BLG 444E: Assignment 4
Robots and Key Frame Animator

Due: 5 May 2014, Monday, 23:00

Assignment Objectives

In this project you will complete the code necessary for drawing robots and 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.

Notation

For exposition here is some notation we will use:

$$o^t = w^t O$$
$$s^t = o^t S$$
$$l^t = s^t L$$

'w' is the world frame, 'o' is an object frame, 's' is a shoulder frame, 'l' is an elbow frame. All the matrices are orthonormal.

The accumulated operator C will be used to relate a frame to the world, as in C(l) = OSL

Task 1: Drawing the Robots Using A Scene Graph (25%)

We will represent the scene structure (and within it, the two robots) hierarchically, and for this we will use a scene graph structure. The scene graph is a tree structure where the parent-child relationship will capture the relationship between objects and sub-objects in a hierarchy. When we manipulate an object, all of the sub-objects transform along for the ride. (In more generality, the graph can be any directed acyclic graph, which allows us to draw multiple instanced copies of parts of the scene.)

There are two kinds of nodes in our scene graph:

Transform nodes : A transform node represents a child's frame with respect to its parent frame through a rigid body transform (RBT). A transform node will store that RBT. For example, it may represent the transform relating the elbow frame l^t to the shoulder frame s^t, as in $l^t = s^t L$. A transform node can have any number of children.

One special transform node is the root node, which is simply the root of our entire scene graph. We will call it `g_world` in the codes. It correspond to the world frame w^t.

Recall that in Assignment 3, the scene we rendered contains quite a few `RigTForms`: `g_skyRbt` for the "sky camera", `g_objectRbt[]` for the two cubes. Now we can encode all of the above as transform nodes. Let us call them `g_skyNode`, `g_robot1Node`, and `g_robot2Node`. They will be children of the

root node g_world. Moreover, since each robot will consist of a tree of articulated joints, each of g_robot1Node and g_robot2Node will in fact have child transform nodes of its own that correspond to the left shoulder frame, right shoulder frame, and so on.

Shape nodes : A shape node represents actual things that get drawn (such as a cube, a sphere, or some other piece of geometry). A shape node can not have any children. The shape to be drawn will be situated somehow with respect to its parent frame (which is a transform node). Since this relationship is not necessarily rigid-body, a shape node stores a Matrix4.

To see why this affine transform is necessary, suppose the lower arm is represented by a cube. The actually coordinates of vertices in the cube are encapsulated by the Geometry object that you have encountered since Assignment 2, which stores the vertices and indices that form the cube as OpenGL VBOs and IBOs. Our shape node will store a shared_ptr pointing to the cube geometry. However the coordinates in the cube assume that the cube center is at $(0, 0, 0)$ and sides of the cubes are of length 1. Thus to draw this as a child of the elbow, we need a frame $\mathcal{B}^t$ that is translated to the lower arm's center and then scaled (to elongate in the direction of the arm), as in $\mathcal{B}^t = \mathcal{T}^t B = \mathcal{T}^t \cdot \text{Trans} \cdot \text{Scale}$. The transform B hence needs to be stored in the shape node.

A big advantage of having the scene graph is that you can then write a generic routine to render it, or perform other operations, as opposed to having to hard code a sequence of

```
set body's MVM matrix;
draw body's geometry;
set shoulder's MVM matrix;
draw shoulder's geometry;
...
```

Code migration

Now we will dive in to the code and help your migrate your previous rendering code in Assignment 3 to using scene graph for drawing. There are quite a few new files in the starter archive. To start off, copy all of them to you project folder. Note that the new Makefile assumes that the main program file is named asst4.cpp, so you might want to rename your old asst3.cpp. If you're are using Visual Studio, you need to add the new files (BUT DO NOT ADD asst4-snippets.cpp) into your project by choose from menu Project | Add Existing Items.

The file asst4-snippets.cpp contains instructions and snippets of code for modifying your asst4.cpp. The snippets are ordered roughly according to the order they will appear in asst4.cpp.

Please read through the remainder of this TASK first, and then follow the TASK 1 portion of asst4-snippets.cpp to migrate your code to drawing using a scene graph.

Scene Graph Codes

Of the new files, scenegraph.{cpp|h} contain the implementation of the scene graph data structure that we talked about. It defines a few types.

In the following description, an abstract base class refers to a class with unimplemented virtual functions. So they are kind of like Interfaces in Java, although they can contain concrete member variables and function implementations unlike Java Interfaces. An abstract base type cannot be instantiated since it has unimplemented virtual methods. In contrast a concrete class has all its virtual methods implemented, and hence can be instantiated.

SgNode : Abstract base class of all scene graph nodes. It defines a comparison operators (==, !=) testing whether two scene graph nodes are the same node, which might come handy at times. It declares a virtual function accept(...) which needs to be implemented by all derived types to support the so called Visitor pattern, which we will talk about in the next section.

SgTransformNode : Abstract base class of all transform nodes. Derives from SgNode. Recall from the previous section that transform node encodes an RBT that transforms from a parent frame to the current frame. Thus SgTransformNode declares a virtual function getRbt() returning this RBT. You can imagine many different concrete classes deriving from SgTransformNode and implementing getRbt() in different ways:

- One type of transform node could allow only rotation along a single axis;
- One type of transform node could allow only translation along a single axis;
- One type of transform node could allow full rotation, modeling a ball joint.
- In this assignment, we will have transform nodes that allow full RBTs by wrapping around a RigTForm object.

Also since SgTransformNode can have children, it implements a bunch of function calls like addChild, removeChild, getNumChildren, getChild(i).

SgShapeNode : Abstract base class of all shape nodes. A shape nodes knows how to draw itself, and needs to store a (not-necessarily rigid body) affine transform. Hence it has two virtual methods draw and getAffineMatrix. draw will take in the current ShaderState as argument, and draw the node itself. It does not set model view matrix though since it does not have that information locally.

SgRootNode : Concrete class deriving from SgTransformNode. This is a transform node corresponding to the root of the scene tree. Its getRbt() always returns an identity transform.

SgRbtNode : Another concrete class deriving from SgTransformNode. This is a transform node that wraps a RigTForm and allows full freedom of rotation/translation.

SgGeometryShapeNode : A templated concrete class deriving from SgShapeNode, parameterized by a user specified type Geometry. It stores a shared_ptr to a Geometry object (which stores OpenGL VBO/IBO), and a color attribute. Its draw() implementation simply sets the uColor uniform variable and delegates to Geometry's draw() function.

We can use the following to instantiate the template with our own Geometry class.

```
typedef SgGeometryShapeNode<Geometry> MyShapeNode;
```

MyShapeNode is then available to use as a concrete implementation of SgShapeNode.

Construct the Scene Graph

At this point, we are ready to construct the scene graph. We almost always use shared_ptrs to store pointers to scene graph nodes, since it facilitates automatic memory management via reference counting.

asst4-snippets.cpp instructs you to do the following:

1. To start, declare g_world, g_skyNode, g_groundNode, g_robot1Node, g_robot2Node, and g_currentPickedRbtNode which will point to suitable nodes in the scene graph.

2. Insert constructRobot and initScene() after the initGeometry() function, and call initScene() after initGeometry() in your main() function. This initializes the scene graph with g_world as the root, with g_skyNode, g_groundNode, g_robot1Node, and g_robot2Node as its children. Moreover, g_groundNode will have a SgGeomeryShapeNode as its child referring to the ground geometry, and g_robot{1|2}Node will have more children nodes that model the torso, upper right arm, and lower right arm of the robot.

3. Note that constructRobot assumes a cube Geometry of side length 1 is stored as g_cube. If the cube Geometry is called cube in your code, you should replace the occurrence in constructRobot

Draw the Scene Graph

Again refer to asst4-snippet.cpp for the following:

1. Modify the drawStuff to take in const ShaderState& curSS and bool picking as arguments. picking will always be false for now. This is to make your later job of implementing picking easier. Inside drawStuff, remove the line

   ```
   const ShaderState& curSS = *g_shaderStates[g_activeShader];
   ```

   so that whenever curSS is referred to inside drawStuff, the passed in argument is used.

2. Replace the code for drawing the ground and the two cubes with the following, as in asst4-snippet.cpp:

   ```
   Drawer drawer(invEyeRbt, curSS);
   g_world->accept(drawer);
   ```

   Note that you still need to get invEyeRbt from the current eye, and the current eye is not dependent on the scene graph yet. We will fix that later. Likewise, you still need to pass the light-related uniform variables yourself.

3. In display(), replace the drawStuff call with the following version:

   ```
   drawStuff(*g_shaderStates[g_activeShader], false);
   ```

Now that you have read the detailed manual of asst4-snippets.cpp, go ahead and following the instructions in it to perform the migration.

Try to build and run the program once you are done. Two partial robots should now be drawn. You cannot manipulate them yet.

Build the robot

Now that everything is working, you should build a complete robot in constructRobot(). Understand the codes there, and add more joints and shapes to model a complete robot with at least the following parts: head, left/right upper arm, left/right lower arm, left/right upper leg, left/right lower leg.

Visitor Pattern

Notice how easy it is to draw the scene graph using a single Drawer object. If you look at drawer.h you will see that it is derived from the SgNodeVisitor class implemented. The SgNodeVisitor base class is part of the so called visitor pattern that our scene graph implements.

Often we need to operate on the scene graph to perform certain operations. One could write a recursive function that perform depth-first traversal. However it quickly becomes troublesome and error prone to have to write recursive traversal code every time you need to operate on the scene.

In order to separate between the generic graph traversal code, and "what gets done" once we get to each node, we use something called a visitor class. You implement a visitor class by deriving from the SgNodeVisitor class, defined below, and overriding any of its functions to provide custom functionality:

```
class SgNodeVisitor {
public:
  virtual bool visit(SgTransformNode& node);
  virtual bool visit(SgShapeNode& node);

  virtual bool postVisit(SgTransformNode& node);
  virtual bool postVisit(SgShapeNode& node);
};
```

The scene graph can then call your provided code as it traverses the tree structure, as described below.

This is a bit tricky, so read this slowly. During the graph traversal, a visitor object is passed along from node to node. The visitor is passed to some node by calling that node's `accept` member. On the receiving end of an accept call, the node first calls the visitor's `visit` member function, and passes itself as the argument. The visitor can then do "its job" (whatever that may be). The visitor keeps track of any state that it may need to do its job. Once the visit returns, the node can then passes the visitor on to its children, using a recursive call to `accept`. After all the children calls return, the node calls `postVisit` of the visitor, passing itself as the argument, to deal with any state cleaning that is needed.

Both `visit` or `postVisit` return a `bool` value. If at any point, `false` is returned, the graph traversal terminates immediately. Note that there are two `visit` functions, taking in either a `SgTransformNode` or a `SgShapeNode`. This allows you to handle different type of nodes differently. The compiler determines which `visit` to use based on the node's type.

In the code, we already provide you a "drawing" visitor `Drawer` that will draw the entire scene. This visitor maintains a `RigTForm` stack. When the visitor acts on a transform node, it pushes a new RBT on the stack. The pushed RBT is the product of two RBTs, reading left to right: the previously topmost RBT from the stack and the RBT stored in this transform node. A drawing visitor acting on a shape node will use this stack as well as the non rigid transform stored in the shape node to pass the appropriate MVM matrix to the active shaders. The visitor then calls the node's own `draw` function, which will lead to the OpenGL draw calls being made. The `postVisit` function of the drawing visitor will pop the top RBT off the stack.

We use the standard `vector` to implement a stack. Methods of interest are `push back`, `back`, `empty` and `size`, which you can learn more about at `http://www.cplusplus.com/reference/stl/vector/` if their meanings are not obvious.

Please read the code in `drawer.h` and make sure you understand it.

Task 2: Keyframes (30%)

Continue from your code that encodes the robots and camera poses in a scene graph. In this task, you will 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 `SgRbtNodes` 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 3: Linear interpolation (25%)

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

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

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

Let us call our quaternions q_0 and q_1. Then

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

"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 $\varphi \in [-\pi..\pi]$ such that $\sin(\varphi) = a$ and $\cos(\varphi) = b$.

Task 4: Playing the animation (20%)

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

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, ..., $n + 1$, 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_animateFramesPerSeco
                      nd, 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 immedi- ately 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/`