7 Collision Shapes

So far we've only built box collision shapes, but Bullet offers much more variety than this. In this chapter, we'll explore some more built-in collision shapes to bring more variety into our scene. We will initially cover some more built-in shapes such as spheres and cylinders, and then explore how to build unique, customized shapes through convex hulls and compound shapes.

Spheres and cylinders

The `btSphereShape` and `btCylinderShape` are the two collision shapes that could not be more simple; they define a sphere and cylinder, respectively.

[Continue from here using the `Chapter7.1_SpheresAndCylinders` project files.]

We already know how to create `btBoxShape`, and building these new built-in shapes is not much different. Note that nothing about the way we handle rigid bodies or motion states changes when we start working with different collision shapes. This is one of the big advantages of the Bullet's modular object system. These new shapes can be instantiated in the creation of a new `GameObject` as follows:

```
CreateGameObject(new btSphereShape(1.0f), 1.0, btVector3(0.7f,
  0.7f, 0.0f), btVector3(-5.0, 10.0f, 0.0f));
CreateGameObject(new btCylinderShape(btVector3(1,1,1)), 1.0,
  btVector3(0.0f, 0.7f, 0.0f), btVector3(-2, 10.0f, 0.0f));
```

Simple, right? Unfortunately, we now face a more significant challenge: how to render these new objects? We could just render them like boxes, but this won't be ideal. We'll need to introduce some new drawing functions, akin to `DrawBox()`, which render objects of differing shapes and sizes. Thanks to the rigorous refactoring we performed on our rendering code back in *Chapter 4, Object Management and Debug Rendering*, we have made this whole process fairly trivial on ourselves.

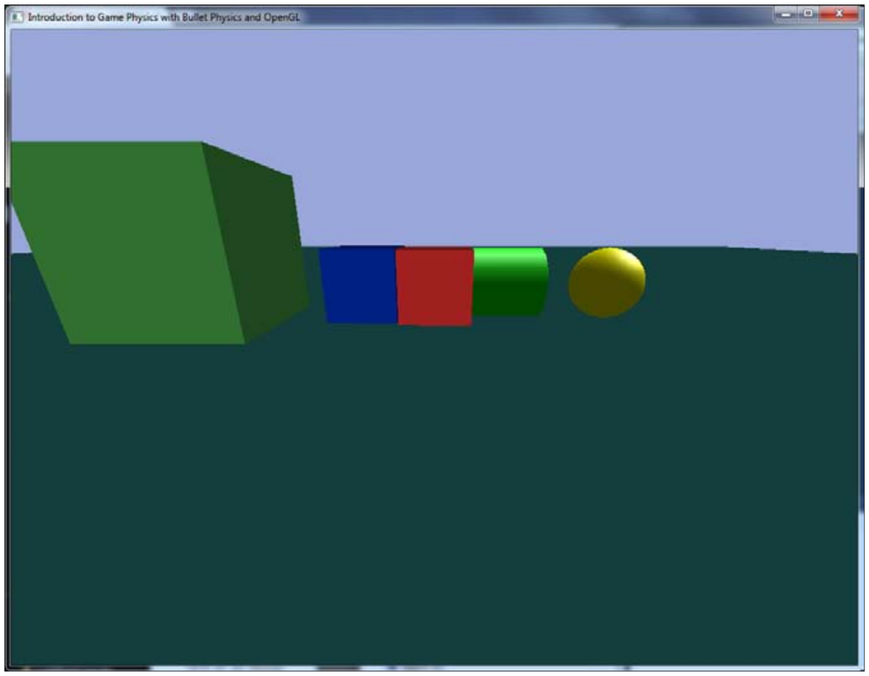
`DrawSphere()` uses a new OpenGL primitive type, `GL_QUAD_STRIP`, to create strips of quads. A quad is made of four points, rather than three points for a triangle. A strip of quads is built two vertices at a time, since they are connected together in strips end-to-end. This is a much more efficient way of rendering many primitives in one step.

In order to generate a spherical object of a given radius with quads, we have to iterate laterally, find the angles of this segment, then iterate longitudinally, and draw them.

Meanwhile, to draw a cylinder we can make use of a new helper function in OpenGL to build the cylinder piece-by-piece using quadrics to build two disks via `gluDisk()`, and a cylindrical hull via `gluCylinder()`. These quadric functions are built into FreeGLUT, providing an interface with which we can build meshes using simple mathematical equations. There are various types of quadrics that are available in the FreeGLUT library, which you can find in the documentation and/or source code.

To save space we won't cover any code snippets here, since there are far too many bite-size simple commands being introduced. But, take the time to look at the new functions `DrawSphere()`, `DrawCylinder()`, and the changes to `DrawShape()`.

Our application now renders a yellow sphere and green cylinder to accompany our two original boxes. Try shooting boxes at them and observe their motion. They behave exactly as we would expect a physics object of that shape to behave! The following screenshot shows our new cylindrical and spherical objects added to our scene:



Convex hulls

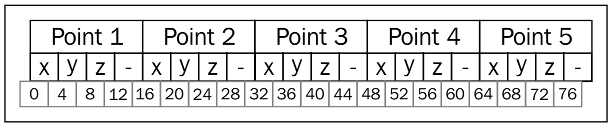
Next, we'll explore how Bullet lets us build custom collision shapes through the use of convex hulls.

Continue from here using the `Chapter7.2_ConvexHulls` project files.

Much like our OpenGL code, we provide the vertex points from which to build the object and Bullet takes care of the hard work for us; in fact Bullet makes it even easier than that, because we don't need to define indices or provide them in a specific order of rotation. Bullet will always try to create the simplest convex shape it can from the given set of points (also known as a **point cloud** or **vertex cloud** in this context). It is important for the objects to be convex because it is orders of magnitude is easier to calculate collisions with convex shapes (those without any internal dips or caves in the shape) than with concave shapes (those with caves in its surface).

A convex hull is defined by the `btConvexHullShape` class. We must perform a little programming gymnastics to create our convex hull, by generating an array of five `btVector3s`, and passing the memory address of the first point's x coordinate into our convex hull. This may seem confusing at first, but it's straightforward once we appreciate the importance of contiguous memory.

A `btVector3` consists of four floats: x, y, z, and an unused buffer float. Why is there an unused variable in this object? Because CPUs are very efficient while working in powers of 2, and since a float is 4 bytes large, that makes an entire `btVector3` object 16 bytes large. Throwing in an unused float like this is a good way to force the compiled code to make these objects 16 bytes large. This is yet another one of those low-level optimizations to be found in Bullet. In addition, an array of `btVector3s` are contiguous in memory (by definition) such that they follow one another sequentially by address.



Five btVector3 objects contiguous in memory.

The `btConvexHullShape` constructor expects us to provide three things: the memory address of start of a list of vertex points, the number of vertices, and the stride, or how many bytes in memory it should jump to reach the next vertex.

Since the memory address must be provided in the form of `btScalar`, we will call `getX()` on the first point to get the memory address that we need. The number of vertices is required so that it knows when to stop counting (computers are stupid like that), and the stride is necessary to determine how to count. The default value for the stride is 16 bytes, which is (not by coincidence) the size of a `btVector3` object; so there's actually no need to provide this argument, but it is worth mentioning because this concept appears all the time when working with vertex and index buffers.

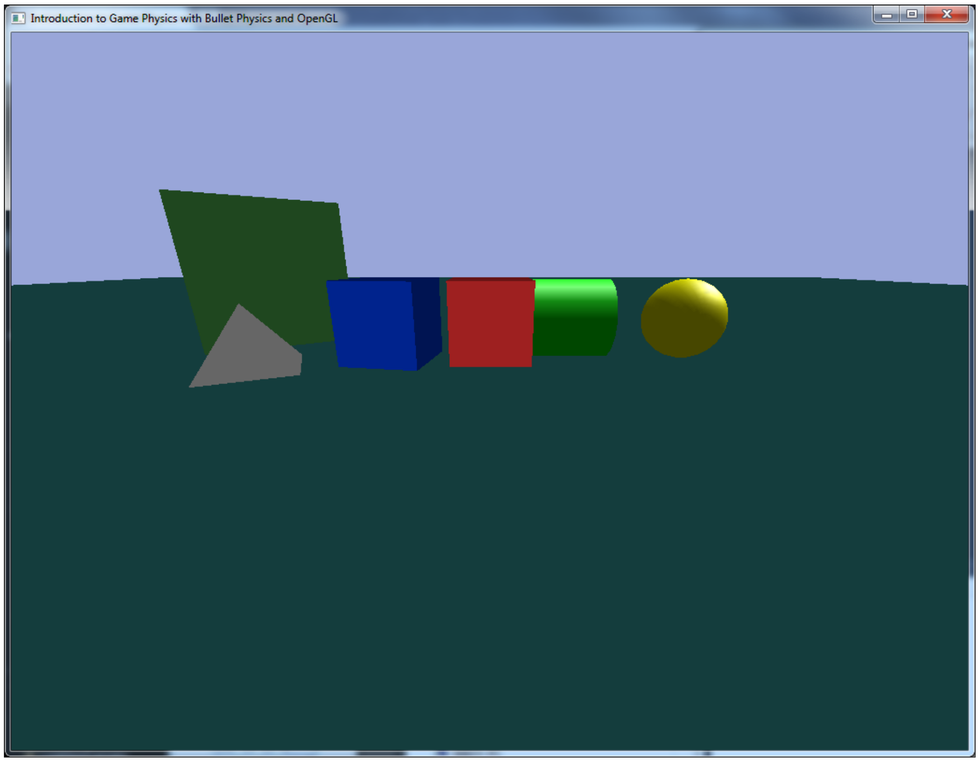
Hopefully things become clear once we explore the code for this procedure:

```
// create a vertex cloud defining a square-based pyramid
  btVector3 points[5] = {btVector3(-0.5,1,1),
  btVector3(-0.5,1,-1),
  btVector3(-0.5,-1,1),
  btVector3(-0.5,-1,-1),
  btVector3(1,0,0)};
// create our convex hull
  btConvexHullShape* pShape = new
    btConvexHullShape(&points[0].getX(),5);
// initialize the object as a polyhedron
pShape->initializePolyhedralFeatures();
// create the game object using our convex hull shape
CreateGameObject(pShape, 1.0, btVector3(1,1,1), btVector3(5, 15,
  0));
```

There's one erroneous function call above, `initializePolyhedralFeatures()`. This function can be called on a convex hull shape to convert the data into a special format that gives us access to some convenient functions that we'll need to render the object later. It essentially builds the indices and vertices for us, so we don't have to.

Once again, we can throw this shape into our `GameObject` constructor and it is none the wiser. The only part of our code that cares is our rendering code. Once again we will skip providing the actual code here, but check out the new function `DrawConvexHull()` and changes to `DrawShape()` to observe the process of rendering these shapes. It is doing little more than grabbing the polyhedral vertex/index data and rendering the relevant triangles.

The following screenshot shows our application, which now includes a white pyramid shape falling from the sky along with our original shapes:



An important point to note before we move on is that Bullet assumes the center of mass of the object is at (`0,0,0`) relative to the given points, ergo the points must be defined around that location. If we wish to set the center of mass to a different location, then we must call the `setCenterOfMassTransform()` function on the object's rigid body.

Creating convex hulls from mesh data

Building a convex hull by manually typing in the vertex data can be incredibly tedious. So Bullet provides methods for loading customized mesh file data into the desired vertex format that we used earlier, provided the data has been stored in the `.obj` format (a common format that every 3D modeling tool supports these days). To see this process in action, check out the `App_ConvexDecompositionDemo` application in the Bullet source code.

However, be warned that creating convex hulls from a complex shape (such as a table or a four-legged chair) requires a lot of CPU cycles to generate accurate collision responses for them. It is wise to stick with simple collision shapes that estimate the physical object, such as boxes and spheres, unless absolutely necessary.

Compound shapes

Bullet also allows us to build another type of customized physics object by combining multiple child shapes together into a parent compound shape.

[Continue from here using the `Chapter7.3_CompoundShapes` project files.]

Compound shapes are treated much the same way as any other shape, except its constituent pieces are stuck together by a set of very rigid constraints (much like the constraints we explored in *Chapter 5*, *Raycasting and Constraints*). We'll use compound shapes to create a dumbbell, a pair of spheres connected via a connecting rod.

[Note that the child shapes do not need to touch one another for the compound shape feature to work. The child shapes could be separated by great distances and still behave as if they were tightly coupled.]

The class in question is the `btCompoundShape` class. The member function `addChildShape()`, attaches the given child shape into the compound shape at the given transform. Therefore a simple compound shape can be built as follows:

```
// create two shapes for the rod and the load
btCollisionShape* pRod = new btBoxShape(btVector3(1.5f, 0.2f,
  0.2f));
btCollisionShape* pLoad = new btSphereShape(0.5f);
// create a transform we'll use to set each object's position
  btTransform trans;
```

```
trans.setIdentity();
// create our compound shape
btCompoundShape* pCompound = new btCompoundShape();
// add the rod
pCompound->addChildShape(trans, pRod);
trans.setOrigin(btVector3(-1.75f, 0.0f, 0.0f));
// add the top load
pCompound->addChildShape(trans, pLoad);
trans.setIdentity();
// add the bottom load
trans.setOrigin(btVector3(1.75f, 0.0f, 0.0f));
pCompound->addChildShape(trans, pLoad);
// create a game object using the compound shape
CreateGameObject(pCompound, 2.0f, btVector3(0.8,0.4,0.1),
  btVector3(-4, 10.0f, 0.0f));
```

Bullet lets us create yet another complex physics object with only a handful of instructions. But, yet again, we have the problem of rendering this shape. We can use the compound shape's member functions `getNumChildShapes()`, `getChildTransform()`, and `getChildShape()` to iterate through the child shapes, but remember that our `DrawShape()` command only accepts a single shape to draw at a time, and if we push our compound shape into a game object and render it, it would not draw anything, because the parent itself is not one of the supported types.

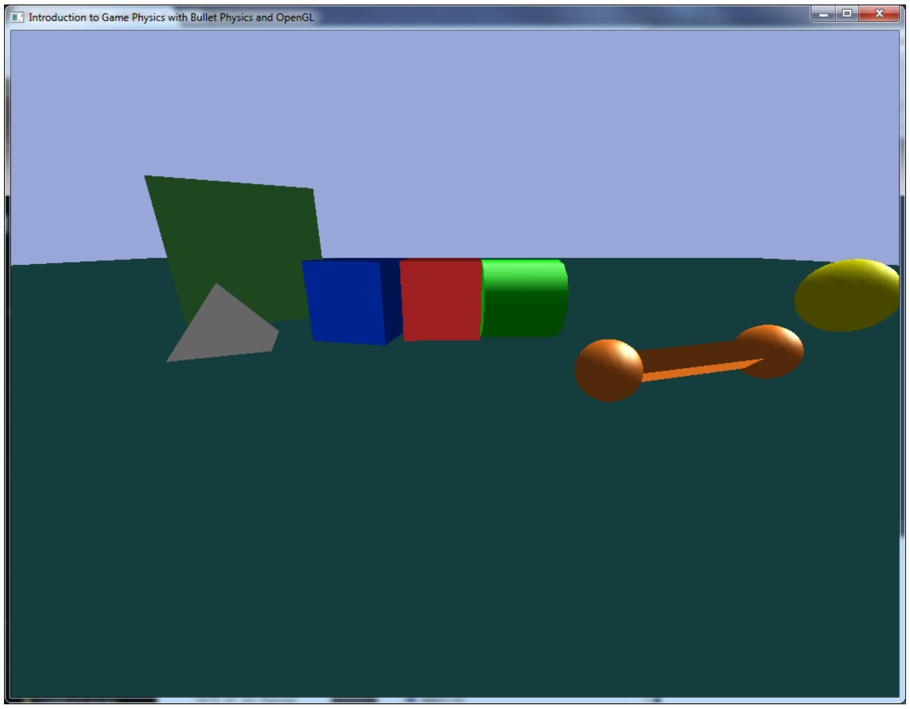
What we must do is to call the `DrawShape()` function recursively for each child as follows:

```
case COMPOUND_SHAPE_PROXYTYPE:
{
  // get the shape
  const btCompoundShape* pCompound = static_cast<const
    btCompoundShape*>(pShape);
  // iterate through the children
  for (int i = 0; i < pCompound->getNumChildShapes(); ++i) {
    // get the transform of the sub-shape
    btTransform thisTransform = pCompound->getChildTransform(i);
    btScalar thisMatrix[16];
    thisTransform.getOpenGLMatrix(thisMatrix);
    // call drawshape recursively for each child. The matrix
    // stack takes care of positioning/orienting the object for us
    DrawShape(thisMatrix, pCompound->getChildShape(i), color);
  }
  break;
}
```

If the purpose of the matrix stack wasn't clear earlier, then the preceding exercise might help. When the two weights of the dumbbell are drawn, it might appear that the only transform information given are the positions at (`-1.75,0,0`) or (`1.75,0,0`), and yet it doesn't always render at those exact world space coordinates.

In reality, it renders at the above location relative to the parent object. But why? This powerful mechanism is achieved by adding the child's transformation matrix to the stack (with another call to `glPushMatrix()`) rendering the child, removing its transform from the stack (with another call to `glPopMatrix()`), and repeating for the next child. Thus, wherever the parent's transform begins, the matrix stack ensures that the children are always drawn relative to that starting location.

Our application now features our new dumbbell object:



Our new dumbbell object

It's worth mentioning that we only created two unique shapes in memory for our dumbbell: one for the rod and one for the load. Yet, our dumbbell is built from three unique shapes. This is an essential memory saving feature of collision shapes. Their data can be shared by more than one collision object, and still be treated as two unique instances.

Summary

We have created four new types of Bullet collision shapes in our application by introducing more case statements to our `DrawShape()` function. Any object can be built from primitive shapes such as triangles, quads, and so on (which is why they're called primitives), but we have also discovered that there are helper functions inside FreeGLUT called quadrics which make this process easier for us.

In the next chapter, we will explore how collision filtering can be used to develop interesting physics behavior and game logic.