Virtual Reality
Bowling Game using leap motion controller

Author Łukasz Słowik
Winter Semester 2016/2017
Novi Sad, Serbia

Project Requirements

- Leap motion controller
- The newest unity version (https://unity3d.com)
- Leap motion example (https://github.com/leapmotion/LeapMotionCoreAssets)
- At least 5GB free space on hard disc

Minimum System Requirements

Windows® 7/8 or Mac® OS X 10.7
AMD Phenom™ II or Intel® Core™ i3/i5/i7 processor
2 GB RAM
USB 2.0 port

In this project was used Unity 5.4.2f2

1. Preface

Main goal for project is to make an good example of showing Virtual Reality. Nowadays technology has grown very high. There exist various examples on many Virtual Reality devices. One of them is leap motion controller which is device for hands recognition.

The project is made using leap motion technology.

2. Leap Motion Controller

Leap motion controller

Picture copied from http://store-eur.leapmotion.com

To use the leap motion controller we have to connect device to a computer via usb cable.

Next step is to download the “orion sdk” from the producer site.

https://developer.leapmotion.com/get-started

To download we should click on Orion beta button.

After finished installation process, we will see on the device shining two red lights. That means that drivers was installed properly and device is ready to use.

Before starting making project we should firstly test if device works.

To do that we need appropriate example. We can use the newest example from the leap motion site, but instead of it, we will use the older example saved in github repository. The reason for that is that there are more useful examples and also script that we will use later.

The link of the example

https://github.com/leapmotion/LeapMotionCoreAssets

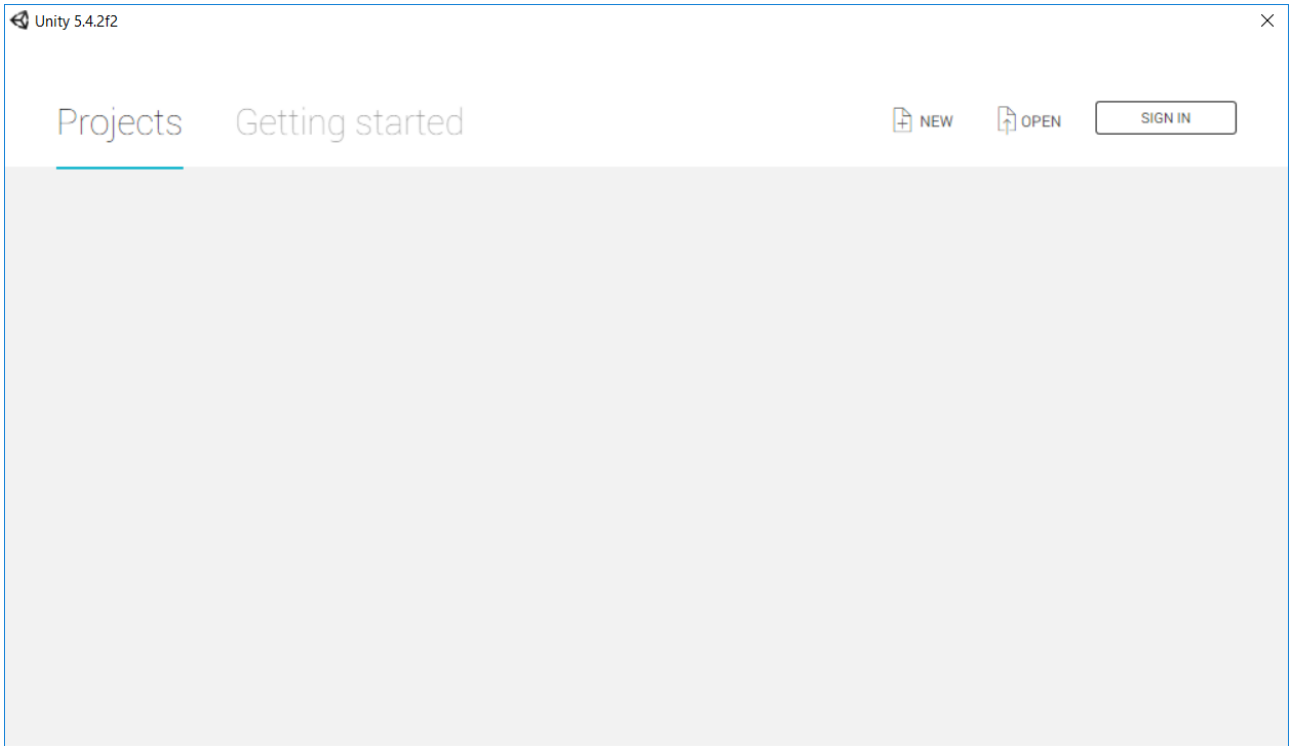
We can get it simple by downloading ZIP file or cloning repo using git.

If we have git installed we can simply clone repo by following command :

git clone https://github.com/leapmotion/LeapMotionCoreAssets

Next step is install Unity I recommend to install only proposed Unity features. For this kind of installation we need 2.1GB space on hard disc.

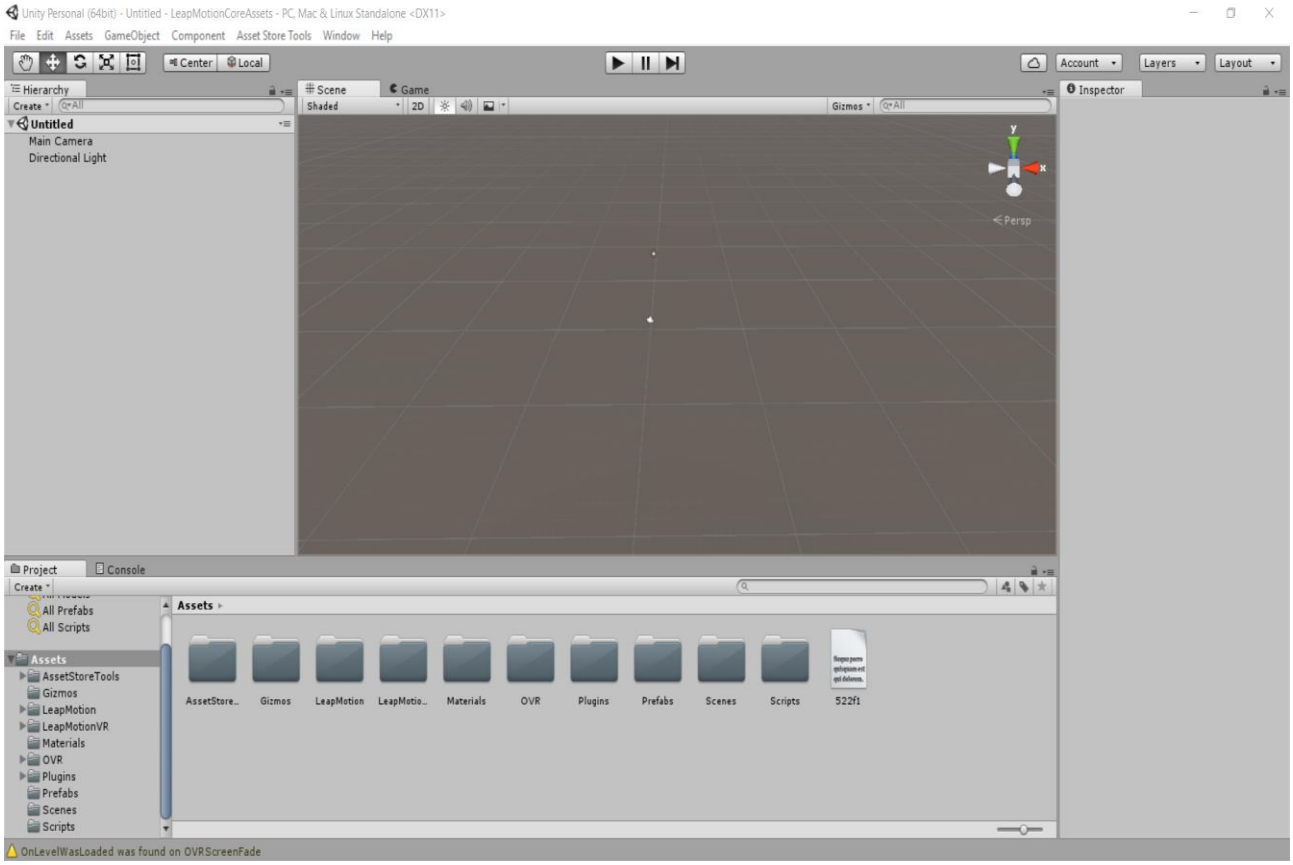
After installation we can run Unity Editor, welcome view should looks similar like on presented picture below.



To open LeapMotionCoreAssets example we use “OPEN” button and then we choose the path where we store this example.

Eg. D:\Unity\LeapMotionCoreAssets

After we select the project, we should see unity editor with asset files



This is Unity editor, It consist from couple of views.

Gameobject – kind of object with game usage properties

Hierarchy :

On this view there are game object that can be used on scene.

Scene :

Here we can use the game object and make them visible in game.

Game :

After pressing play button game view is turned on, based on scene, objects and scripts.

Inspector :

View that shows editable details of selected object/option.

Project :

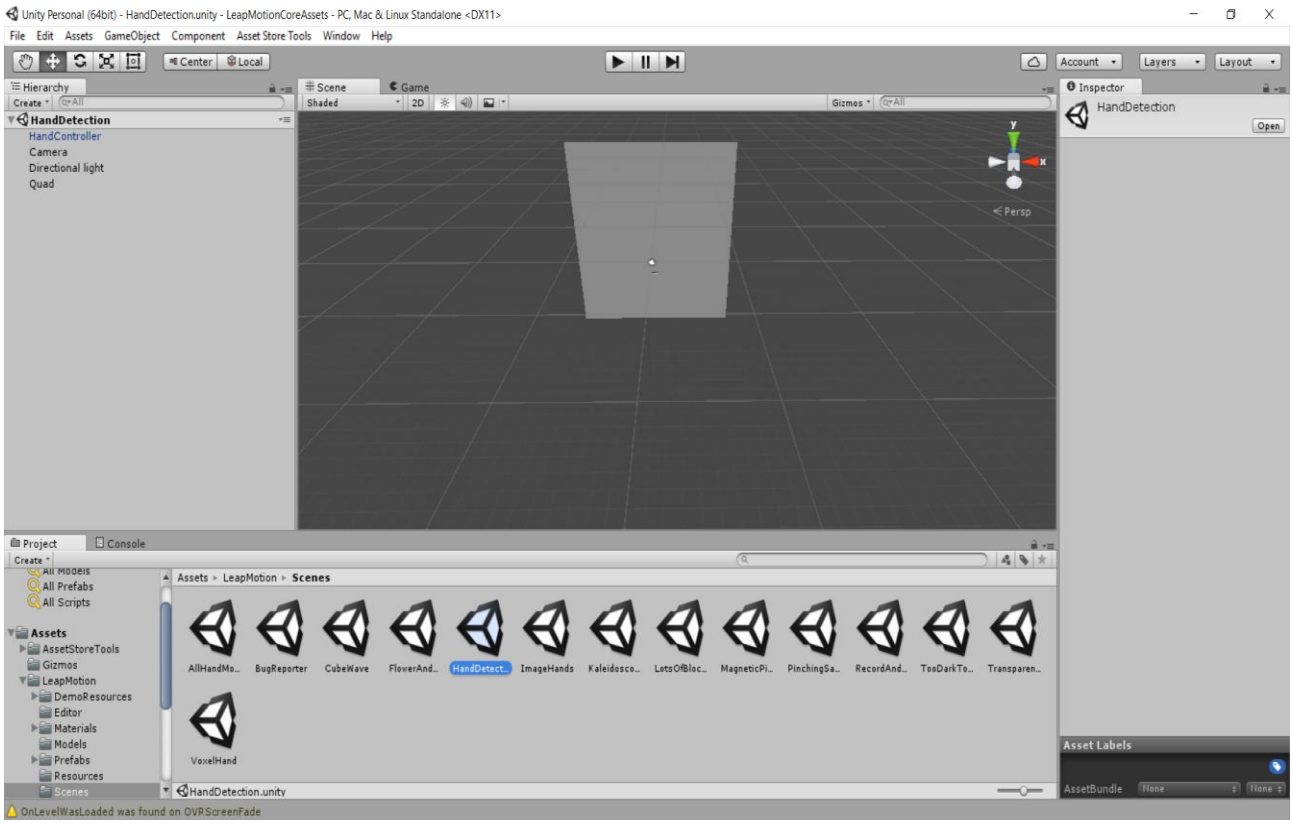
View that shows folders and files.

Console :

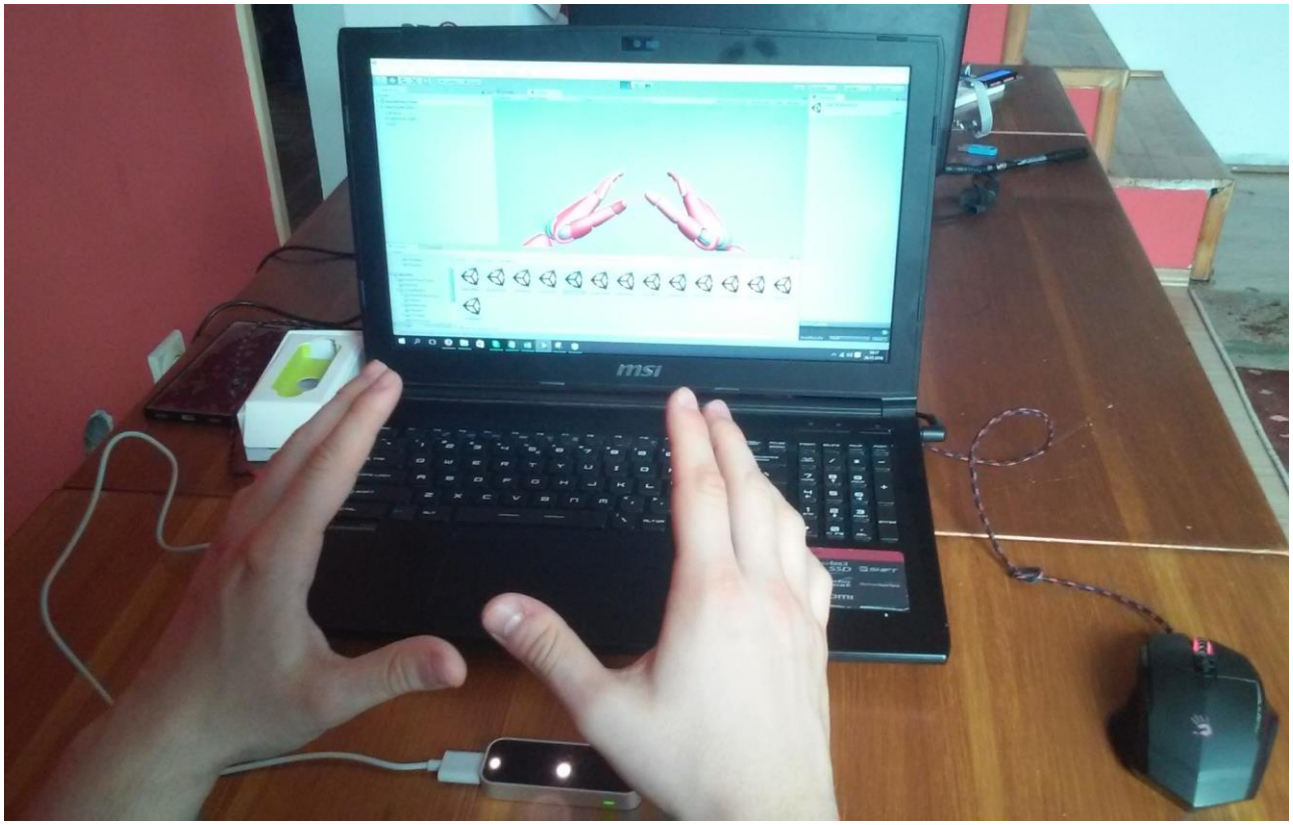
Console of running game.

To run the test device example we select Assets -> LeapMotion -> Scenes -> HandDetection

After select we should see following



Then we press play and when we put our hands over device we should see visualisation of them.



Ok, if everything has worked well, we ran our first virtual reality example.

3. BOWLING GAME

To start bowling project, we create folders Scenes, Scripts and Prefabs in Assert folder. That's for making separation from example project. Then we should copy example scene FlowerAndRobot from Assets -> LeapMotion -> Scenes to Assets -> Scenes. We use that scene because there is already configured hard controller and grabbing option, what's would be useful in bowling game. We can delete the FloweryPlant object from hierarchy, there is no need to use it.

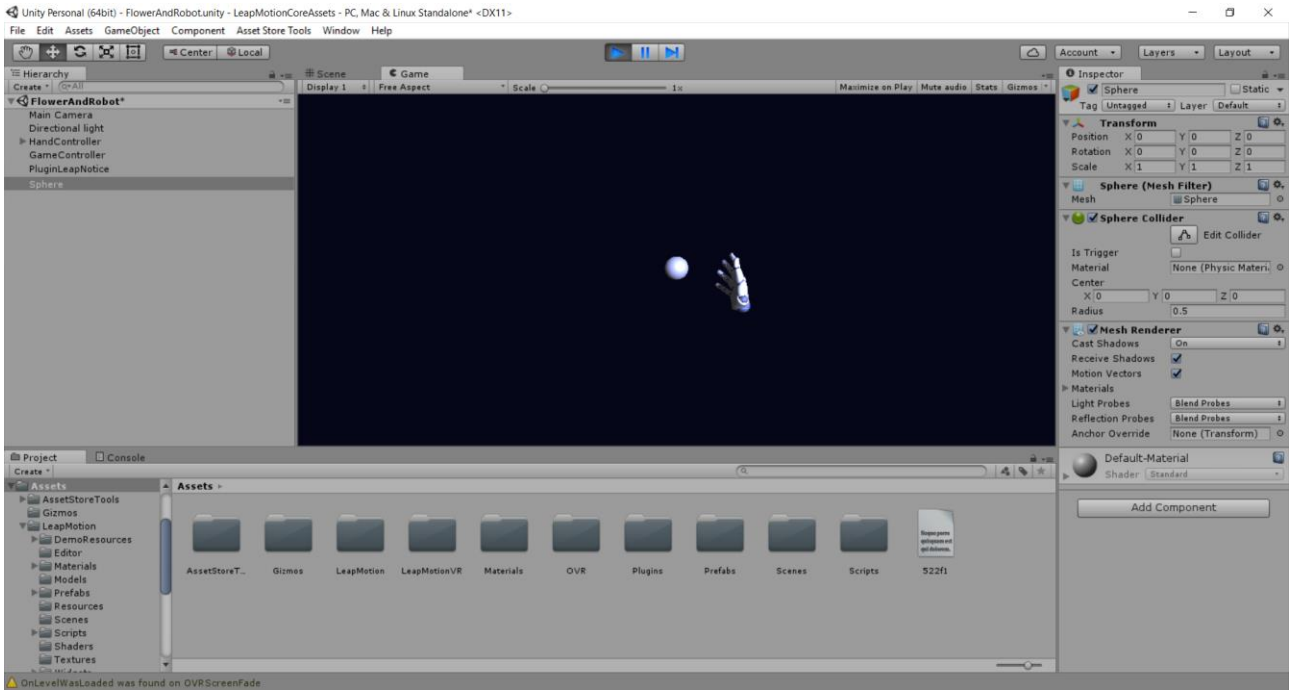
We need to build the scene for bowling. To do that we should imagine how to do it in the simplest way using objects available in Unity.

Firstly we will create a ball, to do that we will use sphere object. Right clicking on hierarchy will open menu with different option, but now we choose

3d Objects -> Sphere

This operation should create a Sphere object in hierarchy. Afterwards we select sphere and change in inspector position to x : 0, y : 0, z : 0. Then run game and move your hand close to ball.

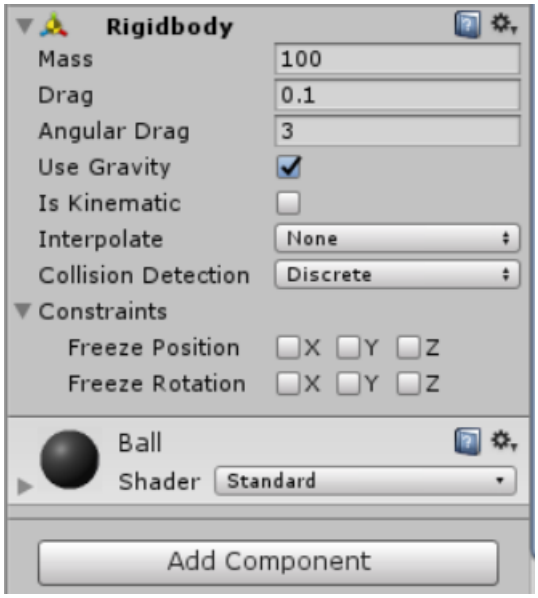
After this steps we should see something like that below



As we can see now, there is no colour on ball, we can simply add colour to our sphere by creating new material. Let's go inside Materials folder, right click and Create -> Material. Rename it to Ball, next in inspector we can change for example Albedo effect and choose one colour from pallet. After setting color grab material on Sphere object. Sphere should be now coloured.

By the occasion let's change also Sphere object name to Ball to make names more bowling familiar. After ran game we could see, that our ball hadn't any gravity force, what is obvious in our real world, to make object physical dependent, we have to add to it component Rigidbody.

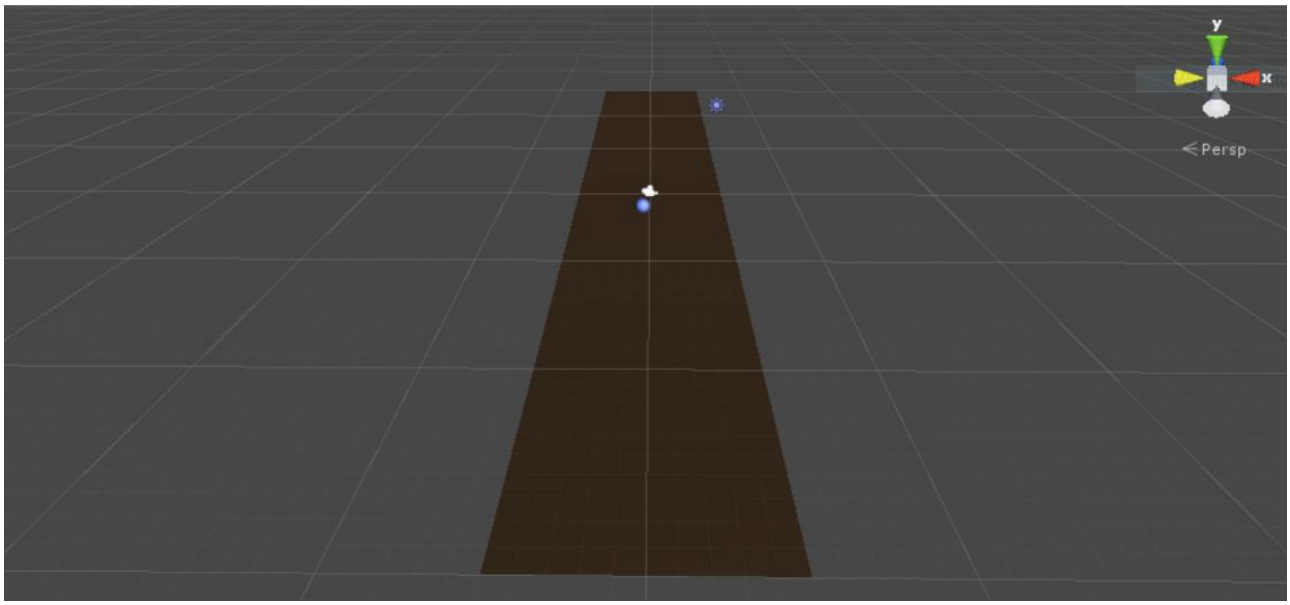
Let's do that by clicking in Inspector Add Component -> Rigidbody. After this step a new component should be added in Inspector Tab. To make object gravity dependent, option "Use Gravity" inside Rigidbody have to be marked.



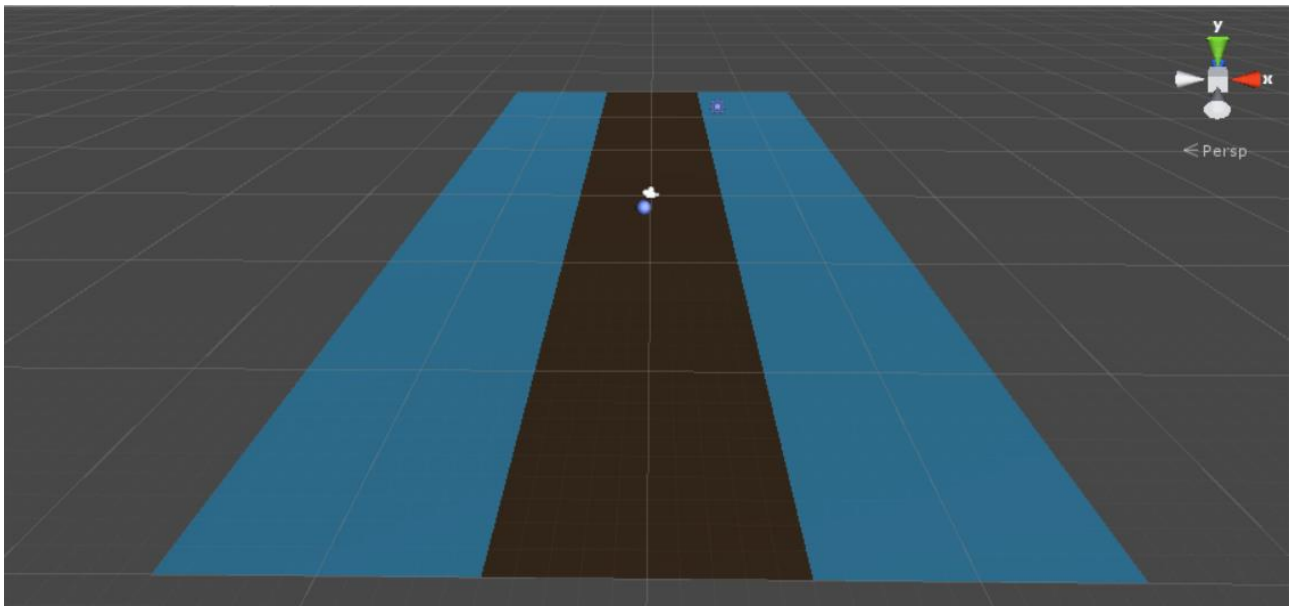
Picture shows the Rigidbody component from Inspector and Ball Material.

In bowling there is not only ball, we have to create also playing ground. For this porpoise we create 3 planes; one as truck and 2 for both truck sides, left and right.

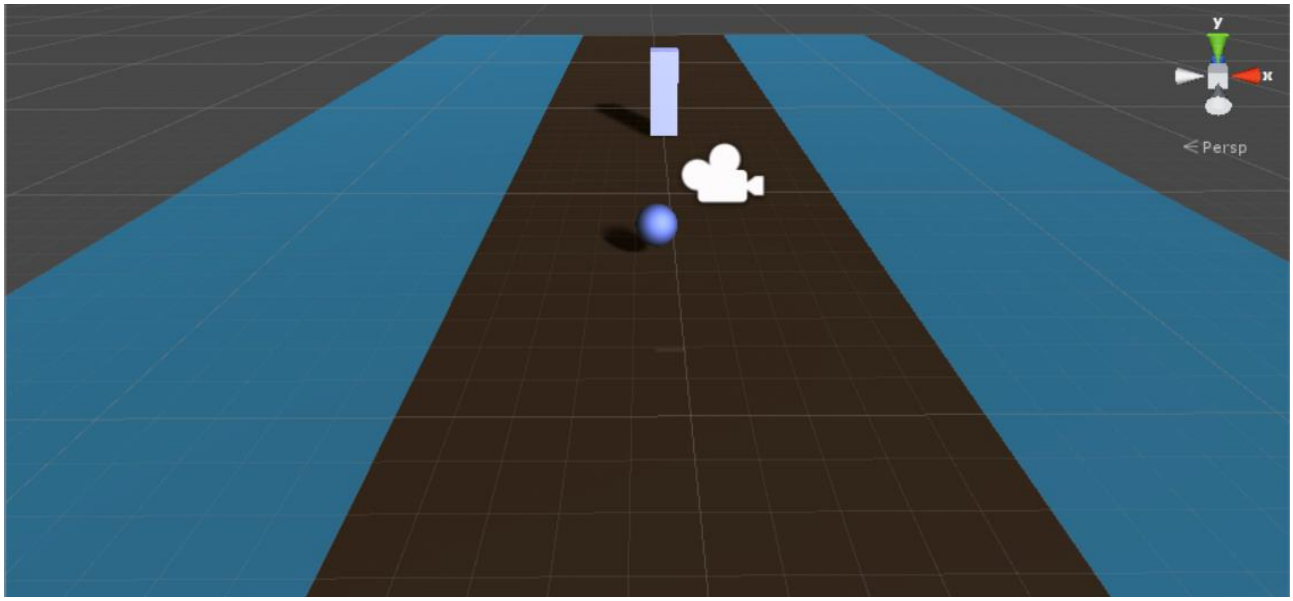
On hierarchy, right click -> 3d Object -> Plane, this object going to be our track, the place where we can push our ball. Let's rename it to Truck, and in transform component, set position to x : 0, y : 0, z : 0 and scale to x : 1, y : 3, z : 6. This scale will provide us view as real track. Create a material with brown colour and attach it to our track. This process should lead us to that



We have already 2 objects, ball and track, ball will be grabbed by our virtual hand though track we see. We need to create 2 planes. Procedure of creating is also the same and scale. Different attributes we have is position and name. However, name of this 2 new creating objects will be the same. In this case is good to use prefabs. Prefabs are copy of object that you can hold in folder. Very useful when you have to create the same object few or more times with small change or no change, in our case only position will be changed. Let's copy the track object in Hierarchy and then paste it in the same place. Denote the copy object as RestArea. Create a material with blue colour and attach to the object. Now grab the object from Hierarchy to the prefab folder. When you click on prefab you create, you should see properties of the prefab, now there are the same as our object in Hierarchy. Grab the prefab back to Hierarchy, our object should be copied, but the difference between ordinary coping and prefab grabbing is we have save state of our object using prefabs. Change position of our RestArea's, left to x : -10, y : 0, z : 0, right to x : 10, y : 0, z : 0. The view we now see



This is time for creating pins, in bowling pins is one of biggest requirements, speaking shortly the game is ball, track and pins, goal of every bowling match is sliding ball though track in that way we get down all the pins. Our Pin will be cube in stick shape with RigidBody property. Let’s add to Hierachy new object, right click on Hierachy -> 3D Object -> Cube. Rename it to Pin and scale it to to x : 1, y : 3, z : 1. Attach to the object component RigidBody. Then grab the object to the Prefab folder. From closer point of view, we should see



In bowling in each game there are 10 pins, set in specific position. We will create an empty object PinGroup to store Pins together under one object. To create it in Hierarchy, right click -> Create Empty. Rename new created object to PinGroup. Copy 10 pins prefabs and grab them on PinGroup object, pins should be now seeing under PinGroup object. Rename each pin as presented on picture

In formula it can be presented as for loop

For I := 1 to 4 do

  For J := 1 to 5 - I do

    Next Pin Name := Pin + I + J

  END

END

+ detones string concatenation

:= denotes assigment operation

Now we have to set position for each pin, under are presented cordinates for each object that should be changed.

Pin Group

x = 0

y= 0

z = 17

For each pin y condinate will be there same, because every pin is on the same height

y = 1.5

For first line {1, 2, 3, 4}

x = {1 : -3.5, 2 : -1.2, 3 : 1.2, 4 : 3.5}

z = -3

For second line {1, 2, 3}

x = {1 : -2.5, 2 : 0, 3 : 2.5}

z = {1 : -4.5, 2 : -4.7, 3: -4.5}

For third line {1, 2}

x = {1 : -1.3, 2 : 1.3}

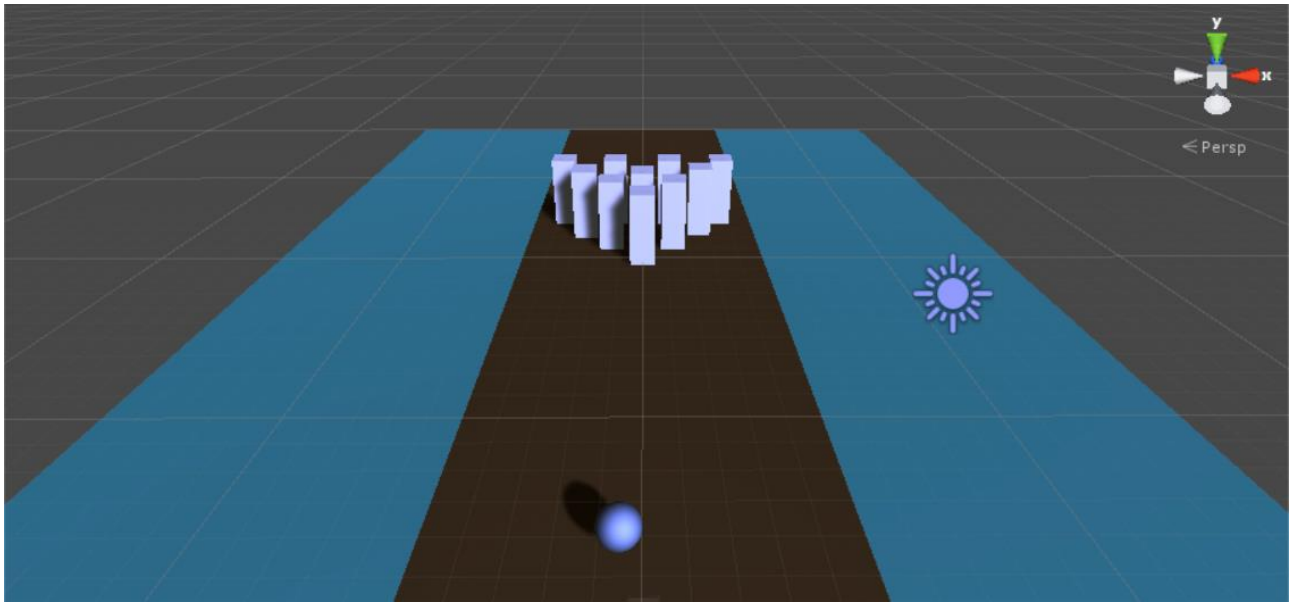
z = -5.75

For fourh line {1}

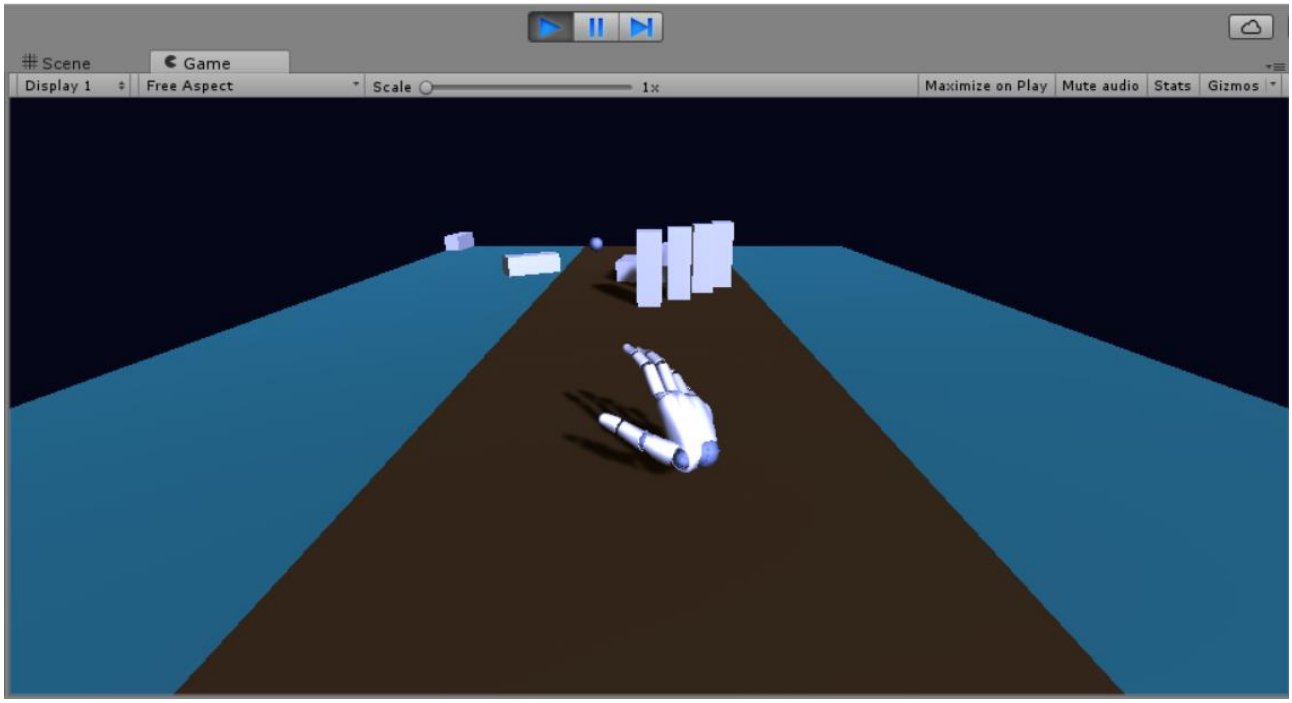
x = 0

z = -7.2

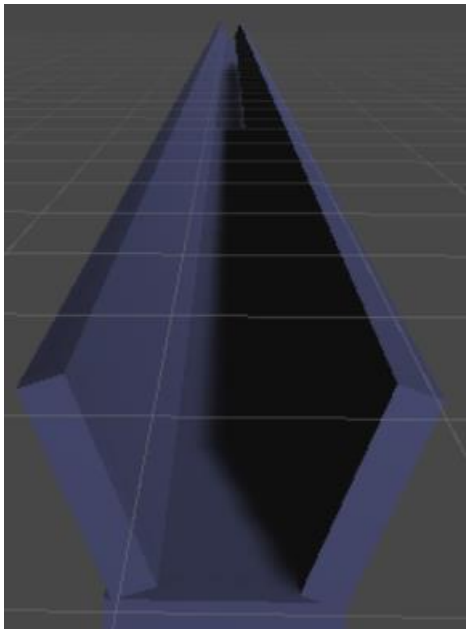
After correctly setting position, we should see, view as below



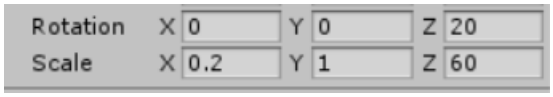
This is the time, when we can try basics of bowling; using ball to fall some pins. Picture below shows play test from current state of game.



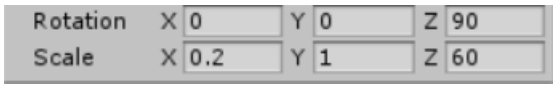
To make bowling more realistic we will add gutters, between track and rest areas. Create an empty object and rename it to Gutter. Then create a cube and scale it to x : 0.2, y : 1, z : 60. This shape makes similar size to our track and rest area. Attach this object to Gutter. Next step is to copy the object and rotate it z : 90º (Rotation z : 90), and original object z : 20º. Move the object on left corner of object below and copy it again. Now for the new object we reverse the rotation from 20 to -20, and then moving that to the right corner of object below. This action should make a kind of Gutter object created by 3 cubes.



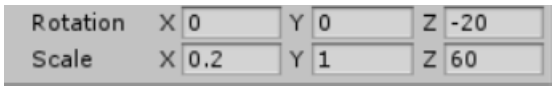
First Cube Rotation and Scale



Second Cube Rotation and Scale

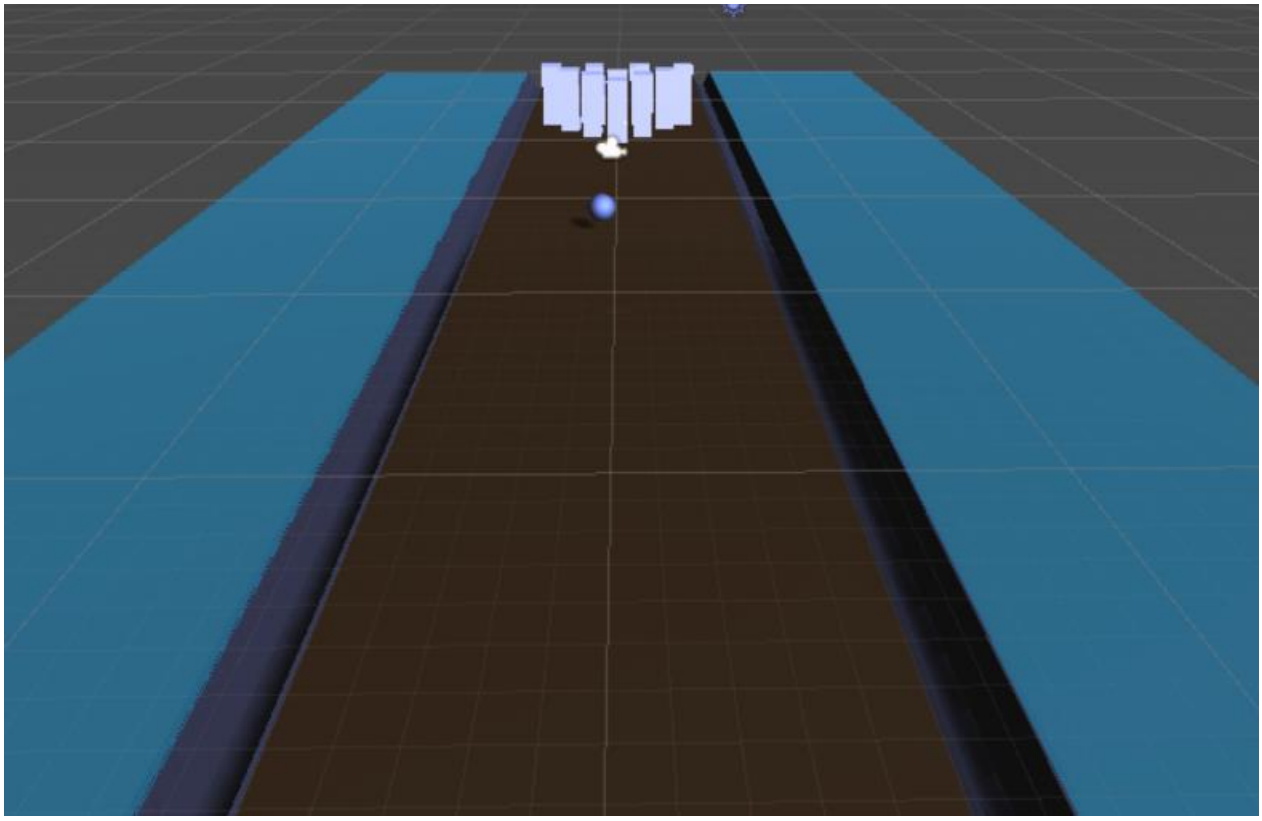


Third Cube Rotation and Scale

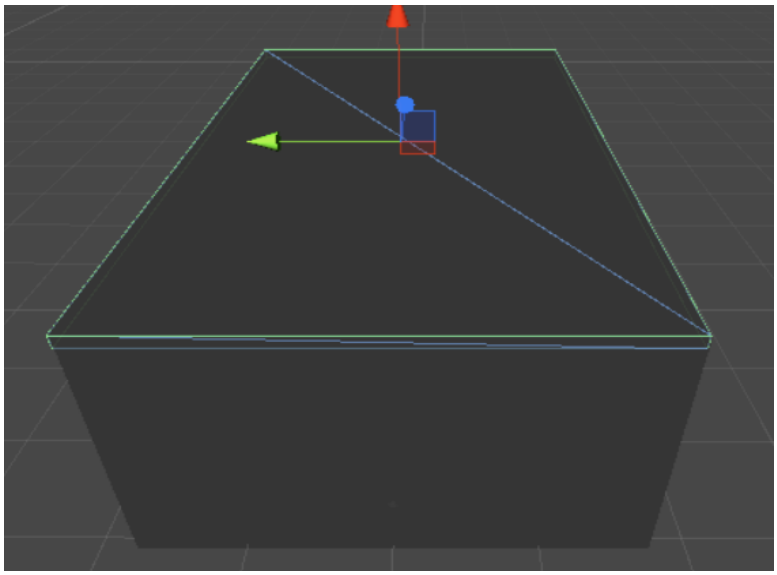


The last thing to do with gutter is putting it between, track and rest areas. It is also good idea to make a prefab of this object.

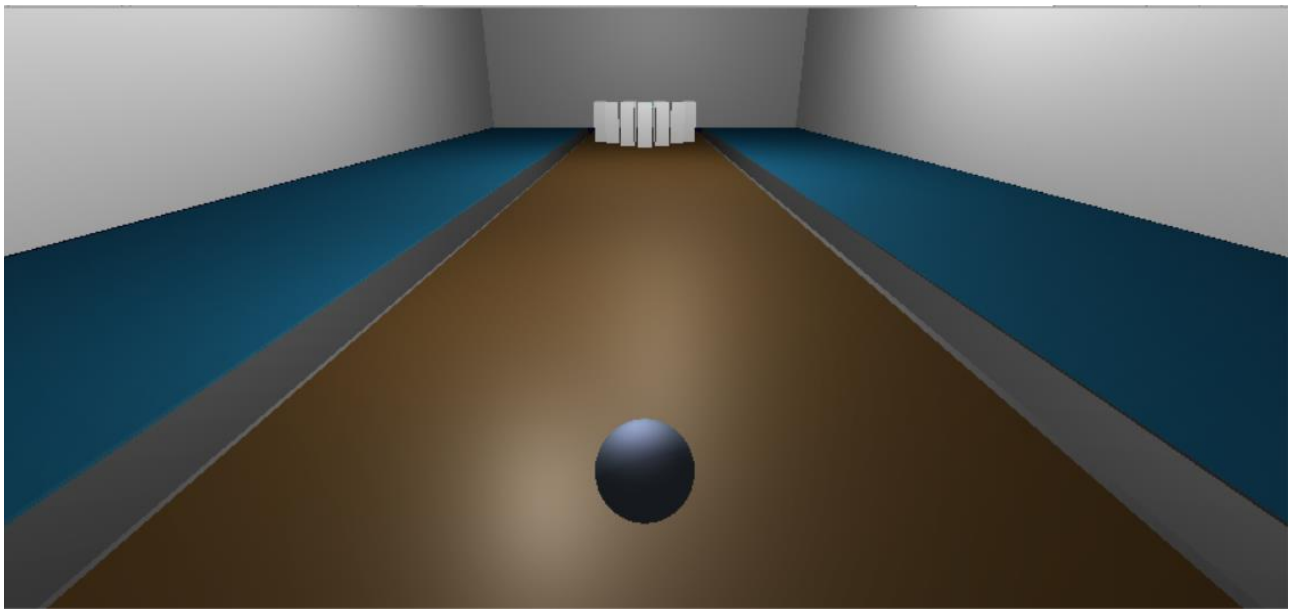
Finally it should looks like below



Closing game in box, that's one of requirements, if we play game, we will see that's possible to throw the ball out of the created map (The grey fields beyond), to avoid it, we should close our game in box, to do it, we will create cubes in every map side. I will not go into details in this case, it's very similar step to creating gutter, on every side of our created field we match box, let's say with height 10, we don't need to create a floor, because of existing our track, gutters and rest areas which covers this place. View outside after boxing should looks like below

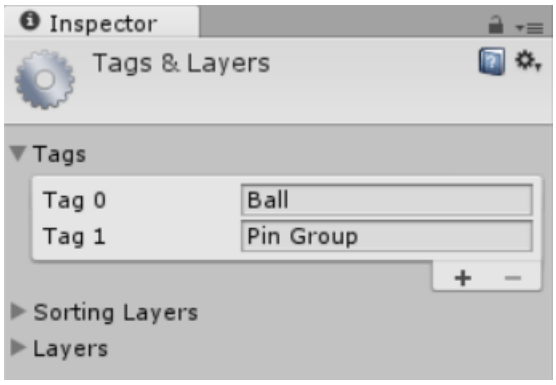


And inside boxes, game, in my opinion seems to be more fabulous



I didn't mention, but I also have added 2 point lights, to make game more lit. To add directional light object, right click on Hierarchy -> Light -> Point Light. In this example I set range of light to 50, and put it above track near ball and pins.

Let's now write some game useful scripts, namely for restoring position of ball and pins. After every round in the game, we will have to restore our ball and pins to them origin position to begin new round. Go to hierarchy and create an empty object with name GameController. Then go to scripts folder and create new script with the same name as object. Double click on the script, and after it, editor of code should be seen. But one thing before writing code, we have to add Tags to our objects to easy find them. To do that go to hierarchy click on Ball object, and on the top of Inspector find Tag. At beginning Tag should be "Untagged", click on on the property, and then "Add Tag…", then click on "+" and add Tags with name "Ball" and "Pin Group".



Go back to Ball Object, and select "Ball" Tag. Then go to PinGroup object and select "Pin Group" Tag.

Now you can go back to the code editor and paste following code

```
using UnityEngine;
using System.Collections;

public class GameController : MonoBehaviour {

    public float xpos = 0f;
    public float ypos = 0.479f;
    public float zpos = -25.83f;
```

```
    // Update is called once per frame
    void Update () {
        if (Input.GetKeyDown(KeyCode.R))
        {
            RestartBallState();
        }
    }

    public void RestartBallState()
    {
        GameObject existingBall = GameObject.FindWithTag("Ball");
        if (existingBall != null)
        {
            Rigidbody rigidbody = existingBall.GetComponent<Rigidbody>();
            rigidbody.velocity = Vector3.zero;
            rigidbody.angularVelocity = Vector3.zero;

            existingBall.transform.position = new Vector3(xpos, ypos, zpos);
        }
    }
}
```

Then you can save the file and attach script to the GameController object. Generally script per every frame checks if “R” button on keyboard was pressed, and if pressed looking for object with Tab “Ball”. We obviously know that is our Ball object, and then remove all forces from object, to avoid situation, when we set position of ball and it’s still rolling somewhere. To do that, we have to get RigidBody component,, which is responsible for holding forces, and change properties, velocity and angular velocity. And then setting position to the start one, defined by variables xpos, ypos and zpos. When you run game, you can now, always restart position of ball by simply pressed “R” button.

But what with Pins, we have added Tag to the PinGroup object, but what more? With pins it’s like more complicated, we have to attach script to each pin that resetState, and then make loop from GameController and call each script from each pin. To do that, we have to create new script named PinController. Inside the script paste following code.

```
using UnityEngine;
using System.Collections;

public class PinController : MonoBehaviour {

    private float pinX;
    private float pinY;
    private float pinZ;

    void Start() {
        Vector3 pinPosition = transform.position;
        pinX = pinPosition.x;
        pinY = pinPosition.y;
        pinZ = pinPosition.z;
    }

    public void RestartPinState()
    {
        RemoveForce();
        RestartPosition();
    }

    private void RemoveForce()
    {
        Rigidbody rigidbody = this.GetComponent<Rigidbody>();
```

```
        rigidbody.velocity = Vector3.zero;
        rigidbody.angularVelocity = Vector3.zero;
    }

    private void RestartPosition()
    {
        transform.position = new Vector3(pinX, pinY, pinZ);
        transform.rotation = new Quaternion(0, 0, 0, 0);
    }
}
```

The script, works similar, but showing other approach. At start of the game, we store pin position as vector <x,y,z> in separated variables. When we call RestartPinState() in removes forces like in last cases, and then restart position stored in variables, but this time without ours fixed values. Save the script and attach it to each pin in PinGroup object. It's relevant to have it on every pin, because of separated pin effect. After it go to GameController script and edit it to following

```
using UnityEngine;
using System.Collections;

public class GameController : MonoBehaviour {

    public float xpos = 0f;
    public float ypos = 0.479f;
    public float zpos = -25.83f;

    void Update () {
        if (Input.GetKeyDown(KeyCode.R))
        {
            RestartBallState();
        }
        if (Input.GetKeyDown(KeyCode.Y))
        {
            RestartPinsState();
        }
    }

    public void RestartBallState()
    {
        GameObject existingBall = GameObject.FindWithTag("Ball");
        if (existingBall != null)
        {
            Rigidbody rigidbody = existingBall.GetComponent<Rigidbody>();
            rigidbody.velocity = Vector3.zero;
            rigidbody.angularVelocity = Vector3.zero;
            existingBall.transform.position = new Vector3(xpos, ypos, zpos);
        }
    }

    public void RestartPinsState()
    {
        PinController[] pinsTransform = GameObject.FindWithTag("Pin
Group").GetComponentsInChildren<PinController>();
        if (pinsTransform != null)
        {
            foreach (PinController pinTransform in pinsTransform)
            {
                pinTransform.RestartPinState();
            }
        }
    }
}
```

Now we can restore the state of pins simply but pressed “Y” button. As we see in script it finds the object with “Pin Group” Tag, and to every child (Object under the PinGroup object; every pin), calling the RestartPinState method.

We have to also implement one matter event, which is calculation if our pin is moved down. For this, we will look into our rotation, we will assume that exist a margin of rotation over pin is for sure down. We know that rotation of x or z equal to 90 is when pin is totally on the floor. y factor makes rotation only by pin height. My idea is following, either angle of rotation factor x or z is greater or equal 50, we mark pin as Down. I tested the idea few times and was no negatives pin down marking. So I assume it’s enough correct. Open PinController script and add following code

```
public class PinController : MonoBehaviour {

    private float pinX;
    private float pinY;
    private float pinZ;

    private float rotationMargin = 50;

    public bool pinDown = false;

    // Use this for initialization
    void Start() {
        Vector3 pinPosition = transform.position;
        pinX = pinPosition.x;
        pinY = pinPosition.y;
        pinZ = pinPosition.z;
    }

    void Update()
    {
        valuex = transform.eulerAngles.x;
        valuez = transform.eulerAngles.z;
        if (!pinDown &&
            ((valuex >= rotationMargin) && (valuex <= (360 - rotationMargin)))
           || (valuez >= rotationMargin) && (valuez <= (360 - rotationMargin)))
        {
            pinDown = true;
        }

    }
```

We store every factor of rotation, and check to of them in every tick of the game. If pin is marked already down, only this case would be checked.

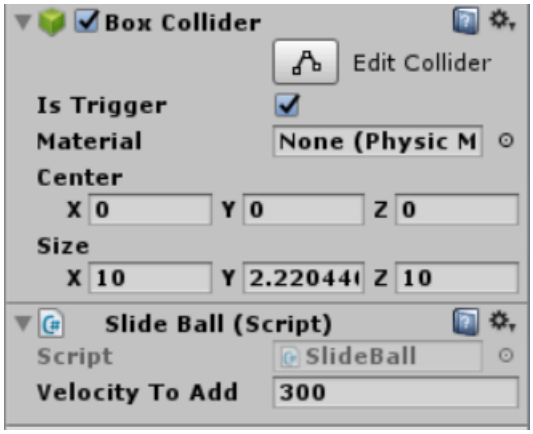
We should also change the RestartPinState method in the PinController, to have again pinDown state as at start.

```
    public void RestartPinState()
    {
        RemoveForce();
        RestartPosition();
        pinDown = false;
    }
```

Sliding floor is another interesting thing I my opinion required in this game. That means that if we push our ball on for example track, the track floor should be slideful and roll our ball in specific direction with small beginning force. For this porpoise create new script named SlideBall and inside write following code

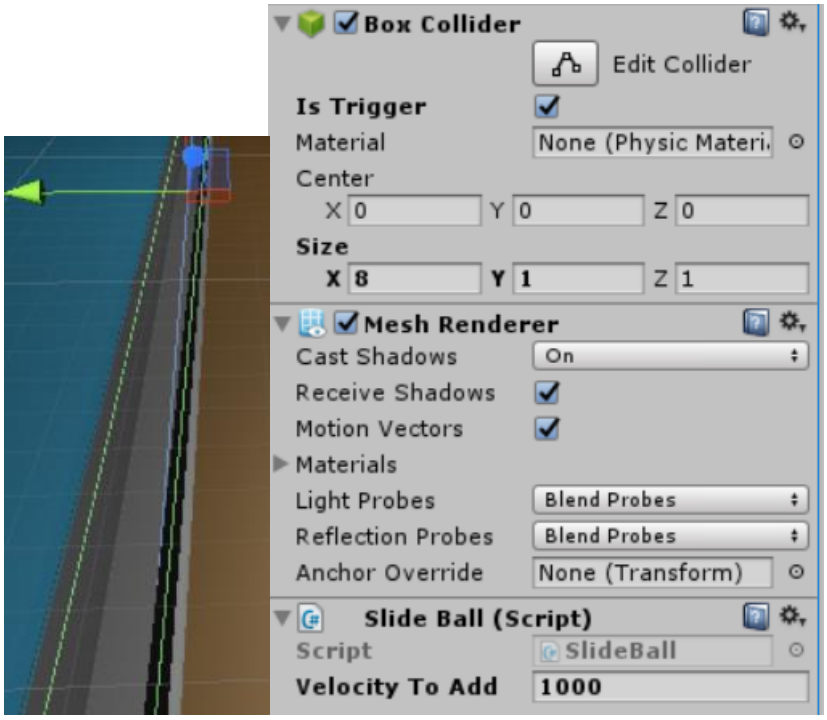
```
public class SlideBall : MonoBehaviour {

    public int velocityToAdd;

    void OnTriggerStay(Collider other)
    {
        if (other.tag == "Ball")
        {
            other.gameObject.GetComponent<Rigidbody>().AddForce(Vector3.forward *
            velocityToAdd);
        }
    }
}
```

Then attach the script to the Track object. You see that in the script is public int field velocityToAdd without fixed value, however you can add this value from Unity Editor. We use function OnTriggerStay, that's one of special functions that requires triggering. To active triggering we have to mark box "Is Trigger" inside Track object.



Let's also set the velocity to 300, if you run game and push the ball on track, you will see the slide effect. In my opinion it's also good idea to attach it to our Gutters, select the cube down, from our gutter, same as presented on picture. I gave additional names to them to be more intuitive.

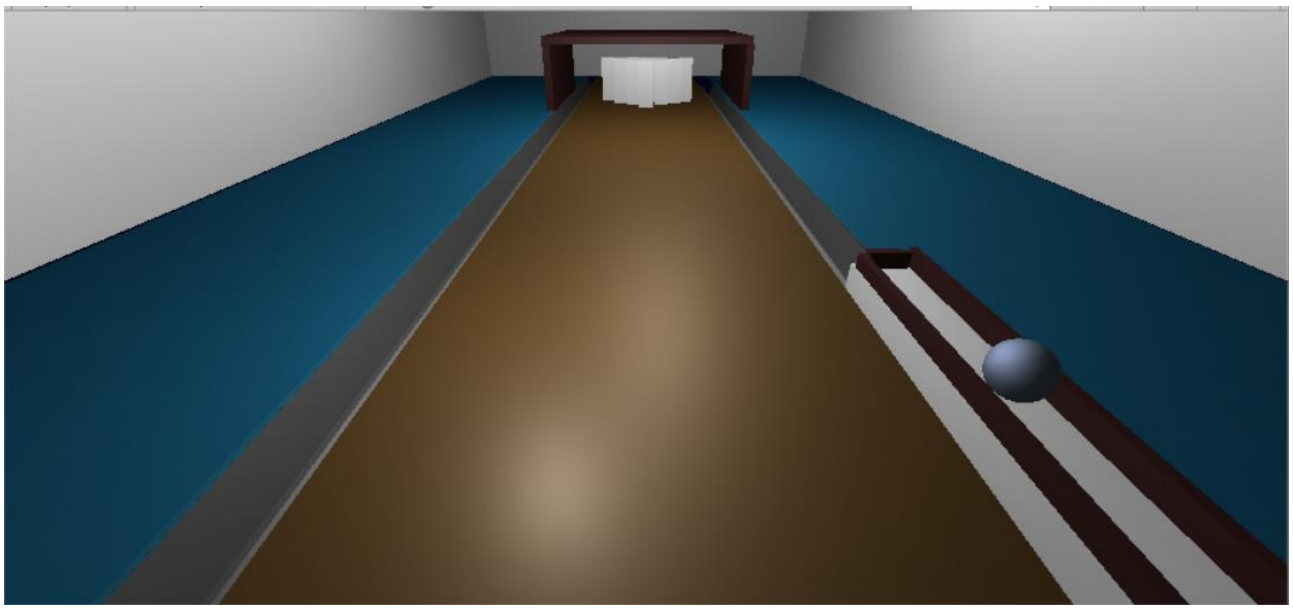
Attach the script to cube down, mark “Is Trigger” property, and change Velocity To Add to 1000. There is one problem, we created the Gutters a little bit smaller that we wan’t, so we have to increase the Collider size to detect correctly the ball, please change Y size of Collider to 8. That solve the problem.



Do the same thing to the second gutter.

After this steps, ball should slide on both surface, track and gutters.

But we see one more problem, when we start game, ball is on the track, and moves forward, by itself, let’s then move our ball to other place, let’s name it BallHolder. To build it it’s very trivial, using few cubes. Additionally I have added, a small barrier for pins, to make better simulation of real bowling, and discovered that pins should be more closer to them self, because it was too unreachable to fall down all the pins. You can see the changes on following picture



One clever thing, we always will have the same pin objects on the map and during game they will always exist, so we can hold them in private variable in GameController script.

```
    private PinController[] pinsTransform;
```

And at Start of the game find them and assign to variable.

```
    void Start()
    {
        pinsTransform = GameObject.FindWithTag("Pin
Group").GetComponentsInChildren<PinController>();
        roundM.HideRound();
       ...
    }
```

In every round in the game, we have to calculate how many pins was down, we can do it simply by count every occurance of being pinDown. We can add following method to the GameController.

```
    public int getAmountOfPinsDown()
    {
        int counter = 0;
        if (pinsTransform != null)
        {
            foreach (PinController pinTransform in pinsTransform)
            {
                if (pinTransform.isPinDown())
                {
                    counter++;
                }
            }
        }
        return counter;
    }
```

We can test our method by showing current amount of pin down in console, we can do that for example using “J” key, add following code in Update method

```
    void Update () {
         if (Input.GetKeyDown(KeyCode.J))
         {
             Debug.Log(getAmountOfPinsDown().ToString());

         }
         ...
    }
```

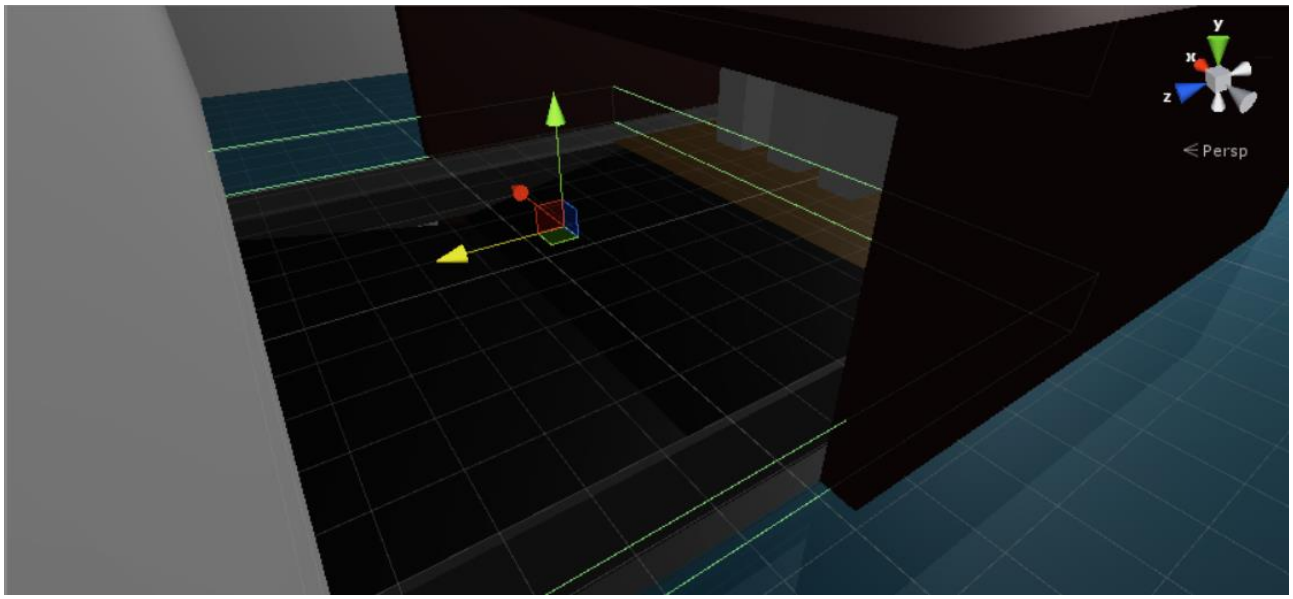
Game have to be well managed, so firsly we will add 2 configuration variables to our GameController, numberOfPlayers and numberOfRounds with presented values.

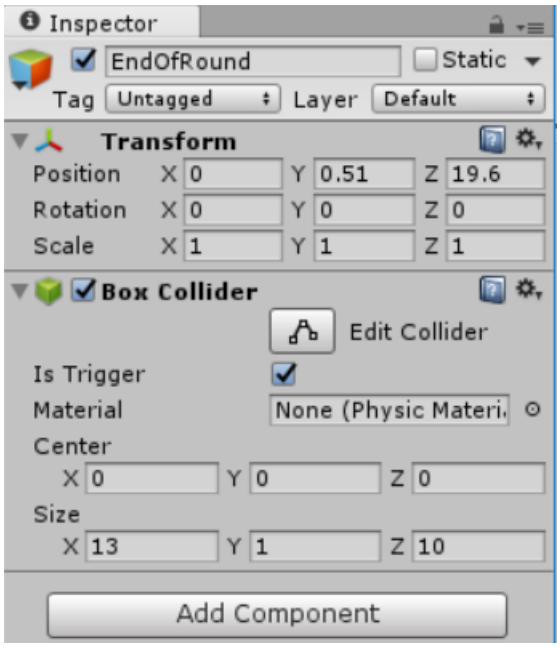
```
    public int numberOfPlayers = 2;
    public int numberOfRounds = 3;
```

Another matter thing is, the end of each round, in our currently game state we didn’t do anything with it, let’s assume that the round end in following cases :

1. The ball moved though the pins and stopped.
2. The ball came to gutter, and rolled to the end of it.
3. The ball is out of playing area.

We can manage point 1 and 2 by creating empty object with collider that leaps end of the track with gutters. Denote it as EndOfRound. We add to the object, BoxCollider and mark triggering.





Now we will create a script that checks if ball collides with this object. We have already created similar script for slide areas, but now instead of adding force we store only info of event. Denote script as CollisionDetector.

```
public class CollisionDetector : MonoBehaviour {

    public bool ballCollided = false;

    public bool isBallCollided()
    {
        return ballCollided;
    }

    void OnTriggerEnter(Collider other)
    {
```

```
        if (other.tag == "Ball")
        {
            ballCollided = true;
        }
    }
}
```

Then we attach the script to objects, RestArea's and EndOfRound. We add also tags to RestArea's and EndOfRound. We have 2 of Rest Areas, so let change their name to make them unique objects. Rest Area on Left side denote as RestAreaLeft, and analogous Rest Area on Right, RestAreaRight. Tags would be the same as name object but with spaces.

RestAreaLeft – Rest Area Left

RestAreaRight – Rest Area Right

EndOfRound – End of Round

For every of object above apply the tags.

In GameController, find the object by tags and assign to variables with the same name as object.

```
    private CollisionDetector restAreaLeft;
    private CollisionDetector restAreaRight;
    private CollisionDetector endOfRound;

    void Start()
    {
        restAreaLeft =
GameObject.FindWithTag("Rest Area Left").GetComponent<CollisionDetector>();
        restAreaRight =
GameObject.FindWithTag("Rest Area Right").GetComponent<CollisionDetector>();
        endOfRound =
GameObject.FindWithTag("End Of Round").GetComponent<CollisionDetector>();

       ...

    }
```

To cover our conditions we can create 2 methods inside GameController.

```
    public bool isBallOutOfMap()
    {
        if (restAreaLeft.isBallCollided() || restAreaRight.isBallCollided())
        {
            return true;
        }
        return false;
    }

    public bool isEndOfRoundReached()
    {
        if (endOfRound.isBallCollided())
        {
            return true;
        }
        return false;
    }
```

We have to also add variable isBusy for control our update function, there we recognise the collision do few things and then reset the variable. That's required, because of update function is call in every game frame tick.

We will also use IEnumarator function to call some function with late.

```
    void Update () {

        if (!isBusy && (isBallOutOfMap() || isEndOfRoundReached()))
        {
            isBusy = true;
            StartCoroutine(WaitAndShowScoreAndStartNewRoundWithLate());
        }

        ...

    }
```

```
    IEnumerator WaitAndShowScoreAndStartNewRoundWithLate()
    {
        yield return new WaitForSeconds(3);
        score = GetAmountOfPinsDown();
        pinsDown.text = __SCORE + score.ToString();
        StartCoroutine(StartNewRoundWithLate());
    }
```

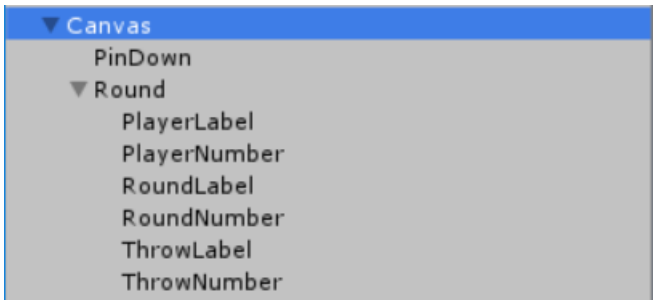
Every IEnumarator function used to be called Routines, to start them we have to use StartCoroutine function.

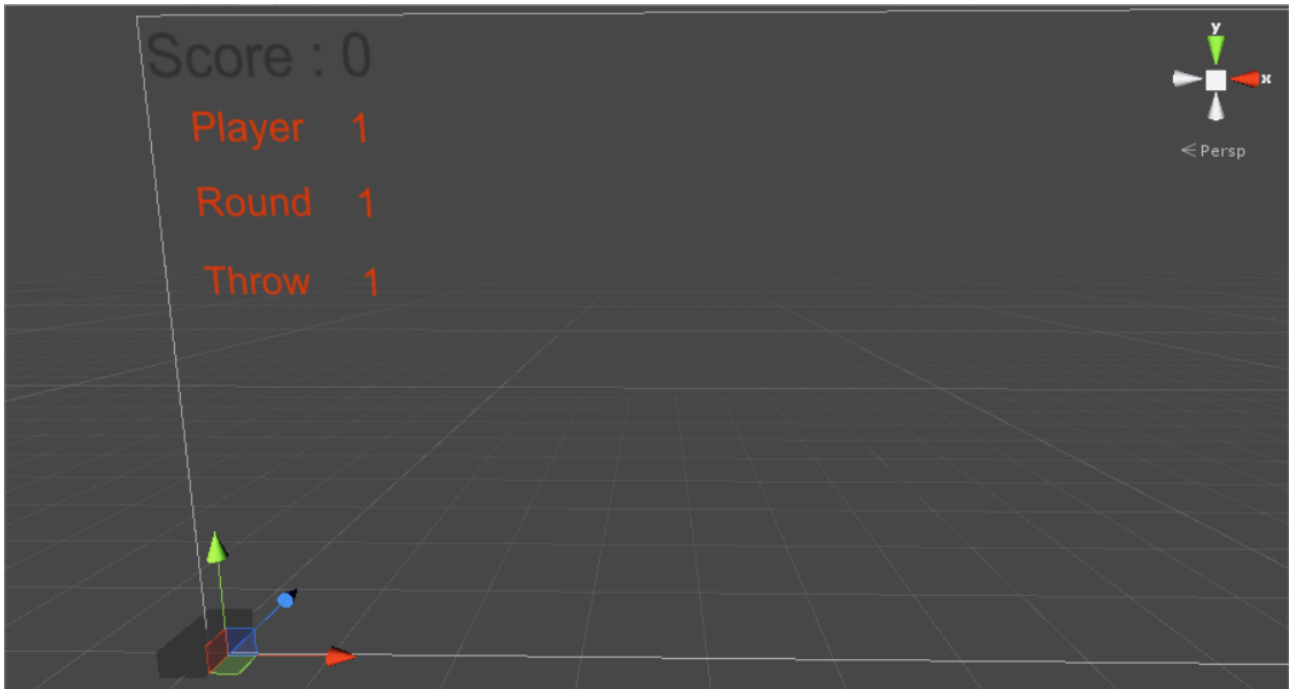
```
    IEnumerator StartNewRoundWithLate()
    {
        yield return new WaitForSeconds(3);
        StartNewRound();
    }

    public void StartNewRound()
    {
        RestartBallState();
        ResetBallColliders();
        if (score == 10 || numberOfThrow == 2)
        {
            RestartPinsState();
            score = 0;
            pinsDown.text = __SCORE + score;
            currentPlayer = (currentPlayer + 1) % players.Length;
            if (currentPlayer == 0)
            {
                currentRound++;
            }
            roundM.StartNewRound(players[currentPlayer], currentRound);
            numberOfThrow = 1;
        }
        else
        {
            numberOfThrow = 2;
            DisactivePinsDown();
        }
        roundM.StartNewThrow(numberOfThrow);
        isBusy = false;
    }
```

A lot of new things are there. I will explain that shortly. pinsDown variable represents gui object, it will be shown on our screen during game. This element will store our score, means the pins down during round. In

every round we have to chances of throwing, we called it numberOfThrow. To add Gui Text Element right click on Hierarchy -> Gui -> Text. I also have created roundM, which represents round manager. It has 2 functions StartNewRound and StartNewThrow. The only thing that the functions do is changing of Gui text.

The Gui Text elements I have added.

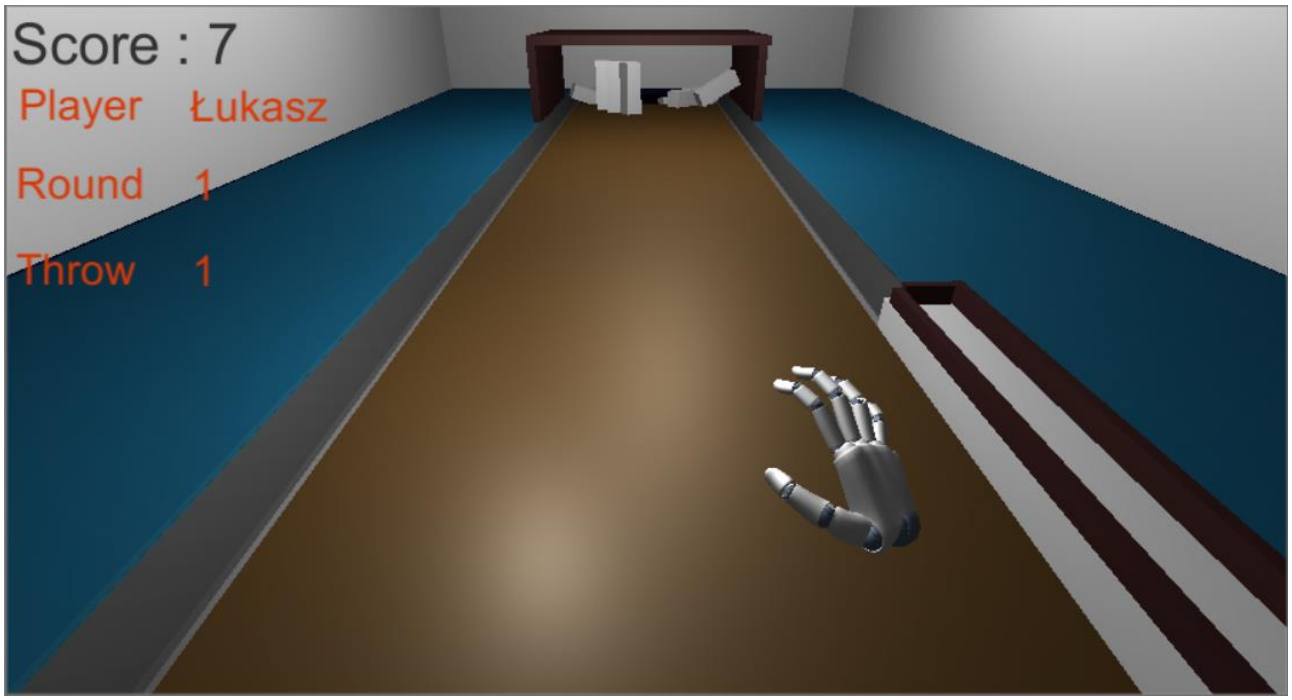




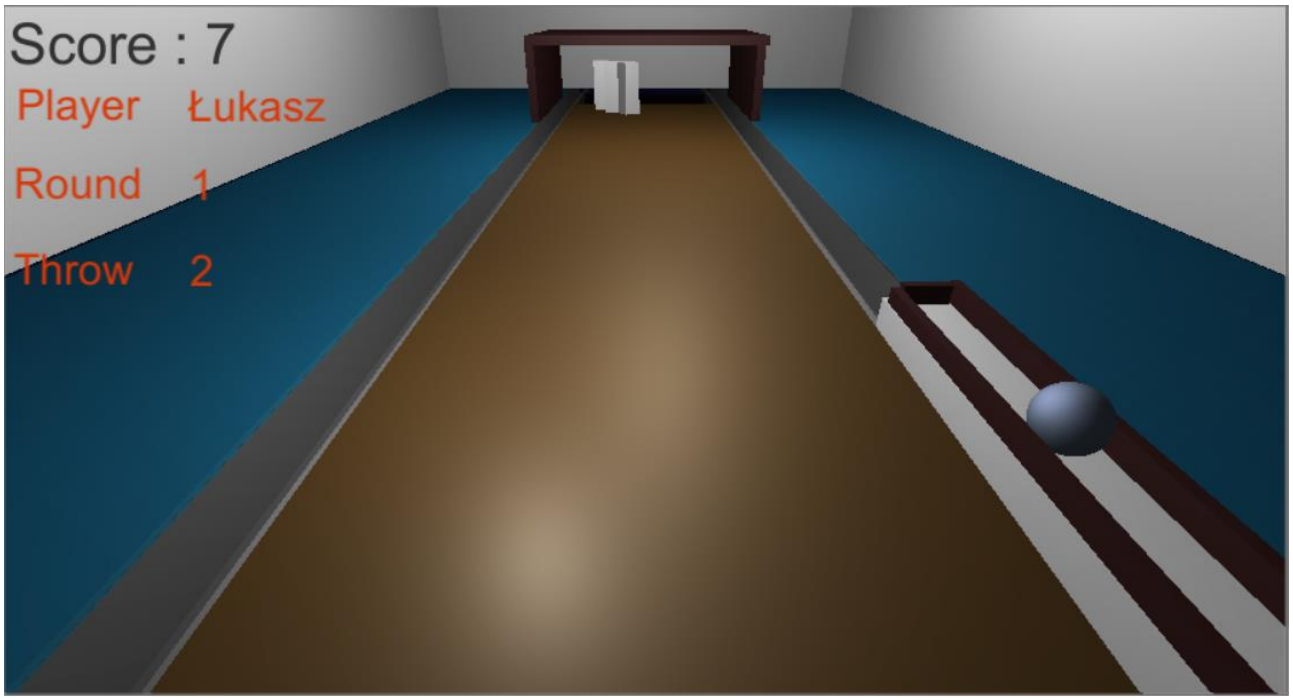
After starting new round we also reset state of Ball, CollidersDetectors and Pins.

In Game Controller we also added few parameters used also the previous presented part of code.

In current state of game if we press play and fall down some pins, It would be as below.



And after 6 seconds



One of most matter things are scores and statistics of them. For this purpose we create one class Stats, and class Score.

```
public class Score {

    public int score1 = -1;
    public int score2 = -1;

    public Score(int score1, int score2)
    {
        this.score1 = score1;
        this.score2 = score2;
```

```
    }
    public Score(int score1)
    {
        this.score1 = score1;
        score2 = -1;
    }
}
```

```
public class Stats : MonoBehaviour {

    private string __Spaces = "    ";

    private string __players = "Player";
    private string __round = "Round";
    private string __total = "Total";

    public Text header;
    public Text[] players;

    private List<string> playersNames = new List<string>();

    List<List<Score>> playersScore = new List<List<Score>>();


    public void AddFinalStats()
    {
        int sum;
        for (int i = 0; i < playersScore.Count; i++)
        {
            sum = 0;
            for (int j = 0; j < playersScore[i].Count; j++)
            {
                sum += playersScore[i][j].score2;
            }
            players[i].text += __Spaces + __Spaces + sum;
        }
    }

    public void AddScoreThrowOne(int playerIndex, int scoreThrowOne, bool was10PointScored)
    {
        if (was10PointScored)
        {
            playersScore[playerIndex].Add(new Score(scoreThrowOne, scoreThrowOne));
        }
        else
        {
            playersScore[playerIndex].Add(new Score(scoreThrowOne));
        }
    }

    public void AddScoreThrowTwo(int playerIndex, int scoreThrowTwo)
    {
        playersScore[playerIndex][playersScore[playerIndex].Count - 1].score2 =
scoreThrowTwo;
    }

    public void SetupPlayers(string[] playersNames)
    {
        for (int i = 0; i < playersNames.Length; i++)
        {
            this.playersNames.Add(playersNames[i]);
            playersScore.Add(new List<Score>());
        }
    }
```

```
    public void UpdateStats()
    {
        for (int i = 0; i < playersNames.Count; i++)
        {
            players[i].text = playersNames[i];
        }
        for (int i = 0; i < playersNames.Count; i++)
        {
            for (int j = 0; j < playersScore[i].Count; j++) {
                players[i].text += __Spaces + __Spaces + playersScore[i][j].score1;
                if (playersScore[i][j].score2 != -1) {
                    players[i].text += "/" + playersScore[i][j].score2;
                }
            }
        }
    }


    public void GenerateStatsAtStart(int rounds)
    {
        AdjustHeader(rounds);
    }

    private void AdjustHeader(int rounds)
    {
        header.text = __players + __Spaces;
        for (int i = 1; i <= rounds; i++)
        {
            header.text += __round + i + __Spaces;
        }
        header.text += __total;
    }


    public void ShowStats()
    {
        gameObject.SetActive(true);
    }

    public void HideStats()
    {
        gameObject.SetActive(false);
    }

}
```

In variable Text[] Players, we store references to Gui Objects, that represents every stats for each player. The script is adjust to the player amount configured in GameController.

Score are stored in complexed variable

```
List<List<Score>> playersScore = new List<List<Score>>();
```

That complicated list represents, List of scores, for every player on List.

```
playersScore[PlayerIndex][ScoreIndex].(score1 or score2)
```

We call all function from the script from the GameController.

```
    void Start()
    {
        stats = GameObject.FindWithTag("Stats").GetComponent<Stats>();
        pinsDown = GameObject.FindWithTag("Pin Down").GetComponent<Text>();
        stats.GenerateStatsAtStart(numberOfRounds);
```

```
        stats.SetupPlayers(players);
        stats.UpdateStats();
        stats.HideStats();
    }
```

And also we have to adapt Start New Round function

```
    public void StartNewRound()
    {
        RestartBallState();
        ResetBallColliders();
        if (score == 10 || numberOfThrow == 2)
        {
            if (score == 10 && numberOfThrow == 1)
            {
                stats.AddScoreThrowOne(currentPlayer, score, true);
            }
            else
            {
                stats.AddScoreThrowTwo(currentPlayer, score);
            }
            RestartPinsState();
            score = 0;
            pinsDown.text = __SCORE + score;
            currentPlayer = (currentPlayer + 1) % players.Length;
            if (currentPlayer == 0)
            {
                currentRound++;
            }
            roundM.StartNewRound(players[currentPlayer], currentRound);
            numberOfThrow = 1;
        }
        else
        {
            stats.AddScoreThrowOne(currentPlayer, score, false);
            numberOfThrow = 2;
            DisactivePinsDown();
        }
        stats.UpdateStats();
        roundM.StartNewThrow(numberOfThrow);
        isBusy = false;
    }
```

And the last condition to over the game, we create bool gameOver = false and inside Update function we write

```
        if (!gameOver)
        {
            if (Input.GetKeyDown(KeyCode.Tab))
            {
                stats.ShowStats();
            }
            if (Input.GetKeyUp(KeyCode.Tab))
            {
                stats.HideStats();
            }
            if (lastRoundPassed)
            {
                gameOver = true;
                stats.AddFinalStats();
                stats.ShowStats();
            }
```

```
            if (!isBusy && (isBallOutOfMap() || isEndOfRoundReached()))
            {
                isBusy = true;
                StartCoroutine(WaitAndShowScoreAndStartNewRoundWithLate());
            }
        }
```

And if we pass the previous selected number of rounds, game will end with score shown. We can also check Stats during game by pressing TAB button.