Foundations of 3D Computer Graphics
Key Frame Animator (Part I)

Assignment Objectives

In this project you will complete the code necessary for a keyframe animation system with linear interpolation. In such a system, the user defines the state of the world at *fixed key times*. *Intermediate frames* are generated via interpolation (for now just linear) to yield an animation.

Task 1: Keyframes

You begin with code from the previous assignment that encodes the robots and camera poses in a scene graph, and implements the arcball interface. Your first task is to add in a keyframe infrastructure.

A *frame* represents state of the scene at one instant in time. It stores one RBT for each `SgRbtNode` in the scene graph. Coding suggestions: to code a frame, we recommend simply using a `vector` of `RigTForms`.

In order to allow you to move data back and forth between a frame and the scene graph, you also will need to maintain a vector of node-pointers that point back into the corresponding `SgRbtNode` of the scene graph. So the ith entry in the vector is a `shared_ptr<SgRbtNode>` that points to the ith `SgRbtNode` in the scene graph. You can then use the `getRbt` and `setRbt` member functions of `SgRbtNode` to pull/push RBT values from/to the scene graph. In `sgutils.h`, we provide you with a utility function that will dump all of the `SgRbtNode`s from the scene graph and store pointers to them into a vector (that you pass in).

```
void dumpSgRbtNodes(shared_ptr<SgNode> root, vector<shared_ptr<SgRbtNode> >& rbtNodes);
```

The *script* for your animation will be stored as a list of frames, called key frames. This should be maintained as a linked list so that you can easily insert and delete frames in the middle. In C++, the STL `list` data structure may be used. For editing purposes, at any given time, your program will have a variable to represent which key frame is the *current frame*. When the program starts, your key frame list will be empty. So in that case the *current frame* is *undefined*.

Meanwhile at any given time you will be displaying the robot as stored in the scene graph. When the user manipulates the scene using the arcballs, the scene graph is updated. When the user presses certain keys, you either pull values from the scene graph, or push values to the scene graph.

You should implement the following hot keys:

- Space : Copy current key frame RBT data to the scene graph if the current key frame is defined (i.e., the list of frames is not empty).
- “u” : update: Copy the scene graph RBT data to the current key frame if the current key frame is defined (i.e., the list of frames is not empty). If the list of frames is empty, apply the action corresponding to hotkey “n”.
- “>” : advance to next key frame (if possible). Then copy current key frame data to the scene graph.
- “<” : retreat to previous key frame (if possible). Then copy current key frame data to scene graph.
- “d” : If the current key frame is defined, delete the current key frame and do the following:
  - If the list of frames is empty after the deletion, set the current key frame to undefined.

- Otherwise
  * If the deleted frame is not the first frame, set the current frame to the frame immediately before the deleted frame
  * Else set the current frame to the frame immediately after the deleted frame
  * Copy RBT data from the new current frame to the scene graph.

- "n" : If the current key frame is defined, create a new key frame immediately after current key frame. Otherwise just create a new key frame. Copy scene graph RBT data to the new key frame. Set the current key frame to the newly created key frame.

- "i" : input key frames from input file. (You are free to choose your own file format.) Set current key frame to the first frame. Copy this frame to the scene graph.

- "w" : output key frames to output file. Make sure file format is consistent with input format.

Task 2: Linear interpolation

During the animation, you will need to know how to create an *interpolated frame* that is some linear blend of two frames. Thus you will need to implement linear interpolation that acts on two RBTs. Let us call the interpolating factor α, where $\alpha \in 0..1$.

For the translational component of the RBTs, this will be done with linear interpolation acting component-wise on the entries of the Cvec3s, as in

$$\mathrm{lerp}(\mathbf{c_0}, \mathbf{c_1}, \alpha) := (1-\alpha)\mathbf{c_0} + \alpha\mathbf{c_1}$$

For the rotational component, this will be done using spherical linear interpolation as discussed in class/book.

Let us call our quaternions $\mathbf{q_0}$ and $\mathbf{q_1}$. Then

$$\mathrm{slerp}(\mathbf{q_0}, \mathbf{q_1}, \alpha) := (\mathrm{cn}(\mathbf{q_1}\mathbf{q_0}^{-1}))^{\alpha}\mathbf{q_0}$$

"cn" is the conditional negate operation that negates all four entries of a quaternion if that is needed to make the first entry non-negative. The quaternion power operator needs to be implemented by you. When implementing the power operator, the C/C++ function "atan2" (in `<cmath>`) will be useful: `atan(a,b)` returns a unique $\phi \in [-\pi..\pi]$ such that $\sin(\phi) =$ `a` and $\cos(\phi) =$ `b` .

Task 3: Playing the animation

The animation will be viewed from whatever viewpoint is current. This can be changed using the 'v' key from asst2/3/4.

You can think of the sequence of keyframes as being numbered from -1 to n. You will only show the animation between frames 0 and $n-1$. (This will be useful when doing Catmull-Rom interpolation in the next assignment.) Because the animation only runs between keyframes 0 and $n-1$, you will need at least 4 keyframes (Key frame -1, 0, 1, and 2) in order to display an animation. If the user tries to play the animation and there are less than 4 keyframes you can ignore the command and print a warning to the console. After playing the animation, you should make the current state be keyframe $n-1$, and display this frame. (Notation clarification: In the text and notes we numbered the keyframes $-1, 0, \ldots, n+1$, and the valid range for the time parameter was only $[0, n]$.)

For any real value of t between 0 and $n-1$, you can find the "surrounding" key frames as `floor(t)` and `floor(t)+1`. You can compute α as `t-floor(t)`. You can then interpolate between the two key frames to get the intermediate frame for time t, and copy its data to the scene graph for immediate display. A complete animation is played by simply running over a suitably sequence of t values.

Hot Keys:

- "y" : Play/Stop the animation

- "+" : Make the animation go faster, this is accomplished by having one fewer interpolated frame between each pair of keyframes.

- "−" : Make the animation go slower, this is accomplished by having one more interpolated frame between each pair of keyframes..

For playing back the animation, you can use the `GLUT` timer function `glutTimerFunc(int ms, timerCallback, int value)`. This asks `GLUT` to invoke `timerCallback` with argument `value` after `ms` milliseconds have passed. Inside `timerCallback` you can call `glutTimerFunc` again to schedule another invocation of the `timerCallback`.

A possible way of implementing the animation playback is listed below. Calling `animateTimerCallback(0)` triggers the playing of the animation sequence.

```
static int g_msBetweenKeyFrames = 2000; // 2 seconds between keyframes
static int g_animateFramesPerSecond = 60; // frames to render per second during animation playback

// Given t in the range [0, n], perform interpolation and draw the scene
// for the particular t. Returns true if we are at the end of the animation
// sequence, or false otherwise.
bool interpolateAndDisplay(float t) {...}

// Interpret "ms" as milliseconds into the animation
static void animateTimerCallback(int ms) {
    float t = (float)ms/(float)g_msBetweenKeyFrames;

    bool endReached = interpolateAndDisplay(t);
    if (!endReached)
        glutTimerFunc(1000/g_animateFramesPerSecond,
                      animateTimerCallback,
                      ms + 1000/g_animateFramesPerSecon);
    else { ... }
}
```

Appendix: C++ Standard Template Library `list` Usage

For this assignment, you need to implement some kind of linked list data structure to store the list of key frames. While you're free to roll your own, using the C++ STL `list` will save your time (in the long run at least).

Like the `vector`, `list` is a templated container class. To declare a double linked list containing objects of type `Thing`, you would write

```
list<Thing> things;
```

The important feature of the double linked list data structure is that you can insert and remove items from the middle of the list in constant time (unlike `vector`, for which inserting and removing takes linear time). To tell the linked list where in the sequence you want to add or remove an entry, you need to use an `iterator`. An iterator of a `list<Thing>` has the type `list<Thing>::iterator`. You can think of an iterator as a pointer pointing to one element in the list. The iterator pointing to the first element list is returned by the `begin()` member function of the list, so the following code

```
list<Thing>::iterator iter = things.begin();
```

will intialize an iterator called `iter` to point to the first item in `things`. You can point to the next item in the list by incrementing the iterator, so after

```
++iter;
```

`iter` now points to the second element in the list. Likewise you can decrement an iterator by doing `--iter;`

You can dereference an iterator by writing `*iter` or `iter->member_of_Thing`, as you would do for an actual pointer pointing to an object of type `Thing`. This gives you access to the item that the iterator points to.

Now the list is of limited size, so you need a way to test whether you have reached the end of the list. When `iter` has reached the end of the list, it will take on a special iterator value returned by `things.end()`. Importantly, `things.end()` is not the iterator that points to the last element in the list. Conceptually `thing.end()` points to one element beyond the last element, so it is undefined behavior to dereference `things.end()`.

The following snippets of code will print out all entries in, say, a `list` of `int`

```
// things has the type list<int>
for (list<int>::iterator iter = things.begin(), end = things.end(); iter != end; ++iter) {
  cout << (*iter) << endl;
}
```

If the list is empty to start with, its `begin()` will return the same value as its `end()`.
Quick quiz: How do you access the last entry of a list (assuming its not empty) using iterators? `*things.end()` won't work. The following will work.

```
list<int>::iterator iter = things.end();
--iter;
// now iter points to the last element.
```

Now suppose you have a valid iterator pointing to some entry in the list. Calling the `erase` member function of `list` allows you to erase the element from the list, e.g.,

```
things.erase(iter);
```

Note after you have erased the element in the list pointed to by `iter`, the iterator `iter` itself becomes undefined, and doing things `++iter, --iter, *iter` will result in undefined behavior.

So suppose you want to erase the element pointed to by `iter`, and then set `iter` to the element immediately following what you have deleted, what would you do? Simply save `iter` by making a copy, increment the copy, call erase, and set `iter` to the incremented copy, like below

```
list<int>::iterator iter2 = iter;
++iter2;
things.erase(iter);
iter = iter2;
```

Note that in the spec, there is a current frame concept. We recommend that you implement the current frame as an iterator. Recall that when the list of frames is empty, the current frame is undefined. You can use the `end()` iterator to represent this "undefined value".

Now suppose you want to insert an element to the list. You use the `insert` member function of `list`. The following

```
// Assume things is of type list<int>
things.insert(iter, 10);
```

will insert 10 right **before** the element pointed to by `iter`.
Quick quiz: How do you append entries to the end of the list using `insert`? You do `things.insert(things.end(), 10);` .

There are a bunch of other functions that you might find helpful, like `back()`, `push_back`, and so on.

By the way, one key advantage of STL and using the iterator concept is that you can write code that works on any kind of container (array, double linked list, map, etc) as long as they expose the same iterator interfaces (which they do). For example, if you replace every occurrence of `list` in this section with `vector`, the snippets would still work, except that `insert` and `erase` function of `vector` will take linear time as opposed to constant time.

Finally here are a few good intro and references site. I **highly** recommend that you at least skim the first two. Use the last one for references.

- Introduction to the Standard Template Library: `http://www.sgi.com/tech/stl/stl_introduction.html`
- An Introduction to the Standard Template Library (STL): `http://www.mochima.com/tutorials/STL.html`
- STL Containers reference: `http://www.cplusplus.com/reference/stl/`