Foundations of 3D Computer Graphics
Robots and Part Picking

Assignment Objectives

This assignment will build on the previous assignment and extend it by implementing a system for drawing articulated bodies, as well as the ability to select objects on the screen.

Your program will draw two robots, instead of two cubes as done in the previous assignments, and allow the manipulation of robot parts in a way the preserves the hierarchical structure of the robot. You will allow the user to rotate and translate the robots as well each of their movable joints.

Notation

For exposition here is some notation we will use:

$$\begin{aligned} \vec{\mathbf{o}}^t &= \vec{\mathbf{w}}^t O \\ \vec{\mathbf{s}}^t &= \vec{\mathbf{o}}^t S \\ \vec{\mathbf{l}}^t &= \vec{\mathbf{s}}^t L \end{aligned}$$

'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

We will represent the scene structure (and within it, the two robots) hierarchically, and for this we will use a *s*cene 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 $\vec{\mathbf{l}}^t$ to the shoulder frame $\vec{\mathbf{s}}^t$, as in $\vec{\mathbf{l}}^t = \vec{\mathbf{s}}^t L$. 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 $\vec{\mathbf{w}}^t$.

Recall that in the quaterion project, the scene we rendered contains quite a few `RigTForm`s: `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 the HW3D project, 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 $\vec{\mathbf{b}}^t$ that is translated to the lower arm's center and then scaled (to elongate in the direction of the arm), as in $\vec{\mathbf{b}}^t = \vec{\mathbf{l}}^t B = \vec{\mathbf{l}}^t \cdot \text{Trans} \cdot \text{Scale}$. 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 of the quaternions project 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. There are two new shaders: `pick-gl{2|3}.fshader` as well, so copy them to your `shaders` directory. For convenience you can also add them to your Visual Studio project.

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_ptr`s 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.

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: Part Picking

We now turn to allowing the user to manipulate the robot parts. The user should be allowed to press “p” and then use the mouse to left click on a desired robot part. The determination of which part the user clicked on is called “picking.” After picking a part, the user should then be allowed to use the mouse to rotate the part about its joint (its parent transform node in the scene graph). We say that the body part has been picked, and its associate joint has been **activated**. When the trunk is picked, its parent in the scene graph is the “entire robot” node (one of g_robot1Node and g_robot2Node), which is then activated so that the entire robot can be moved around in space.

Picking will be specified as follows: When “p” is pressed, the next left mouse click will trigger the following: The current scene is rendered, but this time instead of drawing the usual colors into the (back) framebuffer, you will assign each part to be drawn a unique ID that is drawn into the framebuffer. Drawing this ID tagged scene will involve using a different fragment shader that uses the id, passed in as a uniform variable, as the solid color for the object. After rendering the scene, you will call glFlush() but not glutSwapBuffer(). Then you read back a pixel from the (back) framebuffer and thereby determine which part was picked by the mouse.

- If the mouse click was not over any robot, then the eye’s (sky-camera, or robot, depending on the ’v’ key setting) transform node is activated. (so it can be manipulated as is done in the quaternion assignment.)
- If the mouse click was over a robot’s part, then the chosen robot part is picked, and its parent node is activated.

Once a node is activated you use the mouse motion to to update the node’s transformation. (At this point, for debugging purposes, just print out the node information).

We have provided the skeleton of a picking visitor in pick.h and pick.cpp. In pick.h, you can see that the picker contains a stack of scene graph nodes nodeStack_, and a map from integer id to SgRbtNode called idToRbtNode_, an id counter idCounter_, and a Drawer visitor drawer_. The idea is that you can use the stack to maintain the nodes from the root node to the current visited node. Then when you have encountered a shape node, you increase the id counter, find the SgRbtNode that is closest from the top of the stack (which should be the shape node’s parent), and add the association between the id counter and the SgRbtNode to the map.

Before drawing the shape using drawer_, you should convert the ID to RGB color, set it to the uniform variable uIdColor through the handle h_uIdColor in ShaderState. You can query the drawer_ for the current ShaderState by calling its getCurSS() method.

Then you just delegate the work of drawing to the drawer_. Just call its respective methods as in picker.cpp.

You need to fill out the body of all TODO marked functions in picker.cpp.

Here are a few things that might come handy as you implement the above:

- The scene graph will pass references to the scene graph nodes to your visitor. To store it in a stack, you want to convert it to a shared_ptr. This is accomplished by calling node.shared_from_this(), which has the return type shared_ptr<SgNode>.

- Give a shared_ptr<SgNode>, say called p. You can test whether it points to a SgRbtNode by doing

  ```
  shared_ptr<SgRbtNode> q = dynamic_pointer_cast<SgRbtNode>(p);
  ```

  If the cast succeeds, then q will point to what p points to, but it now has the type: pointer to SgRbtNode instead of SgNode. This is necessary because because while you can always cast a pointer of a derived class to a pointer of a base class, the other way around is not guaranteed to work. A SgNode doesn't always have to be a SgRbtNode. It might have been a SgShapeNode or SgRootNode, neither of which can be cast to an SgRbtNode. If the cast fails, then q will be NULL.

- Utility functions are provided that translate between integer IDs and RGB values. They are member functions of Picker and named idToColor and colorToId.

- Utility functions for maintaining the map from ID to SgRbtNode are provided. They are member functions of Picker and named addToMap and find. If the ID is not in the map, find just returns NULL.

- In Picker::getRbtNodeAtXY, you need to read a pixel from the framebuffer, convert it to an ID, and looks up the ID in the map for the RbtNode. To read back from the framebuffer, you use the OpenGL call glReadPixels. One usage of it is in ppm.cpp where the entire screen is read back and dumped to a PPM file. You should adapt it, but only read back 1 pixel positioned at the passed in x, y coordinates.

Then to use the picking visitor that you have completed, refer to asst4-snippets.cpp. In drawStuff, if picking is true, you should create an instance of Picker, and pass it to the scene graph using g_world->accept. As the scene graph is traversed, this visitor, when called on a shape node, computes a part ID and associates, in some lookup table, this ID with a pointer to the parent's transform node.

After the traversal is done, the member function getRbtNodeAtXY can be called on this picker that reads the back buffer, looks up the color in the table, and outputs a pointer to the appropriate transform node.

We recommend that you keep a global variable shared_ptr<SgRbtNode> g_currentPickedRbtNode to record the current picked node. It is used in the snippet.

After the modification to drawStuff, you can use the picker() function provided in asst4-snippets.cpp to do the picking. It takes care of using the right shader.

Task 3: Transform any Part

Next we will want to apply transform to the activated SgRbtNode node. Doing this properly will involve a bit of matrix work.

To do arcball, we will need the RBT relating the active node's frame to the world. For the elbow, this is $C(l)$. We provide this functionality in the function defined in scenegraph.{cpp|h}.

```
getPathAccumRbt(shared_ptr<SgTransformNode> source,
                shared_ptr<SgTransformNode> destination,
                int offsetFromDestination=0);
```

You pass it two transform nodes from the scene graph, it will return the multiplication of all RBTs associated with the transform nodes on the path from the source node to the destination node, EXCLUDING the rbt associated with the source node. So for example

- getPathAccumRbt(w, l)$= OSL = C(l)$,
- getPathAccumRbt(o, l) $= SL$,
- getPathAccumRbt(s, l) $= L$,
- getPathAccumRbt(l, l) $= I$.

The offsetFromDestination argument allows to back up from the destination node. For example getPathAccumRbt(w, l, 1) should return $C(s) = OS$.

RBT Accumulator

Internally `getPathAccumRbt` uses the visitor `RbtAccumVisitor` defined in `scenegraph.cpp`. You job is to complete the `TODO` marked member functions of `RbtAccumVisitor` so that `getPathAccumRbt` works properly.

This visitor's job is to build up an accumulated RBT stack as the scene is traversed. For joint "j", this is the accumulated $C(j)$ matrix described above. Once the target `target` is hit, `visit` should return `false`. This will cause the graph traverse to exit and stop the updating of the RBT stack. Then the `getAccumulatedRbt` member will allow you to get the last accumulated matrix from that saved stack in the visitor. A non zero offset value lets you get entries further down the stack.

To compare if the two nodes, `target_` and `node`, are the same, you can just write `if (target_ == node)` since the comparison operator is correctly defined for `SgNode` and its derived classes.

Viewing Using Any Transform Node

Recall that in `drawStuff`, the `invEyeRbt` is not calculated from the scene graph yet. Now suppose you want to view from the sky camera, which is `g_skyNode` in the scene graph. You can simply use

```
RigTForm eyeRbt = getPathAccumRbt(g_world, g_skyNode);
RigTForm invEyeRbt = inv(eyeRbt);
```

to get the `invEyeRbt` for `g_skyNode`. In fact, you can use the frame associated with any transform node as the eye frame.

So now you should make changes to your code so that when 'v' is pressed, you alternate between `g_skyNode`, `g_robot1Node`, and `g_robot2Node` as the eye frame.

Manipulating Any Transform Node

As with the quaternion assignment, the arcball calculation will give us the M action matrix. When right and middle buttons are pressed, your old code should give you the M action matrix used for translation. In either case, you already have M.

We will also need an auxiliary frame $\vec{\mathbf{a}}^t = \vec{\mathbf{w}}^t A$ that has its origin at the node's center (say the elbow), and has the directions of the eye. You will need to compute $A = (C(l))_T (C(e))_R$, where l is the elbow frame and e is the eye frame. Note that the translational part of A will require (from right to left) all of the RBTs starting from the world and ending at the elbow multiplied together (see `getPathAccumRbt` above). Similarly the rotational part of A will require all the RBTs starting from the world and ending at the eye frame.

We would like to do M to the elbow, with respect to $\vec{\mathbf{a}}$, but in our representation, we store (and wish to update) the relationship between the elbow and the shoulder, as in $\vec{\mathbf{l}}^t = \vec{\mathbf{s}}^t L$. Thus we want to do our subsequent work with the $\vec{\mathbf{s}}^t$ frame as the base, and not the world frame. So you will need to calculate a matrix A_s from A, such that $\vec{\mathbf{a}}^t = \vec{\mathbf{s}} A_s$. Then you can call `L = doMtoOwrtA(M,L,`$A_s$`)`. Recall that you can get $C(s)$ by calling `getPathAccumRbt(g_world,` l`, 1)`.

At this point, you should completely get rid of the old `g_skyRbt` and `g_objectRbt` from your code. Also remove the functionality of the 'o' key since we are using picking to activate a transform node for manipulation. Make suitable change so your code compiles and runs. Make sure the different viewing mode still works.

Task 4: 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.