Modern Robots: Evolutionary Robotics

Programming Assignment 5 of 10

Description

In this assignment you will start with the empty simulation you created in assignment 4 and incrementally add objects to construct a quadrupedal robot. In the next assignment you will add joints to your simulation so that the objects are connected together, then sensors, then motors, and then a controlling neural network.

1. Create a directory, Assignment 4, that contains your assignment 4 submitted document. Copy the bullet-2.81 (version may vary) directory into this directory as well. Back up this directory. Now in subsequent assignments if you find your simulation becomes unusable you can go back and retrieve the empty simulation stored in this directory.

2. Create a new directory, Assignment 5, and copy the bullet-2.81 directory here. For the remainder of this assignment make changes to the files in this bullet directory.

3. Take a moment to understand Fig. 1. Either draw it on a piece of paper, or print it. Keep this around, as you will be annotating the image through the next few assignments.

4. Now draw the robot from the three additional perspectives.

5. For the four upper and four lower legs, mark their positions, sizes (radius and length) and orientations in the three panels. You can choose the lengths and radii of the legs to your liking.

6. Now, in the RagdollDemo.h program, we need to create body and geom data structures to store all the objects that make up the robot. Find the member variable declarations for the RagdollDemo class

```cpp
class RagdollDemo : public GlutDemoApplication
{
    btAlignedObjectArray<class RagDoll*> m_ragdolls;
    //keep the collision shapes, for deletion/cleanup
    btAlignedObjectArray<btCollisionShape*> m_collisionShapes;
    btBroadphaseInterface* m_broadphase;
    btCollisionDispatcher* m_dispatcher;
    btConstraintSolver* m_solver;
    btDefaultCollisionConfiguration* m_collisionConfiguration;
    btDefaultCollisionConfiguration* m_collisionConfiguration;
}
```

and add the following variables:

*Original material was graciously provided by Josh Bongard. Jeff Clune slightly modified it.*
btRigidBody* body[9]; // one main body, 4 x 2 leg segments
btCollisionShape* geom[9];
bool pause;

7. Recompile the code until you compile and run without errors.

8. Now you will create some functions that can be used to add objects to the empty simulator. Create a function of the form

```cpp
void CreateBox( int index, double x, double y, double z, double length, double width, double height) {
...
  body[index] = ...
...
  geom[index] = ...
...
  m_dynamicsWorld->addRigidBody(body[index]);
}
```

which will be used to create the main body of the robot. Refer to the rag doll code to figure out how to create an object. Assume the mass of all objects for now is 1.

Note: Do NOT call the localCreateRigidBody function, but instead use its implementation in Ragdoll to discover how to add rigid objects to the simulation. FYI: the localCreateRigidBody function is both a private function of Ragdoll and a public function of DemoApplication. To find out how to create a rigid body, you'll need the localCreateRigidBody of the Ragdoll implementation, but if you (incorrectly) call localCreateRigidBody, you would actually be calling the public function of the DemoApplication.

Without calling the function, recompile and run until you have no errors.

9. Create a similar function for creating cylinders `CreateCylinder(index,...)`, which will be used to create the upper and lower legs of the robot. You will need to add additional parameters for specifying the orientation of the cylinder.

Note: Create your cylinders using: `btCylinderShapeX`, `btCylinderShape`, and `btCylinderShapeZ`. `btCylinderShapeY` does not exist (because it is the default dimension). Do NOT rotate these objects (e.g. by passing a Quaternion with a value other than 0 for the radians).

Recompile and run until error-free.

Create another function

```cpp
void DeleteObject( int index ) { ... }
```

which can be used to remove objects from the simulation. This function should destroy both the body and the geom associated with the object using delete. Recompile and run until error-free.

11. Now, in the `initPhysics` method in RagdollDemo.cpp, add code to create the first object

```cpp
//spawnRagdoll(startOffset); Commented out in exercise 4b.
CreateBox(0, 0., 1., 0., 0., 0., 0.2, 0.); // Create the box
clientResetScene();
```
Compile and run until error-free. You should see the box fall and come to rest on the ground as in Fig. 2a. (You can toggle texture drawing by hitting the u key.) Screencapture this image and paste into your document.

12. Create box will allocate memory for the body and geom objects. As it is, our simulation will run, but it will leak memory. If you reset the simulation enough times by hitting the space bar, it will eventually run out of memory. So we need to make sure to delete the objects we create. In RagdollDemos exitPhysics method, delete objects at the beginning of the method. The following is an example for deleting object 0:

```cpp
void RagdollDemo::exitPhysics() {
    DeleteObject(0);
    ...  
}
```

13. We're going to add the ability to pause the simulation. Modify the call to clientMoveAndDisplay to look like the following.

```cpp
if (!pause) {
    m_dynamicsWorld->clientMoveAndDisplay(ms / 1000000.f);
}
```

Recompile until there are no errors. Note that the simulation can technically be paused, but there’s currently no means of pausing it from the UI.

14. We want to be able to toggle whether the simulation is paused while it is running. Modify the method keyboardCallback in RagdollDemo.cpp to toggle the variable pause when the key ‘p’ is pressed. Recompile and run and verify that after ‘p’ is pressed, the simulation is paused. When ‘p’ is pressed again, the simulation continues running.

15. Now pause your simulation by hitting ‘p’, then reset your simulation by hitting the space key. You should see the box hanging in midair as in Fig. 2b. Screencapture and paste into your document. Hitting ‘p’ will unpause the simulation and cause the object to fall to the ground.

16. Add CreateCylinder(1,...) after the call to CreateBox(0,...) is called to add the next object to the robot. Note that you will have to specify the orientation of the cylinder, which you do not have to do for the main body. When run in paused mode you should see both objects as in Fig. 2c. Screencapture and copy and paste into your document.

17. Add a third object, compile, run and ensure that the object appears where you expect it. Continue to add an object and recompile until all nine objects are added. This should produce a simulation as shown in Fig. 2d. Screencapture, copy and paste into your document, and submit.
Figure 1: A template for sketching the quadrupedal robot to be simulated.

Figure 2: The quadrupedal robot under construction.