Wildbunny blog
Developer knowledge centre

Implementing a wrap-around world

Posted on April 13, 2012 by Paul Firth

Hello and welcome back to my blog!

This time I'm going to be talking about something related to the game I'm currently working on, which is a little asteroids style MMO: a wrap-around world. This was necessary because the asteroids world I'd made was infinite in every direction, which meant finding other players could quickly get difficult.

The wrap around

What I mean by wrap-around, is that when you fly off one edge of the world, you get warped to the other side, so you can travel infinitely but the world has finite area.

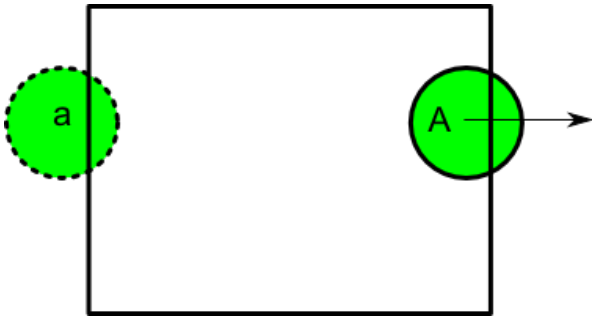


Figure 1

In *Figure 1*, *A* travels off the right hand edge of the world and is then wrapped around to the left hand edge.

'Trivial!' I hear you cry, just wrap the position of the object as it crosses the boundary, right? Wrong. This will only work if your objects are 0 size, like points. Furthermore, what happens if the player has a view of the world which also intersects the world boundary?

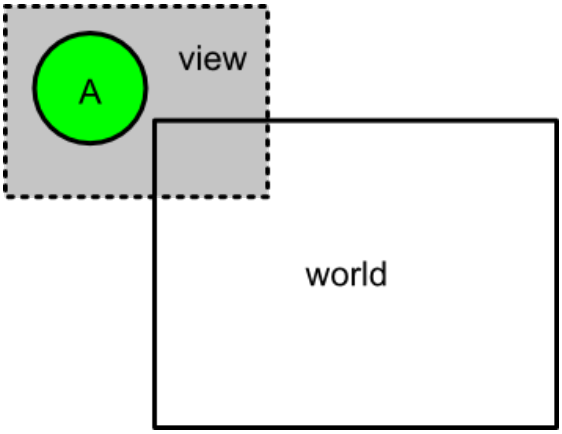
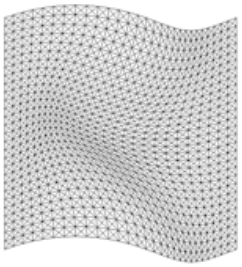


Figure 2

Figure 2 shows just such a predicament. Things are suddenly a lot more complex because the things in the player's view, such as object *A* are not actually there at all, since they have been wrapped around into the world bounds.

Toroidal geometry

This kind of wrapping is not actually spherical as you might imagine, but rather toroidal as that supports an entirely rectangular area with no distortions like you get when trying to map a rectangle onto the surface of a sphere.



```
    ∇ MAT←MAT projection TOR
[1]   αx←1↑⍴MAT⌽αx←○(÷αx)×¯1+⍳αx
[2]   αy←1↓⍴MAT⌽αy←○(÷αy)×¯1+⍳αy
[3]   ßx←(2○αx)×(1○αx)
[4]   ßy←(2○αy)×(1○αx)*2
[5]   ßz←(1○αy)×(1○αx)*2
[6]   MAT←αx αy⍴↑ßx⊖ßz⌽ßy⊖[2]TOR
    ∇
```

Image taken from harmonicresolution.com

The process

When dealing with this in a game there are several problems to solve – rendering of wrapped objects and collision detection. Both of these problems require that we can find out what objects are supposed to be in the viewing/collision area when it lies outside the bounds of the world.

In order to satisfy this requirement we must first start with the basics – given an AABB (bounding rectangle) which intersects the world bounds, what are the other AABBs within the world where there might be objects we ought to consider?

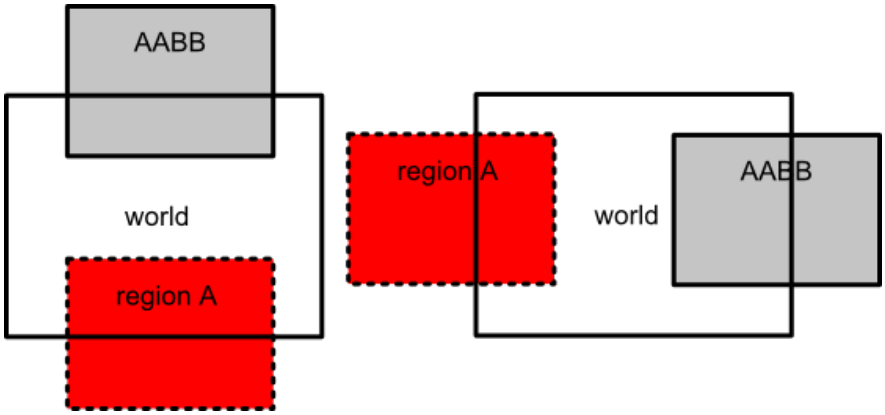


Figure 3

Figure 3 shows the two easy cases: the query AABB is simply wrapped off the offending axis. In both these cases there are two AABBs which need checking in total – the original AABB and the new wrapped region AABB (shown in red). Actually, there are two more cases for -x and +y, but they are very similar.

But what happens when an AABB straddles both x and y edges?

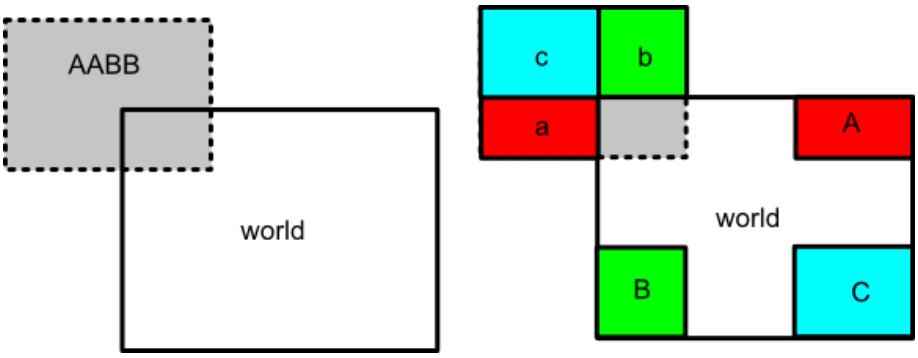


Figure 4

Figure 4 shows on the left a candidate AABB straddling both x and y edgse. On the right there are now **three** new AABB regions, *A, B* and *C* generated from the original AABB cut into sub-regions *a, b* and *c*.

A and *B* just represent the two cases we've seen earlier but *C* is different, in that the position in both axis has

been wrapped.

There are now four regions in total we must check for objects.

> *Note that for simplification* Figure 4 *shows* A, B *and* C *as being smaller than the original AABB. In practice this is not necessary – just keep them the same size.*

Detecting regions

Here is some code which will mark which axis need checking as a candidate AABB begins to intersect the world bounds:

```
/**
Flags for overlap
*/
public class OverlapFlags
{
        static public const kMinusX:uint = 1;
        static public const kMinusY:uint = 2;
        static public const kPlusX:uint = 4;
        static public const kPlusY:uint = 8;
}

/**
The overlap check itself
*/
static public function AabbWithinFlags( candidate:AABB, container:AABB ):uint
{
        var flags:uint = 0;
        var delta:Vector2 = candidate.m_Centre.Sub( container.m_Centre );
```

The wrapping

Once we have our regions to check, whenever we find objects in those regions we need to temporarily move them into a common space in order to do comparisons between them (like computing distances for example), or for rendering them.

> *In this demo you can move the black rectangle around with the mouse, the generated regions are shown as it intersects the world bounds.*

Typically, an object is only actually fully wrapped when it is completely outside the world bounds.

Download the source

[You can download the source for free here, its provided with no warranty]

That's it!

Until next time, have fun!

Cheers, Paul.

About Paul Firth

A games industry veteran of ten years, seven of which spent at Sony Computer Entertainment Europe, he has had key technical roles on triple-A titles like the Bafta Award Winning Little Big Planet (PSP), 24: The Game (PS2), special effects work on Heavenly Sword (PS3), some in-show graphics on the BBC's version of Robot Wars, the TV show, as well as a few more obscure projects. Now joint CEO of Wildbunny, he is able to give himself hiccups simply by coughing.
1NobNQ88UoYePFi5QbibuRJP3TtLhh65Jp
View all posts by Paul Firth →

This entry was posted in AS3, Geometry, Graphics, Technical and tagged adobe flash, bounding box, game developement, tutorial. Bookmark the permalink.

5 Responses to *Implementing a wrap-around world*

raigan *says:*
May 9, 2012 at 4:38 pm

Warning: I might be one of those “trivial!” people

Just modulo-ing the position isn’t a proper solution, but isn’t that general idea — treating the x and y coordinates “like angles” (e.g 1d points in coordinate systems which wrap) — useful?

Specifically, for AABB and circle collision, you just need a delta between two points.

In a non-wrapping world this is d = b – a.

For angles, you would do: (assuming a and b are in [0,2pi])
d = b – a;
if(d pi) d -= 2pi;

This makes sure the delta is the shortest path between a and b; in a non-wrapping space there’s only one vector between two points, but in a wrapping space there are two and we want to choose the shortest.

Couldn’t you do the exact same thing for the vector between two points in a toroidal space?

dx = bx – ax;

if(dx 0.5*width) dx += width;

..and then the same for dy; now you have the correct shortest vector between two points, which is all you need for collision detection.

At least, AFAICT it should be possible to do all the simulation as if you were in a non-wrapping space, simply by adjusting the delta vectors.

I’ve never actually tried implementing this though, so possibly I’m an idiot and missing something fundamental!

Drawing certainly seems like it might need explicit duplication though, since as you pointed out each object needs to be drawn 1-4 times depending on which axes it straddles.

Anyway, thanks yet again for an awesome tutorial, this and the rolling one are both really interesting little concepts; there are so many of these little details which never get discussed in e.g Game Programming Gems but which are in many ways more fundamental/useful.

Reply

Paul Firth *says:*
May 9, 2012 at 10:44 pm

Hi Raigan,

Yes, the technique you describe will work – its basically about making sure the things you want to collide are in the same ‘space’, using a wrapped delta like you suggest is one way of doing that 😀

Cheers, Paul.

Reply

raigan *says:*
May 10, 2012 at 6:00 pm

Sorry for all the comment pollution, please feel free to delete all of these 😀

Reply

raigan *says:*
May 9, 2012 at 4:39 pm

(crap, it ate my less-than and greater-than symbols…)

Reply

raigan *says:*
May 9, 2012 at 4:46 pm

Anyway, the wrapping can be implemented however you want, but the idea is to bring the delta back into [-halfwidth, halfwidth] so that it’s always the shortest vector between two points.

Reply