9
Soft Body Dynamics

Soft bodies are an alternate type of collision object to rigid bodies; they deform as a result of collisions and that allows us to simulate objects made from soft and malleable materials. In this chapter, we will be exploring the nuances of soft body physics and adding a soft body to our scene.

[Continue from here using the `Chapter9_SoftBodyDynamics` project files.]

Soft body requirements

One way to think of soft bodies is having a non-rigid constraint on each vertex. When one vertex is moved, the rest move with it, but not to the same extent, and each tries to maintain their distance from their nearest neighbors, or in other words, maintain their original pose. Thus, with a robust constraint system in place (like Bullet has), the mathematics of soft bodies is essentially a larger scale version of the same concept. There's a lot more that goes into the soft body physics simulation, but we don't need a PHD in theoretical physics to add one to our simulation.

The `btSoftRigidDynamicsWorld` world object is required for soft body simulation. This is a requirement since soft bodies are much more mathematically complex than rigid bodies, and so an entirely different world is required to perform the work necessary to move them through space and simulate them correctly.

In addition, this world also requires a slightly different collision configuration object called `btSoftBodyRigidBodyCollisionConfiguration`, and an extra object called `btSoftBodyWorldInfo` is an added attachment onto the soft body worlds, which performs some extra initialization for our world to function.

Note that this chapter's source code uses a `SoftBodyDemo` application to specifically test this feature.

Finally, because of the complexity of soft bodies, an entire project/library file must be included in the project in order to compile and launch the application with soft bodies. We must add the `BulletSoftBody` project, and the `BulletSoftBody_vs2010_debug.lib` file.

Initialization

Initialization of the world and collision configuration for soft bodies are not particularly special.

```
m_pCollisionConfiguration = new
  btSoftBodyRigidBodyCollisionConfiguration();
m_pWorld = new btSoftRigidDynamicsWorld(m_pDispatcher,
  m_pBroadphase, m_pSolver, m_pCollisionConfiguration);
```

These calls are identical to the calls used earlier, except using new class types (with much longer names). The most significant change to the soft body world initialization is with the `btSoftBodyWorldInfo` object:

```
btSoftBodyWorldInfo m_softBodyWorldInfo;
m_softBodyWorldInfo.m_dispatcher = m_pDispatcher;
m_softBodyWorldInfo.m_broadphase = m_pBroadphase;
m_softBodyWorldInfo.m_sparsesdf.Initialize();
```

Suffice it to say that this object simply needs the pointers for the collision dispatcher and broad phase objects, and must have it's **Signed Distance Field** (**SDF** for short) initialized so that it can generate proper collision detection for the world's soft bodies. Diving into the guts of soft bodies could take forever; there are entire volumes of scientific papers on the subject, so we will only be giving concepts such as SDFs a very cursory examination.

The SDF is a data structure that generates a more simplified (sparse) version of the soft body for collision detection in order to improve processing time, and is used to detect the distances between the soft body and other objects. After initialization, it will actively communicate with the dispatcher and the broad phase to generate collision responses for soft bodies (hence it needed the pointers to them).

Creating soft bodies

To create a soft body, we will utilize `btSoftBodyHelpers`, which contains many useful functions to simplify the act of generating these complex objects. We will use `CreateEllipsoid()` to build a sphere of triangles (an ellipsoid with equal dimensions is just a sphere), and then configure it with some additional commands. We won't be using `GameObject` for this object, because `CreateEllipsoid()` already generates the entire object for us.

```
btSoftBody* pSoftBody =
  btSoftBodyHelpers::CreateEllipsoid
  (m_softBodyWorldInfo,btVector3(0,0,0),btVector3(3,3,3),128);
m_pSoftBodyWorld->addSoftBody(pSoftBody);
```

As with rigid bodies and constraints, we need to specifically add a soft body to the scene before it shows up. This is accomplished by calling the `addSoftBody()` function on our world object. But, before we attach it, we need to perform some additional initialization.

A soft body is (obviously) not a very rigid structure. It deforms as it collides with other objects. But, the question is how does it deform? What is this object's resistance to being crushed? What is it's ability to maintain its own shape when sitting stationary? How heavy is the entire volume? How much friction does it suffer when morphing and rolling over surfaces? These are all the variables that can be tweaked in any given soft body, making them the most complex type of collision shape that can be found in a Bullet.

The two key values that we can set are the soft body's volume conservation coefficient and it's linear stiffness. Each of these values affects a specific property of the soft body, altering how well it maintains its original shape as it moves and collides with other objects.

```
// set the 'volume conservation coefficient'
pSoftBody->m_cfg.kVC = 0.5;
// set the 'linear stiffness'
pSoftBody->m_materials[0]->m_kLST = 0.5;
```

Note that `m_materials` is an array of different materials, which can be assigned to different sections of the same soft body if desired. Make sure that you manipulate it keeping this is mind.

The remaining initialization for our soft body comes through the `setTotalMass()` and `setPose()` functions. As we might expect, `setTotalMass()` simply sets the mass of the soft body, but it also has a profound effect on how the object deforms when it collides with other objects. If we want our shootable boxes to distort the soft body, its mass needs to be relatively low. If we want it to ignore them, then we should set the mass very high.

Finally, `setPose()` is used to generate the necessary constraints of the soft body, telling it to maintain the current pose in which its constituent vertices are positioned. This function takes two booleans, determining if the soft body should attempt to maintain its volume and frame, respectively; each has a significant effect on how the soft body moves.

```
// set the total mass of the soft body
pSoftBody->setTotalMass(5);
// tell the soft body to initialize and
// attempt to maintain the current pose
pSoftBody->setPose(true,false);
```

Rendering soft bodies

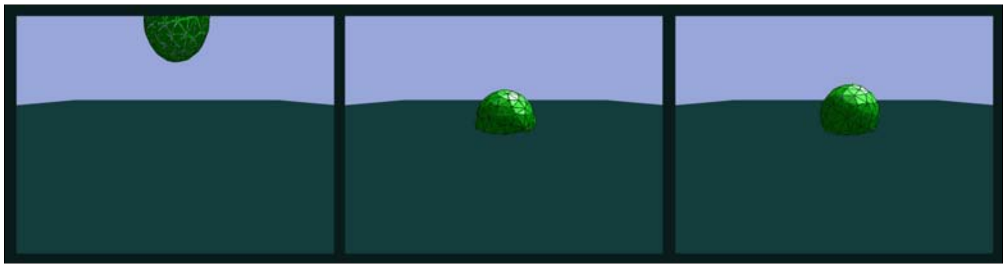
Our base application doesn't know anything about soft bodies (nor should it), so we will need to extend the `RenderScene()` function to handle our soft body rendering code. We make use of `btSoftBodyHelpers` again, which contains a function that will help us render our soft body through the very same rendering code that we use to draw the debug lines on the screen. This will require us to add one more function override in our debug drawer to render triangles in addition to lines.

Because our soft body is not built from `GameObject`, we need to handle its rendering a little differently than before. We can obtain and iterate through any soft bodies in our scene by calling the `getSoftBodyArray()` function on our world object and then use our debug drawer to render each of its triangles:

```
// check the list of our world's soft bodies
for (int i=0; i< m_pSoftBodyWorld->getSoftBodyArray().size(); i++)
  {
  // get the body
  btSoftBody*  pBody = (btSoftBody*)m_pSoftBodyWorld-
    >getSoftBodyArray()[i];
  // is it possible to render?
```

```
    if (m_pSoftBodyWorld->getDebugDrawer() && !(m_pSoftBodyWorld-
      >getDebugDrawer()->getDebugMode() &
        (btIDebugDraw::DBG_DrawWireframe))) {
      // draw it
      btSoftBodyHelpers::Draw(pBody, m_pSoftBodyWorld-
        >getDebugDrawer(), m_pSoftBodyWorld->getDrawFlags());
    }
  }
```

Launching our application now, we should observe a sphere fall, collide with the ground, and deform. We can also launch boxes with the right-mouse button to deform it even more. The following screenshot shows our soft body falling from the sky and deforming under its own weight:



Note that we cannot pick up and move this object with the left-mouse button because it is not a rigid body, and our picking code exits only if it detects so (otherwise it would crash!).

Soft bodies are very complex objects with a lot of mathematics behind them. Consequently, they are processor intensive, and there are many values that can be tweaked to generate the desired effect. This data can be accessed through the following two member variables of the soft body object: `m_cfg` and `m_materials`.

To see more interesting scenarios involving soft bodies, check out the `App_SoftBody` demo in Bullet's demo applications.

Summary

In this chapter, we have taken an introductory look at soft body physics, and several helper functions that can provide more advanced functionality if needed. There is much more that can be done with these interesting objects, and there are even more features that the Bullet library offers us, but regrettably we must begin wrapping up the book and leaving behind some closing thoughts.

All that's left is to say is farewell and good luck with your future game development projects. There's always more to learn and understand about game development, which makes it a very tough field to work in and keep pace with. But, for many of us, we wouldn't want it any other way, because if it was easy it would be boring!

We hope that you've learned a great deal about the fundamentals of game physics and graphics with this book, and have the drive to continue learning everything you need to bring your awesome game ideas to life!