CS 4621 Practicum Programming Assignment 3 Animation

out: Tuesday 26 November 2013
due: : Friday 6 December 2013

1 Introduction

In this assignment, you will explore a common topic in animation: key frame animation. For the first (and only) problem, you will add a keyframe animation system to the modeler from Manipulators and create a short animation.

As with previous assignments, you will have to copy over some functionality that you implemented in previous assignments. If you have any lingering bugs in these components, please visit office hours so that one of the TAs can help you out.

2 Problem 1: Keyframe Animation

In the first part of the assignment, you will implement the main features of a keyframe animation framework. We have added a framework that stores keyframes and provides an interface for editing them; you will build on that framework and implement a few main features that enable correct interpolation of animations between keyframes.

2.1 The Interface

We have added a new "Animation" panel, which contains several controls for creating and editing an animation:

- Moving to a frame: The slider and text box on the left of the screen allow you to move between the frames of the animation.
- Adding a frame: To add a keyframe at the current frame, click the "Keyframe it" button. The frame will appear in the keyframe list on the left. The new keyframe will be initialized to the current state of the scene when you clicked the button.
- Editing a keyframe: To edit a keyframe, click on its entry in the list and then edit the transformations in the scene.

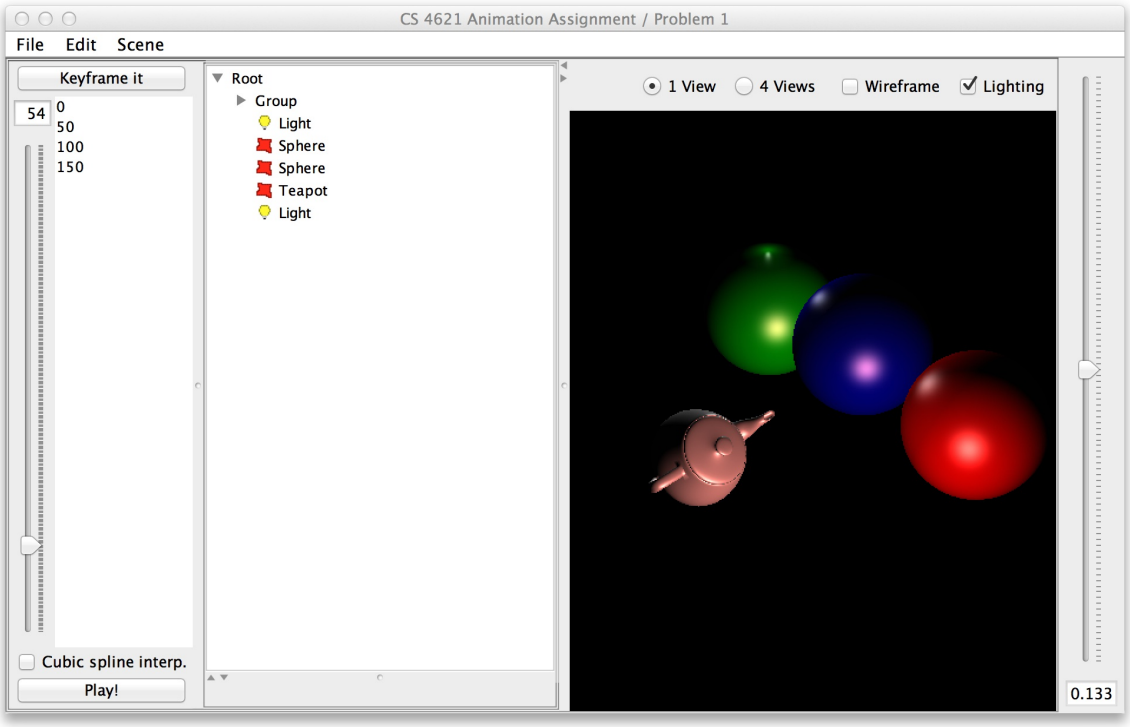

Figure 1: The keyframe animation interface.

- Removing a keyframe: Right click on the keyframe in the list and choose "Remove Keyframe" from the popup menu. Frame 0 cannot be removed.
- Playing the animation: Click the "Play!" / "Stop!" button to start and stop the animation.
- Selecting interpolation method: Check "Cubic spline interpolation" to interpolate using splines.
- Saving/loading animations: Use the *File* → *Save As* and *File* → *Open* dialogs. We have provided some sample animation files for reference, located in `data/scenes/animation`.

2.2 Keyframes

For this problem, we have extended the scene graph system to store information about multiple keyframes. The `SceneKeyframeable` class stores a keyframed scene graph. The scene contains special scene nodes (`SceneNodeKeyframeable`, `MeshNodeKeyframeable`, and `LightNodeKeyframeable`) that support keyframing. These nodes store two pieces of information:

- The current state of the node. Each of the "keyframeable" node classes extends the non-keyframed version. The member variables of the base class (e.g., the `rotation`, `scaling`, and `translation` members) store the interpolated state of this node at the current frame of the animation.

- The node's state at each keyframe. Each node has a map called `keyframes`, which maps keyframe numbers to snapshots of the node's state at those keyframes. You will use this information when you implement linear and cubic-spline keyframe interpolation.

We already implement the functionality of adding and deleting keyframes from the scene. You will have to copy your implementations of manipulators from the first practicum assignment, but since the keyframeable scene nodes extend `SceneNode`, `MeshNode`, and `LightNode`, you will not need to change your code for it to work.

2.3 Functionality to Implement

You will implement all the functionality needed to interpolate between the keyframes of the animation. For translation, scaling, and light source parameters (intensity), you will implement linear and cubic spline interpolation. For rotations, we only ask you to implement spherical linear interpolation as described in the lecture.

The scene is interpolated using two methods:

- `void linearInterpolateTo(int frame)` and
- `void catmullRomInterpolateTo(int frame)`.

Each of the keyframeable scene node classes contains these two methods. When the user selects a new frame in the interface, the scene graph calls the appropriate method for each node in the graph, interpolating the entire scene to the selected frame. You will implement these methods for the three types of keyframed scene node.

2.3.1 Interpolation Overview

For both types of interpolation, the basic outline will look like this:

1. If the desired frame is a keyframe, set the state of the node to the state at that keyframe.
2. Otherwise, find the 2 (linear) or 4 (Catmull-Rom) keyframes to interpolate, and the time t to use for the interpolation.
3. Interpolate the states of the keyframes, and assign the resulting state to the node.

Specific steps are outlined in more detail below.

2.3.2 Catmull-Rom Spline Interpolation

The Catmull-Rom spline is a cubic interpolating spline that passes through all its control points. For the interval of the curve specified by control points $P_{i-2}, P_{i-1}, P_i, P_{i+1}$, the spline has the formula

$$P(t) = \frac{1}{2}\begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix}\begin{bmatrix}0 & 2 & 0 & 0\\ -1 & 0 & 1 & 0\\ 2 & -5 & 4 & -1\\ -1 & 3 & -3 & 1\end{bmatrix}\begin{bmatrix}P_{i-2}\\ P_{i-1}\\ P_i\\ P_{i+1}\end{bmatrix}$$

Note that each patch interpolates its middle two control points. That is,

$$P(0) = P_{i-1}, \quad P(1) = P_i.$$

Additionally, the tangents at these points are determined by the remaining two control points as follows:

$$v_{i-1} = \frac{1}{2}(P_i - P_{i-2}) \quad v_i = \frac{1}{2}(P_{i+1} - P_{i-1}).$$

2.3.3 Finding Control Points

To interpolate the state at some frame f using Catmull-Rom spline interpolation, we need to find four frames to define our control points. Call these frames f_0, f_1, f_2, and f_3. The frames are chosen as follows:

- f_1 is the closest keyframe before frame f.
- f_0 is the closest keyframe before frame f_1.
- f_2 is the closest keyframe after frame f.
- f_3 is the closest keyframe after frame f_2.

That is, we find the four subsequent keyframes in our animation that satisfy $f_0 < f_1 < f < f_2 < f_3$. The time t for the interpolation is determined by how far along frame f lies between f_1 and f_2.

We must also consider a few corner cases. When f is later than the final keyframe of the animation, we can simply copy the state of the final keyframe. Otherwise, if fewer than four control points can be found, you can duplicate keyframes where necessary (e.g., set $f_0 = f_1$ or $f_3 = f_2$ if there is only one keyframe before or after f). Done correctly, your spline should successfully pass through every keyframe in the animation.

For linear interpolation, the situation is the same, except that you only need to find the keyframes f_1 and f_2 above.

2.3.4 Converting between Euler Angles and Quaternions

We ask you to interpolate rotations using spherical linear interpolation (`slerp`), which operates on quaternions, but the rotations of the scene nodes are specified using Euler angles. To get around this, you will implement conversions between these two representations.

From Euler Angles to Quaternions The rotation specified by Euler angle (a, b, c) is the composition of three axis-angle rotations: a rotation of angle a about $(1, 0, 0)$, of angle b about $(0, 1, 0)$, and of angle c about $(0, 0, 1)$.

To represent this with quaternions, first write the three axis-angle rotations as quaternions $q = \langle s, v \rangle$:

$$q_x = \langle \cos(a/2),\ \sin(a/2) \cdot (1, 0, 0) \rangle$$
$$q_y = \langle \cos(b/2),\ \sin(b/2) \cdot (0, 1, 0) \rangle$$
$$q_z = \langle \cos(c/2),\ \sin(c/2) \cdot (0, 0, 1) \rangle$$

Finally, the total rotation described by the Euler angles is $q = q_z q_y q_x$.

From Quaternions to Euler Angles To convert from quaternion $q = \langle w, x, y, z \rangle$ to Euler angle (a, b, c), use the following equation:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \operatorname{atan2}(2(wx + yz), 1 - 2(x^2 + y^2)) \\ \operatorname{asin}(2(wy - zx)) \\ \operatorname{atan2}(2(wz + xy), 1 - 2(y^2 + z^2)) \end{bmatrix}$$

A little care must be taken in one special case. If the absolute value of the argument to the asin above comes too close to 1, this formula can become unstable. (Interestingly, this is a direct consequence of the "gimbal lock" problem we discussed earlier in the semester.) The following pseudocode outlines how to solve this problem:

```
test ← 2(wy − zx)
if abs(test) > 0.9999 then
    if test > 0 then
        a ← 2 atan2(x, w)
        b ← π/2
        c ← 0
    else
        a ← −2 atan2(x, w)
        b ← −π/2
        c ← 0
    end if
else
    use the standard conversion above
end if
```

Note: This section assumes that the Euler angles are specified in radians, but the scene nodes store their rotation angles in degrees (to match the convention used by OpenGL). You must handle the conversion between the two when implementing the formulas above.

2.4 Creating an Animation

With the above components in place, you will be able to specify keyframes, interpolate between them, and produce animations. To demonstrate your animation system, create and submit with your solution an animation that includes a link and joint structure. Your structure should have several joints, and you should use the structure's degrees of freedom in an interesting way, but otherwise feel free to animate whatever you want (e.g., an arm waving, a plant swaying in the wind, a robot flailing around, etc.).

3 Implementation Notes

As with previous assignments, places where you will add code are marked with `TODO`; note that some of these are marked as optional. Below are some suggestions:

3.1 Keyframe Animation

- In class, we have used the order $q = \langle w, x, y, z \rangle$ for the four components of a quaternion, but the `Quat4f` class provided by `javax.vecmath` uses the order `(x, y, z, w)` for its constructor and `set` method. You have been warned.
- We have provided the `KeyframeAnimation` class as a place to hold tools (e.g., conversion between quaternions and Euler angles) that will be useful in multiple places in your solution. You do not need to implement or use the exact function stubs we provide, but they might help guide your thinking.
- When implementing the interpolation methods in the `*NodeKeyframeable` classes, you may find the Javadoc for TreeMap (`http://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html`) useful.
- The interface is capable of using manipulators. We do not require you to copy your implementations over, but you may find it useful to do so for editing your animations.

4 What to Submit

Submit a zip file containing your solution organized the same way as the code on CMS. Your animation file(s) should be in the `data/scenes` directory. Include a readme in your zip that contains you and your partner's names and NetIDs, any problems with your solution, and anything else you want us to know.