Bullet Physics and the Blender Game Engine

Blender is a unique 3D animation suite in that it contains a full-featured game engine for the creation of games and other free-standing interactive content. Nevertheless, in spite of its core of devoted fans, the game engine is an all-too-often overlooked feature of Blender.

In this chapter, you'll learn the basics of working with BGE and Bullet Physics to the point that you can easily create dynamic rigid body animations, which can then be used in the Blender animation environment you're accustomed to. If you are new to BGE, you will be surprised by how much amazing functionality it will open up for you. If not, this chapter should provide some interesting ideas for how to integrate real-time generated physics simulations with rendered animations.

6

Chapter Contents

The Blender Game Engine

It goes without saying that for people who are mainly interested in creating games, the game engine is one of Blender's major attractions. The BGE is widely used by hobbyist game creators, and lately its appeal has begun to broaden to larger game projects. Luma Studios used BGE to create their prototype racing game ClubSilo, shown in Figure 6.1. The Blender Institute, the creative production extension of the Blender Foundation, is currently planning "Project Apricot," which will use BGE in conjunction with the Crystal Space game development kit and other open source tools in producing an open game of professional quality to be released under the Creative Commons license.

Figure 6.1 ClubSilo

As everybody who's ever played a game knows, game worlds can be pretty rough-and-tumble places. Cars crash, walls crumble, and bad guys go flying across rooms. This is all made possible by physics simulation, and any quality game-creation tool these days needs a good real-time Newtonian physics library. For Blender, this is the open source Bullet Physics Library. Indeed, Blender is not alone in this; the Bullet Physics Library, authored by Erwin Coumans, simulation lead at Sony Computer Entertainment America, is being used by professional game developers with parallel optimizations for PlayStation 3 and Xbox 360. The companies that use Bullet contribute back to Bullet under the zlib license.

Recently Bullet was showcased in an entertaining way in the Bullet Physics Contest 2007, in which contestants competed to create the most impressive Rube Goldberg machine in BGE. Some stills from Christopher Plush's winning machine can be seen in Figure 6.2.

This chapter shows you how to access the features of this powerful library for use in your own animation work.

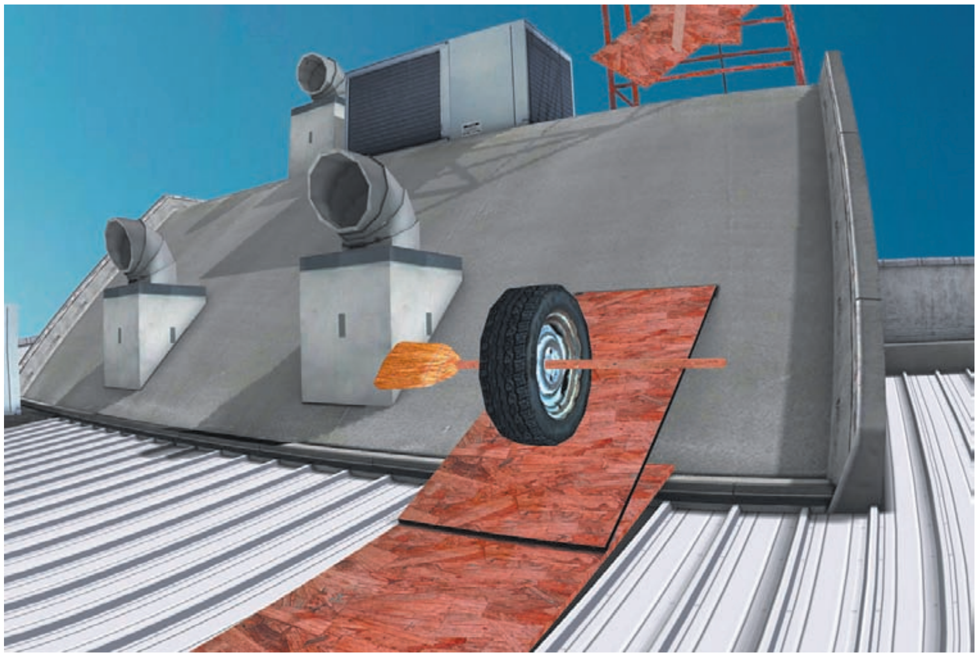
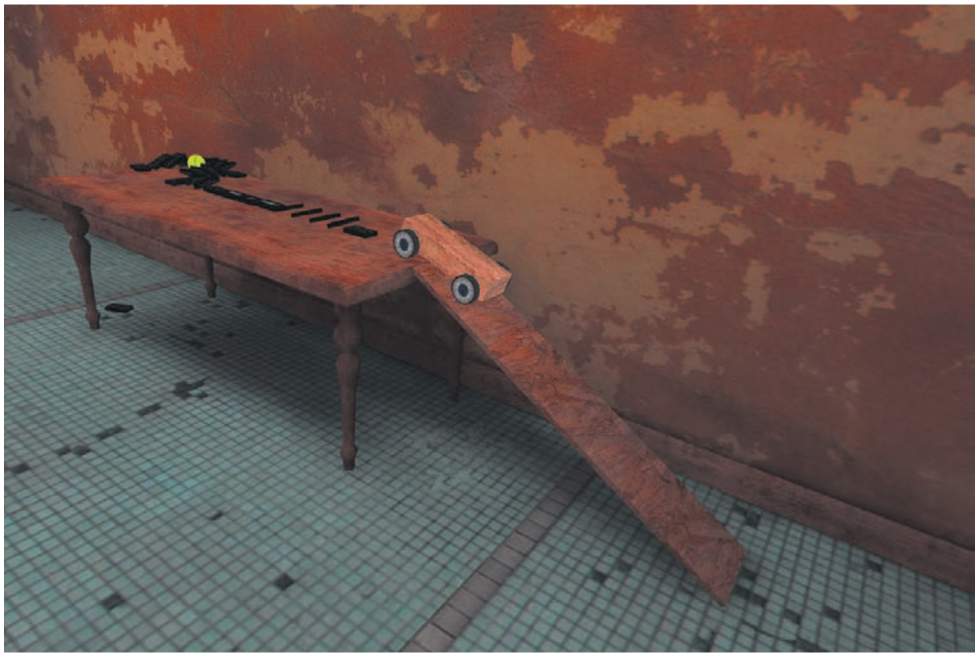


Figure 6.2 Stills from Christopher Plush's winning Bullet Rube Goldberg machine



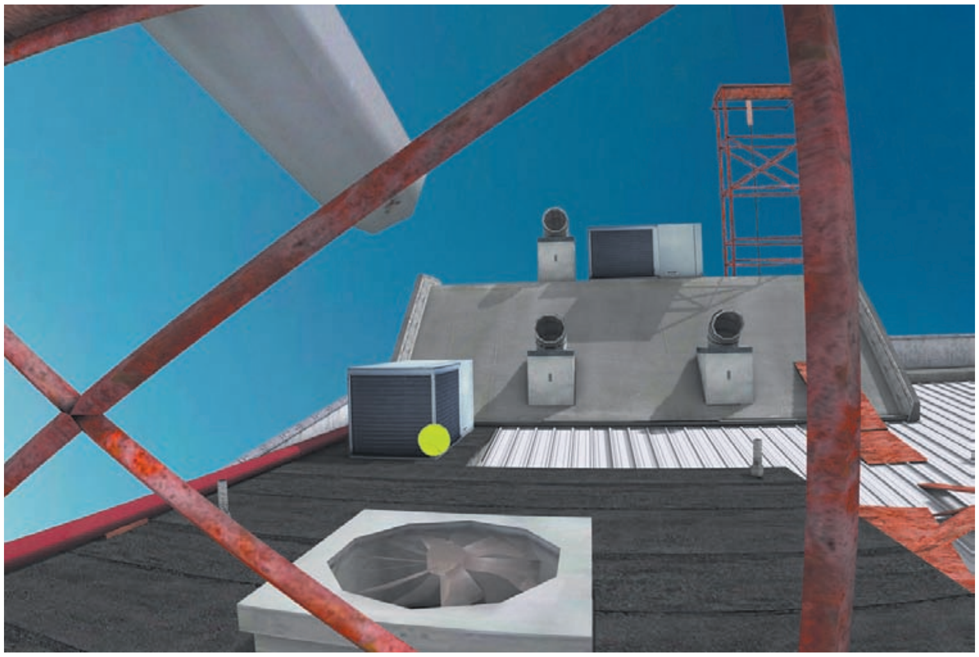
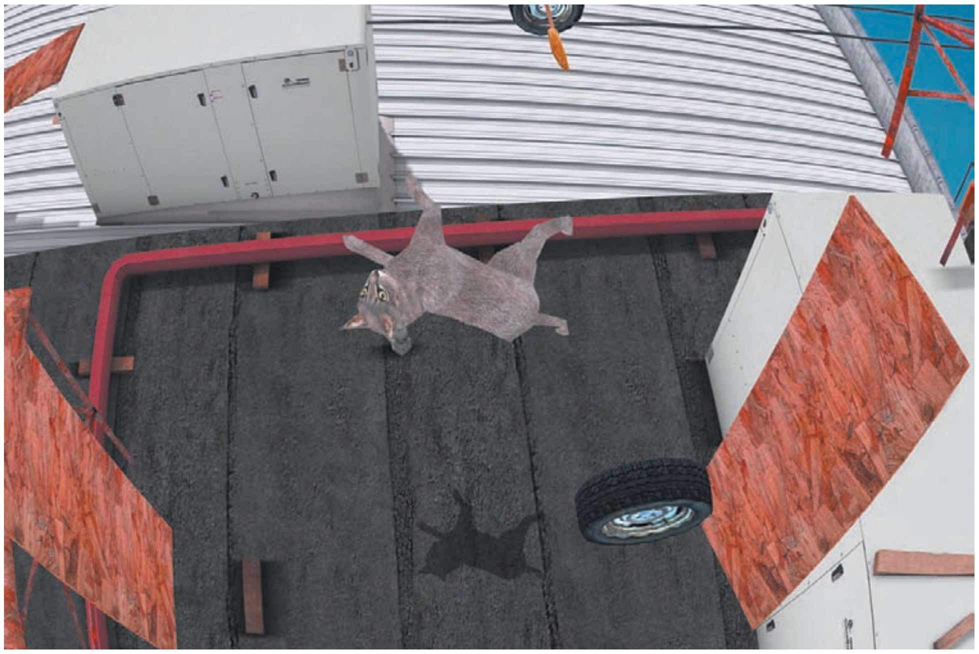


Figure 6.2 *(continued)*



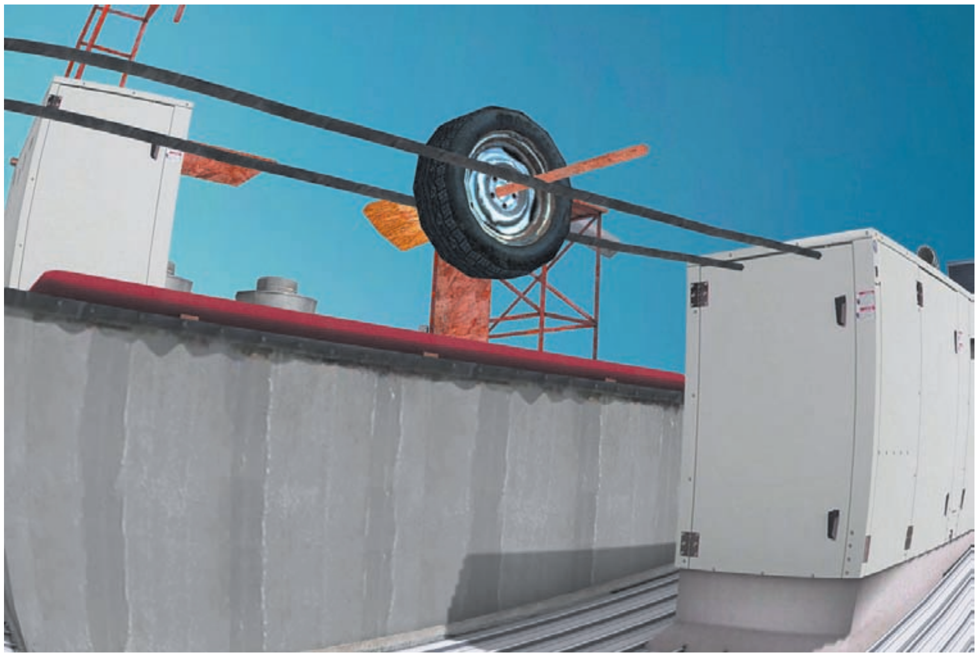


Figure 6.2 *(continued)*

Getting Started with BGE

This book is written primarily with animators in mind, so I'm going to assume that you've never touched BGE. For a lot of animators and illustrators, the little Pac-Manesque Logic Buttons icon in the corner of the Buttons area is ignored and slightly intimidating, and pressing P by accident can be a nasty surprise. If that sounds like you, then it's time to change that. If you do have experience with BGE, you might want to skim this section. The goal here is to get comfortable enough with BGE to make full use of its physics simulation functionality. I'm not going to tell you much about making actual games, although after you get accustomed to using BGE, you'll probably find that doing so quickly becomes intuitive.

In this chapter, I will also discuss an excellent new method of getting game physics into your animation environment: By using the Ctrl+Alt+Shift+P key combination, you can bypass many of the inconveniences of using BGE directly. However, in order to get the most out of that method, it is necessary to have some idea of what is going on in BGE.

To get started, fire up a session of Blender. In the top view (NUM7), do the following:

1. Add a plane and scale up as shown in Figure 6.3.

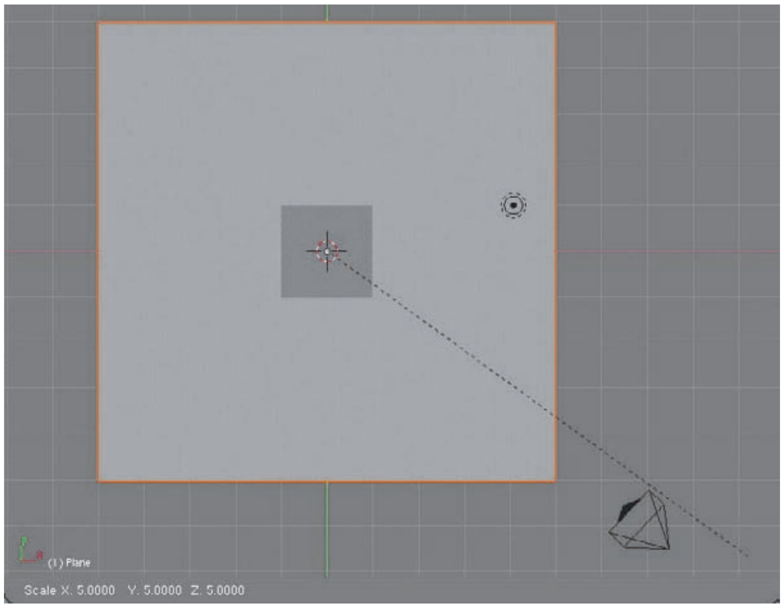
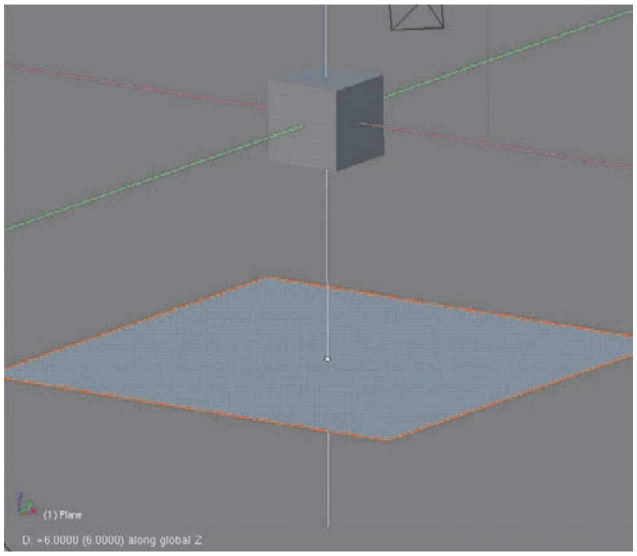


Figure 6.3 Add a plane and scale up.

2. Translate the plane downward along the Z axis, as shown in Figure 6.4.
3. With the cube selected, press F4 to enter the Logic Buttons context, or press the Logic Buttons icon in the Buttons window header.
4. In the Buttons area, click Actor, as shown in Figure 6.5. Set the parameters of that panel as shown in Figure 6.6. Select Dynamic and Rigid Body, and select Box from the Bounds drop-down menu. You have now activated the cube as a physical object in BGE.



D: -6.0000 (6.0000) along global Z

Figure 6.4
Translate downward.

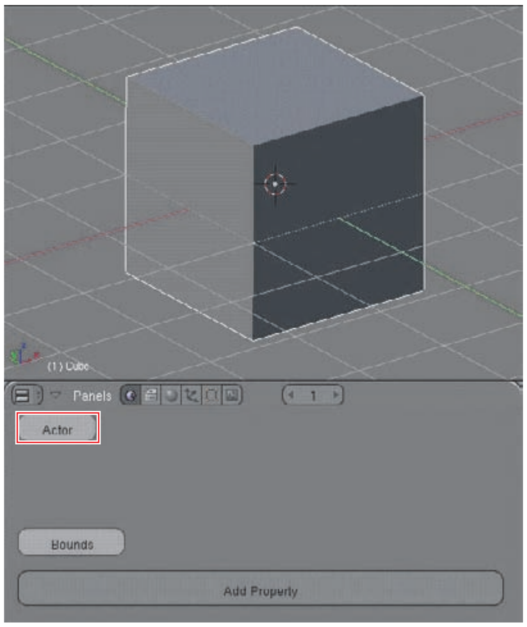
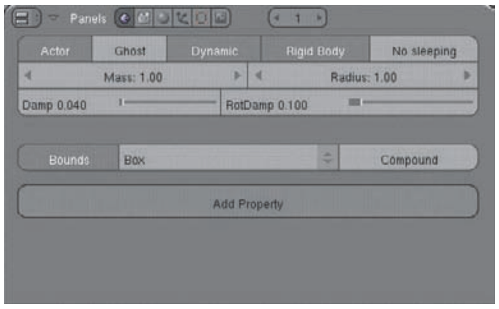


Figure 6.5
Enabling the cube as an actor in BGE



Actor | Ghost | Dynamic | Rigid Body | No sleeping

Mass: 1.00 | Radius: 1.00

Damp 0.040 | RotDamp 0.100

Bounds | Box | Compound

Figure 6.6
Actor parameters

5. Before entering Game Play mode, switch to Shaded view mode from the drop-down menu in the 3D viewport header, as shown in Figure 6.7. This isn't strictly necessary, but things will look much better in the game-play environment if you do this. After you've set this, press the P key to begin Game Play mode. You should see the cube fall and bounce against the plane, following a trajectory similar to what's shown in Figure 6.8. The angle you view this from will depend on the angle you're viewing the scene from when you press P. After the cube has come to rest, press Esc to leave Game Play mode.

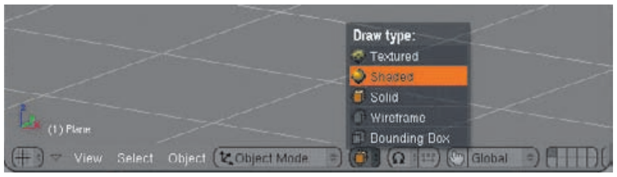


Figure 6.7 Switch to Shaded view mode.

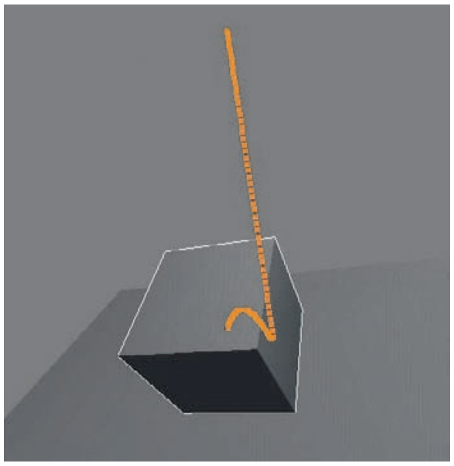


Figure 6.8 The cube bounces against the plane.

6. You can now begin setting up some basic game logic by using the buttons shown in Figure 6.9. Game logic is the main way to define what events or actions lead to others inside the game environment. Interactive controls are set up here. To set up the game logic, begin by adding a sensor. Click Add, and then select Keyboard, as shown in Figure 6.10.

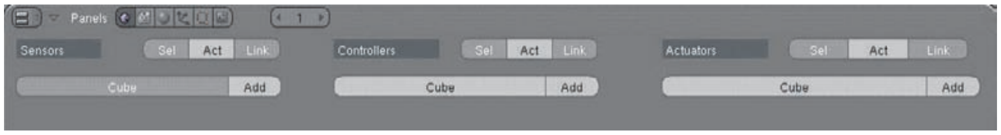


Figure 6.9 The Game Logic area

7. The game action you're setting up will be triggered by pressing the spacebar. To determine this, click the button indicated in Figure 6.11. When prompted to press a key, press the spacebar.

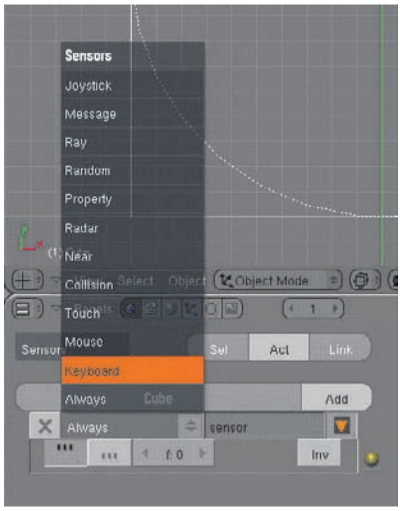


Figure 6.10 Setting a keyboard action sensor

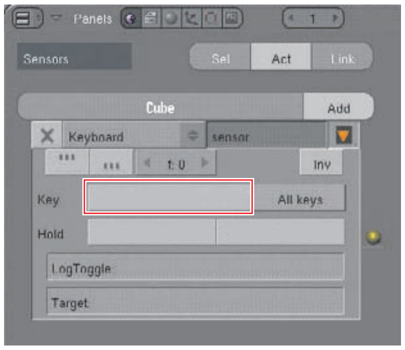


Figure 6.11 Press the button indicated, and then press the spacebar when prompted.

8. A sensor registers the event that will trigger an action. An actuator defines the resultant action. In order to connect these two things, you need to create a controller. Do this by clicking Add under the Controllers (middle) column. Link the sensor to the controller by clicking the LMB and dragging the mouse between the two logic connectors, as indicated in Figure 6.12.

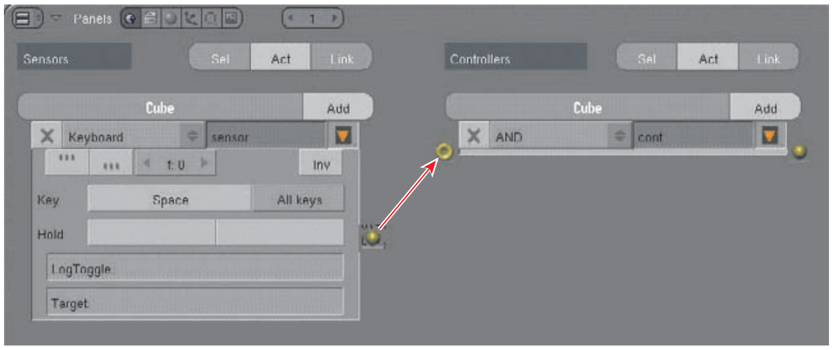


Figure 6.12 Connecting the sensor to the controller

9. Add an actuator by connecting the controller and the actuator in the same way you connected the sensor and the controller (see Figure 6.13). Set the value for the middle column of the dLoc row to **0.10**.

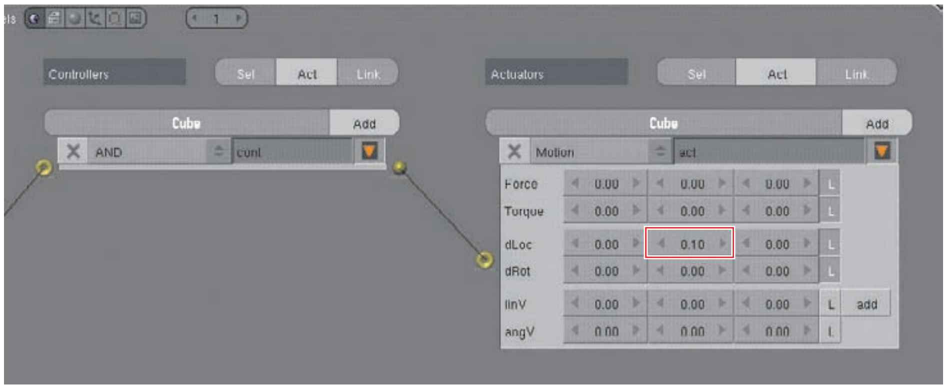


Figure 6.13 Adding an actuator

10. In the 3D viewport, press P again to enter Game Play mode. This time, try pressing the spacebar while in Game Play mode. You should now have some control over the movement of the cube. Giving the spacebar a good press will send the cube flying off the plane as in Figure 6.14. Press Esc to get out of this mode.

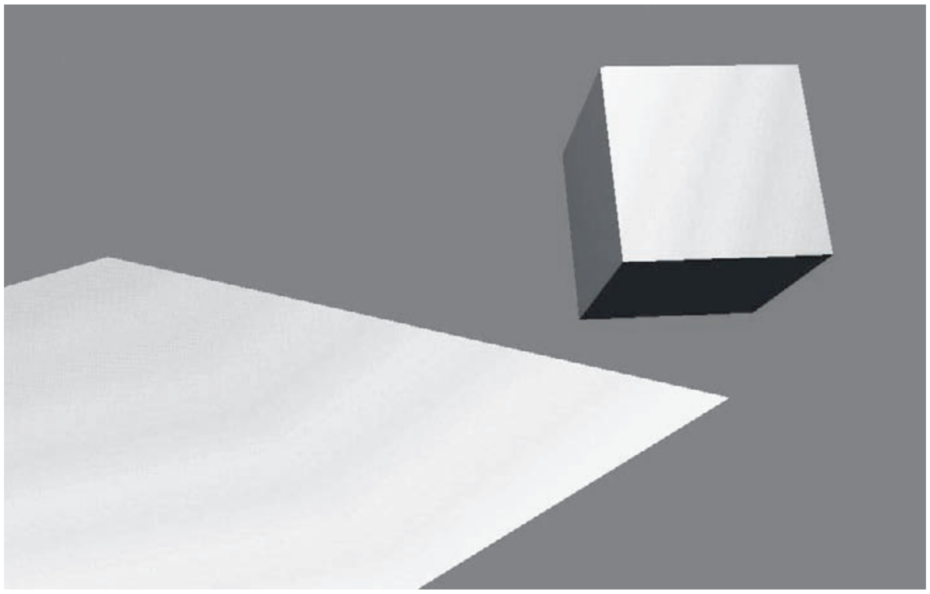
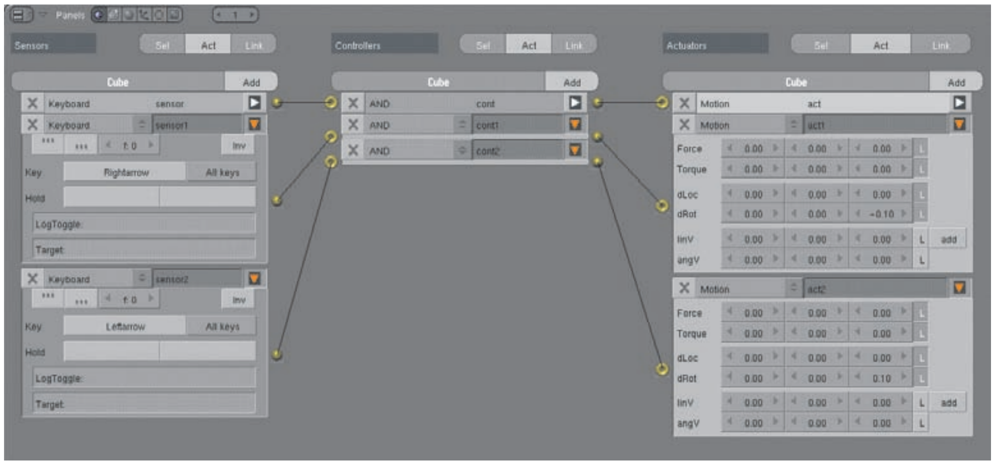


Figure 6.14 Enjoying an exciting game of "Push the Cube into the Abyss"

11. You can minimize and maximize the view of logic blocks by clicking the triangle in the upper-right corner of the logic block. In Figure 6.15, the blocks you just created are minimized, and two new sensors, two new controllers, and two new actuators have been added. The sensors are for the right-arrow and left-arrow keys, and the actuators represent right and left rotation around the local Z axis, with –0.10 and 0.10 values, respectively, in the third column of the dRot rows. Set these logic blocks up as they are shown in Figure 6.15.



Sensors (Cube)
Keyboard sensor
Keyboard sensor1: f: 0, Inv; Key: Rightarrow, All keys; Hold; LogToggle:; Target:
Keyboard sensor2: f: 0, Inv; Key: Leftarrow, All keys; Hold; LogToggle:; Target:

Controllers (Cube)
AND cont
AND cont1
AND cont2

Actuators (Cube)
Motion act
Motion act1

Force	0.00	0.00	0.00	L
Torque	0.00	0.00	0.00	L
dLoc	0.00	0.00	0.00	L
dRot	0.00	0.00	-0.10	L
linV	0.00	0.00	0.00	L add
angV	0.00	0.00	0.00	L

Motion act2

Force	0.00	0.00	0.00	L
Torque	0.00	0.00	0.00	L
dLoc	0.00	0.00	0.00	L
dRot	0.00	0.00	0.10	L
linV	0.00	0.00	0.00	L add
angV	0.00	0.00	0.00	L

Figure 6.15 More controls

12. Add an Icosphere next to the cube, as shown in Figure 6.16. Using the Logic buttons, set the sphere as an actor with the same parameters as you set for the cube, except with Sphere selected under Bounds, as shown in Figure 6.17.

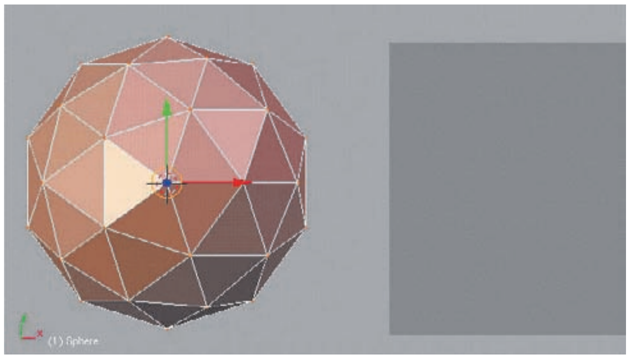
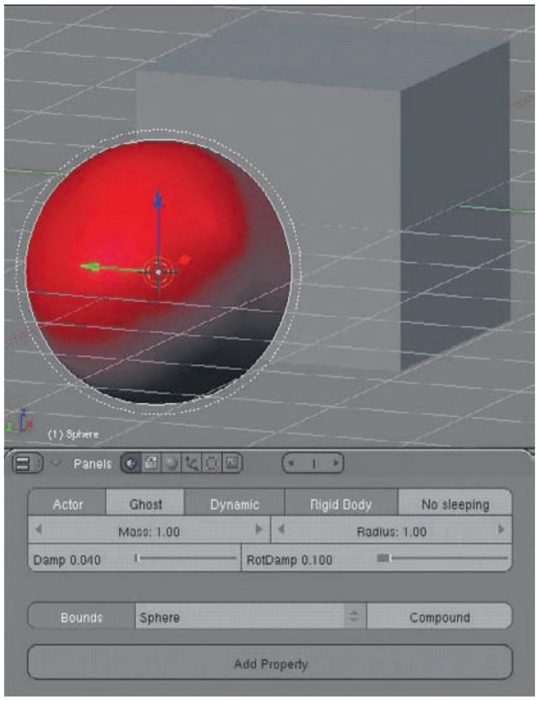


Figure 6.16 Add an Icosphere.

13. Press P again. Notice how the sphere and the cube interact. You can now drive the cube around by rotating right and left with the arrow keys and moving forward with the spacebar. Try pushing the sphere off the plane as in Figure 6.18. Press Esc to get out of this. Experiment with adding more objects with and without Actor enabled, and playing around with the angle and shape of the plane.

You now know most of what you need to know to get started with BGE. As you can see, setting up interactive logic is not too complicated. For the purposes of this book, minimal interactive logic is necessary, so this is all the detail I'm going to cover here. Much of the time, no interaction will be necessary—you'll just set up a physical situation and let the simulator take over. However, sometimes using interactive triggers can be very handy to get the effect you want.



Mass: 1.00
Radius: 1.00
Damp 0.040
RotDamp 0.100
Bounds Sphere

Figure 6.17 Parameters for the sphere

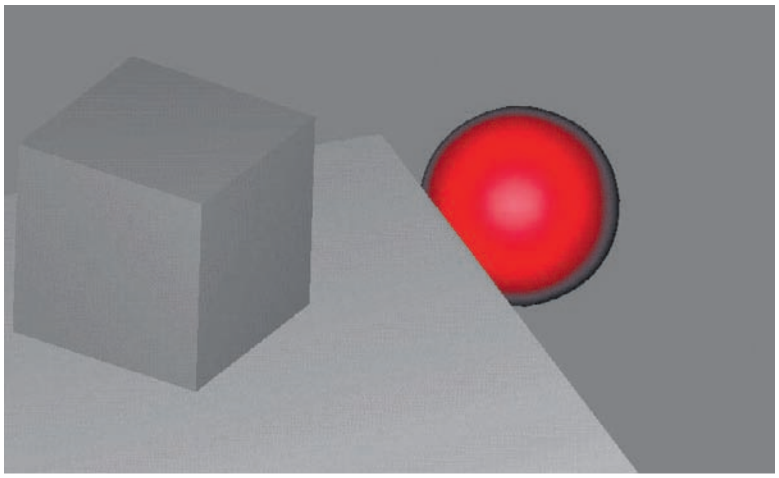


Figure 6.18 An even more exciting game of "Push the Sphere into the Abyss with the Cube"

Using Ipos and Actions in BGE

For people designing games, it's important to be able to have animated objects and armature motions accessible to the game logic. Mostly, animators are best off doing sophisticated animation in the ordinary Blender animation environment—animation in the game engine is far more restrictive than it is in Blender itself. Nevertheless, there are times when you will want to simulate physical reactions to hand-keyed animation. To do this with BGE, you use Ipo objects or actions in the game-play environment, as follows:

1. Start a fresh session of Blender and key the location and rotation of the default cube at frame 1.
2. Advance to frame 31 (click the Up button three times), move the cube up along the Z axis, and rotate it slightly to the left. Key this location and rotation as shown in Figure 6.19. Note that the figure shows the keyframe view in both the Ipo Editor and the 3D window. You can toggle the keyframe view by pressing the K key. The keyframe view in the 3D window is convenient for adjusting actual keyed objects' positions.

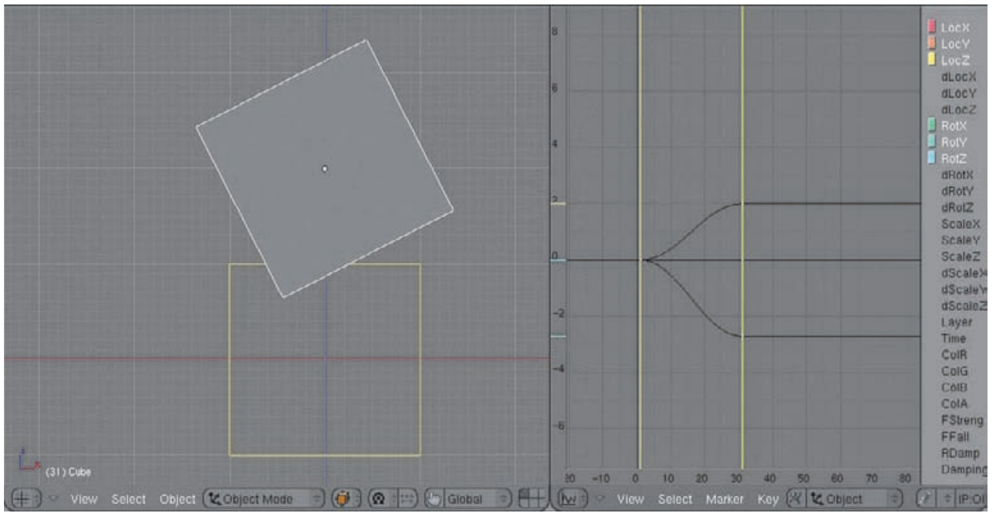


Figure 6.19 Keyed location and rotation at frame 31

3. In the Logic Buttons area, with the cube selected, set up a sensor, controller, and Ipo actuator as in Figure 6.20. In the setup, I have the sensor selected as Always, and the Ipo actuator type is Play. The Start and End frames of the Ipo to be played, in the Sta and End fields, are 0 and 30. When you press P, the cube will go through the motion of the active Ipo from frame 0 to 30 repeatedly. If you use a different sensor type, such as a keyboard action sensor, the Ipo motion will be triggered when you press a key. Also, if you change the Start and End frames of the Ipo actuator, a different (potentially longer or shorter) section of the Ipo will be played. Other Ipo actuator types can also be selected besides Play. (See if you can guess what the Ping Pong and Flipper types do.)

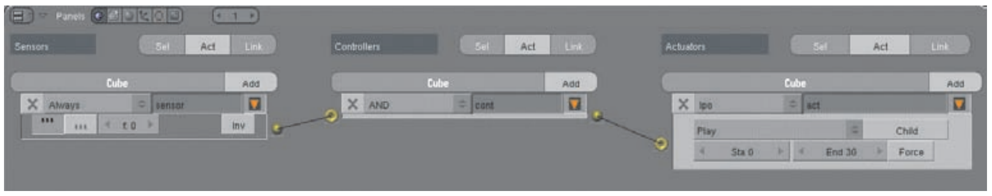
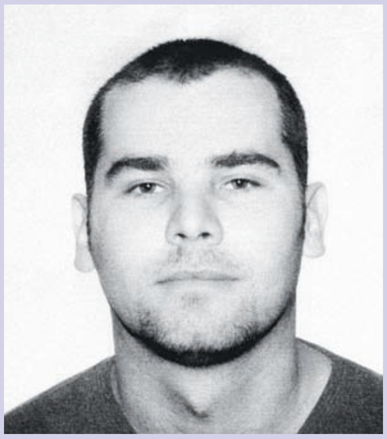


Figure 6.20 Sensor, controller, and Ipo actuator setup

Dale Best

Dale Best is the design director of ClubSilo, the driving game prototype produced by Luma Studios in South Africa. Although Luma has used Blender in various capacities since 2001, Dale himself has been using it only since March 2006. "The initial learning of the interface took about a day, and then it was full steam ahead for three months," he says. "During that time I hardly used another 3D app, and found (Blender) very intuitive to use." For creating ClubSilo, Dale worked alongside programmer Herman Tulleken. Luma Studios has also used Blender as an animation tool for television broadcasts. Check out their website at www.luma.co.za.



4. Experiment with combining this setup with the setup from the previous section. Try to set up a flipper-style controller to swing at a falling cube, knocking it into space before it hits the plane. Note how the cube interacts with Rigid Body objects such as the sphere from the previous section's example.

Rigid Body Simulation and Ipos

You now know how to set up basic physics simulations in BGE. But if you're reading this book, chances are you're most interested in using these simulations in your rendered animations. What you want is not real-time simulation, but Ipo curves. There are two ways to get these Ipos into your animation environment: the hard way and the easy way.

The most common way up until now has been to do it the hard way, going directly into BGE and recording the game physics to Ipos. This method had various problems, mostly to do with differences between how Blender handles Ipos and how BGE handles real-time animation. There is now a better way to get the benefits of Bullet Physics in your animation without going into Game Play mode at all, by simply using the Ctrl+Alt+Shift+P key combination. I'm going to describe both approaches. For people with an interest in the game engine, there's a lot to be learned by going through the steps of the former, but the latter is the approach I recommend for actual use. If you're impatient to get your rigid body simulations underway, you can skip forward to the section on Ctrl+Alt+Shift+P for a complete demonstration of how to use that method.

Baking Game Ipos

You can save the movements of Dynamic objects in BGE by selecting Record Game Physics To IPO from the Game menu in the User Preferences/Information header at the top of your Blender workspace, as shown in Figure 6.21. After you exit from Game Play mode by pressing Esc, the Ipos will be associated with whatever objects were enabled as Dynamic objects in the physics simulation. You can press Alt+A or the Play button on the Timeline and play the animations like any other animation, and you can render it as you would anything else in Blender.

The animated shot shown in Figure 6.22 was created in this way. Six cubes were created and set up vertically in space along the lines of the first image in the sequence. Each cube was enabled as an actor and made Dynamic. Rigid Body was selected to enable the blocks to roll and tumble. Another cube was scaled to the shape of a book, and under that a shadow-only plane was added for the blocks to land on. After baking the Ipo from BGE, some image textures were added to the objects, a sky map was applied, and the animation was rendered in the ordinary way by using ray shadows.

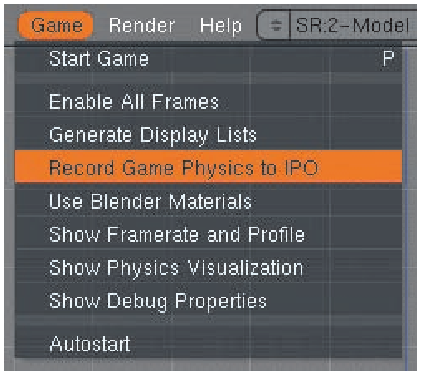


Figure 6.21
Record Game Physics
To IPO menu option

Note: After you've selected Record Game Physics To IPO, a check mark will appear beside the option, and it will remain active until you deselect it. This means that every time you press P to enter Game Play mode, the game physics Ipos will be overwritten, so be careful. If you want to run a simulation and record it to a different Ipo than a previous simulation, select the object, and in the Ipo Editor window click the X beside the drop-down menu where the Ipo you want to save is shown. This disconnects the Ipo from the object, so that a new game physics Ipo object can be written without overwriting what you already have.

Frame Rate and Simulation Speed

If you try this with the first example of this chapter or with an example of your own, you will surely notice that when you press Alt+A or render your animation to a movie, the resulting simulation is in slow motion. This occurs because the game engine uses a default frame rate of 60 frames per second (fps). This is desirable for real-time graphics for several reasons, but it is much faster than is necessary for animated movies, which are usually somewhere around 25 fps (the default frame rate in Blender). This means that when you record Ipos from the game engine, the resulting Ipos will be normal speed when played back at 60 fps, but will be about half speed when played back at 25 fps. Furthermore, if you use hand-keyed Ipos in the animation in the form of Ipo actuators, as described in the second example in this chapter, you'll find that the resulting baked physics Ipos do not sync up properly with the hand-keyed Ipos in the animation because of mismatched frame rates.

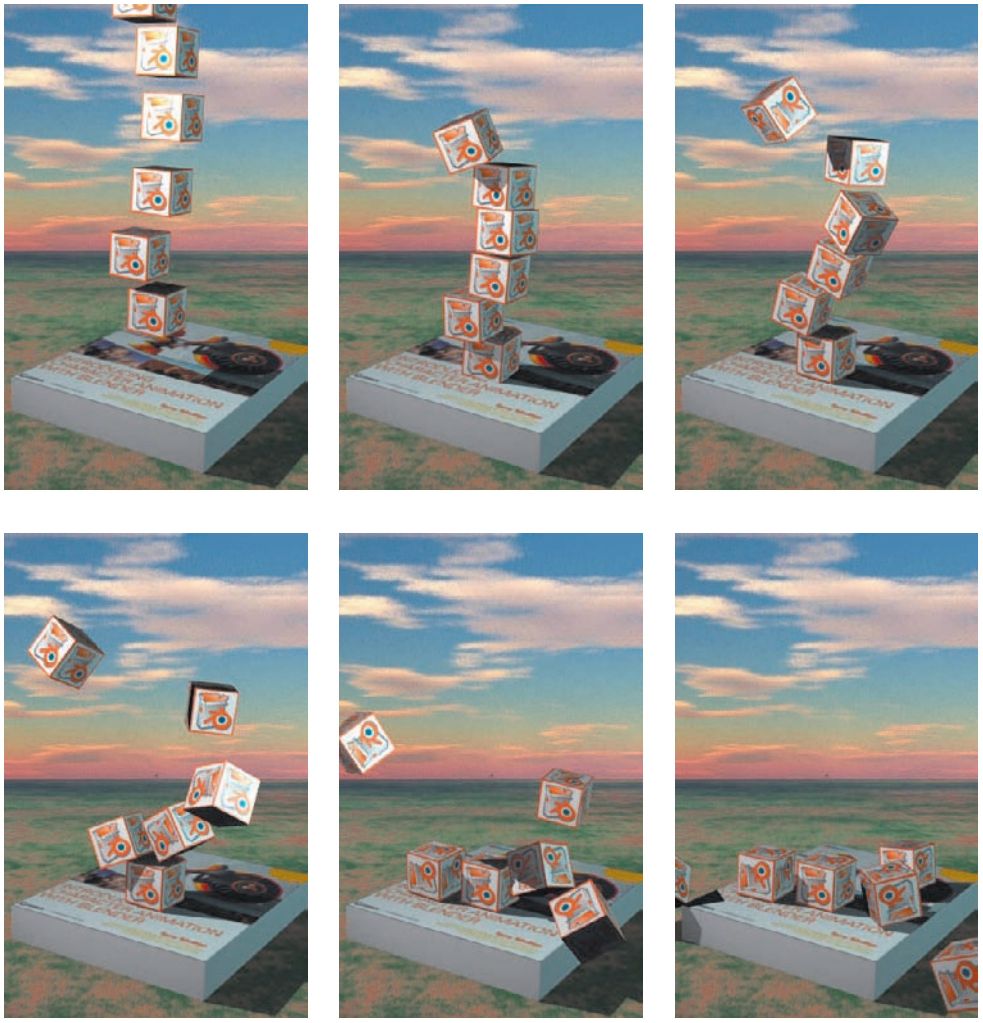


Figure 6.22 Rigid body blocks in a rendered animation

To correct this in the game engine, you need to reset the BGE frame rate to match the frame rate of the animation you want to create, such as 25 fps. In order to do this, it is necessary to take a brief detour into Python scripting for BGE.

To see how scripts are called in BGE, repeat the first five steps of the first example in this chapter to set up a Dynamic Actor object falling onto a plane. When you press P, the cube falls onto the plane, and if you select Record Game Physics To IPO, you will see that the playback is in slow motion. To set up the Python script, follow these steps:

1. Open a Text Editor window by creating a new work area and selecting the Window Type menu item, as shown in Figure 6.23. From the header menu of the Text Editor window, select Text (see Figure 6.24). This is the name of the default text file in the editor.

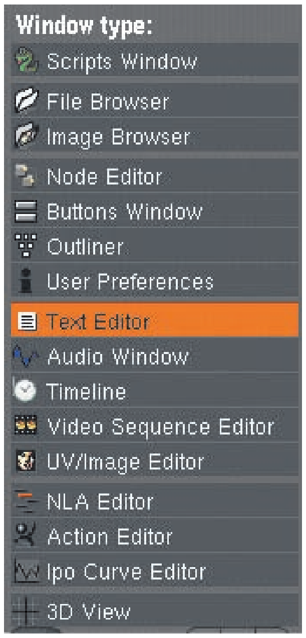


Figure 6.23 Text Editor Window Type menu item

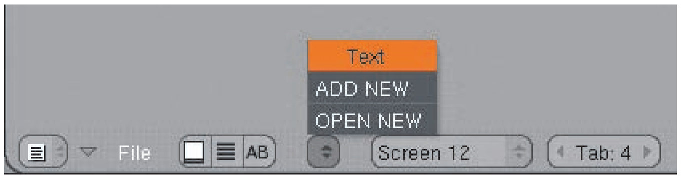


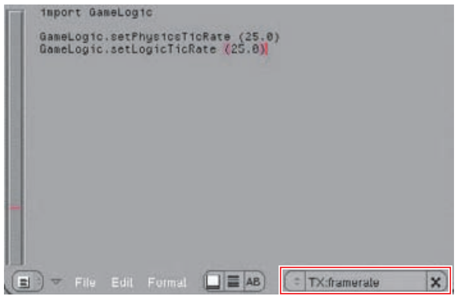
Figure 6.24 Select Text from the header drop-down menu.

2. In the text area, type the following code, exactly as it is shown in Figure 6.25:

```
import GameLogic

GameLogic.setPhysicsTicRate (25.0)
GameLogic.setLogicTicRate (25.0)
```

Also, change the name of the script from the default Text to **framerate**, as shown in the highlighted area of the figure.



```
import GameLogic

GameLogic.setPhysicsTicRate (25.0)
GameLogic.setLogicTicRate (25.0)
```

Figure 6.25 The framerate script

3. Now you need to set up the logic blocks to call the code from within the game engine. To do this, you will associate an Always sensor and a Python controller with an object in the game. The plane is a good choice for this. Select the plane and set up the logic as shown in Figure 6.26. Type **framerate** into the Script field, as shown in Figure 6.27.

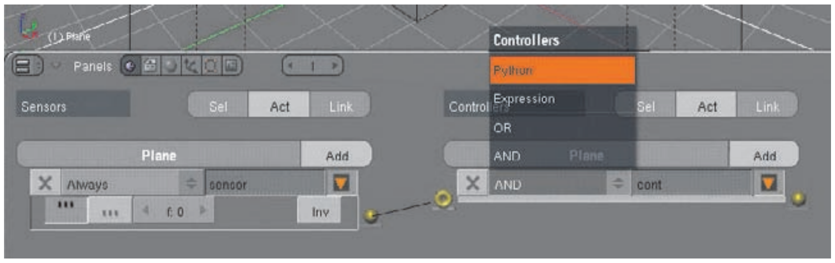


Figure 6.26 Python controllers

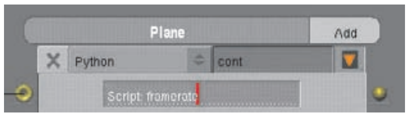


Figure 6.27 Enter the script name.

4. Make sure BGE is set to record game physics to Ipos by using the menu selection shown in Figure 6.21. Press P to run the simulation. The frame rate will be 25 fps, just as you specified in the script. Try changing the value in the script to see how other values change the speed of the resulting simulation. Higher values such as the default value of 60 will result in slower motion in the final animation.

Note: Remember, if you duplicate an object that has Ipos associated with it, the new object will have the same Ipos associated with it. In the case of a location Ipo, this means that no matter where you place the new object, it will jump back to where the Ipo places it, which will be the same place as the original object it was duplicated from. To make the new object act independently of the old object, it is necessary to disconnect the Ipo from the new object, so you can key a fresh Ipo to the object. It is also possible to make Blender create a new Ipo for duplicated objects. In the Edit Methods section of the User Preferences window, select Ipo from the Duplicate With Object options. In this case, a new Ipo will be created when you duplicate the object. However, the new Ipo will still be an exact copy of the original, so your new object will still jump to follow the original object unless you alter or delete the Ipo keys.

Ctrl+Alt+Shift+P

You can use the Ctrl+Alt+Shift+P key combination to record physics simulations straight from Bullet without entering Game Play mode at all. This method also enables you to bypass a number of issues relating to how BGE works, such as the mismatched frame rates in the previous section. In this method, you do all of your animation as you ordinarily would, keying values in Blender. You should not add

Ipo or Action actuators, and it is not necessary to use Python controllers. In fact, no logic blocks of any kind are used. The only thing you use in the Logic Buttons area is the Actor tab, where you set the parameters for Rigid Body dynamic objects.

In this section, you'll set up a complete rigid body simulation from scratch. The completed simulation can be found on the CD that accompanies this book in the file `ballandcubes.blend`.

Follow these steps to put the elements in place and run the simulation:

1. Set up a scene as shown in Figure 6.28, with the default cube in its original position. Add the following objects at the original cursor location, and move them along the Z axis while holding down the Ctrl key:

- A second cube located 6 Blender Units (BUs) under the first on the Z axis
- A floor plane scaled up to about 14 times its original size and located 8 BUs beneath the first cube on the Z axis
- An Icosphere located 3 BUs above the cube

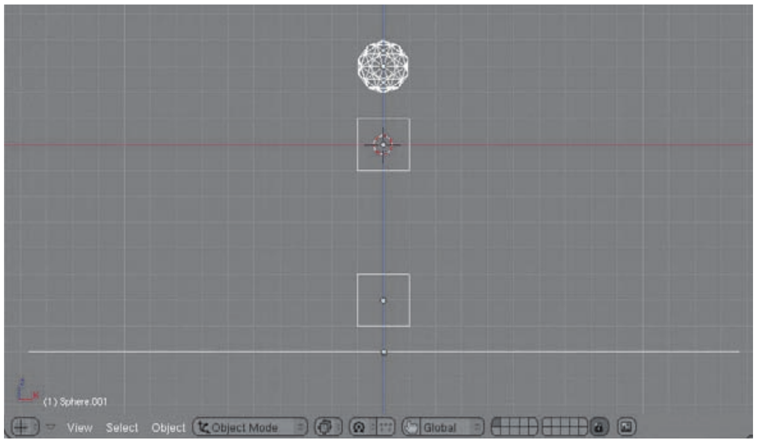


Figure 6.28 Setting up the rigid body scene

2. Add an armature at the original cube's location, as in Figure 6.29. In Object mode, translate the Armature object 3 BUs down the Z axis, as shown in Figure 6.30. In Edit mode, translate the tip of the armature downward to the middle of the second cube, as shown in Figure 6.31.

3. Bone-parent the cube to the bone. To do this, select the armature in Object mode and change to Pose mode. Select the cube (you will return to Object mode automatically). Hold down the Shift key and select the armature so that both the Cube object and the armature are selected in Pose mode, as shown in Figure 6.32. Press Ctrl+P and select Bone.

4. Key the motion of the topmost cube and the plane. The topmost cube will rotate counterclockwise. Key the rotation at frame 1, and then go to frame 11, rotate the cube to an angle of –45 degrees around the Y axis, and key the rotation at that frame. Choose Curve > Extend Mode > Extrapolation for the Ipo curve. For the plane, key its rotation at frame 61, and then advance to frame 81, rotate the plane around the Y axis 25 degrees, and key the rotation. Leave the curve's Extend mode at Constant.

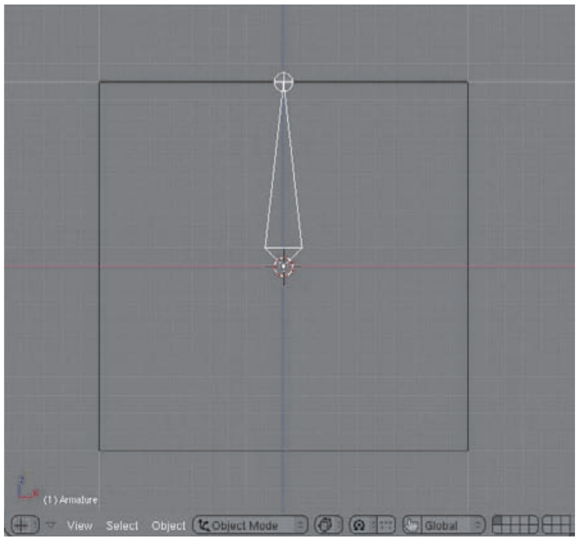


Figure 6.29
Add an armature.

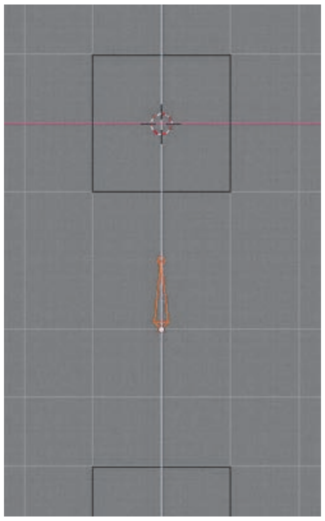


Figure 6.30
Translate the armature downward.

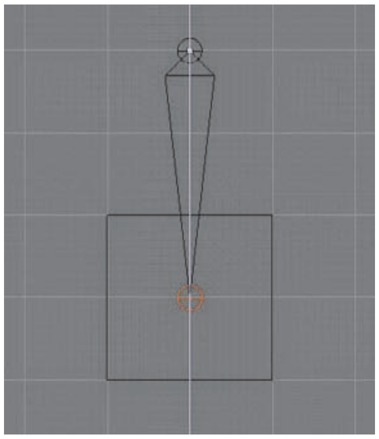


Figure 6.31
Translate the tip in Edit mode.

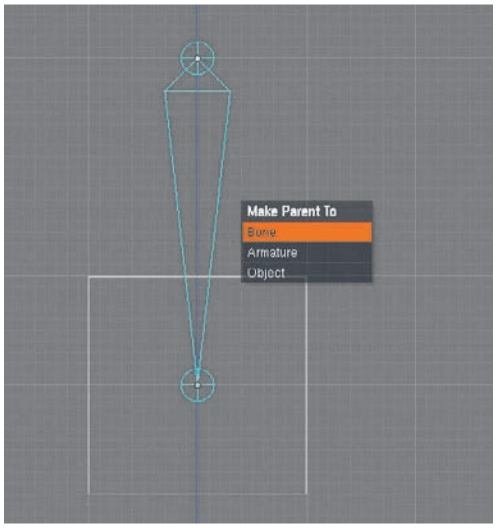


Figure 6.32 Bone-parent the cube.

5. Select the Icosphere. Go into the Logic buttons and make the sphere an actor with Dynamic and Rigid Body selected, as shown in Figure 6.33. Under Bounds, select Sphere. Next, select each of the Cube objects in the 3D viewport, go into the Logic buttons, and press the Bounds button to select Box. You do not need to make the Cube objects actors. I've also added a level 2 Subsurf modifier to the sphere and used Set Smooth. This is not necessary, but it will make your scene look a bit nicer.

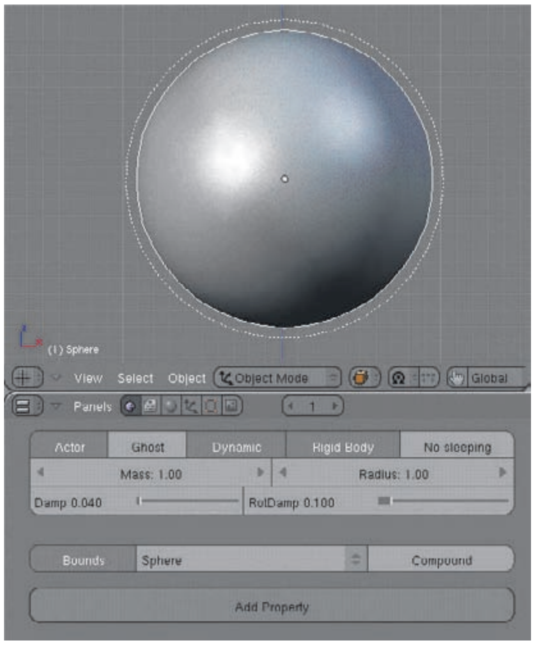


Figure 6.33 Making the sphere a Rigid Body object

6. Now's the big moment. Press Ctrl+Alt+Shift+P to record the physics simulation to Ipos. You won't see anything happen, but your system's standby icon may be displayed for a few moments. After this finishes, you can press Alt+A to run the animation. You'll see that it looks something like Figure 6.34.

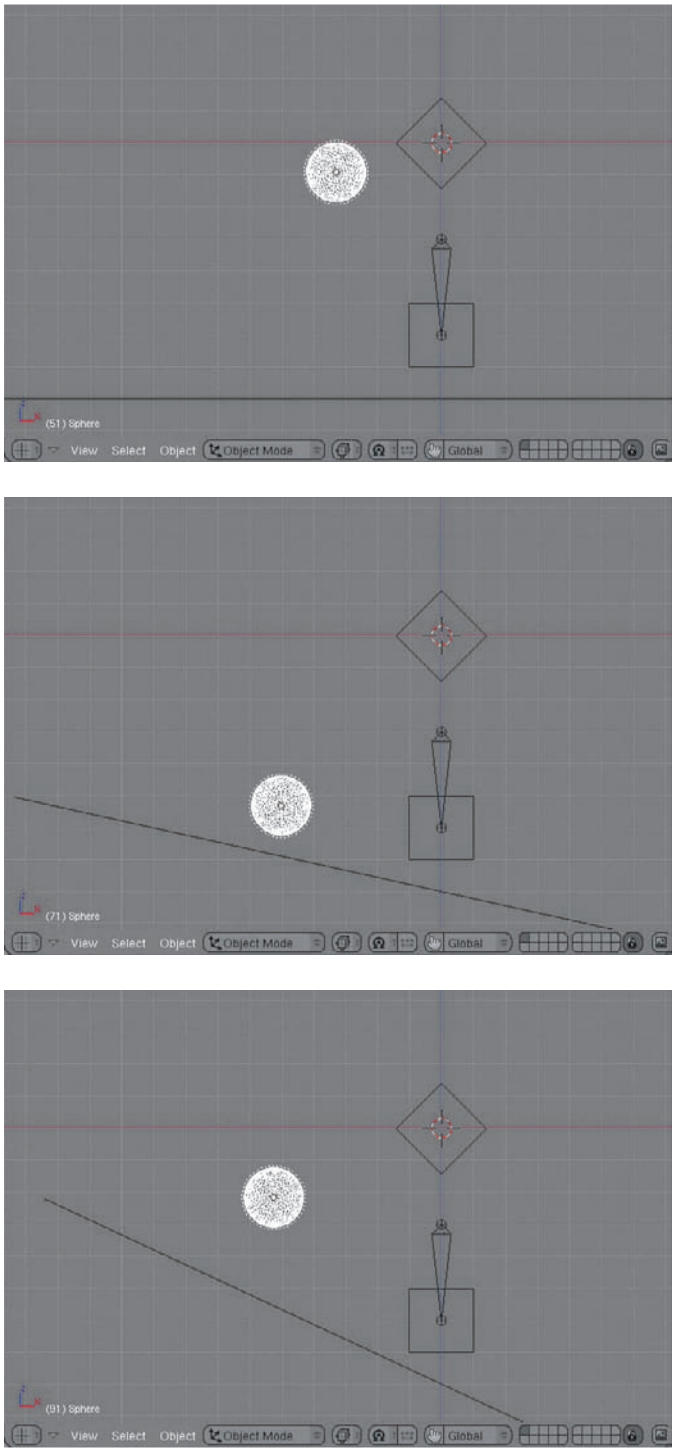

(51) Sphere
View Select Object Object Mode Global
(71) Sphere
View Select Object Object Mode Global
(91) Sphere
View Select Object Object Mode Global

Figure 6.34 The animated simulation

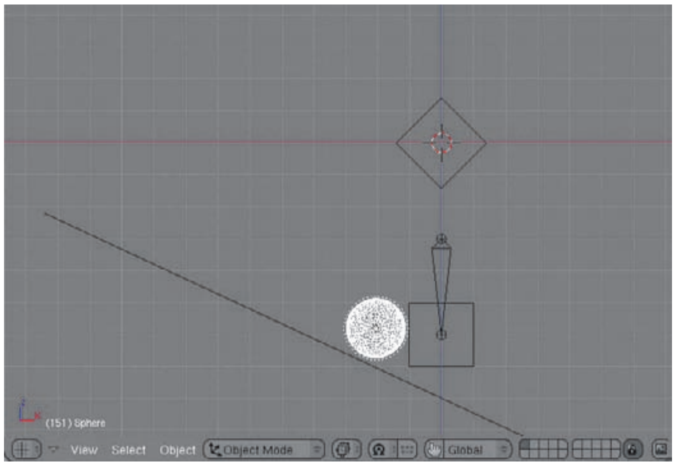


Figure 6.34 *(continued)*

7. As you can see, the sphere comes to rest against the second cube at about frame 131. Knowing this, it is possible to key the motion of the armature so that the second cube will "kick" the sphere at exactly the right moment. To do that, key the beginning and ending of the kicking movement about 20 frames before and after the point where the cube should impact the sphere, around frames 111 and 151, as shown in Figure 6.35.

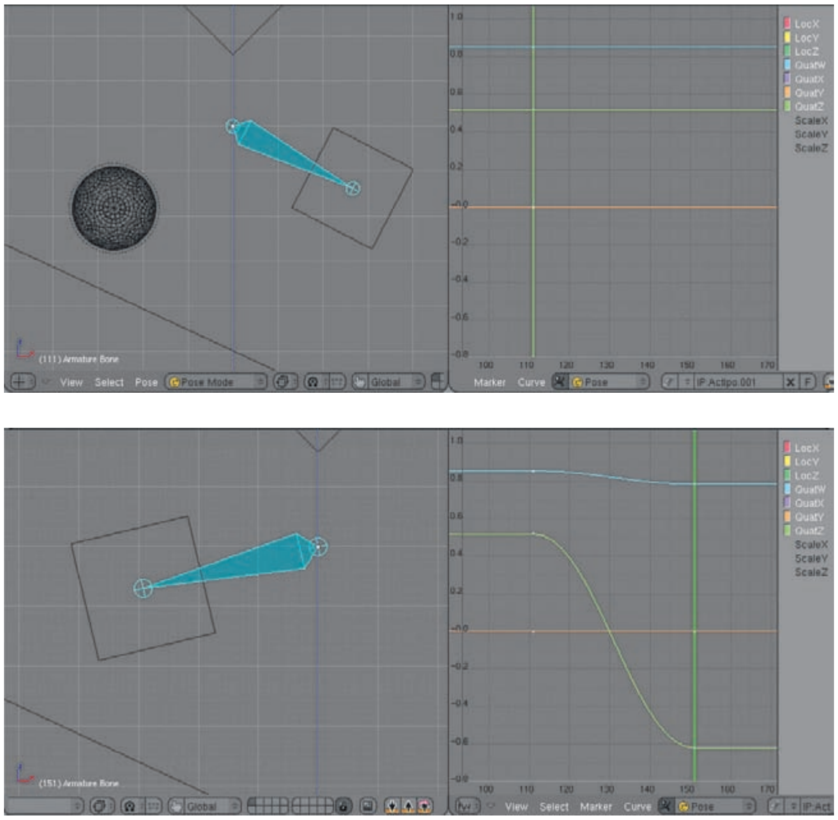


Figure 6.35 Keying the "kick"

8. Press Ctrl+Alt+Shift+P again, and run the animation with Alt+A again. The first part of the simulation will be the same, but instead of coming to rest, the sphere will be kicked off the plane as in Figure 6.36. Doing this enables you to skip entering the BGE and adjusting the frame rate with scripts.

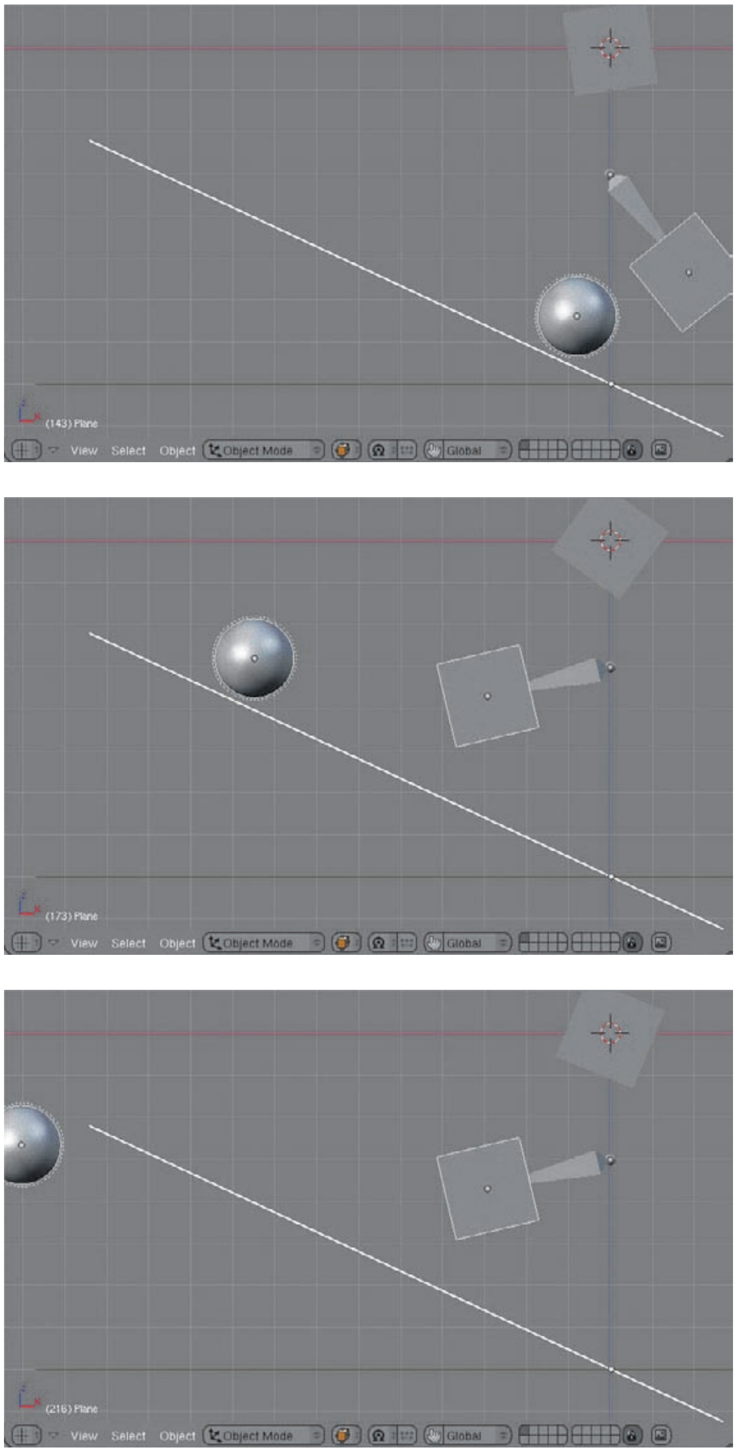

(143) Plane
View Select Object Object Mode Global
(173) Plane
View Select Object Object Mode Global
(216) Plane
View Select Object Object Mode Global

Figure 6.36 The sphere kicked off the plane

Actor Parameters, Boundaries, and Hull Types

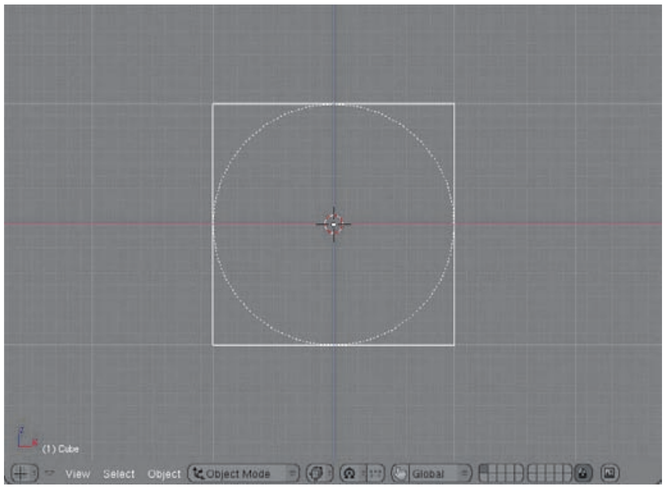
There are several qualities that objects need to have in order to behave naturally in physics simulations. You've already come across several of these as you've been setting up objects for Bullet simulations. Now I'll discuss what the terms mean in a little more detail. The top row of the rigid body physics settings in the Logic Buttons context contains the following five options:

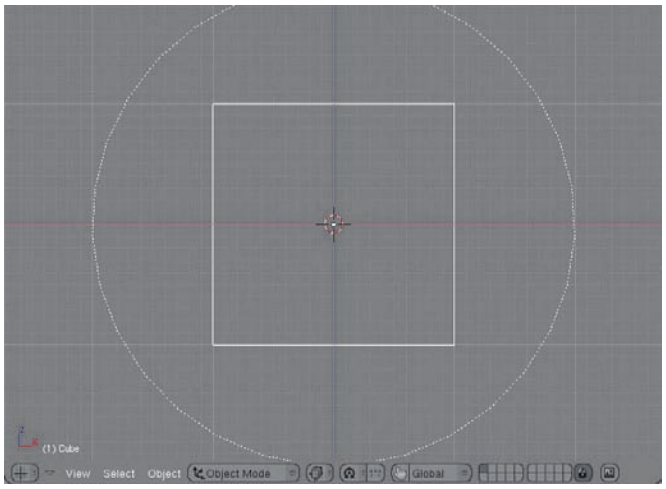
> **Actor** enables the rest of the buttons so that physics parameters can be set for an object. If this is not selected, the object will be evaluated only as an obstacle. You can still set the collision Bounds for a nonactor.
>
> **Ghost** enables in-game objects to *not* have physical interactions with others, so that they can be passed through like a ghost. Because ordinary animation does not have physics calculated automatically, this option is not usually needed in nongame simulations.
>
> **Dynamic** enables the object to be moved by physical forces. Only objects with this option selected will have Ipos generated for them during a simulation.
>
> **Rigid Body** enables spring-based collision behavior and rolling. This is a necessary option for convincing physical simulation of multiobject interaction.
>
> **No Sleeping** disables BGE's default behavior of turning off physics on objects that have not been moved or touched for some time. If you find that an object is becoming unresponsive over the course of a long simulation, select this option. Don't select it if it is not necessary.

In the next row in the panel are the Mass and Radius parameters. The Mass value determines the mass of the object relative to other objects in the simulation. When an object with a greater mass collides with an object with less mass, the trajectory of the object with less mass is more affected. As in the real world, objects with greater mass have greater inertia. This means that they require more force to get moving and also that they are more difficult to slow down when they are moving, making them less subject to damping in the physics engine. Due to current limitations in the Bullet physics engine (and in fact all physics engines), simulations become unstable when objects with extremely different masses interact in certain ways. Objects in constraint chains should be within a few kilograms of each other, and very heavy objects should not be placed on top of very light objects. These kinds of interactions between objects with extreme differences in mass should probably be keyed by hand or simulated in another way, such as with particles.

The Radius parameter indicates the size of the sphere collision bounds. When you enable an object as a dynamic actor, the sphere is displayed as a dotted line representing the default collision boundary for the object (as shown in Figure 6.37). This is accurate only if your object is a sphere. If not, you will need to select a more-appropriate collision boundary type, and Radius is not pertinent (although the dotted circle will continue to display).

In the area below the Mass and Radius fields are the damping fields. Both Damp and RotDamp remove motion energy from the object over time—Damp from translation, and RotDamp from rotation. When these values are zero, no damping occurs and the movement of the object is unimpeded. With higher values, movement and rotation is diminished to mimic the effects of air friction.





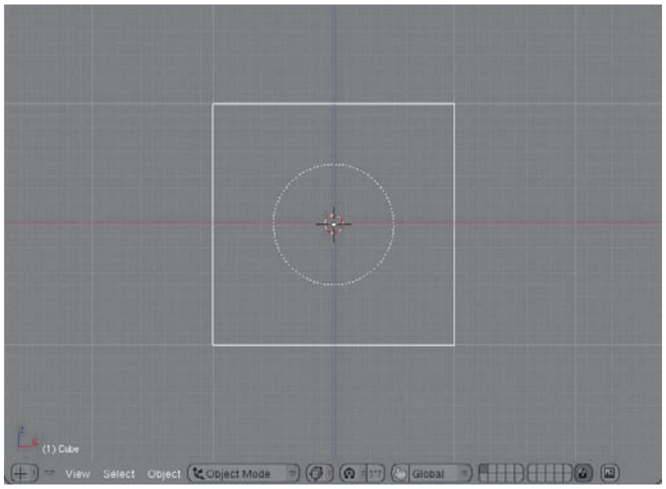


Figure 6.37 Radius values of 1.0 (default), 2.0, and 0.5

Collision Boundary Types

By default, a Rigid Body object behaves like a sphere, rolling on the boundary defined by its Radius value. For most shapes, this is not what you want. In order to have Bullet calculate the collision boundaries accurately for the object, you must select a collision boundary type by clicking Bounds and selecting one of the following options from the drop-down menu:

Box calculates the collision boundary as a cube-shaped box. If your object is a cube shape, this is the fastest and most accurate option.

Sphere calculates the collision boundary as a sphere. If your object is a sphere, this is the fastest and most accurate option.

Cylinder calculates the collision boundary as a cylinder. If your object is a cylinder, use this.

Cone calculates the collision boundary as a cone. If your object is a cone, use this.

Convex Hull Polytope calculates a "convex hull" around the shape of the object, yielding an accurate collision boundary for many complex shapes. There are two important qualities to note about the Convex Hull Polytope. The first is that it can represent only convex shapes; concavities, such as the inside of a cup or a bell, are not accurately simulated. The second point is that it must be a low poly mesh. Meshes of more than about 100 vertices should not be used as Convex Hull Polytopes. Ideally, Convex Hull Polytope meshes should be between 4 and 30 vertices.

Static Triangle Mesh can simulate highly accurate, high-poly shapes, including concavities. There are also two notable qualities of Static Triangle Meshes. The first is that they can be used only for static (nonmoving) objects (that is, the object should not move on its own; a static object can move as the child of a dynamic object). Static Triangle Mesh bounded objects will not receive Ipos from the physics simulation. The second notable quality is that two Static Triangle Meshes will not collide with each other. A Static Triangle Mesh will collide only with dynamic objects.

It is worth noting that there is no visual feedback for these boundary types. Regardless of which you choose, the dotted sphere will be shown. To the right of the drop-down menu is the Compound button. This button must be selected if the object has children. When you select this, the object and its children will be considered collectively as a single compound object. The Compound option enables you to add multiple collision shapes in a single Rigid Body object. The secret of doing more-advanced physics simulations is in cleverly combining Convex Hull Polytopes and Static Triangle Mesh bound objects as compound objects in such a way that all the objects appear to interact with each other in an accurate way.

Example: Glass and Ball

In this example, the solution uses several boundary types together. The goal is to create a rigid body simulation with a ball and a glass, in which the ball can fall into the glass in a natural way and the glass can interact with both the ball and the floor plane correctly, as shown in Figure 6.38.

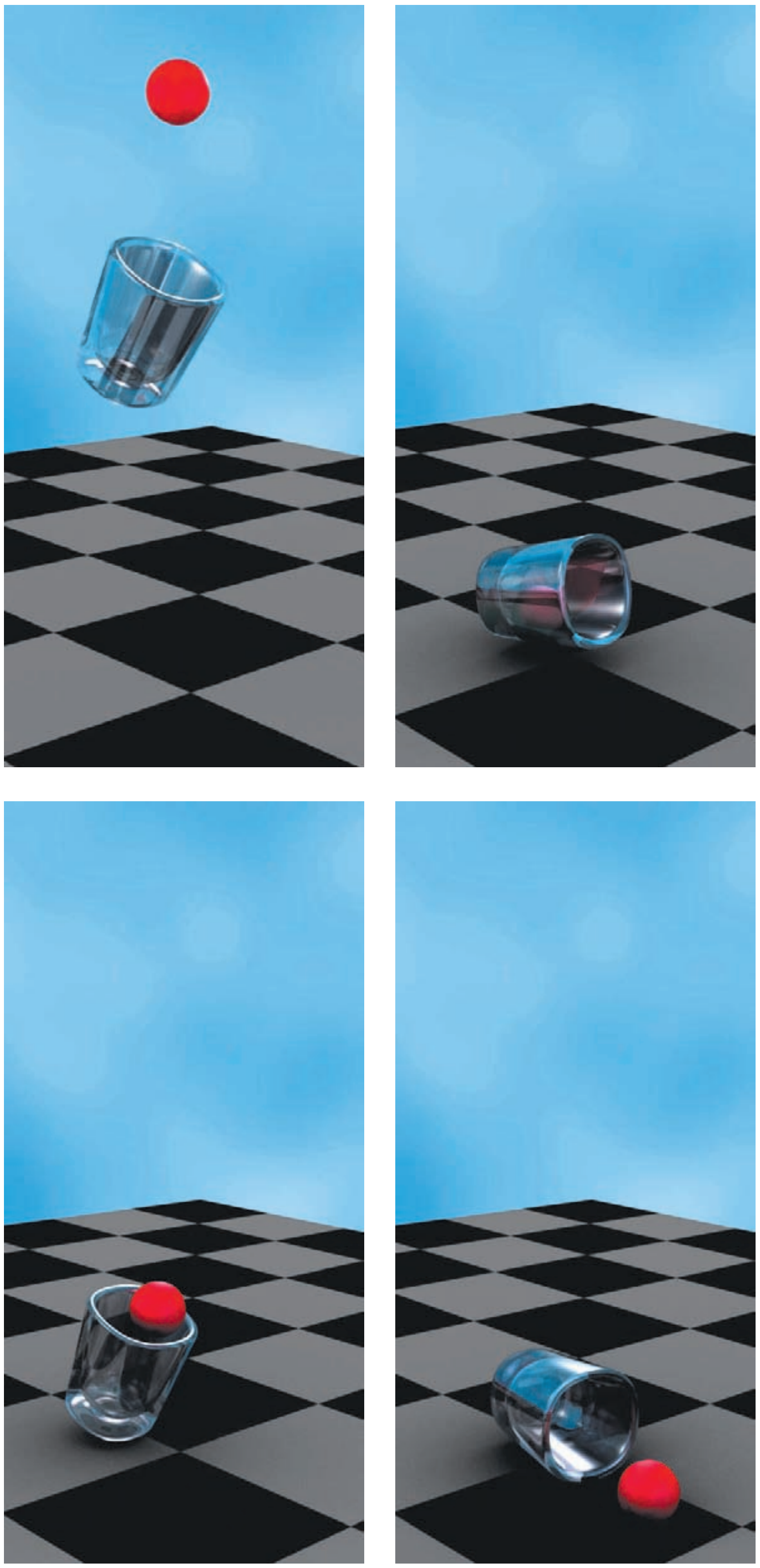


Figure 6.38 The ball interacting with the glass and the floor

As you've seen in previous examples, the floor and the ball are simple to set up. The difficulty here is the glass. The glass must be a dynamic Rigid Body object in order to fall and tumble naturally, so simply making it a Static Triangle Mesh is not an option. It does not fit into one of the predefined shapes such as sphere or box, so those are also out. Finally, using a Convex Hull Polytope boundary is also not possible, because this boundary type cannot represent concavities. If the glass is a Convex Hull Polytope, the ball will settle on "top" of the glass, rather than falling into the mouth of the glass, as shown in Figure 6.39. To get around these problems, it will be necessary to create a compound object for the cup composed of several component objects of different boundary types.

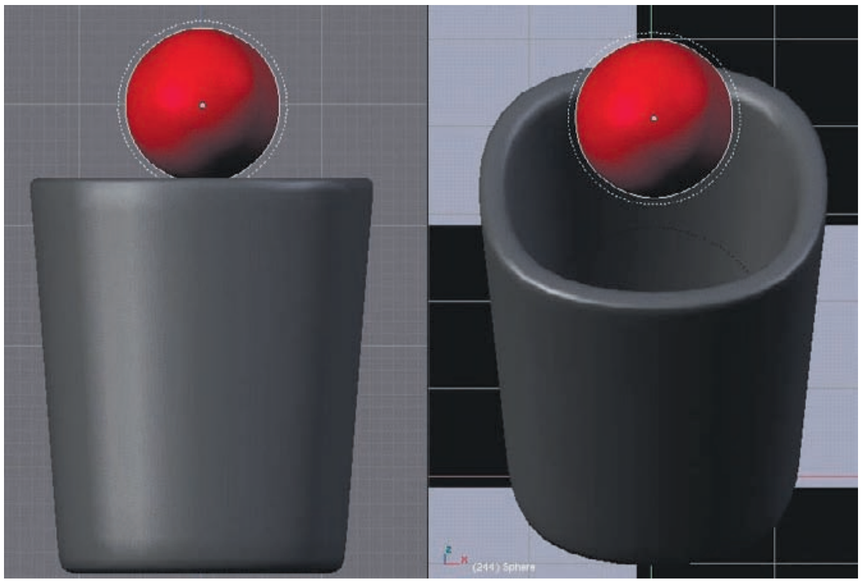


Figure 6.39 If the glass is a Convex Hull Polytope, the ball sits on top.

To see how this can be done, first model a glass with geometry as shown in Figure 6.40, or else append the object glass from the .blend file glassandball.blend, included on the CD that accompanies this book. Before beginning the following process, make sure that any object-level rotation or scaling has been applied with Ctrl+A, and any translations cleared with Alt+G. Also make sure that any Ipos associated with the object at this point are disconnected by clicking the X button to the right of the Ipo drop-down menu in the Ipo Editor header.

As mentioned earlier, it is possible to have a Static Triangle Mesh move if it is parented to a moving object. This means that it is possible to have the glass itself be a Static Triangle Mesh in order to interact correctly with the ball. However, if this is the case, the glass cannot collide with the ground; two Static Triangle Meshes do not collide with each other. It is necessary to create a set of special collision meshes to do this.

To create the special collision meshes, select the vertices of the cup shown in Figure 6.41. Duplicate these vertices by pressing Shift+D, and then press P and choose Selected to create a new object with these vertices. You can see the new object in Edit mode in Figure 6.42. Note the four vertices shown in orange around the lip of the cup.

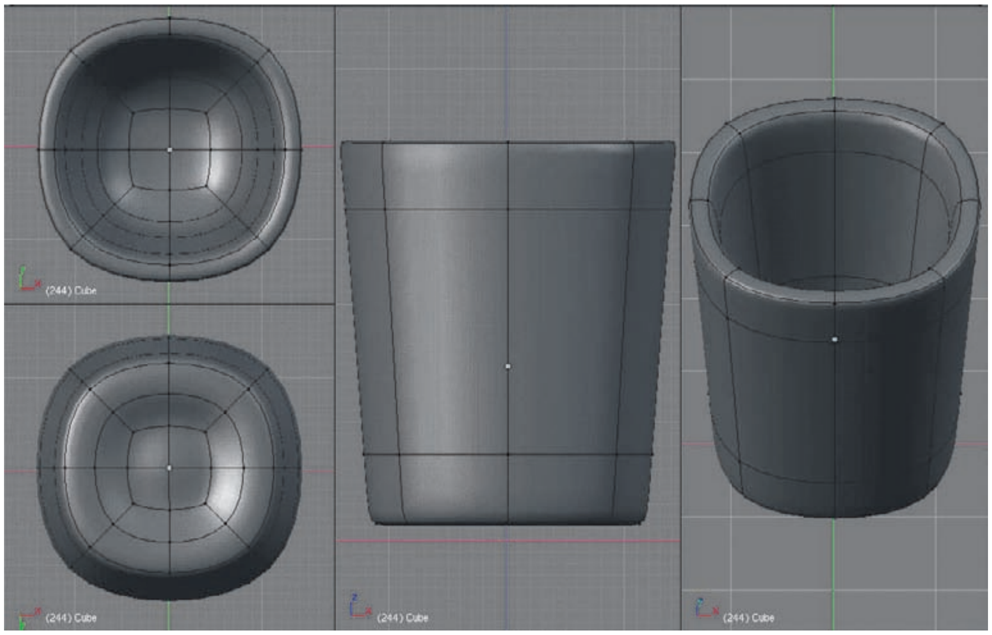


Figure 6.40 Model the glass with this geometry.

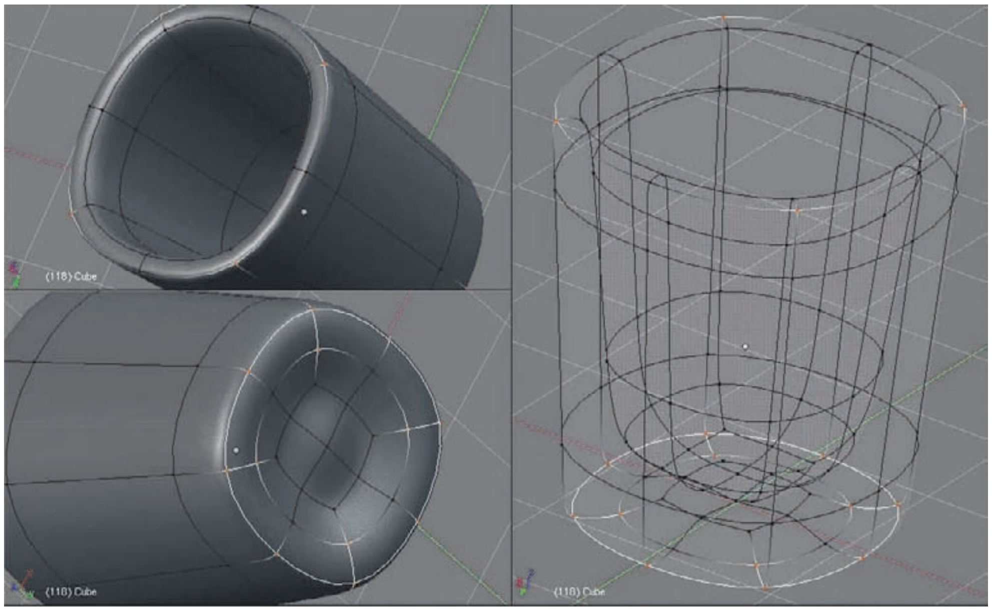


Figure 6.41 Separating the vertices

Eventually, this new collision mesh will use the Convex Hull Polytope collision bounds. Another quality of Convex Hull Polytopes is that only vertices that are part of faces contribute to the collision boundaries. So to make this collision mesh work, it is necessary to fill in some triangular faces between the vertices, as shown in Figure 6.43. Do this by pressing the F key while in Edit mode. Turn off subsurfing for this object. You can also delete the extra edges, because they don't do anything.

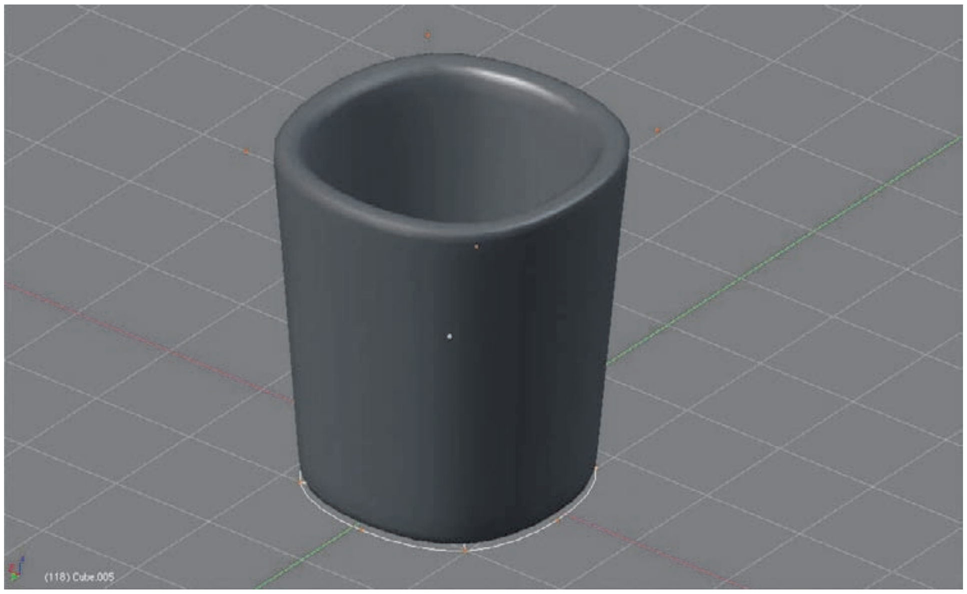


Figure 6.42 The separated collision mesh in Edit mode

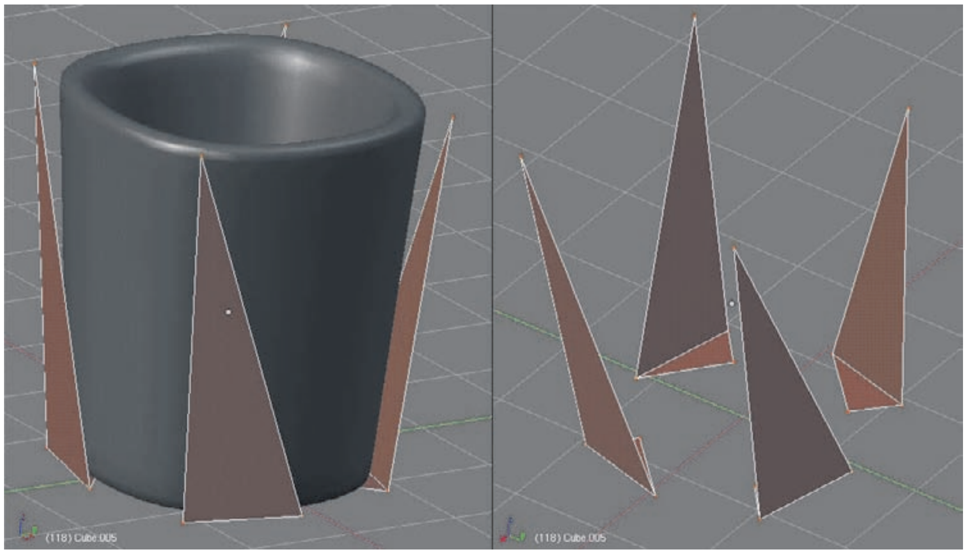


Figure 6.43 Filling in faces between vertices in the collision mesh

Because the model is subsurfaced, making its visible outline smaller than the actual mesh, the collision mesh should be reduced in size slightly. Reduce it along the X and Y axes (see Figure 6.44) by pressing the S key followed by Shift+Z. Then reduce it slightly less along the Z axis (see Figure 6.45) by pressing S followed by Z.

Now, separate each of the four corner segments of the mesh, one by one, by selecting their vertices, pressing P, and choosing Selected (see Figure 6.46). You should wind up with four objects. Name these objects *hull1*, *hull2*, *hull3*, and *hull4*. It doesn't matter which is which.

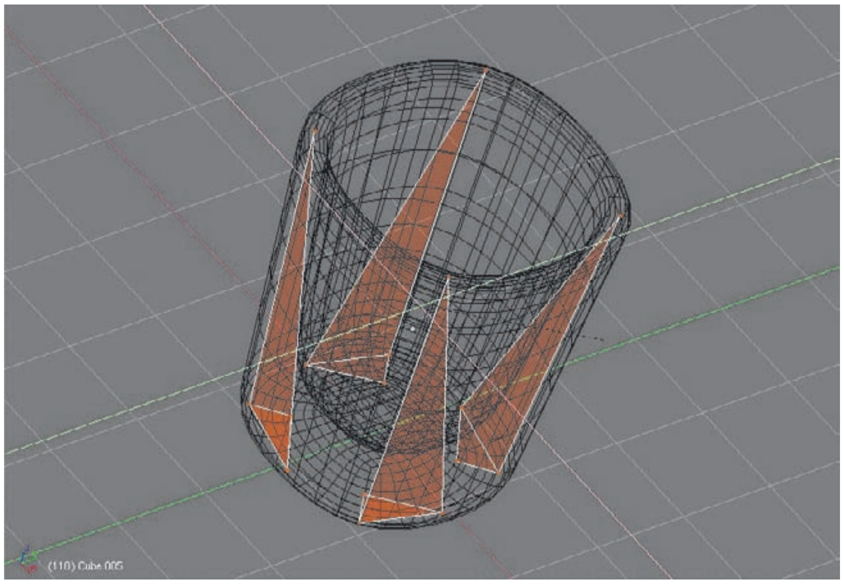


Figure 6.44 Scaling down along X and Y axes

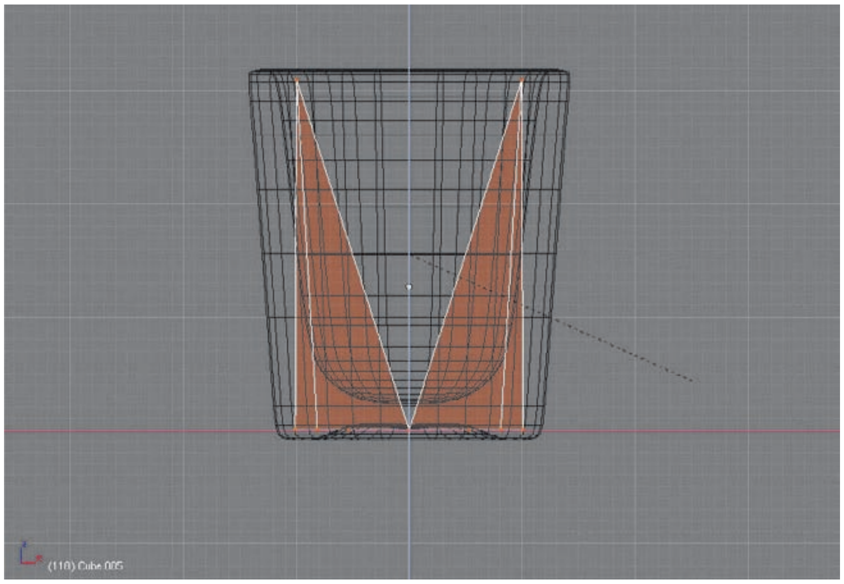


Figure 6.45 Scaling down along the Z axis

The object called hull1 is going to be the parent object for the compound Rigid Body object. In order for that to happen, each of the other objects (the other three hull objects *and* the glass mesh) all need to be parented to hull1. Furthermore, they need to be parented in a slightly special way. Parent each of the objects one by one to hull1 by selecting the child-to-be object, selecting hull1, pressing Ctrl+P,and selecting Make Parent (see Figure 6.47). Immediately after parenting each object, press Alt+P and select Clear Parent Inverse (see Figure 6.48). This step is easy to overlook, but it's necessary to make this setup work properly. Do this for each hull object and for the glass mesh.

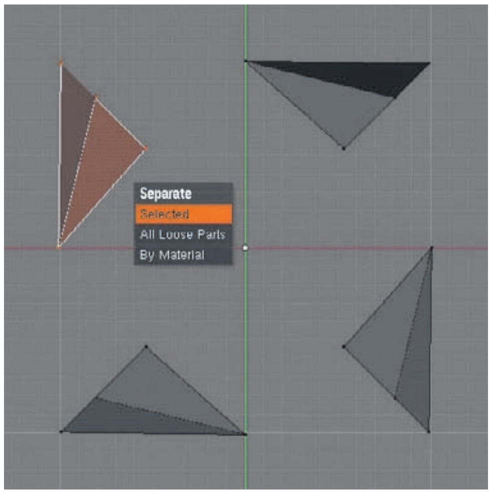


Figure 6.46 Separating one corner segment into its own object

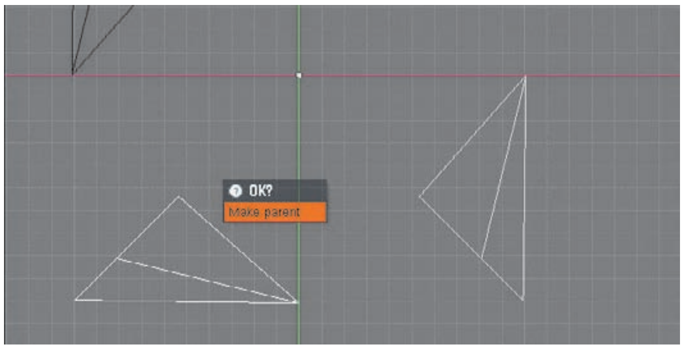


Figure 6.47 Parenting hull objects

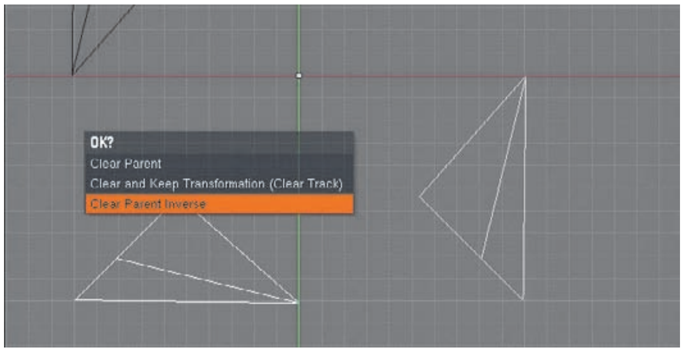


Figure 6.48 Clear parent inverse

After you have parented all the objects, set their boundary types. First, set hull1 as a Dynamic Rigid Body actor with boundary type Convex Hull Polytope and with Compound enabled (see Figure 6.49). Set hull2, hull3, and hull4 to have Convex Hull Polytope bounds. Finally, set the glass object to be a Static Triangle Mesh.

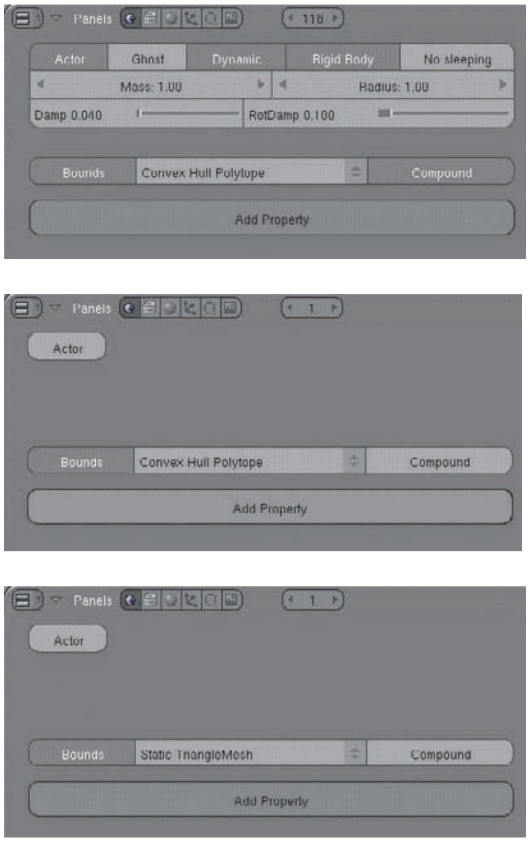


Figure 6.49 Settings for hull1, for hulls 2 through 4, and for glass

After you've set these values, set your ball and glass up where you want them at the beginning of the simulation. Use hull1 to move the glass object around. You can preview the simulation in the game engine quickly by pressing P, and press Esc when it finishes. Ctrl+Alt+Shift+P will create the Ipos.

Because you won't be rendering the Convex Hull Polytope collision meshes, either place them on an inactive layer or set them to not be renderable in the Outliner window, as shown in Figure 6.50.

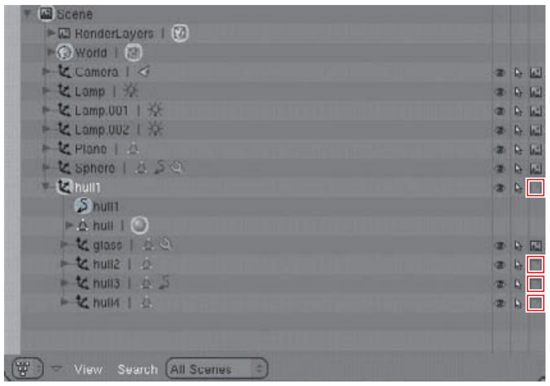


Figure 6.50 Setting the collision meshes not to render

Note: If some of your objects seem to go crazy after you bake the Ipos, you may have inadvertently left a shared Ipo between the parent object and one or more of the children objects. Make sure that this is not the case. If there are shared Ipos, disconnect them from the child objects by clicking X next to the Ipo name in the Ipo drop-down menu.

General Tips on Working with Bullet

There are several further points to keep in mind when working with Bullet:

- Avoid scaling rigid bodies. If you need to scale an object, disable the rigid body, scale the object, and apply the scale and rotation by pressing Ctrl+A.
- Avoid very small (<0.2 Blender Units in any direction) or very large rigid bodies.
- Avoid large mass ratios. Keep the ratio of masses between interacting rigid bodies less than 1:100.
- Avoid degenerate triangles. Degenerate triangles are extremely long, thin sliver-like triangles. Bullet will not handle these well.
- Make sure the center of the object is in the middle of the mesh.
- Don't make child objects in a Compound object dynamic.

Adjusting Simulation Quality with Python

It's possible to increase the quality of your simulations by increasing the number of internal simulation substeps. This can be done by simply running two lines of code, in the same way that you ran the Python script for adjusting the frame rate, described previously in this chapter. The code for changing the time steps is as follows:

```
import PhysicsConstraints
PhysicsConstraints.setNumTimeSubSteps(10)
```

The argument `10` in this example can be set to whatever value you want, with the default being `1`, and higher numbers resulting in better-quality simulation.

Using Physics Visualization

You can see more information about how the physics is working by selecting Show Physics Visualization in the Game drop-down of the Information window header, shown in Figure 6.51. When this is selected, color-coded physics information will be visible in Game Play mode when you press P.

The example in Figure 6.52 shows a small chain of three objects that are constrained to each other by rigid body joins. The leftmost cube is not dynamic; the other two objects are Rigid Body objects. When P is pressed, the Rigid Body objects fall in a chain and their physics is highlighted as shown in Figure 6.53. Bounding boxes are shown in red. Physics is calculated on the structures displayed in white, which indicate the Rigid Body object's center of gravity, and when the simulation reaches a state of equilibrium, those structures turn green, indicating that the Rigid Body object is "sleeping." Sleeping increases the efficiency of the simulation and reduces the possibility of jitter for motionless objects. Physics visualization is helpful for seeing how objects are colliding with each other and also to make sure that Compound objects are set up correctly.

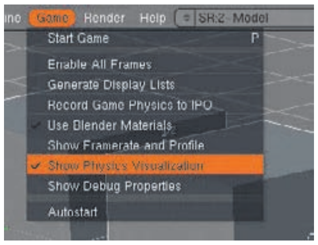


Figure 6.51
Show Physics Visualization

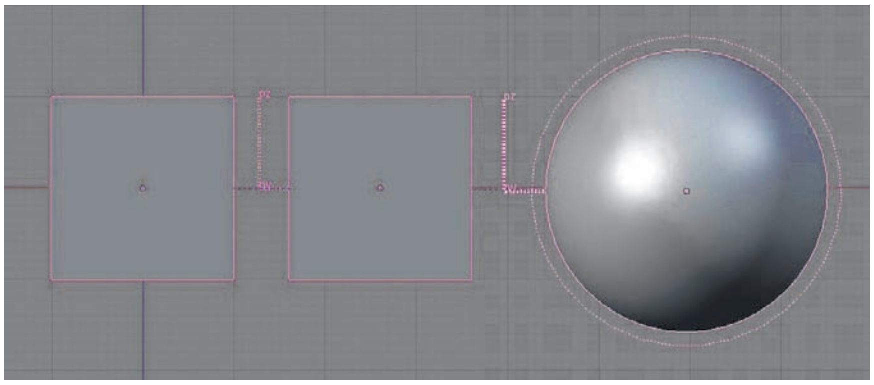


Figure 6.52 Objects with Rigid Body constraints

Joints, Ragdolls, and Robots

In 3D, as in the real world, joints are mechanisms that hold two solid objects together, allow certain freedoms in how the objects move in relation to each other, and restrict other kinds of movements. In 3D there are usually considered to be three fundamental types of joints that differ in the kinds of movement they allow. These are hinges, ball-and-socket joints, and slider joints.

Figure 6.54 illustrates the different ways that these joints behave. The hinge allows the objects to rotate on a specific axis around a common point with respect to each other. The ball-and-socket joint allows combinations of rotation around all three axes. The slider joint allows the objects not to rotate but to translate (move) along an axis with respect to each other.

Using Rigid Body Joint Constraints

In Blender, you can set up joints by using the Rigid Body Joint constraint found in the Constraints tab of the Object buttons context (F7). To get an idea of these, start up a fresh session of Blender and duplicate the default cube and move the new copy, Cube.001 along the X axis so that it is next to the original cube with a small space between them. With the new cube selected, select the Rigid Body Joint constraint from the Constraints drop-down menu, as shown in Figure 6.55.

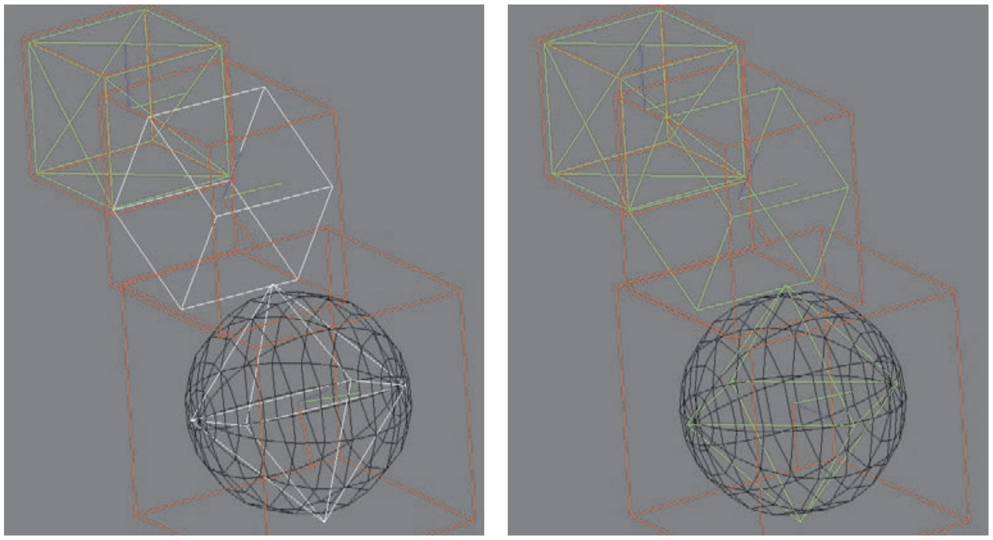


Figure 6.53 Visualizing the simulation in action and sleeping

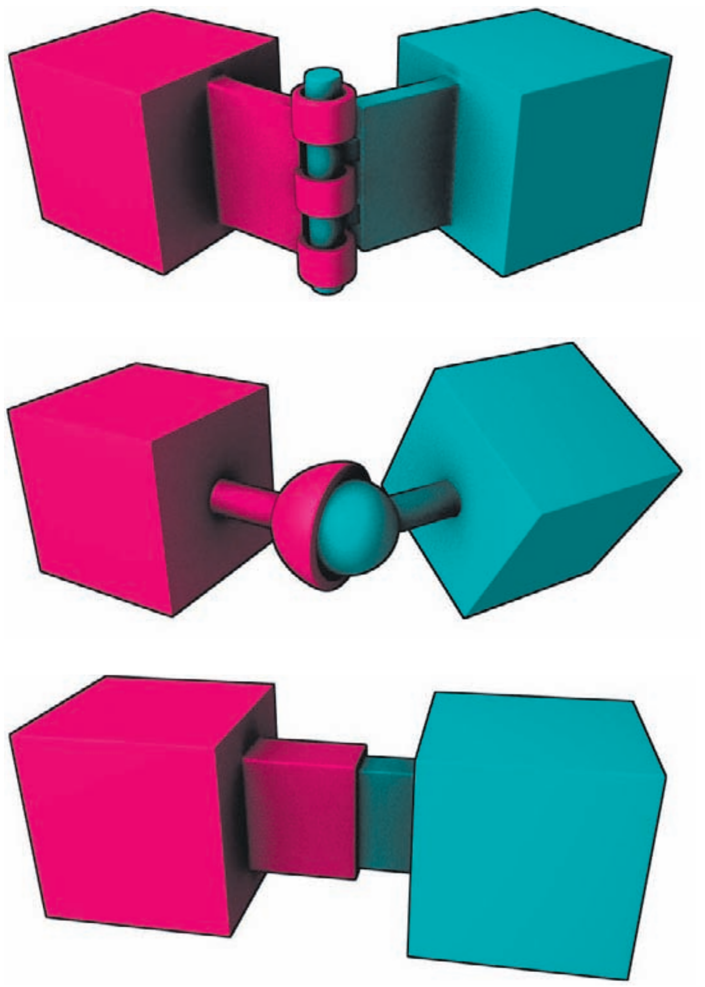


Figure 6.54 Hinge, ball-and-socket, and slider joints

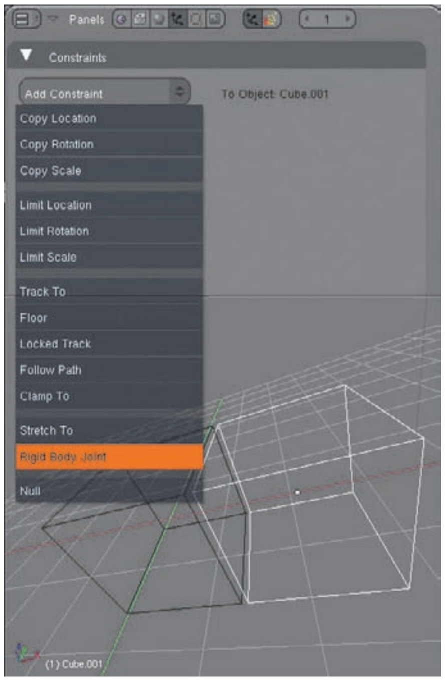
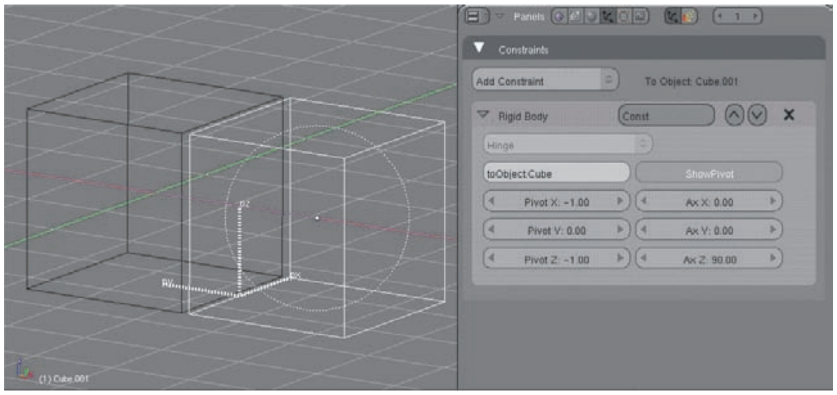


Figure 6.55
Rigid Body Joint constraint

The options for the type of joint are Ball, Hinge, and Generic (experimental). Ball and Hinge are ball-and-socket joints and hinges (respectively) along the lines of the earlier diagrams. The Generic joint option enables you to set numerical restrictions on six degrees of freedom, three axes of rotation and three axes of translation. This enables you to not only represent slider joints but also to lock or restrict movement to a specified range for each degree of freedom. For this example, select Hinge.

Set the Hinge parameters as shown in Figure 6.56. The toObject field is for the object that the hinge attaches to, in this case the first cube. ShowPivot makes the constraint's pivot point visible in the 3D viewport. Pivot X, Pivot Y, and Pivot Z locate the pivot in space with relation to the object's center (measured in Blender Units). The Ax X, Ax Y, and Ax Z rotate the constraint. The Hinge constraint rotates around its own X axis; therefore, to line the hinge up along the bottom edges of the cube, enter a Pivot X value of –1, a Pivot Z value of –1, and an Ax Z value of 90.



To Object: Cube.001
Const
Hinge
toObject:Cube
Pivot X: -1.00
Pivot Y: 0.00
Pivot Z: -1.00
Ax X: 0.00
Ax Y: 0.00
Ax Z: 90.00

Figure 6.56 Hinge parameters

In the Logic buttons, set Cube.001 to be a dynamic rigid body actor. Leave Cube as it is. Press Ctrl+Alt+Shift+P to generate the physics simulation Ipos. When you play back the animation, it should look something like Figure 6.57. As an experiment, try switching the joint type to Ball and baking the Ipos again to see how the two joints behave differently.

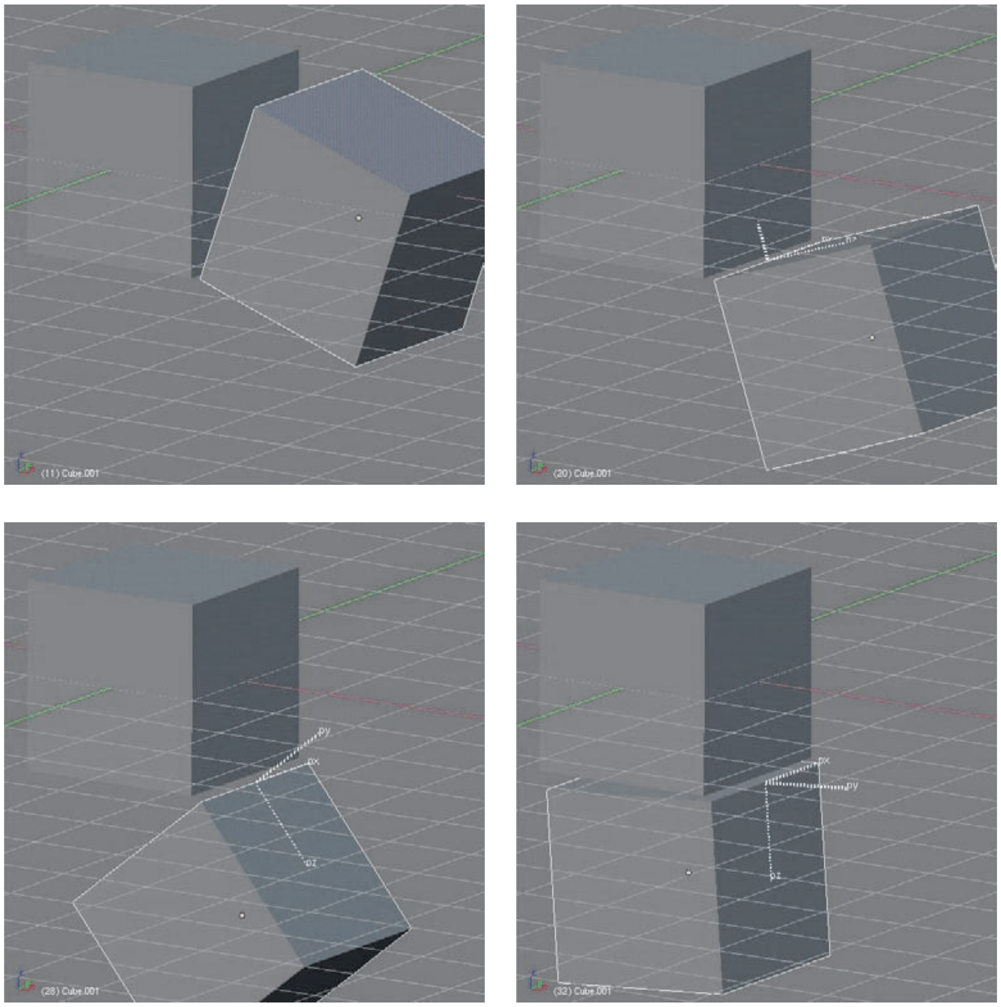
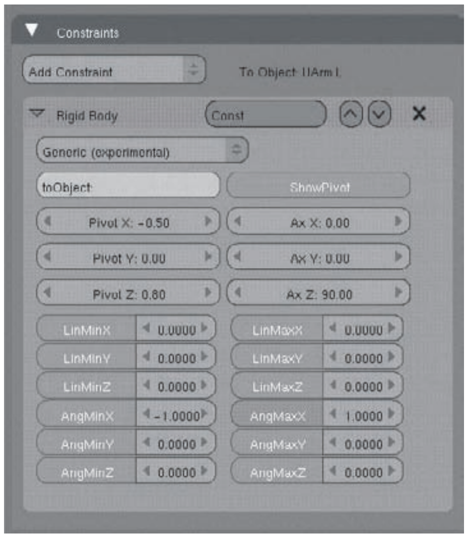


Figure 6.57 Hinge in action

Using Generic (6DoF) Joints

If you took the next logical step in the preceding example and experimented with switching the joint type to Generic (experimental), you may have gotten a little bit of a surprise when Cube.001 just dropped into oblivion with no apparent connection to Cube at all. By default, all six degrees of freedom in the Generic joint are unconstrained, which means that when you first set the joint, it has no effect at all. To constrain them, click the button corresponding to the degree of freedom you want to constrain, and set minimum and maximum values for their range. If you leave the default values of 0 and 0, the object will be fully constrained in that degree of freedom. For linear values, the units are Blender

Units. For rotation values, the possible range is from –5 (representing a 180-degree counterclockwise rotation) to 5 (representing a 180-degree clockwise rotation). For example, in the settings shown in Figure 6.58, all the buttons are depressed, so the object would be prevented from moving in any degree of freedom except rotating 36 degrees in either direction around the X axis.


Constraints

Add Constraint   To Object: UArm.L

Rigid Body   Const

Generic (experimental)

toObject: ShowPivot

Pivot X: -0.50   Ax X: 0.00
Pivot Y: 0.00   Ax Y: 0.00
Pivot Z: 0.80   Ax Z: 90.00

LinMinX 0.0000   LinMaxX 0.0000
LinMinY 0.0000   LinMaxY 0.0000
LinMinZ 0.0000   LinMaxZ 0.0000
AngMinX -1.0000   AngMaxX 1.0000
AngMinY 0.0000   AngMaxY 0.0000
AngMinZ 0.0000   AngMaxZ 0.0000

Figure 6.58 Generic constraint settings

Another joint type, the Cone Twist joint, is currently under development and may be ready to use in an official version by the time this book has gone to print. Check the resources listed at the end of this chapter for information on how to use this joint type.

Setting Up a Ragdoll Armature

Animators and game creators spend a lot of time with human figures, and for this reason a complete discussion of physics should touch on how to make it act on the human form. This approach to treating the human body as a purely physical collection of parts connected by joints is referred to as *ragdoll physics*.

In this section, you'll see how to use ragdoll physics to control a simple armature. In this way, you can use ragdoll physics to control the movements of mesh characters. If you don't yet know the basics of working with armatures and using them to deform meshes, you can find out everything you need to know in my book *Introducing Character Animation with Blender* (Sybex, 2007).

Ragdolls can be simple or complex. Likewise, they can be more or less restricted in their range of motions. A less-restricted ragdoll can more accurately mimic the range of positions of a human, but it is also more likely to get into impossible positions. Setting up a ragdoll is similar in some ways to setting up an armature, but with an important difference: An armature is intended to be posed by hand, so if it is more flexible than necessary, this is not usually a problem. A ragdoll is intended to be "posed" only by the forces that act on it; therefore, it must be very carefully constrained so as to avoid getting into unnatural positions. At present, constructing a very sophisticated ragdoll like this in Blender is a painstaking and difficult process, but improvements are planned for Bullet's constraint system that will make this process much easier within the next release or two. For now, though, I'll stick with a very simple ragdoll. When you create your own ragdoll, you can use whatever shapes you like: cylinders, spheres, or other convex polyhedrons. Don't use objects that are too small, though, as this can cause instability in the simulation. In this example, because I am using mainly the mesh objects to control the movement of empties and an armature, I will stick with the simplest possible construction of just boxes, most of which are approximately the scale of the default cube.

This ragdoll is composed of 11 separate Cube objects, scaled and positioned as shown in Figure 6.59. Rename these objects **Head**, **Torso**, **Midriff**, **UArm.L**, **UArm.R**, **LArm.L**, **LArm.R**, **ULeg.L**, **ULeg.R**, **LLeg.L**, and **LLeg.R**. When you have the objects scaled and placed as you want them, press Ctrl+A on each object to apply scale and rotation.

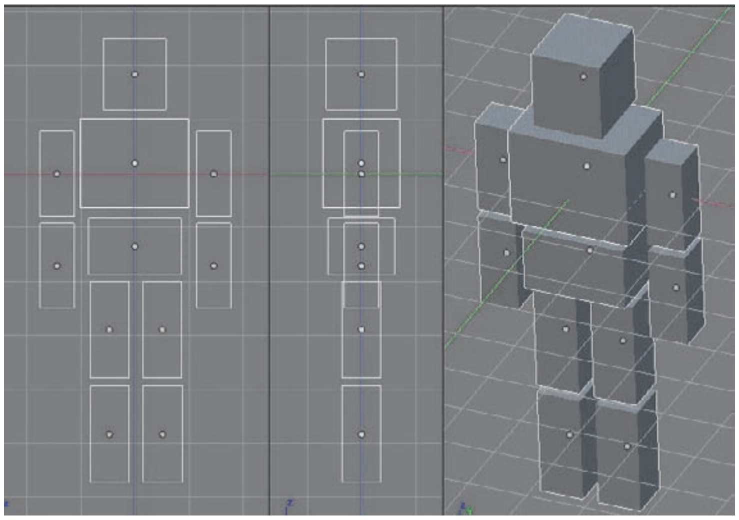
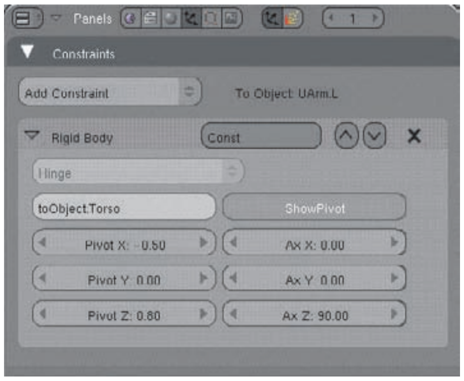


Figure 6.59 The objects that compose the ragdoll

The shoulders, elbows, and knees of this ragdoll will be joined by Hinge constraints. The constraints themselves have no built-in restrictions, but by placing the hinges correctly, it is possible to make the meshes restrict the way the joints bend. Begin with the left upper arm, object UArm.L. Add a constraint of type Hinge with a toObject value of Torso. Set the values as shown in Figure 6.60 so that the pivot is visible and positioned as in Figure 6.61. The only way to adjust the positioning of the pivots is through these numerical values. If you need to use slightly different values to get your pivot positioned correctly, that's no problem. Set the right shoulder pivot up as a mirror image of the left shoulder by following the same steps and setting the values as shown in Figure 6.62.



To Object: UArm.L

Const

Hinge

toObject:Torso

Pivot X: -0.50 | Ax X: 0.00

Pivot Y: 0.00 | Ax Y: 0.00

Pivot Z: 0.80 | Ax Z: 90.00

Figure 6.60 Hinge constraint values

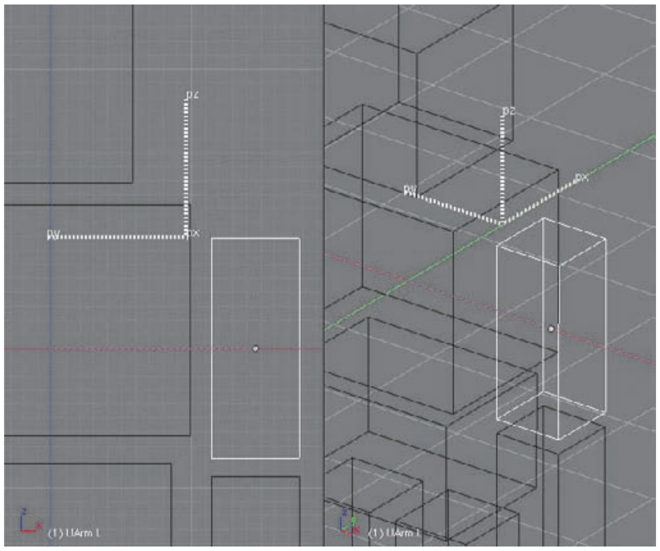
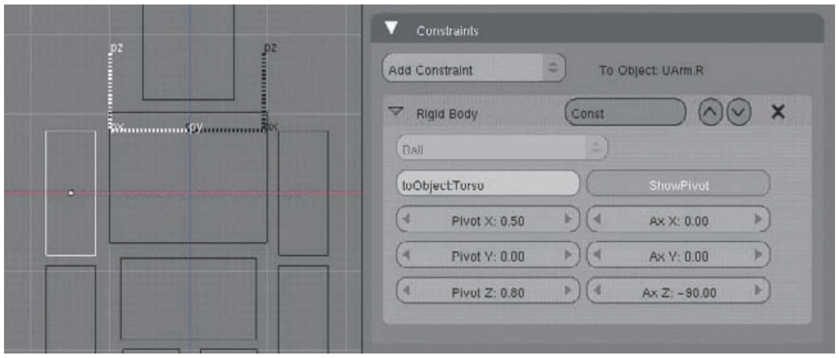


Figure 6.61 Position of the shoulder pivot



To Object: UArm.R

Const

Ball

toObject:Torso

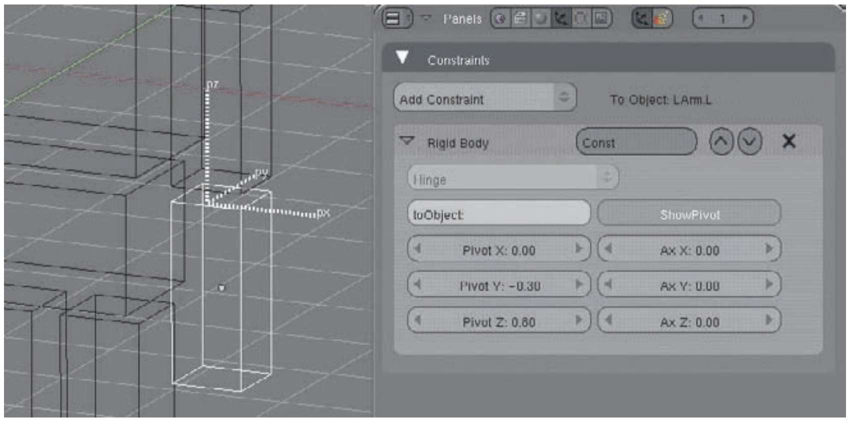
Pivot X: 0.50 | Ax X: 0.00

Pivot Y: 0.00 | Ax Y: 0.00

Pivot Z: 0.80 | Ax Z: -90.00

Figure 6.62 Right shoulder pivot

Create an elbow joint by adding a Hinge constraint on LArm.L targeted to UArm.L, as shown in Figure 6.63. Do the same on the right side.



Constraints

Add Constraint To Object: LArm.L

Rigid Body Const

Hinge

toObject: ShowPivot

Pivot X: 0.00 Ax X: 0.00

Pivot Y: -0.30 Ax Y: 0.00

Pivot Z: 0.80 Ax Z: 0.00

Figure 6.63 Left elbow

To see the constraints in action, go to the logic buttons and set Torso, UArm.L, UArm.R, LArm.L, and LArm.R as actors. Make them Dynamic Rigid Body objects and set their Bounds to Box. You can tell which objects are enabled as actors in the 3D viewport by whether their bounds radius is visible, as shown in Figure 6.64. Make sure that Record Physics To Ipos is *not* selected, and press P to take a look at the ragdoll physics so far. You should see the arms and torso dangle freely, as in Figure 6.65. They do not fall down, because they are being supported by nondynamic objects below the torso.

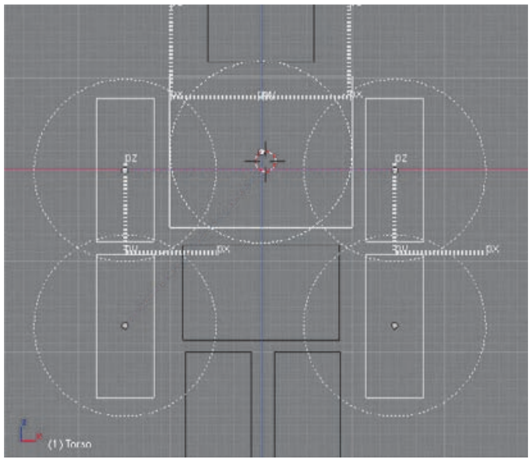


Figure 6.64 Torso, UArm.L, UArm.R, LArm.L, and LArm.R set as dynamic objects

Now set up the rest of the ragdoll. In this example, the head will not move freely—it will be fixed to Torso by parenting. To do this, parent Head to Torso as shown in Figure 6.66. Be sure to select Compound for the Torso Bounds.

The hinges at the hips should be placed flush with the front faces of the legs, so that the hinges bend forward, as shown in Figure 6.67. By contrast, the knee constraints should be flush with the back faces of the legs, as shown in Figure 6.68, so that the knees bend in the opposite direction. Note that all of these constraints are hinges. In real life, the human hip is a restricted ball-and-socket joint, but I'm opting for simplicity in this example.

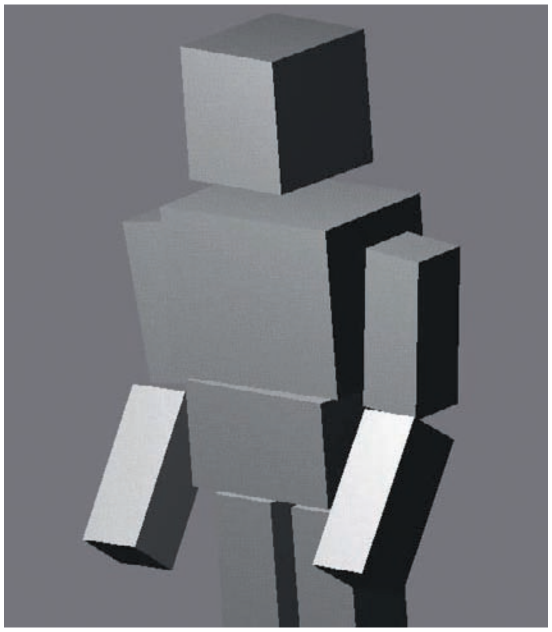


Figure 6.65 Ragdoll physics for the arms and torso

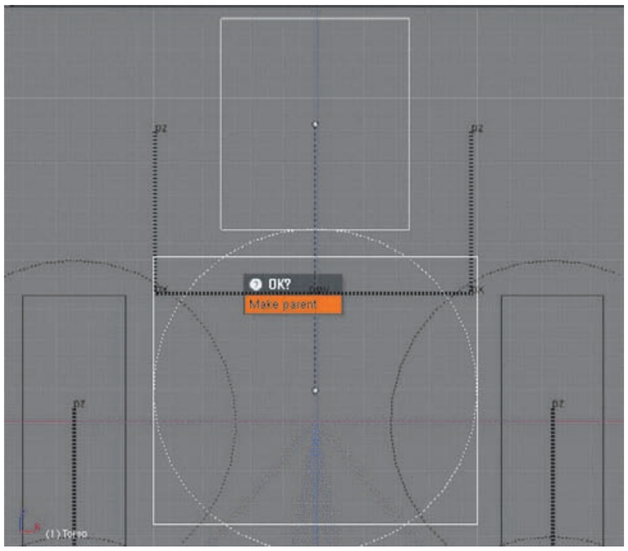


Figure 6.66 Parent Head to Torso

Add a Hinge constraint from Midriff to Torso, as shown in Figure 6.69. This will enable the body to bend slightly at the middle. Again, the movement is much more restricted than in a real human body, but this will be plenty for now. Finally, in the Logic buttons, set the rest of the objects up as dynamic rigid body actors.

If you press P now, your ragdoll should fall straight down into oblivion. If not, double-check to make sure that all the objects are set as dynamic rigid body objects. If parts of the body separate from others, double-check that the correct constraints are present and that they are targeted to the correct objects. If parts go haywire, make sure there are no parent-child relationships between meshes that are not part of Compound dynamic objects. Only Torso should be a compound object.

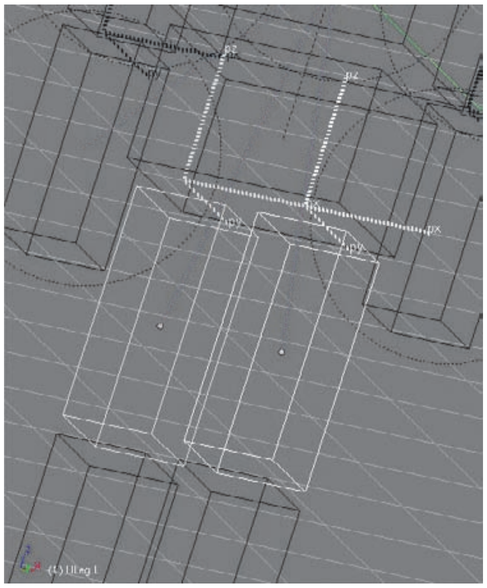


Figure 6.67 Hip joint placement

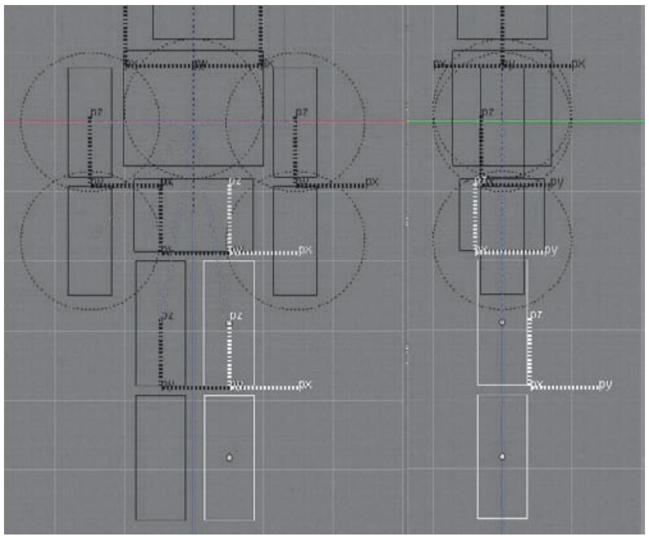


Figure 6.68 Hips and knees in relation to each other

Falling straight into oblivion isn't very interesting. Add a plane below the ragdoll and tilt it to one direction so that the ragdoll falls onto the plane and slides off. You can press P to see the action, and press Ctrl+Alt+Shift+P to bake the physics. After you render the ragdoll with some camera motion, you should see it take a tumble along the lines of Figure 6.70.

For slightly more realistic motion, you can adjust the masses of the various parts of the body to have more-accurate relative masses.

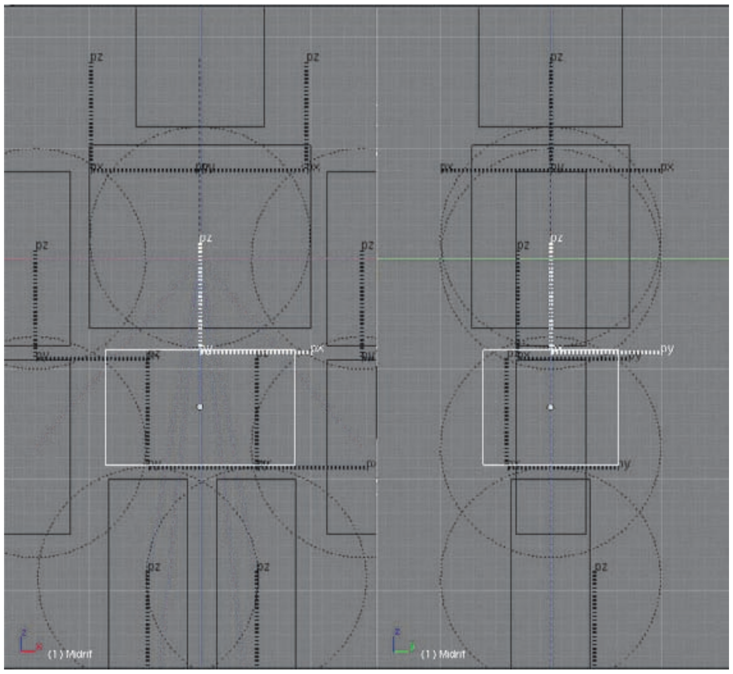


Figure 6.69 Midriff connected to Torso with a hinge

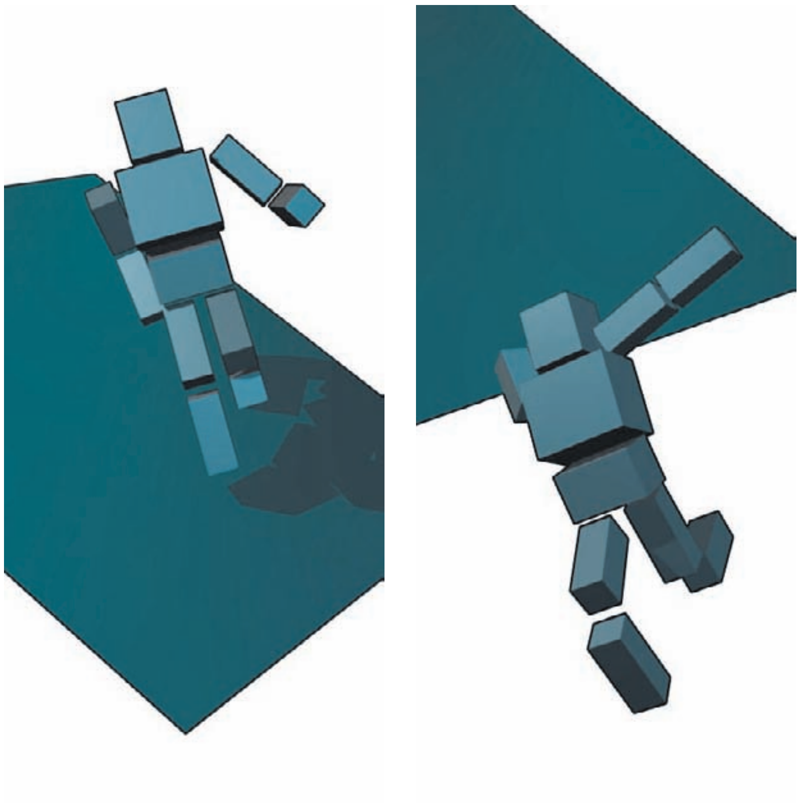


Figure 6.70 The ragdoll takes a fall.

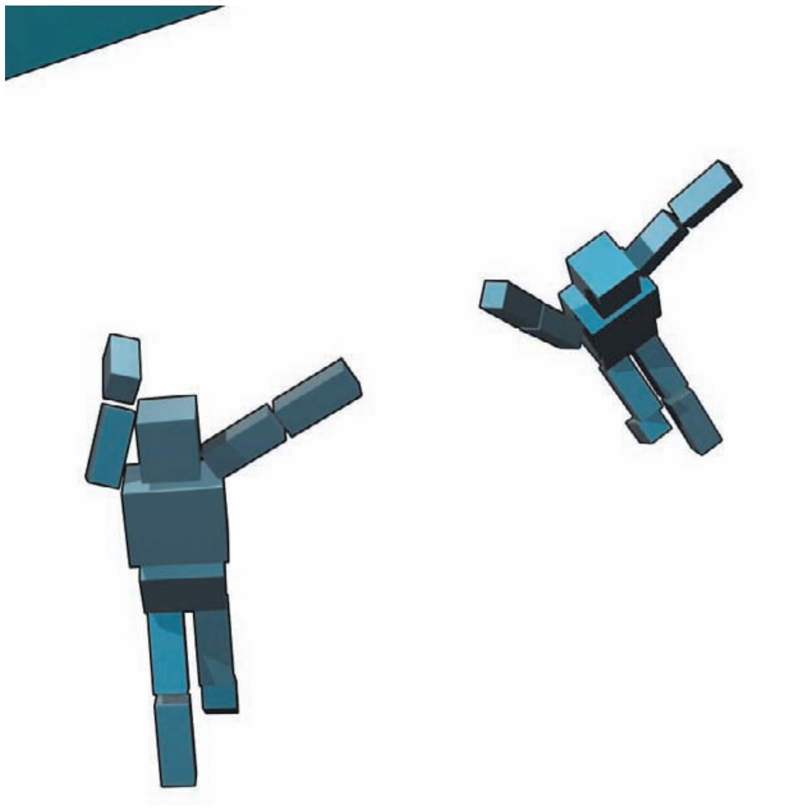


Figure 6.70 *(continued)*

Controlling an Armature

It would be nice to be able to use ragdoll physics to control an armature, in order to deform a mesh for animation. In this section, I'll show you a simple way to do this. The example is a little backward; ordinarily you would begin with a mesh character, create a ragdoll armature to fit the character, and then set up the ragdoll as in the previous section, so that the ragdoll's shape matches the character. In this case, the ragdoll came first. Nevertheless, the way to control the armature is the same.

The first thing to do is to add empties to key points, where the tips of the bones will be, as shown in Figure 6.71. An easy way to position empties is to select the vertices around where the empty should go and snap the cursor to selection by pressing Shift+S (as shown in Figure 6.72 for positioning the 3D cursor at the end of the left arm), and then add the empty in Object mode (Figure 6.73), and parent it to the body part that should control it (Figure 6.74). Parent the "hand" and "foot" empties to the lower arms and legs, respectively, the knee empties to the upper legs, the elbow empties to the upper arms, and the hip empties both to Midriff.

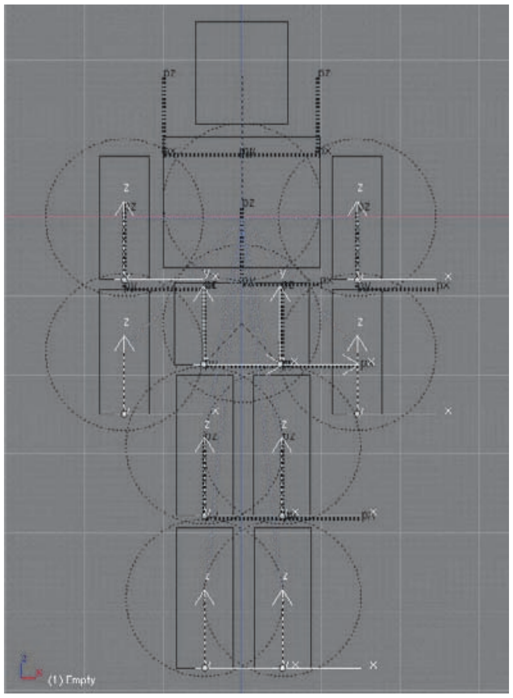


Figure 6.71 Placing empties at key posing points

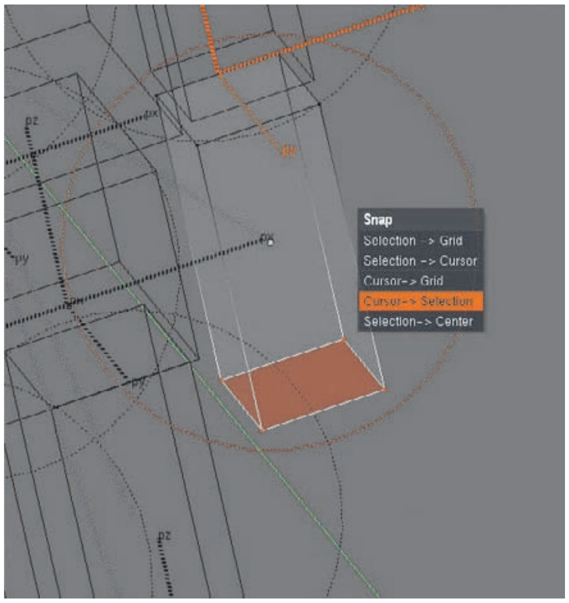


Figure 6.72 Snapping to a face

Now for the armature. In Object mode, select the Midriff object and snap the cursor to the selection. Add an Armature object and edit it as shown in Figure 6.75. Snap the bone tips to the empties you just added.

Finally, enter Pose mode and set up IK constraints on each of the bones shown in yellow in Figure 6.76. Enter the name of the empty at the bone's tip as the IK target, and make sure that chain length is set to 1 for each bone.

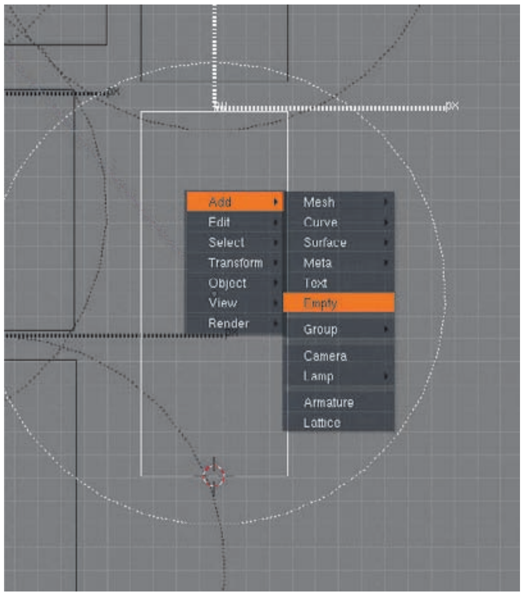


Figure 6.73 Add an empty.

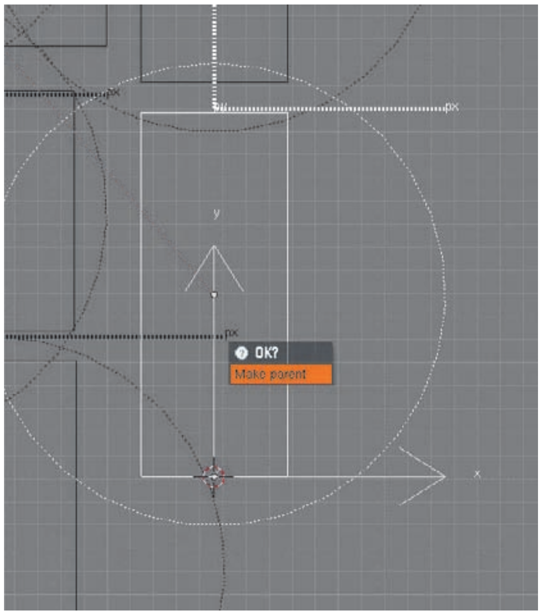


Figure 6.74 Parent the empty to the object.

Press Ctrl+Alt+Shift+P and run the animation again to see the ragdoll armature in action, as in Figure 6.77.

You'll find the full .blend file for this ragdoll (ragdoll.blend) on the CD that accompanies this book.

A Passive-Walking Robot

A current area of robotics research is *passive walking*. This is the study of the way that walking can be accomplished by means of a repetition of "controlled falls," where the walker simply follows the forces of physics at certain points in the walk cycle, and then exerts itself at key points to adjust its balance and ensure that it is in the right position to take the next step. In fact, this kind of passive walking is an important part of how humans actually walk, and a big reason why human walking is so fluid and smooth when compared to the way most current robots walk. Rather than controlling every movement with our muscles, a lot of what we do is to allow the forces of physics to work for us.

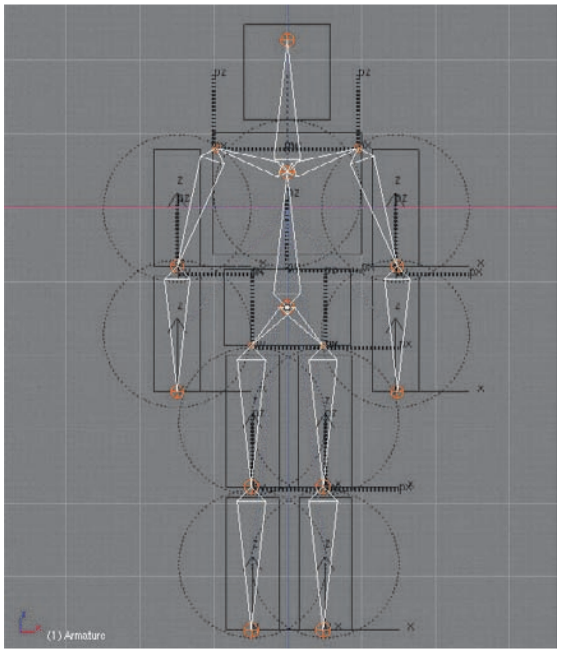


Figure 6.75 The armature

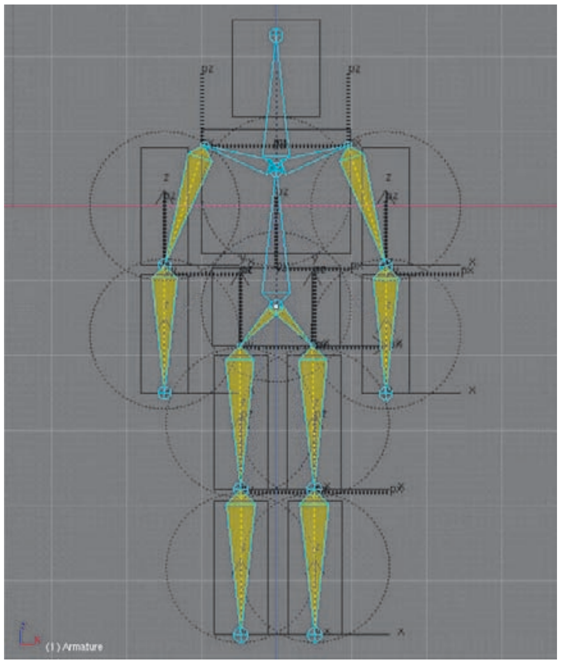


Figure 6.76 IK setup

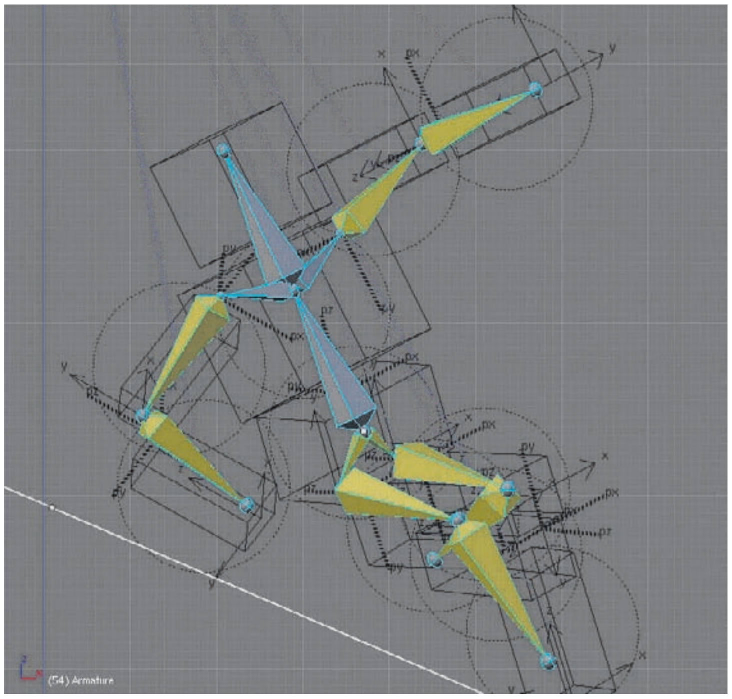


Figure 6.77 Ragdoll armature in action

Passive walking is closely related to ragdoll behavior, and it is possible to construct ragdoll-like walkers that can carry out passive walking in Blender. Yuta Fuji (Hans), a researcher who studies passive walking, has done just this with his intriguing Bullet/Blender passive-walking robot. You can find the simulation in the file `KneeRoboWalk3pv.blend` on the CD that accompanies this book.

This visualization uses a three-legged robot, as shown in Figure 6.78. As you can see in the figure, the robot's natural joints are represented by Hinge constraints, just as they were in the ragdoll example in the previous section.

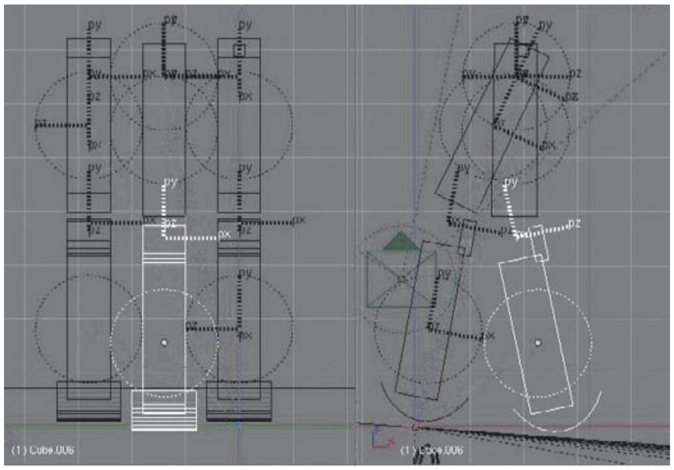
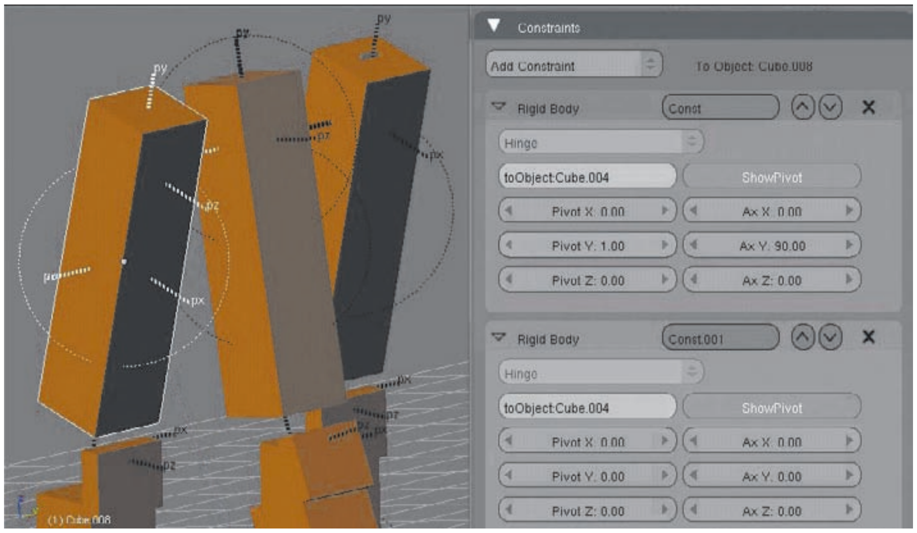


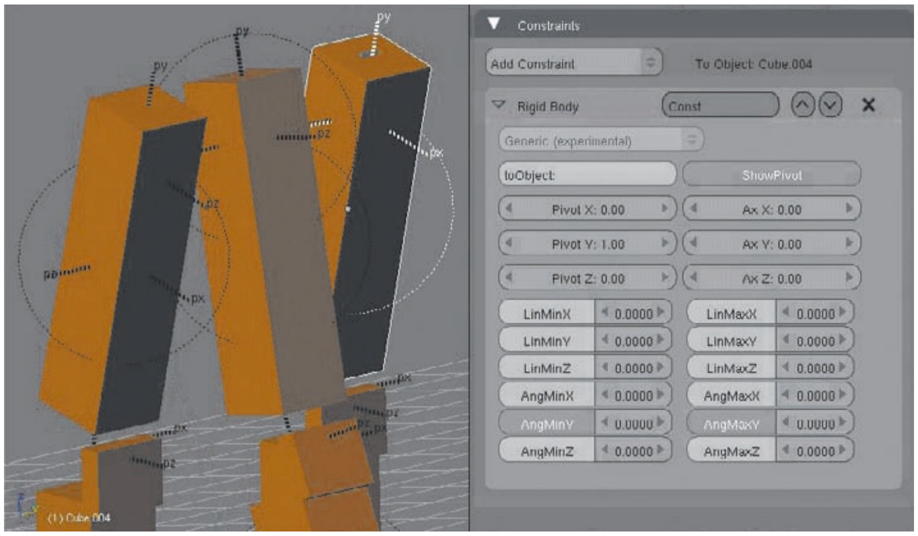
Figure 6.78 The passive-walking robot

Constraints are used in a few other interesting ways. The robot's right leg is made to follow its left leg's motion by means of two orthogonal Hinge constraints, as shown in Figure 6.79. The left leg is also constrained by a Generic constraint with no target, set to be unrestricted on all degrees of freedom except Y rotation, which it constrains fully (0 min, 0 max), as shown in Figure 6.80. This ensures that the object can move and rotate freely in any direction except twisting.



```
Constraints
Add Constraint          To Object: Cube.008
Rigid Body   Const
  Hinge
  toObject:Cube.004     ShowPivot
  Pivot X: 0.00         Ax X: 0.00
  Pivot Y: 1.00         Ax Y: 90.00
  Pivot Z: 0.00         Ax Z: 0.00
Rigid Body   Const.001
  Hinge
  toObject:Cube.004     ShowPivot
  Pivot X: 0.00         Ax X: 0.00
  Pivot Y: 0.00         Ax Y: 0.00
  Pivot Z: 0.00         Ax Z: 0.00
```

Figure 6.79 Hinge constraints



```
Constraints
Add Constraint          To Object: Cube.004
Rigid Body   Const
  Generic (experimental)
  toObject:             ShowPivot
  Pivot X: 0.00         Ax X: 0.00
  Pivot Y: 1.00         Ax Y: 0.00
  Pivot Z: 0.00         Ax Z: 0.00
  LinMinX  0.0000       LinMaxX  0.0000
  LinMinY  0.0000       LinMaxY  0.0000
  LinMinZ  0.0000       LinMaxZ  0.0000
  AngMinX  0.0000       AngMaxX  0.0000
  AngMinY  0.0000       AngMaxY  0.0000
  AngMinZ  0.0000       AngMaxZ  0.0000
```

Figure 6.80 An untargeted Generic constraint

If you press P to run the game engine, you can watch the robot walk, as in Figure 6.81. Don't use Ctrl+Shift+Alt+P for this, though. The walker is sensitive to changes in any aspect of the environment; mass of objects, speed, relative distances, and the slope of the walking surface all need to be set just right. The simulation was set up with game engine parameters in mind, and because certain parameters (speed, in particular) are different when Ctrl+Shift+Alt+P is used, the robot's walk becomes unstable, and it takes a nasty tumble after only a step or two. Note also that the simulation was created in Blender 2.44, and may be unstable in other versions of Blender. It may also be unstable on operating systems other than Windows. On my Mac Pro, the robot stumbles at about the 4 marker.

You can watch the movie of the robot online at the following site:

```
http://video.google.com/videoplay?docid=-7369019972032648620
```

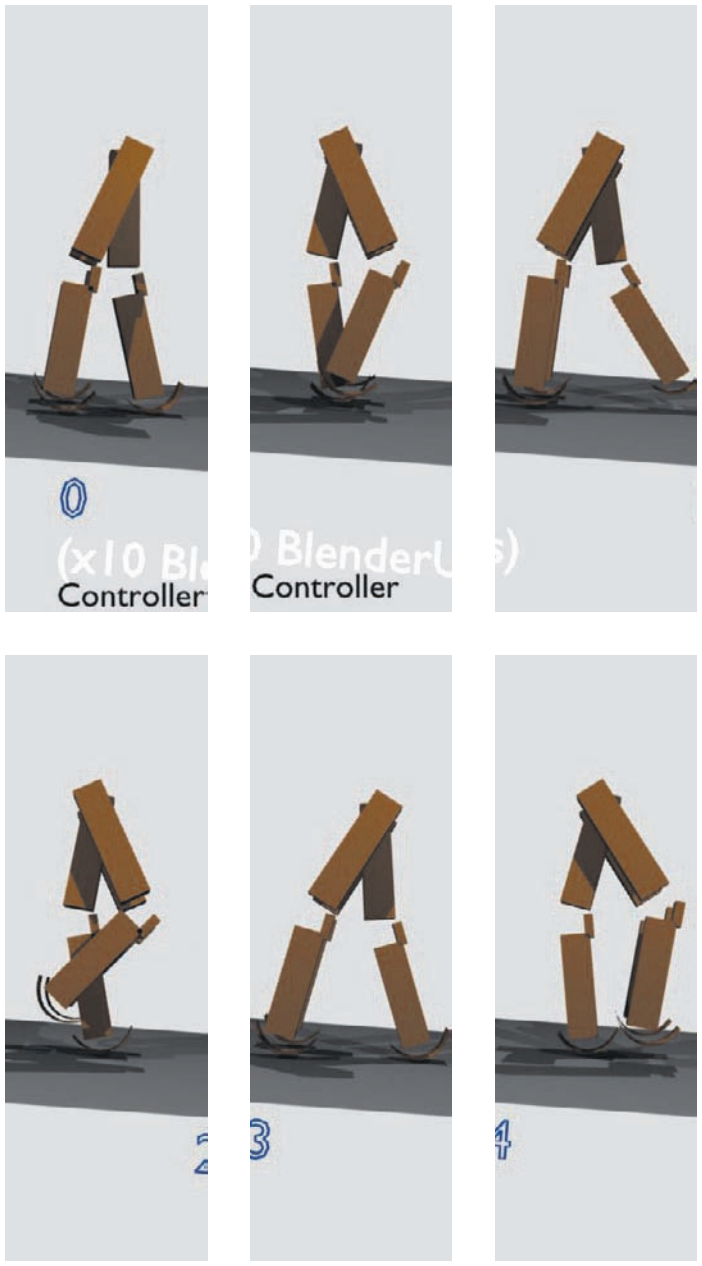


Figure 6.81 The walking robot in action

Erwin Coumans

Erwin Coumans wrote and integrated Blender's rigid body dynamics system. He is currently simulation team lead at Sony Computer Entertainment America. Since studying computer science at Eindhoven University, he has worked for Guerrilla Games, Blender, SCEE London, and Havok in Ireland doing collision detection and physics development. He actively promotes the sharing of ideas and technology through forum discussion, open standards and the use of open source software. He is the main author of the Bullet Physics Library, used by several game companies for games on XBox 360, PC, Wii and PS3. At the moment, he is writing a book on advanced game physics which will cover topics including continuous collision detection, iterative constraint solvers, and taking advantage of multi-threaded architectures like multi-core CPUs, GPGPU and Playstation 3 Cell. His book will be published by Morgan Kaufmann.

Further Resources

You can find out much more about the Bullet Physics Library at the project's website at http://bulletphysics.com. Documentation for new features and a forum devoted to Bullet can be found there. For general user support, go to the Game Engine forum at http://blenderartists.org/forum.

On the CD that accompanies this book, you'll find the physics regression test files for Bullet, which are a great source of examples for learning more about the features described in this chapter. You'll also find the .blend files for the top three winners of the Rube Goldberg machine competition. These Rube Goldberg machine files were created to work properly with Blender version 2.43, and because of the sensitivity of the physics simulations, they may not work properly in other versions, or on platforms other than the one they were created on. If the Rube Goldberg machines in the files are not working as you think they should, you might want to download and install Blender 2.43 from www.blender.org, where previous versions of Blender are always available.

Also on the CD, you'll find a file submitted by Blender user Kawl Nevina illustrating an interesting approach to making noncolliding joints. Nevina takes advantage of the fact that Static Triangle Mesh objects cannot collide with each other, and parents them to dynamic, jointed compound objects. Also in that file is an interesting example of an inner tube filled with balls. These you can find in the file ultraoll.blend. Change scenes in the file to see the different examples.

As you've seen by now, there are a variety of ways to use physics to bring the effect of life (or, in the case of ragdolls, lifelessness!) to inanimate objects. In the next chapter, you'll look at a totally different kind of simulation, not of physics, but of the very processes of life itself.