3
Physics Initialization

In this chapter, we will discover how to initialize the Bullet library, and learn how to build our first physical rigid body object, which is the simplest object available in Bullet.

The core bullet objects

Bullet is designed from the ground up to be highly customizable. Each major task that the physics engine performs is isolated into its own modular component, allowing them to be replaced with ease as long as they inherit from the appropriate interface/base classes.

Most applications, such as ours, can make do with a generic, one-size-fits-all selection of components, but if we ever find ourselves needing something more advanced, optimized, or technically superior, then there is nothing stopping us from interchanging the components or even building our own.

There are a handful of components that need to be created and hooked together in order to initialize Bullet. We'll cover some essential theory on each of these components, and then run through the code to create/destroy them.

Continue from here using the `Chapter3.1_TheCoreBulletObjects` project files.

The world object

The primary control object for a Bullet physics simulation is an instance of `btDynamicsWorld`. All of our physical objects will be controlled by the rules defined by this class. There are several types of `btDynamicsWorld` that can be used, depending on how you want to customize your physics simulation, but the one we will be using is `btDiscreteDynamicsWorld`. This world moves objects in discrete steps (hence the name) in space as time advances.

This class doesn't define how to detect collisions, or how objects respond to collisions. It only defines how they move in response to stepping the simulation through time.

The broad phase

A physics simulation runs in real time, but it does so in discrete steps of time. Each step, there would be some number of objects which may have moved a small distance based on their motion and how much time has passed. After this movement has completed, a verification process checks whether a collision has occurred, and if so, then it must generate the appropriate response.

Generating an accurate collision response alone can be highly computationally expensive, but we also have to worry about how much time we spend checking for collisions in the first place. The brute force method is to make sure that no collisions have been missed by comparing every object against every other object, and finding any overlaps in space, and doing this *every* step.

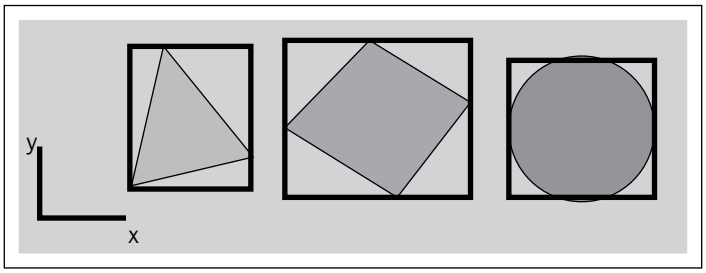
This would be all well and good for simple simulations with few objects, but not when we potentially have hundreds of objects moving simultaneously, such as in a videogame. If we brute force our way through the collision checks, then we need to check all the `N` objects against the other `N-1` objects. In Big O notation this is an $O(N^2)$ situation. This design scales badly for increasing values of `N`, generating an enormous performance bottleneck as the CPU buckles under the strain of having so much work to do every step. For example, if we have 100 objects in our world, then we have 100*99 = 9,900 pairs of objects to check!

Brute forcing one's way through physics collision detection is typically the first performance bottleneck an inexperienced game programmer comes across. Understanding what's happening, and how to optimize these kinds of bulk processes is a key component in becoming an effective game developer.

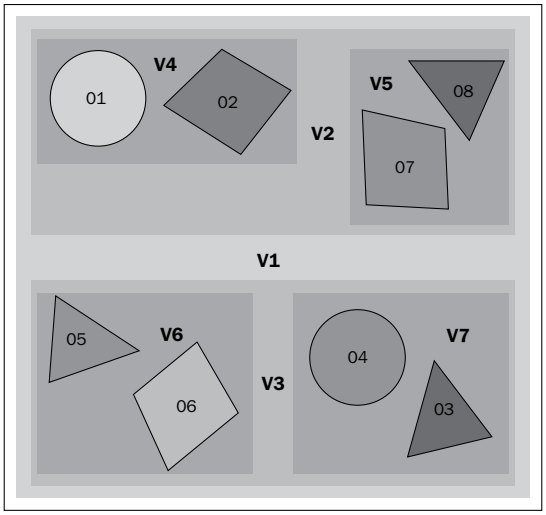
But, imagine if only two of those 100 objects are even remotely close together and the rest are spread too far apart to matter; why would we waste the time doing precise collision checks on the other 9,899 pairs? This is the purpose of **broad phase collision detection**. It is the process of quickly culling away object pairs, which have little or no chance of collision in the current step, and then creating a shortlist of those that could collide. This is an important point because the process merely provides a rough estimate, in order to keep the mathematics computationally cheap. It does not miss any legitimate collision pairs, but it will return some that aren't actually colliding.

Once we have shortlisted the potential collisions, we pass them on to another component of the physics simulation called narrow phase collision detection, which checks the shortlist for legitimate collisions using more intense, but accurate mathematical techniques.

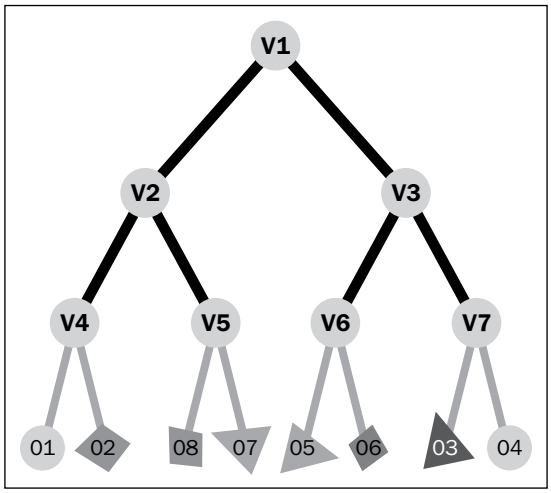
For our project we will use a broad phase technique based on dynamic bounding volumes. This algorithm create volumes of space which envelop pairs of objects in a tree hierarchy using **Axis-aligned bounding boxes** (**AABBs**). These AABBs surround the object with the smallest box shaped volume possible, that is aligned with each axis, as we can see here:



It is a relatively cheap task to check if two AABBs overlap, but we can still overburden ourselves if we don't perform some additional optimization. By organizing the objects by pairs in a tree structure, we naturally organize our objects by distance from one another, thus automatically culling away the object pairs which are too far apart to be worth checking, as we can see here:



It takes some processing work to maintain the tree hierarchy as objects move around, since the AABBs have to be dynamically altered on occasion. But, this is much less expensive than performing AABB overlap checks on every pair, every iteration. You may recognize this tree structure as a simple binary tree:



This is no coincidence. The intent is to use the simplicity and speed of searching a binary tree in order to quickly asses which objects are closest to others.

A `btBroadPhaseInterface` object is needed to tell our world object what technique to use for its broad phase collision detection and the built-in type we will be using is `btDbvtBroadphase`.

The collision configuration

This is a relatively simple component on the surface, but under the hood it provides the physics simulation with components that handle essential tasks such as determining how Bullet manages memory allocation, provides the algorithms for solving various collisions (box-box, sphere-box, and so on), and how to manage the data that comes out of the broad phase collision detection called **Manifolds** (we will explore these in *Chapter 6*, *Events, Triggers, and Explosions*).

For this project, we'll keep things simple and use Bullet's default collision configuration object, `btDefaultCollisionConfiguration`.

The collision dispatcher

The **collision dispatcher**, as the name implies, dispatches collisions into our application. For a video game, it is practically guaranteed that we will want to be informed of inter-object collision at some point, and this is the purpose of the collision dispatcher.

One of the built-in collision dispatcher class definitions that come with Bullet is the basic `btCollisionDispatcher`. The only requirement is that it must be fed with the collision configuration object in its constructor (which forces us to create this object second).

The constraint solver

The constraint solver's job is to make our objects respond to specific constraints. We will learn more about the constraints in *Chapter 5*, *Raycasting and Constraints*. We will be using `btSequentialImpulseConstraintSolver` for our project.

Note that our application class will be derived from and customized in the `BasicDemo` class for the next several chapters. This keeps our application layer code isolated from our physics/game logic.

Each of the components described previously can be customized to fit our needs; for instance, we might be working within extremely tight memory requirements (such as a mobile device), and so we might consider completely replacing the stack allocator with our own to optimize Bullet's memory allocation processes.

Creating the Bullet components

Contrary to what the laborious explanations in the previous section might have you believe, creating the necessary Bullet objects is relatively simple. Our application layer class contains a handful of pointers that all the derived classes can use:

```
btBroadphaseInterface* m_pBroadphase;
btCollisionConfiguration* m_pCollisionConfiguration;
btCollisionDispatcher* m_pDispatcher;
btConstraintSolver* m_pSolver;
btDynamicsWorld* m_pWorld;
```

Meanwhile, the code to initialize Bullet can be found in `BasicDemo` and looks as shown in the following code snippet:

```
m_pCollisionConfiguration = new btDefaultCollisionConfiguration();
m_pDispatcher = new
  btCollisionDispatcher(m_pCollisionConfiguration);
m_pBroadphase = new btDbvtBroadphase();
m_pSolver = new btSequentialImpulseConstraintSolver();
m_pWorld = new btDiscreteDynamicsWorld(m_pDispatcher,
  m_pBroadphase, m_pSolver, m_pCollisionConfiguration);
```

Creating our first physics object

Bullet maintains the same modular design of its core components even down to individual physics objects. This allows us to customize physics objects through their components by interchanging or replacing them at will.

Three components are necessary to build a physics object in Bullet:

- A **collision shape**, defining the object's volume and how it should respond to the collisions with other collision shapes
- A **motion state**, which keeps track of the motion of the object
- A **collision object**, which acts as a master controller of the object, managing the previously mentioned components and the physical properties of the object

We'll cover some essential theory on these components before we build one in code. We'll also make some changes to our rendering system so that we can observe the effects of gravity on our object.

Continue from here using the `Chapter3.2_CreatingOurFirstPhysicsObject` project files.

The collision shape

The collision shape represents the volume of an object in space, be it a box, a sphere, a cylinder, or some other more complex shape. Collision shapes do not care about the rules of the world. They only care about how they should interact with other shapes, and so it is the collision shape's responsibility to inform Bullet what kind of shape it is, so that it can respond appropriately.

We will cover more varieties and intricacies of collision shapes in *Chapter 7*, *Collision Shapes*, but for now we will build a simple collision shape component using `btBoxShape`. This object's only requirement is to define its size upon creation.

As an interesting side note, spheres take a lot of polygons in order to generate an accurate graphical representation of them. But, the physical representation of a sphere is little more than a position and a radius, and calculating a sphere-to-sphere collision is very simple (check the distance between their centers against the sum of their radii). Meanwhile, a box is the exact opposite; they're cheap to generate graphically (only 12 triangles), but expensive to generate physically and requires much more complex mathematics to resolve collisions between them. Because of this, spheres are commonly used for the collision shape of an object, even if its graphical representation is not a sphere.

In addition, a newbie physics programmer would typically start out by using spheres to represent the bounding volumes for objects to generate their very first broad phase system. But, they will later graduate to using AABBs (similar to those described previously) as they find that the spheres are not very good at representing long, thin objects, and the mathematics aren't quite as efficient as AABB overlap checks. Even though AABBs are technically boxes, they don't rotate (since the AA stands for Axis-aligned), making the overlap math very simple—even simpler than comparing two spheres for overlap.

The motion state

The motion state's job is to catalogue the object's current position and orientation. This lets us to use it as a hook to grab the object's transformation matrix (also known as a transform). We can then pass the object's transform into our rendering system in order to update the graphical object to match that of the physics system.

This is an incredibly important point that cannot be ignored, forgotten, or otherwise misplaced; Bullet does not know, nor does it care, how we render its objects. We could add 100 physics objects into our application right now, and Bullet would move them, detect collisions, and resolve them as we would expect a physics engine to do; but unless we tell our OpenGL code to draw a graphical box object in the same location, we will have no idea about what's going on (besides doing some step through debugging and scanning the code, of course). Our physics and graphics engines are completely isolated from one another, and they have different ways of representing the same object.

Having no dependency between our graphics and physics is ideal because it means that we could completely replace the physics engine without having to touch the graphics engine, and vice versa. It also means that we can have invisible physics objects (such as force fields), or graphical objects that don't need a physical presence (such as particle effects).

It is not uncommon for a game engine to separate these components entirely with three different libraries, resulting in the three different sets of `Vector3/Matrix4x4/Quaternion` classes in the lowest levels; one set for the physics, one set for the graphics, and one set for general game logic.

As an example of extending Bullet, we will be creating our own motion state class called `OpenGLMotionState`. This class will extend Bullet's `btDefaultMotionState` to provide a useful helper function that simplifies the process of extracting the objects transform data into a format our rendering system can use.

The collision object

Collision objects are the essential building blocks of our physics objects, since they maintain the object's physical properties, and give us a hook from which to alter the object's velocity, acceleration, apply a force upon it, and so on. When we query the motion state for the transform, it actually comes to this object to obtain it.

The simplest and most commonly used type of collision object is a `btRigidBody`. Rigid bodies are physics objects that do not deform as a result of collisions, as opposed to soft bodies which do. Rigid bodies are the easiest collision objects to deal with because their behavior is consistent, and doesn't require extravagant mathematics to handle basic collisions, unlike soft bodies which are far more difficult and expensive to simulate.

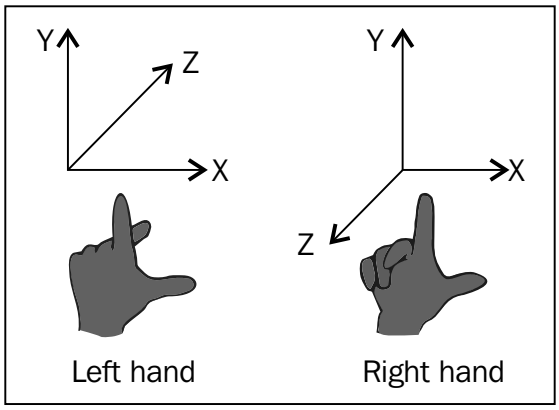
Rigid bodies also require a `btRigidBodyConstructionInfo` object to be passed through their constructor. This object stores data of important physical properties for the rigid body, such as mass, friction, restitution, and so on.

Building a custom motion state

The entire code for our custom motion state can be found in a single header file `OpenGLMotionState.h`. The only interesting function is `GetWorldTransform()`, which takes an array of `btScalars` (16 of them to be precise, representing a 4 x 4 matrix), and performs a little math to return the same data in a format that OpenGL understands. `getOpenGLMatrix()` is a helper function built into `btTransform` that does this for us. OpenGL and Bullet are used together so often (*the* open source graphics library used together with *the* open source physics engine; who would have guessed?) that the developers of Bullet felt it was prudent to do this.

`btScalar` is a simple `float` by default, but could also be a `double` if `#define BT_USE_DOUBLE_PRECISION` is placed somewhere in the code. We'll continue to use floats for this project.

It is a clean and efficient process to feed data between Bullet and OpenGL because they both use right-handed coordinate systems, which defines how the **x**, **y**, and **z** axes relate to one another. If we used a different physics and/or graphics library, we might find that our objects move or render backwards on one of the axes. In that case we may have a disconnection between our coordinate systems, and we would need to determine which axis has been flipped, and make the necessary adjustments. The following diagram shows the difference between the left-handed and right-handed coordinate systems:



Creating a box

Creating a box-shaped rigid body is fairly simple; all of the code to create one can be found in the `CreateObjects()` function of this chapter's source code. We simply create the three modular components described previously (motion state, collision shape, and collision object), hook them together, and then inform the world object of its existence through `addRigidBody()`. The only awkward step is of using `btRigidBodyConstructionInfo`. This object is an intermediate step in creating `btRigidBody` and requires the mass, motion state, and collision shape objects before it can be built, although it has other properties that can be modified such as the coefficients of restitution (how much it bounces), and friction.

```
btRigidBody::btRigidBodyConstructionInfo rbInfo(1.0f,
  m_pMotionState, pBoxShape);
btRigidBody* pRigidBody = new btRigidBody(rbInfo);
```

Rendering from transform data

We still have the problem that our `DrawBox()` function always draws a box at (`0`,`0`,`0`) in world space. Before we can make use of the data from our object's motion state, we will have to modify our rendering system to draw our object at a given location instead. In order to do this, we'll need to introduce a few more OpenGL commands.

`glPushMatrix()` and `glPopMatrix()` are another pair of OpenGL delimiter functions that work together in much the same way as `glBegin()` and `glEnd()` do. They are used to control the **matrix stack**, which is very helpful while drawing multiple objects, and objects that are meant to be connected together. As mentioned previously, transformation matrices can be combined to get a resultant transformation, and if we have multiple objects that share a similar transformation, we can optimize our processing time by sharing information through the matrix stack, instead of recalculating the same value over and over again.

This feature is particularly useful when we have object hierarchies such as a knight riding on top of a steed, or moons orbiting planets, which themselves orbit stars. This is the basic concept of a Scene Graph in 3D rendering (which is beyond the scope of this book). The function to multiply the current matrix stack by a given matrix is `glMultMatrixf()`.

In this project, `DrawBox()` has been changed to collect an array of `btScalars` for the transform, and uses the methods explained previously to manipulate the matrix stack by surrounding the previous rendering code with calls to push and pop the stack.

Note that the `/*REM*/` comment tags in the source code represent code that has been removed and/or replaced since the previous section.

Stepping the simulation

So, now we're rendering a graphical box in the same place as our physical box, but we still don't see it moving. This is because the box isn't actually moving. Why? Because Bullet has not been told to step the simulation, yet!

In order to do this, we simply call `stepSimulation()` on our `btDynamicsWorld` object, providing the number of seconds that have elapsed since the last iteration. The challenge here is counting the amount of time that has passed, since the last time we called `stepSimulation()`.

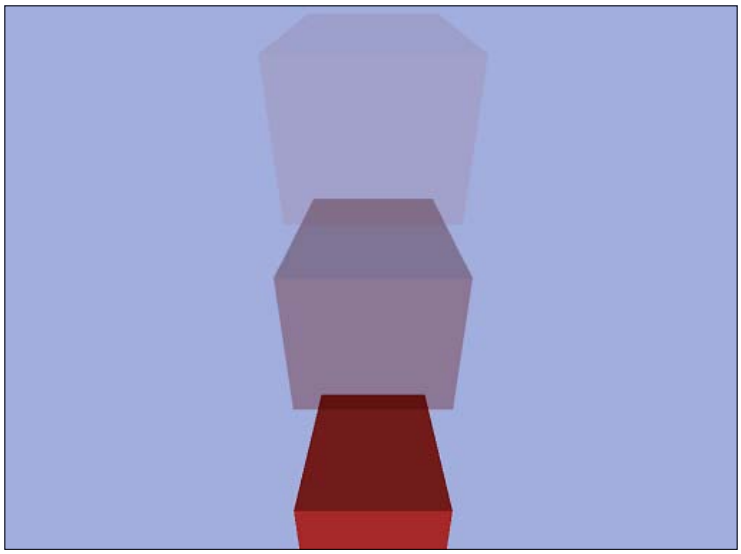
Bullet comes with a built-in `btClock` object, but if you have your own clock tool in mind, which you trust to be more precise, then there's nothing stopping you from using that for a counter instead. A good place to handle this logic is in the application layer class with the following member variable:

```
btClock m_clock;
```

The clock can be used to calculate the time since the last step and updating the application.

```
float dt = m_clock.getTimeMilliseconds();
m_clock.reset();
UpdateScene(dt / 1000.0f);
```

When we launch our application again, we should observe our box falling under the effects of gravity as in the following screenshot:



The `stepSimulation()` function also accepts two more parameters; the second parameter is the maximum number of substeps that can be calculated this iteration, and the third is the desired frequency of step calculations.

If we ask Bullet to perform calculations at a rate of 60 Hz (the third parameter – 60 Hz is also the default), but one second has gone by (maybe our application froze for a moment), Bullet will make 60 separate step calculations before returning. This prevents Bullet from jumping every object for one full second in time all at once, and possibly missing some collisions.

However, calculating so many iterations at once could take a long time to process, causing our simulation to slow down for a while as it tries to catch up. To solve this, we can use the second parameter to limit the maximum number of steps it's allowed to take in case one of these spikes occur. In this case, the world's physics will appear to slow down, but it also means that your application won't suffer from a long period of low frame rates.

In addition, the function returns the number of actual steps that took place, which will either be the maximum, or less if it processed everything quickly enough.

Summary

We've created the essential components that Bullet needs to initialize, and hooked them all together. We then created our first object in Bullet and extended our rendering system to render the object as it moves through space. To do this we created a custom class, `OpenGLMotionState`, which extends one of Bullet's own objects. This class simplifies the process of obtaining the correct OpenGL transform matrix from our object.

In the next chapter, we will implement a more robust system to handle multiple objects, and look into extracting useful physics debug information from Bullet.