How to make Angry Birds – part 2

by PAUL FIRTH • JUNE 7, 2011

Hello and welcome back to my blog!

This is the 2nd part of my series on how to make a game like Angry Birds by Rovio - and its been a while coming... You can read the first part here.

The Game

Ok, so here is the game so far; there are three demo levels to show the level progress system and some simple looking characters and block types. Apologies for the programmer art

Catch up

Ok, so last time I had covered how to draw the background graphics and made a start on how the world is going to be composed in terms of collision.

What I'm going to cover in this article is the physics part; stability and optimisations. The physics engine that I've created for this game is based on the technology I've discussed in my previous articles:

Physics engines for dummies
Collision detection for dummies
Speculative contacts – a continuous collision engine approach

Level

A level is defined as a bunch of *instances* of static rigid bodies and dynamic rigid bodies plus a visual layer which covers up the statics – I use this other layer because otherwise there would be a slight visible gap between adjacent static blocks, which we don't want since the texture is supposed to be continuous.

A level must inherit the base class *Code.Level*. I've done this so I can predefine certain things about each level – such as the instance names for the hero characters, and the sling-shot which I need to reference in code.

Characters and blocks

Each of the two characters and every block used in the game are designed in the Flash IDE, and each of them references a base class that I've defined in the code. This is so that I can parse the children of a level at runtime and create the appropriate rigid bodies at the correct mapped locations for the physics engine.

Rigid-bodies

There are the following base classes (stone and wood use the same base class):

- HeroCharacter
- EnemyCharacter
- ProxyRectangleIce
- ProxyRectangle
- ProxyRectangleStatic
- ProxyTriangleStaticDown
- ProxyTriangleStaticUp

These are interpreted at level creation time, using a loop similar to this:

```
// create physics objects to go with the render objects in level
for ( var i:int = 0; i<level.numChildren; i++ )
{
    var child:DisplayObject = level.getChildAt( i );
    if ( child is Character )
    {
        ...
    }
    else if ( child is ProxyRectangleStatic )
    {
        ...
    }
    else if ( child is ProxyRectangle )
    {
        ...
    }
    else if ( child is ProxyRectangleIce )
    {
        ...
    }
    else if ( child is ProxyTriangleStaticDown )
    {
        ...
    }
    else if ( child is ProxyTriangleStaticUp )
    {
        ...
    }
    child.cacheAsBitmap = true;
}
```

In each part of the if-else ladder I create the appropriate rigid body to match the proxy, and these get added to the physics engine.

Level sequence

Thanks to Flash's neat way of storing class types as variables, I represent the level sequence as a simple array of class types:

```
private var m_currentLevel:int;
private var m_levelSequence:Array =
[
    Level3Fla,
    Level2Fla,
    LevelFla
];
```

Every time a level is completed, I can pick the new one from the array and instantiate it like this:

```
// create new level
m_level = new m_levelSequence[m_currentLevel];
```

It then gets passed to the *SetupLevel* function which handles all the rigid body instantiation that I talked about above.

Physics engine

The physics engine is similar in design to the one I laid out in Physics engines for dummies, except this one is a lot more advanced.

It now has the concepts of *angular-velocity*, *polygon contact set generation*, *contact group generation*, *object sleeping*, *collision callbacks* and *persistent contacts*. All these components were necessary to create the game.

Physics engine – angular velocity

Angular velocity is handled in a very similar way to linear velocity, except its a scalar and not a vector, since we're in 2d and it gets integrated the same way. Where it gets more tricky is in the actual impulse equations.

I'm going to refer the reader to the following set of articles by Chris Hecker which describe the impulse equations for angular and linear velocity rather well: http://chrishecker.com/Rigid_Body_Dynamics.

I do intend to write a more in depth follow up on this subject at some point.

Physics engine – polygon contact set generation

Although, technically when two convex shapes collide there are only two closest points (one on each object) this is not enough for the physics engine to produce a stable simulation with anything more than circles. As soon as you have a rectangle or a polygon you need to have the full contact-set between both objects.

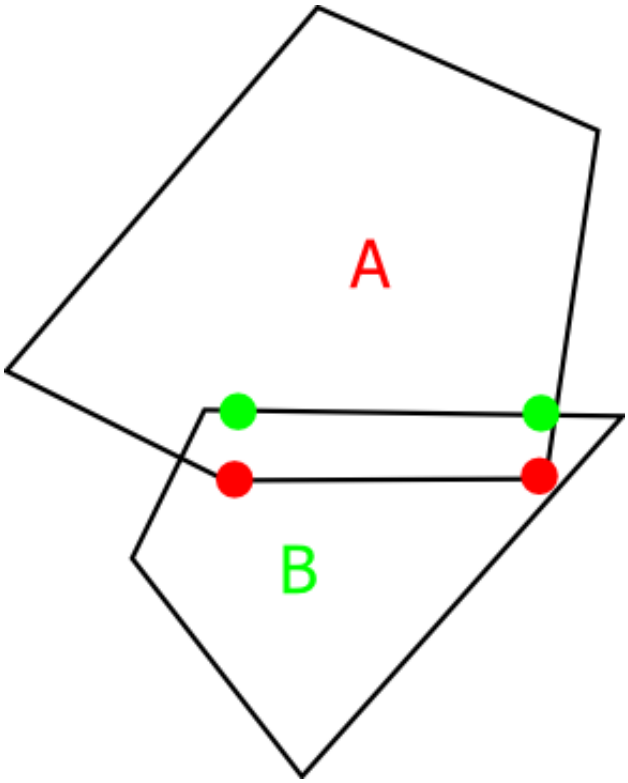


Figure 1

Figure 1 shows the full contact set between two polygons A and B. To generate this we need to start with the *feature-pair* returned by our collision system (vertex and edge, or edge and vertex).

Figure 2

Figure 2 shows one such possible case: a vertex from B and an edge from A were returned as the closest features. Starting with this information we can then do a local search of the the vertices adjacent to the red point on B (one on either side) to find which edge has a normal that is most opposed to our collision normal (the edge normal).

Figure 3

Figure 3 shows the correct edge and the other edge which was considered from B. Once we have these two edges we have all the information we need to generate the contact set. What we do is to project each vertex from each edge onto the other edge, making sure to clamp the projections so they lie wholly within the bounds of the edge.

Figure 4

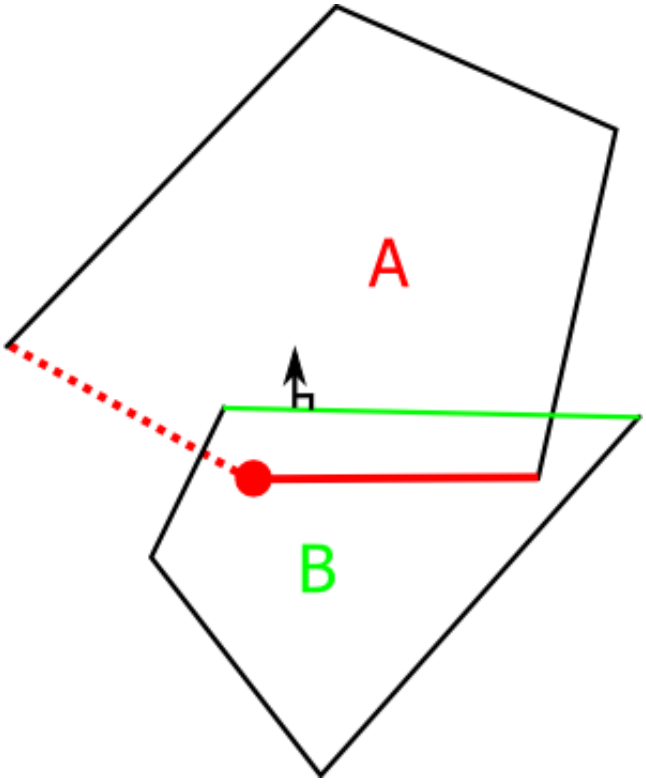
Figure 4 shows the red vertices of the edge from A being projected onto the green edge of B. The first vertex becomes projected point *b0* and the second becomes *b1*.

Figure 5

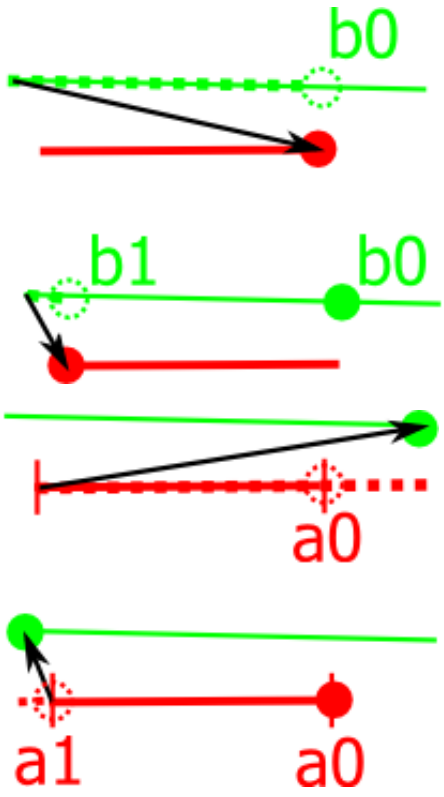
In Figure 5 we can see the same process repeated for the other edge; note how both projections of the green vertices would lie outside the bounds of the red edge. Because this would lead to an invalid contact we must clamp these at the bounds of the red edge. Vertices *a0* and *a1* result and this is the completed contact set!

Figure 6

Here is the code for projecting a point onto an edge and clamping against the edge bounds:




```
public function ProjectPointOntoEdge(p:Vector2
, e0:Vector2, e1:Vector2):Vector2
{
    // vector from edge to point
    var v:Vector2 = p.Sub(e0);

    // edge vector
    var e:Vector2 = e1.Sub(e0);

    // time along edge
    var t:Number = e.Dot(v) / e.m_LenSqr;

    // clamp to edge bounds
    t = Scalar.Clamp(t, 0, 1);

    // form point and return
    return e0.Add( e.MulScalar(t) );
}
```



Contact group generation and object sleeping

The game logic for angry birds relies on being able to tell when all the movement in the level is stopped, because otherwise you would never know when an individual try is over since objects may yet fall down crushing an enemy character.

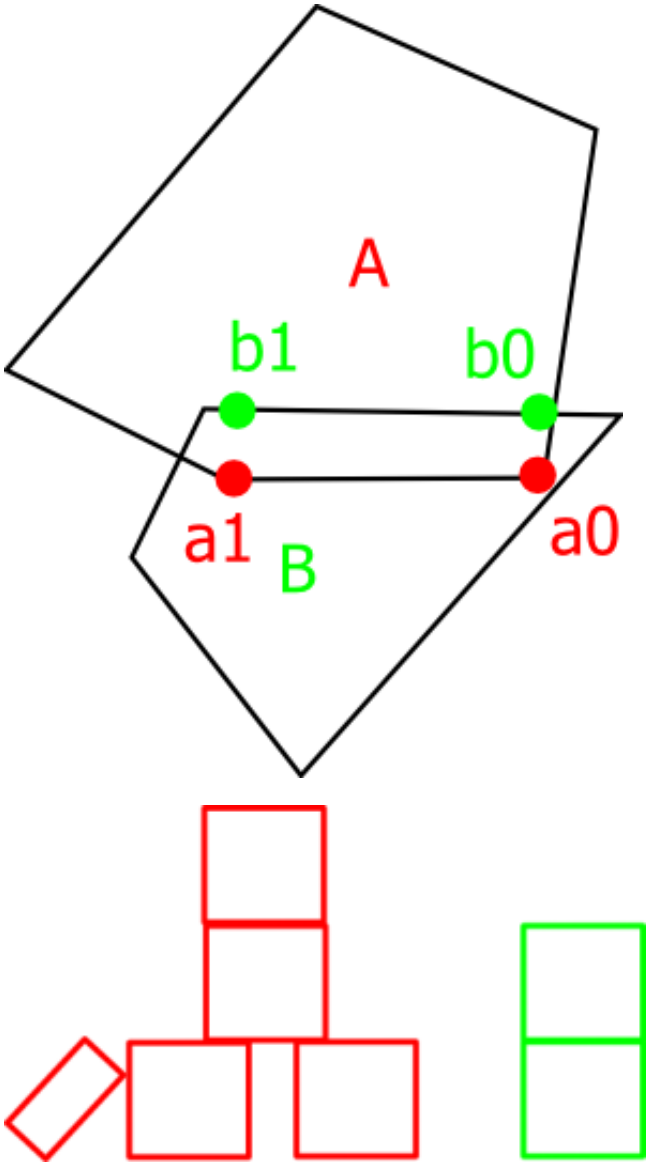
In order to facilitate this and also to act as an optimisation an object sleeping system is required. What this does is to deactivate or 'sleep' any rigid bodies which have become sufficiently still for a set period of time, thereby saving CPU cycles and also giving us an indication about the state of the game.

For this to be possible we first have to be able to generate *contact groups*. A contact group is a collection of rigid bodies which are all touching each other.

Figure 7

Figure 7 shows two contact groups.

In order to generate these, the simplest way is to use recursion. Each object must maintain a list of every object that's touching it. An outer loop over all rigid bodies generates individual contact group containers and then recurses within adding objects which are touching to the given contact group.



Static objects are not followed because we don't want to include them in any contact group – they would link the entire level together into one giant, inefficient contact group!

The code looks like this:

```
private function BuildContactGroup( r:RigidBody
, cg:ContactGroup, dt:Number ):void
{
    if ( !r.m_visited && !r.m_Static )
    {
        r.m_visited = true;
        r.m_contactGroup = cg;

        // add the object
        cg.Add( r );

        for ( var i:int = 0; i<r.m_rbsTouching.
m_Num; i++ )
        {
            BuildContactGroup( r.m_rbsTouching.
Get(i), cg, dt );
        }
    }
}

///
/// Go through and build all the contact groups
 in the world
///
internal function BuildAllContactGroups( dt:Num
ber, rigidBodies:ReferenceArray ):void
{
    Reset( );

    // start with a new group
    var cg:ContactGroup=null;

    for ( var i:int=0; i<rigidBodies.m_Num; i++
 )
    {
```

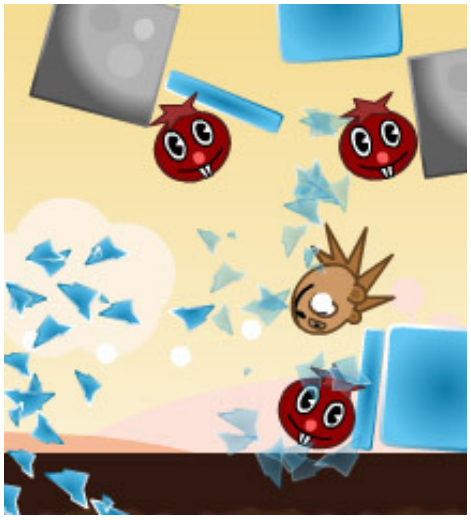
```
            var r:RigidBody = rigidBodies.Get(i);

            // if this object isn't static and hasn
't yet been visited...
            if ( !r.m_Static && r.m_visited==false
)
            {
                if ( cg==null || cg.m_NumRigidBodie
s>0)
                {
                    // get new contact group
                    cg = GetNewContactGroup( );
                }

                // go an recursivly add objects to
this new contact group
                BuildContactGroup(r, cg, dt);
            }
        }

        // sleep check the contact groups
        SleepCheck( );
    }
```

Its actually quite simple and works rather well, taking minimal CPU time.

To enable the sleeping of rigid bodies, each rigid body maintains a counter which counts the number of seconds that rigid body has had angular and linear velocities below some threshold values. Then, once all the rigid bodies in a contact group have counter values over a threshold (1 second in this game's case), the entire contact group is sent to sleep. Likewise, if any rigid body in a sleeping contact group wakes up, the entire contact group must wake up with it.

Physics engine – collision callbacks

For some objects the game needs to know about any collisions that happen between those objects and the hero character, or the world. To facilitate this, each rigid body has a collision callback delegate which can be set at runtime. Then, the physics engine will call this delegate whenever it detects a collision, passing the callback information about the collision including which objects were hit and what the relative normal velocities were at the time of collision. This information allows to game logic to kill enemies and smash blocks of ice.



Physics engine – persistent contacts

This physics engine feature is absolutely essential to the stability of the physics; without it, the game would not be possible with as much creative freedom in level design.

So, what are persistent contacts?

Regular contacts are the things which stop two rigid bodies from falling through each other – they are generated by the collision detection system and used by the physics solver but they are temporary and exist for the current frame only.

The impulses generated by the physics solver converge the entire simulation towards stability, but there are not enough CPU cycles for it to be resolved completely in one frame… So rather than throw away all of last frames impulses we would like to be able to remember them and then apply them next frame in order to 'warm start' the engine. This leads to massive stability improvements.

Persistent contacts are a way of caching the impulses between frames by identifying contacts which are logically the same across a series of frames. By logically I mean rather than using something crude like the position of two contacts, we want to use something which identifies contacts uniquely. To do this we can use feature pair indicies; so for example, the index of the vertex from object A combined with the index of the edge from object B. These will be combined together into one uint hash value which can be looked up and compared across frames. When the hash tags match, we have a cache hit and can copy the impulses across.

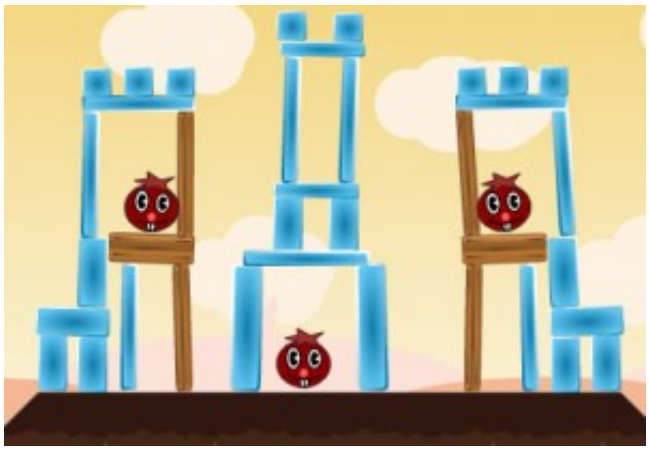
The code I use for caching the impulses looks like this:

```
public class TouchingContact
{
    public var m_featurePairKey:uint;
    public var m_accumulatedNormalImpulse:Number;
    public var m_accumulatedTangentImpulse:Number;

    public function TouchingContact( featurePairKey:int )
    {
        m_featurePairKey = featurePairKey;
        m_accumulatedNormalImpulse = 0;
        m_accumulatedTangentImpulse = 0;
    }

    ///
    /// Something to identify this contact uniquely
    ///
    static public function GenerateFeaturePairKey( featureToUse:FeaturePair, supportIn
dex:int ):uint
    {
        return featureToUse.m_fp|( featureToUse.m_face<<2 )|( supportIndex<<16 );
    }
}
```

When two objects touch for the first time, a persistent contact entry is generated for the collision and stored on the lower indexed object, so that I avoid duplicating data. When objects stop touching, these contact entries are deleted again. During the time when they are touching, I cache up to 4 feature pairs for later lookup – the reason I chose 4 and not 2 (which would be the logical choice) is that I noticed there was a fair amount of flip-flop over the course of a few frames in certain contact configurations; one frame two contacts would be generated from a certain feature pair, the next frame another two contacts from a different feature pair and then repeat forever. Rather than throw away and regenerating, storing 4 allowed me to catch this case and achieve a 100% cache hit rate for stable contact configurations.

Optimisation

After I had implemented all these features and gotten the engine stable enough to be able to run angry birds, I noticed that it was in fact, far too slow to actually use. This made me sad, but a little digging into action-script optimisation techniques netted me the answer.



References

In action-script, every complex type is passed by reference and there are no complex types which can be put on the stack, unlike c# or c++. This is particularly irksome if you have something like a Vector2 class for doing all your geometric calculations, because it means every time you do a new(), an allocation is made on the **heap**, which of course means you risk the slowness of the garbage collector when doing temporary calculations.

This is an absolute killer in terms of performance – I found I was doing something on the order of **7k-12k** such temporary allocations every single frame!

The solution – pools

Of course, the internet informed me that the accepted solution here is to employ an *object pool* for complex types – pre-allocated up-front with a known fixed capacity. Allocations can then be made and freed using the pool, thereby avoiding the garbage collector.

I couldn't find an implementation that did exactly what I wanted, so I wrote my own:

```
package Code.System
{
    import Code.Assert;
    import Code.System.UnexpectedCaseException;

    public class Pool
    {
        private var m_counter:int;
        private var m_maxObjects:int;
        private var m_pool:Array;
        private var m_type:Class;

        /// <summary>
        ///
        /// </summary>
        public function Pool( maxObjects:int, type:Class )
        {
            m_pool = new Array( maxObjects );

            // construct all objects
```

```
    // construct all objects
    for (var i:int=0; i<maxObjects; i++)
    {
        m_pool[i] = new type();
    }

    m_counter=0;
    m_type=type;
    m_maxObjects=maxObjects;
}

/// <summary>
///
/// </summary>
public function Allocate( ...args ):*
{
    Assert( m_counter<m_maxObjects, "Pool.GetObject(): pool out of space!" );

    var obj:* = m_pool[m_counter++];

    if ( args.length>0 )
    {
        if ( args.length==1 )
        {
            obj.Initialise( args[0] );
        }
        else if ( args.length==2 )
        {
            obj.Initialise( args[0], args[1] );
        }
        else if ( args.length==3 )
        {
            obj.Initialise( args[0], args[1], args[2] );
        }
        else if ( args.length==4 )
        {
            obj.Initialise( args[0], args[1], args[2], args[3] );
        }
        else if ( args.length==5 )
        {
            obj.Initialise( args[0], args[1], args[2], args[3], args[4] );
        }
```

```
                else if ( args.length==6 )
                {
                    obj.Initialise( args[0], args[1], args[2], args[3], args[4], args[
5] );
                }
                else if ( args.length==7 )
                {
                    obj.Initialise( args[0], args[1], args[2], args[3], args[4], args[
5], args[6] );
                }
                else
                {
                    throw new UnexpectedCaseException;
                }
```

Original URL:

http://www.wildbunny.co.uk/blog/2011/06/07/how-to-make-angry-birds-part-2/