What is needed is to reverse the orientation of the camera back to a coordinate system with axis aligned with the world coordinate system, and then we can add the camera position to the object's position relative to the camera. This sum will provide the position of the object in world coordinates.

In order to reverse the orientation of the camera we use the inverse of M1. As mentioned before, the inverse of M1 is the transpose.

The object's position in world coordinates is therefore given by the expression

```
objPosWC = camPos + M1T * V
```

where camPos is the camera position in world coordinates, and M1T is the transpose of M1.