Masaryk University
Faculty of Informatics

GAME DEVELOPMENT TEACHING AT FI MU

Bachelor thesis

Milan Doležal

Brno, 2015

Annotation

The goal of this thesis is to prepare study materials and lesson topics for seminars of the subject "*PV255 Digital games*". The subject is taught since the semester autumn 2014 at the Faculty of Informatics at the Masaryk University. Students will learn basics of the game development in the Unity game engine. The topics of seminars are closely related to lecture topics.

Keywords

Unity, game development, seminar, digital games, videogames, gamestudies, indie games

Declaration

I hereby declare that this thesis is my original copyrighted work which I developed alone. All resources, sources, and literature, which I used are my own or quoted in the thesis properly, stating the full reference to the source.

Milan Doležal

Thesis supervisor: RNDr. Barbora Kozlíková, Ph.D.

Acknowledgements

I would like to thank my supervisor Barbora Kozlíková for leading this work and making it possible. I would also like to thank Antonín Hojný and Jiří Chmelík for teaching the first semester of the subject with me. And finally I would like to thank my family for encouraging me to keep studying even in times when I was convinced I can‘t pass finals.

Table of contents

1 Introduction

1.1 Game studies at universities

Over the last 40 years, digital games have been more and more popular. The game industry currently profits more than the movie industry. Also, the game as the medium can be more immersive, than any other medium. Unlike movies, which has its own field, they are still missing in the academia, especially at Czech schools. Some universities have one or two subjects dedicated to digital games as a medium but none for game development. The exception is the Charles University in Prague, where the subject named "*Computer Games Development*[1]" is taught.

The topic of studying games as a separate field in the czech environment tries to establish the group called "*Game Studies of Masaryk University*[2]", or "*MUGS*" for short. The group was officially founded in September 2012. This group of students organizes events related to games, they help other students with their game research or thesis and also help to create game subjects. They already have six digital game oriented subjects on the list[3] and more are coming. However, since they are placed at the Faculty of Arts, they are unfortunately mostly theoretically based.

1.2 Digital games development at Faculty of Informatics

To fill the gap in teaching the game development, "*MUGS*" organized with the help of the Faculty of Informatics the event named "*Game Making University*[4]" in the semester Autumn 2013. This event consisted of two lectures, one workshop and the daylong game jam, all led by experienced game developers. The first lecture was prepared by Vladimír Geršl, the developer from the Slovak game company "*Cauldron*", currently working in his own game company "*Fun2Robots*". His lecture was about the process of the development. The second lecture was prepared by Jarek Kolář, the designer from the game company named "*2K*". He talked about techniques in game design and how to make the game interesting. The workshop by Dan Doležel, who worked on the "*Mafia*" and "*Mafia 2*" videogames in the past, currently working as an independent game developer, showed basics of the "*Unity*" game engine. He used the game "*Arkanoid*" as an example. Students then used their knowledge from the workshop in the game jam, where they had to design and develop a prototype of an Arkanoid-like game in 8 hours. The game jam was led by Petr Benýšek, the programmer, who worked on big games like "*Silent Hill: Downpour*" or "*Vietcong*". Finished projects,

recorded lectures and workshop and interviews with lecturers can be viewed at "*MUGS*" web sites[1]. The interest between students in "*Game Making University*" surpassed the expectations. In the semester "*Spring 2014*", there was introduced a serious game project in the "*Human-Computer Interaction*" laboratory. The game named "*Newron*[5]" is suppose to help with the treatment of childern with autism. It consists of simple minigames like puzzle or Hanoi towers. Students participating at the project learn how to work in team on a game. Also, an application for the new subject "*PV255 Digital games*" dedicated to the game development was sent. The subject was approved for the following semester.

1.3 Structure of the Thesis

In the second chapter are described considered forms of seminars. It also explains why the Unity game engine was picked as the main software used for the education in the subject. Third chapter consists of practical lessons. In lessons 1 to 3 are explained essentials for working with Unity. Namely orientation in the editor in the first lesson, physics in the second lesson and approaches to make a 2D game in the third lesson. Fourth lesson is not included, since it is meant to be a lesson, where students present their projects. In the fifth lesson are explained approaches to make an animation and how to work with it. The sixth lesson consists of explanation of the game logic using level managers and game managers, it also shows use of differnt types of texts, loading files at runtime from the "*Resources*" folder and usage of sounds. In the lesson seven is shown, how to create a simple artificial intelligence. The lesson eight shows, how to build the game for different platforms and how to make a script, that is specific for certain platform. Ninth and tenth lessons are dedicated for second presentations of student projects, so it is not included. Eleventh lesson object states, which are widely used in point&click adventure games. In the twelfth lesson is talked about useful plugins, that can simplify and speed up the development. And in the final thirteenth lesson, approaches to create an own plugin are described.

[1] http://gamestudies.cz/game-making-university/

2 Form of the education

2.1 Lectures and seminars

The subject is separated into lectures and seminars. The lectures cover the theoretical part of the game development. For some lectures, the developers from several game companies are invited. Seminars are taught by students and present the practical part of the game development. They learn how to properly use the Unity game engine and how to work in teams.

2.2 Form of seminars

The purpose of this thesis is to form the content of the subject seminars and to prepare study materials for them. The text part of the thesis should work as a referrence for teachers, who want to teach this subject, as well as for students. During the preparation of the seminars, three concepts were formed.

2.2.1 Small simple projects

First concept consists of simple games on which the teacher explains the current topic. Examples of such simple games are "*PacMan 3D*", "*Physics Labyrinth*" or "*Arkanoid*". This concept was scrapped after the discussion with game developers, who suggested better ideas.

2.2.2 One project and assignment

Another concept consists of one unprompted game project on which the teacher explains the current topic. Next to the example project, students work on their own game for a half of the session. This concept was tested with the help of "*MUGS*" with volunteers. The downside of this approach is its small flexibility. If for example the project is the first person shooter game, the principles used in 2D platformers[2] for instance are difficult to include.

2.2.3 Simple examples and individual work, approach used in practice

As the cooperating game developers suggested, the motivation for students is mainly the creation of their own game. That's why this concept focuses primarily on that. When working on their own project, students find the information by themselves. Seminars teach only simple examples of the current topic, but it also gives students a space for discussion about their projects. For that reason, this concept was used in practice and it is the main part of this thesis.

[2] 2D side scrolling game like Mario (Nintendo), Sonic (Sega) or more recent Trine (Frozenbyte) where the character jumps between platforms to progress.

2.3 Why Unity

While deciding, which game engine will be used for the subject, four of the currently most used engines were considered. Namely "*Unreal Engine 3*", "*Unreal Engine 4*", "*Cry Engine 3*" and "*Unity 4.5*". Unity became the best candidate, because of its easy to use environment, the possibility to export the game into different platforms, free version for commercial use and the support of three different programming languages. Supported languages are "*C#*", which is used in the subject, "*UnityScript*", the extended version of "*JavaScript*", which is a good choice for beginner programmers and "*Boo*", which has very similar syntax to the language "*Python*". Thanks to the simplicity of the Unity game engine, seminars can focus on the game logic, instead of the language itself.

3 Seminars

As mentioned earlier, the main focus of seminars is to create an own game. The game is developed in groups of three to four people. The first prototype is presented in the fourth week of the semester in the form of the presentation and playable game with at least one essential working game mechanic. In the tenth week, students show the "*Vertical slice*" of their game. It is the nearly finished version of the game. For the purpose of the presentation, students are asked to hide known bugs and prepare a level for the presentation, where everything works without problems. For that reason, the ninth lesson is dedicated for consultations of the projects, so students can prepare for their presentations in the tenth week. The content of the other lessons is described in the following chapters.

3.1 Lesson 1 - Unity basics

In the first lesson, students will familiarize themselves with the Unity game engine. They will learn where to find useful assets for their project, basic controls in editor, how to create first script for writing some message into console and the basic information about the Terrain editor.

3.1.1 Where to look for assets

When starting to develop a game, there is no need to use final versions of assets. Those may be used in the final product. In the first phase of the development, the functionality is more important than the aestetic appearance. To look for some product placement assets, the official Unity Asset Store[6] can be used. In the Asset Store, simple 3D models, sprites (2D images), or even scripts can be found. Some of them are freely available and the more complex ones are paid. For example, when a game developer wants to develop a game controlled by the Kinect for Windows device, he or she can easily download the free asset and basic Kinect control is ensured. Currently, there are many web sites to look for free assets. For example, for music, there is Soundcloud or for royalty free 3D models, TurboSquid may be used.

When got stucked, mostly with scripting, Unity has a great community of developers who are actively participating on the Unity forum, wiki or documentation and with high probability, the answer to the same problem is already there.

As I already mentioned, Unity has a very rich documentation. Every function in scripts as well as in editor is well documented, including a simple example of usage. The documentation is divided into the Scripting API[3] and the Manual[4]. The Scripting API documents the available functions in all supported languages. The Manual focuses mostly on editor rather than scripting. So the Manual is a good place to start searching for answers for your questions.

3.1.2 Creating New Project

A game in Unity is considered as a project. The project may contain one or more scenes. The scene can be a game level, a menu or even a death screen.

To create new project, launch Unity. The window with all opened projects appears. Select tab "*Create New Project*" (Fig. 3.1.1) and by clicking "*Browse*" button, choose empty folder where the project will be stored. From the list of packages, pick "*Character Controller*" and "*Terrain*

[3] http://docs.unity3d.com/ScriptReference/index.html
[4] http://docs.unity3d.com/Manual/index.html

Assets". Those are packages that are going to be used in first lesson. Character Controller package imports basic prefabs with applied scripts to move with a character and Terrain Assets contain a simple tree model, some grass and terrain textures. Press "*Create*" button and the window with Unity editor appears. If this window was opened without asking where to save new project, select "*File/New Project*" to create new project.

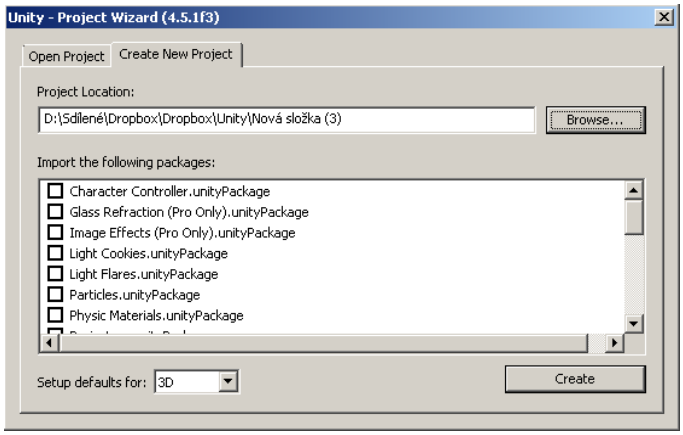

Fig. 3.1.1.: "Create New Project" tab

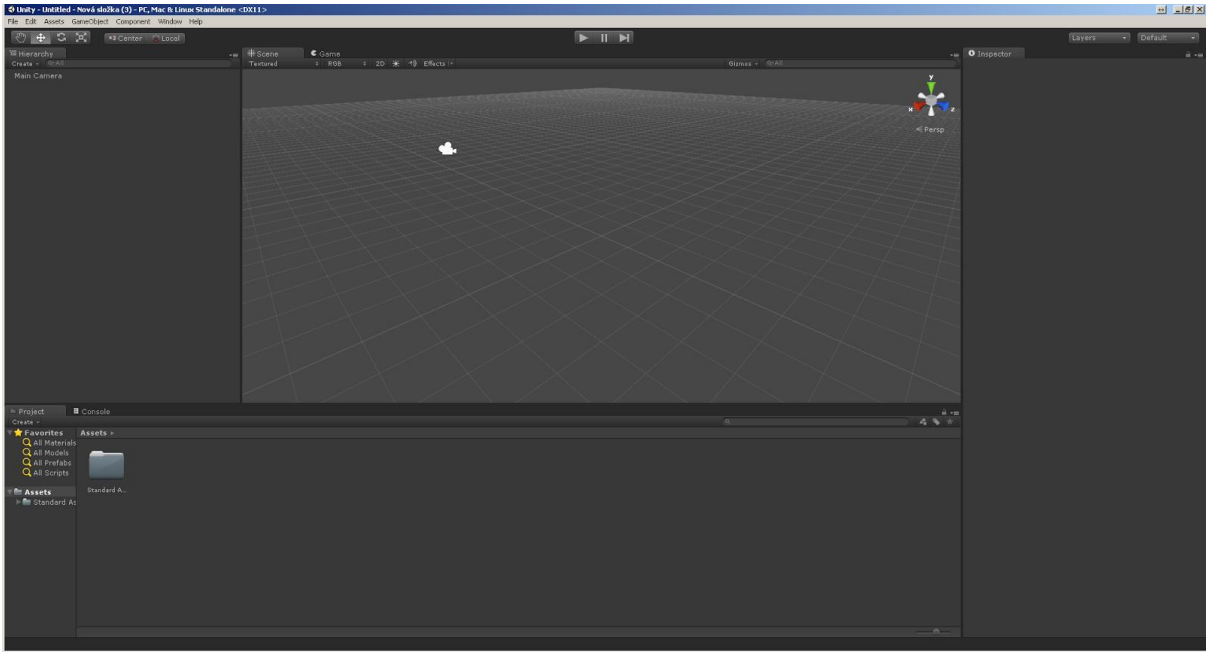

Fig. 3.1.2.: Unity editor window with "Default" layout

3.1.3 Editor

The Editor window (see Fig. 3.1.2) consists of several tabs. These tabs are customizable. The user can change its width and height or, by holding the left mouse button on its top tail, move the whole tab wherever needed. The most important tabs are:

3.1.3.1 Scene

The Scene tab provides a preview into the virtual 3D world of current scene. The objects are highlighted by axis arrows when selected. The newly created scene contains only "*Main Camera*" object that is displayed as white camera icon.

3.1.3.2 Hierarchy

In the Hierarchy tab we can find every object, that is located in the currently opened scene. It is represented by its name. It is allowed to have multiple objects with exactly the same name in one scene. Newly created scene contains only the "*Main Camera*" object.

3.1.3.3 Inspector

When an object in the Scene or Hierarchy is selected, the components of this object are shown in the Inspector tab. "*Main Camera*" includes components "*Transform*", which consists of position, rotation and scale, "*Camera*", a script that captures and displays the virtual 3D world to the player[7], "*GUILayer*", another script to read GUI elements, "*Flare Layer*", which makes Lens Flares visible in the image and "*Audio Listener*" handling sounds coming from various sound sources.

3.1.3.4 Project

The Project keeps all assets that can be used in scenes. Not all assets need to be necessarily used in the current scene and not all objects in scenes need to be in the Project. The Project is basically a folder, from where assets could be put into selected scenes.

3.1.3.5 Game view

The preview of the scene from gamer's view is available when pressing the "*Play*" button. At least one camera needs to be in the scene to be able to see some result in the Game view. Game view makes easier to create scenes without need of building the whole project. All scripts must be compiled correctly to be able to launch the Game view.

Important note: When performing some changes in the scene (e.g., changing some object's position) and the Play mode is in progress, then these changes won't be saved. To make permanent changes in scene, we have to stop the Play mode.

3.1.3.6 Console

Writes messages, warnings and errors.

3.1.4 Hello World - The first game

The first game, that is usually made in Unity, is fairly simple. It consists only of a Cube object as floor, a prefab[5] named "*First Person Controller*" and a "*Directional Light*".

The game can be created by following these steps:

Create the Cube object by selecting "*GameObject/Create Other/Cube*". Select the Cube in the Hierarchy or Scene to see its components. Scale the Cube to flat floor by setting values of Transform and Scale to X = 20, Y = 0.1, Z = 20.

Now, put the "*First Person Controller*" prefab into the scene from the Project. If it was correctly imported when new project was created, it is located in "*Assets/Standart Assets/Character Controllers*" folder. Move the First Person Controller slightly above the floor so it doesn't fall through. The Main Camera object is now useless, because the First Person Controller has its own Camera, so delete the Main Camera. Add the light from "*GameObject/Create Other/Directional Light*". Now if Play button is pressed, the game is playable in the Game view. The First Person Controller has standard FPS controls - 'W', 'S', 'A', 'D' for movement and mouse for looking around.

To familiarize with scripting, create a new C# script. Scripts can be created within the Project tab. Press right mouse button in the Project tab and select "*Create/C# Script*". This creates a new script. Name the newly created file and open it by double clicking on it. With default settings, MonoDevelop opens the script. Two fuctions are already there. First, the "*Start()*" function executes only the first frame, when it appears on the scene during runtime. On the other hand, the "*Update()*" function executes every frame, if it is on the scene during runtime. Add line to write some text into those functions. Write into the Start function the following line:

[5] Prefab is pre-made asset with set of components and values. The user can make his or her own prefabs and instantiate them into the scene.

```
Debug.Log("Hello");
```

Write into the Update function the following line:

```
Debug.Log("World");
```

Now back in editor, attach this script to any object in the scene (e. g., First Person Controller) simply by by using the drag and drop function. The attached script should be visible in the Component tab of an object. In the Play mode, select the Console tab and see messages written there (Fig. 3.1.3). First message says "Hello", all other messages say "World". This corresponds to the repeating execution of the Update function.

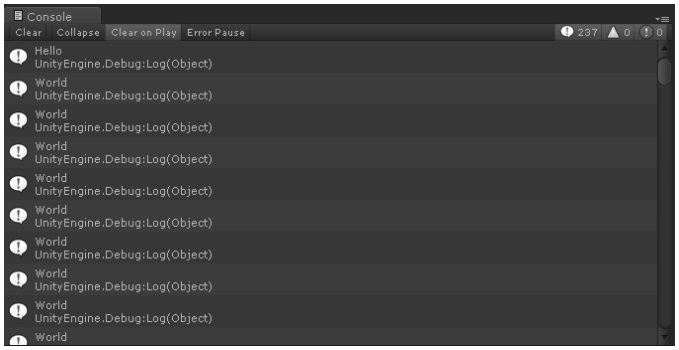

Fig. 3.1.3.: Console

3.1.5 Terrain

To make things more interesting, we can add the Terrain into the scene. The Terrain can be created by selecting "GameObject/Create Other/Terrain". The default terrain is not too big to cover the simple scene. Select the Terrain in the Hierarchy or Scene view to view its components. It has the "Terrain (Script)" component. Press the "wheel" icon to show its settings (Fig. 3.1.4). Change resolution by rewriting values "*Terrain Width*" and "*Terrain Length"* to something smaller (e.g., 200 x 200). Other buttons serve as tools to edit the Terrain.

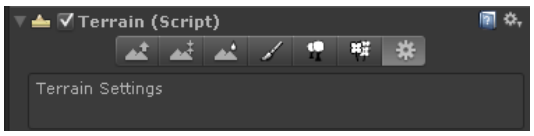

Fig. 3.1.4.: Terrain Settings

3.1.5.1 Raise / Lower Terrain

Makes hills and valleys in the Terrain.

3.1.5.2 Paint height

Makes hill/crater of defined height.

3.1.5.3 Smooth Height

Smooths the terrain deformations.

3.1.5.4 Paint Texture

Paints selected texture on the terrain. To add texture, press "*Edit Textures.../Add Texture...*" Pick the desired texture and press "*Add*" button. The first texture added will be painted on the whole terrain. Add another texture, select it in Inspector of the Terrain and paint it with brush within the Scene view.

3.1.5.5 Place Trees

Similar as "*Paint Texture*". Add tree game object into the terrain by clicking "*Edit Trees.../Add Tree*". Drag and drop the tree game object into "*Tree*" and confirm by pressing "*Add*". Now, when added tree is selected, paint the whole forest into the terrain.

3.1.5.6 Paint Details

Similar as "Place Trees" to paint grass.

3.1.6 Unity and Git

To be able to work in team, some version control is necessary. Git is a great tool to use with Unity projects. In order to use it correctly and avoid the loss of data, some settings need to be done first. In "*Edit/Project Settings/Editor*" change "*Version Control*" to "*Visible Meta Files*" in order to see meta files by Git. Also change "*Asset Serialization*" to "*Force Text*". This allows to see changes in most of files, even materials or scenes. Some folders shouldn't be versioned. For this reason, ".gitignore" file is needed to exclude "*Library*" and "*Temp*" folders from versioning. Also files with extensions like "*.svd", "*.userprefs", "*.csproj", "*.pidb", "*.suo", "*.sln", "*.user", "*.unityproj" or "*.booproj" can be ignored because these files are re-generated each time when MonoDevelop or Visual Studio is synchronized with Unity editor[8].

3.2 Lesson 2 - Physics

The second lesson introduces the basics of physics in the Unity game engine. It will consist of the explanation of "*Colliders*", "*Triggers*", "*Rigidbody*", "*Physics Material*", "*Raycast*", "*Tags*" and "*Layers*".

3.2.1 Colliders

It is natural that two objects can collide with each other. To be able to simulate the collision between two objects, these objects need to have a component called "*Collider*". It is represented in the Scene as a green line (Fig. 3.2.1) and it is the physical border of an object. There are several types of basic colliders (e.g., "*Box Collider*", "*Sphere Collider*"), some of them are more complicated, e.g., "*Mesh Collider*". The basic colliders create the primitive shape collider. The mesh collider on the other hand generates the collider from polygons of the object. Because of that, mesh colliders are more precise, but also more computationally demanding.

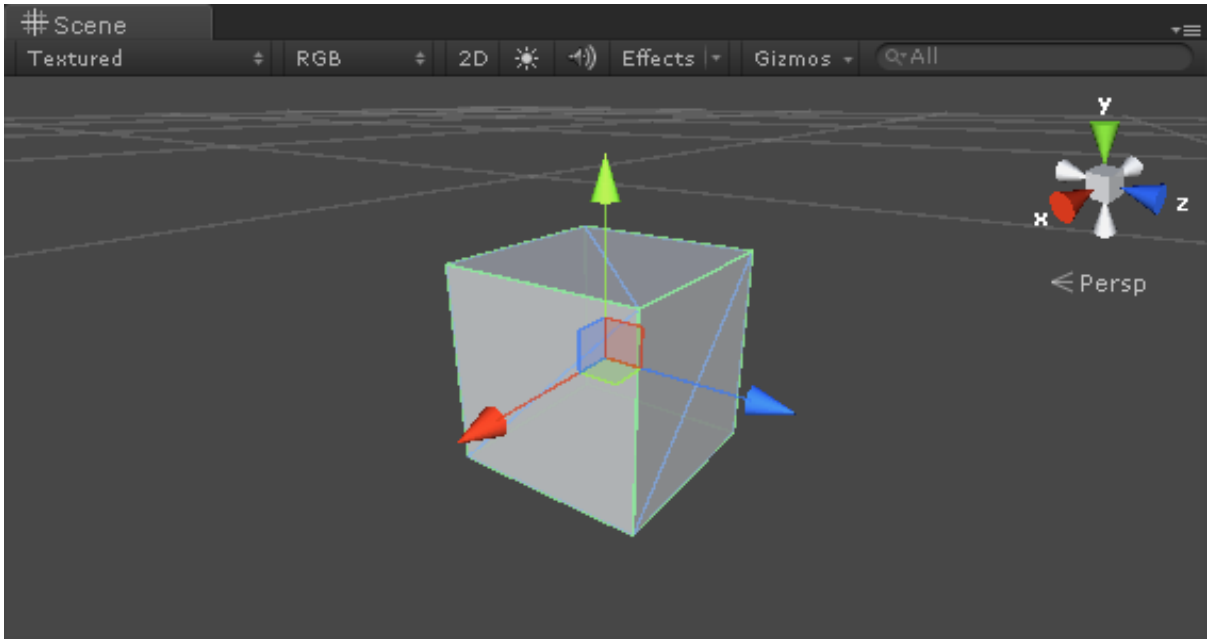

Fig. 3.2.1.: Box with the "Box Collider" component attached

To test how colliders work, create a cube, name it "*ColliderCube*" and move it to the position X = 0, Y = 1.5 and Z = 0. It already has attached the Box Collider component. Add another cube and shape it into a flat floor as done in Lesson 1. And finally, add the First Person Controller prefab from Standard Assets. It also possess a collider called "*Character Controller*". This is a type of collider specific to controlled characters. First Person Controller,

ColliderCube and Floor now all have colliders. In the Play mode, try to walk into the cube. It is not possible to walk through, but it is possible to jump on it.

Another possiblity is to create a script which will react on the collision. Stop the Play mode and create a new script. Name it "*CollisionReaction.cs*" and attach it to the ColliderCube. In the script, create a new function:

```
void OnCollisionEnter()
{
    Debug.Log("Colliding with something");
}
```

Function "*OnCollisionEnter()*" executes when any object starts colliding with the object having attached this script. Both objects must have the collider component. In the Play mode, if the First Person Controller collides with the ColliderCube, the debug message is written on Console.

3.2.2 Triggers

Triggers are similar to colliders. The difference is that objects can pass through objects with triggers. In games, this could be used to open the door when the player gets in a predefined proximity (enters the trigger).

To demonstrate the work with triggers, create a sphere and name it "*TriggerSphere*". Move it close to the First Person Controller. It has the "*Sphere Collider*" component. Check the checkbox "*Is Trigger*" in this component. Now the collider of this object becomes a trigger. In the Play mode, the player is able to walk through this object.

Again, a function from the script can be called, when something "*enters*" the object's trigger. Create a new script named "*TriggerReaction.cs*" and attach it to TriggerSphere object. In the script, create the following function:

```
void OnTriggerEnter()
{
    Debug.Log("Entered trigger.");
}
```

The "*OnTriggerEnter()*" function executes when any object enters the object's trigger having attached to this script. Both objects must have the collider component. In the Play mode, if the First Person Controller enters the TriggerSphere, the debug message is written on Console.

3.2.3 Rigidbody

So far our objects always floated in the air. If we want to add the ability to fall down like in real world because of gravity and correctly interact with other objects, the "*Rigidbody*" component must be attached.

To demonstrate this, add a sphere and two cubese to the scene. Scale one cube to the shape of a simple platform (e.g., X = 1, Y = 0.1, Z = 3) and rotate it a little bit as seen in Fig. 3.2.2.: Objects with rigidbodies and physics materials (e.g., 10 degrees around the X axis). Move the cube above the platform. Add the Rigidbody component to the cube and the sphere. In the Play mode, the cube as well as the sphere fall down. The cube falls on the platform and slides on it until it falls down.

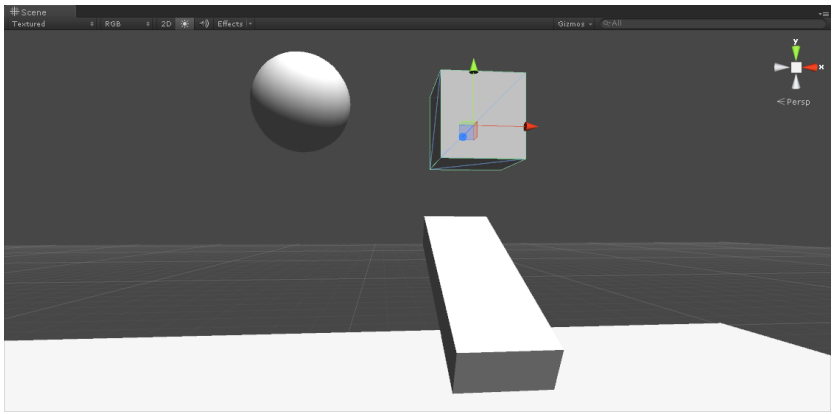

Fig. 3.2.2.: Objects with rigidbodies and physics materials

3.2.4 Physics Material

Different types of objects can interact in different ways. For example, if ice cube falls on a bent floor, it slides on the floor. However, if the cube is not ice but some kind of bouncy blob, it bounces on the same floor. For this reason, Unity provides the "*Physics Materials*". Few examples are included in Standard Assets (if they are not in Assets folder, import them).

In the scene, make a cube. Add another cube above the platform. Add the "*Ice*" Physics Material to its Box Collider component (Fig. 3.2.3). Also attach the Rigidbody to make it fall on the platform.

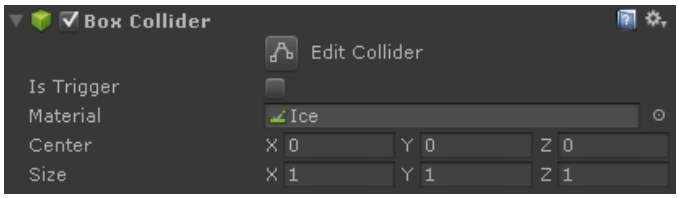

Fig. 3.2.3.: "Box Collider" component with "Physics Material"

In the Play mode, the cube falls on the platform and slides on it until it falls.

Stop the Play mode and change the Physics Material to "*Bouncy*". Now in the Play mode, the cube bounces on the platform.

3.2.5 Raycast

Objects are able to check, if they can see each other. For this, the "*Raycast*" option is used. It throws a ray from the object's center to the other object's center. Raycast can have predefined length. It can get to the closest object the ray reaches.

To see the Raycast in action, add a cube to the scene. This is going to be the cube that is throwing a raycast in its Z axis[6]. To make it more interesting, it will also check, if the ray hits the player or something else and writes a debug message to the Console accordingly.

To test this function, create a new script, name it "*RaycastExample.cs*" and attach it to the cube. Open the script in editor and in Update() function, add following lines:

RaycastHit hit;

```
    if (Physics.Raycast(transform.position, transform.forward, out
hit, 10))
    {
        if (hit.transform.name == "First Person Controller")
            Debug.Log ("Player is in front of the cube");
        else
            Debug.Log ("There is something in front of the
object!");
    }
```

[6] Z axis can also be described in Unity as the "*forward*" vector.

RaycastHit stores informations about the object that the ray hits. If ray hits some object, the condition of the function is met and the block is executed. The Raycast inside the "*if*" statement takes four parameters. The first parameter is its point of origin. From this point, the ray starts. The second parameter is its direction. The "*transform.forward*" parameter access the object's Transform component and gets the direction of its Z axis. The third parameter sets values of the "*hit*" variable and the last parameter represents the maximal distance the ray can reach. Inside the "*if*" block, there is another condition check where the name of the hit object is compared.

In the Play mode, if the player goes in front of the cube (front side is where the Z axis arrow is pointing), the debug message detecting the player is written to Console in every frame. If the player gets further than 10 meters, messages will stop being written. And finally, if some other object gets in front of the cube within 10 meters, another debug messages appear in the Console.

3.2.6 Tags

When multiple types of objects with similar characteristics interact with an object, it would be difficult and slow for scripts to check, if the current object has a required property. Much simpler solution is to add a "*Tag*" to objects having similar characteristics. To do that, select an object and in Inspector click on the "*Tag*" combobox. Most objects are "*Untagged*" by default. The combobox contains several predefined tags. The last value is "*Add Tag…*". This option shows "*Tags & Layers*". Create a new tag by writing its name into the "*Element 0*" textbox. Select an object again and show "*Tag*" combobox again. Newly created tag is there and the object can be tagged by it.

To use it in script, let's modify the "*RaycastExample.cs*" script. Change the second if statement to this:

```
if (hit.transform.tag == "Player")
```

Tag First Person Controller to "*Player*" and check it in the Play mode. If the player has the correct tag, the message about his detection is written to Console.

3.2.7 Layers

If for example a cube is in the layer "*Ignore*" and the light has unchecked this layer in the "*Culling Mask*", then the cube is not lit by this light. This example shows the possibility of using layers to ignore other layers. To create a new layer, select an object, in Inspector expand

the “*Layer*” combobox and select “*Add Layer...*”. Layers from 8 to 31 are “*User Layer*” and can be edited. Others are “*Builtin Layer*” that are firmly defined and can’t be changed.

Let’s practice again. Create a new layer named “*Ignore*”. Create a new cube and set its layer to “*Ignore*”. Create a light and uncheck “*Ignore*” layer of the “*Culling Mask*” property of the “*Light*” component. The cube becomes immediately unlit.

Objects can also ignore each other, when they are in the layer intersection. When you show the “*PhysicsManager*” window by choosing “*Edit/Project Settings/Physics*”, the “*Layer Collision Matrix*” is present there (Fig. 3.2.4). If the intersection of “*Default*” and “*Ignore*” layers is unchecked, those layers will no longer see each other. For example, a cube from the “*Default*” layer with rigidbody can fall through a cube from the “*Ignore*” layer even though both cubes have attached the “*Box Collider*” component with unchecked “*Is Trigger*” checkbox.

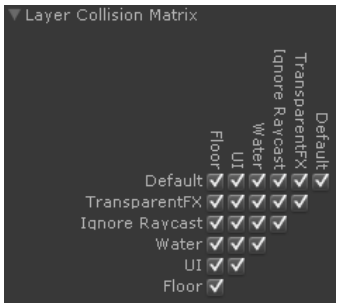

Fig. 3.2.4.: Layer Collision Matrix

3.3 Lesson 3 - Unity 2D

This lesson explains how to make a 2D game. Since version 4.3, Unity includes simplified 2D editor to make two dimensional games easier to develop. 2D games in Unity are in fact 3D games. That means, a 2D project can contain also 3D objects.

3.3.1 Sprite

Sprites are images, that are handled in a scene as objects with the "*Sprite Renderer*" component. That component has a "*Sprite*" variable which is the image. Every image in the Project can be a sprite. The sprite can be either one image or a set of images, also called the sprite sheet[7]. To make an image a sprite, select the image in the "*Project*" tab and in the "*Inspector*" set the "*Texture Type*" to "*Sprite (2D \ uGUI)*". If the image has the alpha channel, this channel becomes transparent. As mentioned, one image can contain multiple frames. Example of that can be the sprite sheet with "*cat ninja*[9]" (see Fig. 3.3.1). Each frame can be cut-out in the "*Sprite Editor*".

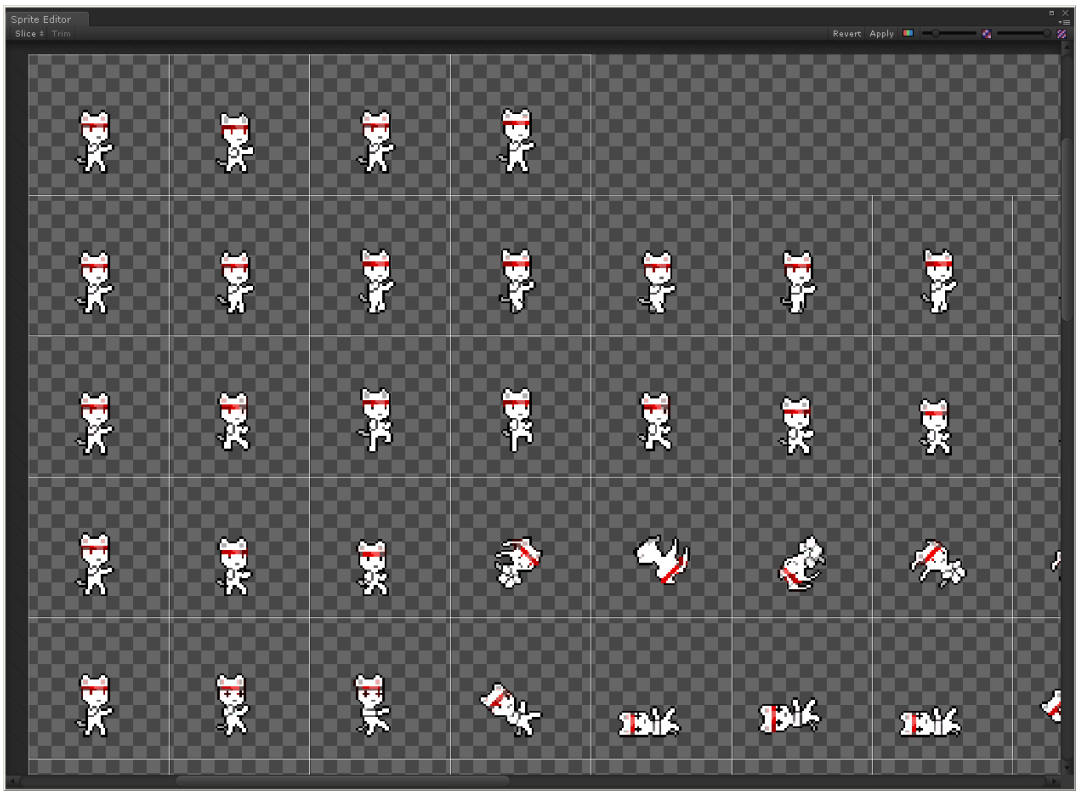

Fig. 3.3.1.: Sprite Editor

[7] Sprite containing all used frames.

Firstly, "*lesson_3-start*" package is required to be imported into the project. To import the package, select "*Assets/Import Package/Custom Package…*" and find the package file in the folder, where it is downloaded. The "*Importing package*" window appears (Fig. 3.3.2). Press the "*Import*" button to confirm the import of all checked assets. Those assets will move to the "*Assets*" folder and are visible from the tab "*Project*".

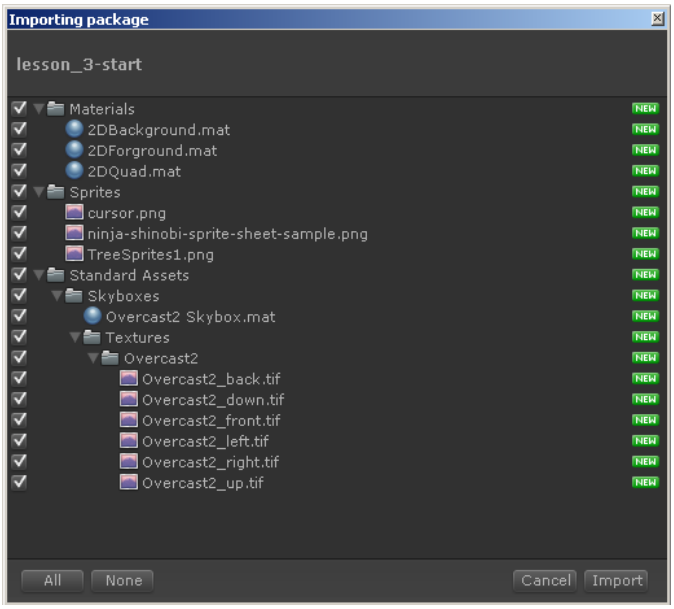

Fig. 3.3.2.: Importing package windowAnimated sprite

With the selected sprite sheet in the Project tab, change "*Sprite Mode*" to "*Multiple*" and then press the "*Sprite Editor*" button in the "*Inspector*". The "*Sprite Editor*" window opens (Fig. 3.3.1). In this window, the frames can be cut-out. After saving the changes, all frames are located uder the sprite in the "*Project*" tab. Any frame of the sprite sheet can be placed in the scene separately.

As mentioned above, all sprites in the scene possess the "Sprite Renderer" component . Next to the "*Sprite*", where the reference to the image from the "*Assets*" folder is stored, the "*Color*", which can add a tint to the sprite and the "*Material*", this component also handles *"Sorting Layers*" (don't confuse with "*Layers*" from Lesson 2). These layers help to sort sprites that overlap with others. The "*Sorting Layers*" layout can be arranged at will in "*Tags & Layers*" (Fig. 3.3.3). Each layer can also have its ordering within that layer thanks to the "Order in Layer" option. The layer's number defines, what is rendered first. The highest number assigned to a sprite in a set of overlapped sprites, that are in the same Sorting Layer, is

rendered. For example, the sprite of layer A ordered and number 10 is rendered after another sprite also in layer A, but with Order in Layer set to 9 (see Fig. 3.3.4).

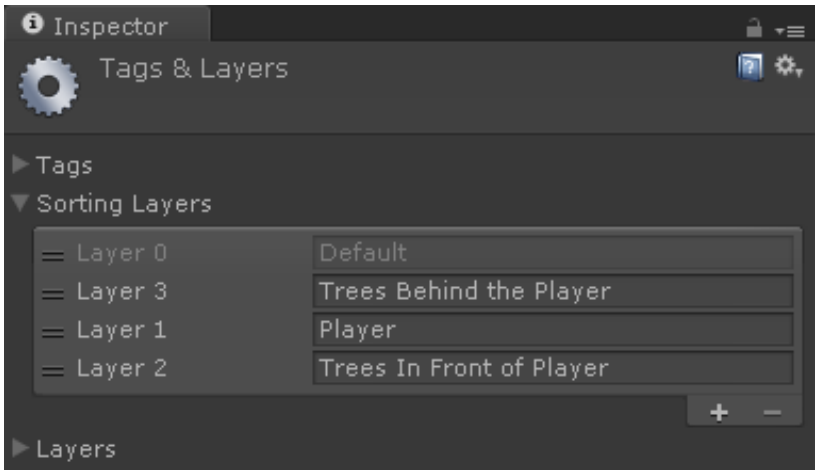

Fig. 3.3.3.: Sorting Layers

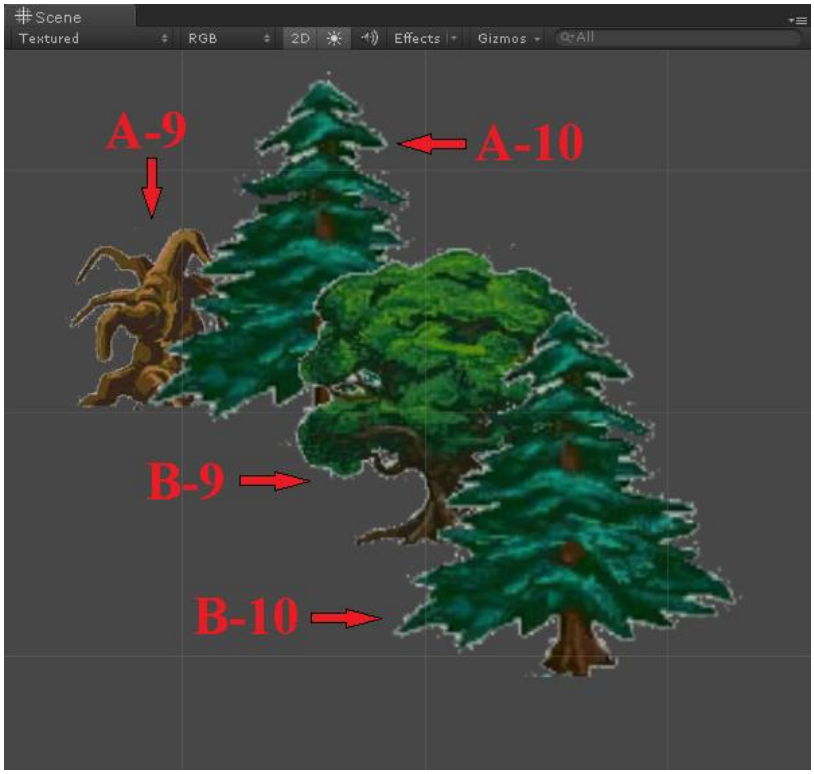

Fig. 3.3.4.: Images in sorting layers A and B, where the layer B is bellow the layer A in Tags & Layers

Usually, it is demanaded to work with the animated sprites instead of the static ones. The simple example can be a moving character. Working with animations will be explained more deeply in lesson 5.

To create the animation, select all sprites, that should be animated and drag & drop it into the *Scene or Hierarchy*. If the Play button is pressed, the sprites are animated. In order to change the order of sprites, add another animation or change the framerate of the animation, open the "*Animation*" tab by selecting "*Window/Animation*". Select the animated sprite object in the scene and the animation appears in the Animation tab (Fig. 3.3.5). The key frames are represented by squares on the timeline. Every change of sprite is reached by adding a key frame. The key frames can be rearanged. By adding another sprites from the assets, new key frames can be added. To create another animation, choose "*[Create New Clip]*" from the listbox of animations located under the red recording button in the top left corner. How to add transitions between multiple animations will be described in Lesson 5.

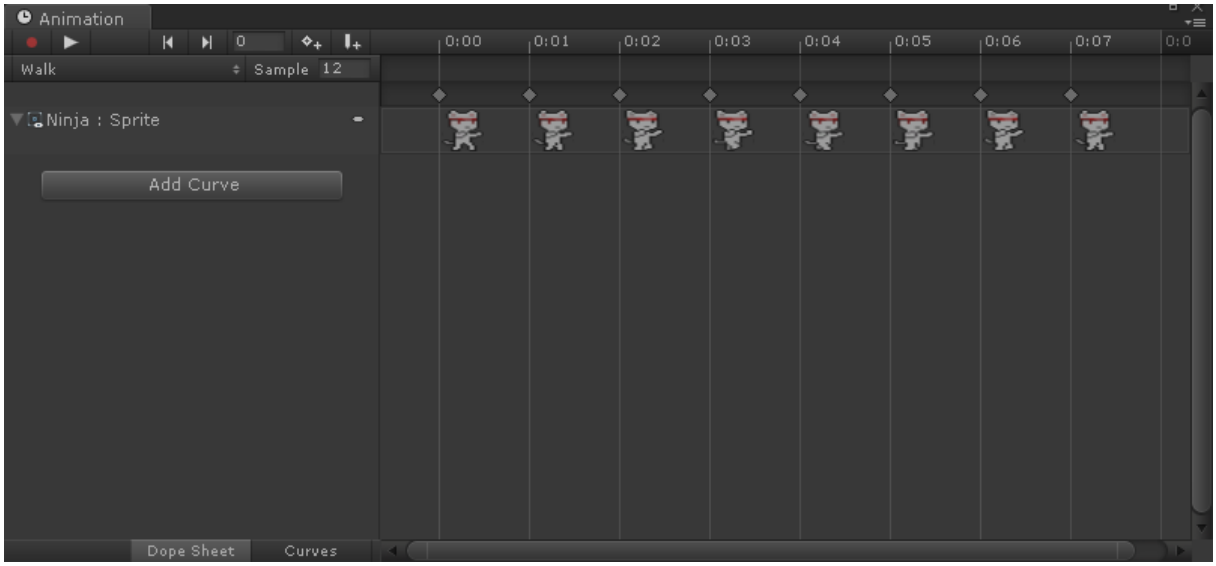

Fig. 3.3.5.: Animation tab with selected object for the animation

3.3.2 2D Physics

Just like in 3D objects, 2D objects can also affected by physics. 2D and 3D physics however can't be combined together because 2D physics doesn't use the Z axis. Besides this, everything works the same as with 3D physics. Thanks to the 2D colliders, objects can collide with each other in the X and Y axis. If the object has the 2D rigidbody, it can be affected by gravity.

To practise, add to the animated sprite the "*Box Collider 2D*" and the "*Rigidbody 2D*" by adding a new component. Add the "*Quad*" to the scene from the "*GameObject/Create Other/Quad*" (note that it has the "*Mesh Collider*", which is 3D collider), move and scale it under the sprite. In the Play mode, the sprite falls through the Quad. Quad's collider has to be changed from 3D

collider to 2D collider. To do that, the current collider has to be removed first. Then, the 2D collider can be added. Now the sprite falls on the Quad and stays on it (Fig. 3.3.6).

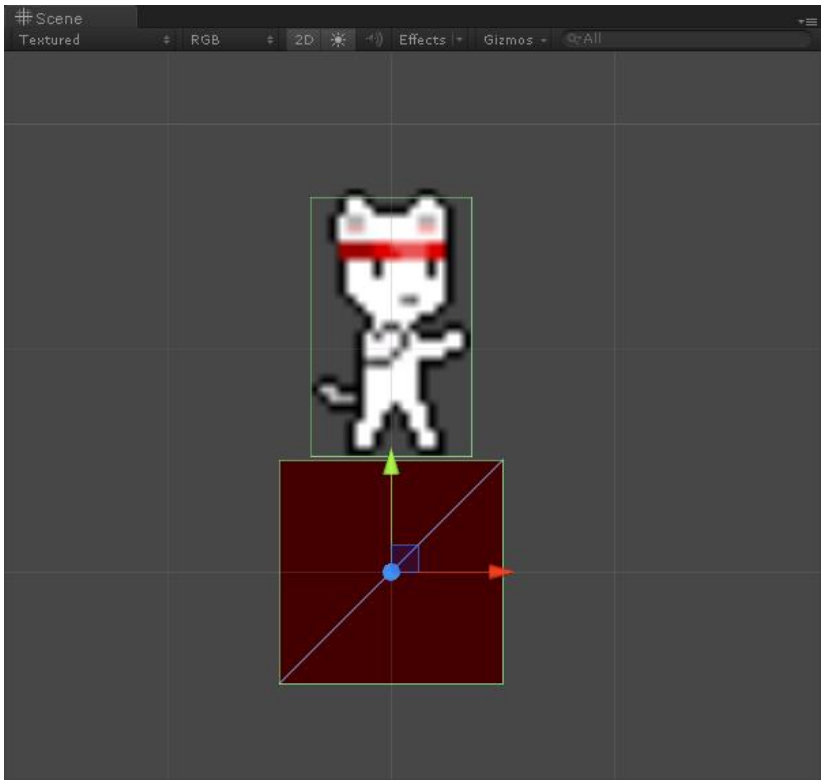

Fig. 3.3.6.: Objects with 2D colliders

3.4 Lesson 5 - Animations

Animations of objects can be imported together with the model or separately. They can also be created directly in Unity. Note, that Unity doesn't support vertex animations, so they need to be created using skeletons or simple object transformations.Similarly, the deformation tools widely used in 3D editors are not supported in Unity. This lessons shows how to create animations in Unity using three different techniques. Those techniques are the script animation, the keyframe animation and imported animation from 3D editors.

3.4.1 Script animation

Simple animations, e.g. rotating a cube, can be performed using the script. This approach is however very limited. More complicated animations are difficult or even impossible to make just by scripting. Smooth transitions between animations could prolongate the script enormously as well. However, if the object is planned to possess a simple animation, such as the already mentioned rotation (e.g., a pickable box with the ammo), script animation is the fastest way to complete it.

Lets describe a simple example of such animation using scripting.

Create a cube. Create and attach a script to this cube. In the "*Update*" function of this script, write the following line:

```
transform.Rotate (Vector3.up * Time.deltaTime * 100);
```

It gets to the "*Transform"* component and calls the "*Rotate*" function. The parameter of this function defines the direction of the rotation using "*Vector3.up*[8]". It is then multiplied by "*Time.deltaTime*", which defines the time since the last update. This is used because some platforms are able to call more updates at the same time than others. By using this multiplication, we are able to minimize the difference in speed of update calls between different platforms.

3.4.2 Keyframe animation

Let's make the rotation animation by using a different approach. In lesson 3, tab "*Animation*" was introduced to view frames of an animated sprite. Now we create the animation using the following procedure. Add another cube to the scene and select it. View the "*Animation*" tab

[8] "*Vector3.up*" is the same as Vector3(0, 1, 0).

and create new animation clip. Select a folder to save the animation file. Press "*Add Curve*" button and pick "*Transform/Rotation*". To make the object rotate along the "*Y*" axis, add the keyframe at the time 0:00 with object's rotation value of Y set to 0 by pressing "*Add Keyframe.*" button. Move the red line in the timeline to the time 1:00, change the object's rotation value of Y to 360 degree. Another keyframe is added automatically. Preview the animation by pressing the "*Play*" button located in the top left corner of the "*Animation*" tab. The cube rotates just along the "*Y*" axis.

Animations can be also used to trigger an animation event. The animation event is an action within the animation that calls the selected user-defined function from scripts, that are assigned to that object. If the animation gets to the frame, where the event is located, the selected function is called. To try it, create a script containing the following function:

```
    private int counter = 0;

    void MyAnimationEvent(string parameter)

    {

        counter++;

        Debug.Log ("Animation event happened " + counter + " times
with parameter '" + parameter + "'.");

    }
```

The function takes the parameter of type "*string*", increments the counter and writes a message to the debug console. Now go back to the "*Animation*" tab and in the animation, move the red cursor in timeline to the time, when the event is suppose to happen (e.g., in time 0:30). Press "*Add Event.*" button to add the event. The "*Edit Animation Event*" window appears (Fig. 3.4.1). From the "*Functions:*" combobox, pick the *"MyAnimationEvent*" function. If the function has any parameters, input lines appear. Fill in those parameters and close this window. Note that the animation events are functional only in the "*Play*" mode. Start the game and check the debug console to see a message from "*MyAnimationEvent*". Every time the animation gets to the event, the message in the debug console is written.

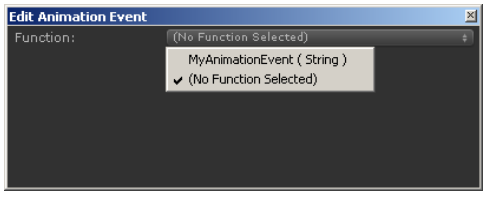

Fig. 3.4.1.: Edit Animation Event window

3.4.3 Import rigged object and animations

The package "*lesson_5-start.unitypackage*", folder "*FBX*[9]" contains a 3D model of a character with rig created using a free software tool called "*MakeHuman*[10]". Select this model in the "*Project*" tab to view import settings in "*Inspector*" (Fig. 3.4.2). In the tab "*Rig*" select the "*Animation Type*" to "*Humanoid*". This animation type is used to models with bone structure similar to human skeleton. Other available options are "*Generic*", which is usable for non-human models with skeleton (e.g,. flower) and "*Legacy*" to support backward compatibility with projects created in older versions of Unity, before Mecanim tool was introduced.

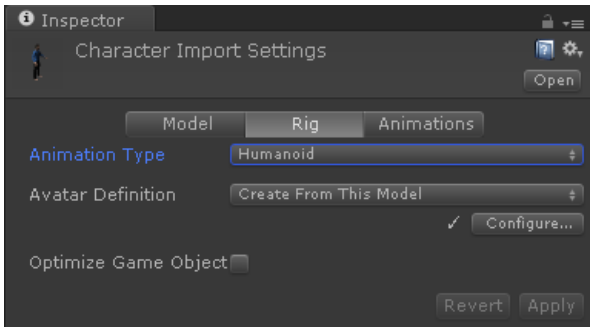

Fig. 3.4.2.: Character Import Settings

Tick next to the button "*Configure*" indicates the correctly mapped bones. To modify the bone mapping, save the scene and press the "*Configure*" button. New scene with the character opens (Fig. 3.4.3). In the "*Inspector*" the "*CharacterAvatar*" is visible. To attach the required bone to the avatar, drag and drop that particular game object from the "*Hierarchy*" to the avatar's bone. When finished with changes, press the "*Done*" button to get back to the previous scene. Add the character into the scene.

[9] FBX is a file format created by Autodesk to provide compatibility between 3D software.

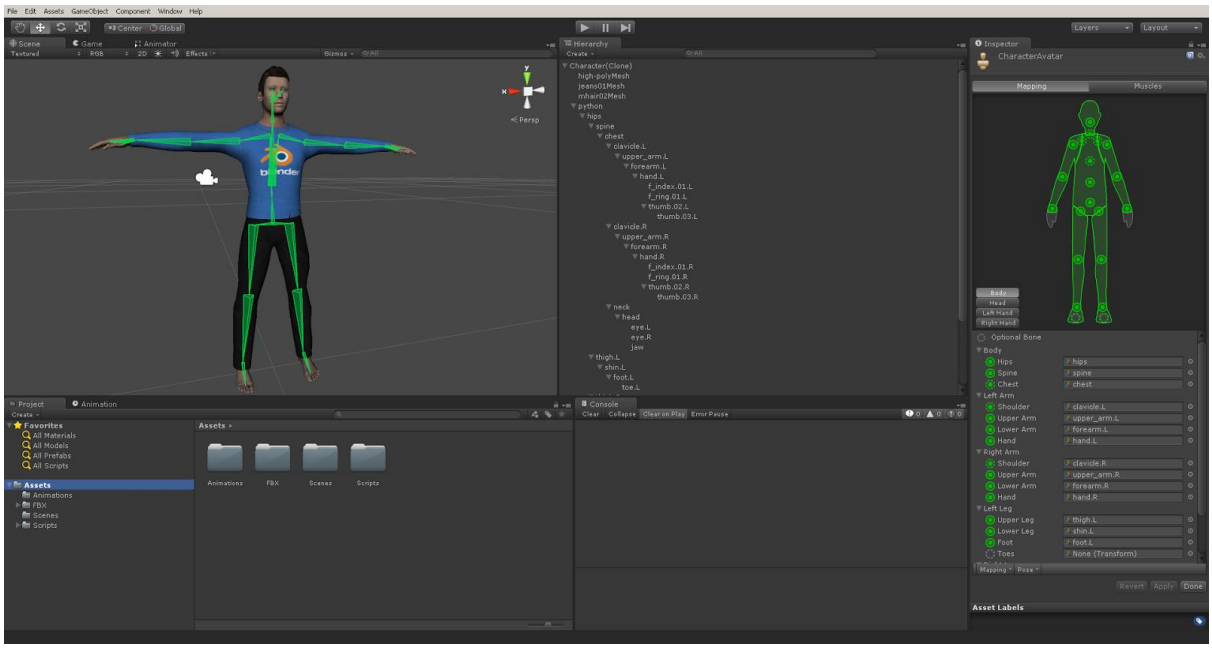

Fig. 3.4.3.: Character Avatar settings

Along with the character model, the "*Assets/FBX*" folder contains also motion captured animations downloaded from "*motcap.com*[11]" and converted to FBX using "*Blender*[12]".

3.4.4 Mecanim

Unity's "*Mecanim*" is a state machine system for animations. Each animation represents a state. States can be connected with the "*Transition*". Transitions can have conditions. If all conditions are evaluated as "*true*" and the state, from which the transition goes is currently active, it will move through that transition into the target state.

Select the character model in the "*Inspector*" and attach the "*Animator*" component. Then create an "*Animator Controller*" in the "*Project*" tab. Drag and drop the animator controller into the "*Controller*" of the character's "*Animator*" component. Open the "*Animator*" tab (Fig. 3.4.4) by choosing "*Window/Animator*" with the character selected in "*Hierarchy*".

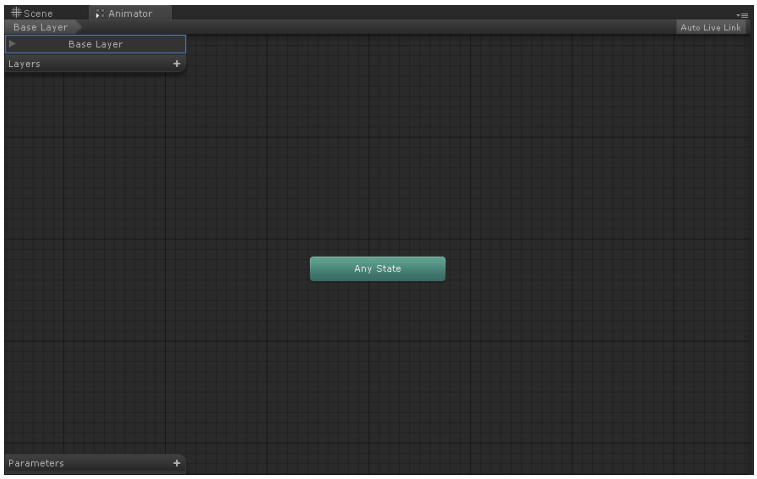

Fig. 3.4.4.: Animator tab

In the Fig. 3.4.4, the animator controller has only one state called "*Any State*". It makes possible to translate to different state from all states if given condition is met. To add new states, select animations from the "*WalkingAnimation.fbx*" and drag and drop them into the space within the "*Animator*" tab. Add both animations from the asset. The states are created from these animations. In the Fig. 3.4.5, one of the states is marked using the orange color.

Fig. 3.4.5.: Animator states with the default state (orange)

This is the default state from which the state machine starts. Any gray state can be changed to the default state by pressing the right mouse button on the state and selecting the "*Set As Default*" option. To connect the states, press the right mouse button on the state, from which the transition goes. Select the "*Make Transition*" option. Click on the other state to make a connection. Make another transition that goes in the other state back to the default state as seen in Fig. 3.4.6.

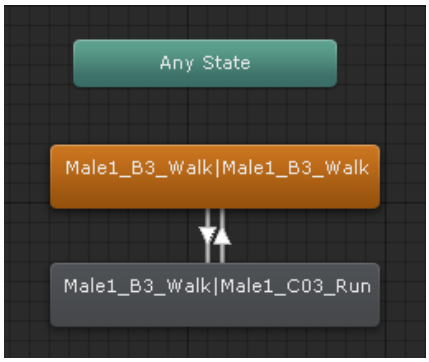

Fig. 3.4.6.: Animation transitions

Transitions can have conditions. Only if all conditions are met, the transition will undergo. To use the conditions, firstly we have to add parameters. In the bottom left corner of the "*Animator*" tab, press the "+" button and select the type of the parameter (Fig. 3.4.7). For this example, type "*bool*" is sufficient. Name the parameter to "*Run*".

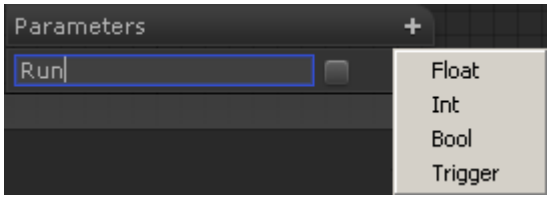

Fig. 3.4.7.: Parameters

Now select the transition arrow leading from the default state in the state machine to show its settings in the "*Inspector*" tab. There is already the default condition called "*Exit Time*". Values of this condition are between 0 and 1 and it indicates the time of the animation, when 1 is the animation's end. When the input animation gets to the value of "*Exit Time*", the transition will be performed. Change the "*Exit Time*" condition to the "*Run*" condition with required value set to true (Fig. 3.4.8). By pressing the "+" button, more conditions can be added. Select the second transition and also change the condition to "*Run*", however this time with required value of false.

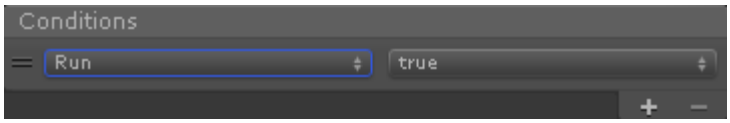

Fig. 3.4.8.: Conditions

Values of parameters in the "*Animator*" component are changed from script. Create a script and attach it to the character. In the script, write the following code:

```
void Update()
{
     if(Input.GetMouseButtonDown(0))
          GetComponent<Animator>().SetBool("Run", true);


     if(Input.GetMouseButtonDown(1))
          GetComponent<Animator>().SetBool("Run", false);
}
```

Now in the Play mode, if the user will press the left mouse button, the script gets to the "*Animator*" component and sets the parameter of the type bool named "*Run*" to true and the character starts to run. If the user will press the right mouse button, the script gets to the "*Animator*" component and sets the parameter of the type bool named "*Run*" to false and the character stops running and starts walking.

3.5 Lesson 6

This lesson explains, how to effectively work with global variables, even across multiple scenes.It also shows different possibilities of showing text and finally there will be shown, how to make sound effects and the background music.

3.5.1 Level manager

In order to remember the variables, that are assigned to objects in scene, it is advisable to store them at a unified place. For this purpose, the level manager is usually used. It is an empty game object that is present the whole time in the current scene. Each scene can have its own level manager. Thanks to the level manager, objects can access and even update global variables, like score, health, and other statistics. It can be also used to handle level's game logic. Imagine the point and click adventure game for example, where the player searches for a hidden door. The player needs to pull the lever to reveal that hidden door. The level manager has a stored information, whether the door is already revealed or not.

3.5.2 Game manager

The game manager works similar as the level manager, but it is available through the whole game. The game manager keeps global informations about the game like for example statististics or variables that will be saved into the file. Those informations are transportable across all scenes. To keep an object across multiple scenes, the following function needs to be called:

```
DontDestroyOnLoad(transform.gameObject);
```

This functionality can brings problems. If the game manager is created in scene A, then the player goes to scene B, all is fine. But when the player then goes back to scene A, where new game manager is created, there are suddenly two game managers in the game. There are several solutions to this problem.

First and the easiest solution is to have a special scene, that is entered before all others and where the game manager is created. After the scene is abandoned, the player can never get back there.

Another solution could be to destroy the existing game manager, when some other is found. The simple check can look like this:

```
void Start()
```

```
{
    if(GameObject.Find("GameManager") != this.gameObject)
    Destroy(this.gameObject);
}
```

The advanced solution would be to use the Singleton. There are tutorials on the internet describing, how to use Singleton in Unity. However, this is an advanced topic overcomming the level of this lesson.

3.5.3 Texts as GUI

3.5.3.1 Unity GUI

Simple user interface, like buttons or texts can be made with the Unity GUI. This approach is good for debugging purposes. Its ineffective, because it's called every frame, just like the Update function. Also it's not very customizable, but since it's definition is quite simple, it is still used in games in development. The Unity GUI is defined isnide the OnGUI fucntion. Button can be created as follows:

```
void OnGUI ()
{
    if(GUI.Button(new Rect(10, 20, 100, 30), "Button"))
    {
        //This block is accessed, when the button is pressed.
    }
}
```

The button is located in 10 pixels from the left border, 20 pixels from top border, its size is 100 by 30 pixels and the text "*Button*" is written on it. If the button is pressed, code located inside the "*if*" block is executed.

3.5.3.2 GUI Text

The advanced approach to the user interface is "*GUI Text*". Advanced because of its "*GUIText*" component. Thanks to it, the text is clickable similarly as an object with attached collider. It is more efficient, because unlike the Unity GUI, it isn't drawn from a function. Also, if we want to have a code, that is called, when the user clicks on the text, we can use the event function named "*OnMouseDown*". To create a GUI Text, select "*GameObject/Create Other/GUI Text*".

The position of the text is adjustable in the "*GUIText*" component via the "*Pixel Offset*" or the position in the "*Transform*" component. Through the "*GUIText*" component can also be easily adjusted the size, color, line spacing, alignment and others (Fig. 3.5.1). Feel free to experiment with it.

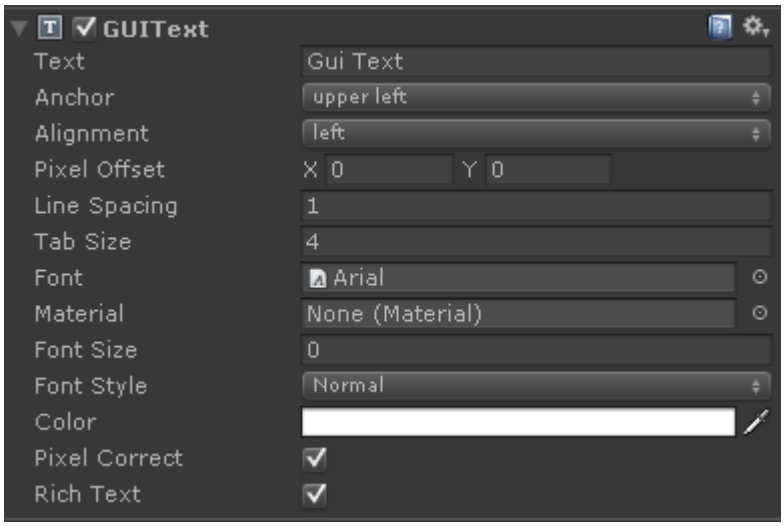

Fig. 3.5.1.: GUIText component

3.5.3.3 3D text

GUI can also be a part of a game environment. Some modern games use this kind of approach to achieve deeper immersive feel. Unfortunetely, this can't be done neither with the Unity GUI, nor the GUI Text. Fortunetely, The Unity game engine offers third type of text, and that is the "*3D Text*". To add the 3D Text into the scene, select "*GameObject/Create Other/3D Text*". Unlike previous variations, 3D text can be transformed in 3D environment. It can be rotated, moved and scaled just like any 3D object. It has the "*Text Mesh*" component assigned. It has similar properties as the "*GUIText*" component showed in the previous example (Fig. 3.5.2). Problem with 3D text is its font resolution. If the mesh is scaled up in the "*Transform*" component, the text becomes more blurry (Fig. 3.5.3). To solve this problem, change the "*Font Size*" of the "*Text Mesh*" component to higher value and then scale down the mesh. The higher the value of "*Font Size*", the sharper the text is.

3D Text uses the "*Text Shader*". This shader makes the text visible above the other objects in the scene. That means, even if the 3D text is behind a box, the text is still visible. If this is not desired, custom shader needs to be created. Shader like this can be found on the internet.

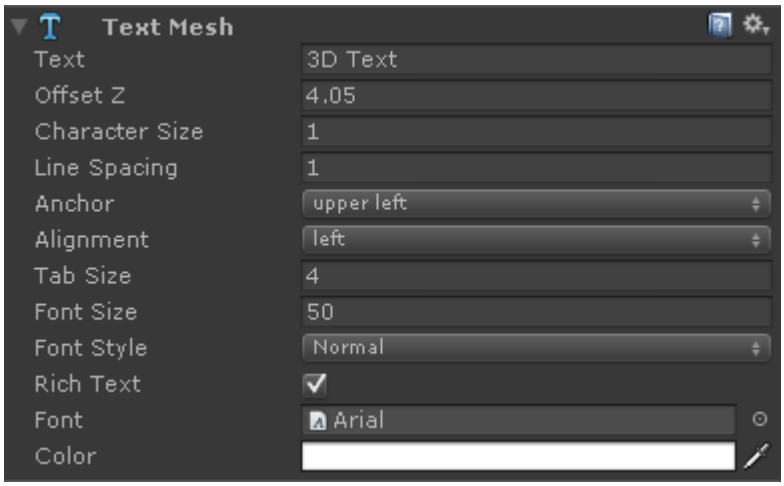

Fig. 3.5.2.: Text Mesh component

Fig. 3.5.3.: Blurry 3D text

3.5.4 Resources

To reserve the memory, Unity supports "*Resources*". It is a special folder within the "*Assets*" folder, from which can be loaded any object in runtime. Thanks to that, other objects don't need to remember the object's referrence the whole time, they just load the object directly. Let's see an example. First, create the folder named "*Resources*". The folder can be located in any subfolder of the "*Assets*" folder and it can contain its own subfolders. Let's have a "*Resources/Audio"* folder containing sprites. Following line takes an image from that folder.

```
    AudioClip soundFX = Resources.Load ("Audio/audioName") as
AudioClip;
```

The static function "*Load*" is called from the class "*Resources*". This function takes a string as parameter, where the path is defined. It returns simple "*object*", so it needs to be convert to required data type. Then it can be assigned to the variable.

3.5.5 Sounds and background music

Audio is an important part of the game. Happy music can add the peaceful feeling, action music on the other hand tells the player to be at attention. Sound effects can reveal player's surroundings and voiceovers usually tell the story. Game without audio loses a crucial immersive part. Here's how to make sounds in Unity.

First of all, there must be an "*Audio Listener*" in the scene. This component usualy have "*Main Camera*" assigned, since it is also the camera of the player. Thanks to "*Audio Listener*", the player can hear the audio. Then, any object, that makes a sound requires an "*Audio Source*" component. In the package "*lesson_6-start*" in the "*Assets/Sounds*" folder is a music audio clip[13]. Select the clip to reveal its settings in Inspector. Since it will be the background music, there is no need to have this as 3D sound. Uncheck the checkbox "*3D sound*". Go back to the scene and select the object with "*Audio Source*" component. In that component drag and drop the music into the "*Audio Clip*" property. Also make sure, that "*Play On Awake*" checkbox is checked so the sound starts with the game. In the Play mode the music plays from the start of the game.

To play a sound effect on the same object while the music still plays, we have to make a script to play only one shot of the sound. On the object with "*Audio Source*", create a new script. Let's have the sound effect played when the player clicks on the object. For this purpose, function "*OnMouseDown*" is requiring. To be able to use this function, make sure that the object has a collider. The body of the function can look as following.

```
AudioClip soundFX = Resources.Load ("pickItem") as AudioClip;
audio.PlayOneShot (soundFX);
```

The audio clip gets loaded from the Resources folder and then it's played once using the "*PlayOneShot*" function of the audio component.

3.6 Lesson 7 - Artificial intelligence

Enemies in games should have basic thinking. They should be able to decide, if the enemy is in sight, so he can attack him, where he can walk and which way to take in order to not walk through the door. In this lesson, example of simple finite state machine will be shown. The enemy will be able to change his internal state, so when the player gets close to him, the enemy starts chasing him. Also, thanks to Unity's "*NavMesh*" features, the enemy will dodge the walls and static obstacles. This lesson was inspired by the second chapter of the book "*Unity 4.x Game AI Programming*[14]".

3.6.1 Simple Finite State Machine (FSM)

The enemy in this example will have three states. Logic of these states are following (Fig. 3.6.1). Initial state is "*Patrolling*". In this state, he walks between waypoints and doesn't know about the player. From this state can go to "*Chasing*" state, where he tries to get in range to the player in order to be able to attack. If the player gets too far from him, he goes back to the "*Patrolling*" state. If he gets close however, he moves to the "*Attacking*" state and starts shooting. If the player gets out of range, the enemy goes back to the "*Chasing*" state.

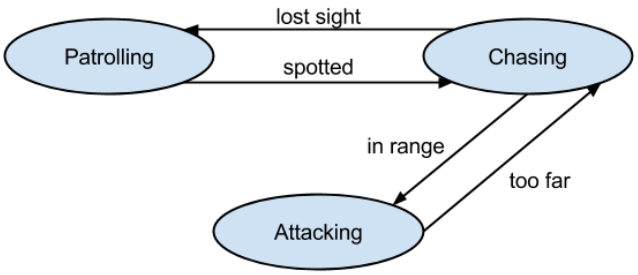

Fig. 3.6.1.: Enemy AI FSM

States in script can be represented as an enumaration. Let's create a script for simple FSM and open it in editor. Next to the class, define an enumaration of states.

```
public enum States
{
    Patrolling,
    Chasing,
```

```
        Attacking
    }
```

Now use this enumeration as variable in the FSM class.

```
    public States state = State.Patrolling;
```

Logic of each state will be handled with individual function. There will be called only one function each Update according to current state using switch statement.

```
    void Update()
    {
        switch (state)
        {
            case FSMState.Patrolling: UpdatePatrolState(); break;
            case FSMState.Chasing: UpdateChaseState(); break;
            case FSMState.Attacking: UpdateAttackState(); break;
        }
    }
```

In each state update function is checked the distance from the player. According to the player's distance, state change is performed, when he is close. The patrol update function can look as follows.

```
    void UpdatePatrolState()
    {
        //If waypoint is reached, find another one
        if(Vector3.Distance(this.transform.position,
targetWaypoint.transform.position) < 2)
        {
            print("Waypoint reached, finding another one.");
            FindNextWaypoint();
        }
        //If player is near, change state to chase
        if(Vector3.Distance(this.transform.position,
player.transform.position) <= 12)
        {
```

```
            print ("It't the enemy! I need help!");
            this.renderer.material.color = Color.yellow;
            state = FSMState.Chase;
        }
        currentTarget = targetWaypoint.transform.position;
        MoveToTarget ();
    }
```

If the player is not in sight, the enemy moves between waypoints. Waypoint is a simple game object with tag "*Waypoint*" and the enemy remembers all the waypoints he uses in the "*List*". Moving of the enemy is handled simply by "*LookAt*" and "*Translate*" functions.

```
    void MoveToTarget()
    {
        this.transform.LookAt (new Vector3(currentTarget.x,
this.transform.position.y, currentTarget.z));
        this.transform.Translate (Vector3.forward * Time.deltaTime *
speed);
    }
```

For complete code, see package "*lesson-7*", script "*EnemyAI.cs*".

3.6.2 Pathfinding with Navigation mesh

Fucntion "*Translate*" from above only moves with the object in the direction specified in the argument of the function. Unfortunetely, when an other object is in the way, the object with the "*Translate*" function goes through it. That is usually not the desired behaviour. If we want the enemy to avoid obstacles, the "*NavMesh*" can serve this purpose. Navigation meshes are special kind of meshes created from navigation static game objects usable for pathfinding. Unity uses "*A**" pathfinding algorithm to calculate the shortest way to a target.

To be able to generate the navigation mesh, objects from which will be the navigation mesh generated must be marked as "*Navigation Static*". Select objects making the level environment, such as floor, walls, obstacles and others. Open the "*Navigation*" tab (Fig. 3.6.2) by selecting "*Window/Navigation*". In the "*Navigation*" tab check the checkbox "*Navigation Static*". All selected objects are now static for navigation and ready for baking a navigation mesh. Press the "*Bake*" button. On the navigation static objects has been created the navigation mesh indicated by blue polygons in the "*Scene*" view (Fig. 3.6.3).

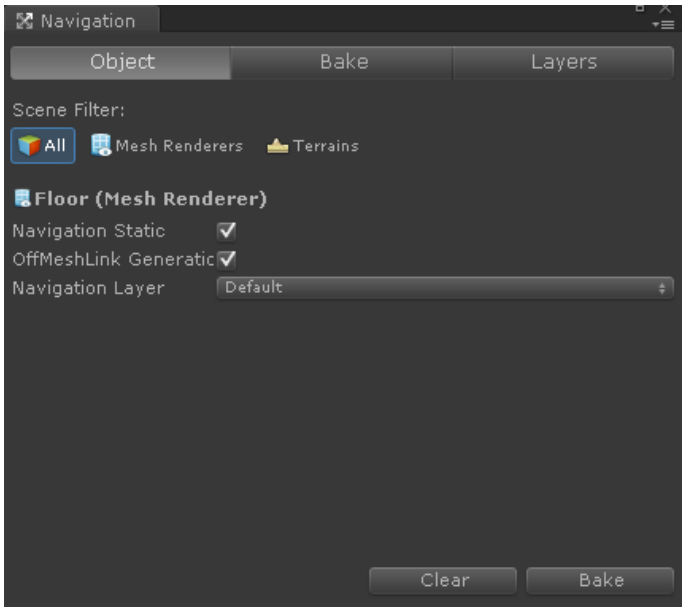

Fig. 3.6.2.: Navigation tab

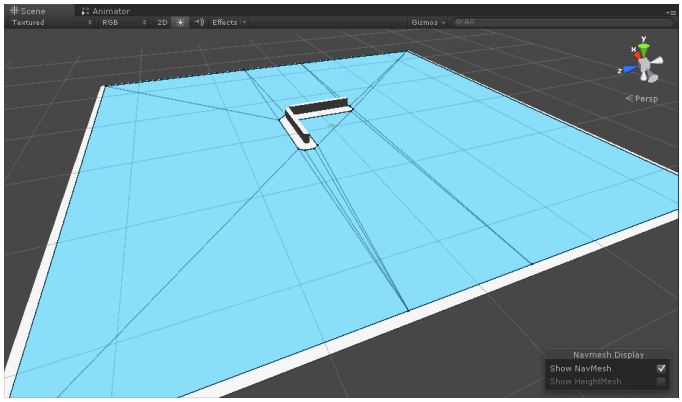

Fig. 3.6.3.: Generated navigation mesh

Object that use the "*NavMesh*" require the component named "*Nav Mesh Agent*" (Fig. 3.6.4). Parameters, such as speed of the object, can be set there. To use navigation mesh, set the "*destination*" variable of the "*NavMeshAgent*" component using the "*Vector3*" coordinates.

```
this.GetComponent<NavMeshAgent> ().destination = target;
```

Whenever the destination is set, the "*NavMeshAgent*" calculates the fastest way to that destination and the object goes the by the speed defined in the "*NavMeshAgent*" component.

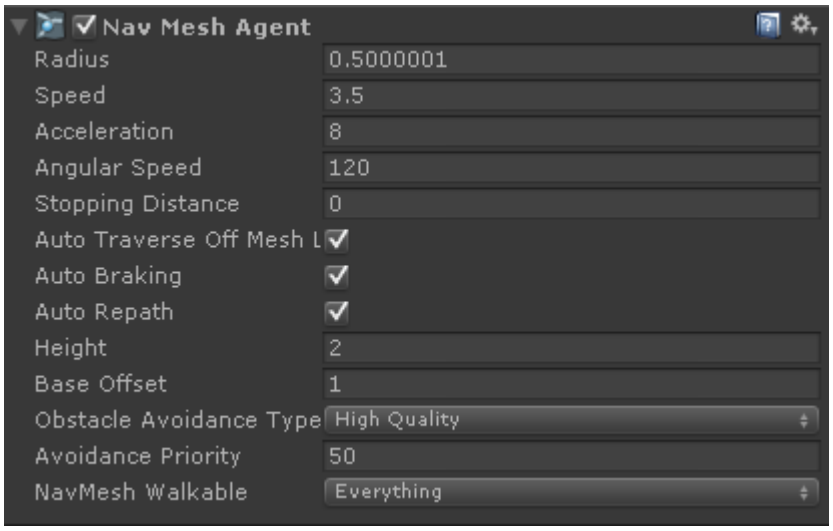

Fig. 3.6.4.: Nav Mesh Agent component

3.7 Lesson 8 - Building, debugging and releasing

Mono Develop offers basic tool for debugging. Thanks to it, the developer can make breakpoints and check values of variables quite easily. When certain feature is finished and functional in editor, it's good to test it on a built version as well. Each platform can have its own specific limitations, so a script, that executes differnt code on specified platform can be really helpful.

3.7.1 Build Settings

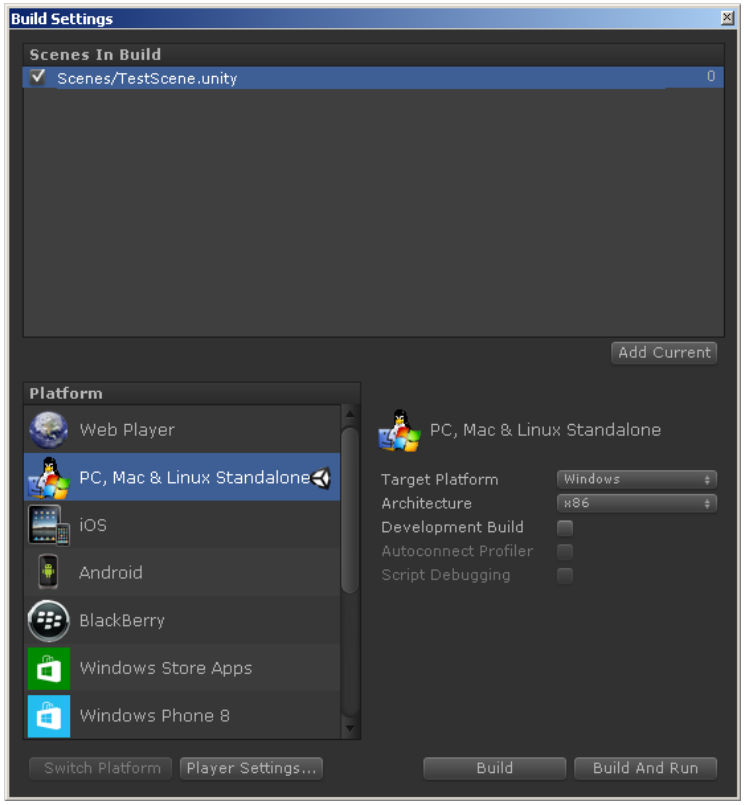

Fig. 3.7.1.: Build Settings

From "*Build Settings*" the build itself can be make. Before bulding, add all scenes that you want to have in build. Simply drag & drop all scene files from the "*Assets*" folder into the "*Scenes In Build*" list. If the scene is not in the list and the script tries to load it using function "*Application.LoadLevel()*", the game level will not be loaded. The first scene in list with ID number "*0*" is going to be loaded first when the game launches. Usually this scene serves as splash screen showing logos and to instantiate the game manager mentioned in lesson 6 or main menu. In the "*Platform*" section output platform can be selected. In default, "*PC, Mac & Linux Standalone*" is selected. After picking the right platform, press "*Build*" or "*Build And Run*" button to build the game. The build contains two parts. An "**.exe*" file from which

the game is launched and a "*_Data*" folder containig most [10]of the used assets. The name of the executable file and the data folder must be the same. For example the "*testGame.exe*" must be in the same folder with the folder named "*testGame_Data*" in order to be functional.

3.7.2 Building on different platforms

Switching between platforms is very easy in Unity. Just select the wanted platform and press the "*Switch Platform*" button. The editor optimizes assets and switches to that platform. Some platforms require additional modules. For instance in "*Xbox One*" case, the official development kit hardware from Microsoft is required. These development kits are provided for free to approved developers for free.

3.7.3 Player Settings

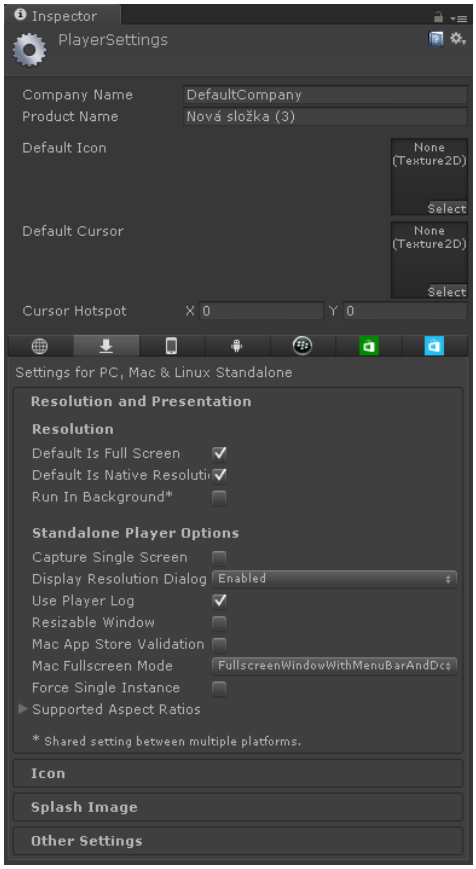

Fig. 3.7.2.: Player settings

Press the "*Player Settings…*" button in "*Build Settings*" to open "*PlayerSettings*" or select "*Edit/Project Settings/Player*". Here can be set things like "*Company Name*", "*Product Name*",

[10] Some assets like textures or save data can be loaded from other sources during runtime.

icon of the executable file, resolution settings and other. Some values like allowed resolutions and aspect ratios can be different for each platform.

3.7.4 Platform dependent script

Some functionality can work only on certain platform. An example could be saving a file. If done properly it can work on most platforms, but even then not all platforms, because of the "*Web Player*". Web player doesn't support work with file system so the script wouldn't work and the game could crash. To solve this issue, preprocessor directives checking conditions can be used. Here's an example of a function that works different on platfom other than web player.

```
#if !UNITY_WEBPLAYER
    void MyFunction()
    {
        //body of the function
        Debug.Log (Application.platform);
    }
#else
    void MyFunction()
    {
        Debug.Log("Not supported on this platform.");
    }
#endif
```

To test the script for different platform, open "*Build Settings*", select required platform and press "*Switch Platform*" button. Switch to the "*Web Player*" platform. Back in the script, the code in "*#if*" is gray and the code in "*#else*" is in normal color (Fig. 3.7.3). This indicates currently selected platform in editor, where gray indicates unused code. The compiler ignores gray lines of codes same as the commented code.

```
26 #if !UNITY_WEBPLAYER
27     void MyFunction()
28     {
29         //body of the function
30         Debug.Log (Application.platform);
31     }
32 #else
33     void MyFunction()
34     {
35         Debug.Log("Not supported on this platform.");
36     }
37 #endif
```

Fig. 3.7.3.: Platform dependent script with platform set to "Web Player"

3.7.5 Breakpoints

To use breakpoints in code, first there must be attached Mono Develop to the Unity. Open the Mono Develop, add breakpoints into the script and save it. Now attach the Mono Develop to the Unity process by selecting "*Run/Attach to Process...*". In the newly opened window (Fig. 3.7.4) select the process "*Unity Editor (Unity)*" and press the "*Attach*" button.

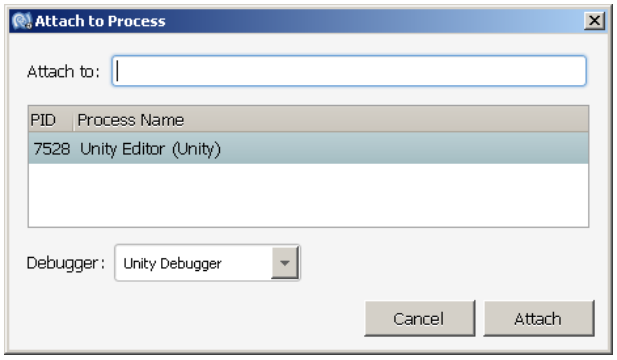

Fig. 3.7.4.: "Attach to Process" window

Now in the Unity editor in the Play mode whenever the line with breakpoint is executed, the Unity stops and freezes. Switch to the Mono Develop to the script with breakpoint. The breakpoint that stopped the editor is marked with yellow point. In the tab "*Locals*" can be seen current values of variables. To continue, press the "*Continue Execution*" button in the header of the Mono Develop window (Fig. 3.7.5). Unity unfreezes until the next breakpoint. Don't forget to detach the process when finished with debugging. Select "*Run/Detach*" and confirm by pressing the "*Detach*" button.

Fig. 3.7.5.: Debugging tools in Mono Develop

3.8 Lesson 11 - Object states / save and load

In an AI script in lesson 7 were used states to switch between different update functions. In this lesson, the use of states will be extended. States are used to know about object's current status. An example could be door. Door can be in state "*closed without key in inventory*", "*closed with key in inventory*", "*opened*" or "*destroyed*". Similar logic is used in the example project, but instead of door, the treasure is used. The treasure is locked. When the player pickes a key, he can open the treaser. It hides an item, which is added into the inventory, when the player clicks on it. Then the treasure closes. Also, the player can save his progress, which can be loaded the next time he plays. This lesson was inspired by chapter 9 of the book " *Beginning 3D Game Development with Unity 4, 2nd Edition*[15]".

3.8.1 Simple state

First of all, import the "*lesson_11-start*" package and open the ObjectStates scene. Then create a script named "*ObjectStates*". Open the script and before the "*ObjectStates*" class create a class "*State*". The "*State*" class contains informations stored in variables about the state. Let's use following variables for the state.

```
    public int stateId;    //identification number of this state

    public AudioClip transitionSound; //sound effect when the object
goes into this state

    public string requiredItem; //required item to be able to leave
this state

    public bool interactive;    //can the player interact with the
object in this state?

    public int nextStateId;     //identification number of the next
state
```

Also mark the class as serializable, because custom classes are not visible in the Inspector tab in the editor. Unity uses its own form of serialization. Add the serialization class attribute before the class.

```
    [System.Serializable]
```

In the "*ObjectStates*" class create a list[11] of "*State*". In this simple example, the "*stateId*" variable corresponds with the order in the list. The finished example shows more complex example, where the order is not required.

```
public List<State> states;
```

Add the "*ObjectStates*" script to the treasure object and show the Inspector tab. Set the "*Size*" of "*States*" to "*4*" and fill in each state according to the "*TreasureStates.txt*" file. Use the same order as in the text file. When finished, go back to the script and and add a variable for the current state. For the simplicity of this excercise, use the "*int*" data type. In the finished example is shown advanced variant using the whole "*State*". Also add a variable of the type "*GameObject*" to store the referrence to the player's game object. In the treasure's inspector set the current state to "*1*", since this will be the original state. As for the player variable, move the player's game object to make the referrence.

3.8.2 Inventory

To be able to continue, let's create a script for the inventory. Again, this will be a simple example with the list of strings as the inventory. Later on the state script will check, if the required item is in the inventory, when an item is required to use the object. Also, the inventory will be used as the data for saving the game. Assign the inventory scrip to the player.

Then Create a script for pickable items. This script simply adds its object name to the list in the inventory script, when it's clicked on the object. Then it destroys itself because it's no longer needed in the scene.

```
public GameObject player;
void OnMouseDown()
{
    player.GetComponent<Inventory> ().inventory.Add (this.name);
    Destroy(this.gameObject);
}
```

[11] In order to be able to use generic collections, add the line "*using System.Collections.Generic;*" in the header of the script.

Add the script to the key object and don't forget to assign the player's game object in the Inspector.

3.8.3 State transition

When the player clicks on the treasure, it should check if it can go into the next state according to the required item and act accordingly. To make the example more interesting, treasure animations are provided with the package. Create the animator controller for the treasure and make state machine as seen on the Fig. 3.8.1.

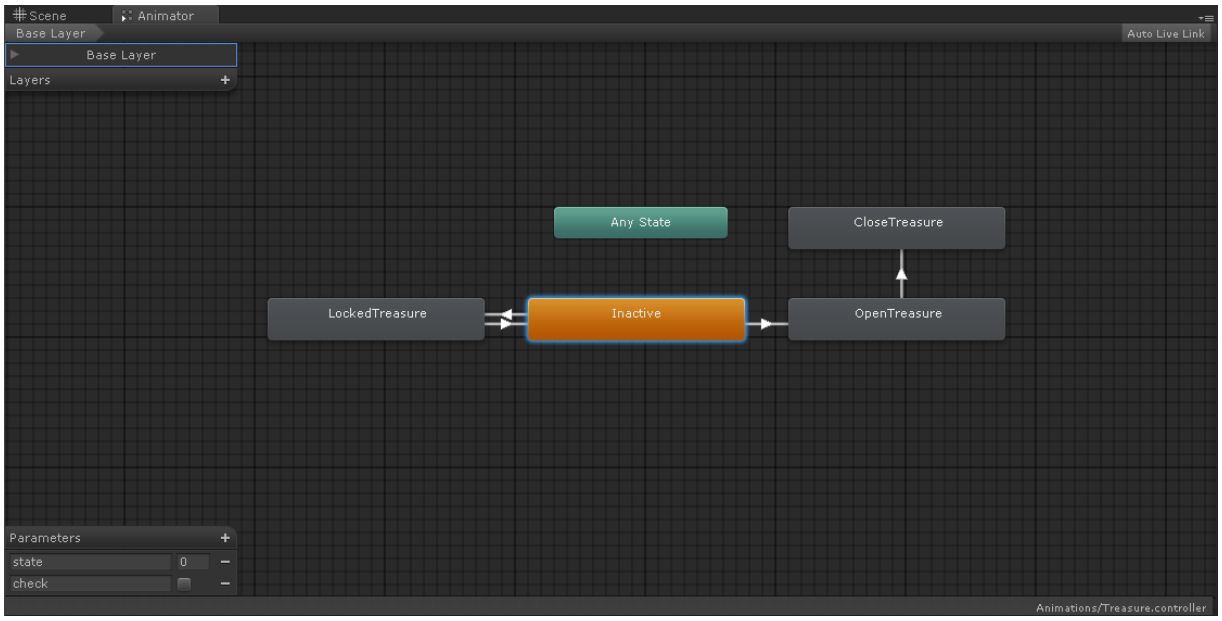

Fig. 3.8.1.: Animator controller of the treasure game object

The "*Inactive*" state is an empty state without any animation. Create the state like this by clicking the right mouse button in the "*Animator*" area and select "*Create State/Empty*". Set this state to the default. Add parameter "*state*" of type "*Int*" and set it to the original value of "*1*", same as in the "*ObjectScript*". Add another parameter of the type "*Trigger*" and name it "*check*". Trigger is a special boolean type, which sets automaticly to false once it's evaluated. It will be useful to avoid the cycle between two transitions. Use those parameters as conditions in state transitions as follows:

Inactive -> LockedTreasure (state == 1; check)

LockedTreasure -> Inactive (Exit Time == 0.50)

Inactive -> OpenTreasure (state == 2)

OpenTreasure -> CloseTreasure (state == 3)

The animation will play only once when the state is entered.

Go back to the "*ObjectStates*" script and add the "*OnMouseDown*" function, where the transition will occur. This function is called whenever the player clicks on this object. That object needs to have a collider assigned. In the body of the function will first be checked if the item is required and if it is in the inventory. If so or no item is required, change the current state value to the next state ID. Also set the "*state*" parameter in the animator to the same next state ID. If the required item is not in inventory, stay in the current state. To avoid infinite cycling between animation states "*LockedTreasure*" and *"Inactive*" and play the animation only once, set the trigger "*check*". The function could look as follows.

```
    void OnMouseDown()
    {
        if(states[currentState].requiredItem != "")
        {
            if(player.GetComponent<Inventory>().inventory.Contains(
states[currentState].requiredItem))
            {
                currentState = states [currentState].nextStateId;
                this.GetComponent<Animator> ().SetInteger ("state",
currentState);
            }
            else
            {
                this.GetComponent<Animator> ().SetTrigger
("check");
            }
        }
        else
        {
            currentState = states [currentState].nextStateId;
            this.GetComponent<Animator> ().SetInteger ("state",
currentState);
```

```
        }
    }
```

Don't use "*null*" in the check whether there is a required item since in the inspector is set an empty string, not "*null*".

The treasure behaves as wanted. Additional features as playing sound effect and disabling/enabling collider were explained in previous lessons. Feel free to try to implement them into this example.

3.8.4 Save / Load feature

Save and load feature an essential functionality of most big current games. Thanks to it, the player can stop the game anytime without the fear of loosing the progress. Saving the game in Unity can be done in several ways. Let's use serialization for storing the data into the file, because it is quite safe and quick method. Thanks to the serialization, nobody else can read or modify the file to cheat in the game.

First make two Unity GUI buttons in the level manager script. One for saving, one for loading. When the player press one of these buttons, appropriate function will be called.

```
if(GUI.Button(new Rect(20,20,100,30), "Save game"))
    SaveGame();
if(GUI.Button(new Rect(20,60,100,30), "Load game"))
    LoadGame();
```

Also create those functions. Use "*void*" as their return type. In the save function, the list of items (strings) from inventory will be stored into the file. Since this is the generic collection, add the required namespace in the top of the script. The manipulation with files requires the "*System.IO*" namespace. Add it ass well. As for the serialization, the class that can serialize and deserialize is the "*BinaryFormatter*". To be able to use this class, import another namespace. Added namespaces are these.

```
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

Create a variable to make a reference to the inventory. Using this variable, the list will be saved. The "*SaveGame()*" function can look like this.

```
    void SaveGame()
    {
        FileStream file = File.Create
(Application.persistentDataPath + "filename.type");
        BinaryFormatter bf = new BinaryFormatter ();
        bf.Serialize (file, inventory.inventory);
        file.Close ();
    }
```

The "*FileStream*" must be opened first. Open the file stream by creating a new file. The file will be located in the "*Application.persistentDataPath*" location, which is usable for any platform other than the "*Web Player*". Define a name of the file in that folder by concatenating the name's string with the data path. Then create the "*BinaryFormatter*". Using this object, you can serialize everything, that is serializable. The "*Inventory*" class is not marked as serializable. Add the serializable attribute to that class to be able to serialize it. Then serialize the list of items from the inventory using the "*BinaryForamatter*" object into the file. When finished, simply call the "*Close()*" function from the file stream.

As for the "*LoadGame()*" fucntion, it can be as follows.

```
    void LoadGame()
    {
        if(File.Exists(Application.persistentDataPath +
"filename.type"))
        {
            FileStream file =
File.Open(Application.persistentDataPath + "filename.type",
FileMode.Open);
            BinaryFormatter bf = new BinaryFormatter();
            inventory.inventory =
(List<string>)bf.Deserialize(file);
            file.Close();
        }
    }
```

First check, if the file already exists. If so, then open a file stream by opening that file. Then create the "*BinaryFormatter*" object. Now you can deserialize data from the file and assing

them to the appropriate object. If there were serialized more than one object in the save function, maintain the same order in deserialization. Otherwise the data could corrupt. Because deserialization function returns a simple "*object*" type, retype that object to the suitable type. The inventory is the list of strings. When finished, close the file.

3.9 Lesson 12 - Useful plugins

Plugins can quicken or simplify the game development. Some plugins however are not maintained anymore, so they might not work with current versions of Unity. Shown plugins are still working on the version 4.5.

3.9.1 Prototype[16]

Prototype plugin serves as a tool to create a simple game environment. When imported into the project, new item "*Tools*" appears in the top menu bar.

Select "*Tools/Prototype/Shape Window*" to open a new window (Fig. 3.9.1). From the "*Shape Selector*" combobox, pick the required mesh. If the checkbox "*Show Preview*" is ticked, the mesh is visible in the scene view with the blue color. This indicates that the object is only previewed. Press the "*Build <object_name>*" button to place into the scene.

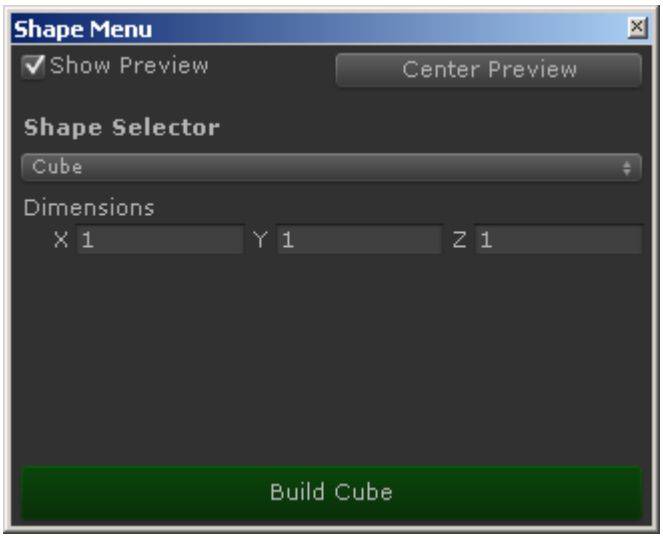

Fig. 3.9.1.: Shape Menu window

Select "*Tools/Prototype/Prototype Window*" to open the "*Prototype*" tab (Fig. 3.9.2). Select the object created in previous steps. Pick edit mode by pressing one of tree options. The left icon indicates vertex edit mode, the middle one is for edge edit mode and the right switches to polygon edit mode. Using this tool, you can do basic edits on the mesh created with the "*Shape Menu*". The buttons works similar as in standard 3D editors. For example by pressing the "*Extrude*" button in edge edit mode, while some edge on the mesh is selected, new edge connecting the previously selected edge by polygon will appear.

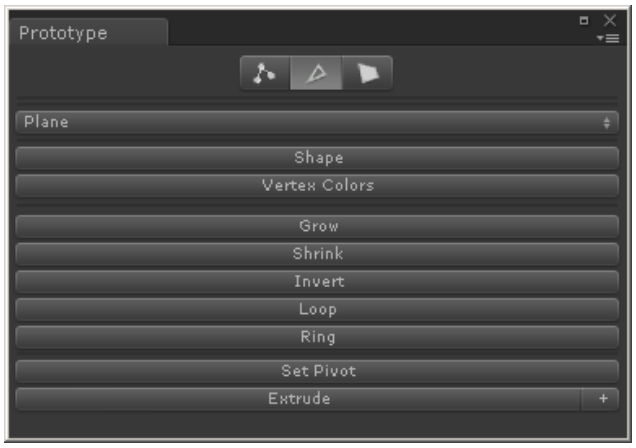

Fig. 3.9.2.: Prototype tab

3.9.2 RAIN[17]

3.9.2.1 About RAIN

Thanks to this plugin, creating artificial intelligence in game is much simpler, than with coding, showed in lesson 7. With RAIN, the simple AI can be done even without the need of writing a single line of code. Let's make an example, in which the enemy walks between waypoints and patrols. When he sees the player, he starts to run towards him. When the player gets out of the field of view, the enemy continues his walk between waypoints. To begin using RAIN, import the package with this plugin into the project. When the package is imported, new item in the top menu bar named "*RAIN*" appears.

3.9.2.2 Navigation Mesh

First of all, the navigation mesh is required for the AI to know, which way he can walk. RAIN has its own navigation mesh. Create it in the scene with objects and the "*First Person Controller*" by picking "*RAIN/Create Navigation Mesh*". New object appears in the scene. It has the component "*Nav Mesh Rig*" (Fig. 3.9.3).

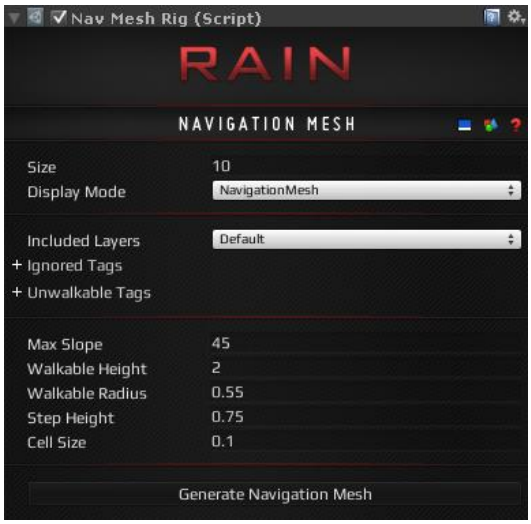

Fig. 3.9.3.: Nav Mesh Rig component

In the Scene view it's represented as an invisible box with white edges (Fig. 3.9.4). This is the area, in which the navigation mesh will be generated. Scale the box so it covers all the objects in the scene. Then press the button "*Generate Navigation Mesh*" in the component. The navigation mesh (Fig. 3.9.4), similar to the Unity's navigation mesh, will generate around all objects with colliders within the navigation mesh cube.

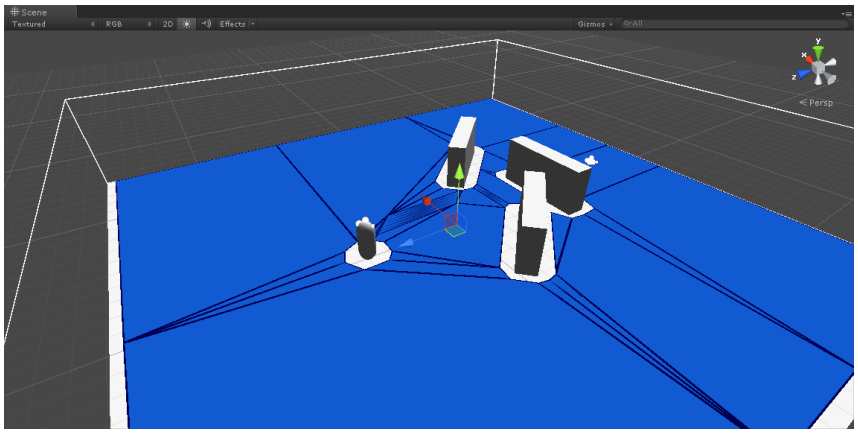

Fig. 3.9.4.: Navigation mesh generated around the "First Person Controller" (the capsule)

There is one problem however. The navigation mesh is generated around the "*First Person Controller*" too. That is not eligible. To exclude the "*First Person Controller*" from the navigation mesh, add it and its children to another layer, e.g. "*Player*". Then in the "*Nav Mesh Rig*" of the navigation mesh object uncheck the layer "*Player*" from the list of "*Included Layers*". Press the "*Generate Navigation Mesh*" button again and this time it will

generate correctly. If there are other objects you don't want to include in the navigation mesh, add it to the appropriate layer and exclude it from the list of "*Included Layers*".

3.9.2.3 Waypoint Route

Select "*RAIN/Create Waypoint Route*" to create a new object, which will later include the way, which the AI would follow. The object contains the "*Waypoint Rig*" component (Fig. 3.9.5).

Fig. 3.9.5.: Waypoint Rig component

By pressing the "*Add (Ctrl W)*" button create a waypoint. Move it to the wanted position. Note that it should be located on the navigation mesh in order to behave correctly. Create another waypoint. It is already connected to the previous waypoint. By adding more waypoints, create a route (Fig. 3.9.6) for the AI to scout.

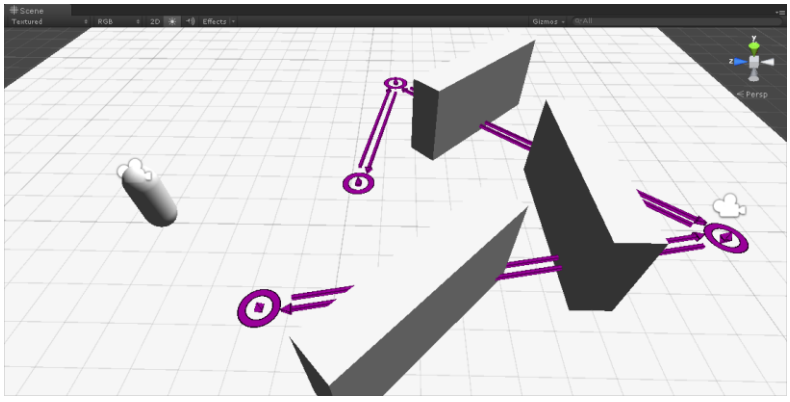

Fig. 3.9.6.: Waypoint route

3.9.2.4 AI

The navigation mesh and the waypoint route would be useless without any kind of artificial intelligence, that would use it. Let's create an object to represent the enemy. Use the primitive

"*Capsule*" object. When created, select it in the "*Hierarchy*" and pick "*RAIN/Create AI*" from the top menu bar. Under the object appears new child object named "*AI*". This object has the component "*AIRig*" (Fig. 3.9.7) attached. The component has 7 tabs. We are going to use the tab "*Mind*" (second from the left) and "*Perception*" (second from the right).

Fig. 3.9.7.: AIRig component with the Mind tab selected

The "*Mind*" tab contains the "*Behaviour Tree Asset*", which handles the intelligence itself. To use the behaviour tree, let's create one by pressing the "*Open Behaviour Editor*" and in the newly opened window (Fig. 3.9.8) select the "*Create New Behaviour Tree*" value from the "*Behaviour Tree*" combobox. Name it appropriately and confirm the name. New tree has already a decition node named "*root*". Decition node picks one or more nodes from its childs. The "*root*" node is of type "*Sequence*". That means, it will execute each node subsequently from the first to the last. Child nodes can be another decition nodes or action nodes. Action node perform the AI behaviour.

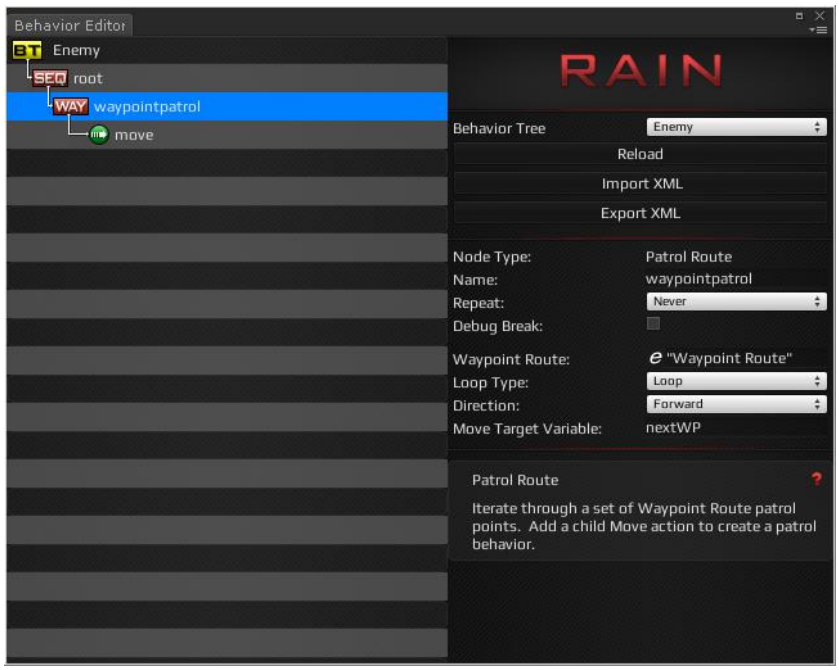

Fig. 3.9.8.: The Behaviour Tree with the Waypoint Patrol node selected

Let's make the AI walking between waypoints first. Under the "*root*" node, create another decition node of the type "*Waypoint Patrol*" by pressing the right mouse button on the "*root*" node and picking "*Create/Decisions/Waypoint Patrol*". Select the newly created node and define the "*Waypoint Route:*" variable by writing the name of the object containing the "*Waypoint Rig*". Since the name can contain spaces, write it into quotes as the "*string*". You can switch the "*Loop Type:*" to the "*Loop*" so when the AI reaches the last waypoint, he continues to the first waypoint. Also define a name of the variable of the next waypoint by writing the variable name, e.g.: "*nextWP*", into the "*Move Target Variable*" textbox. The variable is the waypoint, that the AI is currently trying to reach. Under the waypoint decision node, create an action node of the type "*Move*" by pressing the right mouse button on the waypoint node and selecting "*Create/Actions/Move*". Select the move action node and write the variable name from the "*Move Target Variable*" defined in the parent waypoint node into the "*Move Target:*" textbox (e.g.: "*nextWP*"). To make the AI work with the behaviour tree, assign the tree to the AI's "*Behaviour Tree Asset*" of the "*AIRig*" component. If the correct behaviour tree is assigned to the AI, the capsule walks between waypoints in the play mode using the shortest way on the navigation mesh.

3.9.2.5 Detecting player

In order for the player to be visibible by the AI, select the "*First Person Controller*" and by choosing "*RAIN/Create Entity*" from the top menu bar, add the "*Entity*" object as a child of the "*First Person Controller*". The Entity has the "*Entity Rig*" component attached (Fig. 3.9.9).

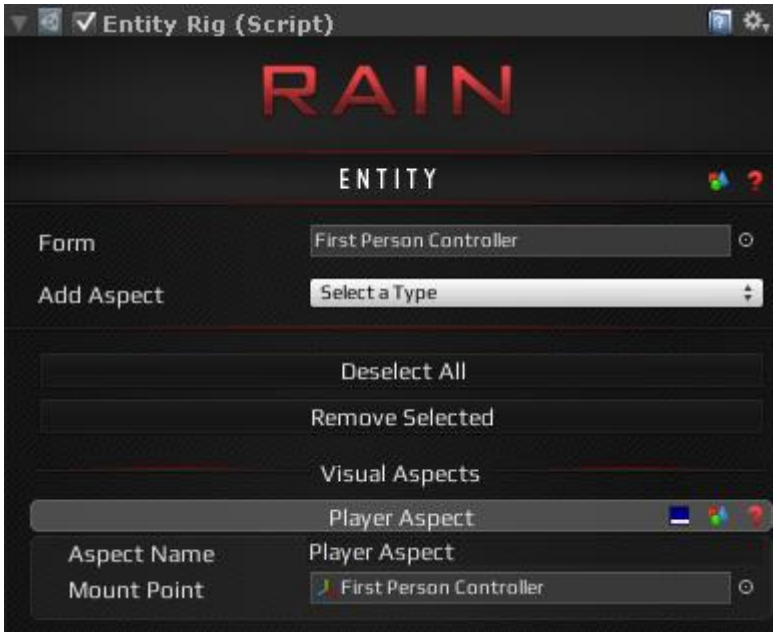

Fig. 3.9.9.: The Entity Rig component with the visual aspect

From the "*Add Aspect*" drop-down menu select the item "*Visual Aspect*". Name the aspect by filling the "*Aspect Name*" textbox. The name can be any string, even with spaces, e.g.: "*Player Aspect*".

Select the AI once more and in the tab "*Perception*" of the "*AIRig*" add the "*Visual Sensor*" by selecting the item from the "*Add Sensor*" drop-down menu. The sensor appears in the list of "*Visual Sensors*". Name the newly created sensor to "*eyes*" for example (The name can be any string even with spaces). Reduce the field of view to correspond the human sight more realistically by changing the "*Horizontal Angle*" to 90 degree and the "*Vertical Angle*" to 70 degree. In the scene, the field of view is represented by wide lines (Fig. 3.9.10).

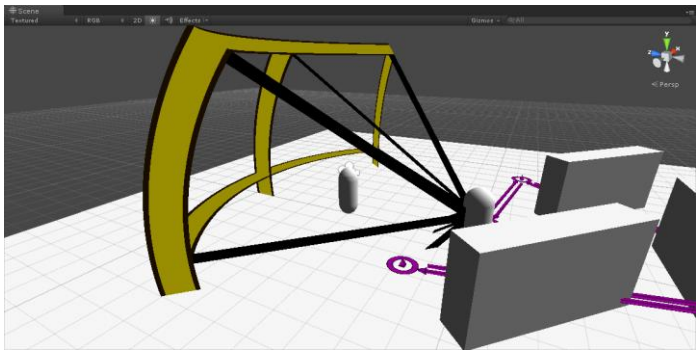

Fig. 3.9.10.: Field of view representation in the Scene view

Open the behaviour tree and change the "*Sequence*" node named "*root*" to the "*Parallel*" node by pressing the right mouse button on the "*root*" node and selecting "*Switch To/Parallel*". Parallel node executes its children simultaniously. Under the "*root*" node, add new action node of the type "*Detect*". In the action node for the detection, fill the "*Sensor*" in brackets. As mentioned above, the sensor is named "*eyes*". Fill the "*Aspect*" by writing the aspect name in brackets. As mentioned above, the aspect's name is named "*Player Aspect*". Also define the name for the "*Aspect Variable:*", this time without brackets, thus without spaces, e.g.: "*varPlayer*".

Under the "*root*" node, add two decision nodes of the type "*Constraint*". This decition node evalutes its condition and if it's evaluated as true, then it continues to its child nodes. Select the first constraint and in the textbox "*Constraint:*" write the following check.

```
varPlayer == null
```

Now move the the node with waypoints under this decision node. This constraint checks, if the sensor sees the aspect defined in the detection node. If not, then the AI walks between waypoints. In the other constraint decision node, write the condition for the oposite result.

```
varPlayer != null
```

Under this constraint, add the action node of the type "*Move*" and as the "*Move Target:*" use the "*varPlayer*" variable. In the play mode, whenever the player gets into the field of view of the AI, the AI walks towards him. If he gets out of the field of view, the AI continues patrolling between waypoints. The final behaviour tree can be seen on the Fig. 3.9.11.

Fig. 3.9.11.: The final scheme of the behaviour tree with the detection node selected

3.9.3 Smart Localization[18]

This plugin offers multilingual support of the game with simple switching ability between individual languages. When the plugin is imported, new option under the "*Window*" combobox in the top menu bar appears. This option is named "*Smart Localization*" Select it to open the localization setup window (Fig. 3.9.12). If the window is opened for the first time, confirm the creation of a workplace by pressing the button.

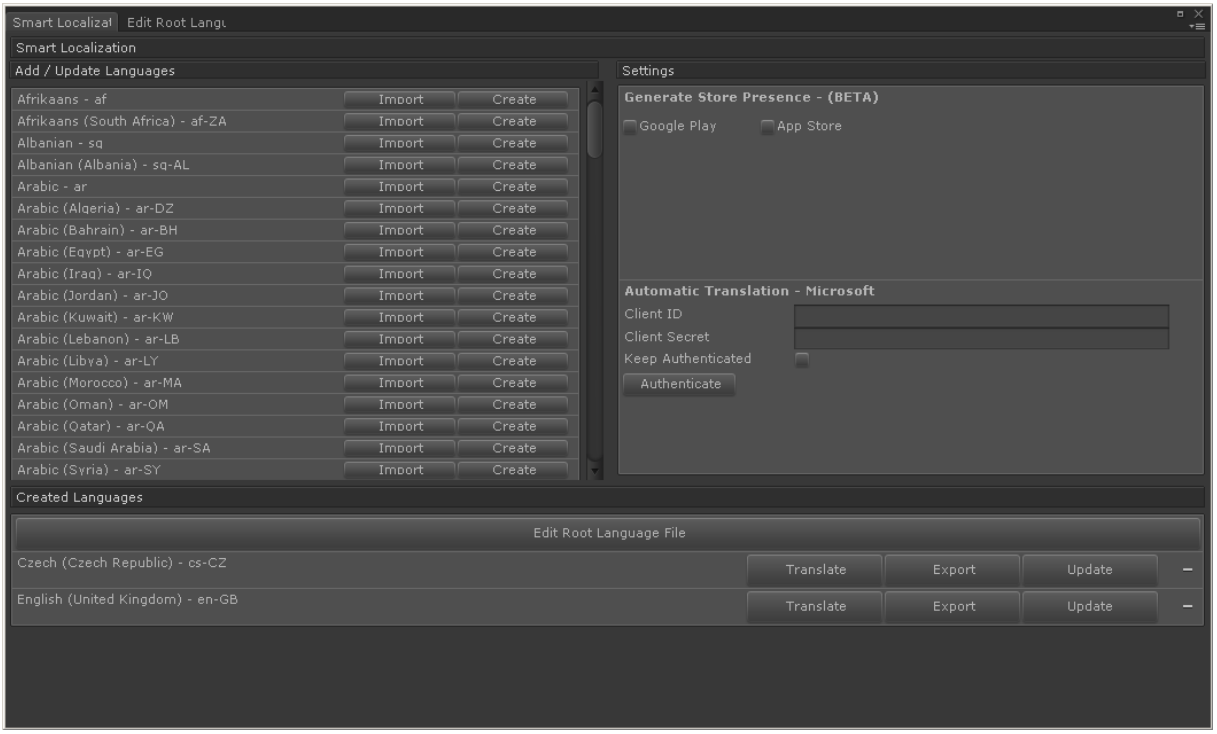

Fig. 3.9.12.: Smart Localization window

Press the "*Edit Root Language File*" To create text key identifications. New tab opens (Fig. 3.9.13). In the "*Key*" column is the identification of the text, which will later be used in the script to print the text from the "*Value*" column from the "*Translate Language*" tab introduced in next steps. The "*Comment*" column can contain either the text in the default language, which will be later copied into the translation of the same language or some comment for better identification of the key. Don't forget to save changes by pressing the "*Save Root Language File*" button in the bottom of the window.

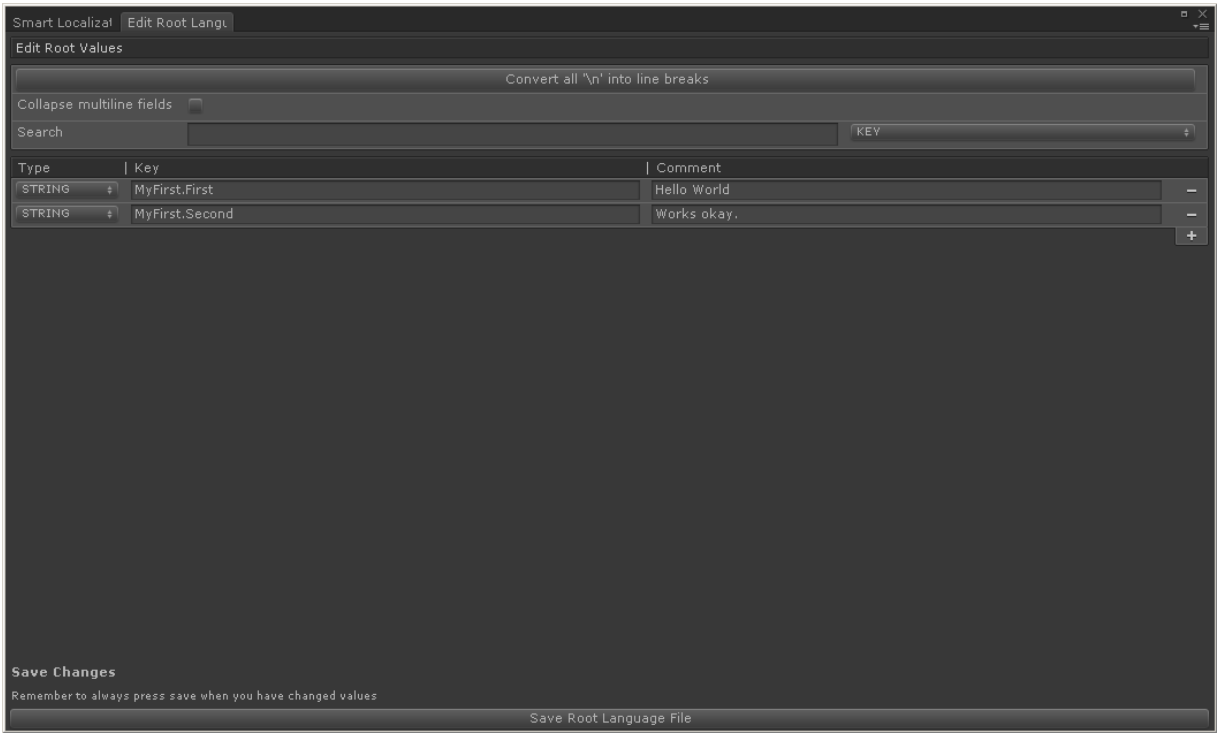

Fig. 3.9.13.: Edit Root Language tab

Go back to the "*Smart Localization*" tab and selected all wanted languages from the list by pressing the "*Create*" button. Let's use czech and english localizations for an example. They appear under the "*Edit Root Language File*" button. Press the "*Translate*" button for the english localization. The "*Translate Language*" tab opens (Fig. 3.9.14). If the root language was english, simply copy the value from the root by pressing the "*Copy Root*" button in the top left column. Don't forget to save changes by pressing the "*Save/Rebuild*" button in the bottom. Repeat the process for the czech language, but instead of copying the root value, write the translation into the column "*Value*".

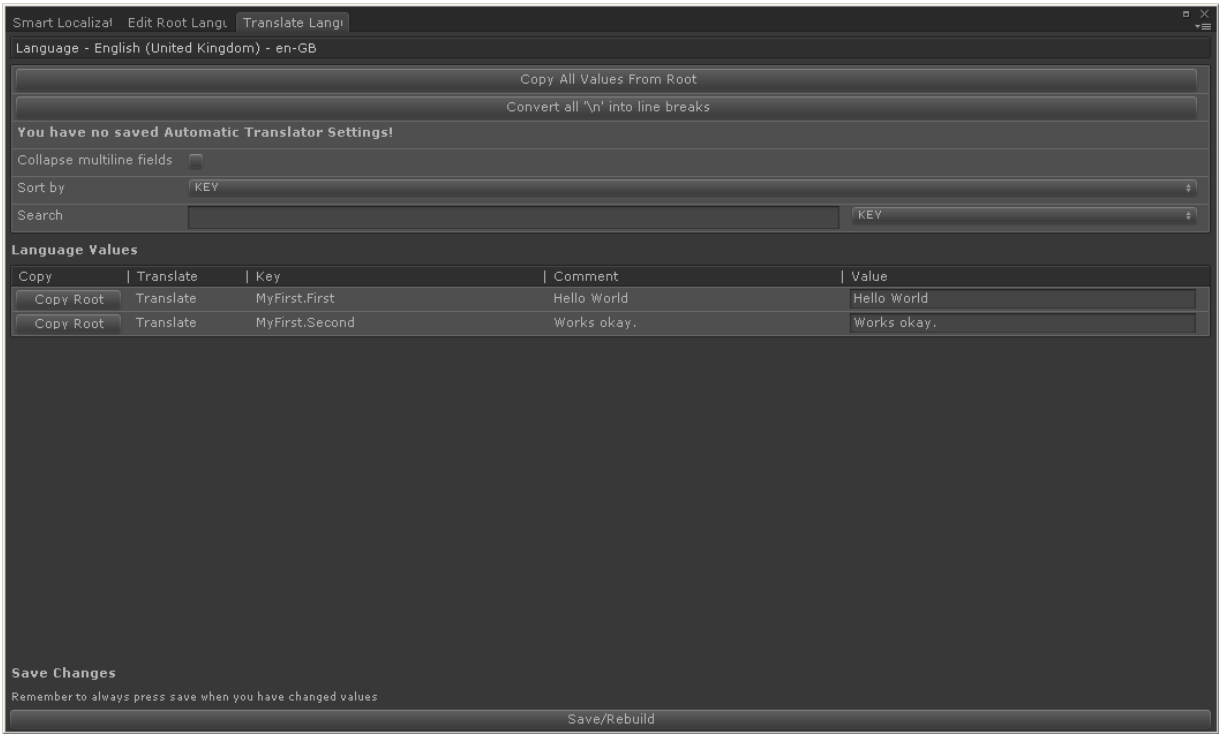

Fig. 3.9.14.: Translate Language tab

Let's write the text by using Unity GUI. To be able to work with the Smart Localization classes, include the "*SmartLocalization*" namespace in the header of the script. Then you can write the text value into the GUI box.

```
    GUI.Box (new Rect (20, 20, 100, 30),
LanguageManager.Instance.GetTextValue ("MyFirst.First"));
```

To access the text value, use the instance of the "*LanguageManager*" class, where the "*GetTextValue()*" function is available. This function takes a string as the parameter. The string is the "*Key*" definded in the "*Root Language*". By that key, the Smart Localization gets the text value in the currently selected language from the "*Value*" column and returns it as a string.

The switching between languages is also done from the instance of the *"LanguageManager*" class. For this purpose is the "*ChangeLanguage()*" function. This function takes an identification of the language in a form of the string. In the Fig. 3.9.12 are identifications visible right from the name of the language. The english on that image has "*en-GB*" as the identification. Also keep in mind, that only languages selected in the "*Smart Localization*" tab can be switched to. For the switching can be used the Unity GUI button.

```
if(GUI.Button(new Rect(20, 100, 30, 30), "EN"))
    LanguageManager.Instance.ChangeLanguage("en-GB");
if(GUI.Button(new Rect(20, 140, 30, 30), "CZ"))
    LanguageManager.Instance.ChangeLanguage("cs-CZ");
```

In the play mode, the switching between languages is instant and the text in the GUI box changes immidiately.

3.10 Lesson 13 - Own plugin

In the previous lesson, some useful plugins were presented. This lesson reassumes to the previous one by introduction the possibilities in Unity used to make plugins. It was prepared by Antonín Hojný, who is one of the teachers of the first semester of this subject. It was inspired by the "*Editor*[19]" tutorials from the official Unity web sites.

3.10.1 Script attributes

It's a command that can for example change the behaviour of the next function or variable. It's written in square brackets. Attributes will be widely used in this lesson.

3.10.2 Execution in edit mode

Let's make some functions, that can be executed even without the need of pressing the "*Play*" button.

3.10.2.1 Context menu and Context menu item

Write the "*ContextMenu*" attribute before the function.

```
[ContextMenu("Name in Inspector")]
```

When the right mouse button is pressed on the header of the custom script component in the "*Inspector*", there is an option named the same as in the "*ContextMenu*" attribute. When it is selected, the function right bellow the attribute executes even when the game is stopped. The only downside is that there is no way of putting parameters to the called function, so only functions without parameters can be used by this attribute.

Very similar is the "*ContextMenuItem*" attribute.

```
[ContextMenuItem("Name in Inspector", "Function")]
```

This attribute works similar as the "*ContextMenu*" attribute, but it is defined above the variables. It works only when there is at least one public variable visible in the "*Inspector*". It takes two parameters. The first parameter is the name, that will show in the "*Inspector*" and the second parameter is name of a fucntion in quotes. As for the "*ContextMenu*" attribute, this attribute also doesn't support functions with parameters. In the "*Inspector*", whenever the right mouse button is pressed on the variable's name, the option named the same as the first parameter of the attribute appears. When selected, the function from the second parameter of the attribute is executed.

3.10.2.2 Execute in edit mode

This attribute makes functions like the "*Update()*", "*OnGUI()*" and similar execute even without the play mode active. It however doesn't work the whole time, as in the play mode but only when some change in the scene is made. To demonstrate this, add the debug text in the "*Update()*" function that writes a number and then increment the number so it changes in each call of the "*Update()*" function. Then before the class add the following attribute.

```
[ExecuteInEditMode]
```

In the play mode the debug text writes a number to the console in each frame. The number is being written in the console even when the play mode is not active, but only when there is being moved with an object in the scene for example[12]. Also, the number keeps changed and when the "*Play*" button is pressed, the incrementation of the number continues from its current value, not from its original value. Beware of any changes in the scene done by script with this attribute, because unlike in the play mode, those changes are permanent.

3.10.3 Default Inspector additions

The "*Inspector*" is one of the most used tabs in Unity. Thanks to it, among other usage dipped in previous lessons, components of the selected object can be seen. Some of the components could be custom scripts.

3.10.3.1 Range of values

The public variables of custom scripts are visible in the "*Inspector*". There is a possibility to limit the range of the numeric variables by adding an attribute to it. Write the following line before the numeric public variable.

```
[Range(0,10)]
```

In the "*Inspector*" this variable is no longer represented as textbox for numbers, but a slider that can only set numbers between 0 and 10 including the borderline ones. In the script however, there can still be assigned a number, that is out of the range. The limit applies only for the "*Inspector*".

[12] It works with any change in the scene.

3.10.3.2 Required component

In certaion circumstances, there is a component required by the script for it to work properly. The good example coul be the "*Audio Source*" component required by the script, which starts playback of an audio clip. The usual approach would be adding the "*Audio Source*" component to the each object individually, which requires it and even still, there might be some overlooked or forgotten objects without that component. Another slightly better approach would be creating a prefab. However, when there are many different types of prefabs, there can still be some without the component. Let's entirely avoid adding the component manually. Write the following attribute right before the class.

```
[RequireComponent (typeof(AudioSource))]
```

Whenever the script is assigned to the object, the "*Audio Source*" component is assigned with it automatically.

3.10.4 Custom Inspector

It is possible to overwrite the default "*Inspector*" of a script by creating a new script in the "*Editor*" folder located in the "*Assets*" folder. The "*Editor*" folder will not be included in the build. In that new script, include the "*UnityEditor*" namespace, and inherit from the Editor class instead of the "*MonoBehaviour*". This enables possibilities of adding labels, fields or buttons to the "*Inspector*". Add the following line before the class.

```
[CustomEditor(typeof(CustomClass))]
```

This tells, which script will be customized. Now add the "*OnInspectorGUI()*" function.

```
public override void OnInspectorGUI()
{
 //your code
}
```

This function is called every time, the "*Inspector*" is drawn. By overwriting the "*Inspector*", nothing is shown in "*Inspector*" of objects using the script defined in the parameter of the "*CustomEditor()*" attribute. Use the "*target*" variable to get the referrence to the object with the overwritten script.

```
CustomClass customClassReferrence = (CustomClass)target;
```

Since the target is of the type "*object*", retyping to the required class is needed. To add an input field for number use the "*EditorGUILayout*" class.

```
    customClassReferrence.numberVariable =
EditorGUILayout.IntField("Number",
customClassReferrence.numberVariable);
```

The first parameter of the function "*IntField()*" defines the name of the field shown in the "*Inspector*". The second parameter takes a value, that will be shown in the input field. The entered number from the "*Inspector*" can be reassigned to the variable or even property of the "*CustomClass*" script.

Similarly could be defined text or float inputs or just labels.

Adding a button is a little tricky. Since the "*EditorGUILayout*" doesn't contain a definition of a button, you need to use the buttom from the "*GUILayout*" class. It's the class that is used in the "*OnGUI*" function used in scripts inheriting from the "*MonoBehaviour*" class. An example of a button would look as following.

```
if(GUILayout.Button("Button"))
{
    //your code
}
```

If the button is pressed, the body of the block will execute.

3.10.5 Custom window/tab

Having a customized tab can lighten a workflow. It could be used for example to generate some content in the scene or showing some values from the scene. This could be achieved with the custom "*Inspector*" as well, but unlike the "*Inspector*", the tab can be seen even without the selected object with the script. To create a custom tab create a new script and include the "*UnityEditor*" namespace. Then change the class, from which this class inherits to the "*EditorWindow*". To be able to open the tab, add it to the top menu bar by using the attribute "*MenuItem*".

```
[MenuItem ("Path/Tab Name")]
```

The parameter of the attribute sets the path in the top menu bar. For example, if the path is "*Window/Tab*", the new item appears in the "*Window*" selection menu named "*Tab*".

The attribute also calls a function. Put the attribute before the function with the "*GetWindow()*" function. That function can have any name.

```
[MenuItem ("Path/Tab Name")]
public static void ShowTab()
{
    EditorWindow.GetWindow(typeof(NameOfThisScript));
}
```

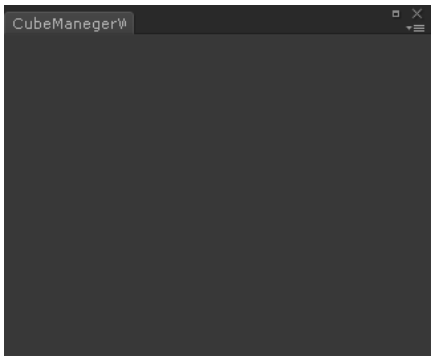

Fig. 3.10.1.: Empty custom window

If the item is selected in the top menu bar, an empty window opens (Fig. 3.10.1). If there are for example to appear items, when some object in the scene is selected, the function "*OnSelectionChange()*" needs to be defined. This function is called whenever some object in the scene is selected. In this function is also used to assign values to variables according to the current selection. Also in order to show gui items, the "*Repaint()*" function needs to be called.

```
void OnSelectionChange()
{
    //value assignments to variables
    Repaint();
}
```

Input fields, labels and buttons are created the same way as with the custom editor with only one difference. They are all defined in the "*OnGUI*" function. If there are too many items so they don't fit the height or width of the window, use the scroll view. It needs to have defined a beginning and an end.

```
Vector2 scrollposition = Vector2.zero;
void OnGUI()
{
     scrollPos = EditorGUILayout.BeginScrollView(scrollPos);
     //GUI items
     EditorGUILayout.EndScrollView();
}
```

The function "*BeginScrollView()*" returns a "*Vector2*" value which tells, where the scroll view currently is. It also takes a parameter of the current position of the scroll view.

4 Conclusion

This thesis focused on preparing study materials forthe subject "*PV255 Digital Games*", which was first introduced in the semester autumn 2014 at the Faculty of Informatics at the Masaryk university. The interest of students surpassed the capacity of the subject. Enrolled students worked in groups of three to four people on 20 projects throughout the semester.

Follow-up subject is planned for the future. It would consist of a one group project that is assigned to students to simulate the environment of a real game company. There is also another subject prepared by Fotios Liarokapis, Ph.D. named "*PA199 Advanced Game Design*" where the game is made from scratch without using a game engine.

5 References

[1] "Computer Games Development." Information System of Charles University. 18 Sept. 2014. Retrieved. 20 Sept. 2014. <https://is.cuni.cz/studium/>.

[2] Game Studies. 1 Sept. 2014. Retrieved. 20 Sept. 2014. <http://gamestudies.cz/>.

[3] Záhora, Zdeněk, and Dominik Jícha. "Herní Předměty v Čechách a Na Slovensku (seznam)." *Game Studies*. 28 Aug. 2014. Retrieved. 20 Sept. 2014. <http://gamestudies.cz/aktuality/hernipredmety/>.

[4] "Game Making University." Game Studies. 1 Oct. 2013. Retrieved. 20 Sept. 2014. <http://gamestudies.cz/game-making-university/>.

[5] Newron. 14 Aug. 2014. Retrieved. 20 Sept. 2014. <http://www.newron.cz/>.

[6] Asset Store. Unity Technologies, 20 Sept. 2014. Retrieved. 20 Sept. 2014. <https://www.assetstore.unity3d.com/>.

[7] "Camera." Unity Docutmentation. Retrieved. 24 Sept. 2014. <http://docs.unity3d.com/Manual/class-Camera.html>.

[8] Zasadnyy, Vitaliy. "Mastering Unity Project Folder Structure. Level 0 – Folders Required for Version Control Systems." Nravo DEV Blog. 12 Feb. 2014. Retrieved. 24 Sept. 2014. <http://developers.nravo.com/mastering-unity-project-folder-structure-level-0-vcs/#.VKL20V4DC>.

[9] "Cat Fighter Sprite Sheet." OpenGameArt.org. 7 Dec. 2012. Retrieved. 24 Sept. 2014. <http://opengameart.org/content/cat-fighter-sprite-sheet>.

[10] MakeHuman. Retrieved. 1 Oct. 2014. <http://www.makehuman.org/>.

[11] Motcap.com. Retrieved. 1 Oct. 2014. <http://motcap.com/>.

[12] Blender. Blender Foundation. Retrieved. 1 Oct. 2014. <http://www.blender.org/>.

[13] Marengo, Riley. "The Zealous Underground (free Videogame Loop)." Soundcloud. 1 Mar. 2013. Retrieved. 5 Oct. 2014. <https://soundcloud.com/rmaren/the-zealous-underground-free>.

[14] Kyaw, Aung Sithu, Clifford Peters, and Thet Naing Swe. "Finite State Machines." Unity 4.x Game AI Programming. Apress, 2013. 232. Print.

[15] Blackman, Sue. "Managing State." Beginning 3D Game Development with Unity 4, 2nd Edition. Apress, 2013. 808. Print.

[16] "Prototype." Asset Store. ProCore, 13 Nov. 2014. Retrieved. 1 Dec. 2014. <https://www.assetstore.unity3d.com/en/#!/content/11919>.

[17] "RAIN AI for Unity." Asset Store. Rival Theory, 19 Dec. 2014. Retrieved. 20 Dec. 2014. <https://www.assetstore.unity3d.com/en/#!/content/23569>.

[18] "Smart Localization 2.1." Asset Store. JaneTech, 17 Dec. 2014. Retrieved. 20 Dec. 2014. <https://www.assetstore.unity3d.com/en/#!/content/7543>.

[19] "Editor." Unity. Unity Technologies. Retrieved. 20 Dec. 2014. <http://unity3d.com/learn/tutorials/modules/beginner/editor>.

Attachments

List of files

On the CD are included packages with complete examples or project in initial states. There are also included plugins used in the lesson 12. Files are following:

```
lesson_1.unitypackage
lesson_2.unitypackage
lesson_3-start.unitypackage
lesson_3-final.unitypackage
lesson_5-start.unitypackage
lesson_5-final.unitypackage
lesson_6-start.unitypackage
lesson_6-final.unitypackage
lesson_7.unitypackage
lesson_11-start.unitypackage
lesson_11-final.unitypackage
lesson_12.unitypackage
Prototype.unitypackage
RAIN_2-1-5.unityPackage
Smart_Localization_2-1.unitypackage
```

Note

New version of Unity was released while writing this thesis. It introduced "*UI*" (User Interface) which makes use of the "*GUIText*" and the "*GUITexture*" obsolete. Also, in the "*GameObject*" selection menu of the top menu bar is restructured. Items in the "*GameObject/CreateOther*" option are now separated into appropriate groups. 3D objects like cube, sphere or capsule are located in the "*GameObject/3D Object*". Sprite can be found under "*GameObject/2D Object/Sprite*", new camera is in "*GameObject/Camera*" and so on. The Fig. A.1 shows the new "*GameObject*" menu.

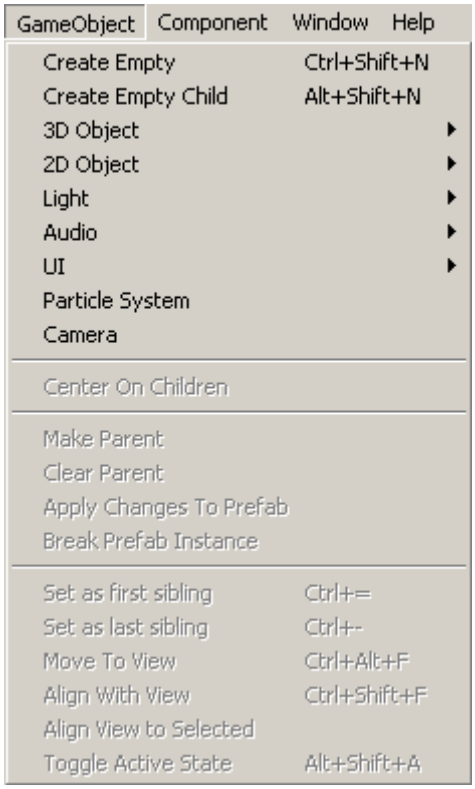

Fig. A.1.: GameObject menu since Unity 4.6