Chapter 1

CS488/688 W12

A3: Introduction

"I consider this assignment to be a gift."
– A former CS 488 TA's evaluation of Assignment 3.

"NO IT'S NOT!!!"
– The response of the Fall 1996 CS 488/688 Class to the above comment

This assignment is due **Wednesday, February 15th [Week 7]**.

If you still need the provided code for this assignment run `/u/gr/cs488/bin/setup A3` from your CSCF account.

1.1 Topics

- Hierarchical models and data structures.
- Matrix stacks, undo/redo stacks.
- Scene parsing/scripting with Lua.
- 3D Picking.
- Z Buffer and backfacing polygon hidden surface removal.
- Filled and lighted polygons.
- Display lists.
- Virtual trackball for 3D rotation.

1.2 Statement

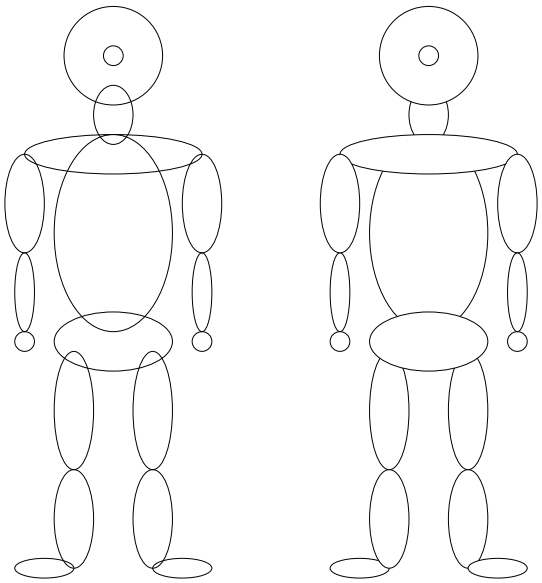
This assignment requires you to create a "puppet" in a hierarchical fashion from a number of instances of a transformed sphere primitive. You will build and manage a hierarchical data structure to represent the puppet. The puppet will be rendered and lighted interactively using OpenGL. You will also build a user interface to selectively manipulate the joint angles of the puppet, and to

globally rotate and translate the entire puppet. An undo/redo stack of joint transformations is maintained.

The faces of the sphere primitive should be composed of filled rectangles ("quads" in OpenGL) to make it appear solid, and should be drawn using an OpenGL display list for maximum efficiency. Models can be displayed with Z-buffer, backface polygon culling, and frontface polygon culling; each of these three OpenGL features can be turned on and off. The final puppet should be drawn using a suitable selection of lights and materials so that the 3D structure of the puppet is obvious. The position of the light sources should be fixed relative to the camera.

To create the puppet model, a sphere primitive should be appropriately transformed and instanced to create the torso (with its center as the origin of the puppet coordinate system). Smaller sphere instances for shoulders and hips should be positioned relative to this system. Their centres should form the origins for two secondary systems. With respect to the shoulder system, a neck sphere and two upper-arm spheres will form the origins of three subsidiary coordinate system origins. A sphere for the head will be positioned relative to the neck's center, and each sphere for the forearm will be positioned relative to the upper arm's center. The hands will be small spheres positioned relative to the forearms. The legs, consisting of thighs, calves, and feet, will be constructed similarly, with respect to the hip coordinate system.

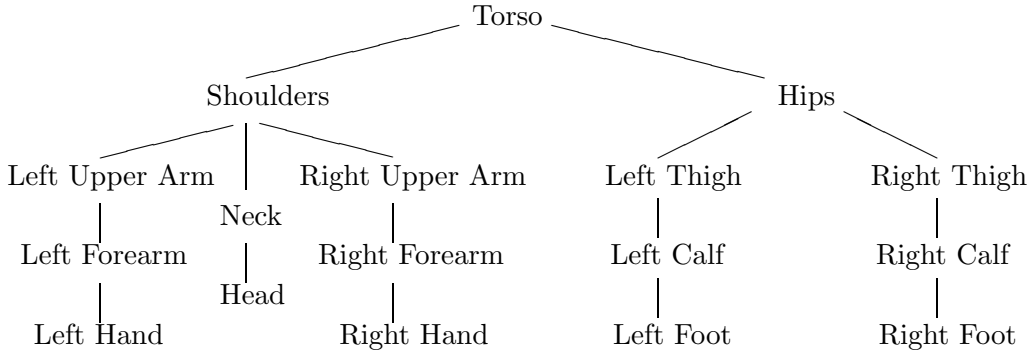
The general effect should be as shown:



where the figure on the left shows all overlapping, scaled spheres, and the figure on the right has hidden surfaces removed (your puppet need not have "pigeon toes"). The circle in the center of the face is a nose. You need at least one feature on the head so we can determine if rotations of the head are correct; it doesn't have to be a nose. Feel free to add eyes, ears, hair, mouth, antennae, etc., to your puppet.

You can be creative with your model, as long as it has the same or more degrees of freedom (number of joints) as this one. In the past people have built models of gorillas, dogs, Teddy bears, aliens, dinosaurs, gerbils, Oktoberfesters, etc. However, a "creative" model is not part of this assignment's requirements. Likewise you can add more modelling primitives if you want beyond the sphere, but it's not a requirement.

The coordinate system of each body part except the torso is therefore defined relative to a parent body part. The dependencies are as shown:



The nose has been left out of this hierarchy as it is part of the Head body part.

You are to implement primitives to build and render an arbitrary hierarchical model. You will then write a Lua script to build a hierarchical data structure to represent the puppet. Note the above structure is a physical representation of which part of the body is connected to what, not the actual data structure. For the actual data structure, you will need additional intermediate "transformation nodes" for each body part to get the spheres properly hinged at the ends and for the joints. It's also a good idea to not put children under nodes with nonuniform scales; create childless side branches instead. In particular, you will only want to put primitives at leaf nodes.

The definition of the puppet is to include **fifteen** free angles, three for each arm and leg, and three for the head and neck. These angles will be used to animate the puppet. The three arm angles will set the amount of rotation of the upper arm forward about the shoulder, the amount of rotation of the forearm forward about the elbow, and the amount of rotation of the hand backward and forward about the wrist, in the same plane as the elbow and shoulder.

The three leg angles will set the amount of rotation of the thigh forward about the hip, the amount of rotation of the calf downward about the knee, and the amount of rotation of the foot forward and backward about the ankle.

Of the remaining three angles, one will set the amount of rotation of the base of the neck forward with respect to the torso/shoulder, one will set the amount of rotation of the base of the head forward with respect to the neck, and the last will set the rotation of the head to the right or left relative to the neck.

All angles will have minimum and maximum values which the interface will enforce. For example, it should not be possible to rotate a knee or elbow the "wrong way", or to rotate a hand or foot more than 90 degrees from its neutral position.

Caution: In the past, students have found the creation of this model to be the most time consuming part of this assignment. You will probably want to design it on paper before writing the Lua code for the model.

1.3 Modelling

You are to model your puppet using a Lua script. The next section describes the Lua/C++ interface you need to implement. First, we take a quick look at how the interface is organized.

Lua is a simple scripting language. By using Lua to describe our scene we do not have to write a special-purpose parser for the scene. In C++, our scene will be represented as a number

of `SceneNode` instances. Each `SceneNode` object has a transformation associated with it, and may have child nodes. There are two classes derived from `SceneNode`: `JointNode` and `GeometryNode`. These classes represent special types of nodes. Geometry nodes are nodes at which actual geometry is present (in this assignment, spheres). Joint nodes are nodes which can be manipulated in the user interface to rotate joints of the puppet.

To begin constructing a puppet, we need a root for our modelling hierarchy. We can get one by asking for a transform node and assigning the result to a Lua variable. The name passed to the function is useful for debugging purposes.

```
myroot = gr.node('root')
```

Functions like `gr.sphere` create new geometry nodes. We can then use `add_child` to make a newly-created sphere a child of the root node:

```
torso = gr.sphere('torso')
myroot:add_child(torso)
```

You may wonder why we used a colon instead of a period in the last line. In Lua, ":" is used to call member functions on objects and "." is used to call regular functions or class functions. We might then set the material properties of and transform the torso:

```
torso:set_material(gr.material(...))
torso:translate(1.0, 2.0, 3.0)
```

Finally, we simply return the root node of the puppet:

```
return myroot
```

Conceptually, the transformation at a node is applied to both its geometry and its children, and matrices deeper in the tree are premultiplied by matrices higher in the tree. This assumes that column vectors are used to represent points.

The tree given in the previous section just relates the pieces of the puppet. The tree you create for your program will need to have additional joint nodes, containing transformations and a joint rotation range. You will transform these "joint" nodes to animate your puppet. You may also need extra nodes to prevent child nodes from being affected by transformations meant to modify only the geometry of a parent node. Watch out, in particular, for scales. You generally will not want these in the middle of a chain of transformations.

1.4 The Interface

The interface of your program should have at least a menu bar and a viewing area. The menu bar will have (at least) an `Application` menu, an `Edit` menu, a `Mode` menu and an `Options` menu. In addition, when the graphics display is resized the aspect ratio should be properly maintained—the rendering of the puppet might change size, but its proportions should not become distorted.

The Lua interface functions have been written for you, but they call other functions which you will need to implement. Here is a list of Lua functions that have been implemented (in `scene_lua.cpp`):

- `gr.sphere(`*`name`*`)` — Return a sphere with name *name*. The sphere should be centered at the origin with radius 1.
- `gr.node(`*`name`*`)` — Return a node *name* that just contains a transformation matrix, which is initialized to the identity matrix.
- `gr.joint(`*`name`*`, {`*`xmin, xinit, xmax`*`}, {`*`ymin, yinit, ymax`*`})` — Create a joint node with minimum rotation angles *xmin* and *ymin*, maximum rotation angles *xmax* and *ymax* and initial rotation angles *xinit* and `yinit` about the x and y axes.
- `onode:add_child(`*`cnode`*`)` — Add *cnode* as a child of *pnode*.
- `gr.material({`*`dr, dg, db`*`}, {`*`sr, sg, sb`*`}, `*`p`*`)` — Return a material with diffuse reflection coefficients *dr*, *dg*, *db*, specular reflection coefficients *sr*, *sg*, *sb*, and Phong coefficient *p*.
- `node:set_material(`*`mat`*`)` — Give the node *node* material *mat*. Node materials can be changed at any time.
- `node:rotate(`*`axis, angle`*`)` — Rotate *node* about *axis* (`'x'`, `'y'` or `'z'`) by *angle* (in degrees).
- `node:translate(`*`dx, dy, dz`* — Translate *node* by *(dx, dy, dz)*.
- `node:scale(`*`sx, sy, sz`*`)` — Scale *node* by *(sx, sy, sz)*.

You can partially verify your implementation with `a3mark.lua` and `a3mark.png`. The TAs will use this script to test some of the functionality of your assignment.

Note that while the Lua-C++ interface for these commands is completely implemented for you (in `scene_lua.cpp`), you will need to fill in the stubs in `scene.cpp`, and extend the scene data structures appropriately.

1.4.1 Application Menu

The `Application` menu should have the following items:

`Reset Position` —Reset the origin of the puppet to its initial position. Keyboard shortcut `I`.

`Reset Orientation` —Reset the puppet to its initial orientation. Keyboard shortcut `O`.

`Reset Joints` —Reset all joint angles, and clear the undo/redo stack. Keyboard shortcut `N`.

`Reset All` —Reset the position, orientation, and joint angles of the puppet, and clear the undo/redo stack. Keyboard shortcut `A`.

`Quit` —Terminate the program. Keyboard shortcut `Q`.

1.4.2 Mode Menu

The `Mode` menu should have the following radiobutton selections:

`Position/Orientation` —Translate and rotate the entire puppet. Keyboard shortcut `P`.

`Joints` —Control joint angles. Keyboard shortcut `J`.

Initially, the program should be in the `Position/Orientation` mode.

The behaviour of this interface is detailed in the following.

`Position/Orientation:` The code in `/u/gr/cs488/demo/trackball` illustrates the interface you should implement for translating and rotating the puppet. You can use the `trackball.c` code here as a basis for your implementation.

The following details the behavior of each mouse button. If you have questions about how this interface works, test `intdemo` and make your interface match it.

- Dragging the mouse with B1 depressed changes the x and y translation of the puppet.
- B2 depressed changes the z translation; moving the mouse up on the screen should move the puppet away from the viewer, while moving the mouse down on the screen should move the puppet closer.
- Use B3 to implement a virtual trackball direct-manipulation-3D-orientation interface. The "virtual sphere" should be centered on the screen and its diameter should be 50% of the width or height of the raster widget, whichever is smaller.

  The skeleton code will display a circle centred on the screen. This circle corresponds to the virtual sphere. The radius of the circle will be a quarter of `min(screen width, screen height)`.

The translation should be done relative to the view frame. The rotation should also be performed relative to the view frame, translated to the puppet's origin. Since in this assignment the view frame is fixed, you can make this easy by putting the camera in a simple canonical world-frame position. Rotating the puppet about its own origin can be done with an appropriate transform node above the torso.

`Joints:` Clicking B1 is used as an event to signal picking. The cursor will be used with this signal to select a part of the puppet model.

Selecting is regarded as a toggle operation. If an item is unselected, and it is picked, it will become selected. If the same item is picked again, it will become unselected. Selecting an item should cause a visible change in the rendering of that item (change the material).

Only items that are directly under a movable joint should be selectable. When an item is selected, the joint immediately above it in the hierarchy will be affected by the user interface. If an item is not selectable, i.e. it is not immediately under a movable joint, picking it should do nothing. In particular, there should be no visible change in unselectable objects when they are picked.

Multiple simultaneous selections should be permitted.

For all selected items, the relative mouse y motion with B2 pressed will be mapped to the angles of the joint immediately above each selected object in the hierarchy. In addition, the

head will rotate to the left and right with B3 depressed, in response to relative x motion of the mouse. It should be possible to depress both B2 and B3 simultaneously and simultaneously nod and rotate the head. The head should only move if it is selected, for both B2 and B3 motions.

1.4.3 Edit Menu

The `Edit` menu should have the following items:

`Undo` —undo the previous transformation on the undo/redo stack. Keyboard shortcut `U`.

`Redo` —redo the next transformation on the undo/redo stack. Keyboard shortcut `R`.

You should maintain an undo/redo stack of transformations. All joint transformations should be saved to the stack. The paradigm is that when you release a mouse button when in joint manipulation mode, the current joint angles are stored on the undo/redo stack. Initially, the stack contains the "reset" joint angles. If we think of the stack as extending upwards, you should maintain a current position in the stack. Selecting undo from the edit menu will restore the joint angles to the entry below the current one on the stack (and update the stack pointer to point at that entry). Selecting redo will update the joint angles to the entry above the current one on the stack (and update the stack pointer to point at that entry).

If several joint transformations are undone, and a new joint transformation is initiated, all undo/redo entries above the current one should be cleared. If Reset All or Reset Joints are selected from the File Menu, the undo/redo stack should be cleared. You should provide reasonable feedback to indicate that an attempt to undo or redo past the end of the stack has been attempted (and not allow the action).

1.4.4 Picking Menu

The `Picking` menu should be implemented if you are unable to get selection with the mouse working properly (objective 3). Create a checkbox menu item for each selectable part. You will not get a mark for objective 3, but you will be able to implement and demonstrate joint movements.

1.4.5 Options Menu

The `Options` menu should have the following checkbox item:

`Circle` —draws the circle for the trackball if checked. Keyboard shortcut `C`.

`Z-buffer` —draws the puppet with the OpenGL z-buffer if checked. Keyboard shortcut `Z`.

`Backface cull` —draws the puppet with backfacing polygons removed. Keyboard shortcut `B`.

`Frontface cull` —draws the puppet with frontfacing polygons removed. Keyboard shortcut `F`.

All options should default to being "off".

1.5 OpenGL

Most of what you need to learn about OpenGL is in the programming guide and man pages. However, here are a few items that are hard to find, and a few hints to make the assignment easier. None of the following are requirements on your program; they're just hints that students in past terms have found helpful.

- Unit normals. To get the lighting calculation correct, you must use unit normals. Although it is trivial to specify unit normals for the faces of the sphere, when you scale the sphere, the normals will also be scaled, making them of non-unit length. To avoid this problem, use the appropriate `glEnable` command to tell OpenGL automatically scale all normals to unit length.

- Material properties. You can set ambient, diffuse, and specular material properties. For debugging purposes, it is probably easiest if you turn off the specular material properties (or rather, if you never turn them on).

- Lights. For debugging purposes, it is easiest to have a single light source, placed at the eye. This will allow you to decide more easily if the shading of the puppet looks correct.

- Normals, Colours, Vertices. When you give the `glVertex` command, OpenGL assigns the normal specified by the most recently issued `glNormal` command; likewise for `glColor`, which can be set up to modify the diffuse reflection colour when lighting is enabled.

  Thus, when specifying the normals and colours of the vertices, you must specify them BEFORE you issue the `glVertex` command.

You are required to use a *display list* to render the sphere. A display list is way of storing all vertices and normals as they are calculated so that OpenGL can reuse them rather than have to recalculate (and retransmit) them to OpenGL. This is sometimes called "retained mode", as opposed to the usual way of using OpenGL, which is "immediate mode".

Although you may use the `gluSphere` command if you want, you may want to generate the sphere yourself using sines and cosines, and drawing the sphere as a set of rectangles. For extra performance improvements, draw your puppet as a sequence of *quad strips* (this is not required, though). Regardless, you should select a resolution of the sphere that results in a good quality sphere while still allowing for smooth motion.

1.6 Donated Code

In `/u/gr/cs488/data/A3` you will find

- `appwindow.cpp` – The application window.

- `algebra.cpp` – Routines for geometry, ferrets, etc.

- `main.cpp` – The program entry point.

- `material.cpp` – Classes and functions for manipulating materials.

- `primitive.cpp` – Classes and functions for manipulating geometric primitives such as spheres, etc.
- `scene.cpp` – Classes and functions defining modelling hierarchies. The layout of these classes should provide some ideas about how to organize your code.
- `scene_lua.cpp` – The Lua/C++ interface.
- `viewer.cpp` – The GTK+ OpenGL viewer widget.
- `Makefile` – Used to build the program.
- `a3mark.lua` – A Lua script to test your implementation.
- `a3mark.png` – An image showing what the screen output of `a3mark.lua` should look like.

These files have been copied to your `handin/A3` directory. You should also use `trackball.c` which you can find in `/u/gr/cs488/demo/trackball`.

Note that the matrix multiplication routines are not needed for rendering, only to set up the matrices in the nodes.

1.7 Deliverables

Executables: The following two files are to be in your `A3/` directory:

`puppeteer` - Your puppet viewer. This program should take a single argument specifying a Lua puppet file to load. If no arguments are given it should load `puppet.lua` from the current directory.

`puppet.lua` - Your puppet.

The supporting source code should also be available in the `src` directory.

Documentation:

- Document changes you make to the data structure.
- Document the structure of the hierarchical model of your puppet.

For this assignment, you are encouraged to make creative models. The instructor and TAs may assign up to 1 bonus mark for interesting puppets.

1.8 Objectives: Assignment 3

Due: Wednesday, February 15th [Week 7].

Name: _____

UserID: _____

Student ID: _____

___ **1:** `a3mark.lua` runs and displays correctly.

___ **2:** The puppet's proportions are reasonable and the spheres are joined together in a logical fashion. The spheres are "hinged" to their neighbours at their ends—they do not rotate about their centres.

___ **3:** Picking works correctly and reliably.

___ **4:** Selection records are kept correctly so that any sequence of picks-to-select and picks-to-unselect works. Multiple selections are supported. Selected portions of the puppet are clearly indicated—e.g. selected parts change colour, or their edges are highlighted.

___ **5:** The puppet can be globally rotated and translated for viewing purposes. The rotation user interface is a virtual trackball. A circle representing the virtual sphere can be turned on or off from the menu. It must be clearly visible when it is turned on. The puppet's configuration can be reset from the menu.

___ **6:** A well-designed hierarchical data structure is employed. Each sphere is drawn as an instance of a single display list sphere.

___ **7:** The joint movements are correct and the angles are restricted so that no grossly unnatural configurations are allowed. The puppet does not fly apart or distort during any sequences or combinations of actions.

___ **8:** Z-buffer, backfacing and frontfacing polygon hidden surface removal are implemented and each can be toggled on and off.

___ **9:** The puppet is lit such that its 3D structure is clearly visible.

___ **10:** An undo/redo stack is maintained.

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature: