8
Collision Filtering

Collision filtering is an act of informing the physics engine to ignore collisions between specific types of objects. This could either be an optimization to minimize physics processing activity, or an essential component of gameplay. Either way, the mechanism to implement this in Bullet is the same, and we will explore how to implement such a system in this chapter using groups and masks

Groups and masks

The code to implement collision filtering is absolutely trivial, but it requires a good amount of explanation before it can be fully appreciated and understood.

Continue from here using the `Chapter8_CollisionFiltering` project files.

When we call the `addRigidBody()` function on our world object, there is an overloaded version of the same function with two more parameters that we can input:

- A `short` representing the object's collision group
- A `short` representing the object's collision mask

Each object in our world can be a part of zero, one, or more collision groups. Groups could represent concepts such as players, power-ups, projectiles, enemies, and so on. Meanwhile, the collision mask indicates which groups this object should collide with. In this way, we can use collision filtering to generate an essential element of gameplay logic by preventing the player from being hit by their own weapon projectiles, or preventing enemies from being able to pick up power-ups.

Bullet treats the `short` values as a list of bit flags, and uses simple bitwise operations to perform tests on them. If bitwise operators and bitmasks sound scary or confusing, then this would be a good moment to break open your computer science manual of choice and do a little brush up on the subject. They are a commonly used tool in C++ programming. So common in fact, that we have already used them three times throughout this book in functions such as `glutInitDisplayMode()`, `glutClear()`, and our `DebugDrawer` class.

Bitmasks are used by Bullet for these operations because performing comparisons on them is absurdly fast and they use a minimal amount of data. With potentially thousands of comparisons to perform per simulation step, this is a very worthwhile optimization that has been built into Bullet.

Because the two values for group and mask are `shorts`, we have 2 bytes, or 16 bits, to work with. But, Bullet reserves one of these bits for internal usage, which gives us access to the remaining 15, however this should be more than enough groups for most situations.

The last thing to consider is that if we want two objects to collide with one another, then both of their masks must include the group of the opposing object. For example, if we want the players to collide with power-ups, we can set the player's mask to do so; but power-ups must also be flagged to collide with player's, or else the collisions will not occur. Remember this when attempting to use this feature in the future, because it is an easy thing to forget.

The `BasicDemo` application is becoming cluttered, so we have created a specific application to test collision filtering named `CollisionFilteringDemo`. Using this object, instead of `BasicDemo`, required a handful of changes to our `main()` function.

To implement collision filtering, we simply pass the two aforementioned `shorts` into our call to the `addRigidBody()` function. This merely requires a change in the parameters and the function calls of `CreateGameObject()`. Because it is so trivial, we won't show the whole source code here, but we will make a couple of relevant points:

```
enum CollisionGroups {
  COLGROUP_NONE = 0,
  COLGROUP_STATIC = 1 << 0,
  COLGROUP_BOX = 1 << 1,
  COLGROUP_SPHERE = 1 << 2
};
```

This `enum`, found in `BulletOpenGLApplication.h`, defines the possible collision groups for our object. Each represents a different group and is represented by a value of 1, but bit shifted left by gradually increasing values of 2, 4, 8, 16, and so on. This is a simple pattern to ensure that each value consumes a unique bit. The same `enum` that defines the values for the groups is also used to determine each new object's collision mask. To set more than one group for an object's collision mask, we use the bitwise-or operator, as follows:

```
COLGROUP_BOX | COLGROUP_STATIC
```

Passing this value as the object's mask makes it collide with any object flagged for either of these groups.

Defining linear and angular freedom

It's becoming increasingly common these days to see the games that are visually 3D, but all gameplay and physics occurs in only two dimensions. These types of games are typically referred to as **2.5D games**. These are either attempts to bring a classic 2D game back to life with modern 3D graphics, or a way to keep the simplicity of 2D gameplay, but give them more life and believability through advanced graphics. To achieve this, physics objects must only be able to move in the X and Y axes, and only able to rotate around the Z axis.

Restrictions of this kind can be applied to any rigid body object in Bullet by setting the linear or angular factor of a rigid body. Simply call the `setLinearFactor()` or `setAngularFactor()` functions on any rigid body, passing in a `btVector3` that specifies which axes are allowed, and which are not. For instance, to restrict the movement of an object to behave as if it was a 2.5D game, we would call:

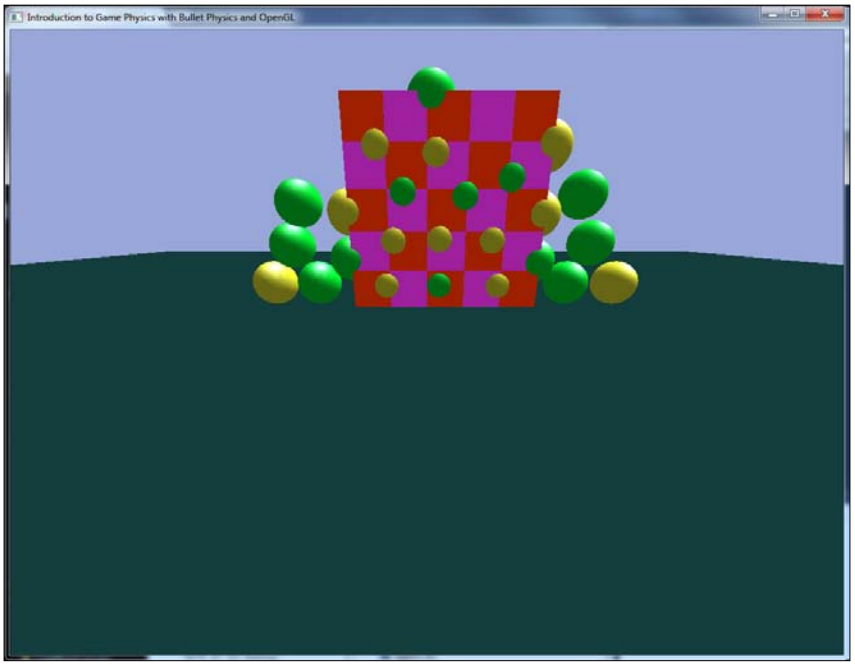
```
pBody->setLinearFactor(btVector3(1,1,0));
pBody->setAngularFactor(btVector3(0,0,1));
```

To demonstrate this feature, this chapter's source code sets the spheres to only move in the X - Y plane (up/down/left/right relative to our camera's starting position), while being constricted along the Z plane (they cannot move towards/away from the camera's starting position). Even if another object (such as a box we shoot with a right-click) collides with one of the spheres along the Z axis, it still cannot move in that direction. The following call restricts the linear motion of the sphere in such a fashion:

```
pSphere->GetRigidBody()->setLinearFactor(btVector3(1, 1, 0));
```

The `CollisionFilteringDemo` application creates 25 boxes and 25 spheres in a stacked 5 x 5 grid formation. It then configures both types of objects to collide with the ground plane, but also configures such that the boxes cannot collide with the spheres, and vice versa.

When we launch this application, we should observe two stacks of boxes and spheres, each occupying the same space without any collisions between them. There can be collisions only with the objects of the same shape. The following screenshot shows the collision filtering in effect:



Note that the default values for group and mask are set to `-1` in the declaration of the `CreateGameObject()` function. If we remember our signed-integer representations, a value of `-1` means every bit is set to 1. Thus, the default values for group and mask make the object a member of every group, and has a mask enabled for every group. This is the reason why the ground plane and our shootable boxes are able to collide with both the boxes and spheres.

Summary

We've explored the power and simplicity of Bullet's collision filtering system, and implemented it into our scene to avoid generating collisions between objects of different groups. This feature can be extended further to all kinds of useful situations, both for the sake of gameplay and for simulation optimization.

In the next chapter, we will explore one final and powerful feature of the Bullet library: Soft bodies!