DEVELOPMENT OF PHYSICS ENGINE

FOR 2.5D XNA GAME

CHOMPAN SRINGAM

TANAWAT PHADUNGKIETSAKUN

SUTEEPAT DAMRONGYINGSUPAB

A Special Project Submitted in Partial Fulfillment of the Requirements for

The Degree of Bachelor of Science

International Programs, Faculty of Science

King Mongkut's Institute of Technology Ladkrabang

Academic Year 2009

Special Project Title	Development of Physics Engine for 2.5D XNA Game		
Name	Chompan	Sringam	ID: 49050371
	Tanawat	Phadungkietsakun	ID: 49050372
	Suteepat	Damrongyingsupab	ID: 49050416
Faculty	Science		
Degree	Bachelor's Degree of Science		
Program	Computer Science (International Programme)		
Special Project Advisor	Assoc. Prof. Dr. Dr.Veera Boonjing		

Abstract

A purpose of this project is to develop Physics Engine for 2.5 dimensions games based on 2D Physics Engine called Farseer Physics Engine. Our Physics Engine is developed by using C# and XNA Framework. The 2.5 dimensions Physics Engine capabilities are force, impulse, collision, friction and restitution. We also develop sample scenes that use our 2.5 dimensions Physics Engine to observe the result of our Physics Engine. Finally, we provide the 2.5 dimensions Physics Engine with an API as a result.

Acknowledgement

This project cannot be success without the kindness from our advisor, Assoc. Prof. Dr. Veera Boonjing. He always gives us invaluable advice, attention and encouragement throughout this project.

Besides, it is the authors' pleasure to thank those who made this project possible. We are also heartily thankful to all professors who have taught us for the past four years at Department of Computer, Faculty of Science, King Mongkut's Institute of Technology Ladkrabang.

We also would like to thank our project committee, Asst. Prof. Dr. Sarun Intakosum and Asst. Prof. Dr. Jeeraporn Werapun for their direction, dedication and invaluable guidance along this project.

Lastly, we offer our regards and blessings to all of our friends and dearest families for their love and support in any respect during the completion of the project.

Miss Chompan Sringam ID 49050371

Mr. Tanawat Phadungkietsakun ID 49050372

Mr. Suteepat Damrongyingsupab ID 49050416

Table of Contents

Page

List of Figures

Page

Chapter 1

Introduction

1.1 Rationale

A Physics Engine is a program that simulates physics models, using variables such as mass, velocity, friction, and wind resistance to simulate and predict effects under different conditions that would approximate what happens in real life or in a fantasy world. It is used to make video game more realistic and enjoyable, by controlling objects movement in game from physics model. Like objects drop from high ground and bounce up, or collision between objects, projectile of objects – bullets, balls, cannonball and so on. While most games currently have a semblance of physics but do not actually use a complex Physics Engine, they just make common things react the way you would expect them to leave it alone from there. Games are lost credibility if their physics model is incorrect. On the other hand, if you get the physics right, your game will be praised by the game community. The Physics Engine is developed by many developers and is implemented into various computer languages, like C++, java, C#, .net. They also come with a lot features and well-developed API, in both 3D and 2D engine that help developers comfortable to implement and integrate Physics Engine into their game.

However the Physics Engines never have been used in 2.5D game before. The problem is in 2.5D games Physics Engine you have to handle 2D environments & 2D objects in 3D manner. In 2D game there only have two axes, x and y. So it should be rather easy. For 3D games they consider to be hard than 2D, but at least they already run on three dimensions environment. However, for 2.5D games, they only have two axes and 2D objects but they have to simulate like they were in 3D world. In order to adapt physical engine to 2.5D games, you have to be able to tell whether objects are near or faraway (depth of object); For example, look at picture below, without any clue to indicate depth of red ball in the picture, you cannot be sure that red ball is flying above square 1 or it is just sitting on square 2. (Figure 1.1)

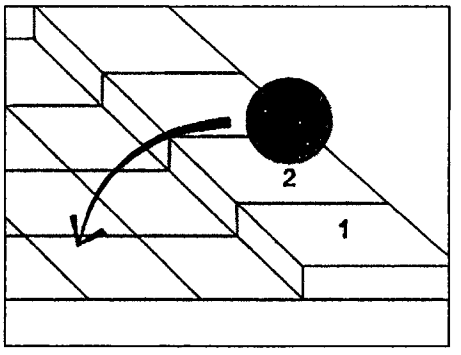

Figure 1.1 A red ball without shadow

Figure 1.1 shows that, without z index (depth), you cannot know position of a red ball, whether it is flying or it just stays still.

If you do not know current position of the object, then how can you calculate the object motion? So you need something to be reference of where the object is near or faraway. You can either setup z index that related to depth of object or attach shadow to the object, so you can tell object's position clearly. (Figure 1.2)

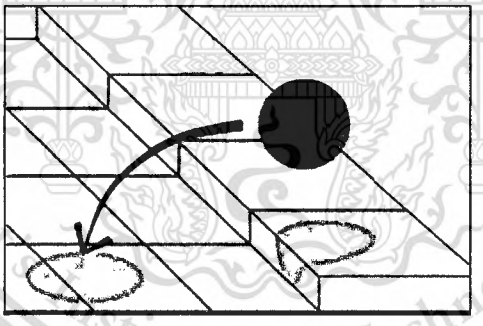

Figure 1.2 A red ball with shadow

With z index we can compute physical process to predict where the ball will fall on, even its background and objects are 2D but they interact like they were in 3D world.

In many of Physics Engine, we choose Farseer Physics Engine to be used in our project. Because it is open source program and develops from C#, and it has been designed for using in XNA and Silverlight. XNA is also a great tool to be tested, because it is easy to develop and it builds under OOP principle that we already have experience, and the fact that games are developed from XNA, so it can run on both XBOX360 and PC, which make XNA were an interesting choice.

At the end of the project, we test 2.5D Physics Engine and compare it with a 2D Physics Engine. We also estimate the efficiency and observe the output between our Physics Engine and the 2D Physics Engine.

1.2 Objective

Our objective is to develop Physics Engine for 2.5D XNA game by using 2D Farseer Physics Engine. Generally, the Physics Engine can perform collision detection and physics response, which can simulate force, impulse, torque, friction and restitution on rigid body.

1.3 Scope of Study

We develop Farseer Physics Engine for 2.5 dimensions games, compatible on both xbox360 and PC. Physics models implemented in 2.5 dimensions Physics Engine are including force, impulse, torque, kinetic and friction. We test our Physics Engine by comparing between our Physics Engine and traditional 2 dimensions Physics Engine. Besides, we also develop it to not be too complicate and focus on skimpiness, so that it can be easily used and understandable by other developers.

In order to develop Farseer Physics Engine to be used with 2.5D game, we have to understand the difference between 2 dimensions and 2.5 dimensions. First of all, in 2 dimensions games, all objects are simple, plain 2d images. So there have only 2 axes which are X- and Y-axis. Therefore, to calculation of physics model in 2 dimensions is forthright and only involves with X- and Y-axis. In contrast, 2.5 dimensions (pseudo-3D), all objects are just 2D images similar to 2 dimensions, but we have to simulate the plain 2d images like they are in 3 dimensions. Therefore, this does not concern with only X- and Y-axis but also the depth - or so called Z-index.

1.4 Organization of the special project

The following chapters of this special project consist of:

Chapter 1 describes the goals and scope of our project. Summary of project purposes, objective and benefits.

Chapter 2 is an explanation of related literature of Physics Engine. This chapter also provides references of information in order to understand and develop this project.

Chapter 3 covers the methods we have done to develop Physics Engine for 2.5D Game by using Farseer Physics Engine in order to make it works properly in 2.5 dimensions.

Chapter 4 shows the results of our 2.5D Physics Engine. We also create demo scenes for testing and comparing the results between original Farseer Physics Engine and our 2.5D Physics Engine.

Chapter 5 covers conclusion and recommendations of our project.

Chapter 2

Related literature

2.1 2.5 dimensions games

2.5D (two-and-a-half-dimension) is an informal term used to describe visual phenomena, which is actually 2D with 3D looking graphics. One such method is where a 2D image has an added "depth" channel or Z-buffer, which may act like a height map. The term is also used to describe 3D scenes built completely or partially from a composite of flat 2D images; and also where gameplay is restricted to a 2D plane while the display uses 3D graphics and 3D models. Base on our project, from now on the term 2.5D is referred as a 2D image, which has the depth or Z-buffer.

For example, in figure 2.1, first picture is an example of 2D game, which composes of 2D entities, and the character can walk only in 2 dimensions (x and y). A second picture is a 3D game, it objects and characters are 3D models place on 3D world, which players can move they character like they were in real world. But, for third picture, it is a 2.5D games, which characters, objects and environment are 2D image but they are simulate like they were in 3D world, like the character in the third picture can walk deep into field like they were on 3D field, but the field is actually a 2D image and character don't really walk into the field, the character just place on the top of flat 2D image.

Figure 2.1, 2.2 and 2.3 shows the example of 2D, 3D and 2.5D game, respectively.

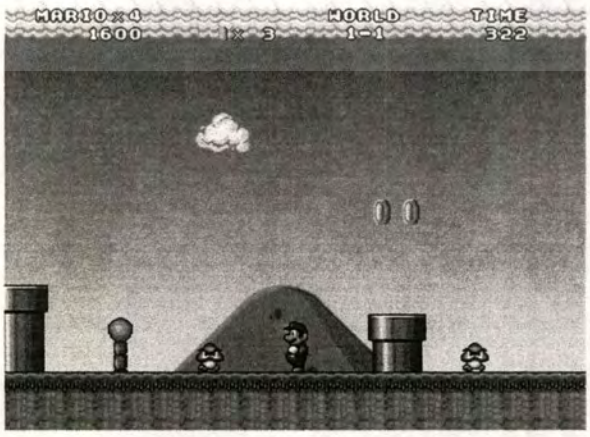

Figure 2.1 An example of 2D game

Figure 2.2 An example of 3D game

Figure 2.3 An example of 2.5D game

2.2 Physics Engine

A Physics Engine is a computer program that simulates physics models, using variables such as mass, velocity, friction, and wind resistance. It can simulate and predict effects under different conditions that would approximate what happens in real life or in a fantasy world. Its main uses are in scientific simulation and in video games.

There are two different kinds of Physics Engines. One of them is used for **soft body** (deformable) objects and the other is used for **rigid body** objects (hard/non-deformable). (Figure 2.4)

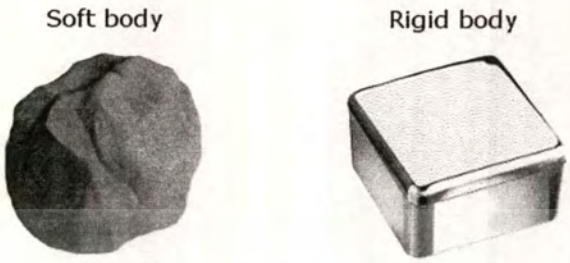

Figure 2.4 Soft body and Rigid body

Soft body physics

Soft body engines are used for deformable physics such as cloth, rubber, jelly or balloons. You can enlarge, rip or puncture a soft body in any way you want. There are both 3D soft body engines and 2D soft body engines. Most of them define their shapes using triangles (or particles) that can change properties (side lengths and angles) and thus deform the body that contains the triangle.

Soft body dynamics are hard to work with because of the deformations applied to a body. If a body needs to try and keep its original shape, all the deformations will have to be kept track off and slowly be resolved to gain its original shape.

Rigid body physics

Rigid body engines are used for non-deformable physics such as spacecrafts, buildings, terrain, player characters and more. They can't be deformed in any way and only support linear (movement) and angular (rotation) motion.

Rigid body engines are a lot easier to work with compared to soft body physics. You only have to keep track of movement on the x, y and z-axis and rotation around the centroid of the object.

High-precision VS. Real-time

There are generally two classes of Physics Engines, real-time and high-precision. High-precision Physics Engines require more processing power to calculate very precise physics and are usually used by scientists and computer animated movies. In video games, or other forms of interactive computing, the

Physics Engine simplifies its calculations and lowers its accuracy so that they can be performed in time for the game to respond at an appropriate rate for gameplay.

Physics video game engine in the past only used rigid body dynamics because they are faster and easier to calculate, but modern games are starting to use soft body physics now that it is possible. Soft body physics are also used for particle effects, liquids and cloth. Some form of limited Fluid dynamics simulation is sometimes provided to simulate water and other liquids as well as the flow of fire and explosions through the air.

Physics Engines can be used for so much more than just fun and games:

- Robot dynamics
- Cloth simulation
- Transport security
- Fluid Simulations
- Wind tunnel testing
- Much more…

2.3 Farseer Physics Engine Concepts

The Farseer Physics Engine is an open source, easy to use 2D Physics Engine designed for Microsoft's XNA and Silverlight platform. The Farseer Physics Engine is designed to control the position and rotation of game entities over time, by imply force, impulse, toque to game entities.

A Physics Engine mainly does two things: Collision detection and physics reaction. They work seamless together to create the illusion of things colliding and bouncing back.

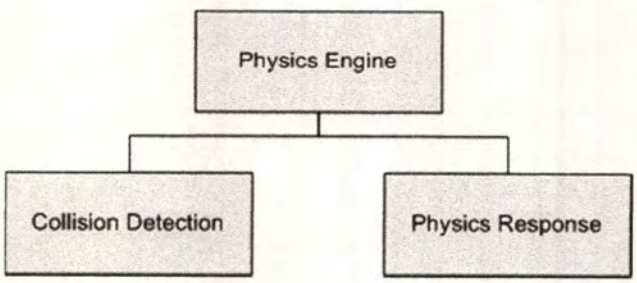

Figure 2.5 Farseer Physics Engine Diagram

2.3.1 Collision detection

Collision detection is an important issue in Physics Engine and responsible for detecting when objects overlap and by how much they overlap. Farseer Physics Engine has separated the way to detect the collision in 2 phases which are Broad phase and Narrow phase.

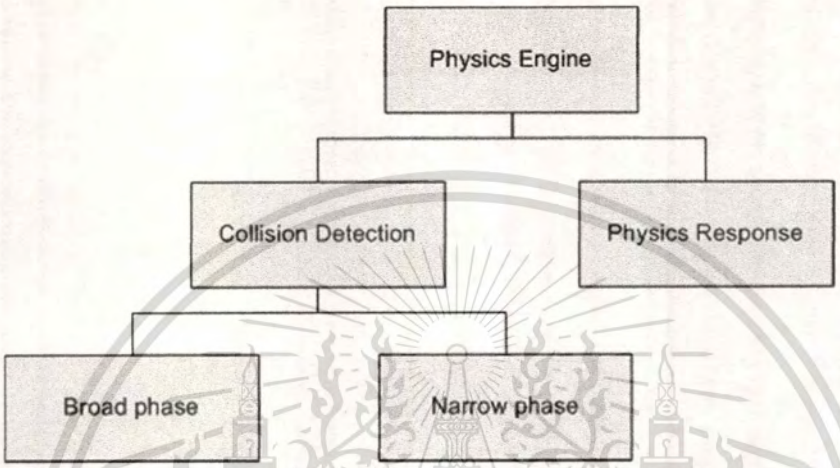

Figure 2.6 More Farseer Physics Engine Diagram

But, first let us introduce you to geometry which being used for collision detection in both broad phase and narrow phase.

Geometry

The geometry is the heart of collision detection. Geometry contains a set of vertices that define the edge of your shape. The geometry is in control of collision detection and calculating the impulses associated with colliding with other geometries.

Figure 2.7 A geometry contains many Vertices around an object

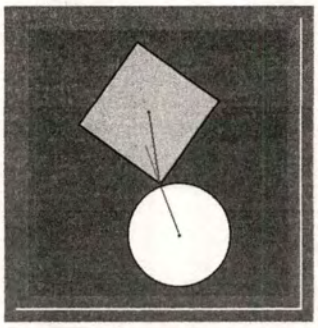

Figure 2.8 A geometry with collision vertices

Broad phase collision detection

The broad phase collision detection relies on advanced algorithms to speed up the collision detection by reducing the work the engine has to do. An effective algorithm used in Broad phase is the AABB (Axis-aligned bounding box). Other 3 kinds of broad phase collision detection algorithms are

1. Sweep And Prune (called SAP) [4]
2. Selective Sweep
3. Brute Force[5]

1.) 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.

2.) 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.

3.)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.

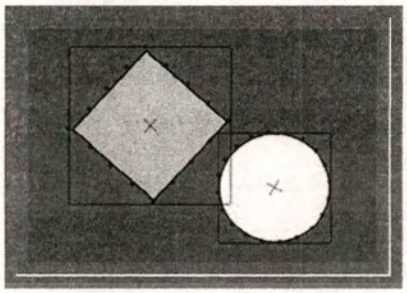

Figure 2.9 Intersection between two bounding boxes

Narrow phase collision detection

The narrow phase is where we take all the collision pairs generated by the broad phase, and do further checking on them. Unlike the broad phase, the narrow phase tests the real shapes (polygons) against each other. Farseer Physics Engine, there are 2 different narrow phase collision detection algorithms:

1. Distance Grid
2. SAT (Separate Axis Theorem) [6]

1.) Distance 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.

2.) SAT or Separate Axis Theorem is an algorithm that came with Farseer Physics. Compared to the Distance Grid, it only supports convex polygons by nature. It is a faster algorithm than the Distance Grid because it takes advantage of the convex-only polygons. As SAT is based on different principals then Distance Grid some changes need to take place.

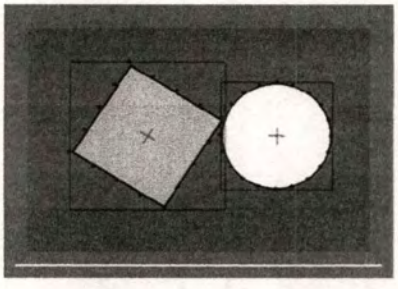

Figure 2.10 Collision detected from vertices

2.3.2 Physics Response

The physics response is one of components in Physics Engine, which been active when 2 geometries collide with each other. The Arbiter class in the "Narrow phase" 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.

You can also apply manual impulses to a body. (The body 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
2) Impulse
3) Torque

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.

Force changes the acceleration of an object. For example, when we hit a baseball the acceleration of the ball is changed because we applied force (in the form of swinging a bat) to the ball. If the ball was pitched to us, then it had a positive acceleration coming toward us; when we hit it we changed the acceleration to the negative direction.

Force is a very important concept in physics. This is understandable because it is constantly in effect. For example, even when we are sitting down there is a force of gravity keeping us in the chair. The first example with the baseball is considered contact force, as an object (the bat) made contact with the ball to make it change its acceleration. Gravity is considered a field force (or a force-at-a-distance) because it does not have to have contact with the object while applying force. Gravity causes items to be drawn to the Earth's core. This is why we can sit (and stand) and not float about. This is why we have weight. It is all because of gravity.

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.

Impulse is defined as a force that acts over a very short period of time. For example, the force exerted on a bullet when fired from a gun is an impulse force. The collision forces between two colliding objects are impulse forces, as when you kick a football or hit a baseball with a bat.

More specifically, impulse is a vector quantity equal to the change in momentum. The so-called impulse-momentum principle says that the change in moment is equal to the applied impulse. For problems involving constant mass and moment of inertia, you can write

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

2.4 Farseer Physics Engine Classes Overview

The important classes that involve with Farseer Physics Engine are listed below.

2.4.1) PhysicsSimulator

The physics simulator manages all physics objects while the simulation is running. It controls the update loop for the simulation.

Public Properties

- **GeometryList**

  The list of geometries that are part of the simulation.

- **BodyList**

  The list of bodies that are part of the simulation.

- **JointList**

  The list of joints that are part of the simulation.

- **ControllerList**

  The list of controllers that are part of the simulation.

- **Vector2 Gravity**

  The gravity currently applied to objects in the simulation.

Public Methods

- **Add(Geometry)**

  Adds a Geometry to the simulation.

- **Add(Body)**

  Adds a Body to the simulation.

- **Add(Controller)**

  Adds a Controller to the simulation.

- **Add(Joint)**

  Adds a Joint to the simulation.

- **Clear()**

  Clears all physics objects from the simulation. More specifically, it clears Bodies, Geometries, Joints, Controllers, and the internal Arbiters.

- **Geometry Collide(Vector2 point)**

Checks to see if a given point collides with any geometries in the simulation. If it does, the geometry is returned. If it does not, null is returned.

- **PhysicsSimulator()**

  Constructor that defaults to zero gravity.

- **PhysicsSimulator(Vector2 gravity)**

  Constructor with gravity.

- **Remove(Geometry)**

  Removes a Geometry from the simulation.

- **Remove(Body)**

  Removes a Body from the simulation.

- **Remove(Controller)**

  Removes a Controller from the simulation.

- **Remove(Joint)**

  Removes a Joint from the simulation.

- **Update(float dt)**

dt is the time in seconds. This method steps the simulation forward in time. It should be called continuously in an update loop while the game is running. This method assumes a fixed time step (dt). This means the value of dt should be the same each update call and it should be called with a frequency equal to dt.

For example, if you want to update the simulation at 100 frames per second, you would call update every 10 milliseconds and pass .01 (10ms converted to seconds) as your value for dt.

2.4.2) Body

Bodies are the core physics objects. 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.

Public Properties

- **AngularVelocity**

  The rate at which a body is rotating

- **Enabled**

  Sets whether or not the body will take part in the simulation. If not enabled, the body will remain in the internal list of bodies but it will not be updated.

- **Force**

  The total amount of force that will be applied to the body in the upcoming loop. The Force is cleared at the end of every update call, so this value should only be called just prior to calling update.

- **Torque**

  The total amount of torque that will be applied to the body in the upcoming loop. The Torque is cleared at the end of every update call, so this value should only be called just prior to calling update.

- **TotalRotation**

  Returns the total rotation of a body. If a body spins around 10 times then TotalRotationwold return 2 * Pi * 10. This property is mostly intended for internal use by the angle joints and springs but it could be useful in some situations for game related things.

- **Position**

  Gets the position of a body in Vector2. It returns X and Y position of the body on the screen.

- **Rotation**

  Gets or sets the rotation of the body.

- **LayerDepth**

  Gets layer depth of a body which can be used in XNA Draw method. In order to draw body's appropriate layer.

Public Methods

- **ApplyForce(Vector2 amount)**

  Apply force to a body.

2.4.3) Geom

The geometry class is the heart of collision detection. A Geom need a body and a set of vertices for its geometry.

Public Properties

- **InSimulation**

  Returns true if the geometry is added to the simulation.Returns false if the geometriy is not.

- **Body**

  Gets the Body attached to the geom.

- **FrictionCoefficient**

  Controls the amount of friction a geometry has when in contact with another geometry. A Value of zero implies no friction. When two geometries collide the average friction coefficient between the two bodies is used.

- **RestitutionCoefficient**

  The coefficient of restitution of the geometry controls how bouncy an object is when it collides with other geometries. Valid values range from 0 to 1 which 1 implies 100% restitution (perfect bounce) whereas 0 implies no restitution (think a ball of clay).

- **AABB**

  Gets or sets the Axis Aligned Bounding Box instance of the geom.

- **CollisionEnabled**

  Gets or sets a Value indicating whether collision is enabled.

- **CollisionResponseEnabled**

  Gets or sets a value indicating whether collision response is enabled.

  If 2 geoms collide and CollisionResponseEnabled is false, then impulses will not be calculated for the 2 colliding geoms. They will pass through each other, but will still be able to fire the OnCollision event.

- **CollisionGroup**

  Gets or sets the collision group. If 2 geoms are in the same collision group, they will not collide.

- **CollisionCategories**

  Gets or sets the collision categories.

- **CollidesWith**

  Gets or sets the collision categories that this geom collides with.

- **Tag**

  Gets or sets the tag. A tag is used to attach a custom object to the Geom.

- **IsSensor**

  Gets or sets a Value indicating whether this instance is a sensor. A sensor does not calculate impulses and does not change position (it's static). It does however detect collisions.

  Sensors can be used to sense other geoms.

Public Methods

- **SetBody(Body bodyToSet)**

  Sets the body attached to this geom.

- **Collide(Vector2 position)**

  Checks to see if the geom collides with the specified point.

- **Collide(Geom geometry)**

  Checks to see if the geom collides with the specified geom.

Chapter 3

Physics Engine for 2.5D XNA Game

This chapter covers the methods we have done to develop Physics Engine for 2.5D Game by using Farseer Physics Engine in order to make it works properly in 2.5 dimensions. We describe changes and improvements we did to Farseer Physics Engine. After that, we show our improved 2.5D Physics Engine.

3.1) Change from Vector2 to Vector3

In Farseer Physics Engine, object is restricted to only 2 dimensions(X and Y). So, to reach our objectives we have to adapt Farseer Physics into 2.5 dimensions.

First we implement Z dimension to our game by declare Z axe to each objects, to represent the depth of objects in game's world.

```
internalVector3 position3D =Vector3.Zero;
internalVector2 drawPosition =Vector2.Zero;
```

But game still is a 2 dimensions picture, it's not have depth of image so we need method to translate 3 dimensions into 2 dimensions for draw object in to the game.

```
internalvoid _3DTo2DPosition()
  {
        drawPosition.X =position3D.X -(float)(positionZ /Math.Tan((180 -
    physSim.Angle)*(Math.PI /180)));
        drawPosition.Y =physSim.Height -(position3D.Y +positionZ);
    if(ground.ObjectGeom !=null)
        drawPosition.Y -=ground.DistanceObjToGround;
        base.Position =drawPosition;
  }
```

By setting *physSimAngle* variable to the game can indicate angle of depth of the game, for angle of object moving in/out in Z direction.(Figure 3.1)

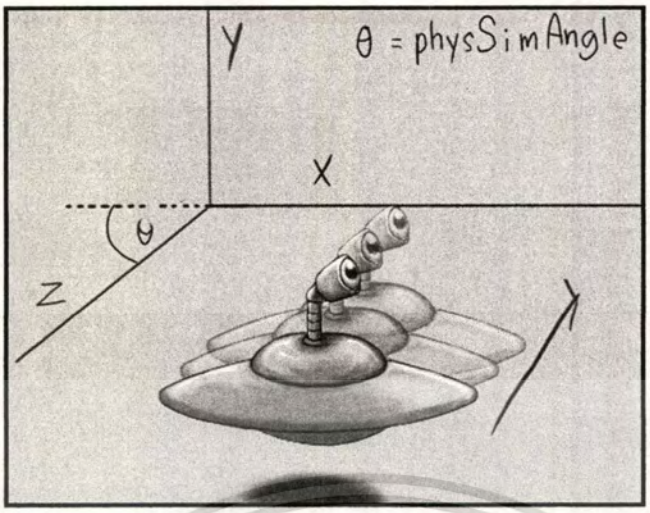

Figure 3.1 Positioning in 3 dimensions

3.2) Add Ground for every object

Physics Engine needs a ground object to calculate bouncing, friction, etc. In original Farseer Physics Engine Create a ground of game is easy, by creating another object floating at the bottom of screen then you have a ground (Figure 3.2).

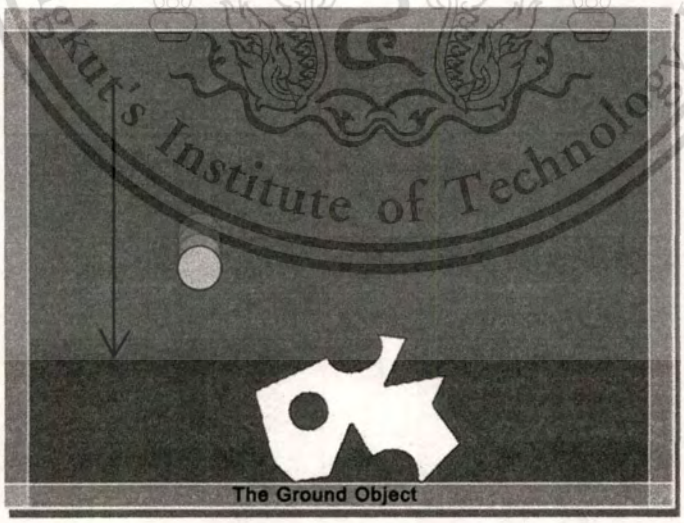

Figure 3.2 Object falls to the ground in 2D

But in 2.5D games project we can't do like this because ground is flattering across all screen and character have to step on it. (Figure 3.3)

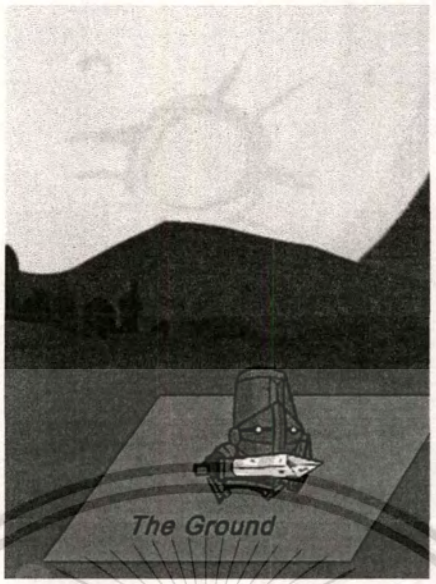

Figure 3.3 Ground in 2.5D

In order to do that, we create a ground class to represent of ground for each object. Each object has a ground for its own and will not collide with ground of other object. It also need to be transparent and move along Z-axis, If object move far from a screen (Z index increase), ground of object will move up to represent a position of the ground in that Z index. (Figure 3.4)

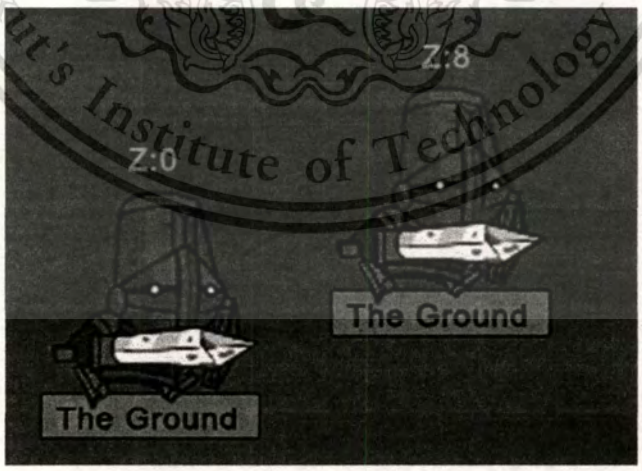

Figure 3.4 Ground supports for object concept

A ground contains a body and geometry for it. The body of a ground is used to represent its position but the geometry is used to support the object in different Z-axis on the ground. The geometry of a ground

can make the collision occur if the object itself collides to it. Although the object is flying in the air, the position of the ground will change relative to the object's position. This way can make the ground to support the object when it falls and collide to the ground. The figure 3.5 below shows the completion of implementing a ground for each object.

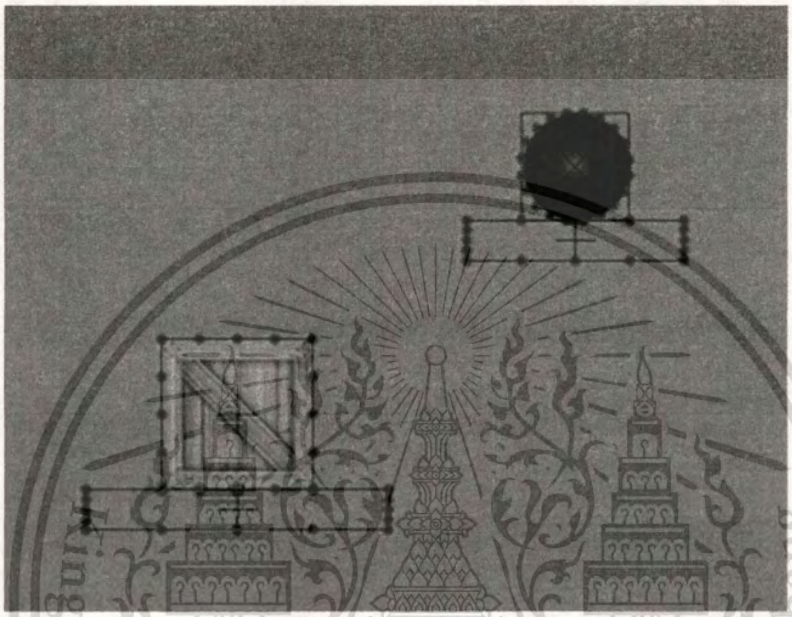

Figure 3.5 Ground supports for object concept implementation

3.3) Implementing Force in Z-axis

Force is very important in Physics Engine. It takes care of object's movement and changes the acceleration of an object. Traditionally, Farseer Physics Engine can apply force to an object in 2 directions only which are X and Y axis. It uses **Vector2 force;** to keep track of object's force.

Since we want to move and apply force to objects in 2.5 Dimension. Also, the object need to move along Z-axis to make it's like in 3D world. Therefore, we need to implement the force in Z-axis.

First, we add new variable, **float forceZ;** to keep track of object's force in Z-axis.

Second, we add calculation of acceleration, velocity and force in Z-axis. These steps of calculation can be seen from the following:

3.3.1) Implement ApplyForceZ method

This ApplyfoceZ method is a utility method that can be used by developers. The amount is kept to forceZ variable. This forceZ is used in calculation of velocity in Z-axis, in the next step.

```
publicvoid ApplyForceZ(float amount)
        {
            forceZ += amount;
        }
```

If developers want an object to move in Z-axis, they can easily call this method.

3.3.2) Calculate and assign velocity of an object from forceZ

We calculate the acceleration in Z-axis of an object from equation: a = f * 1/m; m = object mass, f = force and a = acceleration

```
_accelerationZ = forceZ * inverseMass
```

Then we can calculate the change of velocity in Z-axis from equation: dv = a * dt; dv = delta velocity, a = acceleration and t = delta time;

```
_dvZ = _accelerationZ * dt;
```

Since we got delta velocity in Z-axis, we can calculate current velocity in Z-axis from

```
//Assign current linear velocity in Z-axis to previousLinearVelocityZ
_previousLinearVelocityZ = LinearVelocityZ;
//Set new velocity in Z-axis from previous velocityZ + delta velocityZ
LinearVelocityZ = _previousLinearVelocityZ + _dvZ;
```

3.3.3) Calculate the position in Z-axis of an object from its velocity in Z-axis

We already have velocity in Z-axis (LinearVelocityZ) from previous step, now it is time to calculate the position of an object in Z-axis from LinearVelocityZ. After that, we assign the position in Z-axis to positionZ variable that can be used by PhysicsSimulator to draw object position on screen.

```
//Calculate Position in Z-axis Change
```

```
_tempDeltaPositionZ = LinearVelocityZ * dt;

if (_tempDeltaPositionZ != 0)
{
//Save old position
      _previousPositionZ = positionZ;

//Apply position change
      positionZ = _previousPositionZ + _tempDeltaPositionZ;
}
```

3.4) Add Shadow for every object

We add shadow beneath every object in 2.5D Physics Engine. This makes us know the object's position in Z-axis easily. Besides that, if the object moves along Y-axis (Object's on fly), we scale the shadow size. A more object fly above the ground, its shadow is getting smaller.

Then we implement an easy to use method name DrawShadow() which can be called by developers to draw shadow for each object.

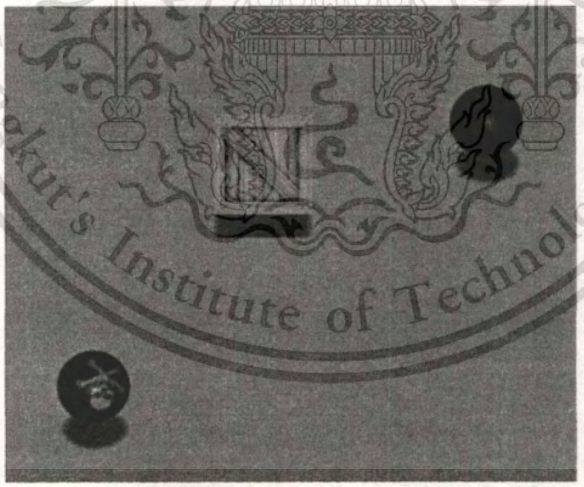

Figure 3.6 Shadow for every object on the ground

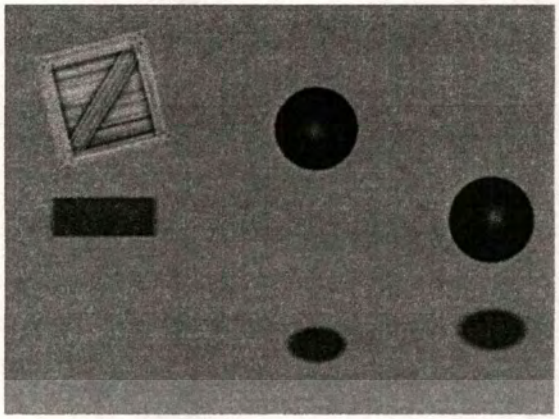

Figure 3.7 Shadow for objects on fly

3.5) Implement virtual geometry for every object to calculate collision in Z-axis

The collision in Z-axis plays a vital role in order to make every object move correctly and act like they are in 3D world. Traditionally, Farseer Physics Engine calculates the collision between objects from their geometry. It can only support collision in 2 dimensions as you can see from the figure 3.8.

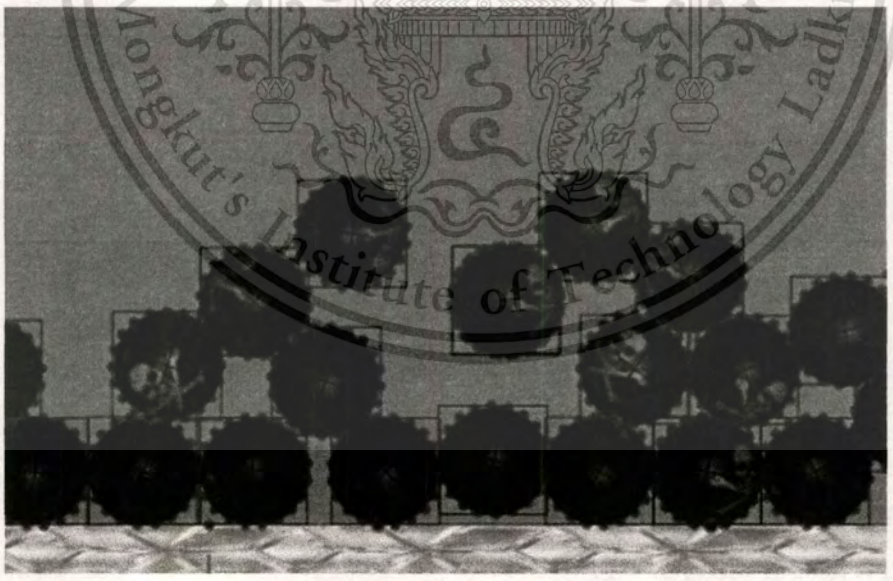

Figure 3.8 Collision between geometries in Farseer Physics Engine

We add geometry for every object to keep track of collision when two objects collide in Z-axis. We call it virtual geometry because it does not collide with other geometry such as ground geometry or object

geometry. This virtual geometry lies under the object and will collide with other virtual geometry only. The collision in Z-axis occurs when two virtual geometries collide to each other.

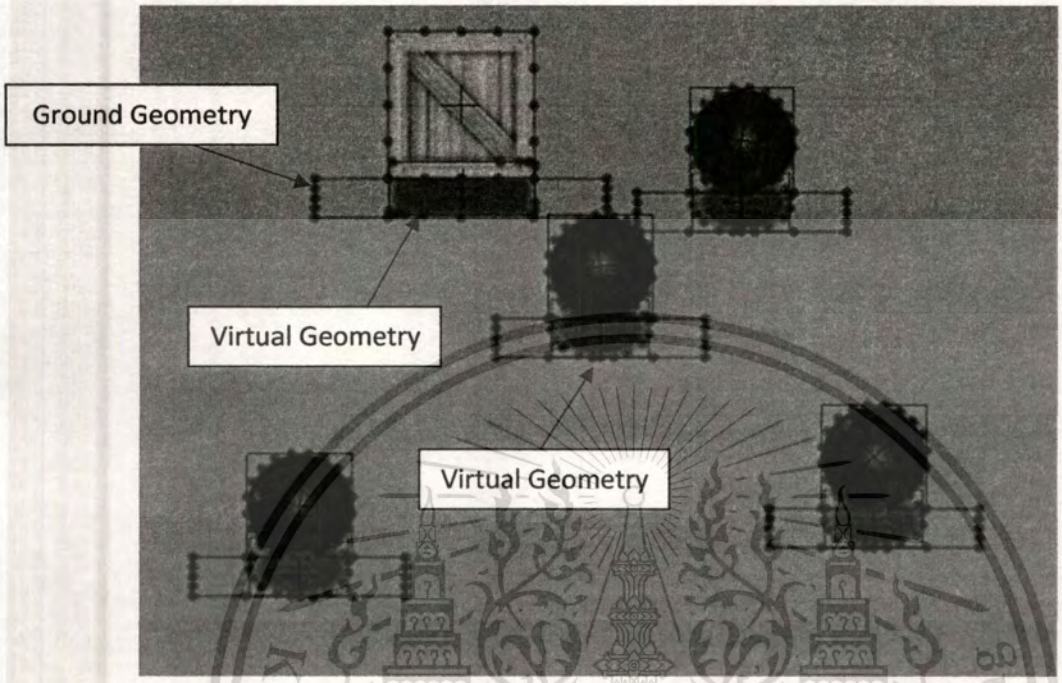

Figure 3.9 Virtual geometries are under each object

By implementing the virtual geometry, it makes us know when the collision occurs in Z-axis easily. If the collision occurs between two virtual geometries, hence there is a collision in Z-axis. The example of collision between two virtual geometries can be seen from the figure 3.10.

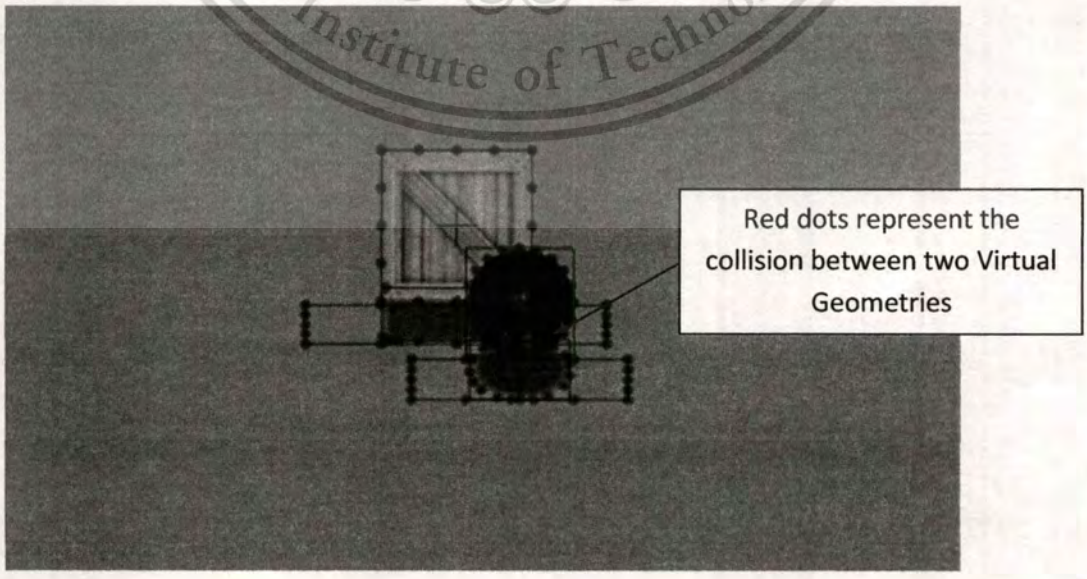

Figure 3.10 Collision between virtual geometries

Another problem that we need to concern about is that, although two virtual geometries collide between each other but the object itself does not collide (Thinking of they are flying on the different height). This situation should not trigger the collision to be occurred. We solved this problem by calculating object's height and its height from the ground every time the collision between virtual geometry occurs. Therefore, if two objects are on different height, we let them pass through each other and the collision will not occur. The figure 3.11 can make it easier to understand.

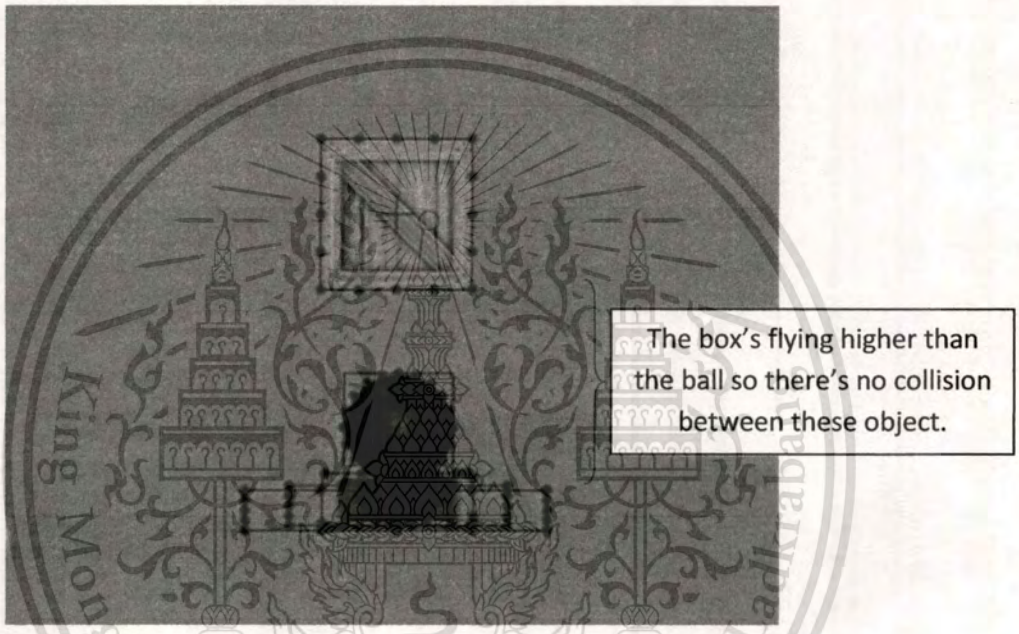

Figure 3.11 No collision between virtual geometries of objects on different height

3.6) Create a new Platform class

Platform is another important thing that our Physics Engine should provide because it must be weird if the object can be only placed on ground. In tradition, Farseer Physics Engine provides us the way to create platform in 2D by just creating a body and geometry for it. It is easy and not complicated just like creating a box object. In contrast, the platform in 2.5D is more complicated than that.

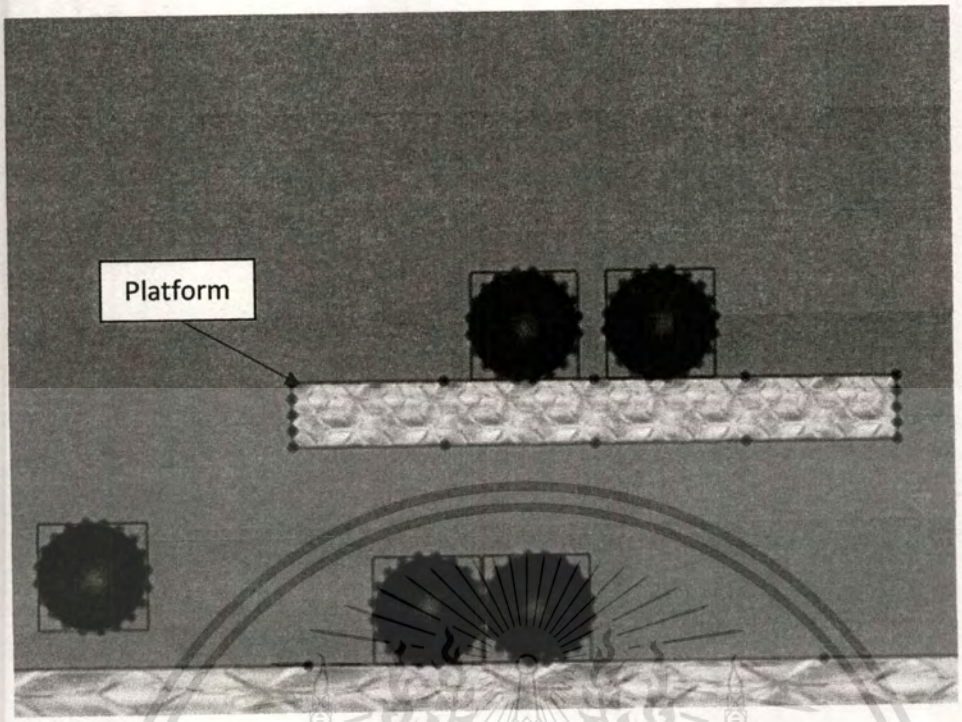

Figure 3.12 Platform in 2 dimensions from Farseer Physics Engine

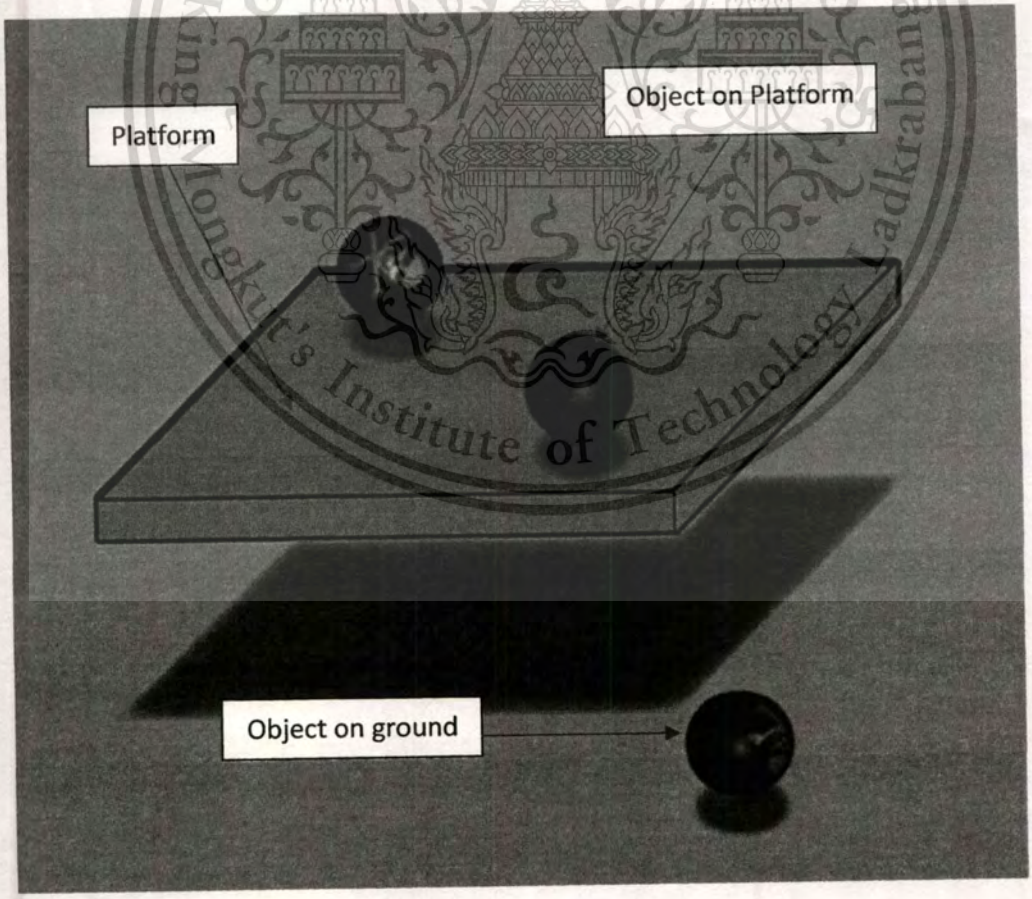

Figure 3.13 Platform in 2.5D concept

Differences between platform in 2D and 2.5D are shown in the above figure. We solve this issue by implementing a new class called Platform. This Platform is an easy to use class for the developers. The solution inside the Platform class is that the platform will create virtual platform geometry for every object on it. The virtual platform geometry is used to support and handle every object on it. This will not let the object pass through the platform and fall to the ground. The most important is that each object must have only one virtual platform geometry and the object must not collide with the other virtual platform geometry. The figure 3.14 shows complete 2.5D Platform and its structure. There are many virtual platform geometries inside a platform to make it handle all objects on it.

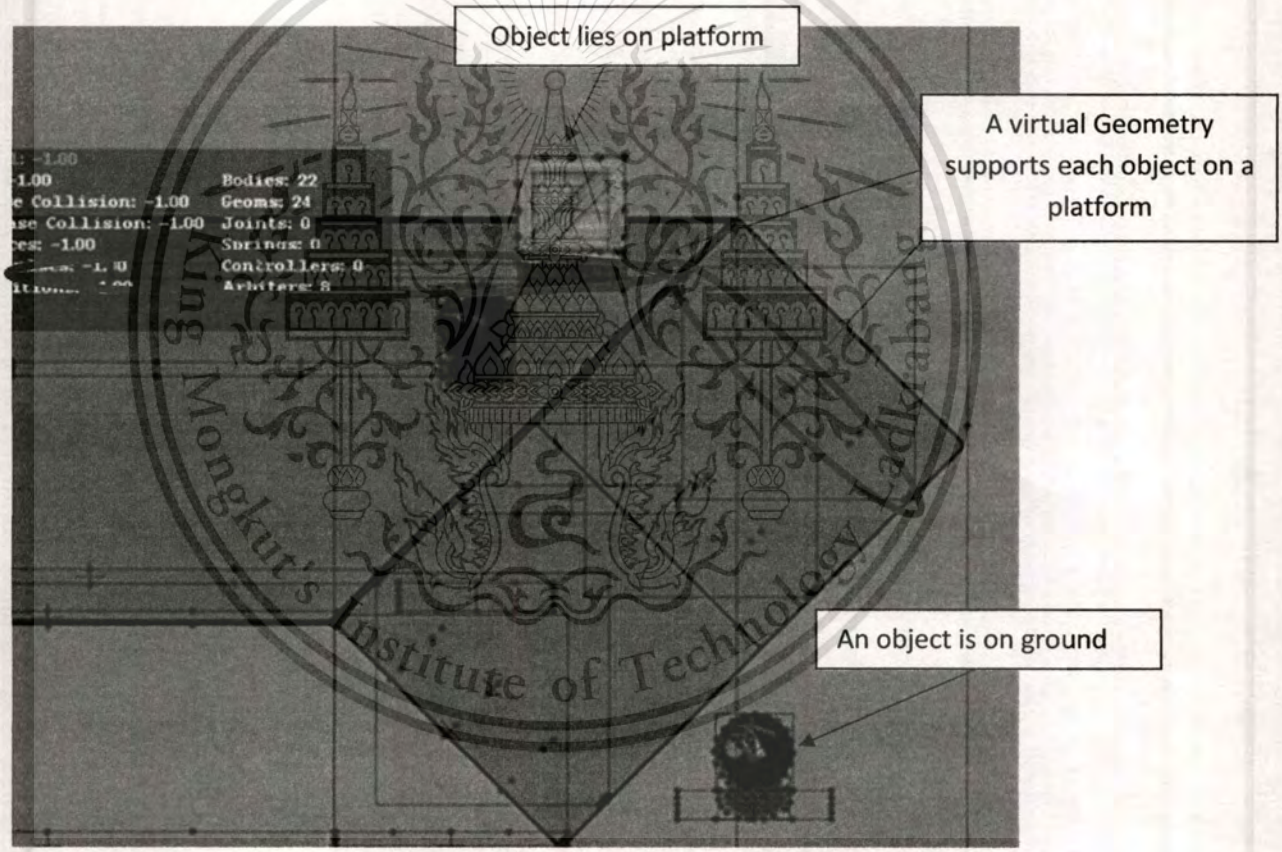

Figure 3.14 Platform and virtual platform in 2.5D

Chapter 4

Results and Discussion

This Chapter shows the results of 2.5D Physics Engine. We create demo scenes for testing by using both original Farseer Physics Engine and 2.5D Physics Engine. So, we can test and observe the result comparing between Farseer and 2.5D Physics Engine.

4.1) Demo Scene1

There are two types of Demo Scene1 which are Demo Scene1 using Farseer Physics Engine and Demo Scene1 using 2.5D Physics Engine. Both of them are simple and consist of many primitive objects, a ground and gravity. So, it can be easily focused on the differences between these two Physics Engines.

4.1.1) Demo Scene1 using Farseer Physics Engine

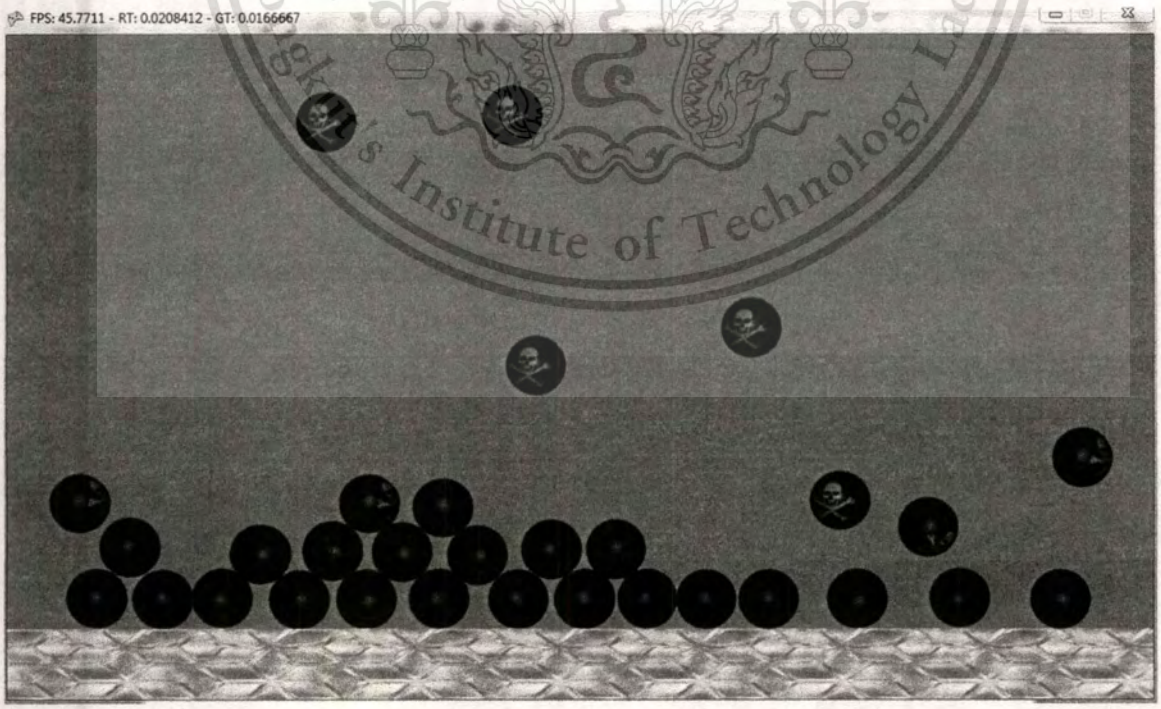

Figure 4.1 Demo Scene1 using Farseer Physics Engine

From the figure 4.1, this is a simple demo scene consists of circle body objects and a ground at the bottom. We also set up the gravity so every object is falling from the sky to the ground. The objects move in only X and Y axis since it uses Farseer Physics Engine.

4.1.2) Demo Scene1 using 2.5D Physics Engine

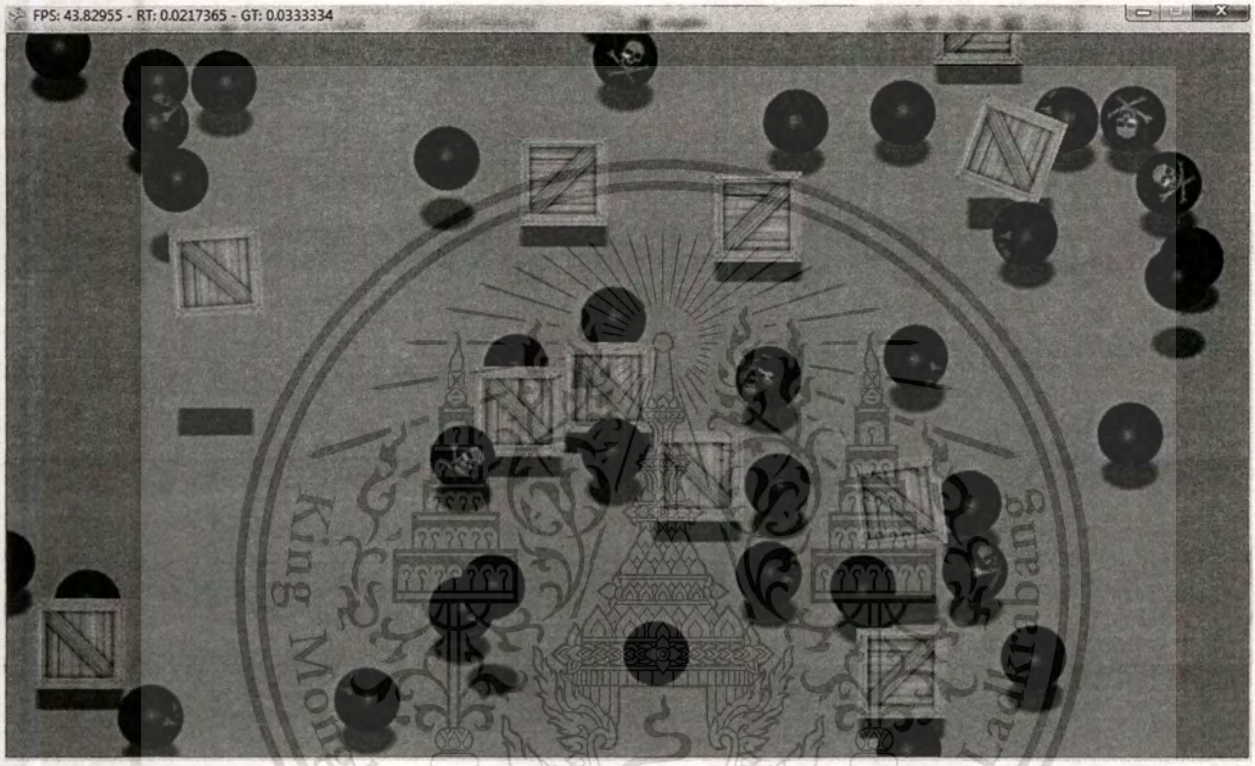

Figure 4.2 Demo Scene1 using 2.5D Physics Engine

From the figure 4.2, this demo scene uses 2.5D Physics Engine. The figure shows that our 2.5D Physics Engine can work properly as desire. There are rectangle body objects and circle body objects in the scene. Every position on the screen can be ground in 2.5D. It can be seen that the objects are falling to the ground and there is a shadow under every object to indicate its position in the Z-axis.

4.2) Demo Scene2

We implemented Demo Scene2 from both Farseer Physics Engine and our 2.5D Physics Engine like in the Demo Scene1. There are platforms added into this Demo Scene2 to compare the result between these

two Physics Engines. The platform in 2.5D is different from the platform in 2D because the 2.5D platform has the depth but in 2D it does not.

4.2.1) Demo Scene2 using Farseer Physics Engine

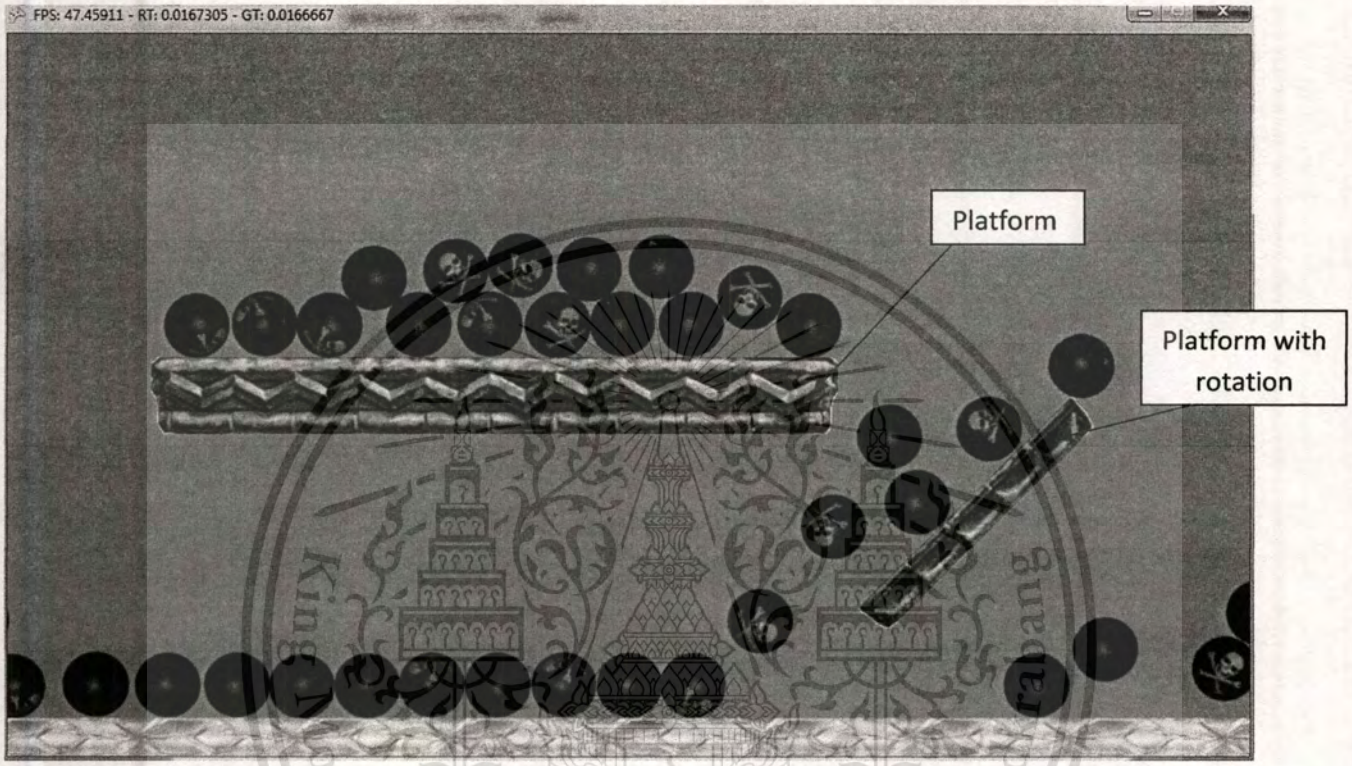

Figure 4.3 Demo Scene2 using Farseer Physics Engine

From the figure 4.3, there are two platforms in this demo scene. One placed horizontally and another with the rotation. The objects can stay on the platform in 2D properly.

4.2.2) Demo Scene2 using 2.5D Physics Engine

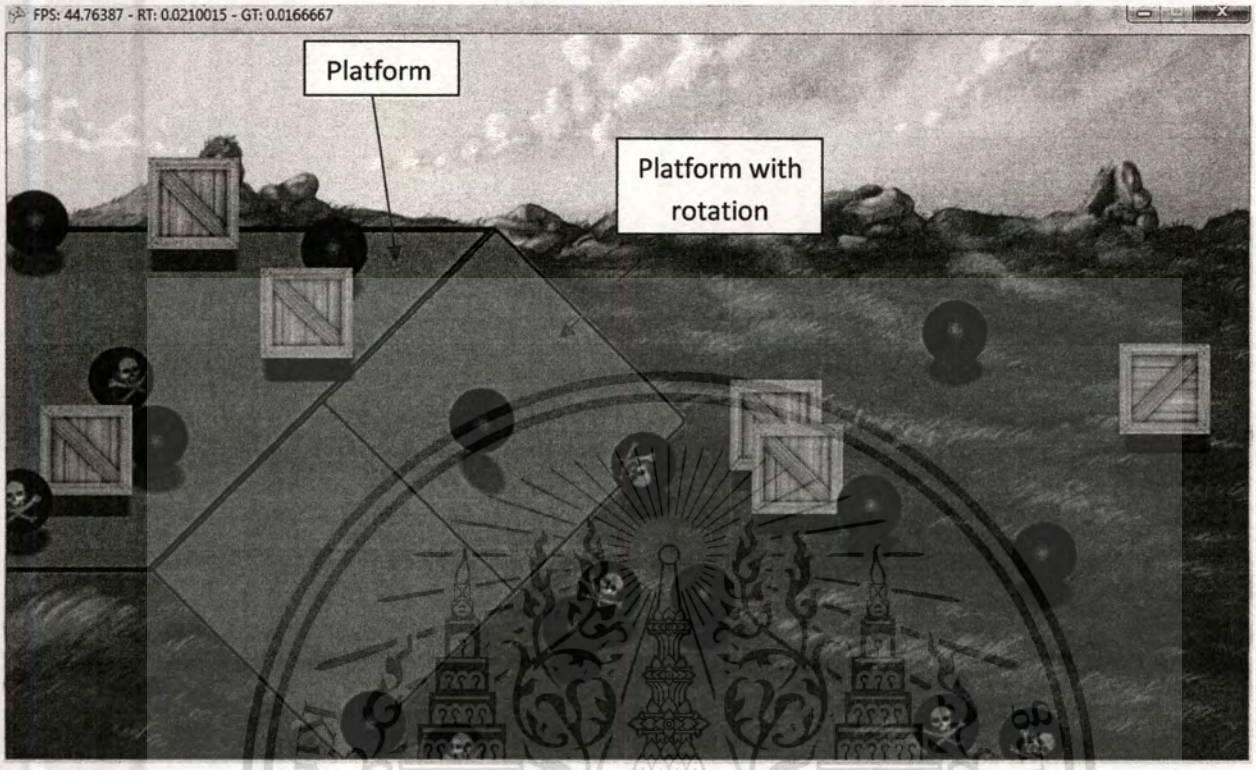

Figure 4.4 Demo Scene2 using 2.5D Physics Engine

This Demo Scene2 from figure 4.4 is the scene that uses 2.5D Physics Engine. There are two platforms added on the left hand side of the screen. The left-most platform is placed horizontally and lifted higher from the ground. Another platform is rotated to make it looks like a slope. The objects and physics simulation are performed properly. We also added the background to make this demo scene more realistic.

Chapter 5

Conclusion and Recommendation

5.1 Conclusion

The Farseer Physics Engine can work pretty well with 2.5 dimensions game, through some adjustment and modification; we are success to stimulate physical to 2.5 dimensions games by using 2 dimensions Physics Engine. The 2D objects can collide and bouncing around like they were in 3 dimensions world. We also can apply mass, frictions, gravity, force, impulse to objects and their reaction is very realistic.

We also include the Platform class to represent a slope or viaduct objects in a game world. That our Physics Engine does not only work on flatting ground. But, also can work on sloping surfaced and floating platform.

The limitation of our project is rotation the objects in Z axis. Because the objects in 2.5 dimensions game still, are 2 dimensions images. So, it is nearly impossible to rotation the objects in Z axis, because it never exists at first. We can virtually, moving object, collide or bouncing the 2 dimensions objects in Z axis. But for the rotation, it is just impossible (or we thought so).

Finally, there may be a performance issue in some cases. Since our 2.5D Physics Engine, one object consists of 3 geometries (Geometry of the object itself, ground geometry and virtual geometry). Although it can handle large amount of objects (20-40) but if there are huge amount of objects simultaneously. This may cause the frame rate drop down.

5.2 Recommendation

There are many ways that can improve our 2.5D Physics Engine further.

1. Current, collision in Z axis are fine but not precisely. When two objects are colliding with each other it can calculate only 2 axes, like X and Y or X and Z axis. But it cannot calculate in 3 axes

together. Because of vertices that use to calculate colliding position cannot be overlapped, but in our 2.5 dimensions game object are need to overlap so if objects are in different Z index if will use vertices of shadow, instead of vertices of objects' images. We suggest to improve this issue when object colliding, it should check in all axes, not just two axes

2. Our 2.5D Physics Engine simulate major principles of physics such as Force, Impulse, Kinetic, Friction and Restitution. So, it can be improved to simulate other physics principle in 2.5D such as joint, spring, fluid simulation.

References

[1] CodePlex Open Source Community. "Farseer Physics Engine." [online]. Available : http://www.codeplex.com/FarseerPhysics. 2006.

[2] Ericson Christer, Real-time collision detection, Morgan Kaufmann series in interactive 3D technology, Amsterdam: Elsevier, pp. 329–338, ISBN 978-1558607323. 2005.

[3] Baraff D., *Dynamic Simulation of Non-Penetrating Rigid Bodies, (Ph. D thesis)*, Computer Science Department, Cornell University, pp. 52–56. 1992.

[4] Lin Ming C. *Efficient Collision Detection for Animation and Robotics (thesis)*. University of California, Berkeley. 1993.

[5] Golshtein E.G.; Tretyakov N.V.; translated by Tretyakov N.V. *Modified Lagrangians and monotone maps in optimization*. New York: Wiley. p. 6. ISBN 0471548219. 1996.

[6] Grant Palmer, "Physics for Game Programmers", Apress, 2005.

[7] David M. Bourg, "Physics for Game Developers", O'Relly & Associates, Inc., 2002.

Appendix A

2.5D Physics Engine API

TwoHalfPhysicsSimulator Class

The physics simulator manages all physics objects while the simulation is running. It controls the update loop for the simulation. It keeps track of bodies, geometries, and other dynamics.

Constructors

Name	Description
TwoHalfPhysicsSimulator(int height)	Initializes a new instance of the TwoHalfPhysicsSimulator class. - Height is height of the screen.
TwoHalfPhysicsSimulator(int height, int angle)	Initializes a new instance of the TwoHalfPhysicsSimulator class. - Height is height of the screen. - Angle is an angle in Z-axis (Counterclockwise).
TwoHalfPhysicsSimulator(int height, Vector2 gravity)	Initializes a new instance of the TwoHalfPhysicsSimulator class. - Height is height of the screen. - Gravity is used to apply gravity force.
TwoHalfPhysicsSimulator(int height, int angle, Vector2 gravity)	Initializes a new instance of the TwoHalfPhysicsSimulator class. - Height is height of the screen. - Angle is an angle in Z-axis (Counterclockwise). - Gravity is used to apply gravity force.

Fields

Name	Description
boolean Enabled	If false, the whole simulation stops. It still processes added and removed geometries.
Vector2 Gravity	Gravity applied to all bodies.
int Iterations	The number of iterations the engine should do when applying forces. Increase this number to have a more stable simulation but it will affect performance.
int MaxContactsToDetect	The maximum number of contacts to detect in the narrow phase. Default is 10.
int MaxContactsToResolve	The maximum number of contacts to resolve in the narrow phase. Default is 4.

Methods

Name	Return Value	Description
Add(Geom)	void	Add geometry to the Physics Simulator.
Add(Body)	void	Add body to the Physics Simulator.
Clear	void	Resets the physics simulator back to its original state. Only gravity is persisted.
Collide(Vector2)	Geom	Checks if any geometries collide with the specified point. Return value is the first geometry.
CollideAll(Vector2)	List<Geom>	Finds all geometries that collides with the specified point.
Remove(Geom)	void	Remove geometry from the Physics Simulator.
Remove(Body)	void	Remove body from the Physics Simulator.

Update(float)	void	Updates the physics simulator with the specified time change.

Properties

Name	Description
int Height	Gets or Sets the height of the game screen.
int Angle	Gets angle of the Z-axis
GenericList<Body> BodyList	Gets a list of bodies in Physics Simulator.
GenericList<Geom> GeomList	Gets a list of geometries in Physics Simulator.
float UpdateTime	Gets an update time of Physics Simulator.

TwoHalfBody Class

TwoHalfBody is used to initiate the body which is the core physics object. Forces, torques, and impulses are applied to bodies and the bodies react by moving accordingly.

Constructors

Name	Description
TwoHalfBody(TwoHalfPhysicsSimulator)	Initializes a new instance of the TwoHalfBody class.
TwoHalfBody(TwoHalfPhysicsSimulator, TwoHalfBody)	TwoHalfBody constructor that makes a copy of another body.

Fields

Name	Description
float AngularVelocity	The rate at which a body is rotating.
bool Enabled	Sets whether or not the body will take part in the simulation. If not enabled, the body will remain in the internal list of bodies but it will not be updated.
bool IgnoreGravity	Gets or sets a Value indicating whether this body ignores gravity.
float LinearDragCoefficient	The linear drag coefficient is the amount of drag a body has. Linear drag is the drag applied when the body travels in a straight line. Default is 0.001f - tuned for a body of mass 1
Vector2 LinearVelocity	Gets or sets the linear velocity.
float RotationalDragCoefficient	Gets or sets the rotational drag coefficient.
Object Tag	Gets or Sets the Object Tag attached to this body.

Methods

Name	Return Value	Description
DrawShadow(SpriteBatch spriteBatch, TwoHalfGeom geom, Vector2 positionOffset, Vector2 scale, Color colorTint, float layerDepth, bool isShadowResizable)	void	Draw shadow for this body. - spriteBatch is a SpriteBatch used to draw. - geom is a geometry of this body. - positionOffset is an offset of shadow. - Scale is used to scale the size of shadow. - colorTint is used to tint the shadow. - layerDepth is the shadow's layer depth. - isShadowResizable is used to set whether shadow can be resized relative to object's height.
ApplyAngularImpulse(float)	void	Applies angular impulse.
ApplyForce(Vector2)	void	Applies force to the body.
ApplyImpulse(Vector2)	void	Stores all applied impulses so that they can be applied at the same time by the Physics Simulator.
ApplyTorque(float)	void	Adds a torque to the body.
ClearForce()	void	Clears the force of the body. This method gets called after each update.
ClearImpulse()	void	Clears the impulse of the body.
ClearTorque()	void	Clears the torque of the body. This method gets called after each update.
Dispose()	void	Dispose of the body from the Physics Simulator.

Properties

Name	Description
Ground Ground	Gets the ground of this body.
float LayerDepth	Gets layer depth of this body that can be used in draw method.
Vector3 Position3D	Gets or Sets the position of the body in virtual 3 Dimensions.
Vector2 Force	The total amount of force that will be applied to the body in the upcoming loop. The force is cleared at the end of every update call, so this Value should only be called just prior to calling update. This property is read-only.
float InverseMass	The inverse of the mass of the body (1/Mass).
bool IsDisposed	Gets or Sets whether the body is disposed.
bool IsStatic	Indicates this body is fixed within the world and will not move no matter what forces are applied.
float Mass	Gets or Sets the mass of the body.
Vector2 Position	Gets or Sets the position of the body in 2 dimensions.
float Rotation	Gets or sets the rotation of the body.
float Torque	The total amount of torque that will be applied to the body in the upcoming loop. The Torque is cleared at the end of every update call, so this value should only be called just prior to calling update. Torque can be thought of as the rotational analog of a force. This property is read-only.

TwoHalfGeom Class

TwoHalfGeom is used to create the geometry (Called Geom in short - Farseer) is the heart of collision detection. The geometry needs a body and a set of vertices that define the edge of the body shape. The geometry is in control of collision detection and calculating the impulses associated with colliding with other geometries.

Constructors

Name	Description
TwoHalfGeom()	Initializes a new instance of the TwoHalfGeom class.
TwoHalfGeom(TwoHalfBody body, TwoHalfGeom geometry)	Creates a clone of an already existing geometry.
TwoHalfGeom(TwoHalfBody body, Vertices vertices, float collisionGridSize)	Initializes a new instance of the TwoHalfGeom class. - body is attached body of this geom. - vertices are vertices of this geom. - collisionGridSize is the size of the grid collision.

Fields

Name	Description
CollisionCategory CollidesWith	Gets or sets the collision categories that this geometry collides with.
CollisionCategory CollisionCategories	Gets or sets the collision categories. Member off all categories by default.
bool CollisionEnabled	Gets or sets a Value indicating whether collision is enabled.
int CollisionGroup	Gets or sets the collision group. If 2 geometries are in the same collision group, they will not collide.
bool CollisionResponseEnabled	Gets or sets a value indicating whether collision response is

	enabled. If 2 geometries collide and CollisionResponseEnabled is false, then impulses will not be calculated for the 2 colliding geometries. They will pass through each other, but will still be able to fire the OnCollision event.
float FrictionCoefficient	Controls the amount of friction a geometry has when in contact with another geometry. A Value of zero implies no friction.
float RestitutionCoefficient	The coefficient of restitution of the geometry. This parameter controls how bouncy an object is when it collides with other geometries. Valid values range from 0 to 1 inclusive. 1 implies 100% restitution (perfect bounce) 0 implies no restitution (think a ball of clay).
Object Tag	Gets or Sets the Object Tag attached to this geometry.
CollisionEventHandler OnCollision	An event fired when this geometry collided with other geometry.
SeparationEventHandler OnSeparation	An event fired when this geometry separate from other geometry.

Methods

Name	Return Value	Description
Collide(Geom)	bool	Checks to see if the geometry collides with the specified geom.
Collide(Vector2)	bool	Checks to see if the geometry collides with the specified point.
Dispose()	void	Dispose of the geometry from the Physics Simulator.
SetBody(Body)	void	Sets the body of this geometry.

Properties

Name	Description
bool IsGround	Returns whether this geometry is the ground geometry.
Body Body	Returns the body attached to this geometry.
bool IsDisposed	Gets or Sets whether the geometry is disposed.
bool IsSensor	Gets or sets a Value indicating whether this instance is a sensor. A sensor does not calculate impulses and does not change position (it is static) it does however detect collisions. Sensors can be used to sense other geometries.
Vector2 Position	Gets the position. Compared to TwoHalfBody.Position, this property takes position offset of the geometry into account.
float Rotation	Gets the rotation. Compared to TwoHalfBody.Rotation, this property takes rotation offset of the geometry into account.

TwoHalfBodyFactory Class

This is an easy to use factory for creating bodies.

Methods

Name	Return Value	Description
CreateBody(TwoHalfPhysicsSimulator physicsSimulator, TwoHalfBody body)	TwoHalfBody	Creates a clone of body. - body is the body to be copied.
CreateBody(TwoHalfPhysicsSimulator physicsSimulator, float mass, float momentOfInertia)	TwoHalfBody	Creates new body. - mass is a mass of the body. - momentOfInertia is a moment of inertia of the body.
CreateCircleBody(TwoHalfPhysicsSimulator physicsSimulator, float radius, float mass)	TwoHalfBody	Creates new circle body. - physicsSimulator is an object of physicsSimulator. - radius is a radius of the circle body. - mass is a mass of the circle body.
CreateEllipseBody(TwoHalfPhysicsSimulator physicsSimulator, float xRadius, float yRadius, float mass)	TwoHalfBody	Creates an ellipse body. - physicsSimulator is an object of physicsSimulator. - xRadius is a major axis radius of the ellipse body. - yRadius is a minor axis radius of the ellipse body. - mass is a mass of the ellipse body.

CreatePolygonBody(TwoHalfPhysicsSimulator physicsSimulator, Vertices vertices, float mass)	TwoHalfBody	Creates a polygon Body. - physicsSimulator is an object of physicsSimulator. - vertices is vertices represent a polygon. - mass is a mass of the body.
CreateRectangleBody(TwoHalfPhysicsSimulator physicsSimulator, float width, float height, float mass)	TwoHalfBody	Creates a rectangle body. - physicsSimulator is an object of physicsSimulator. - width is a width of the rectangle body. - height is a height of the rectangle body. - mass is a mass of the rectangle body.

TwoHalfGeomFactory Class

This is an easy to use factory for creating geometries.

Methods

Name	Return Value	Description
CreateGeom(Body body, TwoHalfGeom geometry)	TwoHalfGeom	Creates a clone of a geometry. - geometry is the geometry to be copied.
CreateGeom(PhysicsSimulator physicsSimulator, Body body, TwoHalfGeom geometry)	TwoHalfGeom	Creates a clone of a geometry. - physicsSimulator is an object of physicsSimulator. - geometry is the geometry to be copied. - body is a body attached to this geometry.
CreateCircleGeom(TwoHalfPhysicsSimulator physicsSimulator, Body body, float radius, int numberOfEdges)	TwoHalfGeom	Creates a circle geom. - physicsSimulator is an object of physicsSimulator. - body is a body attached to this geometry. - radius is a radius of the circle geometry. - numberOfEdges is the number of edges for this circle geometry
CreateEllipseGeom(TwoHalfPhysicsSimulator physicsSimulator, Body body, float xRadius, float yRadius, int numberOfEdges)	TwoHalfGeom	Creates an ellipse geom. - physicsSimulator is an object of physicsSimulator. - body is a body attached to

		this geometry. - xRadius is a major axis radius of the ellipse geometry. - yRadius is a minor axis radius of the ellipse geometry. - numberOfEdges is the number of edges for this ellipse geometry
CreatePolygonGeom(PhysicsSimulator physicsSimulator, Body body, Vertices vertices, float collisionGridSize)	TwoHalfGeom	Creates a polygon geometry. - physicsSimulator is an object of physicsSimulator. - body is a body attached to this geometry. - vertices is vertices represent a polygon. - collisionGridSize is the size of the grid collision.
CreateRectangleGeom(TwoHalfPhysicsSimulator physicsSimulator, Body body, float width, float height)	TwoHalfGeom	Creates a rectangle geometry. - physicsSimulator is an object of physicsSimulator. - body is a body attached to this geometry. - width is a width of the rectangle geometry. - height is a height of the rectangle geometry.

Ground Class

This is the ground used to support under every object.

Properties

Name	Description
TwoHalfGeom ObjectGeom	Gets the object geometry on this ground geometry.
float RestitutionCoefficient	Gets or Sets the coefficient of restitution of the ground geometry.
float FrictionCoefficient	Gets or Sets the coefficient of friction of the ground geometry.

Methods

Name	Return Value	Description
setObjectGeom(TwoHalfGeom objectGeom, Texture2D shadowImage)	void	Sets the object geometry for this ground. - objectGeom is the object geometry on this ground geometry. - shadowImage is the texture image used to be drawn on the shadow position.

Platform Class

Platform class is used to create the platform in 2.5 dimensions.

Constructors

Name	Description
Platform(TwoHalfPhysicsSimulator physSim, int width, int height, int depth, Vector3 position3D, Texture2D shadowImage)	Creates a new platform. - physSim is an object of physicsSimulator. - width is the width of the platform. - height is the height of the platform. - depth is the depth of the platform. - position3D is the position of the platform in 2.5 dimensions. - shadowImage is the texture image used to be drawn under the platform.
Platform(TwoHalfPhysicsSimulator physSim, int width, int height, int depth, Vector3 position3D, float rotation, Texture2D shadowImage)	Creates a new platform. - physSim is an object of physicsSimulator. - width is the width of the platform. - height is the height of the platform. - depth is the depth of the platform. - position3D is the position of the platform in 2.5 dimensions. - rotation is used to rotate the platform. - shadowImage is the texture image used to be drawn under the platform.
Platform(TwoHalfPhysicsSimulator physSim, int width, int height, int depth, Vector3 position3D, float rotation, Texture2D shadowImage,	Creates a new platform. - physSim is an object of physicsSimulator. - width is the width of the platform. - height is the height of the platform.

int angleInZAxis)	- depth is the depth of the platform. - position3D is the position of the platform in 2.5 dimensions. - rotation is used to rotate the platform. - shadowImage is the texture image used to be drawn under the platform. - angleInZAxis is the angle of the platform in Z axis.