Unity Tutorial

This tutorial will focus on creating a basic game using the Unity Engine

The player will be able to control a rocket using the Arrowkeys. The goal is to not hit any other objects. When moving too far away, the player will automatically loose health. Furthermore the spaceship should get destroyed, if 0 health is left. After the ship got destroyed, an explosion should appear as well as some UI Elements to allow the User to quit the application, or give the game another try.

Please note: This Tutorial refers back to the E-Portfolio about Unity from June 04 2018

GENERAL INFORMATION

- Use the Input Manager to control Inputs of the user, this allows you to set multiple buttons, as well as control settings on your commands
- Use tags to define your prefabs. Set e.g. player as player.
- Full code / project can be found on GitHub as well.

DOWNLOADING THE UNITY-ENGINE

Unity is available as download for free right here.

Installing Unity does require some specifications, which should be easy to have in 2018.

OS	32 / 64 bit	OS Version
Windows	64	7, 8, 10
MacOS	64	MacOS X 10.9 and following

Please note: Unity is available for Linux as an experimental version. Therefore using another OS is recommended.

Hardware Specifications are not mentioned on the Unity Homepage itself, but it should be obvious, that better hardware allows smoother work. Since I am only working on Macs, the minimum specification from my point of view is:

OS	Core	RAM	External Graphics
MacOS X 10.9+		4GB	No

Always make sure to include Build-Support during the Unity Engine for at least your development OS, so that you are able to deploy it directly.

STARTING A NEW PROJECT

After Unity was successful installed, simply open the Editor by hitting the Launch Icon.

The Editor will now open a new window, in which you can select existing projects to work on, or create new ones. In our case, we want to create a new project.

Therefore click on the „New"-Button on the upper right of the window and enter a name for the project. Choose a location on your device, where you want to save it and select „3D". Finally hit „Create Project" and Unity will automatically create a new project and open the Editor for you.

Please note: Within the „Create new project" window you are also able to add Standard Packages to your project.

SETTING UP OUR EDITOR-VIEW

HOW TO WORK WITH THE EDITOR

Now that we have a proper Layout to work with, what does each section represent? On the upper left, we have the so called **Scene View**. In here, we can see the game objects we have added to the scene. We are also able to change their position, drag and drop new Objects in here and build our scene together using different objects, animations and more.

Right beneath it you will find the **Game View**. Depending on the positioning of the camera, you can take a direct look on how the finished scene would look like. This helps you to get a better insight on how the objects look in game and if they are positioned correctly, or any other adjustments are needed.

The Layer in the middle on the top is our **Scene Inspector**. Here you can see all of your objects in the scene and make them a parent or child of other objects just by dragging and dropping. Beneath it is our overall **Project Folder** in which we can take a look at our Project Organization, assets and more. This is used to drag and drop general assets into a scene and modify them there.

On the far right you find the **Inspector**. The Inspector gives you additional Information on the object you have selected. E.g. if you have selected an Object of type Cube, you would have the possibility to change its dimensions, style and overall appearance. In addition to that, you can take a look at all applied components and their attributes as well

IMPORTING .BLEND-FILES IN UNITY

Importing and making use of .blend-files in Unity is pretty easy. Simply drag and drop the .blend-file in your Unity assets and you are done. You can either use them now directly in your project and place them in scenes, or manipulate them further by creating prefabs out of them.

CREATE PREFABS

Prefabs are truly essential in game development with Unity. A prefab is a specific data type, which lets you store a GameObject as well as all depending Components you have applied in one single file. If you e.g. create a planet and add a hit zone to it, you can save it as a prefab, so that you do not have to create the hit zone every single time you want to place a new planet and is instead saved in the „planet prefab“ already combined with the planet asset. Simply drag and drop a GameObject from your SceneView in your Assets-Folder and a prefab has been created.

SETUP THE SCENE

After you have setup the .blend-file and managed to create prefabs out of the individual parts, simply drag and drop the rocket in the SceneView. Make sure to also put a planet in your scene, since we will work on it as well.

SETUP THE PLAYER

Since we want to move our Player, we have to add several components to it. First of all, we want the Camera to follow the player, so that we are always able to see him. Furthermore we want to control the rocket using the arrowkeys. In addition to that, the player should have some kind of Death-Handling and therefore a kind of Health-Management System.

We start with the following **camera**, since it is pretty easy to achieve this. Simply drag the MainCamera, which has been automatically generated by Unity, and drop it beneath the Player, so that the camera becomes a child of the player / rocket.

Next we want the player to **move**. Some kind of Afterburner would be nice, so that the player automatically moves forward, without us doing anything. By doing this, we then only have to manipulate two axes, x and y.
For the Player Movement, we want to create a new C#-Script in our Asset Folder and call it „PlayerMovement“. Opening it up will start MonoDevelop. As long as you have not setup another Editor of your choice to create code, this is standard. Within the newly created C# script, we want to have a public variable for the movement speed.

```
Public float m_MovementSpeed;
```

Creating and using a public variable allows us to change its value during emulation in the editor and adjust it to our needs.
Within the automatically generated Update()-function, we simply want to move the rocket forward and check, if the values of the axes have changed, to rotate it.

```
transform.Translate(Vector3.right * m_MovementSpeed);
transform.Rotate(0, 2f * Input.GetAxis(„Vertical“), 2f * Input.GetAxis(„Horizontal“));
```

Finally apply the script as new component to the rocket and hit apply, to save it to the prefab.

Implementing the **Health** System of the player is a little bit more difficult. First of all we will need some public variables, to refer to some game objects during the game. We could also achieve this using a different script management, but due to the fact that we want to keep it simple and easy to understand, we will keep it like this.

```
Public float m_StartingHealth = 100f;
Public GameObject m_Camera;

Private float m_CurrentHealth;
Private bool m_Dead;
```

Within the **void Start()** we want to set the current health to the value of the starting health and also set m_Dead to false, since our player gets spawned and is therefore alive.

Since we do not need to check a condition every frame, we can simply delete the **Update()** function.

Since we have a private variable which holds the information if our player is dead, we want to create a **get** function for this.

```
Public bool getIfDead(){
        return m_Dead;
}
```

Due to the fact that our player should loose health if the rocket flies against another object, we need a public function to achieve this called **TakeDamage()**. We also want to control how much damage using a parameter whenever the function is called.

```
Public void TakeDamage(float amount){
        m_CurrentHealth -= amount;

        if(m_CurrentHealth <= 0f && !m_Dead){
                OnDeath();
        }
}
```

In addition to that, we want to check if the player is dead after he lost this amount of damage. If this condition is applicable, we want to call another function, which deals with the death handling itself called **OnDeath**.

```
Public void OnDeath(){
        m_Camera.transform.parent = null;
        gameObject.SetActive(false);
        m_Dead = true;
}
```

When the player dies, we want to set its gameObject inactive. This will automatically prevent it from rendering. The only issue is, that we have our MainCamera as a Child of this object. So, if we set the player inactive, the camera gets deactivated as well, which is a series issue. Therefore we first have to set the parent of the camera to *null* and then set the rocket inactive. After that, we set our boolean, if the player is dead or not, to true.

HEALTHBAR

Of course we want to see, if our player loses health or not. Therefore we will implement a Healthbar. Although this might sound a little bit tricky, it is pretty easy once you know how it is done.

The goal is to create a 2D Element in our 3D world and set it to ScreenSpace Overlay. This allows us to have it always in front of GameObjects during our game. We basically want to display it the value of our variable **m_CurrentHealth**, which we have just set up in our C# script *PlayerHealth*.

First of all, create a Slider within the SceneMenu and make sure in the Inspector, that it is set to ScreenSpace Overlay. If this is achieved, simply use its anchors on the inspector to place it on the upper left of the camera view. Now that we have this slider, we want to deactivate the ability of the user to manipulate it with the mouse. Therefore simply turn the component within the slider off and reset its Pivot arguments, so that it fills the whole space. Set different colors for „Fill“ and „Background“. In general, Background should be red, since „Fill“ will be modified by how much health the player has. Finally set the max Value of the Slider to the maximum value of the Player.

Within the PlayerHealth script we now want to access this slider and also manipulate its value when necessary. Since this is only necessary if the player loses or gains health, we create an update method of the HealthUI inside the „TakeDamage“ method.

```
Using UnityEngine.UI;

[…]

Public Slider m_HealthUI;

[…]

Private void SetHealthUI(){
        m_HealthUI.value = m_CurrentHealth;
}
```

Finally move over to the SceneView, select your Player, move your mouse over to the Inspector and set the created Slider in the scene as GameObject for the empty spot in m_HealthUI.

COLLISIONS

Interacting with other GameObjects is one of the main aspects in Game Development. One of the most easiest things to happen is the collision between different objects. We want to use collision to damage our player and stop him from moving through objects without doing anything.

Unity provides its own possibility of dealing with Collision. Within C#, you can check whether a Object is entering, leaving or staying within a dedicated zone. Due to the fact that these are normal functions, they also deliver a parameter of type „Collider“, which directly presents to you the object you collided with.

In the Unity Editor you simply have to make sure that the Object, on which these functions should be applied, has its „isTrigger“ checkmark set.

Since we only want to check, if a player collides with one of our planets, we create a new C# script called „Planets“, in which we check, if the colliding object is of type player. If so, we want it to take damage.

```
Void OnTriggerEnter(Collider other){
        if(other.gameObject.tag == „Player“){
                other.gameObject.GetComponent<PlayerHealth>().TakeDamage(100f);
        }
}
```

OUT OF WORLD

We use a similar function to check if our player has moved too far away from the game relevant space. We simply create a cube, turn off its rendering component, make it a trigger, scale it up and check, if an object with tag player exits the zone. If so, we want to take continuously take health of the player. If the rocket returns back in the zone, we want it to stop.

Therefore we simply declare a variable of type boolean, which provides the information if the player is outside the zone or not. So when leaving the zone, we set it to true. Inside the **Update()** method we then check, if this boolean is true. If so, we want the player to loose health. To get a reference on the player, we could either save the collider which is leaving the zone, or create a public variable which holds the information for us and then, again, access its component **PlayerHealth** and make it loose health by calling „TakeDamage()“.

```
Public GameObject m_Player;
Public GameObject m_Warning;
Public float m_DamageOverTime = 0.1f;

Private bool m_OutOfZone = false;

Void Update(){
        if(m_OutOfZone){
                m_Player.GetComponent<PlayerHealth>().TakeDamage(m_DamageOverTime);
        }
}

Void OnTriggerEnter(Collider other){
        if(other.gameObject.tag == „Player“){
                m_OutOfZone = false;
                m_Warning.SetActive(false);
        }
}

Void OnTriggerExit(Collider other){
        if(other.gameObject.tag == „Player“)
        {
                m_OutOfZone = true;
                m_Warning.SetActive(true);
        }
}
```

In this case we also added a UI Text Element and simply showed it, if the player moved out of the zone or hided it, if the player is present.

SETTING UP A BASIC MENU

Simply Create a Button, place it using the same operations as within the positioning of the Healthbar and rename them. If you want to add functions to them, add an event by clicking on the plus in the inspector where it says „Actions“. Drag and drop a game object onto it, which holds a script the function you want to call, or use a predefined one. Then simply select it - done.

Again: Make sure to set it to screen space overlay. For Further explanation on how to pause the game, take a look on the Gamemanager.

ADDING COSMETICS

Now that we have our game, it looks quite boring without any kind of animation. To add some Partical Systems, we now import the Unity Standard Assets for Particle Systems, providing already fully functional animations.

Therefore move up to the Menu, click on **Assets** > **Import Package** > **Particle Systems**.

Now simply drag and drop the finished prefabs you need into your scene and place them where you want. Use e.g. the *Afterburner* and place it behind the rocket. Do not forget to make it a child of the rocket, so that it moves with it (and also gets deactivated, when the player gets deactivated).

It would also be nice to have an explosion, if the player collides with something. Therefore we open up our c# script called **PlayerHealth** and add a new public variable of type GameObject. We will then move to our SceneView, select our Player, and drag and drop the explosion prefab we want from our assets folder in the empty slot.

In the script we now only have to declare, when this explosion should be instantiated. Since we only want it to be shown when the player dies, we place it in our DeathHandling method.

```
Public GameObject m_Explosion;

Private void OnDeath(){
        Instantiate(m_Explosion, transform.position, transform.rotation);
        […]
}
```

CREATE A BASIC GAMEMANAGER

Now that we have the different parts of the Game put together, we need some kind of management script, a Game Manager.

Since our game is pretty easy, the logic behind it is really easy as well. Again, we need a reference to our player. In addition to that we need a reference to our Menu as well, since we want it to be shown, once the player dies.

```
Public GameObject m_Player;
Public GameObject m_MenuUI;
```

Now we want to use the Update Method to check permanently, if our player has died. If the variable m_Dead is true, we want to wait a few seconds, since we want to see the explosion and then turn on the menu. If the player instead presses Escape on the keyboard, the application should be quitted directly.

```
Void Update(){
        if(m_Player.GetComponent<PlayerHealth>().getIfDead()){
                StartCoroutine(WaitForExplosion());
        }
        if(Input.GetButtonDown(„Quit“)) {
                Application.Quit();
        }
}
```

Coroutines can be used in unity to run either coroutines, or wait for them to do something and then do something. Since we want to wait for a few seconds, we do exactly this in a coroutine and then call a method, which pauses the game and opens up a menu.

```
Private IEnumerator WaitForExplosion(){
        yield return new WaitForSeconds(4f);
        CallMenu();
}
```

```
Private void CallMenu(){
	Time.timeScale = 0f;
	m_MenuUI.SetActive(true);
}
```

Within **CallMenu**, we only want to set the timeScale to 0f, since this pauses the overall game. So if there is anything else running in the background, it stops. Furthermore, we want to activate our Menu. Since this is only another game object, this is done within a second.
Here we also implement our other Menu Controls. In case of *StartOverNew* we simply want to set the timescale back to 1f, hide the menu and reload the scene.

```
Private void StartNew(){
	Time.timeScale = 1f;
	m_MenuUI.SetActive(false);
	SceneManager.LoadScene(„Demo“);
}
```