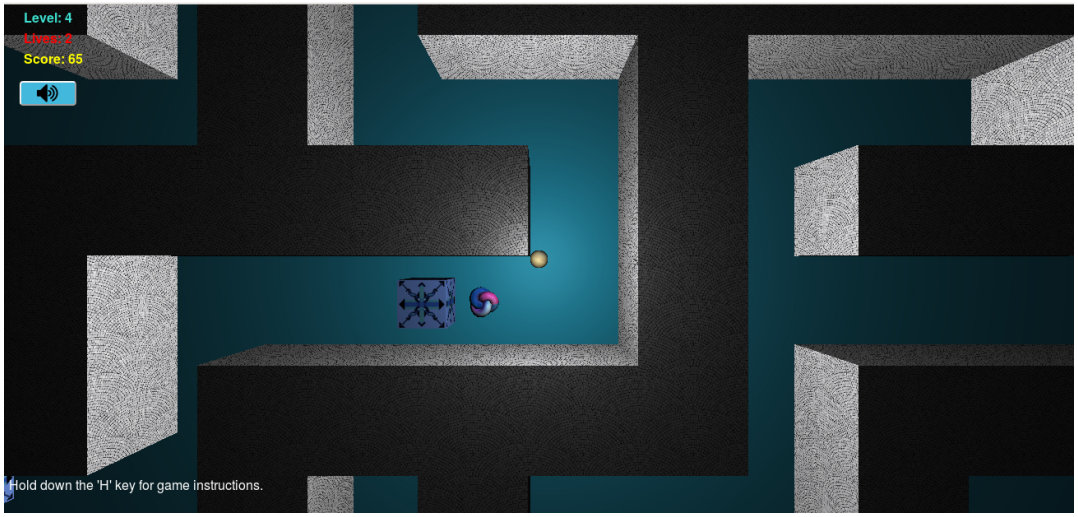
SAPIENZA
UNIVERSITÀ DI ROMA

PROJECT REPORT FOR INTERACTIVE GRAPHICS

Project Implemented - Bounce game

Instructor: Prof. Shaerf Marco



Group Members:

Mohammed Sukhsarwala - 1848668
Maryam Bandali - 1861486
Oluwatoyin Sanni - 1871835
Mariam Garba - 1871871

June 23, 2019.

Table of Contents

Section A: Technical Presentation

The game developed in this project was inspired by a famous game called BOUNCE which was published and launched by Nokia. In the original Nokia BOUNCE game, the player controls a red ball through many levels in a 2D side-scrolling world game. The player dodges the obstacles in the environment, passing them and jumping through various hoops avoiding the obstacles.

The BOUNCE game developed in this project is similar to the original BOUNCE game and also simulates a lot of objects for gravity, friction, elasticity, joined bodies, collision, etc using some javascript libraries.

We also added some additional game functionalities which can make the game more interesting and challenging. You can find the details of these functionalities later in this report.

1.0 Description of the environment and libraries used in the project

The following libraries were used in the project:

- **ThreeJs**: This was used to create and display animated 3D computer graphics in the web browser.
- **Box2DWeb**: This is a 2D Physics Rendering Engine for creating a simulation of gravity, friction, collision, force etc.
- **JQuery**: This was used to perform HTML DOM Tree Traversal and Manipulation.
- **KeyboardJs**: This was used to bind keys to perform actions on the scene.

2.0 Description of all the technical aspects of the project

In order to setup the dynamic world that would represent the ball, maze, obstacles and pillet components in the game scene, we used ThreeJs (for rendering the scene) and Box2DWeb (to create a simulation of friction, density, force, restitution and collision between the ball and the maze). The step by step implementation from creating the dynamic world to starting the game is described as follows:

- ### Creating the dynamic world

  In order to create a dynamic world representation of the components used in the game scene (Ball and Maze), we implemented a *createDynamicWorld()* function. This function is responsible for creating the ball as a dynamic body and placing it in a particular position. It also assigns a density, friction, restitution, gives the ball a circular shape before placing it in the game scene.

  The function is also responsible for creating the maze as a static body, assigns it a polygon shape and places the maze at a predefined position based on the maze dimension. A snippet of this function is shown below:

```
function createDynamicWorld() {

        // Create the world object.

        newWorld = new box2DWorld(new box2DVec2(0, 0), true);

        // Create the ball.

        newBall = newWorld.CreateBody(newBodyDefinition);

        newBall.CreateFixture(newFixtureDefinition);

            // Create the maze.

        for (var i = 0; i < maze.dimension; i++) {

        for (var j = 0; j < maze.dimension; j++) {

        if (maze[i][j]) {

        newBodyDefinition.position.x = i;
```

```
            newBodyDefinition.position.y = j;

        newWorld.CreateBody(newBodyDefinition).CreateFixture(newFixtureDefinition);

            } } } }
```

- **Creating the scene render world**

  In order to render the game scene with all its components (light, camera, obstacle, pillet, ball, exit, maze and plane), we implemented a *createSceneRenderWorld()* function. This function is responsible for creating a ThreeJs Scene, creating a geometry of all the components (except light), assigns a texture to these components (except light), creates a ThreeJs Mesh of the components (except light) and places them at a predefined location in the rendered game scene. In case of the light, a white point light (using *THREE.PointLight()* function) is placed on the rendered game scene. A snippet of this function is shown below:

```
function createSceneRenderWorld() {

            // Create the scene object.

            scene = new THREE.Scene();

            // Add the light.

            scene.add(light);

            // Add the Ball Mesh.

            scene.add(ballMesh);

            // Add the Obstacle Mesh

            addObstacle();

            //Addition of Pillet Mesh

            addPillets(level/2);

            //Adding the Exit Mesh

            scene.add(exitMesh)

            // Adding the Camera.

            scene.add(camera);

            // Adding the Maze Mesh.

            scene.add(mazeMesh);

            // Adding the Plane Mesh.

            scene.add(planeMesh); }
```

- **Generating the maze mesh**

  In order to generate a maze, we implemented a *generate_ maze_ mesh()* function. This function is responsible for creating a cube geometry using ThreeJs which accepts the field parameter (this contains a maze coordinate) to generate different cube geometries and merges these geometries to form a single maze. A snippet of this function is shown below:

```
function generate_maze_mesh(field) {

var emptyGeometry = new THREE.Geometry();

for (var i = 0; i < field.dimension; i++) {

for (var j = 0; j < field.dimension; j++) {

if (field[i][j]) {

. . . . . ...

}

}

                }

var material = new THREE.MeshPhongMaterial({ map: meshTexture} );

var mesh = new THREE.Mesh(emptyGeometry, material)

return mesh;

}
```

- **Updating the dynamic world**

When the ball position changes, there is a need to update the ball's dynamic world properties such as friction and force. The *updateDynamicWorld()* function was implemented to perform these updates after taking a time step. A snippet of this function is shown below:

```
function updateDynamicWorld() {

// Applying Friction to the ball

newBall.SetLinearVelocity(velocity);

// Applying Force to the ball controlled by player

var force = new box2DVec2(animationAxis[0]* newBall.GetMass()* 0.25, animationAxis[1]* newBall.GetMass()*
0.25);

newBall.ApplyImpulse(force, newBall.GetPosition());

// Taking a time step.

newWorld.Step(1/60, 8, 3);

}
```

- **Updating the scene render world**

When the ball is moving in the game scene, it is necessary to update the ball's position and orientation. We also need to check if the ball has collided with an object (obstacles or picked a pillet) in the scene. The collision of the ball with an object determines the lives and reward that would be assigned to the player playing the game. Finally, we then update the camera and light's position (with the ball being their center of attention) as the ball rotates in the scene. The *updateSceneRenderWorld()* function was implemented for this purpose. A snippet of this function can be seen below:

```
function updateSceneRenderWorld() {

        // Updating Ball position.

        ballMesh.position.x += xIncrement;

        ballMesh.position.y += yIncrement;


        // Updating Lives and Score

        checkBallCollisionWithObjects(actualBallPosX,actualBallPosY);


        // Updating Ball rotation.

        ballMesh.matrix = tempMat;

        ballMesh.rotation.getRotationFromMatrix(ballMesh.matrix);


        // Updating Camera Position and Light Position.

        camera.position.x += (ballMesh.position.x - camera.position.x) * 0.1;

        light.position.x = camera.position.x;

        }
```

- **Game State Animation**

In order to start the game, we first initialize the game scene (using the *initializeGame()* function), setup the game scene (using the *setupGameScene()* function) before actually playing the game (using the *playGame()* function). There are different levels in the game and transition between levels is done by the *requestAnimationFrame()* function. All the functions described in the game animation section was implemented except the *requestAnimationFrame()* function that is inbuilt in ThreeJs. A snippet of the *gameAnimation()* function is shown below:

```
    function gameAnimation() {
            if (gameState == "initialize"){
               initializeGame();
            }
            else if (gameState == "setup"){
               setupGameScene();
            }
            else if (gameState == "play"){
               playGame();
            }
            else if (gameState == "next"){
               nextGameLevel()
            }
```

```
        requestAnimationFrame(gameAnimation);
    }
```

3.0 Description of the implemented interactions

This section explains the interactions starting from the modelling structure used to the animations that was implemented in this project.

- **Hierarchical Models**

  In order to create a hierarchical model of the bounce game which is composed of the following parts:

  - Light
  - Camera
  - Ball Mesh
  - Pillet Mesh
  - Obstacle Mesh
  - Plane Mesh
  - Maze Mesh
  - Exit Mesh

  We modeled the game using hierarchical modeling according to the structure in Figure 1. We used *Scene.add()* function of ThreeJs to add all these components to the *Scene* in a hierarchical manner. ThreeJs geometry function was also used to create different parts of these components.

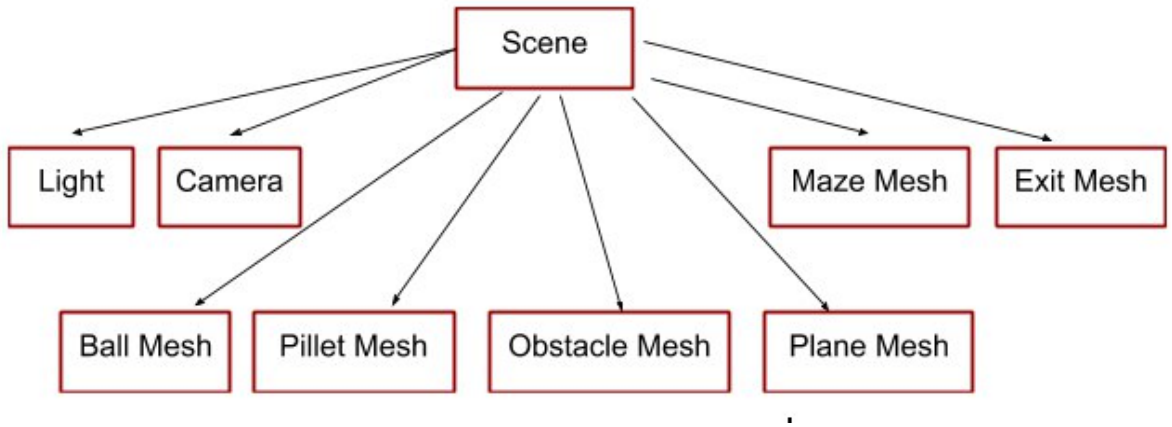


Figure 1: Hierarchical Modelling

- **Lights and Textures**

  To initialize the game, we added a white point light close to the camera's location and increased the light's intensity when the game scene is fully setup. During the playing state of the game, the light's location is updated following the camera's location as the ball moves through the maze. Finally, the light's intensity is decreased at the end of the current game level i.e. when the ball exits the maze before moving to the next game level. The light component was implemented using *THREE.PointLight()* function.

For the textures, 6 components were implemented:

- ballTexture : This is the game's ball texture which was loaded using *THREE.ImageUtils.loadTexture()* function with the image path (that contains the image to be used as the ball's texture) as its argument.
- obstacleTexture : This is the game's obstacle texture which was loaded using *THREE.ImageUtils.loadTexture()* function with the image path (that contains the image to be used as the obstacle's texture) as its argument.
- planeTexture : This is the game's plane texture which was loaded using *THREE.ImageUtils.loadTexture()* function with the image path (that contains the image to be used as the plane's texture) as its argument.
- meshTexture : This is the game's maze texture which was loaded using *THREE.ImageUtils.loadTexture()* function with the image path (that contains the image to be used as the maze texture) as its argument.
- pilletTexture : This is the game's pillet texture which was loaded using *THREE.ImageUtils.loadTexture()* function with the image path (that contains the image to be used as the pillet texture) as its argument.
- exitTexture : This is the game's exit texture which was loaded using *THREE.ImageUtils.loadTexture()* function with the image path (that contains the image to be used as the exit texture) as its argument.

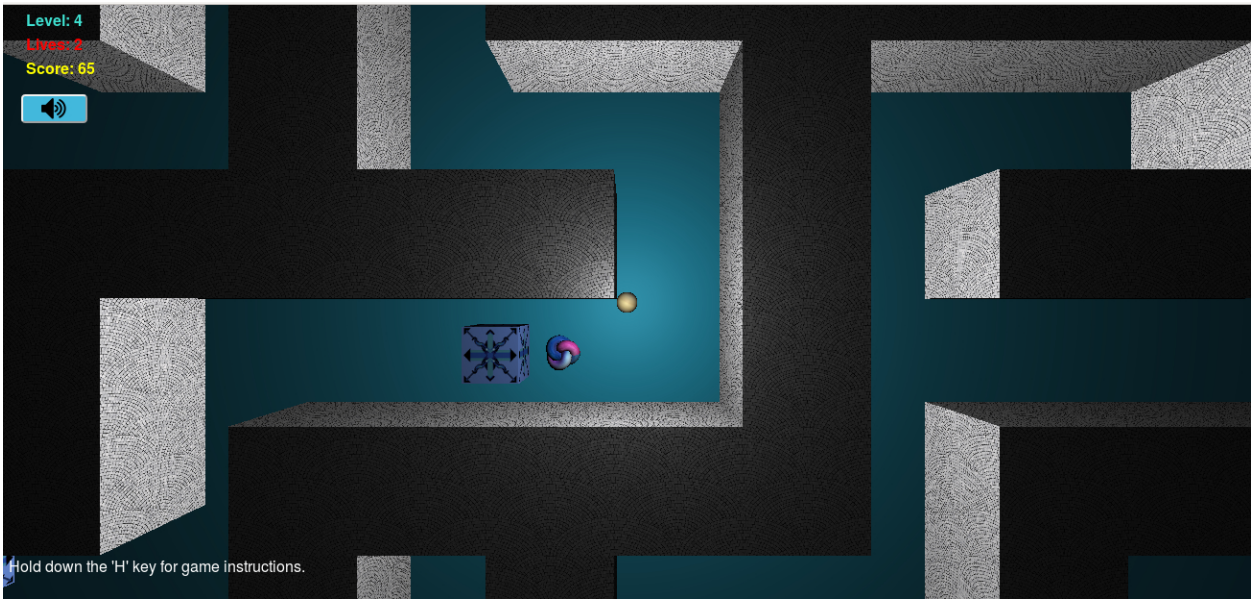


Figure 2: Light and Textures

- ## Player Interaction

To navigate the ball around the maze, we implemented the *onMoveKey()* function which detects what key direction that was pressed by the player. Also, as the ball navigates through the maze, an audio is played to indicate its movement.

```
function onMoveKey(axis) {
        game_audio.play();
        game_audio.volume = 0.3;
        animationAxis = axis.slice(0);
}
```

This key direction can be either '*left*', '*right*', '*down*', '*up*' **or** '*a*', '*d*', '*s*', '*w*' **or** '*1*', '*3*', '*2*', '*5*' because these values have already been bound to the keyboard.

```
KeyboardJS.bind.axis('left', 'right', 'down', 'up', onMoveKey);

KeyboardJS.bind.axis('a', 'd', 's', 'w', onMoveKey);

KeyboardJS.bind.axis('1', '3', '2', '5', onMoveKey);
```

Depending on the key that was pressed, a force is applied to the ball to move it in that particular direction.

```
var force = new box2DVec2(animationAxis[0] * newBall.GetMass() * 0.25, animationAxis[1] * newBall.GetMass() * 0.25);

newBall.ApplyImpulse(force, newBall.GetPosition());
```

A player has **3 lives** in the game and is awarded some rewards (points) in different stages:

- 20 points for completing a level
- 10 points for picking a pillet
- - 5 points for colliding with an obstacle

To update the **player's points and lives**, a collision detection with obstacle is performed.

```
function checkCollisionWithObstacle(ball_Pos_X,ball_Pos_Y){
        if (obstacleLocation.length<=0)
        return false;
            for( i = 0;i<obstacleLocation.length;i+=2){
              if((ball_Pos_X<=(obstacleLocation[i]+0.3)& &  ball_Pos_X>=(obstacleLocation[i]-0.3) )
            & &
            (ball_Pos_Y<=(obstacleLocation[i+1]+0.3)& & ball_Pos_Y>=(obstacleLocation[i+1]-0.3)))
            return true
            }
            return false;
}
```

If collision occurs, the player's points is reduced by 5 (provided that he already has at least 10 points) and lives is reduced by 1. An audio effect is played to indicate the collision. Afterwards, his score and lives would be updated. The player would also have to restart the current level because of the collision. If the lives has been reduced to **zero**, then the game is over and the total score is shown. The player can try again or quit.

We also checked if the player picked a **pillet** with the ball.

```
function pickingPilletByBall(ball_Pos_X,ball_Pos_Y){
        if(pilletLocation.length<=0)
        return false;
            for(i=0;i<pilletLocation.length;i+=2)
            {
            if((ball_Pos_X<=(pilletLocation[i]+0.3)& &  ball_Pos_X>=(pilletLocation[i]-0.3) )
            & &
            (ball_Pos_Y<=(pilletLocation[i+1]+0.3)& & ball_Pos_Y>=(pilletLocation[i+1]-0.3)))
            return i;
            }
            return -1;
}
```

If he does, an audio is played to indicate the picking of pillet. He is rewarded with 10 points and his overall score is updated.

Finally, if the ball gets to the exit of the maze, the scene fades out and moves to the next level.

```
function nextGameLevel(){
        scene.remove(exitMesh)
        updateDynamicWorld();
        updateSceneRenderWorld();
        light.intensity += 0.07 * (0.0 - light.intensity);
        renderer.render(scene, camera);
        if (Math.abs(light.intensity - 0.0) < 0.1) {
            light.intensity = 0.0;
            renderer.render(scene, camera);
            gameState = 'initialize'
        }
}
```

- **Animations**

The game has many animations which makes it an interesting game to play. We created many animations for the game. Some of the animations include obstacle translation, ball movement and rotation, the ExitMesh rotation, removal of pillet on ball collision. We will be describing the animations which we have created in the game below:

- **Obstacle Translation**

  To increase the complexity of the game, we included a feature of obstacle movement. This animation uses functions like the *obstacleTranslation()* and *animatingObstacle()*. After finding the location for the obstacle to be placed, the next task was to find the position of wall nearby the obstacle so that it can do movement. On deciding that the obstacle moves only in the X direction, being positive and negative,hence the task of obstacleTranslation function was to find the wall position for each obstacle and store them in an array. We store the position of wall on right of obstacle first, then the left position, for each obstacle. After we complete finding the locations of wall and adding the obstacles, the next task is to make the obstacle move, which happens in animatingObstacle function. In this function, the logic we applied is updating the array of obstacle location array continuous and also keep a flag check. The flag is used to indicate that a particular obstacle has reached its deadend and will traverse back until it reaches the other deadend. The obstacle doesn't move on the last block , being the deadend block, just before the exit mesh, leaving a place safe for player once he reaches that block. The animation of obstacle happens by using the TRANSLATION function with a particular value. This value is calculated every time player clears a level, this helps in increasing the speed of the obstacle movement as well increase the complexity of the game.

- **Ball Movement and Rotation**

  Making the ball rotate was one of the important animations as it was vital to the fun of the game. We have used the Box2DWeb library for making the ball rotate and move ahead. The game starts with ball at location <1,1> which is our initial position for all levels, and when the player loses a life. We have created ball mesh with the initial position. After the initial position, when the player presses the required keys set up for the game, we get the velocity using the *GetLinearVelocity()* function and get it multiplied with a constant of **0.95** to keep the movement steady. This velocity is the new position of our ball and when subtracted with old position gives us the original position of the ball. We are also taking into consideration the force, which is applied when the player presses the key to keep the ball moving, this is also calculated by using the *Box2DWeb* library. Once we find the updated dynamics for movement, we then make the ball rotate using the following code:

```
var tempMat = new THREE.Matrix4();

tempMat.makeRotationAxis(newTHREE.Vector3(0,1,0), xIncrement/ballRadius);

tempMat.multiplySelf(ballMesh.matrix);

ballMesh.matrix = tempMat;

tempMat = new THREE.Matrix4();

tempMat.makeRotationAxis(new THREE.Vector3(1,0,0), -yIncrement/ballRadius);

tempMat.multiplySelf(ballMesh.matrix);

ballMesh.matrix = tempMat;

ballMesh.rotation.getRotationFromMatrix(ballMesh.matrix)
```

  We make the ball rotate in Y-Axis by step movement, basically by dividing the increment of X by radius of ball, which makes the rotation of ball in X direction with rotation in Y, but to make the ball move in the X direction we make the increment of ball movement by dividing the Y increment by radius of ball.

- **Removal of Pillet on Ball Collision**

  This animation is to help the player achieve a higher score. The pillet adds 10 points more to a score and they exists only on even number of levels. Each even level of the game consists of half the amount of pillets. When an even number of level is loaded, *addPillet()* function is called which calculates and stores the locations for adding the pillet in an array called *pilletLocation*. When the ball reaches a pillet, the location of ball is matched with the pillet location to check whether the ball has touched the pillet, this task is done by *pickingPilletByBall()* function. If the ball location is found to be at pillet location, then this function returns the array index of the pillet, which helps the game in choosing which pillet to be removed. By using the *getChildByName()* function, we achieve that particular pillet mesh, by providing the pilletMesh + its index location as its name. This helps in finding the pillet to be removed. The location of the pillet in array *pilletLocation* is made to -1 for that level and the pillet is removed from scene, simultaneously the score is updated with 10 more points.

- **ExitMesh Animation**

  The ExitMesh Animation consists of decreasing opacity of exit mesh as the ball reaches the end of the maze. The mesh opacity continuously decreases as the ball reaches the end of maze upto value of 0.2 and it becomes 1 the moment ball leaves and enters the maze again. The exit mesh is removed from scene once the ball exits the maze.

Section B: User Manual

This section describes the requirements before a player can play the game and also the instructions on how to play the game.

1.0 Requirements

The player needs a web browser to run the game. The game has been tested on Google Chrome, Microsoft Edge, and Mozilla Firefox, and have been found to be successfully running without any glitches.

The player can run the game in 2 ways. The first step is a developer method, where player runs the game following the steps below:

1) Open cmd and traverse till the path of the game folder.
2) Start "python -m SimpleHTTPServer" in your shell (for Python 3.0 and above type "python -m http.server" in your shell)
3) Open browser and type : localhost:8080/

And these steps will help in loading the game locally. But if u intend to run the game online then you can use the following link for playing the game: https://sapienzainteractivegraphicscourse.github.io/finalproject-ymcube-team/

2.0 How to play

- **Keyboard Controls**

The game is intended to attract young kids as well as current generation students. The game being easy to play doesn't have much trouble in learning it. The controls are very basic and has multiple controllers too.
The basic controllers being the arrow keys, i.e for moving left, right, up and down. The table below shows the respected keys for playing the game depending on the player's preference:

UP	***DOWN***	***RIGHT***	***LEFT***
Arrow key Up	Arrow Key Down	Arrow Key Right	Arrow key Left
w	s	d	a
5	2	3	1

- **Lives and Scorecard**

The simplicity of the game makes the scoring simple as well. The score is increased when the ball hits the pillets, and total score is updated on every level successfully completed. The player has to exit the maze, depicted by an exit rotating image, which leads him/her to the next level. On every even level number, there are half the number of pillets added to increase the chances of higher scores of the player. Each pillet increases the player score by 10 when the ball collides with it. Player has to avoid obstacles but if he/she is unable to avoid, and touches the obstacle, the player loses one life and score is reduced by 5 points. Every initial game has 3 lives and on losing all 3 lives, the game is over and the score is published with a choice to start all over again.

3.0 Troubleshooting

The troubleshooting of the game can be done by:

1) Closing the terminal, and restarting the game.

2) Clearing your cookies and browser history, and then performing the steps for running the game.

Even if following these steps, you are facing an errors, glitches or issues, We would request you to please contact us using the details provided in Technical Support.

4.0 Technical Support

The player can contact us on any of the following email addresses:

1) sukhsarwala.1848668@studenti.uniroma1.it

2) garba.1871871@studenti.uniroma1.it

3) sanni.1871835@studenti.uniroma1.it

4) bandali.1861486@studenti.uniroma1.it

Fill the subject field with "BOUNCE Game". Letter body should contain text information about your: -
Detailed problem description-Screenshot/Video Attached. **We also welcome FEEDBACK on these email address.**

References

https://github.com/wwwtyro/Astray-2
http://threejs.org/
https://api.jquery.com/
https://github.com/hecht-software/box2dweb