Pisheh Var, Mahrad

[Mpishe@essex.ac.uk](mailto:Mpishe@essex.ac.uk)

Git Link = <https://cseegit.essex.ac.uk/ce812/pisheh_var_m/pbg/tree/master/AssignmentPhysicsBasedGame>

Physics based games Assignment

Contents

[**The Game Description:** 3](#_Toc6855096)

[Game controls: 3](#_Toc6855097)

[Key features of the game: 3](#_Toc6855098)

[Similarities to other games: 4](#_Toc6855099)

[**Technical Issues:** 4](#_Toc6855100)

[Box2D physics engine: 4](#_Toc6855101)

[Implemented features: 4](#_Toc6855102)

[Shapes 4](#_Toc6855103)

[Bike components 5](#_Toc6855104)

[Bike rider components 6](#_Toc6855105)

[The Container and Water particles 7](#_Toc6855106)

[Improved Joints (Joint objects) 9](#_Toc6855107)

[Cloud system 9](#_Toc6855108)

[Particle system 9](#_Toc6855109)

[Game Controller and Terrain system 11](#_Toc6855110)

[Parameters tuned: 12](#_Toc6855111)

[**Reflection:** 12](#_Toc6855112)

[References 12](#_Toc6855113)

# **The Game Description:**

The game is a 2D fault-based racing game. The player controls a motorbike where the rider is sat on the bike wearing a container. The container is filled with particles. There are different levels for this game where each level consists of one or more obstacles. The goal for the player is to reach the finishing line without hitting the rider’s body on the ground and keep as many as particles in the container for the bonus points.

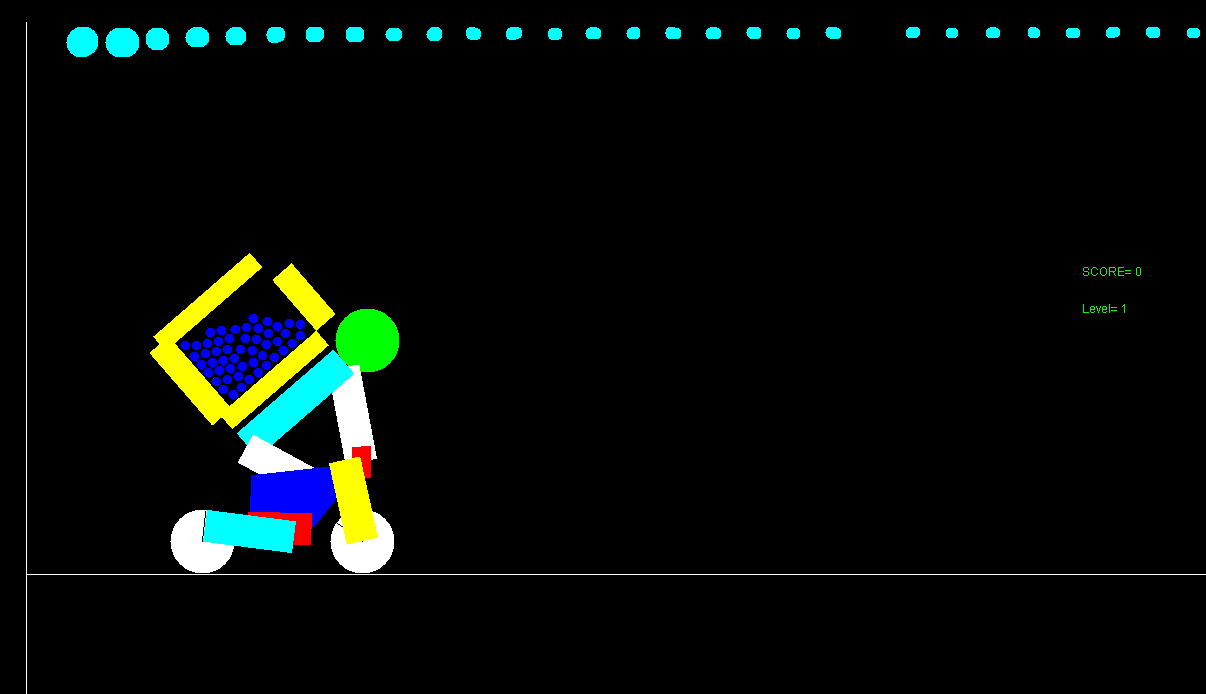


Figure 1 Game View

## Game controls:

The game controls are simple, the bike moves with right and left arrow keys which allows the bike to do a wheelie if pressed and held. It is worth mentioning that the player itself is very sensitive to force and motions. Therefore, a precision in pressing the arrow keys is advised in playing the game.

## Key features of the game:

The game consists of:

1. A game controller to load automatic generated obstacles for each level. Additionally, it holds all the calculation for the score and levels.
2. A motorbike, motorbike rider and a container all made dynamically and consists of multiple parts.
3. Water particles filled inside the container at the start of each level.
4. A particle system with fully customisable movements of particles in the region defined, this is used in the wheels of the bike to imitate dust particles as the wheel is span on the ground or obstacles.
5. Cloud system which is only used visually for adding some graphics to the game.
6. A complete enhanced joint managing class written to manage joints better. for example, if the bike rider hit the ground its joints will be detached and separated from the bike or if the container hit the ground its joint will be detached, and bonus score will not be given to the player.

## Similarities to other games:

The game was inspired by Trials Rising, where it is 2.5D fault-based racing game. The game was developed by *RedLynx* and *Ubisoft Kiev [1]*. The bikes are set to either race each other or beat the clock in the single player mode by riding through multiple obstacles. The game developed contains very similar features to this game as it is also a fault-based racing game, but the rider carries a container filled with particles where it manipulates the score calculation.

# **Technical Issues:**

## Box2D physics engine:

One of the main reasons Box2D was chosen for implementing this game was the easy creation of joints feature. Other physics features have made the implementation a lot easier and allowed more time to be spent on creating other game features. Some of the Box2D features were optimised in the implementation.

## Implemented features:

### Shapes

To make the whole implementation easier, an abstract class was implemented and named *Shapes*. This class is called when a shape is wished to be created. Each object contains a unique identifier named the user-data which was used further in the collision checking system. The class has two separate constructors, each constructor is defined to create either a circle or a polygon. If the polygon constructor was called it allows the developer to call a method named setPolygonPath to enter their desired polygonPath variable to create the desired polygonal shape.

The abstract class “*shapes”* holds *draw* and *update* method for drawing and updating the shape on the frame which can be overridden by their children classes.

Each shape will have their own assigned density which helps any rotation and natural movement of the shape. The circle shape’s density was calculated by:

The polygon shape’s density was calculated by:

### Bike components

To simplify and store bike components in a list together, an abstract class named *Bike* was created to separate its identity from other none bike shapes.

Bike component consists of:

1. BikeBody
2. BikeLimiters
3. BikeWheels

*BikeBody* contains an update function which will apply rolling friction to its centre according Figure 2.

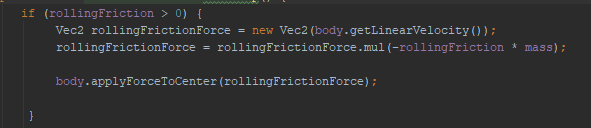


Figure 2 update function in applying rolling friction force

In the same function the player key controls are checked and will apply torque to the body which imitates the wheelie behaviour which is shown in Figure 3.

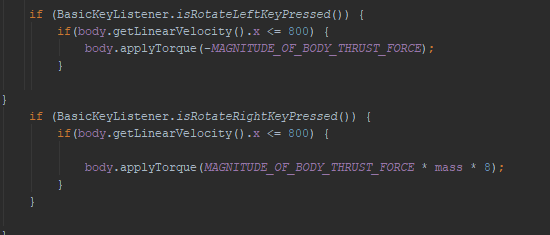


Figure 3 applying torque to the body to imitate wheelie

*Bikelimiters* are used as suspensions in the bike which will be explained more in the joints.

*BikeWheels* contains an update function similarly to BikeBody, which is used to roll the wheel on the ground and let it use the friction to the advantage of pushing the body forward.

### Bike rider components

An abstract class Rider was created for the bike rider components with the similarly to BikeBody class. It has classes extending its class which can be listed as:

1. *BikeRiderBody*: this class is used to create limbs of the rider such as arms and legs.
2. *BikeRiderHead*: this class is used to create the head of the rider.

The abstract class holds a method named Create\_the\_Rider. This function will create Rider’s body by classing BikeRiderBody and BikeRiderHead constructors and stores them in a Map data structure. The data structure will have string data as a key for distinguishing between each body part as shown in Figure .

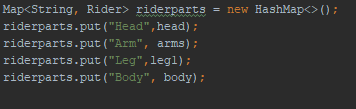


Figure 4 data structure used for storing body parts

In the same function three joints are defined as revolute joint with limitation to their angle. Each joint is used for the list below:

1. Arm to body
2. Head to body
3. Leg to body

### The Container and Water particles

The container class is defined in the *BikeBackpack* class. While creating the container’s shape, it was observed that Box2D is unable to create concaved polygons for example the water drops experimented with did not slide inside the concaved area as shown in Figure 5 .

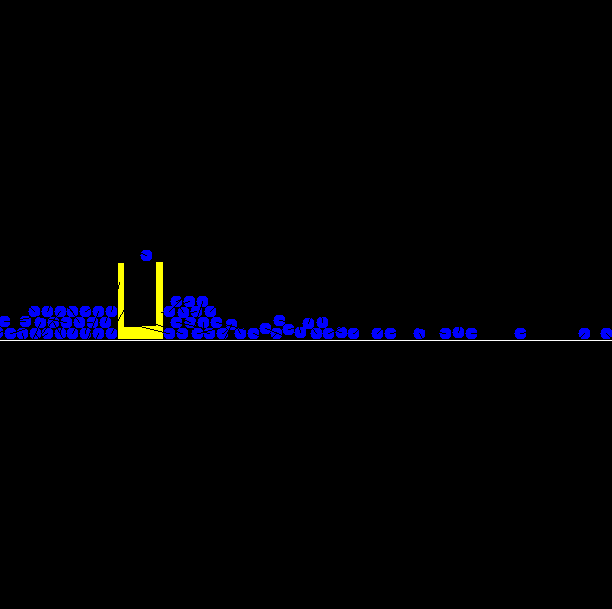


Figure 5 Box2D limitation in making concave shapes

Therefore, it was decided to create two “L” shaped polygons and use joints to attach two shapes together. The result is shown in Figure 6.

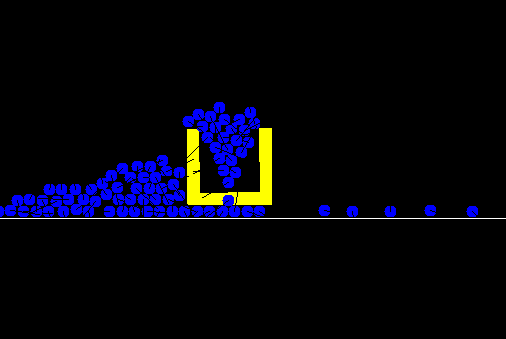


Figure 6 using two "L" shaped polygons to create the container.

Visually the improvement was not adequate, it had flaws where the particles did not distribute properly inside the container.

The final improvement was done on the shape where 4 rectangular shaped polygons were created and attached with joints. The result worked sufficiently in any rotations the container had. The results are shown in Figure 7 .

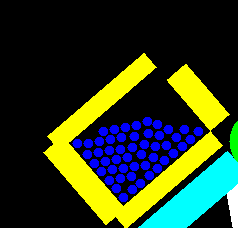


Figure 7 solved issue with concaved shape limitation.

The *BikeBackpack* class works similarly to Rider class where it stores all the container’s components in a map data structure. Additionally, the class calls the WaterDrop class to add the particles inside the container and stores them in a list data structure.

### Improved Joints (Joint objects)

One of the features of the game is bike rider’s body separating from the bike or the container detaching from the bike rider’s body. To make this feature possible, an improved joint must be made to store joints in a data structure and check their availability. Therefore, Imp\_Joint class was made where it stores the joint object, the name of the joint and it will hold their existence as a Boolean value. While the joints are made, each joint will be stored in a list data structure.

The total joints used for creating the bike unit (including the rider attached to the bike and container attached to the ride’s body) is 12. These joints consist of prismatic and revolute joint and weld joint. The distance joint was originally used to attach the bike’s limiters to create natural bike suspension mechanism but the Box2D did not allow the bike to move as a unit, therefore, prismatic joint was used instead. Each joint is checked with the utility class’s functions. The utility class will have functions to easily interact with the joints. It has functionalities to check the existence of the joint in the list, to return the joint object requested or remove the joint requested.

All the functionalities explained is used in *Apply\_contact\_rules* method which is a compact function where it calls the worlds contact listener and applies rules if two specific shapes collided in the world.

### Cloud system

This feature was used in a 3D form in the [2] project. The feature was optimised for two dimensional clouds to resize over time and reach the maximum size and scale down until they reach the minimum size then the object was moved to the front of the clouds to imitate the clouds movement.

### Particle system

Particle system is an important visual component of the game. The system is used and created when the bike’s wheel applies a specific amount of force to the ground.

The particles will have an origin point and a destination point where they move towards their destination. Inspired by Unity engine [3], the system enables the developers to pass in a radius to the system and the particles will be start from one origin point and travel to a point on the radius given. This feature was called the *Cone* feature in the particle system where particles are generated and will travel inside a cone until they reach the surface of the cone.

The system will use a triangle to work similarly to [3]. The system is shown in Figure 8.

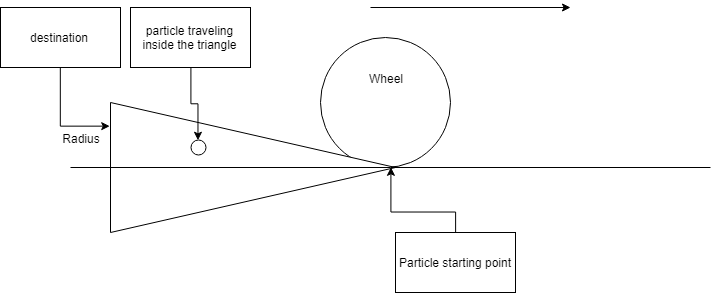


Figure 8 particle system used in wheel

To get the destination point, a complicated calculation was introduced. Originally, the system will have the original point A in the world, the length of the triangle’s edge where the particle will travel to a random point on that edge and the destination point where the edge is placed on called B.

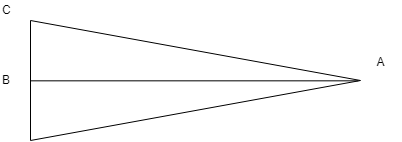


Figure 9 triangle visualisation for finding the destination point on the edge of the triangle.

To find the pint on edge of BC, it is required to find the length of the triangle first. Therefore, length of AB was found by:

The length of BC was found by:

The length of AC was found by:

The point’s coordination on the edge was found by:

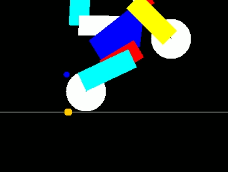


Figure 10 the brown particle is generated by the particle system.

### Game Controller and Terrain system

Game controller will hold all the score calculation and the level creation. Each level is made dynamically by calling Terrain class. There are five different obstacles made shown in Figure .

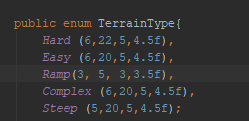


Figure 11 Terrain type

Each enum value will hold an important information about the shape of the obstacle where it will allow them to be created accordingly.

### Parameters tuned:

Parameters such as joints limitations where all set to a suitable value for them to act naturally and make the rider have room to move while it is attached to the bike.

One of the problems with balancing the person correctly on the bike was the water mass in the container where a smallest mass available was given to the water particle for it not to pull the rider’s body towards itself.

# **Reflection:**

One of the biggest problems about the project was the Box2D features which was incomplete according to the conversion it had from C++ to Java platform. It has limited number of features which in my case was the water behaviour. The Java version of Box2D lacks the particle system for the water behaviour and I could not apply any self-made behaviours to the Box2D shapes which made the game crash. I have made *Water* and *WaterParticle* classes with full functionality of water particles to have natural behaviour where they can join and divide. But unfortunately, it did not work with the Box2D physics and it worked fine with Vec2D engine made by Michael Fairbank therefore, I removed it from the project folder to work on in the future. I have created an optimised way to manage joints and worked well in the contact listeners which made my game possible and there could be more done on it to manage the joints parameters externally. The particle system is my favourite part of the program, I have created a reasonable and enough system to work perfectly with my game.

# References

[1]"TRIALS RISING", *Ubisoft.com*, 2019. [Online]. Available: https://www.ubisoft.com/en-gb/game/trials-rising. [Accessed: 22- Apr- 2019].

[2]S. Hamid, M. Pisheh Var and G. Walajahi, "Group Project Final Report", 2019. [Accessed 22 April 2019].

[3]U. Technologies, "Unity - Unity", *Unity*, 2019. [Online]. Available: https://unity.com/. [Accessed: 22- Apr- 2019].

**Apportioning credit:**

It can be mentioned that classes such as BasicView, BasicMouseListener, BasicKeyListener, JEasyFrame, AnchoredBarrier\_StraightLine and AnchoredBarrier the abstract class were taken from Michael Fairbank’s lab assignments. The Shapes abstract class was inspired by the BasicPolygon and BasicParticle classes from Michael Fairbank’s lab assignment, but significant changes was applied.

Classes listed below were created by me:

1. Bike
2. BikeBackPack
3. BikeBody
4. BikeLimiters
5. BikeRiderBody
6. BikeRiderHead
7. BikeWheels
8. Clouds
9. CONSTANTS
10. GameController
11. Imp\_Joint
12. JointSource
13. ParticleSys
14. PhysicsEngine
15. Rider
16. Shapes
17. Terrain
18. TerrainUtility
19. UTIL
20. WaterDrop