6
Events, Triggers, and Explosions

Almost every game needs some form of event system that informs the game logic about collisions that have occurred between objects, and many of these events are triggered by invisible volumes of space that react when certain game objects enter them. In this chapter, we'll learn how to build these features and then apply them by simulating an explosion!

Building a collision event system

In a game such as Angry Birds, we would want to know when a breakable object such as a pig or piece of wood has collided with something, so that we can determine the amount of damage that was dealt, and whether or not the object should be destroyed, which in turn spawns some particle effects and increments the player's score.

It's the game logic's job to distinguish between the objects, but it's the physics engine's responsibility to send these events in the first place and then we can extract this information from Bullet through its **persistent manifolds**.

Continue from here using the `Chapter6.1_CollisionEvents` project files.

Explaining the persistent manifolds

Persistent manifolds are the objects that store information between pairs of objects that pass the broad phase. If we remember our physics engine theory from *Chapter 3*, *Physics Initialization*, the broad phase returns a shortlist of the object pairs that might be touching, but are not necessarily touching. They could still be a short distance apart from one another, so the existence of a manifold does not imply a collision. Once you have the manifolds, there's still a little more work to do to verify if there is a collision between the object pair.

One of the most common mistakes made with the Bullet physics engine is to assume that the existence of a manifold is enough to signal a collision. This results in detecting collision events a couple of frames too early (while the objects are still approaching one another) and detecting separation events too late (once they've separated far enough away that they no longer pass the broad phase). This often results in a desire to blame Bullet for being sluggish, when the fault lies with the user's original assumptions. Be warned!

Manifolds reside within the collision dispatcher (a core Bullet object we created back in *Chapter 3*, *Physics Initialization*), and Bullet keeps the same manifolds in memory for as long as the same object pairs keep passing the broad phase. This is useful if you want to keep querying the same contact information between pairs of objects over time. This is where the persistent part comes in, which serves to optimize the memory allocation process by minimizing how often the manifolds are created and destroyed.

Bullet is absolutely riddled with subtle optimizations and this is just one of them. This is all the more reason to use a known good physics solution like Bullet, instead of trying to take on the world and building your own!

The manifold class in question is `btPersistentManifold` and we can gain access to the manifold list through the collision dispatcher's `getNumManifolds()` and `getManifoldByIndexInternal()` functions.

Each manifold contains a handful of different functions and member variables to make use of, but the ones we're most interested in for now are `getBody0()`, `getBody1()`, and `getNumContacts()`. These functions return the two bodies in the object pair that passed the broad phase, and the number of contacts detected between them. We will use these functions to verify if a collision has actually taken place, and send the involved objects through an event.

Managing the collision event

There are essentially two ways to handle collision events: either send an event every update while two objects are touching (and continuously while they're still touching), or send events both when the objects collide and when the objects separate.

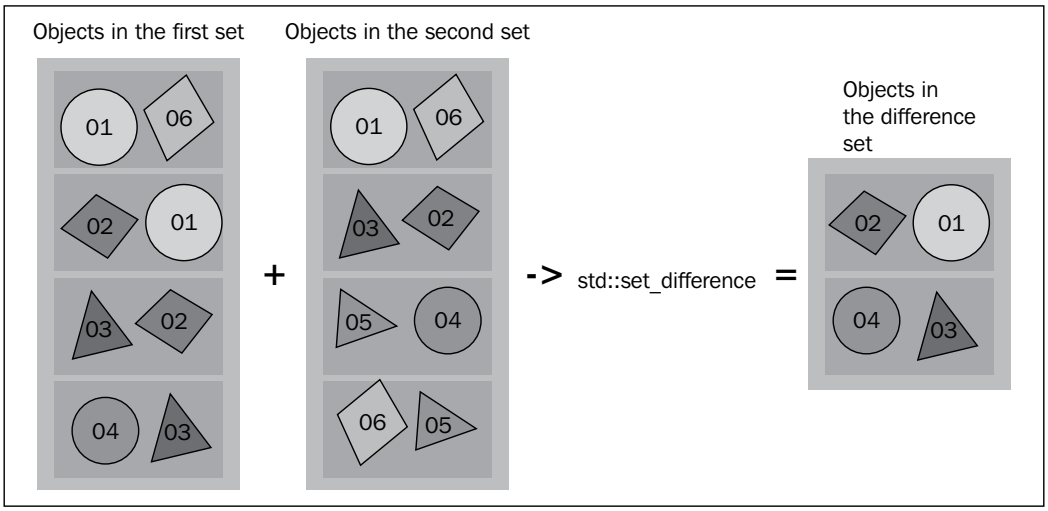
In almost all cases it is wiser to pick the latter option, since it is simply an optimized version of the first. If we know when the objects start and stop touching, then we can assume that the objects are still touching between those two moments in time. So long as the system also informs us of peculiar cases in separation (such as if one object is destroyed, or teleports away while they're still touching), then we have everything we need for a collision event system.

Bullet strives to be feature-rich, but also flexible, allowing us to build custom solutions to problems such as this; so this feature is not built into Bullet by default. In other words, we will need to build this logic ourselves. Our goals are simple; determine if a pair of objects have either collided or separated during the step, and if so, broadcast the corresponding event. The basic process is as follows:

1. For each manifold, check if the two objects are touching (the number of contact points will be greater than zero).
2. If so, add the pair to a list of pairs that we found in this step.
3. If the same pair was not detected during the previous step, broadcast a collision event.
4. Once we've finished checking the manifolds, create another list of collision objects that contains only the missing collision pairs between the previous step and this step.
5. For each pair that is missing, broadcast a separation event.
6. Overwrite the list of collision pairs from the previous step, with the list we created for this step.

There are several **STL** (**Standard Template Library**) objects and functions we can use to make these steps easier. An `std::pair` can be used to store the objects in pairs, and can be stored within an `std::set`. These sets let us perform rapid comparisons between two sets using a helpful function, `std::set_difference()`. This function tells us the elements that are present in the first set, but not in the second.

The following diagram shows how `std::set_difference` returns only objects pairs that are present in the first set, but missing from the second set. Note that it does not return new object pairs from the second set.



The most important function introduced in this chapter's source code is `CheckForCollisionEvents()`. The code may look a little intimidating at first, but it simply implements the steps listed previously. The comments should help us to identify each step.

When we detect a collision or separation, we will want some way to inform the game logic of it. These two functions will do the job nicely:

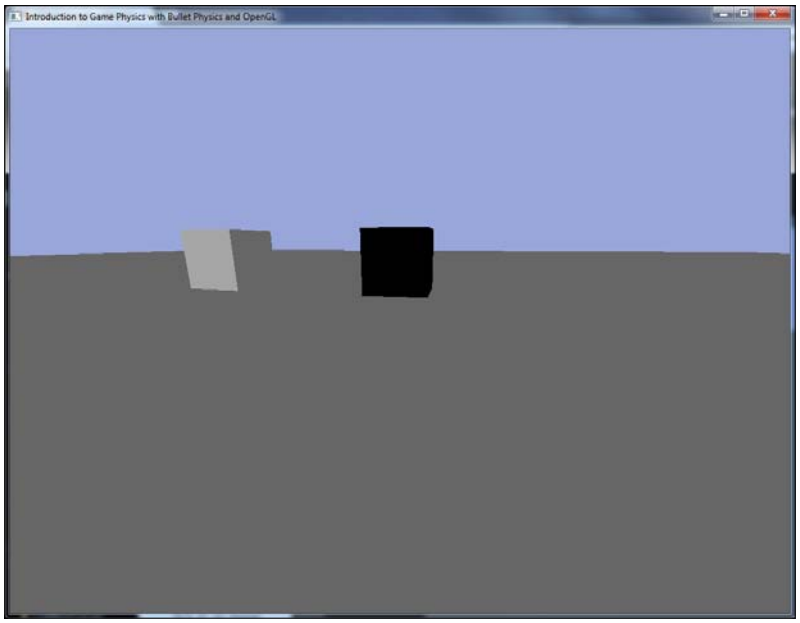
```
virtual void CollisionEvent(btRigidBody* pBody0, btRigidBody *
  pBody1);
virtual void SeparationEvent(btRigidBody * pBody0, btRigidBody *
  pBody1);
```

In order to test this feature, we introduce the following code to turn colliding objects white (and similar code to turn separating objects black):

```
void BulletOpenGLApplication::CollisionEvent(const
  btCollisionObject * pBody0, const btCollisionObject * pBody1) {
  GameObject* pObj0 = FindGameObject((btRigidBody*)pBody0);
  pObj0->SetColor(btVector3(1.0,1.0,1.0));
  GameObject* pObj1 = FindGameObject((btRigidBody*)pBody1);
  pObj1->SetColor(btVector3(1.0,1.0,1.0));
}
```

Note that these color changing commands are commented out in future project code.

When we launch the application, we should expect colliding and separating objects to change to the colors give in `CollisionEvent()`. Colliding objects should turn white, and separated objects should turn black. But, when objects have finished moving, we observe something that might seem a little counterintuitive. The following screenshot shows the two objects colored differently once they come to rest:



But, if we think about the order of events for a moment, it begins to make sense:

- When the first box collides with the ground plane, this turns both objects (the box and the ground plane) white.
- The second box then collides with the first turning the second box white, while the first box stays white.
- Next, the second box separates from the first box, meaning both objects turn black.
- Finally, the second box collides with the ground plane, turning the box white once again.

What was the last color that the first box turned to? The answer is black, because the last event it was involved in was a separation with the second box. But, how can the box be black if it's touching something? This is an intentional design consequence of this particular style of collision event management; one where we only recognize the collision and separation events.

If we wanted objects to remember that they're still touching something, we would have to introduce some internal method of counting how many objects they're still in contact with, and incrementing/decrementing the count each time a collision or separation event comes along. This naturally consumes a little memory and processing time, but it's certainly far more optimized than the alternative of spamming a new collision event every step while two objects are still touching. We want to avoid wasting CPU cycles telling ourselves information that we already know.

The `CollisionEvent()` and `SeparationEvent()` functions can be used by a game logic to determine if, when, and how two objects have collided. Since they hand over the rigid bodies involved in the collision, we can determine all kinds of important physics information, such as the points of contact (where they hit), and the difference in velocity/impulse force of the two bodies (how hard they hit). From there we can construct pretty much whatever physics collision-related game logic we desire.

Try picking up, or introducing more objects with the left/right mouse buttons, causing further separations and collisions until you get a feel for how this system works.

Building trigger volumes

Imagine we want an invisible volume of space, and when the player stumbles into it, it triggers a trap or a cutscene. This concept is used countlessly throughout modern games (in fact, it's difficult to think of one in the last decade that doesn't use them somewhere).

This effect is achieved in Bullet by simply disabling the contact responses for any given rigid body.

Continue from here using the `Chapter6.2_TriggerVolumes` project files.

Disabling contact response

There is no specific class required to build a trigger volume, but there is an essential flag which we can apply to any object: `CF_NO_CONTACT_RESPONSE`. This flag disables all contact response, informing Bullet that it should not calculate any physical response when other objects collide with the flagged object. This does not prevent it from performing broad and narrow phase collision detection and informing us when an overlap occurs, hence our `CollisionEvent()` and `CollisionSeparation()` functions will still be called even for objects flagged in this way. The only difference is that other objects will pass through it unhindered.

Here's a snippet of code from `BasicDemo::CreateObjects()`:

```
// create a trigger volume
m_pTrigger = new btCollisionObject();
// create a box for the trigger's shape
m_pTrigger->setCollisionShape(new btBoxShape(btVector3(1,0.25,1)));
// set the trigger's position
btTransform triggerTrans;
triggerTrans.setIdentity();
triggerTrans.setOrigin(btVector3(0,1.5,0));
m_pTrigger->setWorldTransform(triggerTrans);
// flag the trigger to ignore contact responses
m_pTrigger->setCollisionFlags(btCollisionObject::CF_NO_CONTACT_
RESPONSE);
// add the trigger to our world
m_pWorld->addCollisionObject(m_pTrigger);
```

The previous code creates a trigger volume hovering just above the ground plane. We don't want these trigger volumes to be rendered during runtime since these kinds of triggers usually remain invisible to the player. So we avoided using our `CreateGameObject()` function (which would have added it to the list of objects and automatically render it), and instead we built it manually.

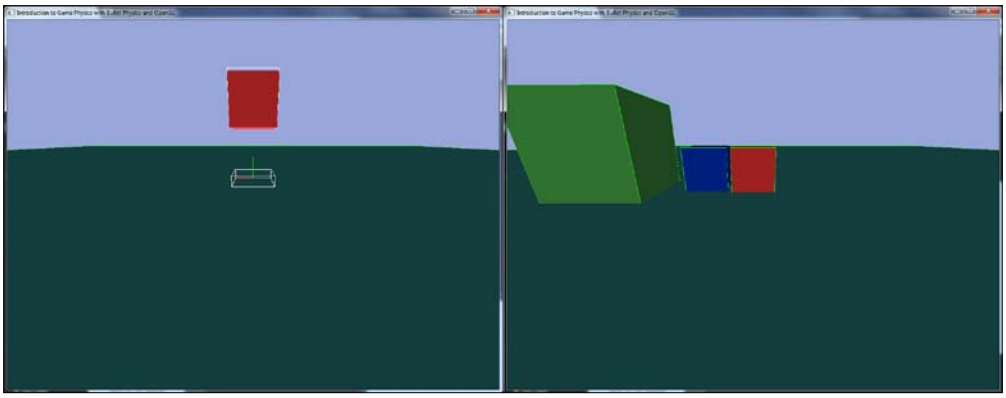
However, even though it is invisible to the player, we can still observe it through the debug renderer. If we enable wireframe mode (the *W* key), Bullet will draw the shape for us so that we can visualize the trigger volume in the space.

Meanwhile, `BasicDemo` includes an override for `CollisionEvent()` which checks if the two objects involved are the box and the trigger, and if so, it spawns a large box besides it. Note that we don't necessarily know if `pBody0` or `pBody1` represents either object, so we need to check both pointers:

```
void BasicDemo::CollisionEvent(btRigidBody* pBody0, btRigidBody*
pBody1) {
```

```
    // did the box collide with the trigger?
    if (pBody0 == m_pBox->GetRigidBody() && pBody1 == m_pTrigger ||
      pBody1 == m_pBox->GetRigidBody() && pBody0 == m_pTrigger) {
      // if yes, create a big green box nearby
        CreateGameObject(new btBoxShape(btVector3(2,2,2)), 2.0,
btVector3(0.3, 0.7, 0.3), btVector3(5, 10, 0));
    }
  }
```

Launch the application, and enable wireframe debugging (the *W* key). We should see a trigger volume (denoted by a white wireframe) just below the spawn point of the first box. Moments after, the box should collide with the trigger, causing `CollisionEvent()` to be called. Since the two objects involved are the trigger volume, and the first box, the `if` statement will become true, and a new game object will be created. The following screenshot shows a new object (the large box) being spawned after the first box collides with the trigger volume:



Force, torque, and impulse

Next, we will explore how to manipulate the motion of our collision objects through forces, torques, and impulses and also discuss the important differences between them.

Continue from here using the `Chapter6.3_ForceTorqueAndImpulse` project files.

Understanding the object motion

We have already observed one method of moving a rigid body with our `ShootBox()` command back in *Chapter 5*, *Raycasting and Constraints*, by calling the `setLinearVelocity()` function on the object's rigid body after creating it. This function sets the magnitude and direction of the object's linear motion. Meanwhile, another commonly used motion altering command is `setAngularVelocity()`, which is used to set the object's rotational velocity.

However, simple velocity altering commands like these do not add much life or believability to a scene, since we humans are also familiar with the concept of acceleration due to effects such as gravity, or the inertia we feel when we drive a car, ride a bike, or even walk. Acceleration can be applied in Bullet through the use of forces. There are different types of force, where each one has some important distinctions that must be understood before making use of them. We'll discuss the following commands that are accessible through any `btRigidBody`:

- `applyForce()`
- `applyTorque()`
- `applyImpulse()`
- `applyTorqueImpulse()`
- `applyCentralForce()`
- `applyCentralImpulse()`

All of the preceding functions require a `btVector3` object to define the direction and the strength of the effect. Just like the Newtonian definition, forces (such as gravity) continuously accelerate an object in a given direction, but do not affect their rotation. Meanwhile, torque is the rotational equivalent of a force, applying a rotational acceleration to an object causing it to rotate in place around its center of mass. Hence, `applyForce()` and `applyTorque()` provide the means for applying these effects, respectively.

Meanwhile, the difference between forces and impulses is that impulses are forces that are independent of time. For instance, if we applied a force to an object for a single step, the resultant acceleration on that object would depend on how much time had passed during that step. Thus, two computers running at slightly different time steps would see two completely different resultant velocities of the object after the same action. This would be very bad for a networked game, and equally bad for a single player game that suffered a sudden spike in activity that increased the step time temporarily.

However, applying an impulse for a single step would give us the exact same result on both computers because the resultant velocity is calculated without any dependence on time. Thus, if we want to apply an instantaneous force, it is better to use `applyImpulse()`. Whereas, if we want to move objects over several iterations, then it is better to use `applyForce()`. Similarly, applying a *Torque Impulse* is an identical concept, except it applies a rotational impulse. Hence, we would use `applyTorqueImpulse()` if we wanted an instantaneous rotational kick.

Finally, the difference between `applyCentralForce()` and `applyForce()` is simply that the former always applies the force to the center of mass of the object, while the latter requires us to provide a position relative to the center of mass (which could always default to the center of mass, anyway). Basically, the `Central` functions are there for convenience, while the rest are more flexible since in the real world if we pushed a box on its edge we would expect it to move linearly (force), but also rotate a little (torque) as it moved. The same distinction applies to `applyCentralImpulse()` and `applyImpulse()`.

Knowing all of this, if we follow the pattern of function names we may notice that `applyCentralTorque()` is missing. This is because there's no such thing in the laws of physics. A torque must always be applied at an offset from the center of mass, since a central torque would simply be a linear force.

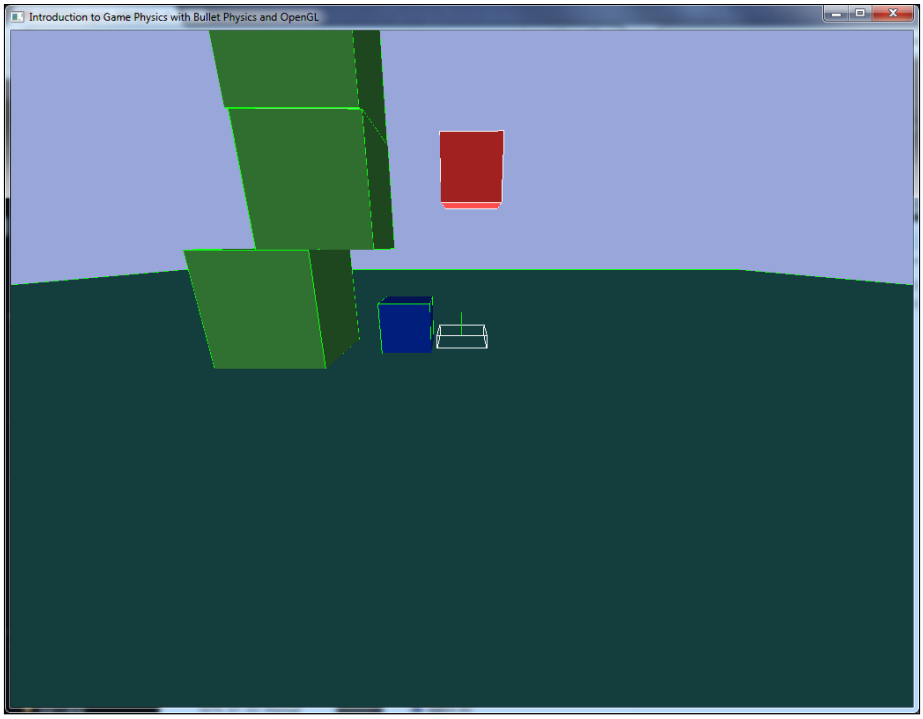
Applying forces

In the source code for this section, `BasicDemo` has been modified to grab the *G* key, and apply a force of 20 units in the y axis to the first box (the red one). This is strong enough to counteract the force of gravity (default of `-10` in the y axis), and cause our object to accelerate upwards while the key is held down.

Check the `Keyboard()`, `KeyboardUp()`, and `UpdateScene()` functions of `BasicDemo` to see this process in action.

Note that each of the override functions used in this process begins by calling back to the base class implementation of the same function. This ensures that our base class code, which handles keyboard input and scene updating, is still called before we do anything unique in our derived class.

Launch our application and try pressing and holding the *G* key. Our first box should now begin to float. The following screenshot shows how our first box can be lifted up, land back in the trigger volume, and summon even more boxes:



Also note that the lifted box may seem to rotate and veer off-course slightly even though we're always applying an upward force. Two effects contribute to this: the natural inaccuracy of floating point numbers and subtle differences in contact responses on each of the different vertices when the box hits the ground.

Applying impulses

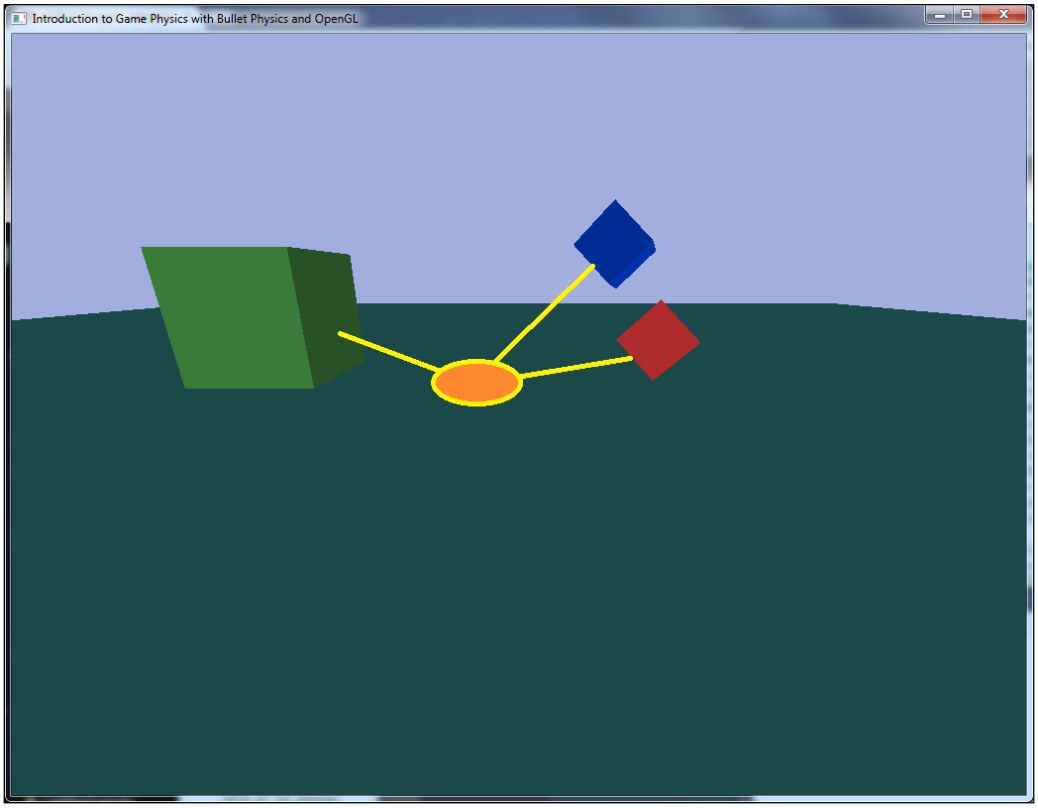
Next, we'll work through an example of an impulse by creating a small explosive force at the location of the mouse cursor. In order to simulate an explosion, we will need to create a spherical trigger volume, instead of a box (since a box-shaped explosion would be really weird). When collisions are detected with this volume we can apply an impulse that points from the center of the explosion towards the target object(s), forcing the object away from its epicenter. However, we only want this object to linger for a single simulation step, so that we can tightly control the amount of acceleration applied to the target objects.

Since we want our explosion to be generated only temporarily when a key is pressed, this presents a problem when we interact with the `Keyboard()` command, since it is called once when a key is pressed, and continuously while the key is still held down. It's possible to tweak our input system to not repeat calls like this with a FreeGLUT function call (as mentioned previously in *Chapter 1*, *Building a Game Application*), but our camera moving code currently depends on the current style, so changing it now would cause a different set of problems.

So, what we can do is use a simple `Boolean` flag that tells us if we can create an explosion object. When we want to create an explosion, we will check if the flag is true. If so, we create the explosion and set the flag to false, and we will not set the flag back to true again until the key is released. This prevents subsequent calls to the `Keyboard()` function from creating another explosion trigger volume unless we detect a key up event.

This is a fairly straightforward process, and the source code for this chapter adds the relevant code to produce this effect with slight tweaks to the `Keyboard()`, `KeyboardUp()`, `UpdateScene()`, and `CollisionEvent()` functions of `BasicDemo`. The 3D math implemented in the code uses some simple vector arithmetic to obtain the final direction by converting the vector between them into a unit vector, and obtaining the final magnitude from the distance between the objects and some constant value (`EXPLOSION_STRENGTH`). With a direction and a magnitude, we can create our final impulse vector.

Launch the application, place the mouse cursor somewhere near the boxes, and press the *E* key. This will result in an invisible explosion that pushes all the nearby objects away through a simple impulse force. The following screenshot shows what happens when an explosion is generated between the boxes (epicenter and direction of travel added for artistic flair):



Note that to simulate the explosion a little more realistically, the strength of the explosion follows an inverse law, since we wouldn't expect an object further from the center to experience the same impulse as those that are near.

```
if (dist != 0.0) strength /= dist;
```

Also note that an additional parameter was added to the `Raycast()` function to allow us to decide whether we want it to return collisions with static objects or not (like the ground plane). These were ignored originally because we didn't want our `DestroyGameObject()` function to destroy the ground plane. But, now we need this special case in order to generate an explosion somewhere on the ground plane's edge; otherwise it would simply ignore them and we could only generate explosions on the edges of the boxes. It's set to `false` by default, to spare us from having to edit our existing calls to `Raycast()`.

```
bool Raycast(const btVector3 &startPosition, const btVector3
  &direction, RayResult &output, bool includeStatic = false);
```

Summary

Very little game logic can be built around a physics engine without a collision event system, so we made Bullet broadcast collision and separation events to our application so that it can be used by our game logic. This works by checking the list of manifolds, and creating logic that keeps track of important changes in these data structures.

Once we have these collision events, we need to do something with them, and we explored how to use a collision event between a box and an invisible trigger volume to instantiate a new object in our world, and how to capture these events within an instant of time when an explosion is generated.

In the next chapter, we will explore some of the more unusual types of collision shapes offered by Bullet.