5
Raycasting and Constraints

Many games use a mouse or touch screen as the primary means of controlling objects; whether it's by selecting, moving, creating, or destroying them.

If we want to implement such a system into our application, then we must find a way to translate a mouse click (x-y coordinates on the screen) into a method of detecting the first object underneath the pointer. The answer to this conundrum is **raycasting**, which we will be exploring in this chapter.

Then, once we have a raycasting system in place, we will explore how we can use it in combination with Bullet's constraint system to move objects with the mouse cursor.

The power of raycasting

Raycasting is a technique that can be used for a variety of different tasks. A common use is to find objects underneath the cursor from the camera's perspective. This is typically referred to as **picking**. However, rays are also used in other tasks, such as in shooter games to cast a line from the barrel of a weapon to where a bullet might strike, which could be a wall or another player.

Another common usage of rays is to surround a player character with many rays that point outwards from the object, that are used as feelers to detect if the player is near other objects. For example, there could be a ray that points downwards from the player's feet a short distance. If the ray collides with a physical object, then we know that the player is touching the ground and telling us to play the appropriate animation, or reset a flag that allows them to jump.

Regardless, all of these concepts are built from the same basic idea; choose a starting point, pick a direction to travel in (a ray), and move along that direction (cast) until it collides with something. Let's create a basic picking ray function which exactly does that.

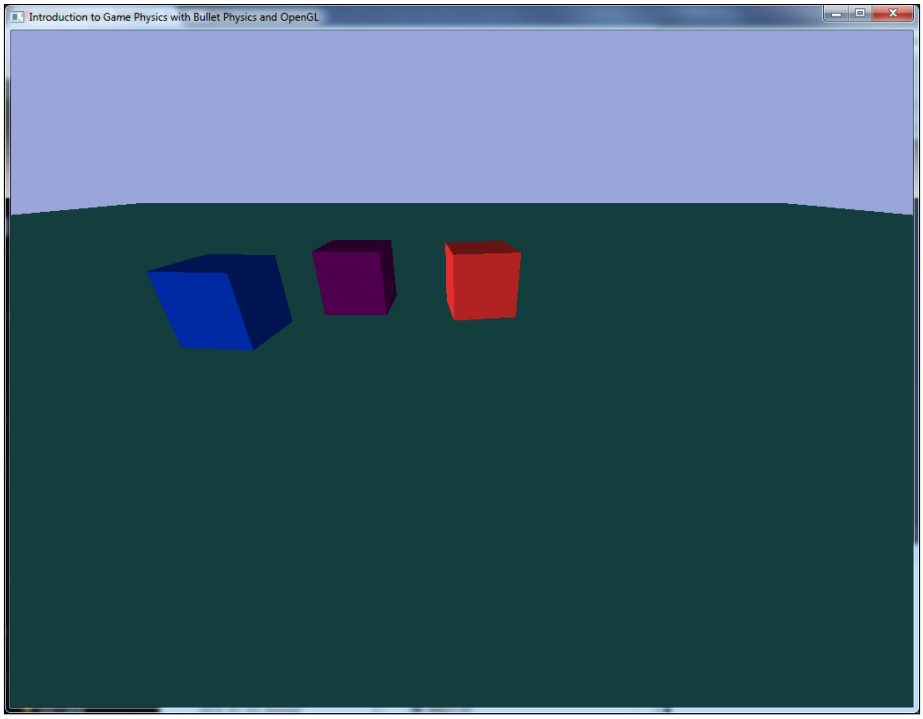
Continue from here using the `Chapter5.1_Raycasting` project files.

Picking rays

The `GetPickingRay()` function in the book's source code involves a large smattering of 3D mathematics that are beyond the scope of this book. It should be enough to know that it takes the x-y coordinates of a mouse click, and uses the camera's data (its position, near plane, far plane, field of view, and aspect ratio) to calculate and return a `btVector3` in world coordinates that points forward from the camera in the corresponding direction. If the camera moves, or we click somewhere else on the screen, then we get a new `btVector3` pointing forward from that position instead. Armed with this function, we can add some simple code to create a new object whenever we click on the right mouse button. This code can be found in the chapter's source code, in the `ShootBox()` function. This function is called by the `Mouse()` function anytime when the application detects that the right mouse button was clicked.

Recall that the `Mouse()` function was called by FreeGLUT anytime a mouse button is clicked. It gives us the button, the state (pressed or released), and the x-y coordinates of the click.

Launch our application and try right-clicking on the mouse. It should create a purple box and launch it towards the mouse cursor. The following screenshot shows this in action:



We've jumped ahead a little with the `setLinearVelocity()` function. This instruction is used to set the collision object's linear velocity. We'll learn more about manipulating the rigid bodies through functions such as this in *Chapter 6, Events, Triggers, and Explosions*.

Destroying objects

So far, we've essentially created a starting point for a picking ray. It is not a true raycast or picking ray until the ray travels forward in space and performs some type of collision detection. To destroy an object in our scene, we'll need to use our picking ray to perform a raycast and tell us the first rigid body with which it collides.

Raycasting in Bullet is handled through the `btDynamicsWorld` object's `rayTest()` function. We provide the starting point (as a `btVector3`), the direction (`btVector3`), and an object to store the raycast data inside, which should be one of two different classes that inherit from `RayResultCallback`. The object could either be:

- `ClosestRayResultCallback`, which gives the closest collision that the ray detected from the start location
- `AllHitsRayResultCallback`, which gives an array filled with all of the collisions the ray detected

Which object we want to use will depend on whether we want only the closest hit, or all of them. We will be using `ClosestRayResultCallback`, which contains useful data and member functions for the collision point, such as:

- `hasHit()`, which returns a boolean value and tells us if there was a collision between the ray and any physics object
- `m_collisionObject`, which is the `btCollisionObject` our ray hit
- `m_hitPointWorld`, which is the coordinate in world space where the ray detected a collision

The `Raycast()` function in the book's source code takes a picking ray and an empty output `RayResult` structure, uses it to create a `ClosestRayResultCallback`, and then performs a raycast test. If the raycast was successful, the function fills out the structure and returns true, allowing us to check the success or failure of the raycast outside of this function.

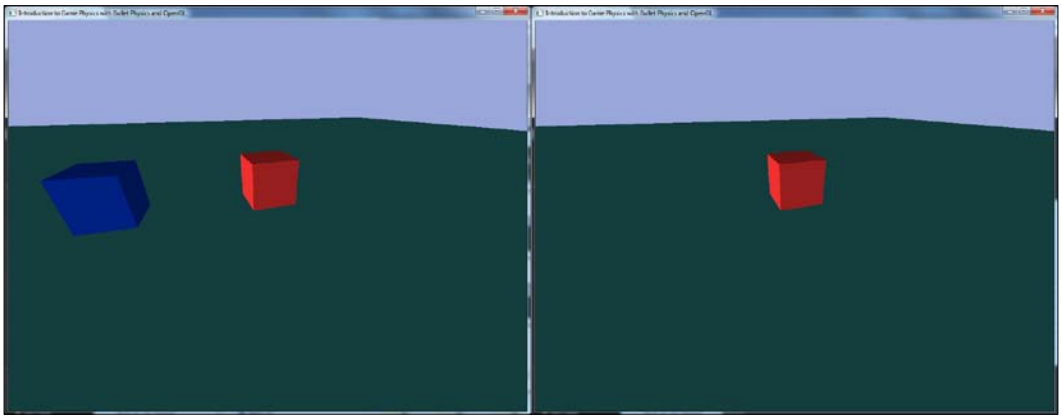
Notice the special case to avoid picking static objects, such as our ground plane. When we gave our ground plane a mass of zero, Bullet automatically set the static flag for us, allowing us to check for it at a later date.

Before we can destroy the picked rigid body we need to know what `GameObject` that corresponds to. We will have to search through our list of game objects, comparing their rigid bodies with the picked one, until we find it. Then, and only then, is it safe to destroy it.

Check the `DestroyGameObject()` function in the chapter's source code for details of this process. This function searches through our list of objects hunting down `GameObject` that corresponds to the given `btRigidBody`. It is then called during the `Keyboard()` function, whenever we detect that the user pressed the *D* key.

Note that the mouse coordinates, `x` and `y`, are also passed into functions such as `Keyboard()`. This greatly simplifies our input handling, preventing us from having to store the current mouse data locally.

Launch the application, hover the mouse cursor over an object, and press *D* on the keyboard. Any objects beneath the cursor should now be instantly destroyed (with the exception of the ground plane). The following are the screenshots before and after destruction of the box on the left:



Constraints

We'll now explore Bullet's constraint feature. These objects limit the range of motion of one object relative to another, giving us the power to create some very interesting and unique gameplay situations.

Continue from here using the `Chapter5.2_Constraints` project files.

Understanding constraints

Constraints, in their most basic form, are the rules which limit the range of motion of an object relative to some specific object or point in space. For example, think of a desk chair. It is made up of multiple parts, but if we push the base of the chair, the rest must move with it. The same happens if we push the top section; so even though the chair is made of multiple pieces, they are constrained to one another by a handful of rules.

Constraints can be used to also simulate the independent rotation of the desk chair's top section relative to the base. The top section is able to rotate around an axis without any dependence on what the bottom section is doing. This constraint is simulated by hooking the top section to an invisible point, and only allowing rotation around a single axis about that point.

Constraints can vary in how strongly they influence their target objects. A strong constraint enforces its limitation on movement at all times as strongly as it can. So, if two objects are connected by a very strong, rigid constraint, it is the equivalent of being attached together by invisible and unbreakable glue. In other words, if one object is moved one unit in space, then the other must move one unit in space to follow it.

Weaker constraints are more like springs. Under the same scenario, the first object might move one unit in space, but the second moves somewhat less, causing the two objects to come closer together, or pushed further apart. In addition, the more they are pushed away from their resting position, the harder the constraint pulls them back; if we recall our Newtonian physics, this is much like how a simple spring functions.

Picking up objects

A feature of most games is to allow the player to pick up and move the objects around with the mouse cursor or touch screen (also useful for debugging and testing!). There are several ways to achieve this, such as with forces, or updating the rigid body's transform each iteration, but we would like to use a constraint to achieve this effect.

The idea is to use our existing raycasting functionality to detect which object was selected and the exact point of a mouse click. We then create a new constraint at that point and attach it to the selected object. Then, every time we move the mouse (while the mouse button is still held down), we update the position of the constraint. The expectation being that our selected object would move with the constraint, and keep the same relative position until it is freed from its influence.

There are a handful of different objects which Bullet provides in order to implement the constraint system. We'll cover the `btGenericDof6Constraint` object, the most generic of the available options (hence the name). Its purpose is to give us an interface to limit the six degrees of freedom (*Dof6* for short) of an object; these refer to the three axes of both linear and angular motion. This constraint can either be used to hook two rigid bodies together, or hook a single object to a single point in space.

Building a constraint

We've seen raycasting in action earlier in this chapter, so all we need to cover is the creation, update, and destruction of the constraint itself. `CreatePickingConstraint()` is a very large function, so we'll explore some code snippets one step at a time:

```
if (!Raycast(m_cameraPosition, GetPickingRay(x, y), output))
  return;
```

This instruction should look familiar, since we used it earlier in this chapter. It performs a raycast and returns true if it finds anything, pushing the relevant data into the output variable.

```
m_pPickedBody->setActivationState(DISABLE_DEACTIVATION);
```

Here we're ensuring the picked object doesn't fall asleep while attached to our constraint. We covered activation states back in *Chapter 4*, *Object Management and Debug Rendering* and the last thing we want is our picked object to freeze in place while we still have it selected!

```
// get the hit position relative to the body we hit
btVector3 localPivot = m_pPickedBody-
  >getCenterOfMassTransform().inverse() * output.hitPoint;
```

We mentioned earlier how we would create the constraint at the exact point of the click, which is exactly what the previous calls do, except it does so in a rather convoluted way.

Constraints must be defined in local space coordinates, for example, let's say we have two objects positioned at `(0,3,0)` and `(0,10,0)` in world space coordinates. But, from the first object's perspective, it is always positioned at `(0,0,0)` in its own local space, regardless of where it is in world space. Also, as far as the first box is concerned, the other box is positioned at `(0,7,0)` in its local space. Meanwhile, from the second object's perspective, it is also positioned at `(0,0,0)` in its local space, and the other box is located at `(0,-7,0)` in its local space.

It's possible to obtain these values mathematically by multiplying the vector representing a point in world space by the inverse of an object's transformation matrix. Therefore in the preceding code, we multiply the hit point by the inverse transform of the box's center of mass, giving us the hit point coordinates from the box's local space perspective.

The previous mathematical calculation is a very important and useful feature of matrices that is worth remembering for the future.

Next we create our constraint object:

```
btGeneric6DofConstraint* dof6 = new
  btGeneric6DofConstraint(*m_pPickedBody, pivot, true);
```

The constraint requires us to provide the body in question, the pivot point (again, in local space coordinates), and a `bool` value. This boolean tells the constraint whether to store various pieces of data relative to object A (the rigid body) or object B (the constraint's pivot point in this case, but could also be a second rigid body). This becomes important when using the constraint later.

```
dof6->setAngularLowerLimit(btVector3(0,0,0));
dof6->setAngularUpperLimit(btVector3(0,0,0));
```

Also, calling the `setAngularUpperLimit()` and `setAngularLowerLimit()` functions with zero's `btVector3`s add a rotational limitation to the box while it is attached to this constraint, preventing it from rotating.

```
m_pWorld->addConstraint(dof6,true);
```

Much like rigid bodies, it's not enough to create the object; we must also inform the world of its existence, hence we call the `addConstraint()` function. The second parameter disables the collisions between the two linked bodies. Since we don't have two bodies in this constraint (we have a body and a pivot point), it would be wise to tell Bullet to save itself some effort by setting the value to `true`. If we had two rigid bodies connected via a weak constraint and were interested in having them collide, we would want to set this value to `false`.

```
// define the 'strength' of our constraint (each axis)
float cfm = 0.5f;
dof6->setParam(BT_CONSTRAINT_STOP_CFM,cfm,0);
dof6->setParam(BT_CONSTRAINT_STOP_CFM,cfm,1);
dof6->setParam(BT_CONSTRAINT_STOP_CFM,cfm,2);
dof6->setParam(BT_CONSTRAINT_STOP_CFM,cfm,3);
dof6->setParam(BT_CONSTRAINT_STOP_CFM,cfm,4);
```

```
dof6->setParam(BT_CONSTRAINT_STOP_CFM,cfm,5);
// define the 'error reduction' of our constraint (each axis)
float erp = 0.5f;
dof6->setParam(BT_CONSTRAINT_STOP_ERP,erp,0);
dof6->setParam(BT_CONSTRAINT_STOP_ERP,erp,1);
dof6->setParam(BT_CONSTRAINT_STOP_ERP,erp,2);
dof6->setParam(BT_CONSTRAINT_STOP_ERP,erp,3);
dof6->setParam(BT_CONSTRAINT_STOP_ERP,erp,4);
dof6->setParam(BT_CONSTRAINT_STOP_ERP,erp,5);
```

This is where things get a little weird. The `setParam()` function sets the value of a number of different constraint variables, two of which are used in the preceding code. It is called a total of twelve times, since there are three axes, two directions for each axis (positive and negative), and two different types of variable to edit (`3x2x2 = 12`). The two aforementioned variables are **CFM** (**Constraint Force Mixing**) and **ERP** (**Error Reduction Parameter**).

CFM is essentially a measure of the strength of the constraint. A value of `0` means a perfectly rigid constraint, while increasing values make the constraint more spring like, up to a value of `1` where it has no effect at all.

ERP represents the fraction of how much joint error will be used in the next simulation step. Many constraints could be working in unison to create a complex interaction (imagine a rope bridge, which can be simulated by a attaching a bunch of springs connected together) and ERP is used to determine how much of the previous data will affect the calculation of future data. This is a difficult concept to explain in such a short space, but imagine that we have multiple constraints acting on the same object, each forcing the others into breaking their own rules. ERP is then the priority of this constraint relative to the others, and helps determine who has higher importance during these types of complex constraint scenarios.

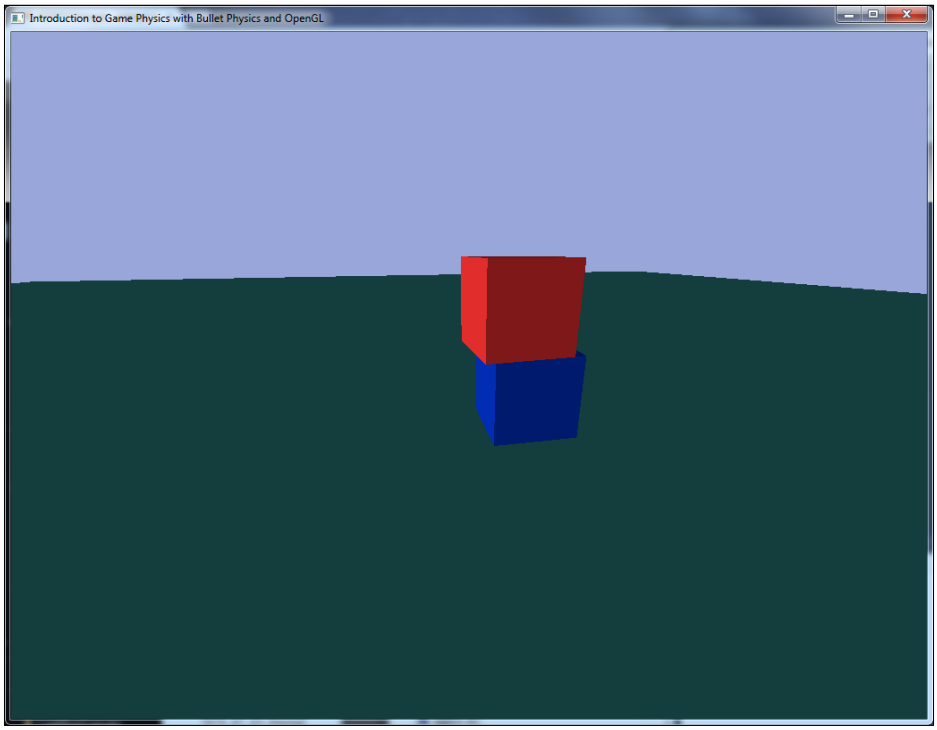
And there we have it. We detected the collision point, and then built our constraint. That wasn't so bad, was it? The last snippet of code to look at is in the `Motion()` function, the code which updates the position of the constraint while we're still holding down the left mouse button.

```
// use another picking ray to get the target direction
btVector3 dir = GetPickingRay(x,y) - m_cameraPosition;
dir.normalize();
// use the same distance as when we originally picked the object
dir *= m_oldPickingDist;
btVector3 newPivot = m_cameraPosition + dir;
// set the position of the constraint
pickCon->getFrameOffsetA().setOrigin(newPivot);
```

It was mentioned earlier that it was possible to get data from the constraint in a form which is relative to one of the two objects involved in the constraint (called A and B). We use the `getFrameOffsetA()` function to get the transform position of the constraint relative to the first object, and then update it with the new value. This is the equivalent to updating the position of the constraint's pivot point. Thus in the next simulation step, the constraint will attempt to move the box to the new position of the mouse, keeping the same distance from the camera as when it was first picked.

The last thing to mention is the `RemovePickingConstraint()` function, which makes sure that we have an existing constraint before attempting to destroy it. If so, we must remove it from the world, destroy the object in memory, nullify the pointers. The re-enable the ability of the picked up object to go back to sleep.

In this section's application we can pick up one of our objects with the left mouse button and move it around. The following screenshot shows that the first box has been moved on top of the second box of to our mouse clicking constraint:



Try tweaking the `bLimitAngularMotion`, `cfm`, and `erp` variables in `CreatePickingConstraint()` and observe the effects they have on the picked object.

Summary

We've witnessed the power of picking, raycasting, and constraints by adding some mouse control to our application. This flexible system is used to create, move, and destroy objects in the scene. This allows for some very creative gameplay mechanics, animations, and effects, since many games rely on these mechanisms as an essential component of gameplay, so these are all lessons to take forward when implementing similar systems in your own projects.

In the next chapter, we'll add more game logic control to our application by adding a collision event system, complete with volumes of space which act as triggers, and manipulating our objects through various types of force.