Farseer Physics Engine 2.0 Manual

## Introduction

The Farseer Physics Engine is an easy to use 2D physics engine designed specifically for Microsoft's XNA platform. The Farseer Physics Engine focuses on simplicity, useful features, and enabling the creation of fun, dynamic games.

## Overview

Getting right to the nut, the Farseer Physics Engine is designed to control the position and rotation of game entities over time.

In the real world, things move and spin due to applied forces and torques. In Farseer, the same is true. Objects called "Bodies" represent the real world things.  As forces and torques are applied, the bodies react according to the laws of 2D physics. The position and rotation of these bodies are then used to update game entities.

In the very simplest outline it works like this:

1. Create "Body" object
2. Add Body object to simulator.
3. Begin Game Loop
   1. Apply forces and torques to Body.
   2. Update the simulator
   3. Set game entity(sprite or whatever) position and rotation equal to Body position and rotation.
   4. Draw game entity
4. End Game Loop

Bodies, by design, have no geometry in the 2D world and therefore have no concept of collisions.

For collision, Farseer has the "Geometry" object. Geometry objects are represented as 2D polygons and can be either concave or convex. They are defined by a set of vertices.  One or more Geometries are attached to a Body in order to give the Body geometrical awareness. This allows the Body to participate in collisions with other Bodies(actually other Geometries attached to other Bodies, but you get the picture.)

## Body

Body is the core physics object in Farseer. Forces, torques, and impulses are applied to bodies and the bodies react by moving realistically. Bodies do not contain any form of collision geometry by themselves. You will need to create a Geom for collision detection. To create a body you usually use BodyFactory.

You create a body like this:

int mass = 1;

float width = 128;

float height = 128;

Body rectBody = new Body();

rectBody.Mass = mass;

rectBody.MomentOfInertia = mass \* (width \* width + height \* height) / 12;

As you can see, calculating MOI (Moment Of Inertia) for a rectangle is done like this:

D:\Projects\TFS\FarseerPhysics\Documentation\ManualImages\MOIRectangle.png

Instead of remembering MOI for each shapes, Farseer can calculate the MOI for you. All you have to do is use the BodyFactory class as described below.

### Body Factory

You create a body with the factory like this:  
Body rectBody = BodyFactory.Instance.CreateRectangleBody(PhysicsSimulator, 128, 128, 1);  
  
This body has a size of 128 width, 128 height and a mass of 1. The MOI (Moment Of Inertia) is calculated for you. Note that the body is added right away by adding the PhysicsSimulator as a parameter.

You can create each of these type of bodies with the factory:

* Rectangle
* Circle
* Polygon
* Body

The last item (body) is for when you want to create a body, without Farseer calculating the MOI for you. You can also use it to create clones of bodies.

There is some overloads for each BodyFactory method. One that takes a PhysicsSimulator object and another one that don't. If you provide a PhysicsSimulator object, the body you create will be added directly to the simulator.

## Geometry

The geometry (Called Geom in Farseer code) is the heart of collision detection. A geometry need a body and a set of vertices that define the edge of your shape.

While the body is in control of forces, torques, and impulses, the geometry is in control of collision detection and calculating the impulses associated with colliding with other geometries.

To create a geometry, do the following:

Body rectBody = BodyFactory.Instance.CreateRectangleBody(128, 128, 1);

Vertices vertices = new Vertices();

vertices.Add(new Vector2(-64, -64));

vertices.Add(new Vector2(64, -64));

vertices.Add(new Vector2(64, 64));

vertices.Add(new Vector2(-64, 64));

Geom rectGeom = new Geom(rect,vertices,11);

This will create a rectangle body (by using the BodyFactory) a set of vertices that represent the outline of a rectangle (relative to 0, 0) and create a new geometry with the defined vertices and a grid cell size of 11.

We will have a deeper look into grid cell size in the **Grid** chapter.

### GeomFactory

Another and much easier way to create a geometry is by using the GeomFactory. The GeomFactory can create a vertices collection for simple shapes such as rectangle and circle. All you need is the width, height or radius.

Here is an example of creating a Geom using the GeomFactory:

Body rectBody = BodyFactory.Instance.CreateRectangleBody(128, 128, 1);

Geom rectGeom = GeomFactory.Instance.CreateRectangleGeom(PhysicsSimulator, rectBody, 128, 128);

Notice that you don't have to supply any vertices or grid cell size. The GeomFactory creates the vertices for you and calculates the grid cell size for you.

There are situations where you would want to control the grid cell size. This is also very easy with the GeomFactory, just use the overloaded methods that takes a grid cell size:

Geom rectGeom = GeomFactory.Instance.CreateRectangleGeom(PhysicsSimulator, rectBody, 128, 128, 12.8f);

Now we have a grid cell size of 12.8.

If you pass 0 into the grid cell size, it will get calculated for you. They way it's calculated is by finding the shortest side of the geometry (128 in this instance) and multiply it by the default grid cell size factor of 0.1. This would yeld 12.8.

You can adjust the default grid cell size by setting the **GridCellSizeAABBFactor** on the GeomFactory object.

## Joints

Farseer Physics Engine provide you with some of the basic joints. You can create almost any dynamic behavior by combining these joints:

* Revolute joint\*
* Angle joint\*
* Angle limit joint\*
* Pin joint
* Slider joint

\* Has a "fixed" version. The fixed versions mean that the joint is anchored to the world, and not to another body as their non-fixed versions.

### Revolute Joint

### Angle Joint

### Pin Joint

### Slider Joint

### Joint Factory

## Springs

### Linear Spring

### Angle Spring

## Collision detection

Farseer Physics Engine provides you with an easy to use collisions system containing 4 different parts:

1. Broad phase collision detection
2. Narrow phase collision detection
3. AABB (Axis Aligned Bounding Box)
4. Grid collision detection

Each system is described below:

### Broad phase

The broad phase collision detection relies on advanced algorithms to speed up the collision detection, by reducing the work the engine has to do.

We currently have 3 kinds of broad phase collision detection algorithms:

1. Sweep And Prune (called SAP)  
2. Selective Sweep  
3. Brute Force

The **Sweep And Prune** algorithm is frame coherent, this means that if objects more a lot around the screen, this might be a bad choice. This also means that if your objects are near the position they were the last frame, this algorithm is good.

Also note that the SAP algorithm does not like teleporting objects or very high speed objects such as moving from one end of the world to the other or bullets. It may break down from it.

More information on SAP can be found [here](http://en.wikipedia.org/wiki/Sweep_and_prune) and [here](http://www.ziggyware.com/readarticle.php?article_id=128) (called sort and sweep).

The **Selective Sweep** algorithm is developed by BioSlayer. The SS algorithm is the default one in Farseer Physics Engine. SS was originally build on Sweep And Prune, but had some changes that made it perform better than SAP.

More information on SS can be found [here](http://www.codeplex.com/FarseerPhysics/Thread/View.aspx?ThreadId=17811).

The **Brute Force** algorithm is the most simple of them all, but also the least performing of the 3. It iterates all the geometries in the world and compare their AABB's. The Brute Force algorithm is O(n^2) complexity, but is still very fast for low geometry count.

### Narrow phase

The narrow phase is where we take all the collision pairs generated by the broad phase, and do further calculation on them. All the narrow phase code lives inside the **Arbiter** class.

Here is a short overview of what happens in the narrow phase:  
We assume that the broad phase provided us with a pair of colliding geometries contained in an Arbiter object.

1. Iterate all the world vertices on the first geometry
2. If the current vector intersects with the second geometry
   1. If false: Continue to next vector in vertice list
   2. If true: Create a contact and insert it in a contact list
3. Do 1. and 2. on the second geometry
4. If there are any contacts in the contact list, fire the OnCollision event providing the 2 geometries and the contact list.

The Arbiter class is also used to calculate the impulse, that should be applied to the geometries, when they collide.

### AABB

AABB stands for Axis Aligned Bounding Box and as the names says, it's a bounding box that aligns itself to an axis. All geometries have an AABB that is recalculated on each update, AABB are relative inexpensive and used to quickly test if 2 geometries are close to each other (or even touching).

You can test if 2 geometries are close to each other by doing this:

if (AABB.Intersect(\_circleGeom.AABB,\_rectangleGeom.AABB))  
{  
 //The 2 AABB's intersect  
}

Remember that because the AABB's are not rotated and they outline the geometry with a rectangle, when you test for intersection between 2 AABB's, the geometries might actually not touch. Have a look at these pictures.

|  |  |
| --- | --- |
| Intersection  C:\Users\Genbox\Desktop\Intersection.png | Intersection and collision (touching)  C:\Users\Genbox\Desktop\IntersectionTouching.png |

The AABB is the black outline of the geometries. As you see, they are not rotated, but are axis aligned. If you look real close, you can see that the contact created when colliding is red.

### Grid

All geometries contains a Grid object. It's used by the narrow phase collision detection. This method is also called "distance grid" and uses a pre-calculated 2D array to test for collisions.

The grid is calculated from the provided grid cell size in the Geom constructor or when using the GeomFactory

Calculation of the grid can be quite time consuming

### Collision group and categories

Farseer provides you with a way of creating different collision groups and the more advanced collision categories.

By default all geometries are in **collision group** 0, this means that it collides with all other geometries. If two geometries are in the same collision group, they will not collide with each other, the 0 collision group is an exception.

Here is how to set the collision group on a geometry:

Body rectBody = BodyFactory.Instance.CreateRectangleBody(PhysicsSimulator, 128, 128, 1);

rectBody.Position = new Vector2(250, 400);

Geom rectGeom = GeomFactory.Instance.CreateRectangleGeom(PhysicsSimulator, rectBody, 128, 128);

rectGeom.CollisionGroup = 10;

Body circleBody = BodyFactory.Instance.CreateCircleBody(PhysicsSimulator, 64, 1);

circleBody.Position = new Vector2(300, 400);

Geom circleGeom = GeomFactory.Instance.CreateCircleGeom(PhysicsSimulator, circleBody, 64, 20);

circleGeom.CollisionGroup = 10;

Even if rectGeom and circleGeom are overlapping each other, they will not collide with each other.

While collision groups are easy to use, they can be very limited, so **collision categories** also exist.

There are two properties of interest when using collision categories:

1. CollisionCategories
   1. Defaults to CollisionCategory.All
   2. Used to define witch categories the geometry a member of.
2. CollidesWith
   1. Defaults to CollisionCategory.All
   2. Used to define witch categories the geometry collides with.

Collision categories uses an enum called CollisionCategory that has a special flag enabled on it, so it's able to do bitwise operations. (more info [here](http://msdn.microsoft.com/en-us/library/system.flagsattribute.aspx))  
  
Example:

Body rectBody = BodyFactory.Instance.CreateRectangleBody(PhysicsSimulator, 128, 128, 1);  
rectBody.Position = new Vector2(250, 400);  
Geom rectGeom = GeomFactory.Instance.CreateRectangleGeom(PhysicsSimulator, rectBody, 128, 128);  
rectGeom.CollisionCategory = CollisionCategories.Cat5;

rectGeom.CollidesWith = CollisionCategories.All & ~CollisionCategories.Cat4;

Body circleBody = BodyFactory.Instance.CreateCircleBody(PhysicsSimulator, 64, 1);

circleBody.Position = new Vector2(300, 400);

Geom circleGeom = GeomFactory.Instance.CreateCircleGeom(PhysicsSimulator, circleBody, 64, 20);

circleGeom.CollisionCategory = CollisionCategories.Cat4;

circleGeom.CollidesWith = CollisionCategories.All & ~CollisionCategories.Cat5;

This time, the rectGeom is a member of Cat5 (Category 5) and collides with All **but** Cat4

The circleGeom is a member of Cat4 and collides with All **but** Cat5.

This means that the two geometries will not collide with each other.

### Collision Events

There are 3 different collision events:

* OnCollision (in Geom class)
* OnSeparation (in Geom class)
* OnBroadPhaseCollision (in IBroadPhaseCollider interface)

The **OnCollision** event is fired when the geometry hits another geometry. You will need to return a boolean inside the event method to indicate if you want the collision to happen or not.

The **OnSeparation** event is fired when the geometry is separated after a collision with another geometry.

The **OnBroadPhaseCollision** event is just like the OnCollision event, but is fired already in the broad phase. Canceling this event prevents an arbiter from being constructed, this means that no impulses are applied and no narrow phase collision is done, to the geometries involved in the collision.

To register the events, do the following:

Body circleBody = BodyFactory.Instance.CreateCircleBody(PhysicsSimulator, 64, 1);

Geom circleGeom = GeomFactory.Instance.CreateCircleGeom(PhysicsSimulator, circleBody, 64, 20);

circleGeom.OnSeparation += OnSeperation;

circleGeom.OnCollision += OnCollision;

PhysicsSimulator.BroadPhaseCollider.OnBroadPhaseCollision += OnBroadPhaseCollision;

Note that the OnBroadPhaseCollision event is registered inside the BroadPhaseCollider of the PhysicsSimulator.

And the methods that is run when the events are fired:

private bool OnCollision(Geom geom1, Geom geom2, ContactList contactList)

{

return true;

}

private void OnSeperation(Geom geom1, Geom geom2)

{

}

private bool OnBroadPhaseCollision(Geom geom1, Geom geom2)

{

return true;  
}

## Impulses

## Performance

Remember to use release compilation.  
Minimal number of bodies/geometries  
Caching  
Multithreading  
Inactivity controller  
Scaling

## Known issues

Tunneling  
Geometries going into each other  
Drawing is off center (http://www.codeplex.com/FarseerPhysics/Thread/View.aspx?ThreadId=28747)  
Circle - Circle optimization: http://www.codeplex.com/FarseerPhysics/Thread/View.aspx?ThreadId=19861