Jitter Managed Physics Engine[1]
Tutorial

1 Tutorial 1 - World creation and raycasting

1.1 The world class

The first step in setting up the engine in your game is to create a new instance of the World class:

```
1            CollisionSystem collisionSystem = new CollisionSystemSAP();
2            World world = new World(collisionSystem);
```

The world instance is created by passing a CollisionSystem class to the constructor. In this case we used CollisionSystemSap. SAP stands for 'Sweep and Prune' which is how the broadphase collision works. There is also a CollisionSystem called CollisionSystemBrute which just uses brute force for the broadphase system - which is pretty fast in smaller scenes. From now on the World instance uses the chosen CollisionSystem to detect collisions. It's possible (and also a good idea) to use Jitter for collision detection only. You just don't create the World class which handles the 'dynamics' and use the methods provides by the CollisionSystem.

1.2 The CollisionSystem

After adding the CollisionSystem to the World the CollisionSystem gets automatically updated. Adding/Removing a RigidBody to/from the World also adds/removes it to/from the CollisionSystem. The World class internally calls 'CollisionSystem.Detect' to get all collisions between all bodies. If you want to use Jitter just for collisions the following code detecting collisions between all bodies and reporting them back is interesting for you:

```
1        private DetectTest()
2        {
3            CollisionSystem collisionSystem = new CollisionSystemSAP();
4            collisionSystem.CollisionDetected += new
                 CollisionDetectedHandler(CollisionDetected);
5
6            // Here some bodies have to be added to the collisionSystem.
7
8            collisionSystem.Detect(true);
9        }
10
11       private void CollisionDetected(RigidBody body1, RigidBody body2,
             JVector point1, JVector point2, JVector normal, float
             penetration)
12       {
13           System.Diagnostics.Debug.WriteLine("Collision detected!");
14       }
```

The method 'CollisionDetected' is called when there is a collision. It's also possible to check just two bodies against each other:

```
1            collisionSystem.Detect(rigidBody1,rigidBody2);
```

If there is a listener of the 'CollisionDetected' event and there is a collision the event gets called providing you with several details about the collision. If you don't want to use this functionality you just don't use it. The world class does the collision detection internally.

[1]Copyright by Thorben Linneweber, 2010

An interesting feature of the engine - no matter if you use just the collision or also the dynamics - is to raycast the entire scene, containing all your bodies:

```
1         private DetectTest()
2         {
3                 RigidBody resBody;
4                 JVector hitNormal:
5                 float fraction;
6                 bool result;
7
8                 bool result = world.CollisionSystem.Raycast(JVector.One,
                      JVector.Right * 100.0f,RaycastCallback, out resBody,
                      out hitNormal, out fraction);
9
10                if(result) System.Diagnostics.Debug.Writeln("Collision
                      detected!");
11          }
12
13        private bool RaycastCallback(RigidBody body, JVector normal, float
              fraction)
14        {
15            return !body.IsStatic;
16        }
```

Raycasting means that a ray (a line with a start and an direction in 3dimensional space. here: (1,1,1) and (-100,0,0)) is 'shot' from it's start along it's direction. The ray 'hits' one of the bodies within the scene and the collision information is reported back. 'fraction' gives you the information where (at the ray) the collision occured:

```
1         JVector hitPoint = JVector.One + fraction * (JVector.Right * 100.0
              f);
```

The 'RaycastCallback' gives you the possibility to decide if a body should be considered as a collision object for the ray. In the sample above static bodies aren't considered. The ray goes right 'through' them. You can also pass 'null' as a parameter for the RaycastCallback - so every body is considered. Raycasting is one of the core functionalities of a physic engine and has many applications: consider for example, you want to implement a gun in your game and apply a force to the object the player hit.
Jitter also gives you the possibility to use core detection functions. These are the static classes 'XenoCollide' and 'GJKCollide'. You can use their Detect functions to check for collisions of shapes which are defined by their ISupportMap implementation. This is for more advanced users.

1.3 Adding bodies to the world

So far we just have created our World class which is connected to the CollisionSystem - our world is pretty empty and we want to add a single box to it:

```
1             CollisionSystem collisionSystem = new CollisionSystemSAP();
2             World world = new World(collisionSystem);
3
4             Shape shape = new BoxShape(JVector.One);
5             RigidBody body = new RigidBody(shape);
6
7             world.AddBody(body);
```

First a Shape is created. The shape represents the collidable part of the RigidBody. We wanted to have a box, so we used the BoxShape. There are much more default shapes in Jitter: Box, Cone, Sphere, Cylinder, Capsule, Compound, TriangleMesh. The shape is passed to the constructor of the body which gets added to the world. Done. Calculating mass, inertia.. is done inside the engine - but you also can set it manually. The body with the side length (1,1,1) is created at the origin (with RigidBody.Position you set the position of the center of mass of the simple shapes).

Units in JITTER: The default gravity which adds force to a body is set to 9.81 $\frac{m}{s^2}$ So, one length unit in JITTER is one meter. Because of numerical instabilities when using smaller objects it's recommended not to use objects which are much smaller than one unit.

The last thing to do, is to update the engine and to integrate the current physics state a timestep further:

```
1            world.Step(1.0f / 100.0f, true);
```

The first parameter is the timestep, the second one tells the engine to use internal multithreading or not. In our simple one-body world the box gets affected by gravity and the position changes. In your 'Draw' method you are now able to draw a box with the position and orientation of the body.

1.4 Complete sample

In Tutorial1 a complete physics scene is build and simulated using just a few lines of code.

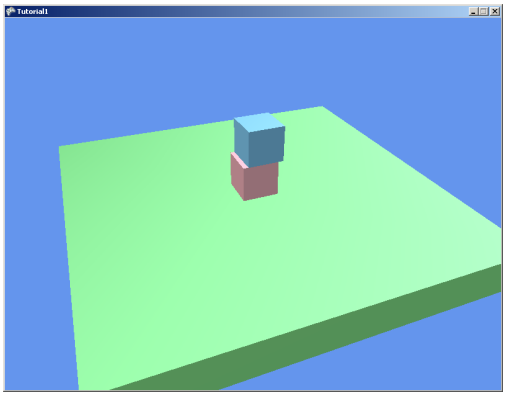

Figure 1: Tutorial1 - Two boxed falling to the ground. Clicking on them moves them away from the camera.

```
1         private void BuildScene()
2         {
3             // creating two boxShapes with different sizes
4             Shape boxShape = new BoxShape(JVector.One);
5             Shape groundShape = new BoxShape(new JVector(10, 1, 10));
6
7             // create new instances of the rigid body class and pass
8             // the boxShapes to them
9             RigidBody boxBody1 = new RigidBody(boxShape);
10            RigidBody boxBody2 = new RigidBody(boxShape);
11
12            RigidBody groundBody = new RigidBody(groundShape);
13
14            // set the position of the box size=(1,1,1)
15            // 2 and 5 units above the ground box size=(10,1,10)
16            boxBody1.Position = new JVector(0, 2, 0.0f);
17            boxBody2.Position = new JVector(0, 5, 0.3f);
18
19            // make the body static, so it can't be moved
20            groundBody.IsStatic = true;
21
22            // add the bodies to the world.
23            world.AddBody(boxBody1);
24            world.AddBody(boxBody2);
25            world.AddBody(groundBody);
26        }
```

The scene is drawn with the XNA framework, but it would be similar for every other graphics framework/engine. JITTER uses it's own math classes (JVector,JMatrix,JBox,JMath,JOctree), so you have to convert orientations (represented by 3x3Matrices) and positions to the type of your graphics framework - this isn't that complicated. For XNA the conversion method looks like this:

```
1         public static Matrix ToXNAMatrix(JMatrix matrix)
2         {
3             return new Matrix(matrix.M11,
4                               matrix.M12,
5                               matrix.M13,
6                               0.0f,
7                               matrix.M21,
8                               matrix.M22,
9                               matrix.M23,
10                              0.0f,
11                              matrix.M31,
12                              matrix.M32,
13                              matrix.M33,
14                              0.0f, 0.0f, 0.0f, 0.0f, 1.0f);
15        }
```

Building a XNA (4x4) world matrix to draw your object is easy:

```
1             Matrix matrix = Conversion.ToXNAMatrix(body.Orientation);
2             matrix.Translation = Conversion.ToXNAVector(body.Position);
```

2 Tutorial2 - Triangle Mesh

This tutorial shows how to create a body represented by custom triangles. A triangleMeshShape should be a static body - but it's also possible to make it dynamic - as shown in this tutorial. At first the vertices and indices have to be extracted from the model. This data is now used to build a JOctree which is passed to the TriangleMeshShape class. The octree makes it possible to use models containing several thousands of vertices. This is very useful because you can build your map/level out of a mesh and use it as a triangleMesh in the physics engine. Theoretically JITTER is able to handle triangle-triangle collision but it would really make your computer cry.

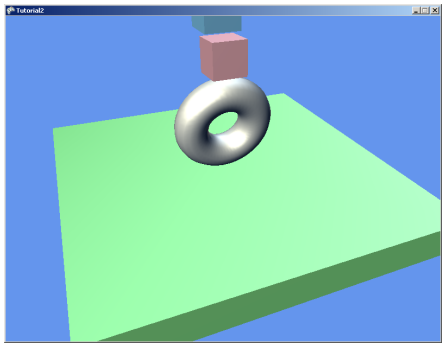

Figure 2: Tutorial2 - Two boxed falling to the ground. The torus is a triangleMeshShape.

```
1             // Build an octree of it
2             JOctree octree = new JOctree(positions, indices);
3             octree.BuildOctree();
4
5             // Pass it to a new instance of the triangleMeshShape
6             TriangleMeshShape triangleMeshShape = new TriangleMeshShape(
                  octree);
7
```

```
8               // Create a body, using the triangleMeshShape
9               RigidBody triangleBody = new RigidBody(triangleMeshShape);
10              triangleBody.Tag = Color.LightGray;
11              triangleBody.Position = new JVector(0, 3, 0);
12
13              // Add the mesh to the world.
14              world.AddBody(triangleBody);
```

3 Tutorial3 - Compound Body

In this tutorial we use the CompoundShape to add two 'sub'-boxShapes to it. You can use compound shapes to build new shapes out of the basic shapes. For example a car chassis could be made of a compound shapes. This is a common technique to save a lot of computation time.

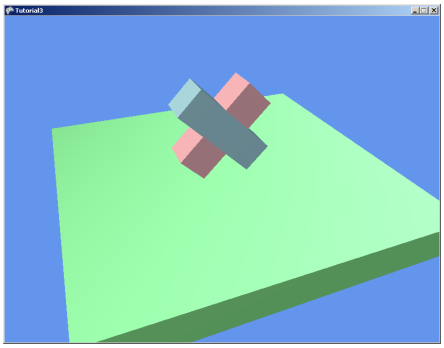

Figure 3: Tutorial3 - Two boxes form a compound compound body.

```
1               // Build the CompoundShape.TransformedShape structure,
2               // containing "normal" shapes and position/orientation
3               // information.
4               CompoundShape.TransformedShape[] transformedShapes =
5                   new CompoundShape.TransformedShape[2];
6
7               // Create a rotation matrix (90 )
8               JMatrix rotated =
9                   Conversion.ToJitterMatrix(Matrix.CreateRotationX(
                        MathHelper.PiOver2));
10
11              // the first "sub" shape. A rotatated boxShape.
12              transformedShapes[0] = new CompoundShape.TransformedShape(
13                  boxShape,rotated,JVector.Zero);
14
15              // the second "sub" shape.
16              transformedShapes[1] = new CompoundShape.TransformedShape(
17                  boxShape, JMatrix.Identity, JVector.Zero);
18
19              // Pass the CompoundShape.TransformedShape structure to the
                    compound shape.
20              CompoundShape compoundShape = new CompoundShape(
                    transformedShapes);
21
22              RigidBody compoundBody = new RigidBody(compoundShape);
```

4 Tutorial4 - Joints and Constraints

In this tutorial we connect the two boxes from tutorial1 with a PointConstraint:

```
1              // set the position of the box size=(1,1,1)
2              // 2 and 5 units above the ground box size=(10,1,10)
3              boxBody1.Position = new JVector(0, 4, 1.0f);
4              boxBody2.Position = new JVector(0, 4, -1.0f);
5
6              pointConstraint = new PointConstraint(
7                 boxBody1, boxBody2, new JVector(0, 4f, 0));
8
9              world.AddConstraint(pointConstraint);
```

Both bodies are given to the constructor. Also the point (in world space) where both bodies get fixed together. The constraint is then added to the world. More information about how which constraints work is included in the documentation.
In JITTER constraints are related to 'joints'. Joints are collections of several constraints. Joints don't need to be added to the world - they add and manage the constraints they own.

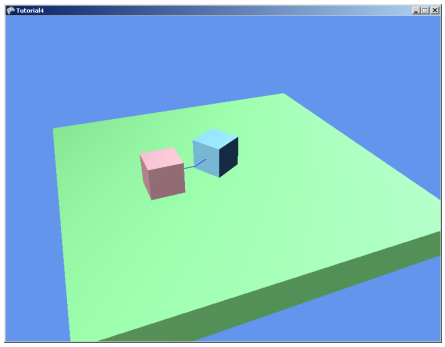

Figure 4: Tutorial4 - Two boxes constrained to a fixed point.

A helpful feature of JITTER is the possibility to get debug data from the constraints:

```
1              pointConstraint.AddToDebugDrawList(lineList, pointList);
```

lineList and pointList contain a list of JVectors. Every two points in lineList form a line.