Collision Detection and Response

Written by Melvyn May for Garage Games

30th March 2008

Rev #2

This document is split-up into three broad areas:

- **Collision Detection**
- **Collision Response**
- **Torque Game Builder**

Collision Detection

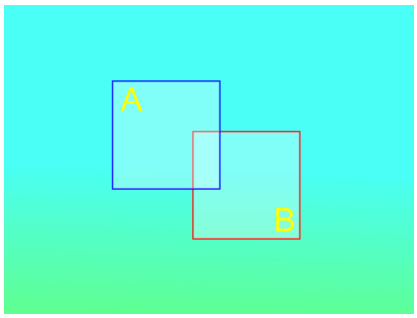
The intent of this article is to convey some of the complexities in a subject which at first seems like a trivial endeavour. As you'll see, that as the **freedom of movement** increases, so does the **complexity**. We say complexity in the negative not because we find the math too hard but rather that the more complex something is the more time your computer spends working it all out and that's time that could be spent better elsewhere or simply allowing your game to run on an even lower spec computer covering a wider audience which can mean more money for developers.

We won't be covering any math or in-depth implementation details here; we'll solely focus on conveying the issues involved. The hope then is that the reader can gain a better appreciation of what is involved. Although a majority of this article could be related to your own custom-engine, we'll take the opportunity to relate key points to aspects of the Torque Game Builder. In this way you can take away a broad knowledge of the subject and be able to use that within the TGB or elsewhere. To that end, we'll stick to operating in two dimensions (2D) but the content does scale up to three dimensions (3D) in most cases.

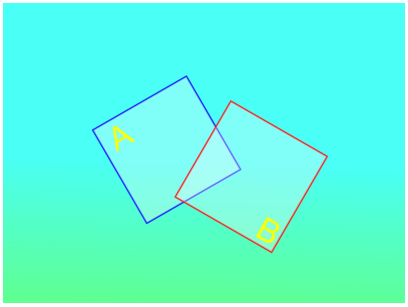
When two objects meet…

Let's start simple. We've got two objects in a scene (both squares) that are not moving, are not rotated with respect to each other and are just plain boring. Hold on; let's pretend that they're not just squares but hyper-transport intergalactic space-warriors doing combat to determine the outcome of the hyper-stellar games. Unfortunately the graphics are from the 1970's and the space-warriors are rendered as squares. You can't have everything.

So "intergalactic space-warriors" is a mouthful so let's call them object "A" and object "B" as follows:



As you can see the objects obviously overlap and there's a simple mathematical way of determining this but that isn't the topic here so you'll just have to believe us that it's simple. So we now know we can easily detect if two objects are overlapping but these are very simple objects and the world isn't that simple. By this we mean that they're not only simple geometrically (they're just squares) but they're not moving and are not rotated. So let's deal with rotation first. Let's rotate the objects and see what that looks like:

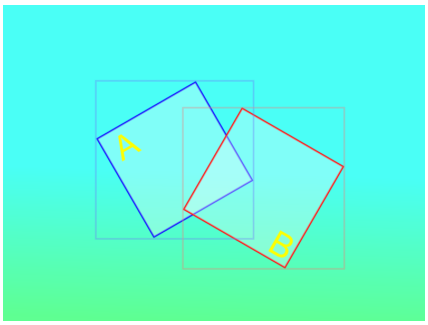


So you can see that the objects do overlap still even though they're rotated. The problem is now that the simple method of detecting whether these objects overlap becomes more complex now meaning it'll take longer to do. So why are rotated objects more complex than non-rotated objects? Surely something that isn't rotated is really rotated but just rotated at zero? If you thought that then you probably did well at geometry in class. The problem is indeed not that they're rotated but the fact that they're not rotated the same. As you can see above object "A" is rotated counter-clockwise 30° and object "B" is rotated clockwise 30°.

Without explaining what the “non-rotated” algorithm was it’s hard to appreciate why having the objects rotated differently becomes a problem but we’re not going into the math here today. So this is the first example of how adding some **freedom of movement** to the objects increases the **complexity**. So as stated previously, the complexity increasing means your computer will have to spend more time working on collision detection. That’s a little depressing as we know that we’ve not even got the objects moving yet and we know that will increase the complexity even further but don’t despair, there’s a solution at hand.

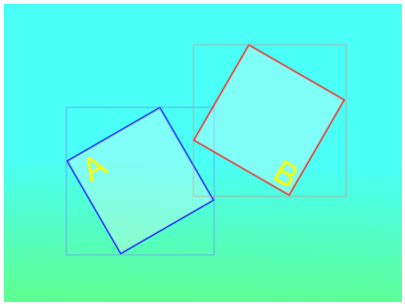
When you’re faced with a complex problem that you know you can’t simplify it’s always good to ask, “Is there a quick way I can check to see if I don’t have to fully work out the complex problem?” Put in collisions terms we’re asking “Is there a way I can check if the objects **definitely don’t** overlap?” This is a nice way of thinking about it. If we can quickly see if the objects have no chance of colliding then we can ignore them. If we can do this it has huge performance advantages as we no longer have to check every object against every other. This is quite often termed as “trivial rejection” meaning it is relatively trivial to reject certain objects from further (and more complex) collision detection.

So what quick ways are there to see if objects definitely don’t overlap? Keep in mind that when we say “quick”, we’re actually saying **quicker** then doing a full complex check, it’s all relative. So we didn’t answer the question did we; here’s a quick method for the rotated objects shown above:



These are the same as before but notice the non-rotated squares that surround the rotated objects. These are often called “bounding boxes” bearing in mind that they’re not really boxes in 2D, they’re squares but in a 3D engine they’d be boxes and lots of terminology comes from the 3D world. Great, now we’ve got some bounding-boxes that we can use the simple detection algorithm on to see if the objects overlap, or can we? We have to be careful here. We said previously that we’re looking for a quick way to see if objects **definitely don’t** overlap, to see if they do we have no choice other than using the full complex algorithm.

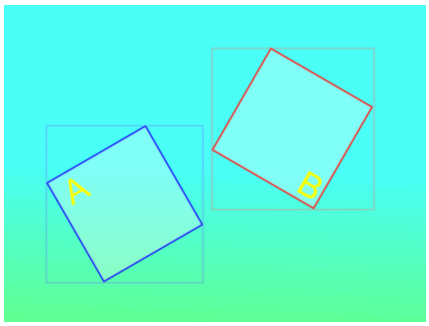
The example above though seems to show that the bounding-boxes do show overlap so what’s going on? Well, that’s the danger of looking at only one example, here’s another with the objects in slightly different positions:



So do the bounding-boxes alone show overlap now? Well, the bounding-boxes themselves overlap but look at the objects, they don’t overlap. This should highlight that the bounding-boxes can only determine if the objects definitely **don’t overlap**, you can’t use them to see if the objects do overlap. The downside to doing “trivial rejection” though is that we not only do the “trivial” check but we might have to do the full complex collision check if it tells us they “might” overlap.

So we spent even more time determining whether the objects overlap, what gives? Well, the hope is that most of the time things don’t overlap each other. Certainly you don’t get situations where every game object might overlap with every other so generally you’ll get a quicker solution. Like lots of things in the world, it’s a compromise.

So let's look at an example where the bounding-box is actually helping us show that the objects definitely don't overlap so we don't have to do a complex collision check:



In this case the simple check says they don't overlap and we saved time going any further, awesome!

Well as awesome as this might look, the astute game developer might say "hold on, what if I've got a thousand objects in my world? Are you saying I've got to check every objects bounding-box against every other objects bounding-box? That means each frame I've got to perform … does quick calculation … nearly a million bounding-box checks … no way!"

Well that game developer certainly understood the flaw in using bounding-boxes where the number of objects is not small. Above is an example of what is termed "exponential grow". This is something that all developers should avoid and there is nearly always a good way of avoiding it. Luckily there's a nice simple algorithm that works well with bounding-boxes to help avoid having to check every bounding-box with every other bounding-box.

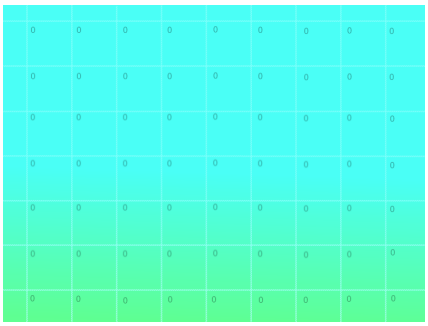
So before we go any further, let's just take a moment to understand where we are:

- We accepted that we have to do a complex check to see if rotated objects do overlap.
- We accepted that if we use bounding-boxes we can "trivially reject" some of these complex checks thus saving time.
- We accepted that doing bounding-box trivial rejection actual adds time but is nearly always outweighed by the total saving of not having to perform complex checks on all objects.
- We accepted the limitation of using bounding-boxes where if we've got lots (hundreds or thousands) of objects even this "trivial rejection" adds up to an awful lot of wasted time.

It's starting to sounds like we need a trivial rejection algorithm for the trivial rejection algorithm, when will this pain end? Well fear not, what's needed now is not so much a new algorithm as a new way of dealing with the essential bounding-box algorithm.

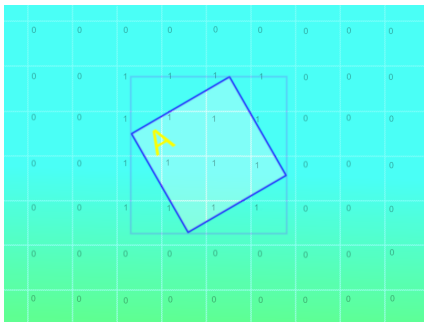
What we really need is to get the bounding-boxes and store them in such a way that we can take an object and immediately see which objects **might** overlap. Then we can proceed as described previously checking to see what the bounding-boxes tell us. So what is this mystical algorithm?

Bring on the spatial partitioning algorithm! That sounds awfully complex but it's a pretty simple idea that has massive advantages. Here's an image we'll use to describe exactly how it works:



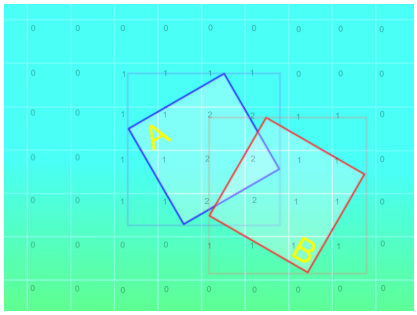
So what we've done here is split the world up into a grid. Each grid is called a "bin" (or bucket) and in these bins we can put references to our scene objects. We'll call all the bins the "scene-container". The number shown on each bin is how many objects it refers to and is called the "bin count" or "bin occupancy". Imagine each bin having a simple list of objects; the number is the size of the list. Everything has zero in it currently because there are no objects in the scene. Pretty dull.

So we add references to objects but why would we do that and how does it help us here? Well, let's add our object "A" back into the scene and see what that does:



Ignoring object "A" for a moment, look at the bin counts, some have increased to 1. What happened here was that bins that overlap the bounding-box of object "A" had a reference to object "A" added to each of their lists. As you can see, 16 bins have 1 in them to show this. Notice the top-left bin with 1 in it: That one doesn't overlap the actual object "A" but it does overlap the bounding-box. This is because the bins don't know anything about the actual shape of the object itself; they're representing the bounding-box only.

So this doesn't really help that much so let's add our object "B" and see what happens:



Again, ignoring objects "A" and "B" for a moment, look at the bin counts, some have increased to 1 whilst some have increased to 2. The bins marked with 2 contain references to both object "A" and object "B" therefore they each have 2 references in their list but again, so what? Well, let's not forget that the numbers only show how many objects are being referenced, in each bin there's an actual list of objects that overlap that bin.

Oh please answer the question, how is this helpful? Okay, here's how we do it but first let's forget what we did before when we explicitly considered checking if object "A" overlaps object "B". What we'll do now is only have knowledge of object "A"; we don't even know object "B" exists and we're certainly not going to ask the system for a list of objects which we'll check if object "A" overlaps. No, what we'll do now is know about object "A" and go to the bins and ask the old question, "What might object "A" overlap?"

We do this quite simply by getting the bounding-box of object "A" (which we know about) and asking the scene-container which object(s) are in those bins overlapping the bounding-box of object "A". If you look at the previous image and go from top-left to bottom-right of the bins overlapping the bounding-box of object "A" you'll get a list like "1,1,1,1,1,1,2,2,1,1,2,2,1,1,2,2" where these numbers refer to the following objects contained in each bins list respectively "A,A,A,A,A,A,AB,AB,A,A,AB,AB,A,A,AB,AB".

So that looks cryptic, what does it mean? Well let's go back and see what we asked. We asked the scene-container for the objects that overlap the bounding-box of object "A" and it returned the list of objects above. We obviously need to ignore object "A" itself as well as duplicates so if we do this to the list we get a list of "B" … one object! The scene-container says that object "B" **might** overlap object "A".

So isn't all that work actually slower than simply using the bounding-boxes? Well yes it is for a simple case of two bounding-boxes but that isn't a realistic scenario. Take our previous example of 1000 scene objects. Assume for a minute that object "A" and object "B" are two of those objects and the other 998 are elsewhere in the scene, nowhere near these objects.

If I wanted to see which objects overlapped object "A" I'd get a fast response of object "B" without the need to check the other 998 objects and that's a huge time saving thus performance increase. Indeed if we doubled the number of scene objects to 2000 we'd still take exactly the same amount of time to determine that only object "B" may overlap!

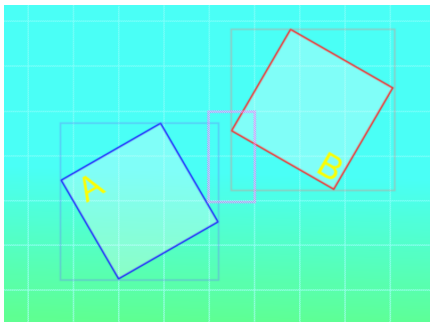
So this means that the time taken for the scene-container to answer our question is almost directly related to how many bins the source object occupies, in this case 16 bins. That's quite nice as there's nothing worse than building upon a system that has huge variations in time answering questions like this.

Putting it together…

So given the previous image as representative of our scene let's dry-run what we know and pretend we're the engine and someone asks us if object "A" overlaps any other object in the scene.

Okay, the question has been asked so we go to object "A" and retrieve its bounding-box. We then ask the scene-container which objects overlap this bounding-box and it gives us a list with only one object in it, object "B". We then check to see if the bounding-boxes of both objects "A" and "B" overlap to see if there may be an overlap. They do overlap so we now go to the more complex algorithm and check the geometry of the object shapes to see if they overlap and they do so we return the answer "yes, object "A" overlaps object "B"".

Notice above that even after the scene-container said the objects **may** overlap, we didn't jumped straight into doing the more complex collision detection algorithm but instead we went ahead and checked the bounding-boxes for overlap. Why did we do this? The reason is that it's possible for the area defined by the bounding-boxes to share the same bin but not actually overlap as illustrated here:

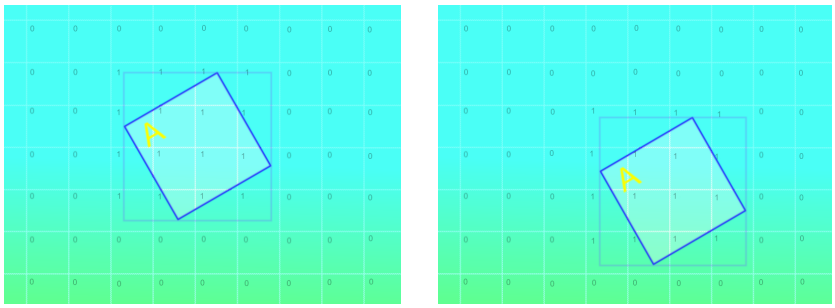


As highlighted by the magenta rectangle, these two bins will each have a reference to object "A" and object "B" but the bounding-boxes for both objects don't overlap. This is simply because of the size we chose for the container bins. If we were to create small bins then this might not have happened here, at least given the positions of the objects above. The point is that the scene-container will only report what's sharing the bins, it doesn't really understand about the spatial relationships but it can help us reduce what we've got to do by an enormous margin.

Let there be movement…

Previously we discussed the many issues of efficiently deciding if objects in a scene at arbitrary positions and rotations overlap and we came up with a nice and fast method of doing so. The only problem with this was, that none of the objects were actually moving, all were stationary.

So do moving objects mean we can't use the scene-container? Sure, we can use the container-system but we have to modify the scene-container whenever an object moves. Here's a before/after snapshot of the container-system as object "A" moves:



We have to adjust the container-system if the object moves position, rotates or is resized. The rule is that if it affects the bounding-box then we need to update the container-system which makes sense.

What this implies though is that every time we set the position, rotation or size of an object we need to update the container-system so there's a performance penalty associated with doing so but this is the price we gladly pay for not having extremely expensive collision-detection operations.

The Infinite World

So far we've avoided as much as possible, specific implementation details but there's one implementation detail when using a container-system that's so important, it's hard to ignore. The detail is a parameter of the container-system. We've already discussed one of the parameters, that being the bin size. This size is the "mapping" of the bin into the real world.

There's one extra parameter we'd like to introduce now and that's the parameter which controls how many bins there are in the scene. This might seem obvious and that we're making a drama over nothing but it has a rather interesting impact on how the container maps to the world.

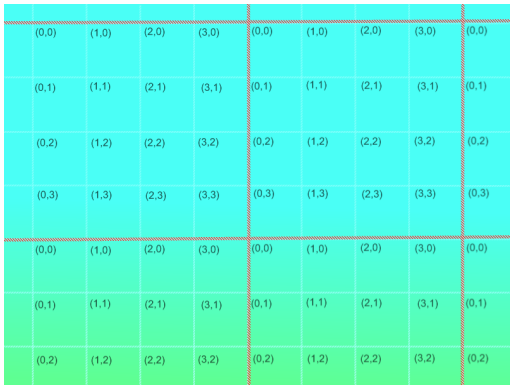
Given a world which has a finite or bounded coordinate system (it doesn't go on forever) you can choose the container bin sizes and how many quite easily. Given the size of the world and how big you want the container bins you divide one by the other and get how many bins are needed.

The problem comes when you don't know how big the world is or you simply don't want to put an artificial bound on its size. In this case you cannot expect to have an infinite number of container bins so what do you do?

The most common method is to be able to set the size and quantity of container bins and have them continually wrap around as you move through the world.

This is best illustrated by an image. Imagine we've set the container bin size to some value but we've set the quantity to be 4. This means that in both X/Y directions the container system will only maintain 4 bins giving a total of 4x4=16 container bins to serve the whole world.

As an example let's give each bin a coordinate (X, Y) starting at (0, 0). These coordinates are **not** world coordinates but simply a method for us to identify each container bin for illustration as follows:

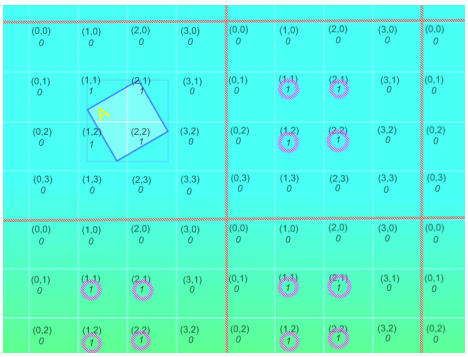


As you can see the container system "wraps" the bins. This means that you get infinite coverage of your world but the bins no longer map to a single location within the world.

This may sound bad until you consider what the container system set out to achieve. Its purpose was to reduce the complex collision checks by making the act of seeing if objects overlap, trivial. Even given the poorly configured system above, it could still do that albeit not as efficiently as you'd like.

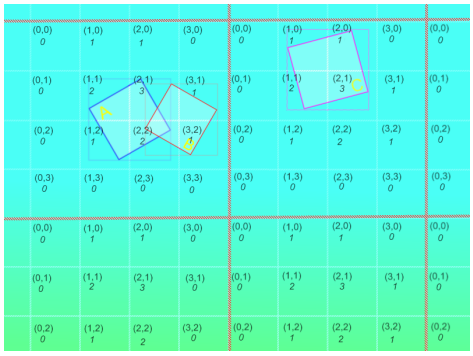
Another good reason for wrapping container bins is to keep memory consumption to a manageable level. You don't want hundreds of thousands of container bins in the world when you can have a small fraction of that and simply allow them to wrap and still get a boost in performance.

So what happens when we add in an object to the world if it's setup as above? Let's add in a smaller version of object "A" and see what it looks like:



As you can see, object "A" occupies 4 container bins namely (1, 1), (2, 1), (1, 2) and (2, 2). There's nothing unexpected here but notice that when those bins wrap (because they're the same bins) they show as being occupied by object "A". Although this looks like a bug, it's completely expected.

Let's put some other objects in the world and see how it affects what we're doing:



Take a minute to make sure you understand why the container bins have the counts they do. To help, here's a list of what's in each bin:

	0	1	2	3
0	-	C	C	-
1	-	A,C	A,B,C	B
2	-	A	A,B	B
3	-	-	-	-

So if we were to ask the container-system which objects might overlap object "A", it'd look at the bounding-box of object "A" and search for objects in the bins (1, 1), (2, 1), (1, 2) and (2, 2) which would return objects "B" and "C". Hopefully you should understand why object "C" was returned as it happens to occupy bins (1, 0), (2, 0), (1, 1) and (2, 1) of which (1, 1) and (2, 1) are also occupied by object "A".

So would object "C" be found to overlap object "A"? We shouldn't forget that the container-system doesn't do the complex determination of whether objects actually overlap; it just provides objects which **might** overlap. So given object "B" and object "C" we'd check their bounding-boxes and find that we only overlap object "B" so all is well.

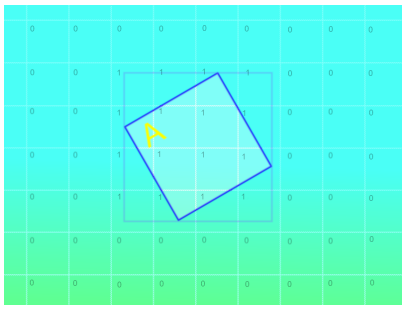
We say all is well but we were forced to consider object "C" when quite clearly it is nowhere near object "A". To rectify this we wouldn't configure the container-system anything like described above. We'd typically configure the container-system bins to be a useful size and quantity so that it better suited our scene.

If our scene isn't that big then we can quite happily have container bins that don't wrap (at least in the space we used) but if we did have a big scene then we'd need to consider balancing more bins with memory consumption. With that said, most container-systems don't use much memory per bin but if memory was at a premium then it's something that the developer should be aware of.

Container-System Penalties

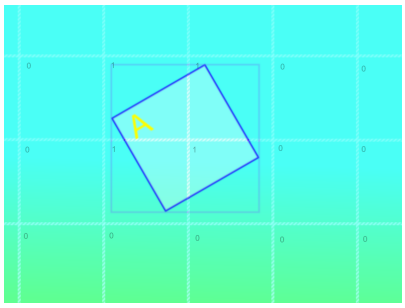
This kind of “calculate it now / get benefit later” is common when trying to offset the cost of an extremely expensive operation by making lots of less-expensive operations as things change. The problem though, is that this is a balancing act with a few parameters that hopefully have “safe” defaults.

The downside of the container-system may not be obvious at first but it’s easy to explain. Let’s bring back an image of object “A” setup in our scene-container:



As you can see, having the object “A” in the scene requires us to add 16 references to it. The reason for 16 is that the bounding-box of object “A” covers 16 container bins. If we increased the size of the object we’d cover more container bins. So the penalty is that as the objects get bigger, the cost of keeping them in the scene-container rises. Thankfully though the cost is relatively small but it can easily add up with lots of objects.

The container-system does have parameters to help with this though. One of the most common parameters is the ability to set the size of the bins themselves. Let’s double the size of the container bins and see how that affects object “A”:



So previously the bounding-box of object “A” overlapped 16 container bins. We doubled the container bin size and now we only require 4 container bins. That sounds like a better situation doesn’t it? Well yes and no.

Let’s take this to the extreme and make the container bins really big so one bin covers the whole game world. This means that objects only ever need 1 container bin. Unfortunately all objects use 1 container bin and it’s the same one for every object. When we ask which objects might overlap object “A” we’ll get all the objects in the scene and will have to test them all which isn’t that helpful.

So what’s the best size for the container bins? Well the answer depends on a number of factors and again, it’s a balancing act of time spent updating container bins due to the bounding-box of objects changing and reducing how many objects are returned for further complex collision processing.

The best way is to have some metrics available such as how many container bin updates, container searches for collision checks, collision objects returned and actual collisions are happening per-frame. With this knowledge you can tweak the size until you balance these metrics. That’s pretty advanced stuff and a decent container system will have some useful default for the container size anyway.

What's in a name?

So far we've gathered some funky names like "container-system", "bins", "trivial rejection" etc. It's worth adding to this list some more standard terms that describe the overall picture of what's trying to be achieved by all this.

When we say we're using a container-system to do trivial rejection to try to reduce doing the computationally expensive collision operations, we're talking about breaking down a process into "phases". This first phase that uses the container-system is known as the "broad-phase". The broad-phase reduces the potential list of objects to save time spent performing more expensive checks. When we're doing these expensive checks, we're in the "narrow-phase".

It's quite common in collision detection to hear the terms **broad-phase** and **narrow-phase**. They don't relate to any specific algorithm but rather relate to a "broad" (quick/trivial) process to reduce the potential processing that the "narrow" (expensive/non-trivial) process has to deal with. In our case we're using a spatial-partitioning scheme (container-bin-system) as the broad-phase and a more complex collision algorithm for the narrow-phase.

Contacts not overlaps

So far, believe it or not, we've kept examples as simple as possible but now we need to "turn the heat up". You may have noticed that the previous wording of what exactly was being determined was carefully chosen!

Looking back you'll notice that we're trying to determine whether one object **overlaps** another. If you do searches on the internet about collision detection for 2D game engines you'll find all sorts of checks for overlaps that mention bounding-boxes and determining if they overlap but not much on anything else.

What else is there you might ask? Well let's just state one thing: an **overlap** is a bad thing to have. The reason for this might not seem obvious as we've so far been working hard to detect this seemingly normal and expected situation but things are not what they seem.

Collision detection actually doesn't want to detect overlaps, it wants to prevent them! Let that one sink in for a while before you continue. Yes, overlaps are collision detection gone wrong. Let us explain this further.

If we had a game with two tanks that were moving towards each other and they became overlapped we could show some nice spark-effect when they did. The thing is that after the sparks have gone we're left with two tanks that overlap on the screen. This isn't real-life nevertheless we don't really want the tanks to actually overlap on the screen do we?

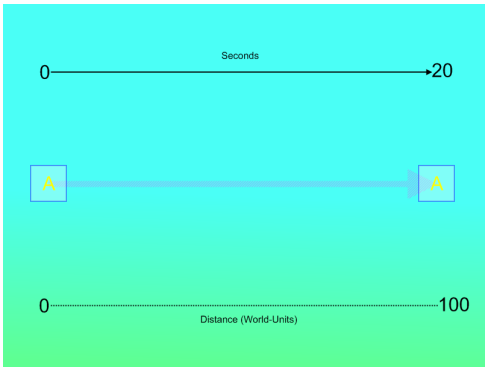
What we'd prefer is to have the tanks come in contact with each other but **not overlap**! This is much more realistic. Let's put it another way, if we did let them overlap then we've got to work out a way to move them out of the overlap, in other words separate them again. If our collision detection could detect a contact between tanks before an overlap occurred we'd be much happier.

Detecting contacts between objects is actually what collision detection is all about. Overlaps can still occur but in most cases they signify a bad situation that needs to be rectified.

So before we discuss this in more detail we've got to spend a little time talking about how objects actually move in a typical game world.

Integrated Moves

Consider the following image:



What we see here is object “A” moving right taking 20 seconds to reach its end-point which is 100 world-units away. We’re using “world units” here instead of meters or miles because it’s not that important what the real unit is.

So we can say that object “A” has a speed of 100/20 = 5 world-units per second. We can also say that object “A” has a velocity of 5 world-units per second moving right. Speed is how fast, velocity is speed but with a direction.

Put another way, 2D velocity is two speeds that are put together in a vector, one element for the X axis and one for the Y axis. The above can then simply be written as a velocity of (+5, 0) e.g. 5 world-units/sec in the X axis and 0 world-units/sec in the Y axis. The “+” indicates right on the diagram whereas “-” would indicate left. In the Y axis “+” indicates down whereas “-” indicates up.

Real game engines use vectors for things such as position and velocity so that’s why we’ve highlighted it here. For this discussion though we’ll simplify things and not use vectors so we’ll use speed instead of velocity where we can and say things like “a speed of 5 moving right”.

So what typically happens during those 20 seconds that the object is moving right? From the point of view of someone simply looking at object “A” it appears to **continuously** move right at 5 world-units/sec. The most important aspect of this discussion is that object “A” **is not continuously moving right**. So what is happening?

The details are complex but an easy way to explain it is that what appears on your screen obviously isn’t a real camera view on a virtual world, it has to be rendered. By rendered we mean that we draw the scene at a snapshot in time. In other words we handle the scene **discreetly** not **continuously**. This means that objects move in short intervals, get rendered and move again. This is discreet movement.

So why does object “A” appear to move continuously when we’re moving it in small steps? The answer is surprisingly simple. What happens is that we keep track of time! If we know the time we can use some simple physical equations to calculate where the object will be at any point in time with:

Speed x Time = Distance

So if we substitute the speed above of 5 world-units per second, at 10 seconds:

Speed(5) x Time(10) = Distance(50)

In 10 seconds we cover a distance of 50 world-units or half-way across the image above. So this helps us greatly. What we can do is note the time, perform the equation above and know where we should be and continue to do so until we reach our end-point.

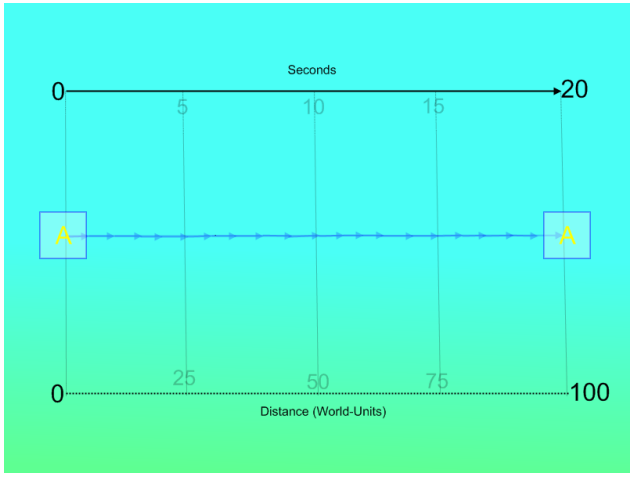
The thing is that we don’t want to calculate the distance from the start-point again and again; we want to calculate the distance to move in the time since the last frame. In other words we need to calculate the difference in distance not the absolute difference from when we started.

This is easy if we do two things: store the current position and calculate the time elapsed since the last frame. Then we can do the following:

Position = Position + (Speed x Elapsed-Time)

This allows us to add to our position the distance moved in the elapsed-time. If we keep track of the elapsed time we can calculate how far to move each frame.

Let's assume we only get roughly 1 frame-per-second (awfully slow) but it makes our diagram easier to understand. So we'd move the object every 1 second about 2 world-units but because it's only approximately 1 second it isn't always 2 world-units as indicated by the blue arrows in this diagram:



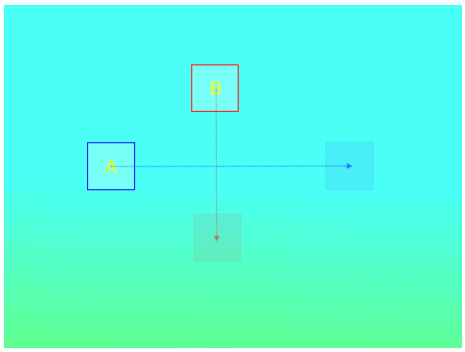
Note that the spacing between the blue arrows is not perfectly 1 second. It's realistic to expect this kind of non-deterministic behaviour although on a real game you'd expect much higher frames-per-second thus much smaller time-intervals on the order of a few milliseconds (1000th second).

So now we know that objects move by "jumping" from position to position based upon elapsed time between frames but what has that got to do with collision detection? Well, let's see where collision detection fits into this discreet movement scheme by setting up an example.

Let's setup a simple loop in our game that does the following:

1. Calculate the elapsed time since last frame
2. Move the object using the elapsed-time
3. Check for collision overlaps.

The scene is comprised of objects "A" and "B". Object "A" is moving right whereas object "B" is moving down. We calculate that half a second has elapsed and move the objects to their new positions based upon their velocities and then check for collision overlaps to see if they hit each other as shown here:



The image shows the new positions of objects "A" and "B" after they have moved. The arrows indicate the direction of movement. Note that the objects are not moving smoothly along that line, they "jump" from their starting position to the end position instantaneously as this is how far they each move during the calculated half-second that's elapsed.

So do the objects overlap after their move? If you answered yes consider again that the lines in the diagram above are for visual purposes, the objects move immediate from their start position to the end position.

The answer of course is no; the objects don't overlap after they move! So some may ask, "Why is this such a bad thing?" Well imagine if you will that object "A" is a projectile and object "B" is an alien. The projectile was fired by the player ahead of the path of the alien in the hope that it'd hit and by all rights it should have but it didn't. Why is this then?

It all comes back to the **continuous** versus **discreet** movement again. The player perceives continuous movement but it's actually discreet e.g. moving in small steps. The source of the problem is that each object in the scene takes it in turn to move as opposed to all objects moving at the same time, something which we can't do. In the above diagram, object "A" moved and checked for overlap and didn't find any then object "B" moved and check for overlap and didn't find any: no overlaps!

Some may say that the problem here was that the elapsed time for this highly contrived example was half a second! That's only 2 frames-per-second, hardly a realistic example. Whilst that's true we have to remember that the time is only half of the movement equation, the other half is velocity. What this means is that if we were to reduce the elapsed-time to something realistic, say 20 milliseconds and object "A" had a very high velocity (being a projectile) then the move would still be the same and it would jump "over" the path of object "B".

Of course, you could assume a decent frame-rate and make sure objects didn't move too fast and you **might** be okay but that's a bit of a balancing act and you don't always have the luxury of guaranteeing the frame-rate or object velocities.

So what's the answer? Well there are several alternatives, each with their own set of pros and cons but before we get to that, we need to consider the point raised at the start of this section which was that overlaps are bad. Let's not forget that we don't want to detect overlaps, we want to detect objects **colliding before they overlap**. To start doing this we need to change how we move objects.

Let's look again at how we handle object movement:

1. Calculate the elapsed time since last frame
2. Move the object using the elapsed-time
3. Check for collision overlaps.

This is clearly inadequate for detecting when objects collide and also has the added problem, as we have demonstrated above, that objects can pass each other without a collision. Another issue that arises is that you can get one object that moves "into" another e.g. an overlap. This is quite commonly called "**collision tunnelling**", "**overlap**" or "**interpenetration**" and again, it's something we want to avoid.

So how should we be detecting collisions? Well first we should modify **when** we expect to move our object like so:

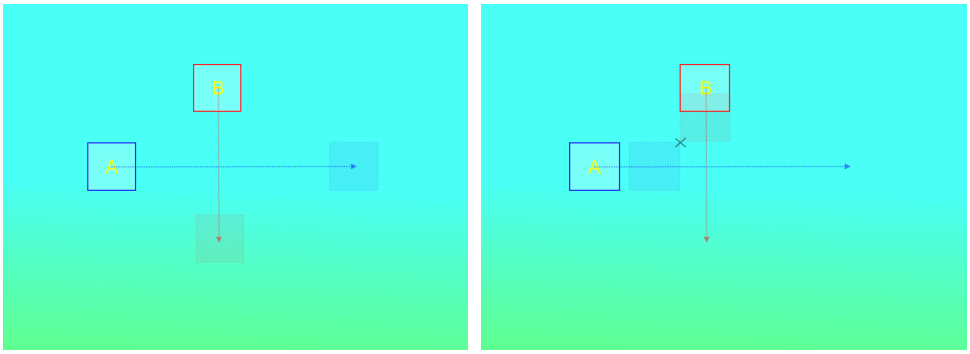
1. Calculate the elapsed time since last frame
2. Check for "swept" collision over the elapsed time
3. if no collision then move the object using the elapsed time
4. If collision then move to point of collision

As you can see we've added a new type of collision detection called "swept collision" but we've not yet discussed what that is. We'll get to that shortly but let's make sure we understand how it's being used first.

What we've said is that we calculate the elapsed time as before but instead of moving the object using the equation as we did previously, we instead check for collisions over that time period for the object that's moving. We then said that if we don't see a collision we move the object using the elapsed time which is exactly what we did before. Now though we've also said that if we do see a collision we move to the point of collision.

The point of collision is exactly at the point **just before** the collision happens. Normally this is the smallest fraction of distance possible "back" from the actual collision point. This is because, as we said before, we don't want to overlap, ever!

It's extremely important to understand the previous change in the objects movement and the motivation for collision detection to find a collision **before** an overlap occurs. Because of this importance, let's modify the previous movement of objects "A" and "B" to incorporate this magical swept collision. Here's the modification side-by-side with the "old" method:



In our old scheme both objects successfully moved to their new position without encountering collisions of any kind but in the new scheme, notice how the objects have not reached their intended end positions? If you're confused still on how this works then consider that in the new scheme we've **not moved** the objects yet. What we have done is **consider** if they'll interact when we move them to their end positions. Put another way we can say that in a sense we did a **continuous** check of the **discreet** movement.

Following the guide above we find that the objects do collide and we move them to the point of contact as shown by the black cross. Ignoring for a moment how we did this, we can see that we've at least achieved our goal of **avoiding overlaps**. Time to crack open the champagne? Well not quite yet but we can at least put it on ice!

So what is this magical swept collision detection? Unfortunately, swept collision detection is complex meaning it isn't easy to describe. The purpose of this article is to describe some of the processes involved in collision detection but to avoid where possible the mathematics. Therefore to give a full description of swept collisions is beyond this text but we'll still attempt to give you an appreciation of just what's involved anyway.

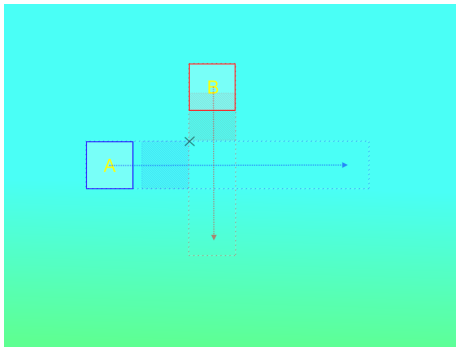
Swept Collisions

Let's recap on what we're about to describe. When we say swept collisions we're somehow talking about a collision detection method that should completely stop collision overlaps or "collision tunnelling". It isn't susceptible to the kinds of problems we've so far discussed. That's great so what's the catch?

Well the catch is that doing swept collisions isn't a computationally cheap operation but it's also not the most expensive one out there. The good news is that the previously discussed container-system can help us avoid doing swept collision checks altogether if they're not needed so we can be confident that we'll only do them when there's a reasonable possibility that objects will collide.

Notice now that we're using the term "**collide"** instead of "**overlap"**? That's completely intentional and by now you should fully understand the difference. No more overlaps!

So let's see our swept collision scene again but let's add something extra:



Notice now that there's two dotted rectangles extended away from object "A" and object "B". These are called the "swept volumes". Actually volumes is a bit misleading here as we're talking about 2D only but you can use swept collision detection in 3D and it's there that the terminology has come from. Technically we should say "swept area" but we're going to use the common convention.

So what is this swept volume? Well the swept volume should completely encapsulate the volume of space that the object will move through (or area in this case). It's the swept volumes of objects that are used to detect collisions and not the objects themselves. Swept volumes act as extrusions of objects through space. In our case we're using squares but we're not limited to squares, we could use triangles or other shapes but the solution is the same.

As you can see, using the swept volume instead of the object itself means that we "extend our reach" on collision detection. To describe the process further would require jumping into the mathematics, something we're trying to avoid. Besides, if you're interested you can find more information on the internet but be prepared for the mathematics!

One thing we've neglected to discuss here though is the container-system. We've already said it can help us avoid doing the actual collision sweeps but how? Well it's pretty simple. All we have to do is to ask the collision system about objects that overlap the collision volume instead of the objects bounding-box. If something doesn't sound quite correct about that then you're being very smart today. If you look at the previous image and imagine us asking what objects might collide with object "A" in its swept volume you should see that **nothing** will collide!

Well, we have to admit that to make things simpler to explain we ignored that but it is something that needs to be dealt with. So how do we resolve this seemingly fundamental flaw using swept collisions? Believe it or not, the swept collisions don't change at all, what we have to do is kind of cool in a geeky way.

Previously when we added our objects to the container-system we added references to our objects to the container bins that overlap the bounding-boxes of our objects and it's this that we have to change. Instead then we can be a little smarter about the information we put into the container system. Instead of the information there being the object at a snapshot in time, let's make the information be the object over a period of time!

So the essence of what we can do is that after we have calculated the objects bounding-box we can calculate the objects swept-volume based upon the bounding-box (not the object itself) and add the object to the container-system using that volume (actually area).

You may be wondering (if you're being really sharp) how much of the velocity is used e.g. how much time when creating a swept volume and that's the tricky part. There are a handful of ways to do this but by far the best method is to guarantee the elapsed-time used from our calculations.

Hold on, didn't we say we couldn't do that? Well we did but swept collisions, particularly if you're trying to be super-accurate as we now are need a much more sophisticated time-base mechanism than simply calculating the elapsed-time between frames.

Now we're really to get into an area that isn't directly the subject of collision detection and more in the area of simulation. Suffice to say though that you can actually guarantee regular and consistent updates if you setup a smart enough simulation.

So why use swept collisions at all if we can actually get guaranteed updates? Why not use the much simpler method we started with and make those time-updates really, really small? Well recall that the real problem with the earlier methods was that they detected when an object overlapped and not when the objects collide. It's the collision contact we wanted, not the overlap.

I'm confused about all this stuff! If you're thinking that then don't worry. Perhaps some of this swept collision detail is confusing and quite frankly it doesn't matter. If you understand that using swept collision detection can give us collision contacts and stops collision overlap then that's all you really need to know.

So moving back we can say that given that our simulation can give us regular and consistent updates we can use, in conjunction with a container-system, swept collisions to extrude our objects' bounding-boxes. We can then get points of collisions otherwise know as "collision contacts".

When we've got collision contacts we can then decide exactly what to do when we collide otherwise known as "collision response".

These are the two edges of the sword, collision detection and collision response. We'll shortly be talking about collision response but first let's expand upon what we've learned already and show a real example.

Collision Regions

As we know most games are not two square objects attacking each other but nice hand-drawn players/aliens etc. How do these complex looking objects collide? Is there even more complex collision detection that's needed?

The good news is no, you've already come far past the point where most people go. So how's it done?

Let's look at a realistic game object:

What we can see here is a cool "Ninja" character. Now you'd think that detecting collisions for this character would be complex but you'd be wrong. The odd looking green shape is actually the region that the collision detection system uses to determine collisions on this character. As you can see it's an extremely rough approximation of the character outline but it's enough!

It's easy to fall into the trap of trying to make everything super realistic and whilst that may have limited benefit with graphics, it's generally overkill for stuff like collision detection. The shape you see is more than enough for the way the character is going to interact with its environment. The actual shape and how close it conforms the actual characters outline is down to the developer. The developer must judge a balance between complexity/cost of collision region and realism.

In the end, most games are about "smoke and mirrors" meaning that they're about faking real-world situations just enough to convince the player so that the player can enjoy the game. There's a lot that can go wrong. If you configure the collision region poorly then you'll find the player getting frustrated when projectiles pass over the object but don't collide with it. In the case of this character it's not an issue at all.

Another thing to consider is that you're not limited to having a single collision region. Some systems allow multiple collision regions for the detection of collisions. Some do this in different ways and it's quite common for a character to be a composite of multiple objects each with their own single collision region. You could imagine a character where the legs, arms, torso and head are different objects linked together each with their own collision region.

This has many benefits apart from the obvious animation freedom. It has the benefit of telling the developer which part of the character was hit e.g. leg left, leg right, torso etc.

There are many types of collision regions that can be used. Most common are circles and squares. These are used because they are generally easy to understand but they also have very simple geometric properties that allow very fast collision detection when they're used as the basis for broad or narrow-phase collision detection.

Another and more complex method is to use an arbitrary sided polygon as shown in the image above. These allow the developer to more closely approximate the outline of the game characters should they wish to.

There are actual two types of polygon collision region: convex and concave. Convex is by far the most commonly used type because convex polygons again have geometric properties that allow much faster collision detection whereas when a polygon is concave, certain assumptions cannot be made and the process is much slower. If you're interested in the difference between convex and concave polygons you can easily find an overwhelming amount of information on the internet about them.

Collision Response

So we've come a very, very long way since we started this discussion. We started by talking about detecting objects overlapping then we introduced the concept of a bounding-box and started using this in the container-system to trivially reject objects that have no chance of overlapping. We then highlighted issues with basic overlap tests and introduced the idea of collision contacts using swept collisions. We then spoke about how realistic game characters can use a variety of collision regions.

That's all useful information but it's only half of the story and less than 1% of the fun! Let's discuss now what we do when we detect a collision otherwise know as "collision response".

We know that a collision response is what we do when we detect a collision but we can't give you a single answer on what the collision response should be so how's this going to work out? The thing is that there are a number of established responses that you could use but what's used is determined by the developer.

For instance a collision response might be to play a "bell ring" sound or start an explosion effect. These are indirect collision responses not direct ones. We're going to limit our discussion here to direct ones. More specifically we're going to limit our discussion to the physical responses of the colliding objects themselves.

So what do we mean by physical responses when it's just a virtual simulation? Well we should by now be comfortable with the concept that our game objects have both a position and a velocity. Those are physical properties. A common way of saying this is that position and velocity are part of the physics system. Now we don't want to scare you by using the word "physics" and you can rest assured that we won't go into any complex math here so it's okay and you can sit back down now.

So what kind of physical responses are we talking about that affect at least position/velocity? What about these for starters:

- Bouncing
- Clamping
- Stopping

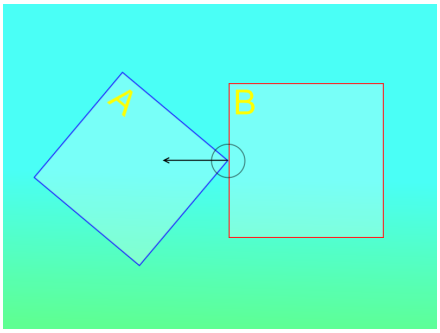
Let's explain those before we continue. By bouncing we mean that when an object hits another object it'll bounce straight back from it just like a rubber-ball bouncing on a wall. When we say clamping we mean that if an object hits another object at an angle, it'll not move into the object but slide along it. When we say stopping we mean stop at the collision point and no longer move afterwards.

So they all sound pretty simple and indeed they are but what do we need to perform them? Well to perform these collision operations we need to have a little collision information. By collision information we can mean all sorts of things dependant upon how complex the collision system is but for this discussion we're only interested in two things, these being:

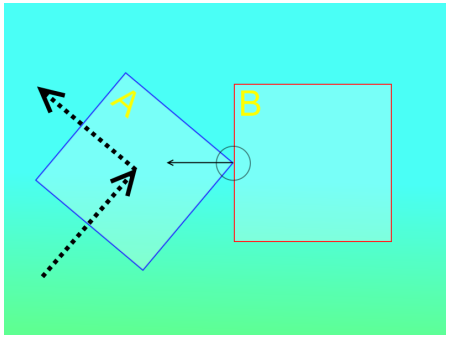
- Collision Contact
- Collision Normal

Let's explain those before we continue. A collision contact is synonymous with the point of collision or in other words the point at which two objects collide. The collision normal is a vector (direction) moving away from the point of collision.

To explain collision contact and collision normal further let's bring back our objects "A" and "B". Imagine that object "B" is stationary and cannot be moved no matter what; it's immovable. Now imagine that object "A" is moving right and hits object "B". When it hits it hits at the collision point (the collision contact) and that collision has a collision normal as shown here:

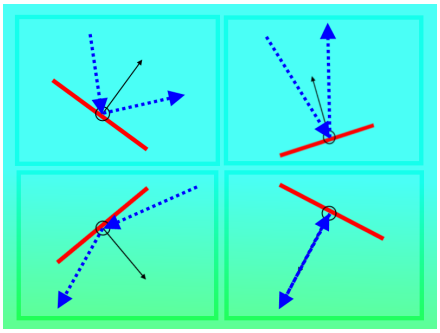


The collisions contact is a point at the centre of the black circle and the collision normal is the direction indicated by the black arrow. Note that the collision normal is pointing in the opposite direction that object "A" was moving. That's coincidence, the arrow points away from the edge of object "B" that object "A" collided with. Indeed it's always **perpendicular** to the edge. So how are these used for a collision response? Let's go through some responses starting with bounce:



This diagram shows the direction of object "A" before it collides with object "B" as shown by the black arrow pointing up/right. After collision the "bounce" response was applied and the new direction of object "A" is shown by the second black arrow pointing up/left.

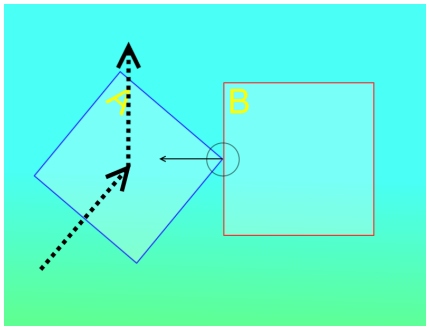
So is there a way of describing what happened other than saying the object bounced? Yes, the objects velocity was reflected around the collision normal. So what does that mean? Here's an image showing different bounces and different collision normals:



The common feature of all these bounce responses is that the incoming velocity is reflected around the collision normal. Imagine that you put a mirror along the collision normal (black line). If you looked at the pre-collision velocity you'd see it reflected as the post-collision velocity. The incoming angle of velocity is equal to the outgoing angle of velocity.

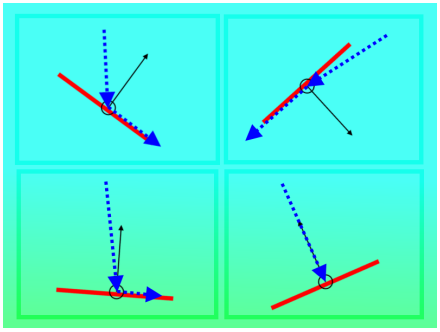
You can imagine this being a rough approximation of a ball bouncing off a wall. Note though that it's a perfect bounce meaning that the "ball" doesn't slow down at all like it would in the real world.

So now we've got the same setup as before but we'll use the clamp collision response:



This diagram shows the direction of object "A" before it collides with object "B" as shown by the black arrow pointing up/right. After collision the "clamp" response was applied and the new direction of object "A" is shown by the second black arrow pointing up.

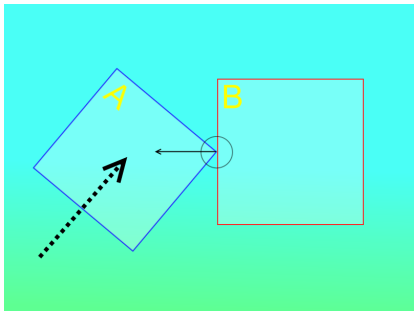
So is there a way of describing what happened other than saying the object clamped? Yes, the objects velocity was cancelled along the collision normal. So what does that mean? Here's an image showing different clamps and different collision normals:



Notice that no matter what the velocity or collision normal the post-collision velocity always ends up being 90° (perpendicular) to the collision normal or along the edge of the object that we collided with. Notice also that the example in the bottom right, where the pre-collision velocity is moving directly along the collision normal towards the surface, the clamp response zeros the velocity (the object stops). Additional to this the **speed** is always reduced upon collision because we're taking something away from the velocity.

Imagine if you will that you move slowly towards a wall at an angle. When you hit the wall (hopefully slowly) you can continue to move not into the wall but along it. Because of this sliding action the clamp collision response is sometimes known as the slide collision response. Play any first person shooter and you'll see this behaviour if the player hits a wall.

So finally let's look at the stop collision response:



After the previous two responses this one should be easy to understand. Notice that the post-collision velocity is zero (the object has stopped). The stop collision response doesn't really need the collision normal as it just sets the velocity to zero. It's pretty easy to understand and implement. Because the stop collision response causes the object to seemingly stick to the other object (it stops in contact with it) it's sometimes known as the sticky collision response.

It takes two to party…

So far we've limited our collisions to one object hitting another immovable object and the moving object having a collision response. Now whilst this situation is common it's by far not the only configuration for collision responses. So what kinds of configuration options are available for collision response?

Each and every collision system will have its own blend of options and we obviously can't cover all those so we'll focus on the ones that are tailored towards keeping the collision systems work to a minimum.

So where do we start? Well, you could say that the collision options start before a collision is even checked. Therefore it's probably best to start there by discussing the various options of collision configuration before collisions begin.

Collision Groups

Let's start by addressing a fundamental issue that we've so far not covered. In the simple examples we've provided we were checking if objects collided with all objects in their vicinity.

The trouble is that in a real game we're likely to have lots of objects in close proximity that we **don't want** to check for collision. An example might be in a platform game where the player wants to collide with the ground but doesn't want to collide with other parts of the level that are just there for decoration.

So the question then becomes one of how do we exclude certain objects irrelevant of their proximity? The most common way of doing this is to assign each object in the scene to a collision group. A collision group is just that, a group of objects.

Using the previous platformer example we could put all the "ground" objects in a collision group called "ground group" and all the decoration objects in a collision group called "decoration group". We can then configure our player object to collide only with the "ground group".

The nice thing about this is that anything we add to the "decoration group", we can be sure won't collide with the player. As well as the obvious benefits of having objects in groups, the assignment of an object to a group or groups is known by that object. In other words given an object we can ask it what collision group or groups it's in.

This has the advantage that features such as the container-system can discount, early-on, the objects you don't want which will mean that the collision system performs more efficiently as it doesn't pass these unneeded objects further up the collision chain.

Let's explain that last one a little further. What we mean is that we can ask more specific questions to the container-system such as show me things that might overlap objects in the "ground group". It can do this because the container can look at the objects it finds before it returns them as candidates for further collision checking. That's a much better way of doing it as irrelevant objects are filtered out right at the start thus improving performance.

Disjoint Responses

Although it's easy to visualise two objects hitting each other and having the same collision response like stopping it's harder to understand when two objects might have different collision responses.

An example of this would be that you want one object to bounce but the other object to explode and disappear. If you've ever played an "arkanoid" style game where you bounce a ball off a paddle into bricks that are destroyed when you hit them with the ball then you can understand this. The ball bounces off the brick but the brick itself explodes.

I Send, You Receive…

If you look at collision situations you find that if an object is configured to check for collisions it'll do so without providing the object it's colliding with the option to cancel the collision but why would we want to do that?

If we had an object that didn't want things to collide with it then we could put it in a collision group and get all the objects to state that they don't collide with that collision group. That sounds like an awful lot of work just to say "don't let anything collide with you"! Besides collision groups work best for saying what to collide with rather than what not to collide with.

Another situation is where we don't want an object to check for collisions when it moves and that's a pretty common requirement in itself.

Both of these seemingly unrelated situations have a common solution which is to say that if we want to control whether an object checks for collisions when it moves we're stating whether it "**sends**" collisions or not. This is the source of all collision processing; the act of checking for collisions.

When we want to control whether an object will allow a collision to take place if another object collides with it, we're saying whether it accepts or "**receives**" collisions.

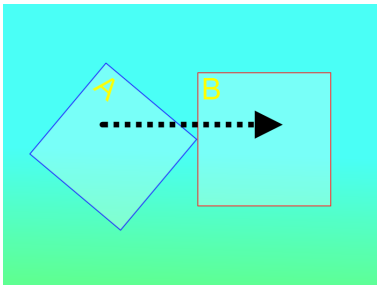
This is confusing, what does all this mean?

Let's slow down a little. This can be a little tough to understand at first but it's very important.

What we're saying quite simply is that an object that checks for collisions is "**sending**" them whereas an object that is collided with is "**receiving**" them.

If an object has "send" collisions enabled it will look for collisions with other objects. This is exactly what we've been talking about so far and should be really easy to understand. If the object we're colliding with has "receive" collisions enabled then a collision takes place and you'd expect collision responses from both objects as normal. So there's nothing new there when both "send" and "receive" collisions are enabled.

Before we create more complex setups let's have an image to work with:



So here we've got object "A" and object "B". Object "A" has "send" collisions **on** so it's looking for collisions and it finds one when it collides with object "B". Before we continue though, what would happen if object "A" has "send" collisions **off**? The answer is simple; it'd move but wouldn't be checking for collisions so it'd pass right through object "B". Collision sending is therefore whether the object checks for collisions.

So back to our example where object "A" is sending collisions and finds one when it hits object "B". What happens next? Well because object "A" collided with object "B" we need to also consider object "B", specifically is object "B" receiving collisions?

If object "B" is receiving collisions then the collision actually happens and both objects proceed to their respective collision responses.

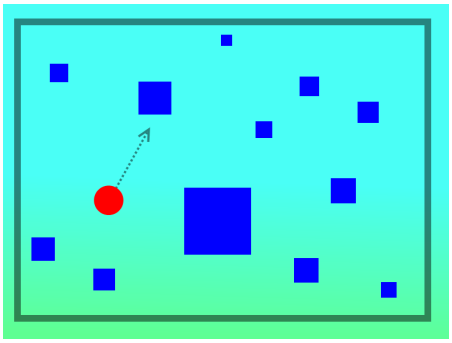
What happens though if object "B" is not receiving collisions? The answer is "nothing"! If object "B" is not receiving collisions it cancels the collision and object "A" doesn't even consider it a collision. It's as if object "B" didn't exist! By configuring object "B" to not receive collisions other objects that encounter it ignore collisions with it.

There's a simple formula for deciding how the interaction of send/receive collisions work:

Send State	Receive State	Collision
On	On	Yes
On	Off	No
Off	On	No
Off	Off	No

As you can see from the above table, the only method of generating a collision is when an object that's sending collisions collides with an object that's receiving collisions.

Let create an example to make the advantage of using send/receive collisions more clear. Imagine I've got a red ball that's bouncing around inside an arena. I've also got a bunch of blue blocks placed randomly inside the arena. I want the red ball to bounce off the arena walls as well as the blue blocks. We've configured the blue blocks and the arena walls as immovable so that even if they do get a collision response they won't move. Here's the setup:



How should I configure send/receive collisions? One way is to just set everything to send/receive collisions but why is that okay? It's not okay but it'll work. Let's work through it assuming everything has send/receive collisions on. We'll pretend we're the simulation itself working through each and every object so that you can get a better appreciation of what's going on:

I start with the red ball and find it has send collisions on so I check for collisions given its current velocity and the time it needs to move. I find it collides with a blue block so I check the blue block and find it has receive collisions on so I process the red ball with the "bounce" response and it bounces away from the blue block and then I attempt to process the blue blocks collision response but it's stopped because the block is marked as immovable. Okay, the blue block was a little waste of my time but it worked out okay.

Next I go to the first blue block and check if it has send collisions on and it has so I check for collisions given its current velocity which is zero and the time it needs to move. I find it doesn't collide at all. I notice that it's marked as immovable but the user has send collisions on for this blue block so I can't really argue that an immovable object shouldn't check for collisions so I complain that I did a bunch of collision checks that seemed to be a waste and move on.

I process all the rest of the blue blocks in exactly the same ways as just described with the same result of wasted collision sends. That adds up to an awful waste of time but I keep the metrics showing this waste and hope the developer will take notice.

Finally I process the area walls and I get the same result as all the blue blocks. I sigh again.

So I have processed all the objects and it's now time to process the red ball again … when will the torment end?

So what did we learn from this?

It seems the simulation is complaining about wasted time but what is it talking about? The ball could collide with any of the non-moving blocks so we need all our objects to send/receive collisions, don't we?

Well actually we don't. If we look back to the simulation text above we see that the ball successfully bounced off a blue block and it only took two things. The first was that the ball was **sending** collisions and the second that the blue block was **receiving** collisions. As we know, when these two conditions are met we get a collision event and in this case the ball bounced away successfully.

Next the simulation spent most of its time going through each and every block checking for collisions. At first you might not think that a problem but if we stop and ask ourselves why the blue blocks are sending collisions we might decide we're doing it so the red ball will bounce. The thing is we've already seen the requirements to get the red ball to bounce above and it didn't include having the blue blocks sending collisions so again, why are we doing this and wasting so much time?

The answer isn't that we're lazy rather we didn't fully appreciate what was required to achieve our goal. We could be forgiven for that but we also didn't appreciate that there were some pretty computationally expensive operations going on just because we didn't turn off a few options.

So now that we feel scolded, how should we be handling it in the future? It sounds like developers should write down how groups of objects should interact and configure each accordingly. That isn't a bad idea and would certainly help make the game more efficient but it does help to have some guidelines here.

The most simple and effective guideline there is for these kinds of collisions is twofold:

1. Unnecessary "send" collisions waste time
2. Unnecessary "receive" collisions allow unneeded collision responses.

From this you should see that by far the worst thing to do is send collisions when they're not needed. By not sending collisions you completely cut out any collision processing. By not receiving collisions you won't get your objects getting collision responses when they're not needed.

So finally let's summarise what send and receive collisions are.

Send collisions are the act of allowing an object to seek-out collisions and this takes time to do. Receive collisions control whether a sent collision is accepted on an object when it's collided with.

Put another way, a scene where all objects send collisions but none receive them will have a scene where objects are constantly checking for collisions but none will ever occur because no objects receive them. A scene where all objects receive collisions but none send them will have a scene where no objects are checking for collisions therefore none will ever occur.

The rules generally are then that you want as few objects as possible sending collisions and only set objects to receive collisions if you want something to collide with them. What objects a sending object collides with can be controlled with collision groups but to collide with other objects they need to "receive" the collision.

In the case of a "platformer" game where the only thing moving is the player and all other objects are the level which don't move, it makes sense that the player is sending collisions but not receiving them and all other objects in the level receive only. This means that the player is constantly checking to see if it collides with any of the level objects and can collide with them because they are receiving collisions. It also means that because the level objects are not sending collisions, the simulation doesn't waste time checking to see if they collide with something. As well as the player you'd probably have other non-player-characters sending collisions as well.

The "platformer" example above demonstrates a situation where you've got a few object(s) that needs to collide with many objects. It makes sense to only check the single object against the many rather than the other way around as that would take much longer to perform.

Torque Game Builder

So now let's associate what we've learned with what TGB does. You'll be pleased to know that TGB does exactly what's been described previously. It uses a spatial-partitioning container-system as well as swept circular/convex-polygon region collisions. It also provides send/receive collision processing.

One thing you should now appreciate though is that when you call the following script methods on a "t2dSceneObject", TGB needs to change the objects bounding-box and therefore needs to update the container-system:

- SetPosition
- SetRotation
- SetSize

TGB doesn't directly expose access to the scene-container, it's hidden from you, but there's a container-system in each and every "*t2dSceneGraph*" you create.

As a side note, if you own the C++ source-code you'll find the container-system in the files "*t2dSceneContainer.cc*" and "*t2dSceneContainer.h*".

TGB does expose some of the parameters of the scene-container through the "*t2dSceneGraph*" itself. These parameters are pretty advanced and relate directly to the previous sections entitled "The Infinite World" and "Container-System Penalties", they are the container bin size and bin quantity.

In TGB the container bin size is in world-units and the top-left of the container bin (0, 0) is at world-coordinate (0, 0). The default container bin size is 20 world-units and the container bin quantity is 256 giving a total of 256*256 = 65536 container bins by default.

More interestingly because the container bin size is 20 and there's 256 in each axis it means that the container system wraps the bins at the world coordinate 20*256 = 5120, 10240, 15360 etc.

Also it wraps in the negative direction so you get a wrap at -5120, -10240, -15360 etc. What this means is that if your scene is no larger than 5120x5120 then your collision searches won't encounter the wrapping issue detailed in the previous section "The Infinite World".

So where are these parameters set? Each "*t2dSceneGraph*" has its own container-system and therefore exposes these parameters itself. Two ways of setting them are either:

- Creating a "*t2dSceneGraphDatablock*" and assigning it to the "*t2dSceneGraph*" upon creation. This datablock has the following fields: *"containerBinSize", "containerBinCount", "useLayerSorting" and "lastInFrontSorting". The first two parameters are the container-system parameters.*
- Using the "*t2dSceneGraph.initialise()*" method which has the same parameters as described above.

TGB and big objects…

One thing that we've neglected to mention is what happens when really big objects are placed into the scene and therefore need representing in the container-system.

Are we saying that an object that overlaps thousands of container bins needs to update those bins every time its bounding-box changes? What fool thought that one up? The good news is that this doesn't happen. As efficient as the container-system is, there's a reasonable limit to how many bins you should be updating for any individual object.

To manage this, TGB has a structure called the "overflow" bin. It's a single container-bin where objects that are over a specified threshold are placed instead of the bins within the container-system. TGB internally manages this differentiation. What this means though is that the container-system will always return the objects in the overflow bin no matter what the bounding-box.

That sounds quite bad but the TGB container-system does the next-phase and compares the bounding-boxes after a potential list has been returned. This means that the container-system in TGB doesn't just return objects that **might** overlap; it returns objects that **might** overlap but also have their bounding-boxes overlapping as well. You could say it returns objects with a high confidence that they overlap.

The reason we mention this is that you should consider that the larger the object, the more container bins it'll occupy up to a limit when it'll only be contained in the overflow bin. If the container bin sizes / quantity are configured correctly though you won't notice any reasonable difference in performance.

TGB sending and receiving collisions...

Previously we mentioned that some collision systems allow objects to separately send and receive collisions. Now that you understand how that works, you'll be glad to know that TGB provides this functionality.

There are different ways of setting send/receive collisions on objects dependent upon whether you're doing it in the TGB editor or not. We'll talk about how to do it in scripts here.

Given a "*t2dSceneObject*" named "%object" you can activate send/receive collisions on it like so:

```
%object.setCollisionActive( true, true );
```

The first parameter is whether to activate send collisions and the second is whether to activate receive collisions. That's pretty easy to do and the send/receive collisions work exactly as described previously with all the performance implications that go with it.

You can also control send/receive collisions individually like so:

```
%object.setCollisionActiveSend( true );
%object.setCollisionActiveReceive( true );
```

As a note, TGB collision send/receive both default to "false" meaning collisions are turned off by default.

TGB sending and receiving collision responses...

Now that we understand what sending/receiving collisions is all about, it shouldn't be that difficult to introduce another useful concept that's particular to TGB: the ability to send and receive collision responses.

Sending and receiving collision responses all starts when you get a valid collision event. By valid collision event we mean that an object sending collisions collides with an object receiving them.

So when a valid collision event happens, rather than always producing a collision response on both the objects (one sending a collision and one receiving the collision), TGB allows you to choose, per object, what happens.

TGB though doesn't call this "collision response"; it calls this "collision physics". The reason for this is that TGB provides a number of stock collision responses, most of which affect the physics of the objects only. For this reason TGB has the notion of a collision causing physics changes in the object. We say that objects "react" to collisions.

So moving back to just after a valid collision, TGB doesn't just process the physics updates for both objects immediately. What it does is see if the objects allow those changes first. It can do this because each object has an option that allows it to control if a physics change is allowed dependent upon the "direction" of collision.

TGB provides an option to change physics if:

- The collision is sent by the object
- The collision is received by the object

This means that each object has an option to control whether it reacts to a collision dependant upon whether it was the one sending the collision or whether it is receiving a collision sent by another object.

So given a "*t2dSceneObject*" named "%object" you can activate send/receive physics on it like so:

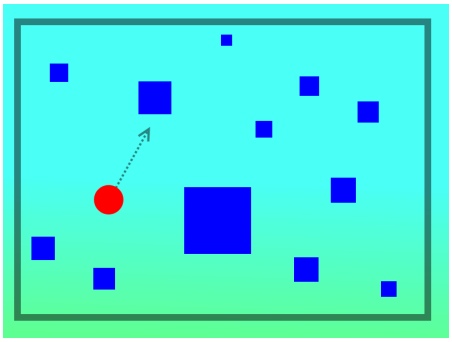
```
%object.setCollisionPhysics( true, true );
```

The first parameter is whether to activate send physics and the second is whether to activate receive physics. You can also control send/receive physics individually like so:

```
%object.setCollisionPhysicsSend( true );
%object.setCollisionPhysicsReceive( true );
```

It's worth noting that the collision physics options don't work the same as the collision send/receive options. You don't have to have collision send physics active on the object sending the collision and collision receive physics active on the object receiving collision for physics to be used at all. They each work independently and control whether the respective object reacts to collision physics.

So what kind of situations would being able to turn send/receive physics on/off help? Well, let's go back to an example we provided earlier:



As before we want the red ball to bounce off the blue blocks and the blue blocks to not move because of the collision. But this time we will occasionally apply a little thrust to the blue blocks to move them around just to make things a little more interesting.

To get the red ball to bounce off the blue blocks we previously set the red ball to send collisions only and the blue blocks were set to receive collisions only and finally we also set the blue blocks as immovable but unfortunately this time we can't set the blue blocks as immovable because we want to move them occasionally with a random thrust. So how do we achieve this? In TGB it's surprisingly easy.

As expected we set the red ball to send collisions and set its collision response to be "bounce". We then set all the blue blocks so that they receive collisions only. This means that the simulation won't waste time sending collisions from blue blocks but they will receive the collision when the red ball hits them.

But we're not going to set the blue blocks as immovable but we don't want the blue block to move when a collision occurs. So we think about it and realise that when the red ball hits the blue block, the red ball **reacts** because it sends the collision and the blue block **reacts** because it receives the collision. Hmm, that sounds familiar.

Indeed it is, if we activate send collision physics on the red ball but deactivate receive collision physics on the blue blocks then, after a collision, the red ball will react because it's sending the collision and its send collision physics is active but the blue ball won't react because it's receiving the collision and its receive collision physics is inactive.

The script might look something like this:

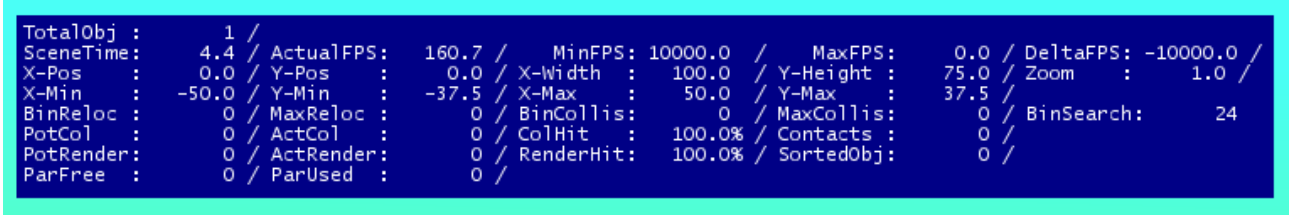
```
%ball.setCollisionActiveSend( true );
%ball.setCollisionPhysicsSend( true );

%blueBlock.setCollisionActiveReceive( true );
%blueBlock.setCollisionPhysicsReceive( false );
```

As a note, TGB collision physics default to "true" for both send and receive so you only really need to set them when you're setting them to off.

Torque Game Builder – Debug Banner

So now that you understand what TGB is doing to determine if objects collide, it'd be nice to see how you can find out more detail about what's going on behind the scenes. TGB gathers a huge array of metrics designed to help developers tune their applications in such a way as to get the best out of available resources. Some of this information can be written to disk when the game exits but by far the best way is to see the metrics as the game is running, this is what the "Debug Banner" provides. Details of how to display this can be found in the TGB documentation.

Here's what the TGB Debug Banner looks like:



Being as we're primarily discussing collision detection here we'll focus on only a few of the metrics shown above. Most of the metrics are abbreviated to reduce space taken on the screen and look a little cryptic at first. Let's start going through them in related groups:

[BinReloc] [MaxReloc]
Previously we discussed that when the bounding-box of an object changes, its representation in the container-system has to change, specifically the reference to the object in the container bins. Making this change to a bin is known as "relocation". For instance, say an object that occupies 4 container bins has "setPosition()" called on it. This will cause 4 relocations in the container-system. If this is the only change during that frame then the [**BinReloc*ation*]** metric will show 4.

This metric is a good way of indicating how much work the container-system is doing simply processing bin relocations. [**MaxReloc*ation*]** simply gives the maximum **[BinReloc*ation*]** since start-up. If you see very high relocations then you must be either repositioning lots of objects (perhaps many times per-frame) and/or the objects occupy lots of container bins because they're large in size and/or the container bins are very small.

[BinCollis] [MaxCollis]
When the collision system queries the container-system, a count is kept on how many container bins were searched to return answers to those queries. **[BinCollis*ion*]** shows the total count of container bins searched for collision detection purposes during the last frame. **[MaxCollis*ion*]** simply gives the maximum **[BinCollis*ion*]** since start-up.

[BinSearch]
A common feature for a game engine to provide is the ability to programmatically select objects in a scene. This normally means that the developer chooses a line, rectangle, circle or some other primitive and gets back a list of objects that overlap the region. This feature is commonly called "picking". The great thing about picking is that it's just another form of collision detection. That means that when we ask the game engine about which objects are in a certain region we can now be sure that the container-system is involved and can therefore do it efficiently.

TGB provides a wealth of picking functions. Because these picking functions use the container-system to search container bins, **[BinSearch**] shows the total count of container bins searched for picking purposes in much the same way that **[BinCollis*ion*]** shows the same but for the specific purpose of collision detection. If your game does a lot of picking, having container bin searches split by collision / picking could be handy. The total number of collision bins searched per frame is **[BinCollis*ion*]** + **[BinSearch]**.

[PotCol] [ActCol] [ColHit] [Contacts]
Because the container-system only determines what **might** collide, it's only part of the story of collision detection. Indeed there's a big difference between a potential collision and an actual collision. A potential collision is what's returned by the container-system e.g. a **might** collide whereas an actual collision is a potential collision that was checked and determined to actually collide. **[Pot*ential*Col*lision*]** and **[Act*ual*Col*lision*]** represent these values. **[Col*ision*Hit]** is the percentage of potential to actual collisions. If you get 20 potential collisions and 2 actual collisions then your collision-hit percentage is 10%. Think of this as part of the collision efficiency. If this is low then you're wasting lots of time doing nothing useful. Sometimes this can't be helped but most of the time it can!

If you're consistently getting a poor **[ColHit]** it's probably because the container bins size is too big and too many objects are getting placed into the same bin even though they don't collide. Before you jump and reduce the size of the container bins though you need to careful consider whether reducing their size will impact the performance of you game compared to the loss of performance by having a poorer collision hit percentage. A quick method would be to reduce the size and see if the **[ColHit]** goes up without too much of a FPS reduction.

Always consider though that you may just be at the mercy of the position and velocity of objects. Even a well configured container-system can be fooled under certain conditions albeit those are nearly always contrived situations. Most of the time you won't get this situation and it's best to use common sense. Don't try to squeeze a few extra percent out of **[ColHit]** but seriously consider what you're doing if you see something low like %10 consistently.

These metrics are a great way to determine how collision efficient your game is. It should be the first place you go when you suspect that too much collision processing is going on or things are just not as fast as you'd expect. If all looks well here then you should consider looking at the previously discussed **[BinCollis] [MaxCollis]** and possibly, although unlikely, **[BinSearch]**.

A common problem is lazy configuration. It's easy to add objects to the scene or even copying existing ones without considering what the collision configuration of those objects is. The objects may have collision active but there's no chance they'll collide with anything. This would show up as a large **[PotCol]** compared to the **[ActCol]**. Put another way, the **[ColHit]**% will be low or reduced.

Finally **[Contacts]** simply counts how many collision contacts were encountered in total during the frame. Although we didn't discuss the different types of collision contacts, only simple contacts, we should be aware that some collisions can produce two simultaneous contacts so a contact count of 2 doesn't necessarily mean two collisions. The Contact count is another measure of how much work the collision system is doing. You can associate **[Contacts]** with **[ActCol]** so that you can see that certain quantities of contacts were produced by so many actual collisions.

[PotRender] [ActRender] [RenderHit]
These metrics relate to rendering and it wouldn't be appropriate to talk about them here apart from the fact that rendering actually uses the container-system so we'll discuss it here.

In TGB the responsibility for rendering is a GUI control called the "*t2dSceneWindow*". This GUI control acts effectively as a camera onto the world (represented by a "*t2dSceneGraph*"). To render the world it needs to determine which objects in the scene are in the rectangular view.

Sound like a familiar problem? Absolutely, this control uses the built-in picking functionality which searches the container-system and returns a list of objects that overlap the specified rectangle. In this case the rectangle is the camera "view". The control then renders all these objects and you see your wonderful scene.

The objects returned potentially overlap the "view". They're then checked to see if they actually overlap the "view". These "potentials" and "actuals" are represented by the metrics **[Pot*ential*Render]** and **[Act*ual*Render]**. **[RenderHit]** is the percentage of potential to actual renders. If you get 60 potential renders and 15 actual renders then your render-hit percentage is 25%. Think of this as part of the render efficiency.

It should be noted that because the rendering internally uses picking to determine objects to be rendered, the **[BinSearch]** includes that rendering overhead.

Finally …

We say finally but this is really just the beginning for a hopefully well informed developer to go away and really squeeze the most efficient collision setups out of their system. Hopefully this'll be using TGB but we also hope that we've provided enough basic information for you to translate to your own home-baked projects and beyond.

So have fun with collision detection and physics and don't forget to show us what you can do.

GG