Foundations of 3D Computer Graphics
Quaternions and Arcball

Assignment Objectives

In this project you will implement the code necessary to apply rigid body transformations, represented as a translation (coordinate vector) and a rotation (quaternion). You will also implement the Arcball interface for rotations. It is based off of the previous assignment and builds this functionality on top of it. You can use as base your solution for the previous project.

Task 1: Rigid Body Transformation

To start off the assignment, you need to copy the files in the starter zip (`arcball.h, quat.h, rigtform.h`) to your project directory for asst2. You will need these files to complete this assignment.

Your first task is to define the Rigid Body Transformation (`RigTForm`) structure, and implement the necessary manipulations of it. The rigid body transformation has two variables, a translation coordinate vector and a rotation quaternion. This structure represents the composition of the translation and the rotation. Pay attention to the order, translation on the left, rotation on the right: TR. You will need to implement inversion, multiplication, conversion to matrix form (that you can later feed to OpenGL), conversion from a translation vector, conversion from a quaternion, and multiplication with a vector. Except for conversion to matrix form, all remaining operations should be implemented without converting the quaternion to a matrix form. We will provide you with a Quat data type. The code below shows the overall structure of the RigTForm type. The code will be labeled with a TODO comment in the parts that you must implement.

Example prototypes:

```
class RigTForm
{
    Cvec3 t_;    // translation
    Quat r_;     // quaternion rotation

public:
    RigTForm();
    RigTForm(const Cvec3& t, const Quat& r);
    RigTForm(const Cvec3& t);
    RigTForm(const Quat& r);

    Cvec4 operator * (const Cvec4 &a) const;
    RigTForm operator * (const RigTForm& a) const;
};

RigTForm inv(const RigTForm& tform);
Matrix4 rigTFormToMatrix(const RigTForm& tform);
```

Now you need to alter your assignment 2 solution (or solution set of the course, available upon request) so that it represents the frames for the cubes and sky camera using Rigid Body Transformation structures, instead of using Matrix4s.

When this step is completed, you should not be using matrix-matrix multiplication, or matrix-coordinate vector multiplication for manipulating the sky camera and cube frames. Instead you will be using the equivalent rigid body transformation functions. During rendering you will convert rigid body transformation into a matrix, and then load these into the vertex shader appropriately as matrix data.

Task 2: Arcball

Your second task is to implement the arcball interface. The arcball interface will only come into play if:

1. You are manipulating the sky camera with respect to the world-sky coordinate system
2. You are manipulating a cube, and it is not with respect to itself.

In case 1, center the arcball about the world's origin. In case 2, center the arcball about the cube's center. You will probably notice that in these two cases, the arcball's center also happens to be the origin of the auxiliary frame that you have constructed in the previous assignment. In each case, you should also draw a sphere (wire-frame or translucent) to represent the arcball.

We provide some helper code. When manipulating an object using arcball, you know the center of the arcball in world coordinates. However, the user clicks somewhere on the screen, and you must use these screen coordinates (measured in pixel units) to figure out where exactly on the arcball the user has clicked.

To facilitate this, we provide two helper functions in `arcball.h`:

```
Cvec2 getScreenSpaceCoord(const Cvec3& p, const Matrix4& projection,
                          double frustNear, double frustFovY,
                          int screenWidth, int screenHeight);
double getScreenToEyeScale(double z, double frustFovY, int screenHeight);
```

`getScreenSpaceCoord` computes the screen-space coordinate (in terms of pixels) of a point. Its arguments are the eye-coordinates of the point, the projection matrix (produced by `makeProjectionMatrix()`), the vertical field-of-view (stored as `g_frustFovY`), the current screen width (stored as `g_windowWidth`), and the current screen height (stored as `g_windowHeight`). You can use this function to compute the screen-space coordinates of the arcball's center. Since the mouse cursor coordinates provided by GLUT are also in screen-space coordinates (though with a flipped Y), you can now work entirely in screen coordinates.

`getScreenToEyeScale` computes the following quantity: Suppose a line segment lies in a plane parallel to the eye-frame XY plane but with eye-frame Z coordinate defined by `z`, then the ratio between the length of the segment in eye-frame coordinates and the length of its projection on the screen (in terms of pixels) is a constant, and is calculated by `getScreenToEyeScale`. You need to pass in the eye-frame Z coordinate, the vertical field-of-view, and the current screen height. This function is useful for the following behavior:

We want the arcball to have a **`constant`** on-screen radius (except when translating; more on that in Task 3), controlled by a global variable `g_arcballScreenRadius`. For example, in the solution binary, we set the the arcball on-screen radius to be `0.25 * min(g_windowWidth, g_windowHeight)` inside the `reshape` callback. This means that the sphere that we are drawing has a dynamic radius in the object frame. The further away the sphere center is from the eye, the larger its radius should be. In fact its radius should exactly equal the desired on-screen radius scaled by the ratio returned by `getScreenToEyeScale`. We recommend you store the result of `getScreenToEyeScale` in a global variable called `g_arcballScale`. Since `g_arcballScale` will change every time the user changes the viewpoint, or the object he/she is manipulating, for simplicity, you might as well update it every frame at the start of `drawStuff()`.

For drawing a sphere, you can make use of the `makeSphere` function call defined in `geometrymaker.h`. Adapt the codes used for creating the cube geometry to create a sphere geometry with radius 1 and store it in a global variable. Then when you draw it, you can right multiply the model view matrix (MVM) by a uniform scale matrix to simulate a sphere with arbitrary radius. In this case, the radius of the sphere should be `g_arcballScale * g_arcballScreenRadius`. To draw the sphere in wire-frame, you can wrap it inside a pair of `glPolygonMode` call, e.g.,

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); // draw wireframe

// compute MVM, taking into account the dynamic radius.
// send in MVM and NMVM
// send in uColor
sphere->draw(curSS)

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); // draw filled again
```

Task 3: Translation Fix Up

Right now When the user translates things the amount of translation is calculated as the mouse's movement (measured in pixels) scaled by some hard-coded number (0.01). Now that you are maintaining `g_arcballScale`, you should change that hard-coded number to `g_arcballScale`, so that when you translate an object, the amount the object moves on screen will be equal to the amount your mouse has moved. When the arcball is not in use (e.g., in ego motion), `g_arcballScale` may not be correctly defined, so feel free to fall back to the hard-coded number in that case.

Moreover, if you run the solution binary, you will notice that as you translate in the z direction, the arcball appears as if it has a fixed size in object space (and hence a changing size in screen space). This provides a depth cue to the user. Moving the object away resulting in the arcball decreasing in size, and moving it closer resulting in the arcball increasing in size. When you release the mouse, the arcball snaps back to its constant screen space size. This nifty effect is accomplished in two simple steps which you should also implement:

- In `drawStuff`, add a conditional check such that you DO NOT update `g_arcballScale` when the user is translating in the Z direction, i.e., either the middle mouse button is clicked, or both the left and right mouse buttons are clicked. You can make use of the `g_mouse{L|R|M}ClickButton` global variables.
- At the end of the `mouse` call back, add a `glutPostRedisplay()` so that a mouse click/release always triggers a screen redraw. This is necessary for the snap-back-to-constant-size effect.