

Farseer Physics Engine 2.0.1 Manual

Contents

Introduction	2
Overview	2
Body	3
Body Factory	3
Geometry	4
GeomFactory.....	5
Physics Simulator View	5
Joints	7
Revolute Joint.....	9
Angle Joint.....	10
Angle Limit Joint.....	10
Pin Joint.....	11
Slider Joint.....	12
Joint Factories	12
Springs.....	13
Linear Spring	15
Angle Spring	15
Spring Factories.....	16
Collision detection	17
Broad phase	17
Narrow phase.....	18
AABB.....	18
Grid.....	19
Collision group and categories.....	20
Collision Events	22
Impulses	23
Physics Properties	24

Extensions to Farseer	26
Texture to vertices	26
Path Generator.....	27
Performance.....	27
Known issues.....	32

Introduction

The Farseer Physics Engine is an easy to use 2D physics engine. It supports a wide range of platforms such as Microsoft's XNA, Silverlight, WPF, and Vanilla .NET. 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
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

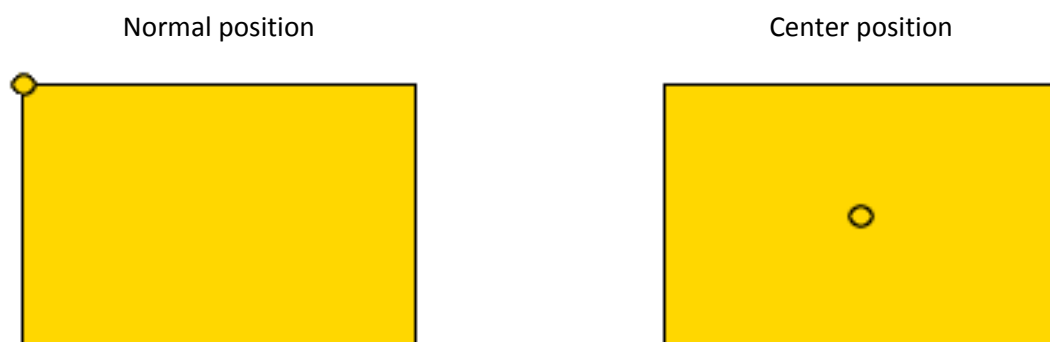
Body is the core physics object in Farseer. Forces, torques, and impulses are applied to bodies and the bodies react by moving accordingly. Bodies do not contain any form of collision awareness by themselves. To create a body you usually use the BodyFactory, but first, here is how you create a body manually:

```
int mass = 1;
float width = 128;
float height = 128;
Body rectBody = new Body();
rectBody.Mass = mass;
rectBody.MomentOfInertia = mass * (width * width + height * height) / 12;
rectBody.Position = new Vector2(100, 200);
```

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

$$I_c = \frac{m(h^2 + w^2)}{12}$$

Instead of remembering MOI for each shape, Farseer Physics can calculate the MOI for you. All you have to do is use the BodyFactory class as described in the "Body Factory" chapter. One thing to note about bodies is that their position is relative to the center of the body.



This has relevance to your positioning and drawing code. When you position your Body, you will need to position it relative to the center of the Body. Also, when you want to draw the Body onto the screen, make sure that you draw it relative to the center.

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

Geometry

The geometry (Called Geom in Farseer) is the heart of collision detection. A geometry needs a body and a set of vertices (laid out counter-clockwise) 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 you usually use the GeomFactory, but first, here is how you create a geometry manually:

```
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(rectBody, vertices, 11);
```

This will create a rectangle body, a set of vertices that represent the outline of a rectangle (relative to 0, 0) and a new geometry with the defined vertices. The 11 inserted as a parameter in the Geom constructor is the grid cell size.

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 for rectangles or radius for circles.

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

```
Body rectBody = BodyFactory.Instance.CreateRectangleBody(128, 64, 1);
Geom rectGeom = GeomFactory.Instance.CreateRectangleGeom(PhysicsSimulator,
rectBody, 128, 64);
```

Notice that you don't have to supply any vertices or grid cell size. The GeomFactory creates the vertices 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, 64, 6.4f);
```

Now we have a grid cell size of 6.4. (the f after the number indicates in C# that it's a float)

If you pass 0 into the grid cell size, it will get calculated for you. The way it's calculated is by finding the shortest side of the geometry (64 in this instance) and multiply it by the default grid cell size factor of 0.1. This would yield 6.4.

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

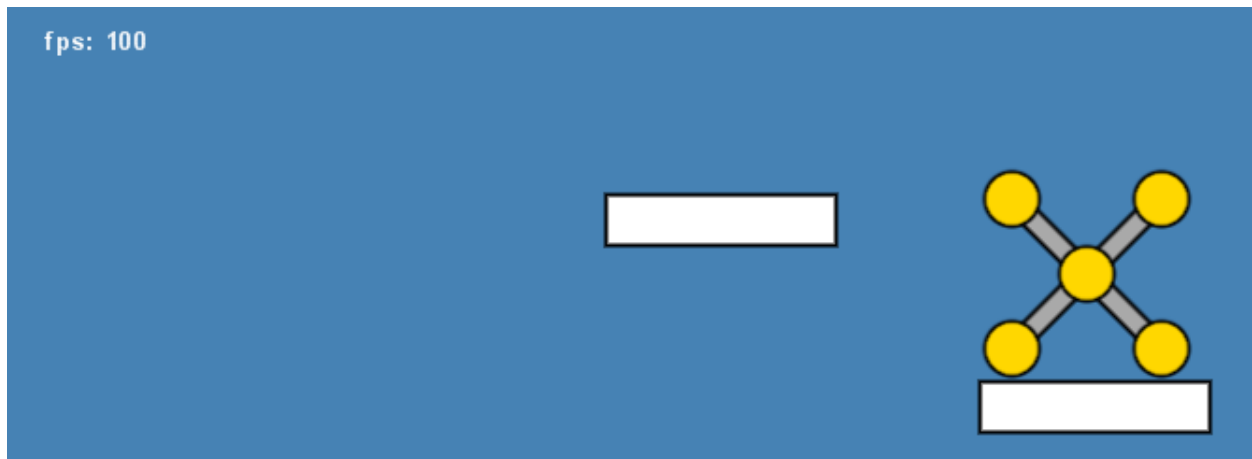
Physics Simulator View

Physics simulator view is used for debugging joint anchors, body positions, geometry alignment and collisions. When you activate the view, you will see the anchors, collisions, contact points,

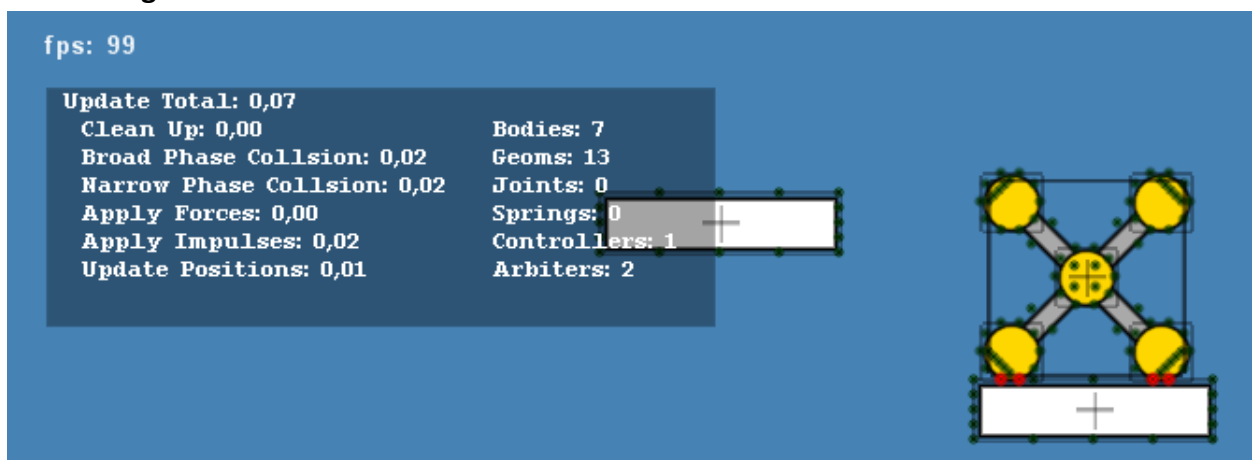
centers, AABB's and a lot of other information that might be essential in debugging a Farseer physics related problem.

The physics simulator view only work in XNA. There will be debug views for other platforms in the future.

Without debug view:



With debug view:



Explanation:

There is a lot of configuration for the debug view. Here is a list of the possibilities:

- **Performance panel**
 - Shows some information about the timing of the engine and the current count of bodies, geometries, joints, springs, controllers and arbiters.
 - Clean up: The time it took to add and remove geometries, bodies, joints, controllers and springs in the last update.
 - Broad Phase Collision: The time it took to do the broad phase collision detection.

- Narrow Phase Collision: The time it took to do the narrow phase collision detection.
- Apply Forces: The time it took to apply forces to all springs, controllers and bodies.
- Apply Impulses: The time it took to apply impulses to joints and arbiters
- Update Positions: The time it took to update the position of all bodies
- **Vertices**
 - Shows the vertices that makes up the geometry. They are viewed as small black pixels around the geometry.
- **AABB**
 - Shows the Axis Aligned Bounding Box (more on AABB's in the AABB chapter).
- **Contacts**
 - Shows the vertices that have contact. Shown as small red pixels.
- **Coordinate axis**
 - Shows the center of bodies. Shown with a black cross.
- **Grid**
 - Shows the geometries distance grid used in narrow phase collision detection. There can be a lot of points in the grid, so this option might slow down the drawing a lot. Shown as black circles. (not enabled on the debug view picture above)
- **Edge**
 - Shows the outlines of the edge of the geometries. Shown as a black edge.
- **Joints**
 - Revolute, Pin and Slider joints have visualizations.
- **Springs**
 - Linear springs (both fixed and normal) are shown with a black line between the two end points. 3 circles are positioned on the line to show the contraction and expansion of the line.

Joints

Farseer Physics Engine provides 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

- Gear joint – experimental

* 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.

Important Notes for all joints

- Some joints need anchors relative to the bodies' position and some need world anchor points. So pay attention to the type of joint your using.
- Joints are attached to bodies.
- All joints share some variables and methods.
- Anchors don't have to be inside a bodies attached geometry.
- When joints break, they are disabled.

If joints don't work as expected

1. Check that your anchors are correct. Use PhysicsSimulatorView to find them (right now this means XNA only).
2. Don't change properties dramatically. Adjust them slowly and recheck your simulation.
3. Anchors don't have to be inside a bodies attached geometry. Don't let your geometry limit your ability to overcome problems. Remember games are 50% fake. Do what it takes to make the end result 'look' right.

Developer note:

All joints are derived from the base class Joint and if you have the knowhow to add your own constraint be sure to derive yours from Joint as well (Please also share it with the community).

Shared variables – (these apply to all the joints)

- **Enabled** – Simply lets the engine know whether to enforce the constraint or not.
- **IsDisposed** – Lets you and the PhysicsSimulator know if the joint has been disposed of.
- **JointError** – Get the error of the joint. Note that not all joints produce an error.
- **Breakpoint** – Defines the maximum allowed value that JointError can reach before the joint is broken. The default is unbreakable (highest floating point number possible).
- **Softness** – This coefficient used to allow joint softness. It should be between 0.0f and 1.0f, everything else is undefined. This is really just used to adjust the simulation to your liking. Something to note is that all joints will have some softness even with softness set to 0.0f.
- **BiasFactor** – This coefficient determines how strongly the error will be corrected. It should be between 0.0f and 1.0f, everything else is undefined. Defaults to 0.2f. Note that setting this too high or low can cause dramatic instability, so change this in small increments.

- **Broke** – This event handler fires when a joint breaks. It is up to you to provide a method here otherwise nothing will happen.
- **Tag** – A generic object you can set to anything you like.

Shared Methods – (these apply to all the joints)

- **Validate** – Determines if all bodies involved are not disposed. If any are then the joint is disposed.
- **PreStep** – Performs the math necessary to update the joint. Should only be used if you are writing your own joint.
- **Update** – Performs the math necessary to update the joint. Should only be used if you are writing your own joint.
- **Dispose** – Disposes the joint.

Revolute Joint

This type of joint is great for adding wheels or linking multiple objects together. This joint takes a world vector but stores the joint's position as two local vectors internally. The joint error is determined by finding the distance between the two local vectors. This joint comes in body to body and body to fixed types. Something to note is that when using the body to body type, the bodies can still move linearly, and when using the body to fixed type, the body can only rotate.

Properties

- **Body1** – Get/set the first body of the joint. This property is named Body in Fixed Revolute joints.
- **Body2** – Get/set the second body of the joint. This property doesn't exist in Fixed Revolute joints.
- **Anchor** – Get/set the anchor of the joint. This is in world coordinates.
- **CurrentAnchor** – Gets the world position of the joint after simulation has started. Useful for drawing the joint.
- **SetInitialAnchor** – Sets the anchor before the simulation has started.

||Demo1||

Demo description:

The yellow rectangle is pinned in place with a revolute joint. It can't move, only rotate. Use the mouse to rotate the rectangle. A fixed revolute joint is used to achieve this.

Next to the rectangle, you will see a green rectangle pinned to a red rectangle. They can move around in the world, but they can't move relative to each other, only rotate. A normal revolute joint is used to achieve this.

Angle Joint

This joint works great for linking two bodies' angles to the same value or to a target angle. The fixed version simply holds a body at a target angle. These work great for programmatically changing a body's angle without causing it to snap to that angle. If you change the angle the body or bodies will respond quickly to achieve that angle.

Properties

- **TargetAngle** – Get/set the angle in radians that the body/bodies will attempt to achieve.
- **MaxImpulse** – Get/set the maximum torque the body/bodies will use in attempting to achieve the TargetAngle. Defaults to the highest floating point number possible.
- **Body1** – Get/set the first body of the joint. This property is named Body in Fixed Angle joints.
- **Body2** – Get/set the second body of the joint. This property doesn't exist in Fixed Angle joints.

|| Demo2 ||

Demo description:

To the left you see a green rectangle that is kept at a specific angle all the time. It can't rotate, only move. A fixed angle joint is used to achieve this.

To the right you see 2 yellow rectangles that share the same angle/rotation. If you rotate one of them, the other one will rotate with the same amount. A normal angle joint is used to achieve this.

Angle Limit Joint

Same as an Angle joint but with limits, no target angle, and no torque impulses.

Properties

- **LowerLimit** – Get/set the minimum angle, in radians, of the body.
- **UpperLimit** – Get/set the maximum angle, in radians, of the body.
- **Slop** – Get/set the slop allowed in the past the Min/Max limits.
- **Body1** – Get/set the first body of the joint. This property is named Body in Fixed Angle Limit joints.

- **Body2** – Get/set the second body of the joint. This property doesn't exist in Fixed Angle Limit joints.

||Demo3||

Demo description:

The green rectangle to the left is kept rotated within a defined limit at all time. Minimum 15 degrees and maximum 50 degrees. A fixed angle limit joint is used to achieve this.

The two yellow rectangles are at a always within a specified angle limit relative to each other. Try putting both of them on the ground and you will see that it's impossible to make both stand firmly on the ground. Minimum 15 degrees and maximum 50 degrees. A normal angle limit joint is used to achieve this.

Pin Joint

This joint should probably be called a rod joint. It will hold two bodies a set distance apart while still allowing them to rotate.

Properties

- **TargetDistance** – Get/set the desired distance between the two anchors. If no distance is specified, the offset between the bodies will be the target distance.
- **Body1** – Get/set the first body of the joint. Changing this without recalculating the TargetDistance could lead to instability.
- **Body2** - Get/set the second body of the joint. Changing this without recalculating the TargetDistance could lead to instability.
- **Anchor1** – Get/set the local anchor for Body1.
- **Anchor2** – Get/set the local anchor for Body2.
- **WorldAnchor1** – Get the world anchor for Body1. Useful for drawing the joint.
- **WorldAnchor2** – Get the world anchor for Body2. Useful for drawing the joint.

||Demo4||

Demo description:

The two red rectangles are using a pin joint to keep a target distance between each other. They can't move further away or closer to each other. Both bodies are still allowed to rotate.

The two yellow rectangles are using a slider joint (seen below) to keep a minimum and maximum distance between each other. They can move further away and closer to each other compared to pin joint.

Slider Joint

This joint is just like a Pin joint but with min/max limits on the distance. It will hold two bodies a set minimum to maximum distance apart while still allowing them to rotate.

Properties

- **TargetDistance** – Get/set the desired distance between the two anchors.
- **Body1** – Get/set the first body of the joint. Changing this without recalculating the TargetDistance could lead to instability.
- **Body2** – Get/set the second body of the joint. Changing this without recalculating the TargetDistance could lead to instability.
- **Anchor1** – Get/set the local anchor for Body1.
- **Anchor2** – Get/set the local anchor for Body2.
- **WorldAnchor1** – Get the world anchor for Body1. Useful for drawing the joint.
- **WorldAnchor2** – Get the world anchor for Body2. Useful for drawing the joint.
- **Min** – Get/set the minimum distance the anchors can come together.
- **Max** – Get/set the maximum distance the anchors can separate from each other.
- **Slop** – Get/set the slop allowed in the past the Min/Max limits.

Joint Factories

Joint Factories allow you to create joints in much the same way as you can create a body with a factory. As all the parameters have been described for the various joints I will just list the various methods used. Note that the factories don't always set all the needed parameters of the joint.

Revolute Joint Factory

1. CreateRevoluteJoint([PhysicsSimulator](#) physicsSimulator, [Body](#) body1, [Body](#) body2, [Vector2](#) initialAnchorPosition)
2. CreateRevoluteJoint([Body](#) body1, [Body](#) body2, [Vector2](#) initialAnchorPosition)

Fixed Revolute Joint Factory

1. CreateFixedRevoluteJoint([PhysicsSimulator](#) physicsSimulator, [Body](#) body, [Vector2](#) anchor)
2. CreateFixedRevoluteJoint([Body](#) body, [Vector2](#) anchor)

Pin Joint Factory

1. CreatePinJoint([PhysicsSimulator](#) physicsSimulator, [Body](#) body1, [Vector2](#) anchor1, [Body](#) body2, [Vector2](#) anchor2)
2. CreatePinJoint([Body](#) body1, [Vector2](#) anchor1, [Body](#) body2, [Vector2](#) anchor2)

Slider Joint Factory

1. CreateSliderJoint(PhysicsSimulator physicsSimulator, Body body1, Vector2 anchor1, Body body2, Vector2 anchor2, float min, float max)
2. CreateSliderJoint(Body body1, Vector2 anchor1, Body body2, Vector2 anchor2, float min, float max)

Angle Joint Factory

1. CreateAngleJoint(PhysicsSimulator physicsSimulator, Body body1, Body body2)
2. CreateAngleJoint(Body body1, Body body2)
3. CreateAngleJoint(PhysicsSimulator physicsSimulator, Body body1, Body body2, float softness, float biasFactor)
4. CreateAngleJoint(Body body1, Body body2, float softness, float biasFactor)

Fixed Angle Joint Factory

1. CreateFixedAngleJoint(PhysicsSimulator physicsSimulator, Body body)
2. CreateFixedAngleJoint(Body body)

Angle Limit Joint Factory

1. CreateAngleLimitJoint (PhysicsSimulator physicsSimulator, Body body1, Body body2, float min, float max)
2. CreateAngleLimitJoint (Body body1, Body body2, float min, float max)

Fixed Angle Limit Joint Factory

1. CreateFixedAngleLimitJoint (PhysicsSimulator physicsSimulator, Body body, float min, float max)
2. CreateFixedAngleLimitJoint (Body body, float min, float max)

Springs

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

- Linear spring*
- Angle spring*

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

Important Notes for all springs

- Some springs need anchors relative to the bodies' position and some need world anchor points. So pay attention to the type of spring your using.
- Springs are attached to bodies.
- All springs share some variables and methods.
- Anchors don't have to be inside a bodies attached geometry.
- When springs break, they are disabled.

If springs don't work as expected

- Check that your anchors are correct. Use PhysicsSimulatorView to find them (right now this means XNA only).
- Don't change properties dramatically. Adjust them slowly and recheck your simulation.
- Anchors don't have to be inside a bodies attached geometry. Don't let your geometry limit your ability to overcome problems. Remember games are 50% fake. Do what it takes to make the end result 'look' right.

Developer note:

All springs are derived from the base class Spring and if you have the knowhow to add your own constraint be sure to derive yours from Spring as well (Please also share it with the community).

Shared variables – (these apply to all the springs)

- **Enabled** – Simply lets the engine know whether to enforce the constraint or not.
- **IsDisposed** – Lets you and the PhysicsSimulator know if the spring has been disposed of.
- **DampningConstant** – Get/set the dampening of the spring. This acts much like a shock absorber.
- **SpringConstant** – Get/set the pull/push of the spring. Could be considered the force of the spring.
- **SpringError** – Get the error of the spring. Note that not all springs produce an error.
- **Breakpoint** – Defines the maximum allowed value that SpringError can reach before the spring is broken. The default is unbreakable (highest floating point number possible).
- **Broke** – This event handler fires when a spring breaks. It is up to you to provide a method here otherwise nothing will happen.
- **Tag** – A generic object you can set to anything you like.

Shared Methods – (these apply to all the joints)

- **Validate** – Ensures the spring's body/bodies still exist. If any are disposed then the spring is disposed also.
- **Update** – Performs the math necessary to update the spring. Should only be used of your writing your own spring.

- **Dispose** – Disposes the spring.

Linear Spring

This spring provides a pull/push of body/bodies. These can push as well as pull and can work in all kinds of ways. When combined with the right joints anything can be simulated.

Properties

- **Body1** – Get/set the first body of the joint. This property is named Body in Fixed Linear springs.
- **Body2** - Get/set the second body of the joint. This property doesn't exist in Fixed Linear springs.
- **Anchor1** – Get/set the local anchor for Body1. This property is named Body in Fixed Linear springs.
- **Anchor2** – Get/set the local anchor for Body2. This property doesn't exist in Fixed Linear springs.
- **RestLength** – This is the length the spring will be when it's not pulling or pushing at all. Note when using a factory the rest length is calculated as the distance between the anchors. If you want a spring to start pulling or pushing as soon as it's created you should scale this property. A bigger scale value will cause the spring to push the body/bodies apart, smaller scale value will cause the body/bodies to pull together.

||Demo5||

Demo description:

The two yellow bodies are kept together with a linear spring. The bodies can be moved apart with enough force. It's like they are joined with a rubber band. A normal linear spring is used in this demo.

All the demos in this manual use a fixed linear spring to move/apply force to bodies. It's illustrated with a black line.

Angle Spring

This spring acts like a torsion bar.

Properties

- **Body1** – Get/set the first body of the joint. This property is named Body in Fixed Angle springs.

- **Body2** - Get/set the second body of the joint. This property doesn't exist in Fixed Angle springs.
- **Anchor1** – Get/set the local anchor for Body1. This property is named Body in Angle Linear springs.
- **Anchor2** – Get/set the local anchor for Body2. This property doesn't exist in Fixed Angle springs.
- **TargetAngle** – This is the angle in radians that the spring will be centered and offers no force.
- **MaxTorque** – This is the maximum torque that will be applied to attempt to reach the TargetAngle.
- **TorqueMultiplier** – The torque is multiplied by this value. Use 1.0f to use normal torque. This defaults to 1.0f.

||Demo6||

Demo description:

The white element in the middle functions just like a springboard. The angle spring is of fixed type. This means that it's attached to the "world". A revolute joint is used to pin the spring board to the world so that it stays in the same position, but still allow it to rotate around the pivot point.

Spring Factories

Spring factories allow you to create springs in much the same way as you can create a body with a factory. As all the parameters have been described for the various springs I will just list the various methods used. Note that the factories don't always set all the needed parameters of the spring.

Linear Spring Factory

- CreateLinearSpring(PhysicsSimulator physicsSimulator, Body body1, Vector2 anchor1, Body body2, Vector2 anchor2, float springConstant, float dampningConstant)
- CreateLinearSpring(Body body1, Vector2 anchor1, Body body2, Vector2 anchor2, float springConstant, float dampningConstant)

Fixed Linear Spring Factory

- CreateFixedLinearSpring(PhysicsSimulator physicsSimulator, Body body, Vector2 anchor1, Vector2 anchor2, float springConstant, float dampningConstant)
- CreateFixedLinearSpring(Body body, Vector2 anchor1, Vector2 anchor2, float springConstant, float dampningConstant)

Angle Spring Factory

- CreateAngleSpring(PhysicsSimulator physicsSimulator, Body body1, Body body2, float springConstant, float dampningConstant)
- CreateAngleSpring(Body body1, Body body2, float springConstant, float dampningConstant)

Fixed Angle Spring Factory

- CreateFixedAngleSpring(PhysicsSimulator physicsSimulator, Body body, float springConstant, float dampningConstant)
- CreateFixedAngleSpring(Body body, float springConstant, float dampningConstant)

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 around the screen a lot, 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 and cause unreliable collisions.

More information on SAP can be found [here](#) and [here](#) (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 built on Sweep And Prune, but had some changes that made it perform better than SAP.

More information on SS can be found [here](#).

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 compares 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
 - a) If false: Continue to next vector in vertices list
 - b) 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

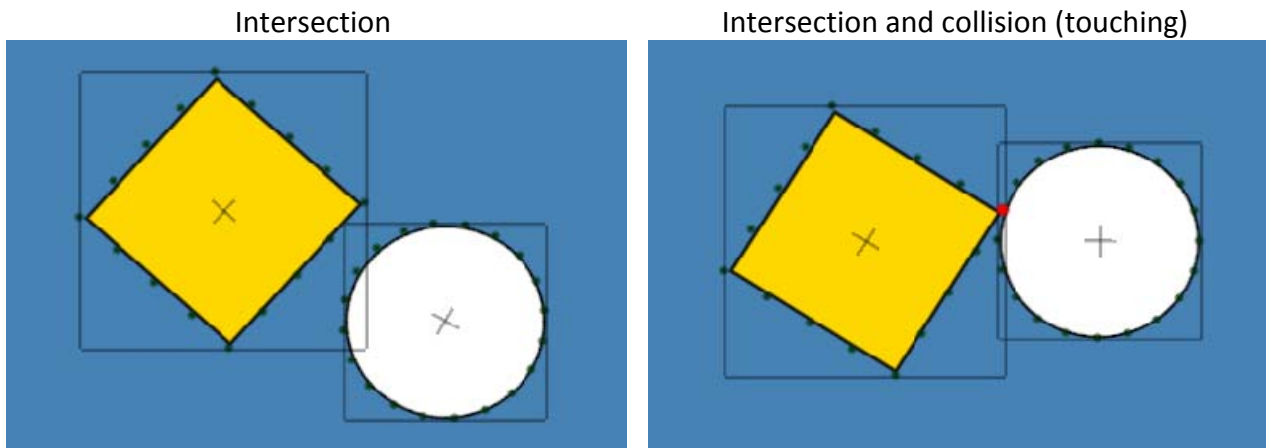
AABB stands for Axis Aligned Bounding Box and as the name says, it's a bounding box that aligns itself to an axis. All geometries have an AABB that is recalculated on each update, AABB are relatively inexpensive and used to quickly test if 2 geometries are close to each other (or even touching).

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.

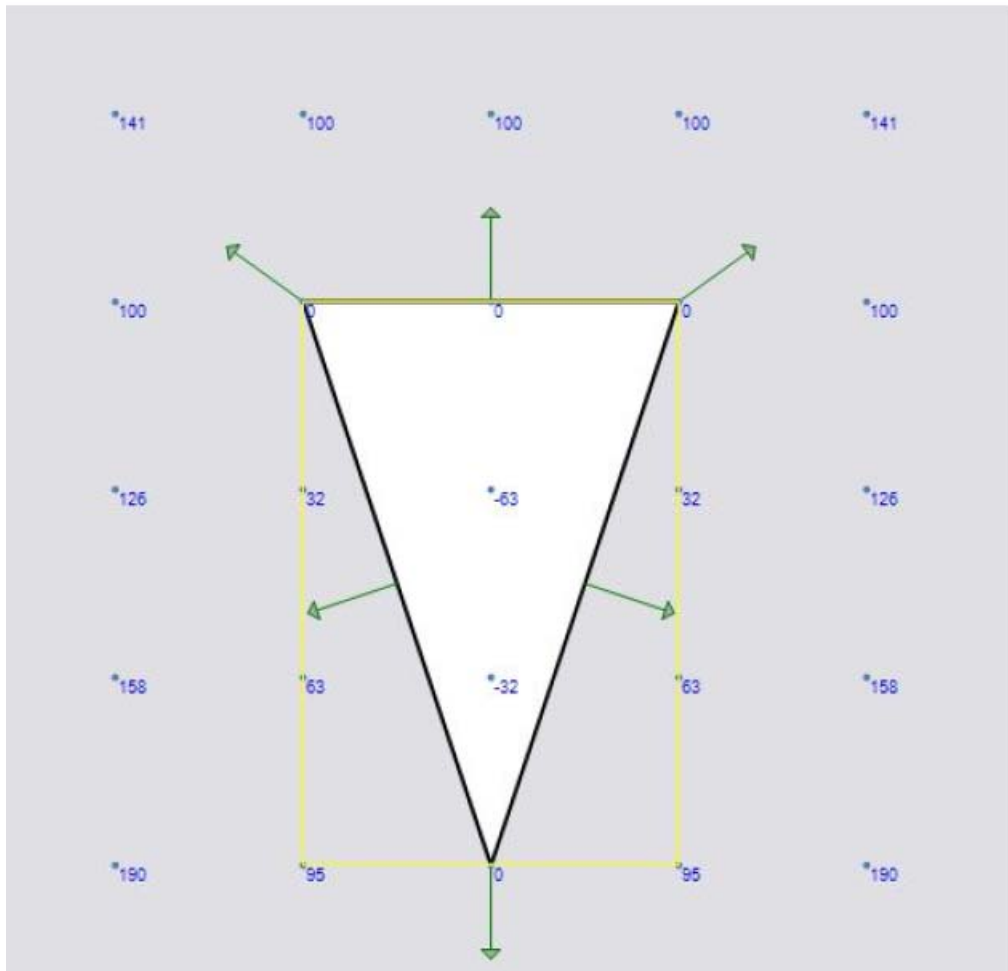


The AABB is the black outline of the geometries. As you see, they are not rotated, but are axis aligned. If you look real close, you can see that the contact created when colliding is red.

Grid

All geometries contains a Grid object. It's used by the narrow phase collision detection and uses a "distance grid". A distance grid is a pre-computed grid where each grid point contains the distance to the closest point on the geometry and the normal at that point. (For more information on normals, see the Physics chapter)

All distances for grid points that fall inside the geometry are negated. A picture of the grid and it's points is shown below:



Once the distance grid is pre-computed, the distance and normal for ANY point within the grid can be computed by interpolating between the known grid point values.

The grid is calculated from the provided grid cell size in the Geom constructor or when using the GeomFactory. A small grid cell size makes the grid more precise and therefore also makes the collision detection more precise.

Calculation of the grid can be quite time consuming, so it might be a good idea to pre-instantiate all your geometries beforehand and choose a grid cell size that is perfect for the geometry. More on this in the "Performance" chapter.

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, that's why **collision categories** also exist.

There are two properties of interest when using collision categories:

1) CollisionCategories

- a) Defaults to CollisionCategory.All
- b) Used to define with categories the geometry is a member of.

2) CollidesWith

- a) Defaults to CollisionCategory.All
- b) Used to define with categories the geometry collides with.

Collision categories use an enum called CollisionCategory that has a special flag enabled on it, so it's able to do bitwise operations. (more info [here](#))

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:

1. OnCollision (in Geom class)
2. OnSeparation (in Geom class)
3. 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

There are 2 systems for impulses:

1. Collision response
2. Manual impulses

The **collision response** is what happens when 2 geometries collide with each other. The `Arbiter` class that was described in the "Narrow phase" chapter is responsible for the calculations of impulses when a collision happens.

If we get a little more technical, what actually happens is that the contacts calculated in the narrow phase collision detection gets an impulse applied so that the geometries behave like real physics.

The collision response can be deactivated by setting the geometries **`CollisionResponseEnabled`** to false, like this:

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

Disabling the collision response means that it will pass through all other geometries. It will still fire the collision events described in the "Collision Events" Chapter.

You can also apply **manual impulses** to a body. (remember that it's the body that controls dynamics and the geometry that controls collision, but arbiter controls geometry impulses that are related to collisions.) You can apply 3 kinds of forces/impulses to a body. The forces and their methods are listed below:

1. Force

1. ApplyForce
2. ApplyForceAtLocalPoint
3. ApplyForceAtWorldPoint
4. ClearForce

2. Impulse

1. ApplyImpulse
2. ClearImpulse
3. ApplyAngularImpulse

3. Torque

1. ApplyTorque
2. ClearTorque

Force is used to accelerate a body. You can apply it to the whole body or at a specific point on the body. You can use this to add a jetpack on your character as the jetpack accelerates over time.

Impulse is used to make an impulse on a body. Impulse updates the body's velocity instead of accelerating the body like force.

You can use this to make your game character jump. A jump is an instant change in moment and does not accelerate.

Torque is used to apply torque (rotation) to your body. You can make wheels turn or make boulders run up hill.

Physics Properties

Farseer has a great interface for changing the physics properties of bodies and geometries. Most properties can be changed on the fly while the physics engine is running, this gives the possibility of creating very dynamic behaviors.

Here is a list of the physics properties and a short description of what they do:

Inside Body class:

AngularVelocity

Angular velocity is the rate at which a body is rotating. This is measured in radians per second. The larger the rate, the faster the body is rotating.

LinearVelocity

Velocity is defined as the rate of change of position. It can also be defined as the displacement of a body in unit time. This is a vector which means that it's not only tells you the amount of change, but also the direction.

LinearDragCoefficient

Drag is the force that resists the movement of a body through a fluid or gas (air). If you have a body moving fast through the air, it will gradually slow down due to drag. In Farseer Physics we don't have a medium (fluid or gas) that the body can move in, so you will have to manually set the drag coefficient of the body. The higher drag coefficient, the more force is needed to move the body and it will slow down faster.

RotationalDragCoefficient

Just as linear drag coefficient, there are also some drag when rotating. If you rotate a body with a rotational drag coefficient of 0, it will spin forever. The higher the rotational drag coefficient, the faster the rotation of the body will slow down.

Moment of Inertia (MOI)

The moment of inertia of a body in 2D is a scalar value that represents how difficult (or not difficult) it is to rotate a body about the center of mass.

Inside Geom class:

RestitutionCoefficient

Restitution coefficient is the ratio between velocities before and after an impact. If you set a restitution coefficient of 1, it will create a perfect bounce (image a ball that impacts with the ground) and if you set it to 0, it will not bounce at all.

FrictionCoefficient

Friction is essentially a force that opposes the relative motion of two material surfaces in contact with one another. The larger friction of a material, the harder it is to move relative to the other material, it's in contact with.

An example would be ice on steel, they have a very small coefficient of friction. They will slide right off each other. Rubber on pavement on the other hand, has a very high coefficient of friction, and does not slide very well.

A thing to note is that Farseer Physics has 2 different ways of handling friction. You can set the `FrictionType` on the `PhysicsSimulator` object to have one of the 2 settings:

- Average
 - If one of the geometries (materials) have a friction of 5 and the other have a friction of 3, the average coefficient of friction would be 4.
- Minimum
 - It picks the smallest friction from one of the two geometries (materials). If the 2 geometries have a friction of 5 and 3, the minimum would be 3.

Farseer Physics defaults to average friction type.

Extensions to Farseer

Since Farseer Physics Engine 2.0, we have included some advance methods to help doing some common operations. The extensions are only higher level functions that do not change the behavior of the physics engine.

Texture to vertices

In Farseer Physics Engine 2.0, we included an algorithm that searches a `uint[]` array for outstanding polygons, and then map their position. This should make it easier for people to make geometries from home made textures.

To use the texture to vertices algorithm, you just do the following:

```
//Load texture that will represent the physics body
Texture2D polygonTexture =
ScreenManager.ContentManager.Load<Texture2D>( "Content/Texture" );

//Create an array to hold the data from the texture
uint[] data = new uint[polygonTexture.Width * polygonTexture.Height];

//Transfer the texture data to the array
polygonTexture.GetData(data);

//Calculate the vertices from the array
Vertices verts = Vertices.CreatePolygon(data, polygonTexture.Width,
polygonTexture.Height);

//Make sure that the origin of the texture is the centroid (real center of
geometry)
Vector2 polygonOrigin = verts.GetCentroid();

//Use the body factory to create the physics body
Body polygonBody = BodyFactory.Instance.CreatePolygonBody(PhysicsSimulator,
verts, 5);
polygonBody.Position = new Vector2(500, 400);
```

```
GeomFactory.Instance.CreatePolygonGeom(PhysicsSimulator, polygonBody, verts, 0);
```

Path Generator

In Farseer Physics Engine 2.0, we included something called a Path Generator. What it does, is that it can create a bodies along a curve, and link the bodies together with a joint or spring. This can be useful in a lot of ways:

- Creating tank tracks
- Rope bridges
- Chains

Just to mention a few.

The Path Generator lives in the Path class and it's used in the ComplexFactory to create chains and rope. By using the Path Generator, only the imagination limits the creation of linked bodies.

You can also ask the Path Generator to create geometries for all your bodies. This enables collision with other objects.

Performance

Performance is really important in many application types, but you should never optimize before the end of you development. If you as an example introduce multithreading into your application in the beginning of your development, you will have a lot of headache from synchronization, locking and race conditions.

Writing clean code and relying on the compiler to do its job is the most important factor. A clean design is so much more important than a thousand micro optimizations.

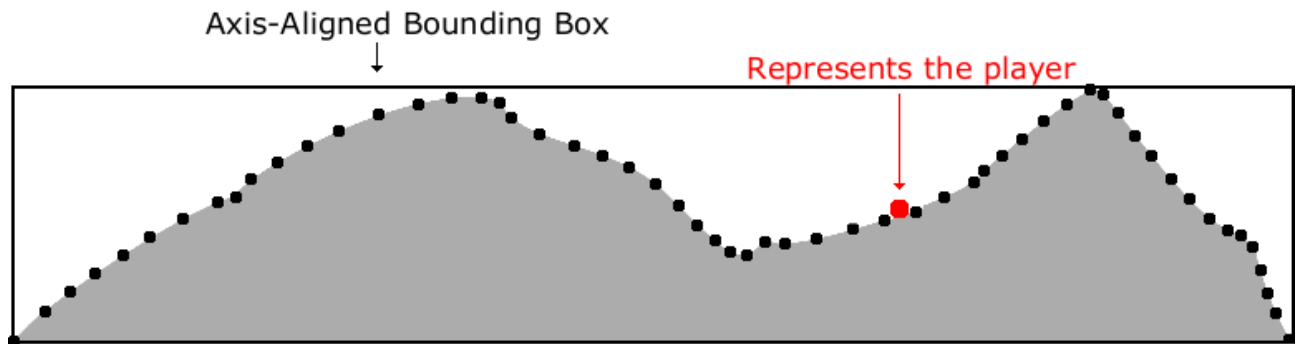
Here are some tips and tricks to improve your application performance:

1. Chunk up the landscape

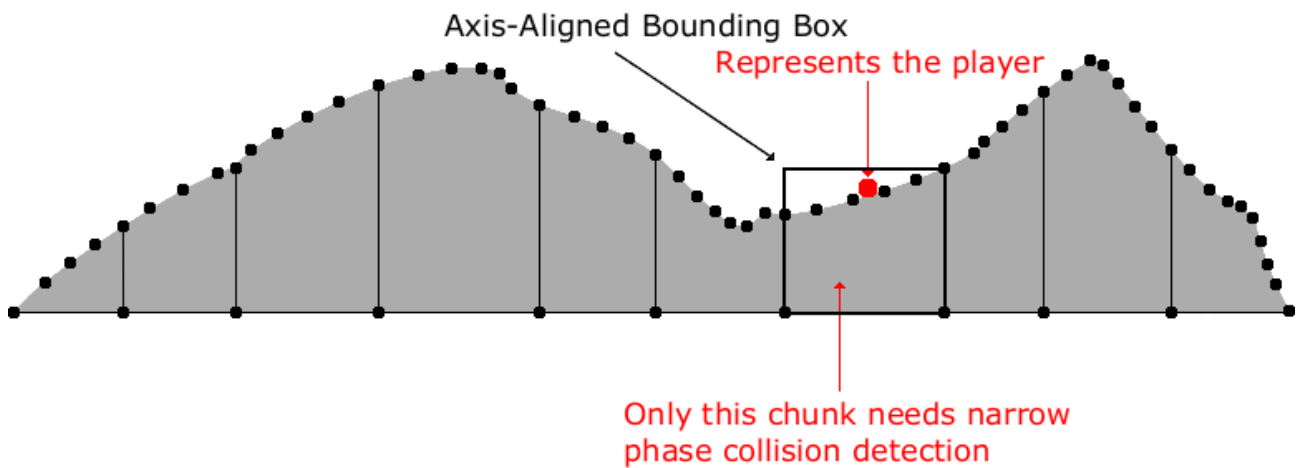
Because the broad phase collider uses AABB's to check for collisions, a large landscape in one piece would cause the broad phase to mark the whole landscape for narrow phase collisions all the time. This can cause a big performance degradation.

The solution is to create your landscape in chunks, so that only the current area the player is standing on, is checked for narrow phase collisions.

Before:



After:



2. Simplify the geometry

As you can see in the landscape image above, the points defining the curves on the top, needs to be there to get all details of the outline. The more points, the greater details. This is a potential problem for performance. The lesser points Farseer Physics need to calculate on, the higher performance, so keeping the amount of points low while still keeping the details of the outline, is the right way to go.

Another thing to note, is that the landscape above does not need a lot of points at the bottom as the player will never touch it. So you do not need to place any extra points at the bottom.

3. Keeping the grid cell size large

As described in the "Grid" chapter, grid's use a grid cell size to determine the precision of the collision detection. A small grid cell size means more precise collision detection, but a small grid cell size also takes a long time to compute.

You can manually find the right grid cell size by first passing in the default size (1/10th the smallest dimension of your geometry's AABB. More info in the "Known issues" chapter under "Geometries going into each other") and then step it up until collisions become unreliable.

Finding the optimal grid cell size can give better performance, but might also give unpredictable collision problems. Remember to do this, only when you have finished developing your application.

4. Minimal number of bodies/geometries

The easiest and most logical way of gaining more performance is to minimize the number of bodies and geometries active at one time. If you have a large level and it takes the player a long time to get to another section of the level, you could keep that part of the level deactivated until the player arrives.

This is really easy in some games. You could for example place a sensor (note: setting `IsSensor` to true on geometry) at certain places on the level, and when the player reaches that sensor, the next part of the level will be activated.

There is a lot of ways of doing this, it all depends on the type of game you are developing.

5. Caching

Another very easy implementation is caching rapid-spawning objects. If you are spawning a lot of enemies or bullets, you can pre-create the bodies and geometries that make up the enemies/bullets.

Farseer Physics uses a pool (cache) for Arbiters, it speeds up the creation a lot. This pool is actually public, so you can use the generic Pool class from Farseer to cache your objects.

To create a pool of 10 soldiers, you could do something like this:

```
Pool<Enemy> pool = new Pool<Enemy>();
for (int i = 0; i < 100; i++)
{
    Enemy enemy = new Enemy(EnemyType.Soldier, Health.100);
    pool.Insert(enemy);
}
```

And when you need the soldier in your game:

```
Enemy enemy = pool.Fetch();
enemy.Shoot();
```

There are 2 reasons why you would want a pool like this, one of them is the garbage collector, and the other is pre-instantiation.

The **garbage collector** cleans up after you, but this also means that if you create an enemy and run `Dispose()` on him when he dies, the garbage collector will remove him from your system memory.

But if you have the enemy inside the Pool, you don't need to call `Dispose()` on him, you just have to deactivate him (and not draw him). This results in fewer garbage collections.

The **pre-instantiation** is also a very good thing since Farser Physics uses what's called a distance grid. This grid is calculated when you create a new geometry and it can be quite time consuming.

So creating the pool of enemies when the game starts (or a new level loads) speeds up the creation of enemies a lot.

6. Remember to use release compilation.

Whenever you release your application to the public, be sure to compile your application with release settings. The .net platform works by first compiling your C#/Vb.net code into IL (Intermediate Code) code and then the JIT (Just-In-Time compiler) compiles your code into native machine code, when you execute your application.

When you turn on release configuration and compile your application, the IL code generated states that when the JIT runs the IL, it should perform optimizations.

This can speed up your application a lot and might reduce the overall size of your application.

7. Pass vectors and matrix by reference

Vectors (`Vector2`, `Vector3`) and Matrix in XNA is what's called structs or value types.

Whenever you put a value type into a method as a parameter, it gets copied. If you have a large matrix or a lot of vectors, this can slow down your code.

So by passing the vectors or matrix by reference instead of value, might speed up your application a little. Farseer supports passing of value types in certain places, here is an example:

```
Body body = BodyFactory.Instance.CreateCircleBody(PhysicsSimulator, 64, 1);
Vector2 force = new Vector2(10,10);

for (int i = 0; i < 100; i++)
{
    body.ApplyForce(ref force);
}
```

```
}
```

In this code example, using the [ref](#) keyword, we save 100 copies of the "force" vector. Some places in your code might benefit a lot from this.

8. Multithreading

Multithreading can be quite a tricky thing and can be hard to accomplish correctly. Using multithreading might increase your game performance, and more often increase the number of concurrent elements on screen. The implementation details about this is out of scope for this manual, but Farseer Physics Engine 2.0 does include an example on multithreading. Have a look at the **Getting Started** sample (demo 4) for more details.

9. Inactivity controller

Farseer Physics Engine 2.0 includes a new thing called an inactivity controller. This controller enables what's called "resting bodies". This means that if your game contains a lot of elements that do not move around a lot, you can get some performance by deactivating them for the time being.

Inactivity controller does this for you. You only have to enable it in the physics simulator and set some basic settings. See the "Inactivity controller" chapter for more information.

10. Scaling

Scaling is yet another new feature of Farseer Physics Engine 2.0. If your game is doing some intensive operations and the frame rate drops below what is acceptable (usually 60 fps), you can make the engine slow down a bit and then accelerate it again later.

All you have to do is activate the Scaling controller like this:

```
PhysicsSimulator.Scaling.Enabled = true;
```

And then set the MaximumUpdateInterval you want specified. The preferred update interval is by default 0.001f and maximum update interval is by default 0.01f.

11. Circle-Circle optimization

This is one of the more rare occasions, but might be useful for someone.

More info here: [Circle - Circle optimization](#)

There are also many other ways of optimizing your code. We are not going into details about them, as they are not Farseer Physics specific, but here is a short list:

Note: Items in this list is classified as micro-optimization and should not be used, unless you have some really performance critical code. Have a look at [Understanding XNA Framework Performance](#) for more information. Farseer Physics Engine 2.0 is already optimized this way.

1. Inline performance critical methods

Before:

```
if (IsColorBlack(new Color(10, 4, 1)))  
{  
}
```

```
private bool IsColorBlack(Color color)  
{  
    return color == Color.Black;  
}
```

After:

```
if (new Color(10, 4, 1) == Color.Black)  
{  
}
```

2. Inline vectors instead of referencing

Before:

```
Vector2 distance = Vector2.Zero;  
Vector2.Subtract(ref GeometryB.Body.Position, ref GeometryA.Body.position,  
out distance);
```

After:

```
Vector2 distance = Vector2.Zero;  
distance.X = GeometryB.Body.Position.X - GeometryA.Body.position.X;  
distance.Y = GeometryB.Body.Position.Y - GeometryA.Body.position.Y;
```

3. Inline constructors

Before:

```
Vector2 distance = new Vector2(10,10);
```

After:

```
Vector2 distance = new Vector2();  
distance.X = 10;  
distance.Y = 10;
```

Known issues

There are some known issues that are related to Farseer Physics. Some of these issues are not only found in Farseer Physics, but also in many other physics engines. They are not easy to fix without sacrificing performance or usability.

1. Tunneling

This phenomenon occurs when you have a fast moving object that hits a wall and gets stuck inside it or even passes through the wall without even colliding.

Farseer Physics 2.0 does not currently have any good solution to this, but we have plans of implementing CCD (Continuous Collision Detection) that would prevent this from happening. Until then, these are your options:

- 1) Make your objects move slower
- 2) Make your objects larger
- 3) Decrease your time step
- 4) Use ray casting
- 5) Swept collision detection ([info here](#))
- 6) Multisampling

2. Geometries going into each other

This can be caused by 2 things: Your geometries have sharp points or too few points, even on straight edges.

If you have **sharp points**, you may need to use smaller grid cell size values, than the default. By default, if you pass 0 for the collisionGridCellSize to the CreatePolygonBody method, the method will calculate a collisionGridCellSize for you based on the size of the AABB.

If you have a geometry with sharp points you will probably want a smaller value than the default. You can either pass in your own non-zero value or you can adjust the default calculation by setting the `GeomFactory.GridCellSizeAABBFactor` property.

This property is used to calculate the default collisionGridCellSize. Currently it is set to .1 which means the collisionGridCellSize will be 1/10th the smallest dimension of your geometry's AABB.

The other thing you could try is inserting **more points** into the geometry. Farseer Physics has a method to help you do this. It's called `SubDivideEdges()` and lives inside the `Vertices` class.

Here is an example:

```

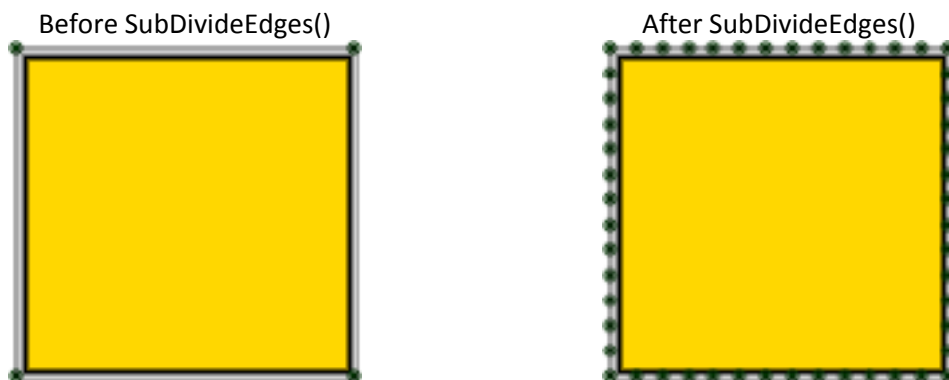
Body rectBody = BodyFactory.Instance.CreateRectangleBody(128, 128, 1);
Vertices vertices = new Vertices();
vertices.Add(new Vector2(-64, -64));
vertices.Add(new Vector2(64, -64));
vertices.Add(new Vector2(64, 64));
vertices.Add(new Vector2(-64, 64));

vertices.SubDivideEdges(10);

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

```

Note: Remember to add the body and geometry to the physics simulator.



3. Drawing is off center

Since Farseer Physics Engine 1.0.0.4, the vertices supplied to the GeomFactory's CreatePolygonGeom method gets centered on the centroid of the vertices.

You can work around it by following the steps described in [this thread](#) or use the `Vector2 offset` parameter in the CreatePolygonGeom method.