Physics - Broad phase and Narrow phase

Introduction

The physics engine that has been built up throughout the tutorial series is getting more powerful - it can detect collisions between simple collision volumes, resolve them, and apply forces to points on the surface of a shape, producing both linear and angular velocity. As we determined in the last tutorial, testing every object against every other object in our scene will quickly lead to a lot of wasted computation. To alleviate this, physics engines will split the process of determining and resolving collisions up into two separate phases, the *broad phase*, and the *narrow phase*. The broad phase roughly determines which physics bodies *might* intersect, and then the narrow phase performs the actual detection, and resolves any collisions that have actually occurred. In this tutorial we'll be investigating how to implement additional data structures in our physics engine's broad phase to quickly determine pairs of objects which might be colliding, and pass the results on to the narrow phase for resolution.

The Broad Phase and Narrow Phase

In the broad phase, the physics engine will determine which objects can possibly collide against each other, and then store the potential object collision in a list - this is often known as a *collision pair*. Then, in the narrow phase, the engine iterates over the list of potential collision pairs, and determines whether they really are colliding, and if so, resolves the collision. This means that if the broad phase is somehow 'clever' enough to know which objects may potentially be colliding (or conversely which definitely *cannot* be colliding), then we can reduce the number of collision checks we do to less than the n^2 worse case scenario we saw in the previous tutorial.

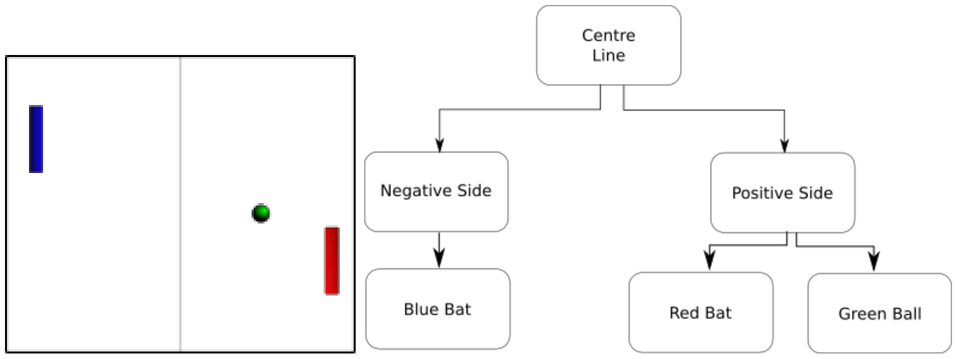
Let's have a think of a high level example of what a broad phase might be. If we were to try and make a simulation of the population of the entire world, we would have to check and see if the object that represents each person is colliding with any other person on Earth - that's *a lot* of collision checks. However, we intuitively know that there is no possibility that someone in London, UK is colliding with someone in Shanghai, China, so why run through all of the collision checks of all 8.5 million people in London, versus all 24 million people in Shanghai, when we know the results will always be false? If we can split up our world into discrete areas, where we know that the areas cannot overlap, and thus objects *within* those areas cannot overlap, we can potentially save a lot of collision detection tests.

Spatial Acceleration Structures

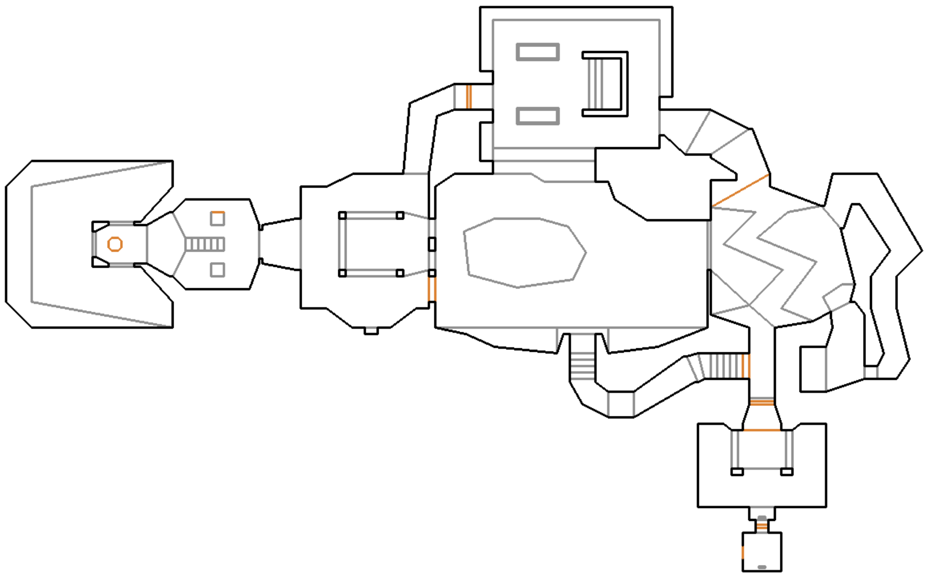
For the broad phase to perform its job, it needs some method of determining which areas of the world each object is in, and working out whether those objects in a particular area can collide. But what constitutes an 'area' exactly? And how do we go about splitting the world up into them? There are a number of ways we can split up the world, broadly categorized as *spatial acceleration structures* - that is, ways in which we can cut up space to make reasoning about it easier.

Binary Space Partitioning

In 1993, id Software's Doom demonstrated that it was possible to have a fully textured, seemingly '3D' world rendered efficiently on the 386 computers of the time. Part of the engineering magic behind its rendering was the use of *binary space partitioning* for its level design. This 'BSP' takes the form of a tree, where every node has a dividing plane (or infinite line, as really Doom was a 2D game with some clever rendering algorithms to make it appear 3D), and two child nodes, representing everything behind the plane, and in front of the plane. As a (very) simple example of this, here's a simple game of Pong, represented both visually, and as a very simple BSP that splits the world up along the 'net' line in the game:



To test for collisions using a BSP, we recursively walk through the tree, determining which side of the plane the tested object is - if it lies behind the BSP node's plane we check the left node, and if it's in front we check the right node - the 'binary' in BSP is that there's only two nodes. Any objects that are not in the same BSP region cannot be colliding and so do not need to be checked - in our simple example above, the blue bat cannot possibly be colliding with the ball as they aren't in the same half space defined by the root's plane, and so we don't need to check its collision shape against either the red bat or the ball. We could further subdivide the right hand node by finding another plane, creating another pair of child nodes that represent further subdivisions of the gameplay area, but we're quickly outgrowing our simple Pong example, as there's no other logical 'plane' that really makes sense. Instead, here's the very first level of Doom, seen from a top down view:

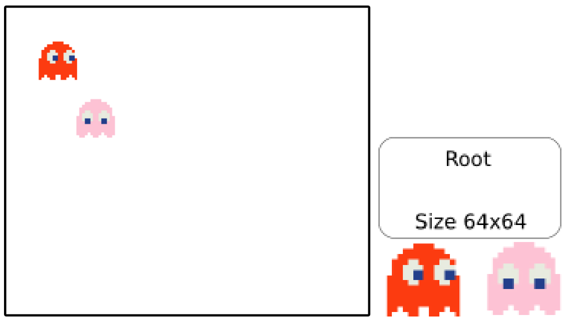


Each of those rooms and corridors is full of demons and collectible items like health and ammo - part of Doom's efficiency was in its ability to very quickly determine which items, walls, and floors to draw. To do this, the geometry of the world was split up using a BSP, to produce a number of *sectors*; convex regions of the world placed in the leaves of a BSP, which each dividing plane being selected from the world geometry.

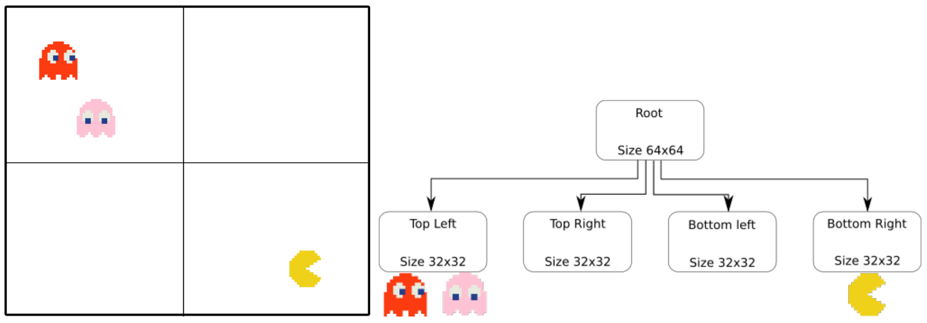
The process of selecting which geometry lines to use to subdivide the world requires a great deal of care. Ideally each split would divide the geometry in its parent node exactly in half, to provide the maximum reduction in processing when determining what collisions can happen. The levels in Doom (and the later *Quake* series) would undergo a a lengthy process of 'compilation' to produce an optimally balanced tree, often taking hours on the hardware of the time. While modern hardware can calculate plane equations much quicker than in 1993, the amount of geometry in a modern game level is also much higher, resulting in a difficult process of determining a good BSP, and which doesn't lend itself well to calculating in real time. Understanding how a BSP broadly works is still beneficial though, as many games have used them for static geometry over the years, making them historically significant.

Quadtrees

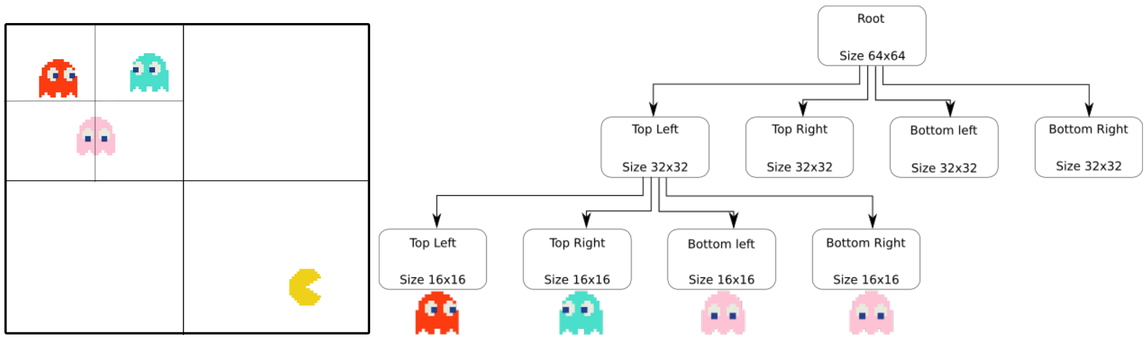
A popular method of partitioning the game world is the *Quadtree*; so named because it will split a parent node into 4 equally sized child nodes at a certain threshold. Each of the nodes of the tree has a size, and children of that node subdivide their parents space such that they fit inside it. For this reason you'll often see quadtrees defined with a size that is a power of 2 on each axis - you can think of the process of creating the tree as being similar to the MIPMapping we looked at earlier, where with enough subdivisions we'll end up with a structure that is 1x1 in size. To see this in action, lets investigate a small game scenario, of a small constrained game level containing a number of ghosts and a yellow player character. To create the quadtree, each ghost/character is inserted into the tree, testing its bounding volume against the size of the node - only if a character fits inside a node does it then go down that branch of the tree, similar to choosing the left/right side of a binary search tree depending on whether a value is greater or lesser than that node's value. Unlike the binary tree, a node can usually hold multiple values, rather than just a single value - these are the subsets of objects that must be tested for collisions between each other, and only if a game object is in the same node is there a chance of collision. If there are too many objects inserted into a particular node, that node splits, and the objects within it redistributed among its child nodes. Sometimes you will see quadtree nodes called 'buckets', as we try and fill them up until they start overflowing. Let's see how this process of filling buckets works, with our ghost game example, and an assumed maximum game object size per node of 2. Inserting the first two game objects into the quadtree is easy, they just both sit in the root node:



When we come to inserting the third object, however, we go over the object limit of the root, and must split it into 4 segments, retesting each object against the sizes of the children to see in which node they lie:



If we were to run a collision detection step at this point, we would only be testing 1 potential collision - the red ghost and the pink ghost lie in the same quadtree segment, and could be colliding. Our yellow player character sits in its own quadtree segment, so there's nothing it can be colliding with. Without a quadtree, we'd instead be performing 3 collision checks per frame (red vs pink, red vs player, and pink vs player). If we add in another ghost into the top left of the playing area, we again hit the limit of what the node can hold, and it must split:

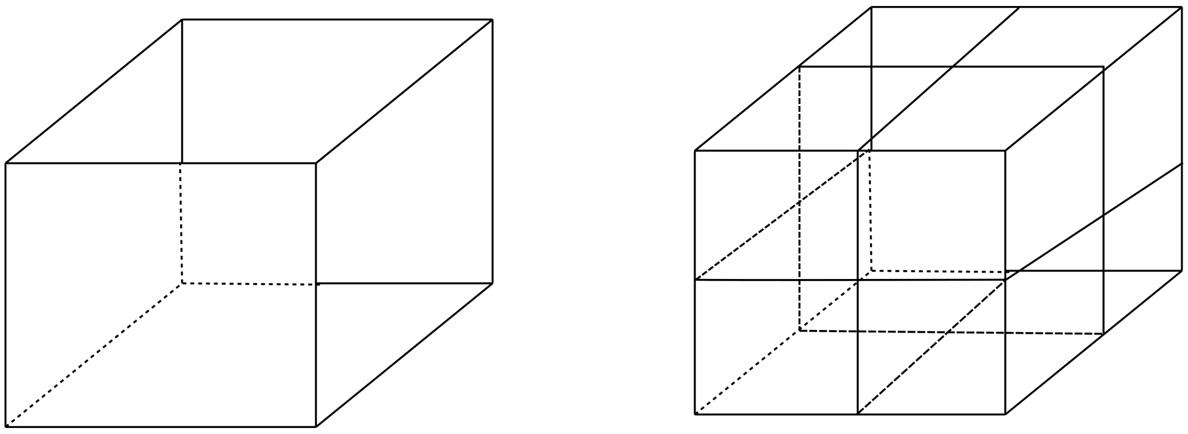


With the blue ghost inserted into the quadtree, we discover an interesting case for handling objects with a size - after the top left node is subdivided once more, the pink ghost sits *inbetween* two nodes. How do we pick which child node the pink ghost should be added to? The answer is it must be added to *both* nodes - if there was any other object in either node it *could* be colliding with the pink ghost, so the only reliable way to detect any possible collisions is to add it to both.

Once the pink ghost has been added to the quadtree, how many collisions will our broadphase have to try and detect? None! There's no node with multiple objects in it, and therefore no chance of collisions. If we didn't use a spatial acceleration structure of some sort, we would instead be calculating 6 collision detections per frame (red vs pink, red vs blue, pink vs blue, red vs character, pink vs character, and blue vs character), so this can be a significant saving in cases where game objects have complex collision volumes (AABBs, or arbitrary covex hulls).

Octrees

The above example of a quadtree only takes into consideration 2 dimensions. This is fine for games which mostly take place on a flat arena; even something like *Rocket League*, although it is '3D' in the sense that objects can move in a full 3 dimensions, is fairly 'flat' in that the game objects are more likely to be in front or behind another, rather than above or below, so its acceptable to discount the third axis for the purpose of a broadphase collision check. But how about something more spatially complex, like *MineCraft*, where character A may have the same x and z coordinates as character B, but be separated by miles of underground tunnels and rocks on the y axis? We can extend the concept of a quadtree to 3 dimensions, making an *octree*. As the name suggests, rather than a node splitting into *4* equally sized children, and octree node will split into *8* equally sized children, split like so:



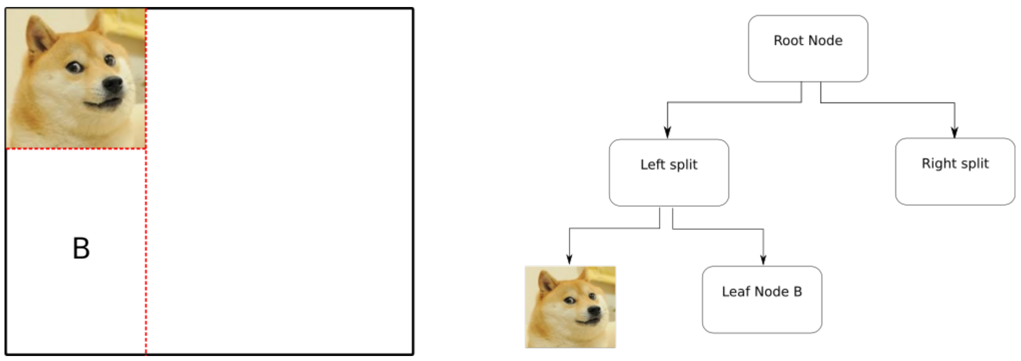
This allows nodes that are above or below each other, but separated by a significant distance, to be split into separate nodes, so that the broadphase collision test does not consider them as a potential collision. The actual process of handling and inserting objects into the octree is exactly the same as with the octree, except that we test for collisions against a full AABB per octree node, rather than just a flat 2D square box.

*k*D-Trees

In the above quadtree and octrees, a node was always split up into equally sized volumes. There's a similar structure known as a kDTree, where node expansion can be split up to 'fit' nodes better to the area of the objects being added.

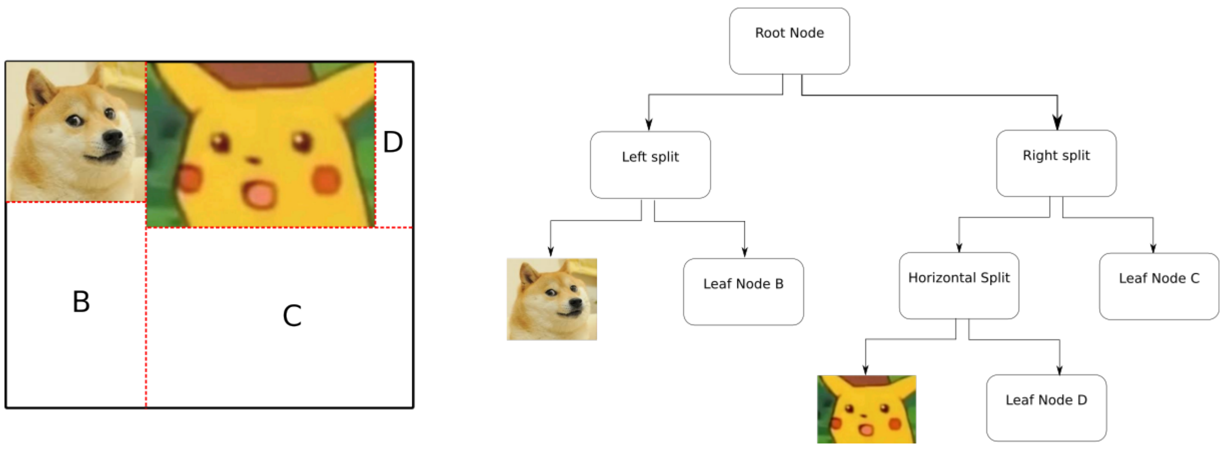
The main drawback to a kDTree is that by fitting the tree nodes exactly to the volume of an object, if that object moves in any way, the tree must be rebuilt, whereas a quad or octree only need their nodes rebuilding when a object moves sufficiently to enter another child node. This makes kDTrees more difficult to use, and its not entirely intuitive how a node should split, or what to do if more nodes must be inserted into the tree after it is constructed. For these reasons kDTrees are not often used as a spatial acceleration structure for physics calculations.

So why would we ever use a kDTree? While no good for physics, they are good for subdividing a volume in entirely static situations, or for determining whether an object of volume A can fit *anywhere* inside the tree (is there an unused leaf node of at least size A somewhere in the tree structure?). This is often used in the creation of a *texture atlas*, a 2D texture containing a number of sub-textures, often used for 'old school' 2D games, or for packing up all of the texture's used within a game's UI, so that the entire UI screen can be rendered with only a single texture binding. In this case, we're taking a source texture, and then copying its contents into a new, larger 2D texture, and determining the location based on a kDTree lookup. As each texture to be added is processed through the kDTree, a node is found that will fit the texture, where it will descend through to the next level, until a leaf node is found - the node will then be split in two, with the new texture being added to one node, and the other remaining as an empty node for adding new textures to later.



After one iteration of inserting a 2D area into the kDTree we'd get the following tree

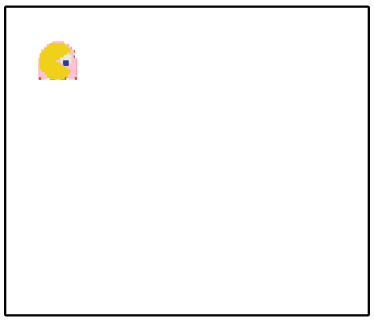
This plays to the strengths of the kDTree, as each 'node' is fitted to exactly the dimensions of the object it contains (an unchanging 2D texture), and can place it as necessary within its volume to avoid leaving any spare space around a node that may be hard to fill.



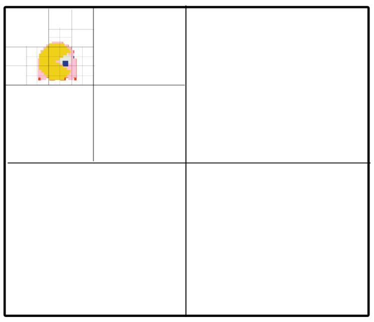
The second 2D area to add is too large for the left split area, and is instead added to the right child node, which is further split, giving us node C

Tree considerations

The tree examples above help show off the benefits of quadtrees and octrees, but they do hide a couple of important things that must be considered when generating acceleration structures for our physics systems. Think about a case where, either through bad luck with the physics integration of position, or via a mistake in object generation code, we end up in a situation where multiple objects end up at the same position:



In this example, a pink ghost, a red ghost, and a yellow character are all in the same position. If we assume that there's a maximum of 2 objects contained within a node, then we must split the node into 4 children:



The problem is, since there's such an overlap, every child node that is split up *still* contains 3 objects, and will get split again. While this splitting and halving of the sizes will mean that eventually child nodes around the edge of the collision objects eventually stop splitting, every node 'inside' the objects will subdivide *forever!* For this reason, we generally define a maximum *depth* of an octree or quadtree; a leaf node at the maximum depth down the tree won't be subdivided any more, and will instead just store more objects within it than a leaf node 'further up' the tree. This still causes a lot of nodes to be created in the worst case scenario of objects full overlapping, but places a maximum limit on the time taken to generate a tree, hopefully allowing the next physics update to detect the collision and separate the objects.

Another consideration to make is how many child nodes to actually instantiate when splitting a parent node. In the example above, the top right, bottom left, and bottom right child nodes of the root were instantiated along with the top left where the 3 objects were. That's possibly 3 object constructions, and depending on the memory model used, perhaps 3 calls to allocate memory, that go completely wasted in this case, as nothing goes in them. It may instead be better to leave the 'unfilled' nodes unallocated until something requires them, and only if there's to be a collision with that node instantiate it.

Node size and position

In the above discussion on deferring allocation of nodes until required, you may find yourself thinking how could a collision test be made on a node that doesn't exist - surely we need to store a position and size of our node in a struct or class somewhere before we can perform intersection tests on it? Perhaps not! The structure of quad and octrees is such that we gain two properties: first, we can use recursive functions to walk through the tree from the root to perform operations, and second, we can derive the properties of a child node from its parent, as long as we know 'which' child it is (the top left or top right node, for a quadtree example). Together, this means we can calculate a node's properties on the fly as we walk the tree! If we know that the root's size is 64, we can quickly calculate that every child 3 levels down is of size 8. The position is only harder to determine, and can be easiest to determine by using parameters into recursive functions to calculate it as necessary.

Memory Usage

You might also be wondering how to store the objects within a quadtree node. A simple solution might be to use a **std::vector** for every node, and emplace pointers to the objects in it as they are inserted into the leaf nodes. That certainly works, but may be memory inefficient. Under the hood, an **std::vector** uses a memory allocation to store its objects in, and whenever that memory allocation is filled up, it must ask the operating system to reallocate a larger chunk of memory. Asking the OS for memory is *slow*, so having to ask it on every node construction, and every time the allocation is filled up, takes time, and slows down the creation of our 'acceleration' structure! it may well be better to use a single large allocation for the entire tree, which can be passed around as a parameter in the recursive functions, where each leaf node that stores an object just emplaces into the single structure, and only stores an index to the first element within it, and a count of how many elements are within it. This, combined with the idea that we might not even need to store the position and size of a node, can make a quadtree or octree node a very efficient data structure.

Static and dynamic objects

Large game worlds can be built up of many hundreds of static objects, that don't necessarily move (i.e they have an inverse mass of 0), but do have collision volumes to impede the player's progress. Think of all of the walls and platforms that make up the levels in an FPS game like *Call of Duty* or *Overwatch* - they never move or get destroyed, but do stop bullets and players from moving through them. So, do these games construct a whole spatial acceleration structure for those 100s of objects every frame? Probably not - that would take far too long every frame to compute an answer that would be the same every frame anyway, except for the moving players and projectiles. For this reason, it may be the case that you have not one, but two spatial acceleration structures - one for *static* objects (the walls and floors of the world), and one for *dynamic* objects (the players, their projectiles, the NPCs, and so on). Then, the acceleration structure for static objects need only be rebuilt if something fundamentally changes about the world and its static objects (perhaps an objective is to blow up a wall, which should be removed from the static structure upon completion). To test a dynamic object versus the static acceleration structure, rather than directly inserting it into a list, we simply need a function to return the nodes that the object *would* be inserted into, and then perform the collision tests on the objects within it that way - as we already know that the static objects aren't possibly colliding, then this is just an O(n) operation, rather than the O(n^2) test we would usually perform for each area with multiple objects within it.

Sleeping Objects

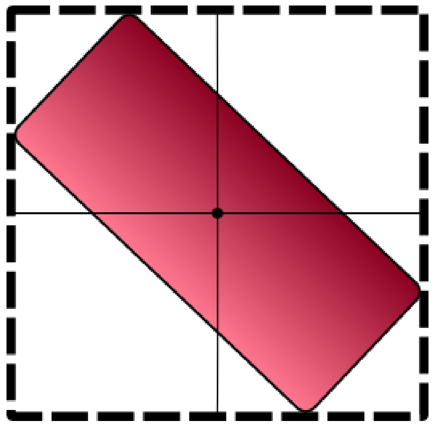
We can also use our knowledge on the state of the physics objects in our simulation to reduce the number of collision tests that occur every frame. In many games, the game level the player is in is populated with a preset number of enemies that they must face - a level designer could have added 10 enemies behind a door, that stay still and don't do anything until the player opens the door, at which point they become 'active' and attack the player. These objects don't quite fit the definition of a 'static' object defined earlier; although they are just standing still until the player enters the room, they will *eventually* move, and thus shouldn't be 'baked' into the same acceleration structure as the walls and floor of the room they are in. So does that mean that we must insert all 10 enemies into a 'dynamic' structure, and then test for collisions between them all each frame? Perhaps not! In this example, the enemies aren't moving, or rotating, or doing anything that would otherwise cause a collision between them. We can state that such an object is 'at rest', and then extend this further to say that two objects that are both at rest cannot be colliding, and therefore there's no point running collision detection between them. If, for any particular area in our acceleration structure, there's no *moving* objects in it, then there's no point running the detection at all, as it will never find a pair of colliding objects!

When inserting an object into an acceleration structure, we can check whether the object is 'sleeping' or not - that is, whether it will move in any way this frame. We can then also store an additional state in the acceleration structure node or not - whether it is currently an 'awake' node or not. We can assume that by default it is not 'awake', and only switch to that state if an awake object is inserted into it. If, after all objects are inserted, there's no awake object in a particular node, then it can be skipped when determining possible collision pairs, as there can't possibly be any in the node. We can extend this further, and set a node's 'awake' state to true when a child node becomes awake due to an active project being inserted into it. That would then mean that, if a node is not awake, then no further recursive processing has to occur in any of its children, as they will not be awake either, and entire branches of the tree can be skipped when determining collision pairs.

Inserting Objects

In this tutorial series so far, we've seen a number of collision volume types - spheres, axis aligned boxes, and oriented bounding boxes. How do we determine which of our tree structure nodes our collision volumes are in? One thing you've probably noticed is that the nodes in a quadtree and an octree are *axis aligned*, meaning we can determine overlaps between our tree shape and our objects using an axis aligned bounding box test. We can take this further, and say that, at the broad phase level, *everything* is an axis aligned bounding box; we've seen that AABB overlap tests can be quite cheap (the *AABBTest* method is only a couple of lines of code!), and as the purpose of the broad phase is to quickly determine possible collisions, we can generate our acceleration structure quicker if we use a quick test.

You might wonder how to turn our other collision shapes into AABBs for the broad phase collision test. To make a quick AABB from a bounding sphere is simple - we assume that each of the half-size axes of the test AABB are equal to the sphere's radius, and use the object's position as normal. What about an oriented bounding box? As ever with this volume type, it's tricky. Or rather, we're going to use a trick to compute it. If we have an object's orientation as a 3x3 matrix M, we can calculate the half sizes of an AABB that will encapsulate the OBB by transforming the object's half sizes by the absolute of matrix M - that is, setting every value in it to be positive.



Using these calculations, we can always have a set of suitable AABB sizes for use in the broadphase for each of the physics bodies in our simulation, which can then use the actual collision volume in the narrow phase, once the likely collision pairs have been found.

Choosing a structure

Reading up on these common structures, you've probably noticed that they're all tree structures, that split the world up recursively. Much as we can use a binary tree to make a *heap* structure that helps us sort and search arrays in an efficient manner, we can use the exact same concepts to help us determine which objects are related spatially - they would have a common parent. But which structure to use? The choice really comes down to the overall shape of your game world, and how often it changes. Plane tests are fast so it may initially seem that BSPs are the best choice. However, their efficiency is highly dependent on which planes are used to split the world - poor choices will lead to a very 'unbalanced' tree that does not reduce the number of test cases much per tree level. Choosing these planes and building the tree is not particularly fast, and not really suited to performing every frame, and so BSPs are usually only used to split up 'static' geometry that never moves or changes during the course of a game - the *Quake* series of games by id Software used BSP trees to represent world geometry, augmented this with an additional visibility matrix that stored which tree nodes could be seen from other nodes, allowing very fast access to which parts of the level to draw, or test collisions against.

For more dynamic scenes, like the test scene we have been building up over these tutorials, quadtrees and octrees are more suitable candidates - quads and boxes are still easy tests, but without some of the drawbacks of using planes - moving a plane could drastically change the shape of a BSP as geometry is moved to one side or the other, but moving the bounding box of a shape into some new nodes is unlikely to change more than a small number of leaf nodes. Choosing between quadtrees and octrees is mostly dependent on the overall shape of the game level - in a reasonably 'flat' world like the stadium of *Rocket League* a quadtree would be fine (the only thing likely to be in the air is a single ball), whereas something that takes place in a tower block would be better off using an octree - the ability to split the y-axis up becomes useful when geometry is stacked on top of each other.

Tutorial Code

To demonstrate the broadphase and narrow phase, we're going to replace the earlier *BasicCollisionDetection* method of the **PhysicsSystem** class with two new methods, appropriately enough called *BroadPhase* and *NarrowPhase*. To act as our spatial acceleration structure, we're going to implement a simple quadtree - this will be enough to demonstrate how tree-based insertion of objects *could* work, and the implementation we use here can be improved and built upon in many areas.

QuadTreeNode Class

To make things *interesting*, we're going to implement our quadtree as a **templated** type, allowing us to reuse the quadtree code for anything else we can think of. To do this, we need three templated classes - **QuadTreeNode** to represent an individual part of our tree (which will be filled with a number of **QuadTreeEntries**, each with a position and size), and **QuadTree** to hold the tree itself. As templates have to all be defined within a header, things are a bit more complex than usual to describe, but we'll start off with an outline of the **QuadTreeNode** class:

```
namespace NCL {
   using namespace NCL::Maths;
   namespace CSC8503 {
      template<class T>
      class QuadTree;

      template<class T>
      struct QuadTreeEntry {
         Vector3 pos;
         Vector3 size;
         T object;

         QuadTreeEntry(T obj, Vector3 pos, Vector3 size) {
            object      = obj;
            this->pos   = pos;
            this->size  = size;
         }
      };

      template<class T>
      class QuadTreeNode   {
      public:
         typedef std::function<
            void(std::list<QuadTreeEntry<T>>&)> QuadTreeFunc;
```

```cpp
25       protected:
26          friend class QuadTree<T>;
27 
28          QuadTreeNode() {}
29          QuadTreeNode(Vector2 pos, Vector2 size)  {
30             children       = nullptr;
31             this->position = pos;
32             this->size     = size;
33          }
34          ~QuadTreeNode() {delete[] children;}
35       protected:
36          std::list< QuadTreeEntry<T> > contents;
37 
38          Vector2 position;
39          Vector2 size;
40 
41          QuadTreeNode<T>* children;
42       };
43    }
44 }
```

QuadTreeNode Class header

Each node in our quadtree will hold a list of some templated type **T** (we'll be using it to store **GameObjects**, but it could be anything!), along with the positions and sizes of the AABB representing the inserted object (in the **QuadTreeEntry** struct), and will have a *position*, and a *size* described in 2D. As each node may at some point split, we'll also have a pointer to some *child* nodes - these *must* be pointers, as otherwise they'd always be instantiated, and our quadtree would recursively construct new nodes forever as soon as we instantiated one. So far, the only other important thing to note is another typedef - this is giving us a shorthand form for a function pointer, that will allow us to operate on the contents of our quadtree. We'll be doing this through the *OperateOnContents* method, which should be added to the public part of the **QuadTreeNode** class:

```cpp
 1 void OperateOnContents(QuadTreeFunc& func) {
 2    if (children) {
 3       for (int i = 0; i < 4; ++i) {
 4          children[i].OperateOnContents(func);
 5       }
 6    }
 7    else {
 8       if (!contents.empty()) {
 9          func(contents);
10       }
11    }
12 }
```

QuadTreeNode::OperateOnContents method

Our quadtree is being designed with the assumption that the only real contents of it will be in the leaf nodes, so we need to check to see if any particular quad tree node has children, and if so recursively call the method until we drop down to some leaf nodes - to save calling any passed in functions on empty lists, we also check to see whether there's actually any contents before calling the passed in function pointer, too (line 41).

At some point, a quadtree node will have to split up into 4 child nodes that when combined, fill the same area (we are, in a way *tessellating* our area). To handle this, we need to add another new method to the **QuadTreeNode** class, *Split*:

```
void Split() {
   Vector2 halfSize = size / 2.0f;
   children = new QuadTreeNode<T>[4];
   children[0] = QuadTreeNode<T>(position +
      Vector2(-halfSize.x, halfSize.y), halfSize);
   children[1] = QuadTreeNode<T>(position +
      Vector2(halfSize.x, halfSize.y), halfSize);
   children[2] = QuadTreeNode<T>(position +
      Vector2(-halfSize.x, -halfSize.y), halfSize);
   children[3] = QuadTreeNode<T>(position +
      Vector2(halfSize.x, -halfSize.y), halfSize);
}
```

QuadTreeNode::Split method

We are also assuming that every quadtree will have either 4 or 0 children all instantiated at once - using some addition and subtraction we can define the 4 points offset from the centre of a node that its child nodes will sit at to fill the area correctly.

The only other thing our **QuadTreeNode** needs to do is allow objects to be inserted into it. We're assuming that each node has a maximum capacity, which, if reached, will cause the node to split, and move its contents into its new child nodes, instead. This is all handled with one final function, *Insert*, which will take some object, at a position in the world, with a given size, and try and insert it into either its own list, or the lists of its child nodes. Add in this method to the **public** part of the **QuadTreeNode** class:

```
void Insert(T& object, const Vector3& objectPos,
      const Vector3& objectSize, int depthLeft, int maxSize) {
   if (!CollisionDetection::AABBTest(objectPos,
      Vector3(position.x, 0, position.y), objectSize,
      Vector3(size.x, 1000.0f, size.y))) {
      return;
   }
   if (children) { //not a leaf node, just descend the tree
      for (int i = 0; i < 4; ++i) {
         children[i].Insert(object, objectPos, objectSize,
                  depthLeft - 1, maxSize);
      }
   }
   else { //currently a leaf node, can just expand
      contents.push_back(QuadTreeEntry<T>(object,objectPos,objectSize));
      if ((int)contents.size() > maxSize && depthLeft > 0) {
         if (!children) {
            Split();
            //we need to reinsert the contents so far!
            for (const auto& i : contents) {
               for (int j = 0; j < 4; ++j) {
                  auto entry = i;
                  children[j].Insert(entry.object, entry.pos,
                     entry.size, depthLeft - 1, maxSize);
               }
            }
            contents.clear(); //contents now distributed!
         }
      }
   }
}
```

QuadTreeNode::Insert method

The first thing a quad tree node must do is an intersection test (line 3), to see if the object being added has an AABB (represented by the objectPos and objectSize variables) that overlaps with the AABB of itself - while our quadtree is really 'flat', we're artificially going to extend it on the y axis, and use the x and y of the node's position and size as the x and z coordinates of the world (this is fine for a top down 3D world like the tutorial code, but for a side-on game, we might just use the x and y coordinates as-is). If the AABBs *don't* overlap (using the intersection test code we added a few tutorials ago), we can just **return** - if an object doesn't overlap a node, it also cannot overlap any of the *children* of that node (line 6). If the object *does* overlap, the node needs to either recursively try and add the object to its children (if its not a leaf node, on line 8), or add the object to its own list of stored objects (line 15). We're not quite done yet, though - a quad tree node has a maximum number of objects it can contain, and once that has been exceeded, the node must split itself up (line 18), *unless* we've got to the maximum tree depth we are allowing for the quadtree (line 16). If the node splits, it should recursively call *Insert* for each object it previously contained, to try and add it to each child's list. Once this is done, the node can remove its own contents, as they will be *somewhere* in the child nodes (line 27).

QuadTree Class

The **QuadTree** class itself is pretty simple, as it's mostly a wrapper around a **QuadTreeNode** that will be our tree root. We define the class in the same namespace setup as the QuadTreeNode, and, just like that class, must declare that our **QuadTree** is a templated type (line 4). We only need 3 member variables for our simple quadtree demonstration - the root node, and integers to represent the maximum depth the tree will be allowed to extend to, and a maximum size to limit how many objects should be contained within each node. The **constructor** of the **QuadTree** will take in a **Vector2** that represents the dimensions of the tree, and defaulted integers for the size and depth. The rest of the public methods we're going to need are more or less 'pass through' methods to insert objects into the root, and to receive a function to call on any contents of our tree - this'll make a bit more sense when we see this feature in action shortly.

```
1  namespace NCL {
2     using namespace NCL::Maths;
3     namespace CSC8503 {
4        template<class T>
5        class QuadTree {
6        public:
7           QuadTree(Vector2 size, int maxDepth = 6, int maxSize = 5) {
8              root = QuadTreeNode<T>(Vector2(), size);
9              this->maxDepth = maxDepth;
10             this->maxSize  = maxSize;
11          }
12          ~QuadTree() {}
13          void Insert(T object,const Vector3&pos,const Vector3&size) {
14             root.Insert(object, pos, size, maxDepth, maxSize);
15          }
16          void OperateOnContents(
17                typename QuadTreeNode<T>::QuadTreeFunc  func) {
18             root.OperateOnContents(func);
19          }
20       protected:
21          QuadTreeNode<T> root;
22          int   maxDepth;   int   maxSize;
23       };
24    }
25 }
```

QuadTree Class header

GameObject Class

That's all we need to *build* a quadtree, but *using* it will take a little more work. We need to be able to turn any of the collision volumes we've been using for our **GameObjects** into an AABB for insertion into the quadtree - the AABB position is easy (it's just the object's position), but getting the box half sizes takes a bit more effort. To do this, we need to add a couple of new methods and a variable to the **GameObject** class - *UpdateBroadphaseAABB* will fill in the correct half sizes into the *broadphaseAABB* variable, and *GetBroadphaseAABB* will return it (as a reference, allowing us to also get a true or false answer as to whether we even have a collision volume).

```
public:
bool GetBroadphaseAABB(Vector3&outsize) const;
void UpdateBroadphaseAABB();
protected:
Vector3 broadphaseAABB;
```

GameObject Class header

Here's the contents of the two functions. Note that we could instead add a **virtual method** to the **CollisionVolume** class to greatly reduce the amount of code we'd need here, but that'll make each volume gain a vtable pointer, which we might want to avoid. Getting the extents of an AABB is clearly trivial (line 14), and not much trickier for a sphere (line 18), but OBBs require us to use the absolute values of a rotation matrix (line 22+23), which will transform the local space half sizes into larger ones that will envelop any rotation in world space.

```
 1 bool GameObject::GetBroadphaseAABB(Vector3&outSize) const {
 2    if (!boundingVolume) {
 3       return false;
 4    }
 5    outSize = broadphaseAABB;
 6    return true;
 7 }
 8
 9 void GameObject::UpdateBroadphaseAABB() {
10    if (!boundingVolume) {
11       return;
12    }
13    if (boundingVolume->type == VolumeType::AABB) {
14       broadphaseAABB =
15          ((AABBVolume&)*boundingVolume).GetHalfDimensions();
16    }
17    else if (boundingVolume->type == VolumeType::Sphere) {
18       float r = ((SphereVolume&)*boundingVolume).GetRadius();
19       broadphaseAABB = Vector3(r, r, r);
20    }
21    else if (boundingVolume->type == VolumeType::OBB) {
22       Matrix3 mat = Matrix3(transform.GetWorldOrientation());
23       mat = mat.Absolute();
24       Vector3 halfSizes =
25          ((OBBVolume&)*boundingVolume).GetHalfDimensions();
26       broadphaseAABB = mat * halfSizes;
27    }
28 }
```

GameObject Class file

PhysicsSystem Class

To use our quadtree as a broadphase structure, we need to find a few currently empty methods within the **PhysicsSystem** class, *BroadPhase*,*NarrowPhase*, and *UpdateObjectAABBs*. The class has been set to use these instead of *BasicCollisionDetection* if the *useBroadPhase* variable has been set to true in the **constructor**, so you might want to set that now, or add a key to toggle it at will.

We also need to add in a new variable to the **PhysicsSystem**, in the **protected** region:

```
    std::set<CollisionDetection::CollisionInfo> broadphaseCollisions;
```

PhysicsSystem header

We'll start off with the *BroadPhase* method. In it, we're going to do two things - first, create a *QuadTree* out of all of the objects in the scene, and secondly, traverse the tree to get to each leaf node, and add any *unique* collisions to a global collisions list to further process in the narrow phase. Remember, objects can end up in multiple tree nodes if their bounding volume intersects multiple quadtree node volumes, so there's a chance that the same pair of objects are in multiple nodes, so we have to detect this to make sure that objects don't have collision resolution calculated for them multiple times.

To start the *BroadPhase* function off, we're simply going to construct a *Quadtree* with some default parameters, and then iterate through all of the objects in the game world, Inserting them into the tree one after another:

```
void PhysicsSystem::BroadPhase() {
    broadphaseCollisions.clear();
    QuadTree<GameObject*> tree(Vector2(1024, 1024),7, 6);

    std::vector<GameObject*>::const_iterator first;
    std::vector<GameObject*>::const_iterator last;
    gameWorld.GetObjectIterators(first, last);
    for (auto i = first; i != last; ++i) {
        Vector3 halfSizes;
        if (!(*i)->GetBroadphaseAABB(halfSizes)) {
            continue;
        }
        Vector3 pos = (*i)->GetTransform().GetPosition();
        tree.Insert(*i, pos, halfSizes);
    }
```

PhysicsSystem::BroadPhase method

To store the collision pairs for further processing, we're going to use a **std::set** *broadphaseCollisions*, stored as a member variable of the **PhysicsSystem**. A set will only hold unique values, and won't add any duplicates, so we can safely keep calling insert even on object pairs that might already be in the container (remember, our physics bodies may end up covering multiple quad tree nodes, but we only want to check their actual collisions once).

With the tree completed, we can determine which sets of objects may be colliding. To do this, we're going to use the *OperateOnContents* method of the **QuadTree**, which if you'll remember, will traverse the tree to find all of the leaf nodes, and then call a function that takes in the node's list of data as its parameter. We're going to define the function to pass in to the method right there in the parameter list, using a *lambda* function:

```
16    tree.OperateOnContents(
17       [&](std::list<QuadTreeEntry<GameObject*>>& data) {
18       CollisionDetection::CollisionInfo info;
19       for (auto i = data.begin(); i != data.end(); ++i) {
20          for (auto j = std::next(i); j != data.end(); ++j) {
21             //is this pair of items already in the collision set -
22             //if the same pair is in another quadtree node together etc
23             info.a = min((*i).object, (*j).object);
24             info.b = max((*i).object, (*j).object);
25             broadphaseCollisions.insert(info);
26          }
27       }
28    });
29 }
```

PhysicsSystem::BroadPhase method

As the lambda will be inserting potential collision pairs into the *broadphaseCollisions* variable of the **PhysicsSystem** that we made earlier, we must *capture* it, allowing us to use it without having to explicitly send it as a parameter, and make our **QuadTree** code more complex. To automatically capture all variables by reference, we need to simply add a & to the capture list of the lambda (the square brackets in front of the function definition). Inside the function itself, we're going to have a nested for loop to iterate through all of the objects inside the quad tree node, similar to how we did in the *BasicCollisionDetection* method earlier. This time though, we're not directly trying to perform collision detection, but to instead add the potential collisions to the set (line 26). To make life a little easier, we're going to define a strict ordering of objects in the collision pair, so that we can avoid adding a collision between objects A and B, and also B and A, as separate collisions. To do this, we're simply going to always make the object pointer with the lowest numerical value the 'a' object in the collision pair, and 'b' the higher. The **std::set** being used for the broad phase collision container relies on the **less than** operator being defined for our **CollisionInfo** struct. Here's the code for the operator, defined in the *CollisionDetection.h* file:

```
bool operator < (const CollisionInfo& other) const {
   size_t otherHash = (size_t)other.a->GetWorldID() +
                      ((size_t)other.b->GetWorldID() << 32);
   size_t thisHash  = (size_t)a->GetWorldID()       +
                      ((size_t)b-> GetWorldID() << 32);

   return (thisHash < otherHash);
}
```

CollisionInfo Struct operator declaration

To work out whether one collision pair is 'less than' another, the code performs a simple 'hashing' function, based on the IDs of each object - this is an integer value created by the **GameWorld** class when an object is added to it. We can take two 32 bit values and turn them into one 64 bit value by bit shifting one of the numbers by 32 bits, and then adding the results together. This then gives us a simple number that can be used to sort collision pairs. As each object in the world has a unique ID, the addition of these two shifted numbers should also be unique for every potential pair of objects in the world. You could save a bit of processing time by calculating this hashed value once, at the same time as the two objects are set in the **CollisionInfo** struct, but the method does work as it is right now.

That's all we need for our example broad phase, so now for the narrow phase. This function is much shorter - we simply iterate through all of the collisions added to the *allCollisions* list, and if the two collision volumes are actually intersecting (this checks the 'true' volume of each shape, not the AABB we used to fill in the broad phase structure), call the *ImpulseResolveCollision* method we made earlier in the tutorial series to apply the correct forces to separate them out.

```
void PhysicsSystem::NarrowPhase() {
   for (std::set<CollisionDetection::CollisionInfo>::iterator
      i = broadphaseCollisions.begin();
      i != broadphaseCollisions.end(); ++i) {
      CollisionDetection::CollisionInfo info = *i;
      if (CollisionDetection::ObjectIntersection(info.a,info.b,info)){
         info.framesLeft = numCollisionFrames;
         ImpulseResolveCollision(*info.a, *info.b, info.point);
         allCollisions.insert(info); //insert into our main set
      }
   }
}
```

PhysicsSystem::NarrowPhase method

Lastly, we need to make a new method *UpdateObjectAABBs*, and fill it with the following code, and then call it every game frame from the **PhysicsSystem** *Update* method:

```
void PhysicsSystem::UpdateObjectAABBs() {
   std::vector<GameObject*>::const_iterator first;
   std::vector<GameObject*>::const_iterator last;
   gameWorld.GetObjectIterators(first, last);
   for (auto i = first; i != last; ++i) {
      (*i)->UpdateBroadphaseAABB();
   }
}
```

PhysicsSystem::UpdateObjectAABBs method

This will let the **PhysicsSystem** see the latest correct bounding boxes for the game objects every frame - if you never change the sizes, you could instead just call *UpdateBroadphaseAABB* once when an object is constructed.

Conclusion

With these changes, we've implemented a basic broad phase and narrow phase to our physics engine. The *broad phase* is designed to overall reduce the number of collision detection calculations that are performed each frame. Instead of testing every object against every other object, we can, with careful use of a spatial partitioning structure, only check objects near each other, and instead of having one huge n^2 collision test, have a greater number of smaller n^2 tests. Once this reduced set of test cases has been calculated, the *narrow phase* can then apply the correct forces to resolve our collisions, and update any of our function callbacks.

Further Work

1) Spatial acceleration structures are useful for more than just the broadphase of your physics engine. They can also be used to speed up your raycasting - if a ray doesn't intersect with the bounding box that represents an octree node, then it stands to reason that it cannot intersect any of the objects within it, potentially saving many ray intersection tests. Try to modify the Raycast methods introduced in the first tutorial to build a list of objects to test based on whether the ray intersects the nodes of your quadtree or octree.

2) Spatial acceleration structures are useful outside of the scope of physics, too. We have learnt previously that we can speed up the rendering of the scene by performing frustum culling to build up a list of scene nodes that are within the viewpoint of the game camera. Consider how you would use plane intersection tests on the collision volumes of your quadtree or octree nodes to get a list of game objects to render.

3) The quadtree example outlined here is functional, but naive - there's plenty of room for performance improvements! Consider storing a **std::vector** of discarded quadtree nodes between frames, and using those instead of calling **new** and **delete** every frame.

4) Perhaps having a single acceleration structure built up every frame isn't the best way to speed up the physics calculations - it'll be repeating the same work every frame for the inclusion of all of your static objects (that is, ones with infinite mass that will never move, such as walls and the floor). Consider having two structures, one for the static objects, and one for the dynamic objects. You may need to then create a method in the **QuadTree** class that tries to determine which nodes a dynamic object intersects with, but rather than inserting it into the tree, just then calls a function on the contents of the node - allowing you to then run intersection code against just the shapes in the area of the object.