# 4
# Object Management and Debug Rendering

Our current application is not very scalable, and does not provide much in the way of debug information to help us work with the physics system. So in this chapter, we will be performing some code refactoring to help us handle these situations in a robust fashion.

## Handling multiple objects

We're currently hard coding one pointer and making one call to `DrawBox()` to create our first object. Continuing down this path by adding a second pointer, and a second call to `DrawBox()` would make introducing even more objects into the scene an awkward and painful process.

A wise course of action to take at this early stage is to build a system that allows us to encapsulate important object data (such as the object's shape, collision shape, motion state, and color) in its own class. This way we can iterate through all of our objects, regardless of private information, and use common calls to update and render them.

In order to accomplish this, we will have to make five significant changes to our object system:

- Create a `GameObject` class to store our object's data
- Instantiate our objects from `GameObject` and store them in a data structure
- Modify our rendering system to iterate through the aforementioned data structure

- Have the rendering system detect the type of shape and render the appropriate polygons
- Write a new function to simplify the creation of objects

Continue from here using the `Chapter4.1_HandlingMultipleObjects` project files.

# Designing our objects

We would like to create simple objects of various shapes such as boxes, spheres, and cylinders, and do so without having to repeat tasks such as creating the motion state, the rigid body, the rigid body construction info, and so on. Essentially, the only data that should be unique between any two objects are:

- The mass
- The shape
- The color
- The initial position and rotation

Because of this, it should be possible to create new objects using only these four parameters, while we automatically build the rest from that information. Hence, the constructor for our new `GameObject` class looks as follows:

```
GameObject(btCollisionShape* pShape, float mass, const btVector3
  &color, const btVector3 &initialPosition = btVector3(0,0,0),
    const btQuaternion &initialRotation = btQuaternion());
```

`GameObject` is a new class that stores important personal data, such as the Bullet object components, and its own color. It also gives us a few helper functions to access this data if we need (and we will). The majority of the work happens in the constructor, requiring us to specify the object's mass, color, shape, and initial transform (in the form of a vector for position and a quaternion for rotation).

Most of the code should be straightforward, with the exception of local inertia. This value is normally referred to as the **moment of inertia** in most physics circles. It essentially boils down to the object's resistance to change in angular velocity around each axis of rotation. The classic example of this is a pole carried by a tight rope walker. Because the pole is very long, it resists rotation allowing them to use it to maintain balance.

We have a special case built in to this function. If we define the mass to be zero, we also set the local inertia to `(0,0,0)`. This will prevent the object from rotating as a result of collisions. In addition, Bullet uses a mass of zero to imply that you really want it to have an infinite mass, and hence don't want the object to move as result of forces like gravity. This is useful for environmental objects that don't move, such as a ground planes walls, and so on.

The **center of mass** (**COM**) of an object is an important property in Newtonian physics and it's just as important in physics simulations. Bullet will assume that the rigid body's COM is equal to its world position, unless told otherwise. Shifting the COM to some other location can be achieved through the usage of compound shapes, which will be explored more in *Chapter 7*, *Collision Shapes*.

# Rendering our objects

We also need a more universal system to render our objects: one that chooses a different rendering function based on the type of object being rendered. The source code for this chapter now features a new `DrawShape()` function, which accepts a `btCollisionShape` pointer of any derived type, determines which type of shape it is using some of Bullet's internal enumerators, and then renders the appropriate shape.

Since `DrawBox()` only covers boxes for now, we will be returning to this function later to expand on the additional shapes we introduce.

This change requires us to move the coloring and positioning of our objects (the `glColor3f()`, `glPushMatrix()`, and `glPopMatrix()` calls) out of `DrawBox()` and into `DrawShape()`, to spare us from duplicating the same instructions each time we render.

When a box is detected, through the `BOX_SHAPE_PROXYTYPE` enumerator, it gets the appropriate size for the box, by typecasting to `btBoxShape` and calling `getHalfExtentsWithMargin()`. This function returns the box's half size, or half the size in each direction. This value is convenient for collision-response mathematics, and since boxes are symmetrical shapes, physics engines typically define boxes this way.

# Storing our objects

Picking the correct data structure for our needs is a matter of determining our requirements and matching them to the properties of the various kinds of data structures available. Tasks such as updating/rendering require quick iteration, since there could be a very long list of world objects we need to check through. Meanwhile, we're not particularly worried about tasks such as random access, or insertion and deletion since these will be fairly rare tasks to perform.

For quick iteration, you can't do much better than one of the array based data structures (of which there are many). A dynamic array or vector (not to be confused with the 3D math variety of vector) should suit us well for our game objects. Although, keep in mind that dynamic arrays are not the most efficient at random access, insertion, or deletion.

There's a common saying in software engineering that *preoptimization is the root of all evil*. Don't spend too much time thinking about the perfect data structure to use (or heaven forbid, building it all from scratch). It's wise to just get your application working first, then go back and optimize if and only if you find out that they turn out to be a significant bottleneck once you begin profiling. You may find that your choice of data structure doesn't matter at all, because there could be bigger optimization problems popping up in areas you never even considered.

The **Standard Template Library** (**STL**) is a library that comes standard with most C++ compilers, particularly Visual Studio. This library is enormous and contains classes and tools to handle many different, common problems that C++ programmers come across. It also contains a collection of useful data structures that we can use to store the objects such as our `GameObject`. A simple STL vector should be sufficient for this task:

```
typedef std::vector<GameObject*> GameObjects;
GameObjects m_objects;
```

This allows us to refer to our vector using a custom `GameObjects` type, instead of using the ugly namespace/template combination each time.

Consequently this allows us to replace our main rendering code with the following:

```
void BulletOpenGLApplication::RenderScene() {
  // create an array of 16 floats (representing a 4x4 matrix)
  btScalar transform[16];
```

```
    // iterate through all of the objects in our world
    for(GameObjects::iterator i = m_objects.begin(); i !=
      m_objects.end(); ++i) {
      // get the object from the iterator
      GameObject* pObj = *i;
      // read the transform
      pObj->GetTransform(transform);
      // get data from the object and draw it
      DrawShape(transform, pObj->GetShape(), pObj->GetColor());
    }
  }
```

Having our objects stored in a data structure like this provides a lot more flexibility to add new objects to our scene without undue effort.

As an interesting side note, Bullet uses a concept called **Single Instruction, Multiple Data** (**SIMD**) on some platforms, which makes the ability to run the same instruction on multiple pieces of data very rapid. In order to use this, the objects must be aligned in memory in steps of 16 bytes, and an STL vector does not naturally do this.

However, there is a built-in object type in Bullet called `btAlignedObjectArray`, which functions similarly to an STL vector and is worth exploring if you wish to make use of performance enhancements like SIMD in the future. Iterating through a large list of game objects and performing updates on them is a perfect situation to apply this technique.

# Creating our objects

With our new system in place, we can introduce a new function to handle the creation of a `GameObject` for us. This function can be examined in the chapter's source code, but to summarize, it gathers the shape, mass, color, position, and rotation of the object we want to create, generates a new `GameObject` for it, and adds it to our dynamic array. As long as `GameObject` remains in the array, it will be automatically rendered in each step, greatly simplifying the act of adding new objects.

With this new function, we easily create our original box with the following call:

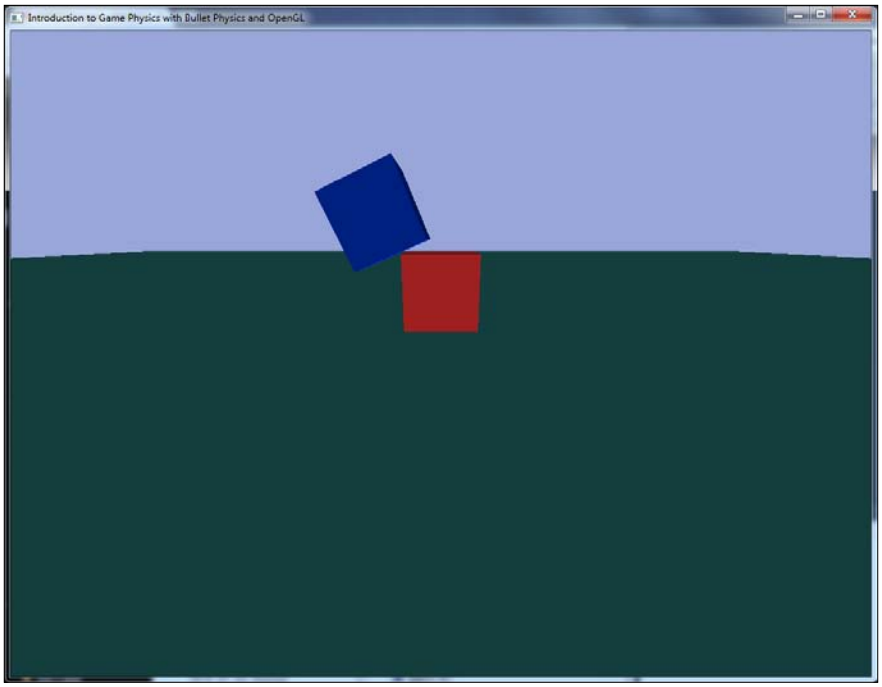
```
CreateGameObject(new btBoxShape(btVector3(1,1,1)), 1.0,
  btVector3(1.0f, 0.2f, 0.2f), btVector3(0.0f, 10.0f, 0.0f));
```

But, we can create two new objects with two more calls, and the majority of the work is taken care of by `GameObject`, `RenderScene()`, and `DrawShape()`.

```
// create a ground plane
CreateGameObject(new btBoxShape(btVector3(1,50,50)), 0,
  btVector3(0.2f, 0.6f, 0.6f), btVector3(0.0f, 0.0f, 0.0f));
// create a second box
CreateGameObject(new btBoxShape(btVector3(1,1,1)), 1.0,
  btVector3(0.0f, 0.2f, 0.8f), btVector3(1.25f, 20.0f, 0.0f));
```

Note that the increasing values of `x` in the initial position means the object starts further left. This is because of the position of our camera relative to the objects.

Now if we run our application, we should see a blue box falling down, hitting the red box, tilting, and falling over. It took us only two extra lines of code to create two new objects; that's a pretty good effort-to-reward ratio! The following screenshot shows our application with our new ground plane, and a new box colliding with our original box:



# Debug rendering

Observing a problem visually is usually the easiest first step in diagnosing it. So it is helpful for a physics engine to provide a means to draw debug information onto the screen whether it's the edges of objects, the points of collision, the depth of penetration, or more. Bullet provides a very simple interface for us to render this kind of debugging data onto our scene, which we will implement in this section.

Continue from here using the `Chapter4.2_DebugDrawer` project files.

# Building the debug drawer

To create our own debug drawer, we will inherit from `btIDebugDraw`, an interface class for debug rendering. This object must override essential functions such as `drawLine()` or `drawContactPoint()`. There are a few functions such as `reportErrorWarning()` and `draw3dText()` that we're not interested in, but are pure virtual, and requires us to at least define them. Since we won't need them, we will leave them empty.

Here is a snippet of one of the functions defined in `DebugDrawer`, which draws a colored line between the two given points:

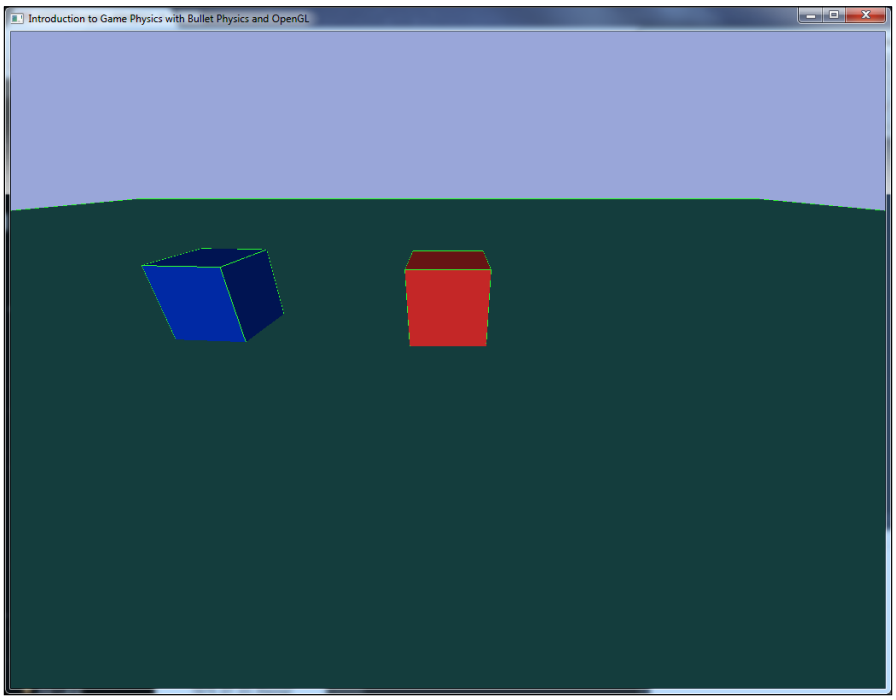
```
void DebugDrawer::drawLine(const btVector3 &from,const btVector3
  &to, const btVector3 &color)
{
  // draws a simple line of pixels between points.
  // use the GL_LINES primitive to draw lines
  glBegin(GL_LINES);
  glColor3f(color.getX(), color.getY(), color.getZ());
  glVertex3f(from.getX(), from.getY(), from.getZ());
  glVertex3f(to.getX(), to.getY(), to.getZ());
  glEnd();
}
```

The `DebugDrawer` class must be handed over to the world object through a call to `setDebugDrawer()`. In each render call, we will ask the world object to render the debug information through a call to `debugDrawWorld()`. This will cause the world to determine what needs to be rendered, based on which debug flags are set in `m_debugFlags`, and leave the actual rendering to our `DebugDrawer` class. Note that the only reason it can accept and use it is because it was derived from the `btIDebugDraw` interface class. If this seems confusing, it's worth brushing up on the concepts of **Polymorphism**.

The following is a snippet from the `Keyboard()` function, which allows us to change a single debug flag. These flags must come from Bullet's own internal enumerator of values, since it uses this to decide what to render for us.

```
case 'w':
  // toggle wireframe debug drawing
  m_pDebugDrawer->ToggleDebugFlag(btIDebugDraw::DBG_DrawWireframe);
  break;
```

Now our application can render debug lines and AABB volumes through a press of *W* or *B*, respectively. The following screenshot shows the application with debug rendering enabled:



# Introducing activation states

It may be apparent that the wireframe color of our boxes changes from white to green a few moments after an object comes to rest. This is yet another optimization that Bullet handles internally, which is only visualized through the debug mode, but has a profound effect on CPU usage. Objects whose velocity is below a given threshold for a given amount of time have their activation state set to deactivated. Meanwhile, there are actually two dynamic bounding volume trees created when you use a `btDbvtBroadphase` object (as we did). One stores the active objects (the active tree), and the other stores any static or deactivated objects (the deactive tree). So, when an object is deactivated, it pushes them into the other tree.

This causes Bullet to skip over them when its time for the world to move objects around, and since the broad phase object only compares the active tree against itself, and the active tree against the deactive tree (more importantly, it doesn't compare the deactive tree against the deactive tree) its impact on processing time is reduced even further. Later, when an active object collides with the deactivated one, it is activated once more, pushed back into the active tree, and Bullet performs the necessary calculations until it decides to deactivate it once more. These activation/deactivation states are typically referred to as putting the object to sleep, or waking it up.

Note that the ground plane is always drawn with a green wireframe (asleep) because Bullet knows that this object has an infinite mass, is static, is never a part of the active tree, and thus will never need to be moved.

This optimization has its drawbacks; sometimes an object may be moving slow intentionally, but if it is moving too slowly, Bullet deactivates it. For example, we might have an object that is very slowly teetering on an edge, which means it has a very low angular velocity for a long time, at which point Bullet may assume that it needs to put the object to sleep. This can cause some bizarre situations, where an object looks like it should be falling over, but is in fact it is frozen in place at an angle that would not be possible in the real world.

The typical workaround is to tweak the sleep threshold of the object, the minimum values of linear and angular velocity, which Bullet considers too low. This can be achieved by calling `setSleepingThresholds()` on any rigid body. As a last resort, we can force all the objects to remain activated, by calling the `setActivationState(DISABLE_DEACTIVATION)` function on every new object, but this will cost us some performance, since every object will now be a part of the active tree, and hence will be checked every iteration.

# The domino effect

While on the subject of activation states, it is worth mentioning the **domino effect**. It is possible for entire groups of touching objects (often called **islands**) to be put to sleep in the same fashion, greatly reducing their impact on the CPU. Once Bullet detects that none of the objects has moved in a while, they will all be put to sleep eventually.

This might seem convenient and allow us to throw more objects into the simulation, but be warned. When these islands get too large, consisting of too many objects, all it takes is for one of them to be nudged and the entire stack could become active simultaneously as they collide with one another, wake one another up, and the entire simulation slows to a crawl until Bullet puts some or all of them back to sleep again.

Too much activity such as this occurring in too short span of time can lead to adverse physics simulation behavior sometimes known as the **Spiral of Death**. This occurs when the simulation takes more time processing the current step, than the step simulates; for example, if we take 20 ms to simulate a 16 ms timestep we're behind by 4 ms, so the simulation needs to compensate in the next step, requiring even more time to process.

This can create a situation of progressively lower and lower frame rates until the simulation finds enough breathing room (simulation steps where very little is going on, such as when the awoken island has finally been dealt with) to catch up to the current moment in time. This would obviously create a very jarring and disappointing gameplay experience.

However, as mentioned towards the end of *Chapter 3*, *Physics Initialization*, Bullet lets you deal with this situation by tweaking the second and third parameters of `stepSimulation()`. But, this information is worth keeping in mind if you use an alternative physics engine, or find yourself building a custom solution in the future.

An old, but still relevant, blog post covers this concept in more detail and suggests good methods to avoid it:

`http://gafferongames.com/game-physics/fix-your-timestep/`

# Summary

We have introduced some generic object management code to better facilitate future projects and expansion. Our `GameObject` class contains the object data, while `BulletOpenGLApplication` ensures our objects are stored, and rendered regardless of how many we create and what their private properties might be.

We have also added debug rendering to our scene, so that we can visualize some of the information coming from our physics objects. This is a useful tool to keep in one's tool chest when working with a physics engine such as Bullet, since attempting to debug complex mathematics at a low level can be mind-bendingly difficult.

In the next chapter, we will implement some advanced mouse control through the power of raycasting and constraints.