Developing a Virtual Environment for Integration with Exoskeleton Arms

Ben Farley

Advisor: Jacob Rosen
Graduate Student: Levi Miller

I spent the summer creating a virtual environment that could be integrated for use with our exoskeleton arms. I did so using an open source graphics platform called Coin3D. The code I have written can be found in the Visual Studio Projects folder on the main VR computer. This is located in My Documents -> Visual Studio 2008 in Levi's account.

Developers wishing to create games using this environment or add to the code I have already written can skip straight to "The simulation." However, the other sections contain information that may be relevant to anyone wishing to modify the existing code.

An in-depth tutorial on how to use Coin3D is far beyond the scope of this paper. Instead, I will focus on my design of the environment and my reasoning behind any decisions that may not appear obvious. For people trying to learn Coin3D for the first time, I would recommend looking at *The Inventor Mentor* [1] and *The Inventor Toolmaker* [2]. These are a pair of excellent guides on how to use the Coin3D environment.

Creating the Entity

The entity that represents the robot and its arms is a simple combination of geometric objects. The torso and head are made up of spheres and cylinders. The entity currently has only one arm, but the existing code could easily be copied with minimal modifications to add a second arm. The arm itself is also made up of spheres and cylinders. The shoulder joint, elbow joint, and wrist joint are all small spheres. These spheres do not perform any particular purpose, other than making the arm more aesthetically pleasing when joints are bent. Between the shoulder and elbow spheres, and between the elbow and the wrist spheres, are cylinders representing the upper and lower arm. The hand itself is currently modeled as a flat cube, to represent a paddle or racquet of some sort.

Since we want to make this virtual arm mirror the movement of the actual arm, we need to give it 7 Degrees of Freedom (DOF) – 3 in the shoulder, 1 in the elbow, and 3 in the wrist. To do this, we add SoRotation nodes before drawing each part of the arm. SoRotation nodes contain an axis of rotation and an angle. We get the axes of rotation from [3], and the angles we receive as data from the arms themselves. The steps in drawing the arm are roughly as follows:

- Move to shoulder location
- Draw shoulder sphere
- Rotate axis by $\Theta_1, \Theta_2, \Theta_3$ (in that order)
- Draw upper arm
- Move to elbow
- Draw elbow sphere
- Rotate axis by Θ_4
- Draw lower arm
- Move to wrist
- Draw wrist sphere

- Rotate axis by Θ_5, Θ_6, Θ_7 (in that order)
- Draw hand

Once we connect to the robot and periodically update the angles of the rotation axes, this will ensure that the arm of the virtual entity reflects the movement of the exoskeleton arms.

Moving the Arms

To move the arms of the virtual entity, we need to update the Rotation nodes of our world with the new joint angles based on how the exoskeleton arms are moving. We do this using sockets. Before beginning the simulation, the program establishes a socket connection to the computer running the exoskeleton arms' software. It then enters an infinite loop where it waits to receive something from the arms. When it does, it copies the data it receives into an array of floats.

This part of the program runs in a separate thread from the rest. The arms are sending data to the program 1000 times a second (1 kHz), so the program must be able to receive data at 1 kHz as well. If it goes slower, the data coming from the robot will build up in its buffer, causing delays between the exoskeleton arms moving and the virtual arms moving. It is not possible to schedule a task to run this fast using Coin3D's scheduling system – too many other things are happening in that thread, so the connection to the robot ends up receiving data less than 1000 times a second, creating the above-mentioned delays. Therefore, we move the connection process to a different thread, where it can run as fast as it can. To do this, we create a simple Thread class, Thread.cpp, and a subclass of Thread, RobotConnection. Upon creation of the thread, RobotConnection opens a socket connection to the computer running the arms' software. It then enters an infinite while loop in which it waits to receive a packet from the arms. When it does, it simply copies that data to a local array and then waits for more packets.

Since Coin3D does not like other threads modifying its SceneGraph, the RobotConnection thread does not directly update the Rotation nodes of the world when it receives data. Instead, it writes the new angles to a local array and Coin3D accesses these variables whenever it needs to redraw.

Physics Engine

One of the main differences between Coin3D and Microsoft Robotics Studio (which previous work on the project had used) is that Coin3D contains no built in physics engine. This is both good and bad – bad because we have to write our own physics engine, but good because we can design the physics of the world however we want. MRS focuses on gaming; as such, it wants to be able to support a physical environment where many objects are interacting with and affecting each other. To do this, it sacrifices some of the accuracy of its physics in order to achieve speed. In the simulations we will be creating, we do not need a complex world. We will have a ball, some objects it can bounce off of, and a hand that can interact with the ball. Therefore, we can focus on the accuracy of our physics rather than the speed.

The only equations we need are Newton's Second Law and some simple rules about distances and positions in 3D space. With Newton's law, we can find an object's position based on the forces acting on it. Therefore, at each time step, we need to calculate the net forces acting on the ball. For this, we have two different options – the ball is colliding with an object, or it is not. If the ball is colliding, we have $F_{net} = F_g + F_n + F_{damp}$. F_g is gravity. F_n is the normal force acting on the ball, with magnitude based on how deep into the object the ball is penetrating. F_{damp} is a dampening force that we artificially add to the world in order to sap some of the ball's energy every time a collision occurs. This is necessary because

every time the hand hits the ball, energy is added to the system. If we didn't have some force sapping energy, the energy of the ball would simply continue to increase and the ball would go faster and faster.

The Simulation

The simulation combines all of the above actions in various classes and methods. The main() method is contained in BasicRoom.cpp. This method creates an object of the Simulation class, adds some objects to it, and begins the simulation. Main does not have to deal with any of Coin3D's internal workings – the Simulation class handles that. Games can be modeled after the current demo, with a main() method in which a Simulation object is instantiated and the desired objects are added to the world.

The Simulation class does most of the work in the program. This is where all the Coin3D rendering happens and where modifications to the scene graph occur. Any adding of objects will be done by the Simulation class, preferably by creating a class method for objects of that type, such as Simulation::addObject(parameters). The \Simulation class currently contains methods to add the robot, a simple room, and a ball to the world. It is assumed that the program will almost always contain the robot and the ball, with additional world objects as necessary.

To control the physical properties of all objects that exist in the world, each instance of a Simulation also contains an instance of a WorldPhysics object. The WorldPhysics object contains a list that holds all of the objects in the scene graph with which the ball can collide. It also contains a separate list for the two planes of the hand, as well as a list for the four bounding planes of the hand. The WorldPhysics object also contains various physical constants, such as gravity, k (for collisions), and the dampening constant. These are set in the constructor of the WorldPhysics class. If necessary, getters and setters for these values could easily be added. There are various internal WorldPhysics methods that the developer can ignore – for example, getHandDistance(), getClosestHandPlane(), and updateHandPhysics(). These are all used in determining collisions and distances in Simulation::TimeStep(), but for simply creating games, these methods can be treated as black boxes.

To add an object to the world, we need to add a visual and, for some objects, a physical representation. The process goes as follows:

- Create Coin3D visual object
- Add physics planes (if any) to the Simulation's WorldPhysics object with WorldPhysics::addPhysicsObject()
- Add Coin3D object to the root node of the scene graph

Note that the arms and body of the entity do not have any physical representation in the world; they are purely visual. This means that the ball will not collide with or bounce off them, it will just pass through them. Future work on this project may include making these objects physical.

The method that is the driving force behind the simulation is Simulation::TimeStep(). This is where all collision detection, position changes, and movement of objects is done. Note that this method is static – this is because we use a Coin3D class called a TimerSensor to schedule the task to occur, which does not work with non-static class methods. The TimeStep() method progresses as follows:

- Update the position of the arm and hand
- Get the position of the ball
- Check for collision between ball and nearest hand plane; modify net force if necessary
- Check for collision between ball and each wall; modify net force if necessary

- Update velocity and position based on net force

For simple programs and games, this method should not have to be modified by the developer. As long as you follow the previously mentioned steps when adding an object to the world, the TimeStep() method will correctly detect collisions between the ball and all physical surfaces in the world. For more complex programs, this method may have to be modified slightly. For example, if the ball needs to collide off of more complex objects than single planes, or if the ball should bounce off of the arm of the entity, changes may be necessary.

Resources

[1] *The Inventor Mentor*
http://webee.technion.ac.il/~cgcourse/InventorMentor/The%20Inventor%20Mentor.pdf

[2] *The Inventor Toolmaker*
http://openscientist.lal.in2p3.fr/download/etc/toolmaker.pdf

[3] Levi's paper containing important information about arm size and joint angles
Gravity Compensation for a 7 Degrees of Freedom Powered Upper Limb Exoskeleton, by Levi Miller

[4] Coin3D online documentation
http://doc.coin3d.org/Coin/

[5] Coin3D mailing list archive
https://mailman.coin3d.org/pipermail/coin-discuss/