Introduction to Game Development in Unity

Radosław Mantiuk

March, 2024

Contents

Chapter 1

Unity3D

Unity (link to Unity website) is a cross-platform **game engine** developed by Unity Technologies that was first unveiled and launched in June 2005 during the Apple Worldwide Developers Conference as a game engine for Mac OS X. Over time, it has expanded its support to encompass various platforms, including desktop, mobile, console, and virtual reality. Unity has garnered particular popularity in mobile game development for iOS and Android, being favored for its accessibility to novice developers and its prevalence in indie game development. Notably, Unity facilitates the creation of both three-dimensional (3D) and two-dimensional (2D) games, as well as interactive simulations.

"A game engine is the point of convergence for all aspects of creating a game. Games, like all applications, are made of smaller pieces like 3D models, scripts, and audio files. When put together, they create the full user experience. If 3D models, scripts, and audio files were ingredients, Unity (and other game engines) would be the stockpot you dropped them into!" ([citation])

Figure 1.1: Game engine ([citation]).

```
if (Input.GetKey(KeyCode.A)) {
    rb.AddTorque(-transform.up * forceTurn);
}
```

1.1 Introduction to Unity Editor

The **Unity Editor** is a **software tool** developed by Unity Technologies for creating games and interactive experiences. It provides a user-friendly interface for designing environments, importing assets, writing scripts, and testing projects. With real-time preview capabilities, it enables quick iteration and collaboration among team members.

Objectives: Become familiar with the Unity Editor user interface.

1. Exercise - Tutorial "Explore the Unity Editor"

1. Register on the Unity website to create your Unity ID (link). You will need an account to log in to Unity Hub and the Unity portal. A personal license will be assigned to the account, which allows you to use Unity also

on your private computer.

2. Launch **Unity Hub** and log in using your Unity account authentication.
3. Explore available Unity Editor installations (see Fig.1.2 (left)).
4. Create new project using **3D Core** template (see Fig.1.2 (right)). Set *Project Name* to **Game3D**. Select project *Location* (e.g., in `Unity/` folder on your computer Desktop).
5. Launch **Unity Editor** by double-clicking the project name in the list (the first time after creating the project, the editor will start automatically).
6. Add **Cube** to the scene (see Fig.1.4).
7. Navigate to Explore the Unity Editor website and follow the tutorial using *Game3D* as a reference project.
8. Become familiar with:
   - Main windows in the *Unity Editor* (see Fig.1.3).
   - Term **scene**.
   - Navigation in the scene.
   - Tools to transform objects (to manipulate **GameObjects** in *Unity Editor*).

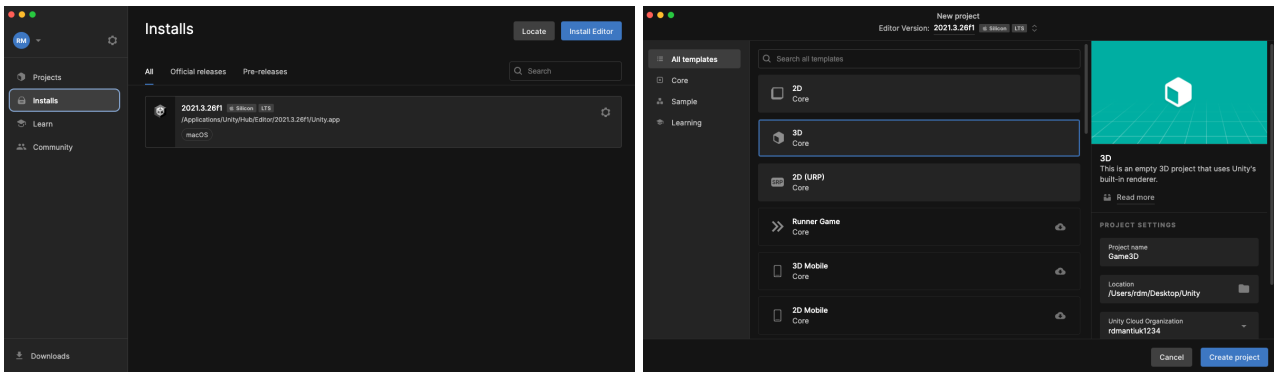

Figure 1.2: Left: Available Unity Editor versions. Right: Creating new project.

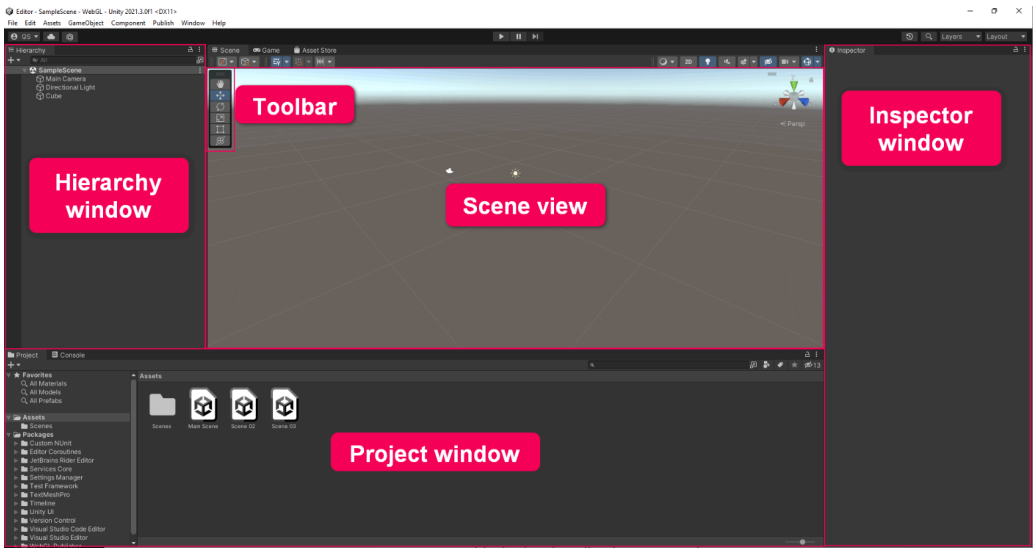

Figure 1.3: Unity Editor.

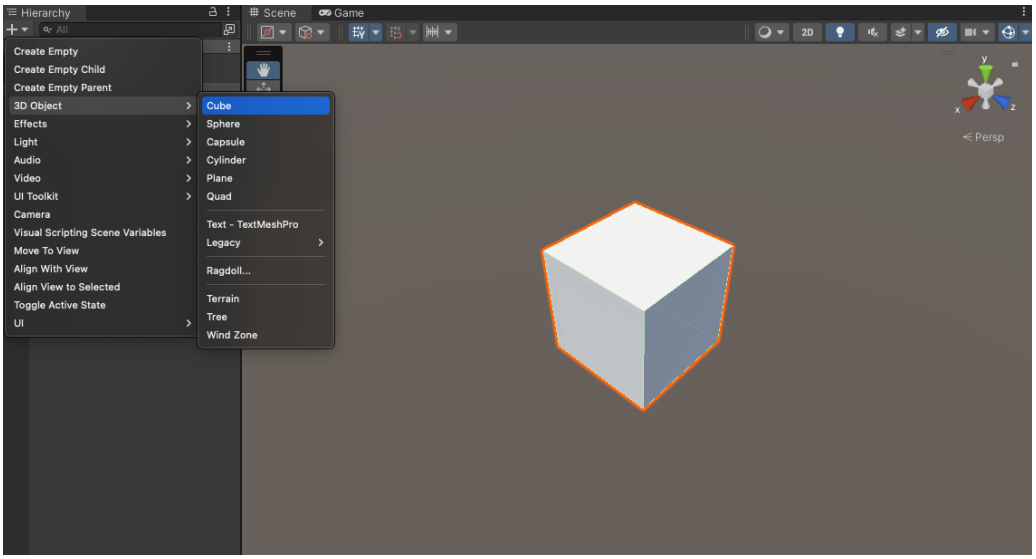

Figure 1.4: Adding a cube to the scene.

1.2 Gameboard structure

Objectives: Build a *GameObject* made up of multiple primitives. Create and manipulate GameObjects. Practice in navigating around a scene. Build a basic gameboard.

2. Exercise - Gameboard

1. Create an empty GameObject. It will be a placeholder object that can be created in the *Hierarchy* (see Fig.1.5(left)). Rename this GameObject to **Gameboard** in the *Inspector*.

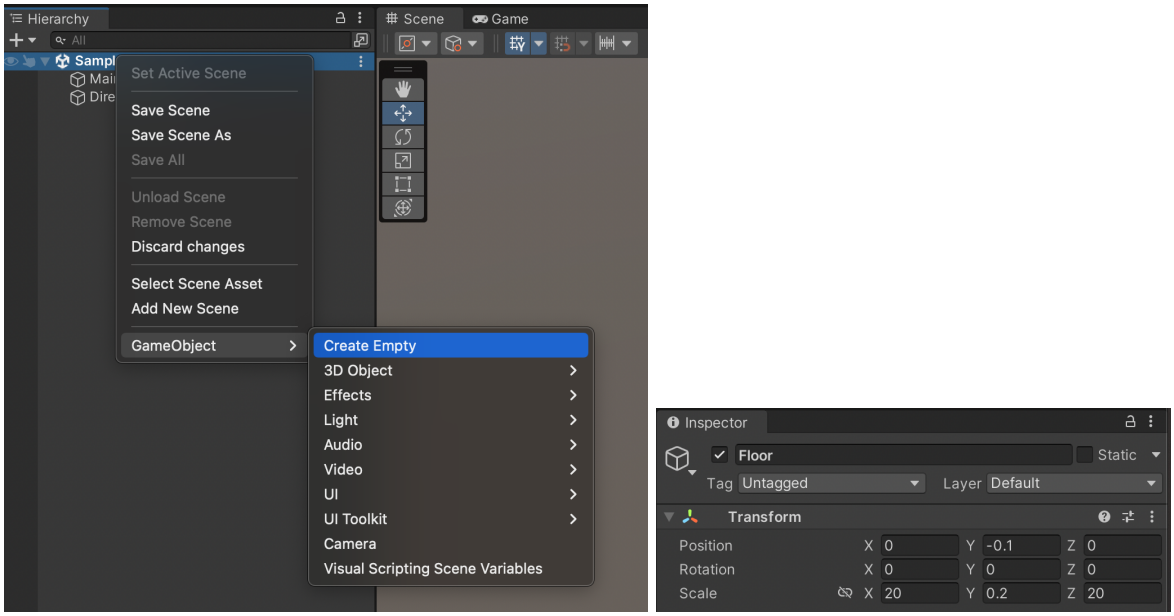

Figure 1.5: Creating empty object.

2. Select *Gameboard* in *Hierarchy* and create a 3D Object -> Cube (use [RMB]). Rename in to **Floor**. Scale this object in *Inspector* modifying **Transform** component to **Scale**=[20,0.2,20] and move to **Position**=[0,-0.1,0] (see Fig.1.5(right)).

3. Select *Gameboard* in *Hierarchy* and create another 3D Object -> Cube (use [RMB]). Rename in to **Wall-X**. Scale this object in *Inspector* to [1,2,20] and move to position of [9.5,1,0].

4. In the same way, create remaining three walls of the gameboard. They should be placed on the edges of the object and become and children of the *Gameboard* object. Objects cannot overlap and must be precisely connected. Use *Copy->Paste* and rotate objects for convenience. Rename them to match the object's orientation in the coordinate system: **Wall-X**, **Wall-XN**, **Wall-Z**, and **Wall-ZN**. See Fig.1.6 for the expected results.

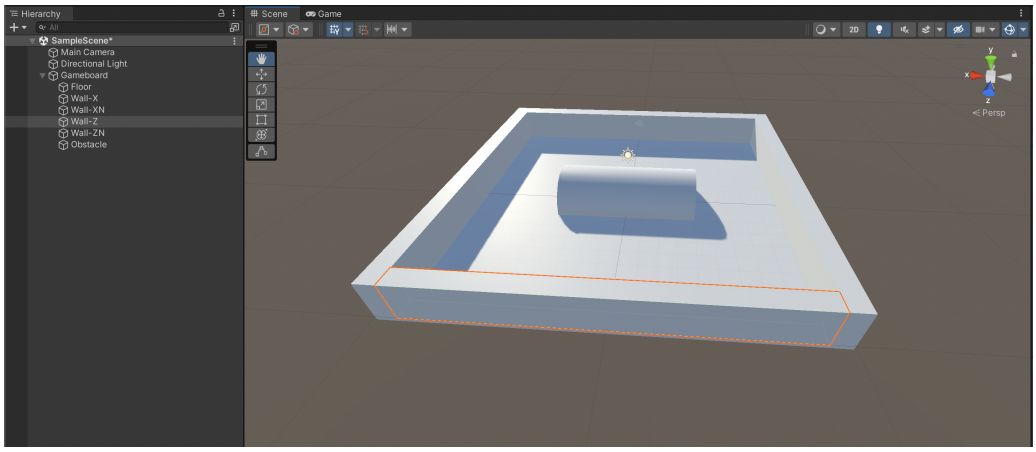

Figure 1.6: Gameboard structure

5. Select *Gameboard* in *Hierarchy* and create cylinder. Rename in to **Obstacle**. Change it scale to [4,4,4], position to [0,0,-1], and rotation to [0,0,90].

6. Select **Directional Light** in *Hierarchy*, rotate the light object in the Scene and adjust the illumination to your preferences (See Fig.1.7).

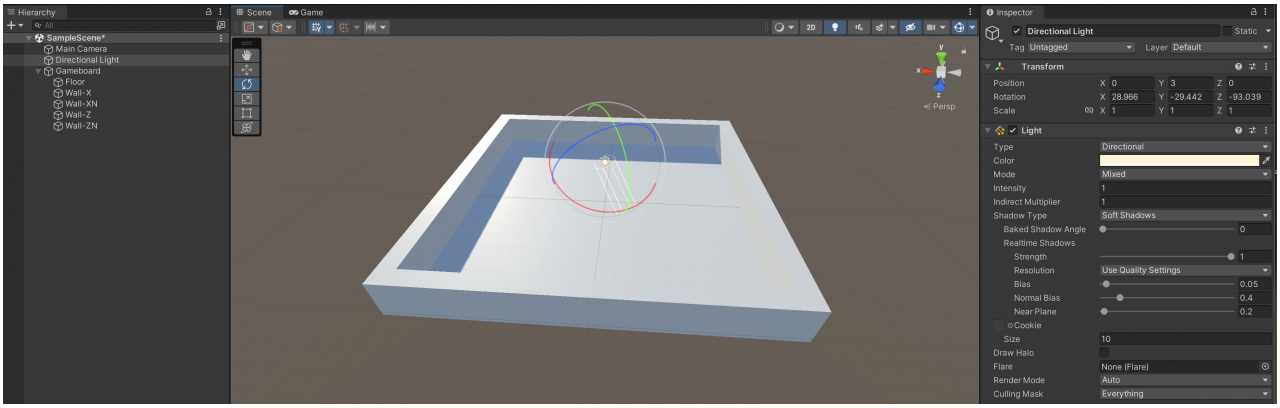

Figure 1.7: Scene illumination

7. Save the project to the repository in the `unity/` subdirectory. See GIT instruction to avoid storing temporal files (`.gitignore` must be defined).

8. Add the **"Exercise - Gameboard"** comment while performing the commit.

1.3 Rigidbody component

Objectives: Become familiar with the **Rigidbody** component, which gives a GameObject physical properties so that it can interact with gravity and other GameObjects. Learn how to create simple **materials**.

3. Exercise - Ball

1. Create a new sphere and rename it to **Ball**. Make sure it is at the top level and not a child of any other GameObject. Move the sphere to the space above the center of the gameboard so that it is positioned in

"mid-air". Use **navigation Gizmo** to see the top-view of the scene. You can switch between the projection and orthogonal views.

2. Set up **Main Camera** to view the *Gameboard* by selecting *Main Camera* in *Hierarchy* and moving-rotating the camera object to find the best view (see Fig.1.8). The **view frustum** of the camera should cover the entire *Gameboard*. Alternatively, you can set the same transformation for the camera as for the *Scene* view by pressing [Ctrl-Shift-F] when the *Main Camera* is selected in *Hierarchy*.

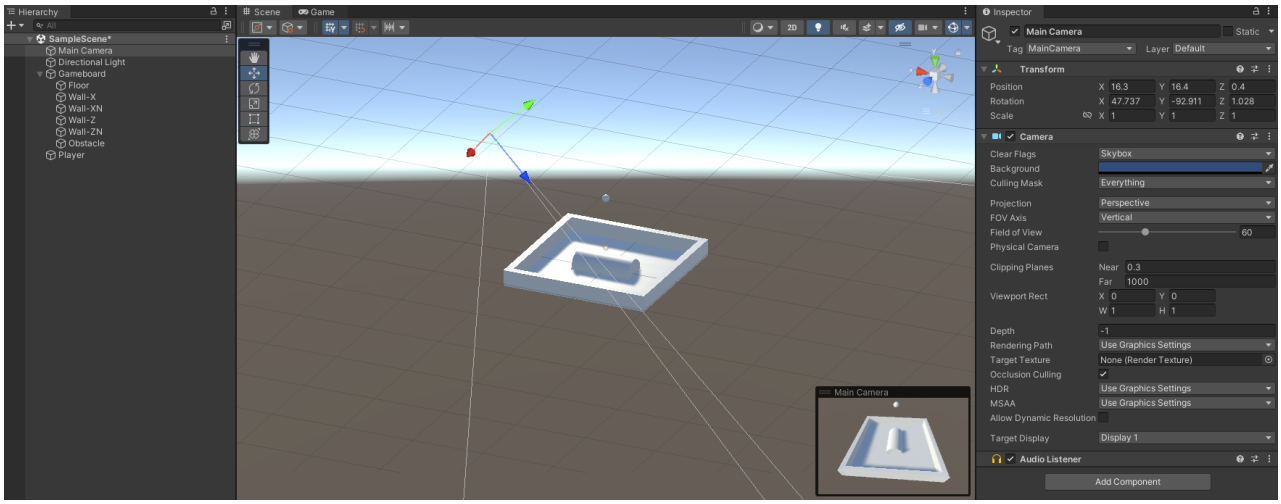

Figure 1.8: Camera setup

3. Give the *Ball* sphere mass by adding the **Rigidbody** component to the *Ball* object. Select *Ball* in *Hierarchy* and add *Rigidbody* in *Inspector* (see Fig.1.9).

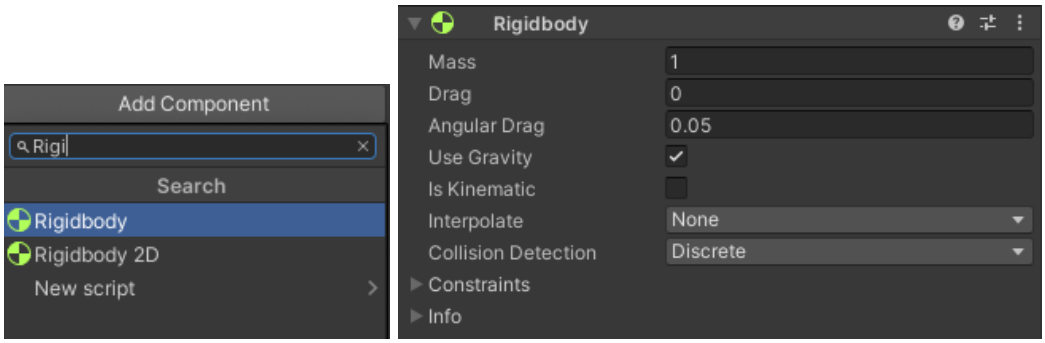

Figure 1.9: *Adding rigidbody* component to the *Ball* object.

4. Run the game by pressing **[PLAY]** button in the middle above the *Scene* window. The *Unity Editor* will automatically switch to the **Play mode** and run the animation. You can modify parameters in this mode (e.g., move objects) but when you press *[PLAY]* again, the settings will be restored to their original values. The changes are permanent only in the *Scene* mode. Notice, that the game is running all the time in the game mode.

5. Change the **playmode tint** color in *Preferences->Colors->Playmode tint* to easily distinguish edit mode from play mode.

6. In the *Scene* mode, select *Ball* object in the *Hierarchy* and add **New Script** component in *Inspector* and name it to **CBall** (see Fig.1.10). It will be also the name of the class. The script will be visible in **Project->Assets->CBall.cs**. Double-click it to open.

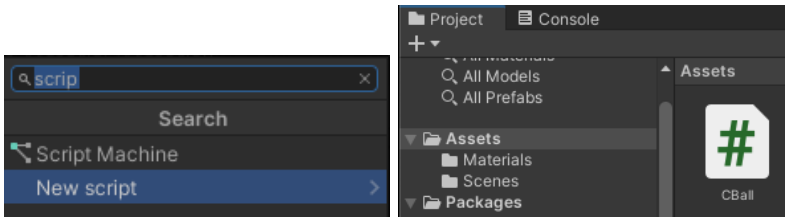

Figure 1.10: The script with `CBall` object.

7. Add the "`Debug.Log("Hello World");`" code to the Start function, and save the script:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CBall : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("Hello World");
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

8. Run the game. The **Hello World** will be displayed in **Console** window (tab next to *Project* - see Fig.1.10(right)).
9. Stop the game, modify the script according to the following listing and run the game again. Notice that the counter is incremented every rendered frame.

```
public class CBall : MonoBehaviour
{
    int frame_counter;

    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("Hello World");
        frame_counter = 0;
    }

    // Update is called once per frame
    void Update()
    {
        Debug.Log("frame: " + frame_counter);
        frame_counter += 1;
    }
}
```

10. Modify the script; change the *Ball* scale in the `Update()` method. The `changeScale` field is defined as the public variable, therefore, its value can be set in *Inspector*. Find **Change Scale** in the *Ball* properties and set its value to [0.001, 0.001, 0.001]. Finally, test the game - the sphere will keep getting bigger.

```
public class CBall : MonoBehaviour
{
    int frame_counter;
    public Vector3 changeScale;

    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("Hello World");
        frame_counter = 0;
    }

    // Update is called once per frame
    void Update()
    {
        transform.localScale += changeScale;

        Debug.Log("frame: " + frame_counter);
        frame_counter += 1;
    }
}
```

11. Create a new **material** and assign it to *Ball* object:

    (a) Create folder `Material` in *Project* window as a subfolder of `Assets` (see Fig.1.11).

    (b) Create new material in this new folder (use [RMB]->*Create*->*Material*). Rename it to **ball-material**.

    (c) Change the **Albedo** of the material in *Inspector* to light green color.

    (d) Assign material to the *Ball* object by dragging it from the *Project* window to the *Ball* object in the *Scene* window.

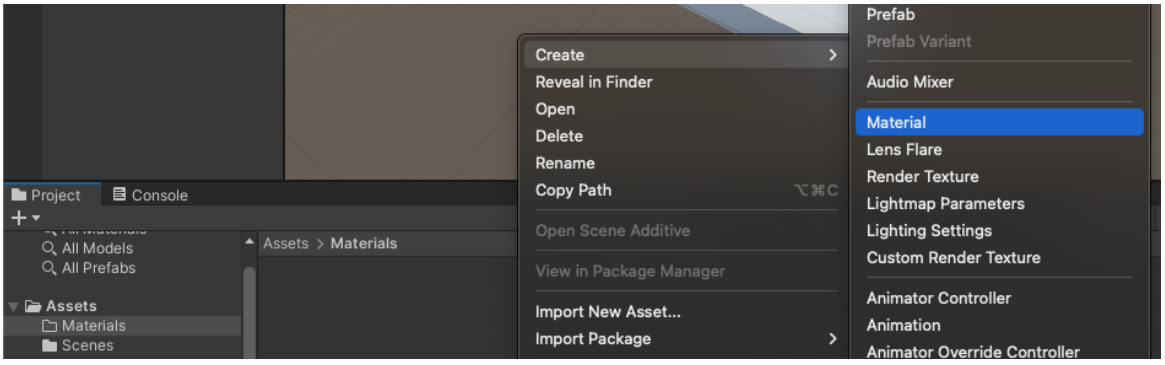

Figure 1.11: Adding new material.

12. Create a new material for *Floor* object:

    (a) Download `Herringbone_Brick_baseColor.png` file and drag it to *Project* -> *Assets* -> *Materials* window.

    (b) Add new material and name it **brick-material**.

    (c) In the *Inspector* window of the new material, select the object picker (circle icon) next to the *Albedo* property and select the Herringbone texture file that you imported.

    (d) Change tiling to [4,4].

    (e) Assign material to the *Floor* object by dragging it from the *Project* window to the *Floor* object in the *Scene* window.

13. Create two more materials and assign the to the walls and the *obstacle* object. Use textures downloaded from the Internet.

14. Save the project to the repository. Add the **"Exercise - Ball"** comment while performing the commit.

1.4 Player Controller object

Objectives: Learn how to control the player movement using the keyboard.

4. Exercise - Player

1. Add **Player** object to the scene:

   (a) Create empty object and renamed it to **PlayerController**.

   (b) Add *Rigidbody* component to this object.

   (c) Add new script and name it **CPlayerController**.

   (d) Add three cube objects and two cylinders to the *PlayerController* and transform this shapes to the object resembling a truck presented in Fig.1.12. Make sure that the object is created in accordance with the direction of the coordinate axis presented in Fig.1.12(right).

   (e) Create new materials and assign them to the truck parts.

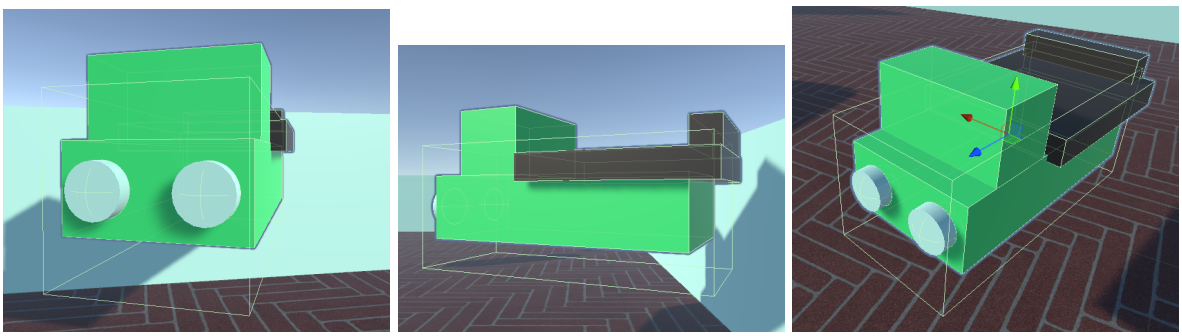

Figure 1.12: The *PlayerController* (truck) object.

2. Open *CPlayerController* script and modify the code in the script according to the following listing. The public variables `force` and `forceTurn` control the movement speed.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CPlayerController : MonoBehaviour
{
    Rigidbody rb;
    public float force = 1.0f;
    public float forceTurn = 1.0f;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    void Update()
    {
```

```
        // Reads the [WSAD] input to control the player movement.
        if (Input.GetKey(KeyCode.W)) {
           rb.AddForce(transform.forward * force);
        }
        if (Input.GetKey(KeyCode.S)) {
            rb.AddForce(-transform.forward * force);
        }

        if (Input.GetKey(KeyCode.D)) {
            rb.AddTorque(transform.up * forceTurn);
        }
        if (Input.GetKey(KeyCode.A)) {
            rb.AddTorque(-transform.up * forceTurn);
        }

    }
}
```

3. Run and test the game.
4. Read in the **Unity Documentation** descriptions of the `AddForce` and `AddTorque` methods.
5. Save the project to the repository. Add the **"Exercise - Player"** comment while performing the commit.

1.5 Camera

Objectives: Learn how to dynamically change the camera parameters.

5. Exercise - Camera

1. Create a new script called **CFollowPlayer** and attach it to the *Main Camera* object.
2. Add a line:

```
    public GameObject player;
```

3. Select the *Main Camera*, then, drag the *PlayerController* object from *Hierarchy* onto the empty player variable in the Inspector of *Main Camera* (see Fig.1.13).

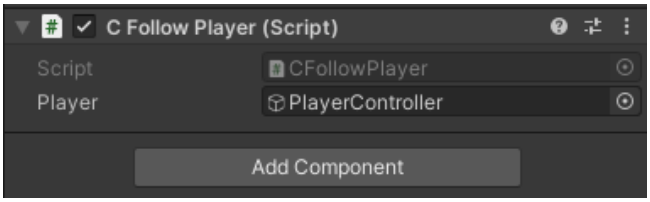

Figure 1.13: Associating *PlayerController* with the `CFollowPlayer` class.

4. In `Update()`, assign the camera's position to the player's position and run the game:

```
    transform.position = player.transform.position;
```

5. Add the camera offset position and rotation and run the game again:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CFollowPlayer : MonoBehaviour
{
    public GameObject player;
    public float offset = 10;

    // Start is called before the first frame update
    void Start()
    {
        transform.eulerAngles = new Vector3(80, 0, 0);
    }

    // Update is called once per frame
    void Update()
    {
       transform.position = player.transform.position + new Vector3(0, offset, 0);
    }
```

6. Modify the script to change the rotation of the camera according the movement of the truck. Use a rotation value (Y) from the *PlayerController* object to modify the camera rotation vector. The camera should rotate in the direction of the truck's movement (see Fig.1.14).
7. Save the project to the repository. Add the **"Exercise - Camera"** comment while performing the commit.

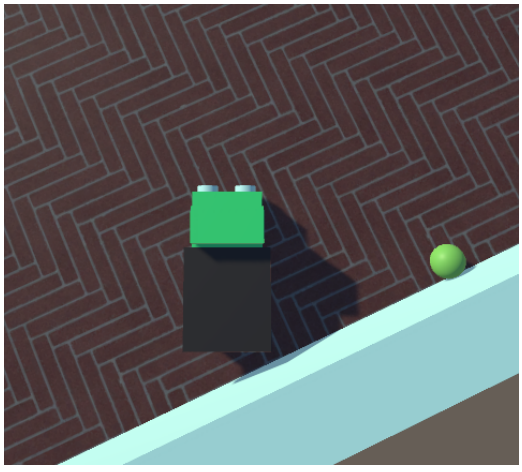

Figure 1.14: Dynamic camera top view.

1.6 Animation

Objectives: Learn how to add moving object to the scene.

6. Exercise - Animation

1. Create a new sphere object and name it **Enemy**. Position the *Enemy* over the *Floor*. Create a new orange (**enemy-material**) and add it to the *Enemy*.
2. Add *Rigidbody* component to *Enemy*.
3. Add new script to the *Enemy* object and name it **CMoveForward**.
4. Add global variable to the `CMoveForward` class:

```
public float speed = 3.0f;
```

5. Add the following line to the `Update()` method:

```
transform.Translate(Vector3.forward * Time.deltaTime * speed);
```

6. Run and test the game.
7. Save the project to the repository. Add the **"Exercise - Animation"** comment while performing the commit.

1.7 Prefabs

Objectives: Learn how to create **prefabs** (see manual for more details on prefabs).

7. Exercise - Prefab

1. Create a new **Capsule** object and name it **Tank**. Position the *Tank* over the *Floor* and scale it to [0.8,0.8,0.8]. Create a new orange material (**tank-material**) and add it to the *Tank*. See Fig.1.15(left).

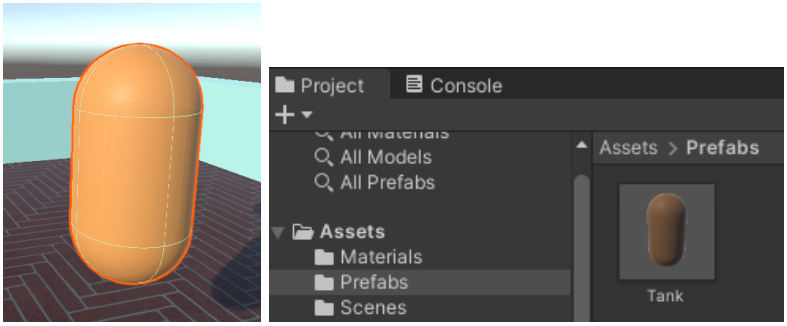

Figure 1.15: *Tank* object and tank prefab.

2. Add *Rigidbody* component to *Tank*.
3. Create a new **tag** and name it **tagTank** in *Inspector* (see Fig.1.16). Then, assign this tag to the *Tank* object.

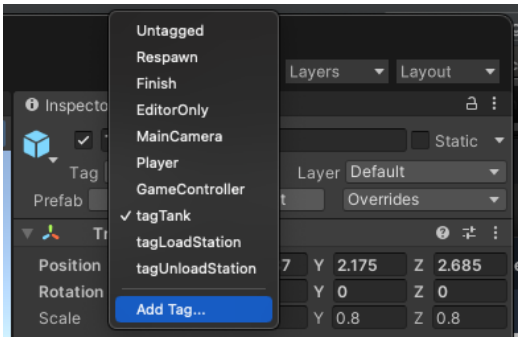

Figure 1.16: Creating and assigning *tag*.

4. Create a new folder **Prefabs** in *Project->Assets*.
5. Drag the *Tank* object from *Hierarchy* to new folder.
6. Double-click *Tank* prefab item in the *Prefabs* folder.
7. Enable **Auto Save** checkbox in the top-right corner of the *Scene* window.

8. Go back to *Scenes* living the prefab's edit mode.

9. Delete the *Tank* object from the scene in *Hierarchy*.

10. Save the project to the repository. Add the **"Exercise - Prefab"** comment while performing the commit.

1.8 Collisions

Objectives: Learn how to add collisions and triggers (see manual for more details on collisions).

8. Exercise - Collisions

1. Select *PlayerController* in *Hierarchy* and add the new **Box Collider** component in *Inspector* (see Fig.1.17).

2. Increase the size of the box collider along OZ axis to cover the entire truck.

3. Enable the **Is Trigger** checkbox.

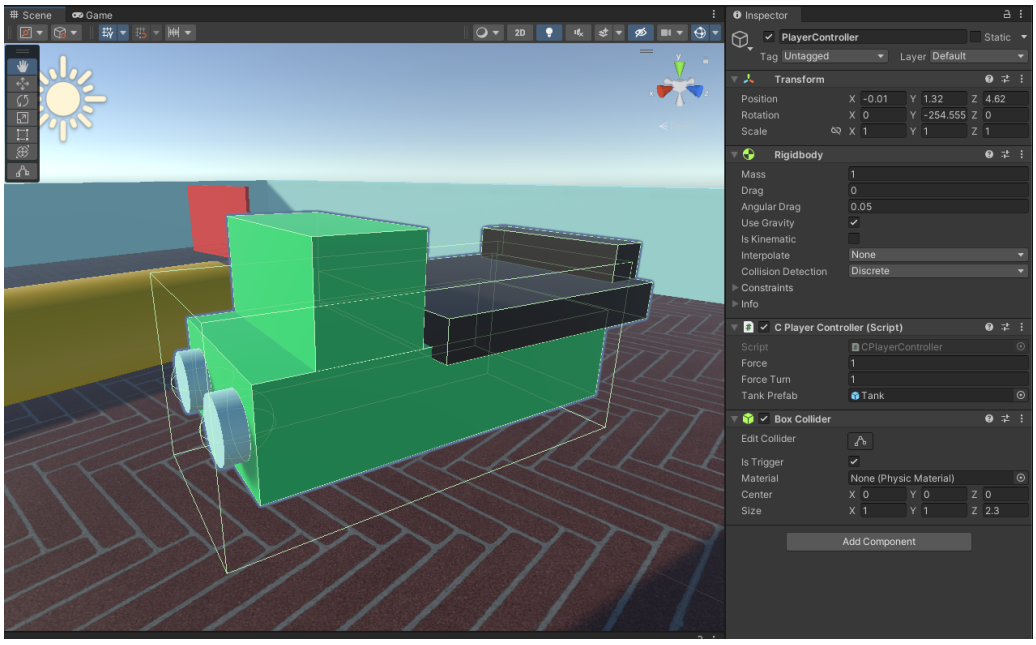

Figure 1.17: New *Box Collider* added to *PlayerController* object.

4. Create a new cube object and name it **LoadStation**. Create a new **tabLoadStation** tab and add it to the object. Create a new **load-station-material** (e.g., green) and add it to the *LoadStation* object (see Fig.1.18(left)).

5. Scale the *LoadStation* to [1,1,0.1] and place in one of the corners of the gameboard.

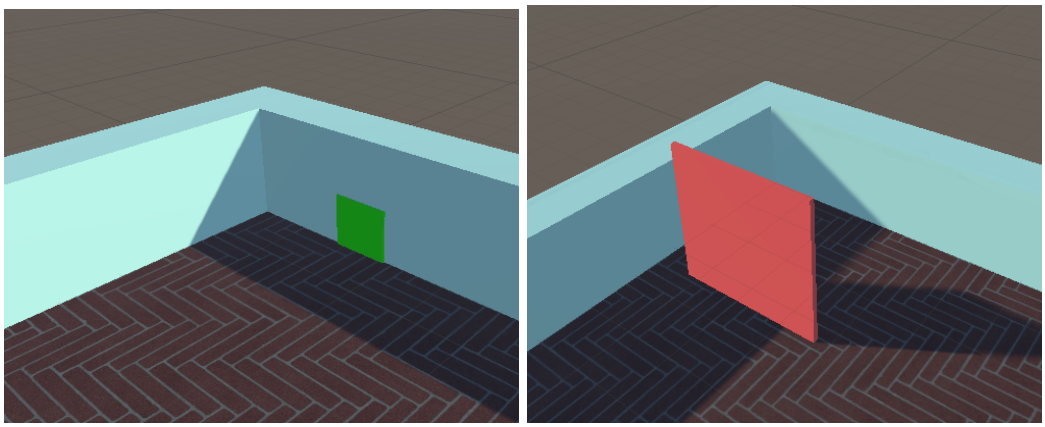

Figure 1.18: Location of the *LoadStation* (left) and *UnloadStation* (right) objects in the opposite corners of the *Gameboard*.

6. Create the similar cube object, name it **UnloadStation**, add **tagUnloadStation** tag, add **unload-station-material** material. The object should be placed in the opposite corner of the gameboard (see Fig.1.18(right)).
7. Enable *Is Trigger* checkbox in the *Box Collider* of the *UnloadStation* object.
8. Edit the *CPlayerController.cs* script and add the global variable to the *CPlayerController* class:

```
// set to true if PlayerController is in collision with the tank object
private bool isLoadStation = false;
```

9. Then, add the following methods. These methods are called when collisions between *PlayerController* and any object is triggered (`OnTriggerEnter()`), and when it is finished (`OnTriggerExit()`). The object with which the collision occurred is recognized by the tag name.

```
void OnTriggerEnter(Collider obj) {
    Debug.Log(obj.tag);
    if (obj.tag == "tagLoadStation") {
        isLoadStation = true;
        // changes color of the collided object material
        obj.GetComponent<Renderer>().material.color = new Color(1.0f, 0, 0, 1);
    }
}

void OnTriggerExit(Collider obj) {
    if (obj.tag == "tagLoadStation") {
        isLoadStation = false;
        obj.GetComponent<Renderer>().material.color = new Color(0, 1.0f, 0, 1);
    }
}
```

10. Add the public variable `tankPrefab` to `CPlayerController`. Then, drug the *Tank* prefab from the *Project* and drop it to the **Tank Prefab** variable in the *Inspector*.

```
    ...
    public GameObject tankPrefab;
    ...
```

11. Add the following lines to the `CPlayerController` class at the end of the `Update()` method:

```
        if (Input.GetKeyDown(KeyCode.Space)) {
            if (isLoadStation == true) {
                // set this object position associated with the PlayerController
                Vector3 pos = transform.position + new Vector3(-0.5f, 2.0f, 0);
                // set rotation
                Quaternion rot = Quaternion.Euler(90.0f, 0, 0);
                // create object using tankPrefab
                Instantiate(tankPrefab, pos, rot);
            }
        }
```

12. Run and test the game. When you press the [spacebar], a tank-like object will be created. This will only be possible when the truck collides with the *LoadStation* object.

13. Select the *UnloadStation* object in *Hierarchy*. Add a *New Script* component to this object, and name it **CUnloadStation**.

14. Edit the *CUnloadStation.cs* script and add the following method:

```
    void OnTriggerExit(Collider obj) {
        if (obj.tag == "tagTank") {
              Destroy(obj);
        }
    }
```

15. Run and test the game. Try to load the tank onto the truck and then drive the tank through the *UnloadStation*. The tank should disappear.

16. Save the project to the repository. Add the **"Exercise - Collision"** comment while performing the commit.

1.9 Timer and simple user interface

Objectives: Learn how to add a canvas with a simple UI in the form of a time counter (see manual for more details on UI programming).

9. Exercise - Timer

1. Launch **Package Manager** (*Window->Package Manager*) and install the **TextMeshPro** package (see Fig.1.19).

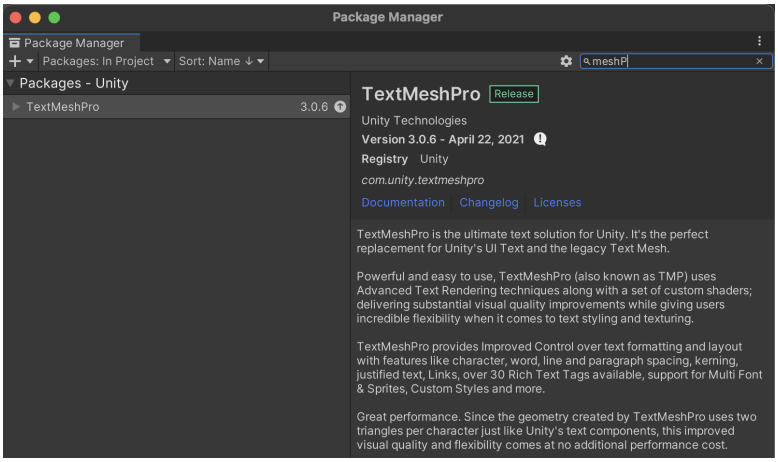

Figure 1.19: The *TextMeshPro* package installation.

2. In *Hierarchy*, add a new **UI->Canvas**, then add the **UI->Text - TextMeshPro** object to this canvas. Rename the object to **Timer**.
3. Select the *Canvas* object in *Hierarchy* and drag the *Main Camera* from *Hierarchy* to **Render Camera** in *Inspector*. Then, change the **Rendering Mode** to **Screen Space - Camera** in *Inspector*. Also change **Plane Distance** to 5.
4. Select the *Timer* object in *Hierarchy*. In *Inspector*, set text to *00:00*, *Font Size* to 50, and enable the *Outline* checkbox.
5. In the *Scene* view, move the *Timer* text to the top-left corner of the *Canvas*.
6. Add a new script to *Timer* and name it **CTimer**.
7. Edit the *CTimer* class and add the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public class CTimer : MonoBehaviour
{
    public float mTime;
    public TMP_Text mTimer;
    [SerializeField] TextMeshProUGUI obj;
    public bool isGameOver;

    // Start is called before the first frame update
    void Start()
    {
        mTime = 0;
        isGameOver = false;
        mTimer = GetComponent<TextMeshProUGUI>();
    }

    // Update is called once per frame
    void Update()
    {
         if (isGameOver == false) {
            mTime += Time.deltaTime;
            float minutes = Mathf.FloorToInt(mTime / 60);
            float seconds = Mathf.FloorToInt(mTime % 60);
            mTimer.text = string.Format("{0:00}:{1:00}", minutes, seconds);
         }
    }
}
```

8. Edit the *CUnloadStation* class and modify the code. When the tank passes through the *UnloadStation*, the `isGameOver` variable from the `CTimer` class will be set to true what will stop the timer.

```
    public CTimer mTimer;

    // Start is called before the first frame update
    void Start()
    {
        mTimer = GameObject.Find ("Timer").GetComponent<CTimer> ();
    }
```

```
void OnTriggerExit(Collider obj) {
    if (obj.tag == "tagTank") {
          Destroy(obj);

          mtimer.isGameOver = true;
    }
}
```

9. Run and test the game.
10. Save the project to the repository. Add the **"Exercise - Timer"** comment while performing the commit.

1.10 Game scenario and player score

Objectives: Implement a mechanism that allows you to compare the results of different players.

10. Exercise - Game Over

1. Create a variable defining the number of tanks to be delivered from the *LoadStation* to the *UnloadStation*. The next variable should record the current number of tanks transported. The values of these variables should be displayed next to the time indicator (see Fig.1.20).
2. After transporting all the tanks, display the words **Game Over** and the final game time.

Figure 1.20: The timer and the tank's counters (0 - collected, 2 - to collect).

3. Run and test the game a number of times. Calibrate the mass of objects and the force controlling the player.
4. Save the project to the repository. Add the **"Exercise - Gave Over"** comment while performing the commit.