FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Sheepless - An Open-source 2D Adventure Game in Unity
Student:	Ian Mustiats
Supervisor:	Ing. Marek Skotnica
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

Sheepless is an open-source art game about a Shepherdess from Prague. EbSynth is a state of the art image synthesis technology developed at DCGI FEL CTU. This technology is intended to make a hand drawing animation easier. A goal of this thesis is to explore how to take advantage of this technology to design a prototype of a 2D game in Unity.

Steps to take:
- Review the EbSynth technology and Unity.
- Design game mechanics and game architecture.
- Create an open-source proof-of-concept implementation.
- Test and evaluate the proof-of-concept on real users.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 8, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Sheepless – An Open-source 2D Adventure Game In Unity

Ian Mustiats

Department of Software Engineering
Supervisor: Ing. Marek Skotnica

June 4, 2020

Acknowledgements

I would like to thank my supervisor Ing. Marek Skotnica for the support during writing this thesis. I would also like to thank the *Sheepless* team.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on June 4, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Mustiats, Ian. *Sheepless – An Open-source 2D Adventure Game In Unity.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Tato práce demonstruje hlavní vývojové fáze dobrodružné 2D koncept hry. Herní engine Unity a program EbSynth byly hlavními technologiemi pro tvorbu hry. V tomto projektu byly prozkoumány základní prvky Unity a výhody technologie EbSynth, která byla použita k vytváření animací ve hře. Dále tato práce představuje nástroje pro vytváření příběhů a psaní scénářů, které mohou usnadnit proces vytváření hry. Pomocí těchto programů a nástrojů byly navrženy a implementovány herní mechaniky. Pro tyto mechaniky byly rovněž vytvořeny speciální pomocné prostředky pro Unity, které zlepšují práci s nimi. Na základě vytvořených nástrojů a mechanik byl implementován koncept 2D dobrodružné hry. Výsledky práce s nástroji a zdrojovým kódem jsou k dispozici v online repozitáři.

Klíčová slova Unity, Dobrodružná hra, EbSynth, 2D, Sheepless

Abstract

This thesis demonstrates the main development stages of the proof-of-concept 2D adventure game. Unity game engine and program EbSynth were the major technologies for the game creation. In the thesis were researched the basic Unity elements and advantages of EbSynth technology that was used to make the animation in the game. Also, were presented the basic storytelling and script writing tools that can facilitate the creation of the game. Using these programs and tools were designed and implemented game mechanics. For these mechanics were created special tools for Unity that improve convenience work with them. Based on the created tools and mechanics were implemented 2D adventure concept game. The result of the work, along with tools and source code, is available in the online repository.

Keywords Unity, Adventure game, EbSynth, 2D, Sheepless

Contents

List of Figures

Code list

Introduction

This thesis is based on the results and achievements of the technical demo game called *Sheepless* [1]. It is the open-source 2D art game about a Shepherdess from Prague. My colleagues, Jan Klicpera and Robert Badronov, also developed different components for this game and the results are presented in their theses[2, 3].

Motivation and Objectives

The gaming industry is one of the largest entertainment industries in the world [4, 5]. Global revenues of the video game industry are growing for the last 20 years [6] and suppose that by the 2025 year it will reach $300 billion [7], that only increases the desire of independent developers to try themselves in this direction. And they decide to make an independent product - a game.

The main development environment for each game is a game engine. And several generally accepted engines are used by most developers. One of these engines is Unity. It is one of the most popular engines of our time [8]. This engine was chosen for creating my game concept. Unity has a lot of advantages that will be discussed in chapter 1.

However, despite all its advantages, Unity is quite difficult to master for people unfamiliar with programming and game development. That's why in the Unity exists capability to add ready-made solutions that will facilitate development. In chapter 3, I explored existing storytelling and script writing tools to choose the most suitable for my concept.

One of the most important elements of the game is a graphic style. This is the first thing that players see in the game. The graphic should be unique or interesting. For my concept game as the graphic style was chosen the hand-drawn style. However, creating animation in such style is very difficult because it requires excellent animation skills. And for me was very important to find out whether it is possible to design and develop a 2D game without using

traditional animation techniques. The alternative was artistic style transfer technologies. A great example of such technology is EbSynth which is described in chapter 2. This program allows transferring ordinary video into an animation that looks like it was drawn by hand.

The main objective was to design and create a 2D proof-of-concept adventure game in Unity using EbSynth technology for creating hand-drawn animation.

Problem statements

2D adventure games have a variety of mechanics and game systems [9]. However, it is impossible to create and use each of them, as it will ruin the game or greatly complicate the development. That's why it's very important to select the appropriate mechanics and accurately determine the scale of the game. And even when the necessary components of the game have been selected, it is very important to choose or create suitable tools for creating and using game mechanics. This problem is considered in chapter 4.

Structure and Methodology

The thesis is organized as follows:

- Chapter 1, the Unity engine and its main components are overviewed with more details.
- Chapter 2, talks about the animation and its production using EbSynth technology.
- Chapter 3, describes the selected tools for creating an adventure game in the Unity engine.
- Chapter 4, game design workflow is more precisely defined.
- Chapter 5, demonstrates the implementation of the 2D proof-of-concept adventure game.
- Conclusions, summarizes the current results and mentions further work.

Chapter 1

Unity

The whole game was implemented using the Unity game engine, which allows creating 2D or 3D games and applications. Using this engine can be created a wide variety of games for a large number of platforms. It's possible thanks to the rich tools that this engine provides.

Unity is distributed free if the annual income from the game does not exceed $100 000 [10]. In other cases, it is possible to choose from several paid models that provide additional services (such as LiveOps analytic, custom splash screen or premium courses). Also in February of 2020 Unity introduced a new plan for students, which "can get free access to the same tools and workflows that professionals use on the job in industries like gaming, architecture, engineering, automotive, and entertainment." [11]

This section will tell about the main parts of Unity. Basic information about the engine was obtained from the manual pages of Unity [12].

1.1 Engine overview

The game development takes place in two main parts of Unity: Unity Editor and scripts editor. Unity Editor consists of areas, tabs, menus and buttons, where developer "put together" all game parts. As scripts editor could be any code editor. To write a script uses the programming language C#.

1.1.1 Common editor windows

Unity editor composes of several windows based on a *Drag and drop* operations and developer can quickly transfer an object from one window to another.

The most popular and important windows are Hierarchy window, Game and Scene view, Inspector, Project window, Toolbar and there are a lot of other windows in the editor.

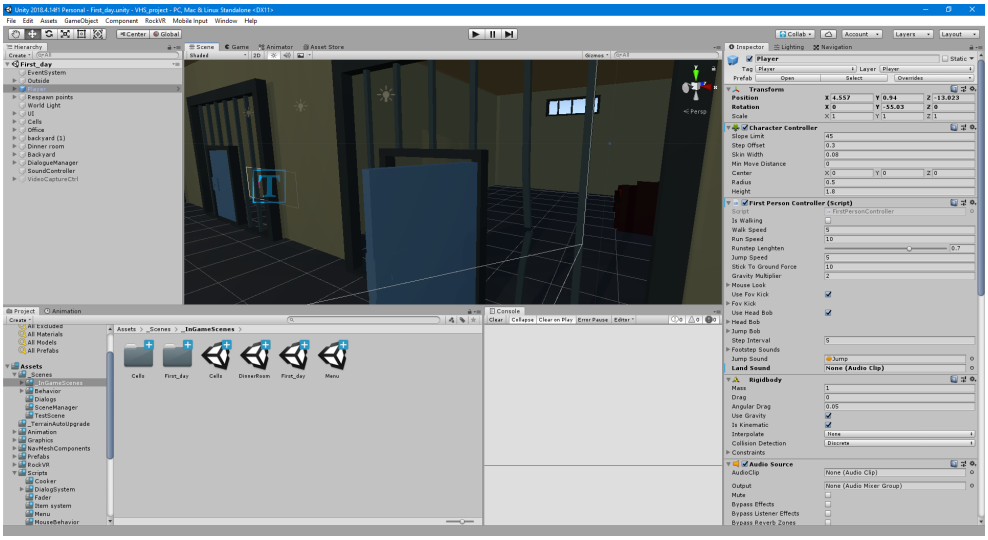

Figure 1.1: Unity Editor with custom settings

Hierarchy window This window contains every GameObject in the active scene. Also, it is possible to open more scenes in the Hierarchy window. The main functions are a creation of new GameObjects or adding instances of Prefabs, sorting them (Alphanumeric sorting) and grouping together (this process is called Parenting).

Game view window Shows how a final rendered game looks like from the camera(s) view in the scene. Using the Control bar of the window, you may change resolution and aspect to simulate different display and check how the game looks.

Scene view window It is an interactive window, which allows visually to edit the scene. You can modify GameObjects, change their position, size, rotation and navigate through the whole scene.

Inspector Allows modifying all the properties and settings of the selected GameObject. For each GameObject you can add, remove or change components.

Project window The window serves for displaying project files. It is possible to create new files or import new Assets from the computer system. An Asset is a representation of any object that can be used in the game.

Toolbar Provides important features for working with the editor. Here you can start a game, pause it or load next frame during pausing. One of the main features is editor layout, which allows customizing the workspace.

1.1.2 Scene

Scene in the Unity represents the 2D or 3D environment of the game, where you can place new objects and decorations. An entire game could consist of several scenes that will switch in a certain or random sequence.

1.1.3 GameObject

GameObject is a fundamental part of Unity because every object in the game is a GameObject. However, a GameObject is just a base that is complemented by components as needed.

Also, GameObject always has a Transform component (or RectTransform component if it has User Interface components). Here could be changed the position and orientation of the GameObject. This information is relative to the Transform's parent or to the world space if the Transform has no parent.

1.1.4 Components

Each GameObject has different components that could change representation and behaviour of the object in a game. An important part of the Components is flexibility. Different properties can be modified during building a game or when a game is running. Be aware that component values don't save when a game is running.

1.1.5 User Interface

Unity provides 3 user interface toolkits for creating UI:

- **UIElements:** is a retained-mode UI framework for creating a user interface. This toolkit is based on web technologies and lets developers create own hierarchy and styling in the separate assets.
- **Unity UI:** It is a GameObject-based UI toolkit for creating in-game UI. It can't be used for creating Unity Editor UI. It allows changing style and orientation of the user interface using components.
- **Immediate Mode Graphical User Interface:** It is a code-driven UI toolkit for creating Unity Editor UI, like custom Inspectors for components or new editor windows, or in-game debugging tools.

1.1.6 Scripting

The main language for creating scripts in Unity is C#, which allows control of different aspects of the GameObject and his components. On the Unity official website mentioned, that "you don't need to create the code that runs

the application, because Unity does it for you. Instead, you focus on the gameplay in your scripts." [13]

Also, scripts should derive a base class for their use as components.

Basic class in the Unity is **MonoBehaviour** which every in-game script should inherit. It provides access to the game object, that have this script as a component, but also afford some important methods, like **Start** or **Update**. Start is called once when a script is enabled before the first call of Update methods. Update is called every frame if a script is enabled.

Another base classes are **Editor** and **EditorWindow**. Editor class provides tools for creating custom Inspector or editor of the existing script. EditorWindow class allows creating any number of custom windows.

1.2 2D and 3D modes

One of the features of my concept game is the combination of 2D and 3D. The whole environment and every object in the scene are 3D. But all the objects in the game are presented in the form of a picture - a **Sprite**.

Unity has different tools for 2D and 3D environment. However, some 2D tools cannot be used with the 3D environment, like 2D Lights from Universal Render Pipeline [14] or various 2D components.

1.2.1 Physics

Unity uses two different engines to handle game physics - **Nvidia PhysX** and **Box2D**. PhysX is used for 3D physics and Box2D for 2D. Each physics engine has its components that may not work in a different environment.

RigidBody

The main component that controls the position of the object is RigidBody. When RigidBody is added, the object will be immediately realistically controlled by the physical engine – the object will be affected by gravity and react to collisions with other incoming objects, even without adding a piece of code. GameObject required **Collider** component for checking collisions.

RigidBody component could be controlled by properties: mass, linear drag or angular drag which help determine a physical behavior of each object. Another important property called **Is Kinematic**. It controls whether physics affects the rigidbody and allows to move the object from a script.

In the [15] recommends to use **FixedUpdate** function for any changes through the script because "physics updates are carried out in measured time steps that don't coincide with the frame update. FixedUpdate is called immediately before each physics update and so any changes made there will be processed directly."

Colliders

Collider components define the shape of a GameObject for physical collisions. It's invisible in the game scene and could be bigger or smaller than GameObject's mesh.

In the Unity exist 3 primitive collider types:

- **Box Collider:** is a rectangular box and useful for walls, crates or chests.
- **Sphere Collider:** is a sphere-shaped primitive and can be used for balls or other rolling objects.
- **Capsule Collider:** looks like two hemispheres connected by a cylinder. The most popular use is a game character.

To create a more complex collider, is used **Mesh collider** which is based on the GameObject's mesh. But there are several limitations to using them. In the manual mentioned, that "these colliders are much more processor-intensive than primitive types, so use them sparingly to maintain good performance. Also, a mesh collider cannot collide with another mesh collider." [12]

Also, colliders interact differently depending on the properties or Rigidbody component. There are 4 important configurations:

- **Static Collider:** is a GameObject with Collider component but without Rigidbody. It is used for level geometry (walls, ground, etc.) or any static GameObjects because they never move.
- **Rigidbody Collider:** is a GameObject with Collider and normal Rigidbody. This configuration allows a GameObject to react to collision and different forces.
- **Kinematic Rigidbody Collider:** is same as **Rigidbody Collider**, but Rigidbody is *kinematic*. This object won't respond to collisions but will register it and can be moved from a script by changing **Transform component**. This property could be switched at any moment of the game between kinematic and normal that that allows to change the object from static to dynamic and back.
- **Trigger:** is a GameObject, where collider is using **Is Trigger** property. It doesn't behave like a solid object but will detect any collisions.

1.3 Graphics

Unity provides a large number of different tools and settings, thanks to which it is possible to create completely different graphics for a game – from realistic to hand-drawn.

1.3.1 Lighting

Lighting is a very important part of the game because it's defining how a scene is perceived. Different parameters of the lights could make the scene more dramatic, scary or cheerful.

In the Unity, Lights are the GameObjects with a Light component. There are 4 types of lights:

- **Directional Light:**is an environment light which illuminates an entire scene (similar to the sun). A light object can be placed anywhere in the scene, and all objects will be illuminated from the same directional.

- **Spotlight:** is a cone-shaped spot light. It works as flashlights or car's headlights.

- **Point Light:** is a spherical-like, omnidirectional light. In the [16] says that "the point light is the most common type of light for illuminating interior areas."

- **Area Light** is a baked-only light type that defined as a rectangle. This feature exists for a process called *lightmap baking* and allows smooth, realistic shading.

Lighting techniques

All lights in the games are more or less computational. And lighting can be figured out by the machine in real-time or precomputed. For creating more fascinating scene can be used both techniques in combination. But these techniques have their opportunities and advantages.

Realtime lighting is a basic way of lighting in Unity, and it's the default for directional, spot and point lights. It updates every frame and useful for illuminating characters or any dynamic objects.

Backing technique, on the other hand, can process static lighting effects (with the feature called **Global Illumination**) and results are written to the texture maps – **lightmaps**. One of the greatest advantages of this technique is that "by saving the effect of lighting into a texture, we can boost performance and improve the look of our game environments". [17]

1.3.2 Particle Systems

For different effects like smoke, explosions, flames and enc. is used particle system. **Particles** are small images or meshes that are emitted by a particle system. Each particle represents a small part of the effect, but all of them are changing and moving together.

Unity **Particle System** is a component made of various modules. And each module allows to change different parts of the particular system and create a suitable effect.

1.4 Optimization

In our days most of the computers are powerful to run complex games or applications. About program optimization reasoned computer scientist Michael A. Jackson: "First rule: don't do it. Second rule (for experts only): don't do it yet - that is, not until you have a perfectly clear and unoptimized solution." [18] This means that standard or naive solutions can be used in the project without any problems. But if a game is developing for mobiles which are not so powerful as computers, so there is some risk to create something unbalance. In this case, Unity offers powerful tools to help with solving problems with optimization.

1.4.1 Profiler

It's a native tool that helps to examine how a game performs. Profiler gives information about CPU and GPU usage, rendering, memory, physics, etc. for each frame. By exploring the detailed information, it's possible to determine the component-level problem. One of the important function of this tool is **Deep Profile** which can examine all script usage including function calls.

1.4.2 Occlusion culling

Another important feature in Unity that will optimize a scene with a lot of geometry is **Occlusion culling**. With this feature, Unity won't render static objects that are not seen in the camera. An example will be building and rooms inside. If the camera sees only external walls of the building, then all the rooms inside won't be rendered.

Chapter 2

Stylized animation

An especially important part of any game is the art style. It can be primitive, fairy or realistic, and it depends on how the player will perceive the game and how he can dive into the game's world. And there are a lot of different styles: from pixel art to anime, from low poly to high poly.

In my adventure prototype game was used the hand-draw style, where all the objects look as "from a real picture." Creating such a style is a rather difficult process. Each object and animation drew manually, increasing the development time. Therefore, such a style is only capable of large teams or very experienced artists.

However, we live in a time of technological, where each process is simplified with the help of smart algorithms. And for creating animation can be used image synthesis software.

2.1 Animation

Animation is a method that creates the illusion of movement to an audience by the presentation of sequential images in rapid succession [19]. Exist a lot of different types of animation, each of which has its own rules and methods of creation. I'm going to focus on the most prominent.

Traditional animation

It's one of the oldest technique where each frame is made individually. Also called hand-drawn animation or cel animation. Traditionally animators draw a sequence of animation on celluloid transparent sheet which is illuminated from the backside. Thanks to this, the animator can see the previous frame of the animation and can create the next one, one frame at a time [20]. But in our days this technique is time-consuming and animators are using more modern methods.

Computer animation

It contains a variety of techniques where animation is created or generated digitally on a computer [21]. Exist two basic techniques: **3D** and **2D**. 2D technique is focused on image manipulation (2D bitmap or 2D vector graphics) while 3D animation is using 3D objects that will be rendered to the frames. For computer animation exists various programs that simplified animator's work and make it possible to create more realistic animation.

Stop-motion animation

Stop-motion animation is a technique in which animation created by moving or changing real-world objects and filming them frame by frame [20]. The most common objects are puppets with movable parts or plasticine figures because they allow to change the movement of an object quickly and easily.

2.2 Image synthesis

Image synthesis is a process of generating new images from various data of images or their description. Manipulations of data result in modifications of the resulting image and knowledge of their relations is necessary for controlling the process of generation [22]. Most often, the generation of synthetic images happens through the use of artificial intelligence algorithms. And especially are using the certain part of these algorithms – **deep learning** or **deep neural networks** [23]. Deep learning refers to the architectures which contain multiple layers to learn different features and improve with experience [24]. As a result, it's possible to generate images with places or objects that don't exist in the real world.

2.2.1 DeepDream

DeepDream is a computer program developed by Google's engineers[25] and uses a **Convolutional Neural Network** (a class of deep neural networks). A convolutional neural network is designed to analyze visual imagery and can help to identify certain objects. Deep dream has expanded the capabilities of this network and able to reproduce these objects based on images patterns.

Each layer of the network analyzes various details of an image. Some layers detect borders and edges of an image, others identify colours and orientation. And the final layers react to more complex objects like buildings, trees, animals [26]. For example, an image can be identified to more "cat-like" and DeepDream will try to find cat patterns in an image. The code of the program was published as open-source [27], due to which appeared a large number of artworks that using this technology.

Figure 2.1: Example of the Deep dream image [28]

2.2.2 StyleGAN

It's a Style-Based architecture for **Generative Adversarial Networks** introduced by NVIDIA researchers [29]. Generative Adversarial Networks is a deep neural network architecture that is composed of two networks: *generator* and *discriminator*. The generator produces new sample data and discriminator is trying to estimate if it was generated or it's real. And through multiple cycles, both networks train each other while trying to outwit each other [30]. After the training, the generator can be used to generate new samples.

The advantage of the StyleGAN is the ability to control style properties of generated images. For the generator, all images are a collection of styles and each style controls a certain effect superimposed on the generated image. There are three basic layers:

1. Coarse styles (that changes a subject's pose and shapes)
2. Middle style (affects finer objects features)
3. Fine style (governs color scheme and small features)

The main principle of StyleGAN is progressive training where the training starts with very low-resolution images (coarse styles) and gradually adding a higher resolution layer (middle and fine styles) [31]. As a result, you can get images with resolution 1024x1024 pixels. This technology allows to create a large number of pictures with unique objects, like fake people faces [32].

Figure 2.2: Generated art works by GAN [33]

2.2.3 StyLit

StyLit demo is an interactive demonstration of StyLit algorithm that was introduced by researches from Czech Technical University in Prague and Adobe Research [34]. The system for the example-based stylization of 3D renderings allows transforming art style from the painting to the 3D model. The algorithm is based on light propagation in a simple 3D scene. It requires three images:

1. An exemplar of the simple 3D scene with important illumination effects (like direct diffuse, direct specular, global illumination, etc.)
2. A stylized example of this scene
3. Image of target 3D scene with the required object

The task of the algorithm is to transfer the style of stylized example to the target picture, based on the light properties of the scene.

StyLit demo makes it easy to produce stylized pictures. As input, it requires stylized example painting that program gets as live video input. So example may be created by hand and captured by a camera or created with using an image editor and caught by screen capture. As an output will be a stylized image of one of ten provided objects.

2.2.4 EbSynth

EbSynth is a free application based on research from the Czech Technical University in Prague and Adobe Research [35]. It's Fast Example-based Image Synthesizer algorithm, with which you can transform video into stylized animation in the shortest time.

The program is easy to learn and doesn't require special knowledge. First of all, will need an image sequence of the video which will be transformed.

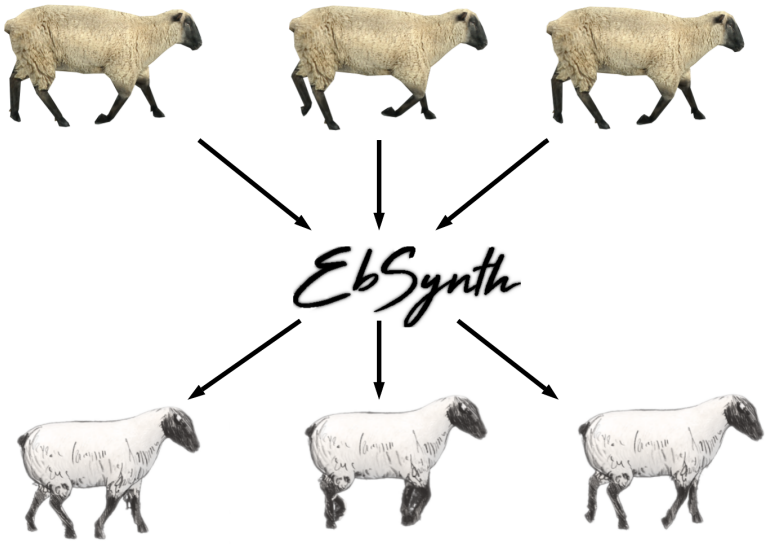

Figure 2.3: Translating video to the stylized animation in the EbSynth. Style exemplars by Polina Akhmetzhanova © 2020

Also, will require keyframe painting that will be the reference to the art style. Based on the keyframe, the program transfers the style to the rest of the video. It's possible to add more keyframes for complex video content. But if video frame contains objects that aren't in the reference painting, then EbSynth won't work correctly [36].

Another important element of the EbSynth is the mask which marks an area that will be stylized. It's using to remove background from the video frames, and EbSynth will focus on the video objects. Each element has a parameter **Weight** that enhances the effect, whereby the result looks more stylized or closer to the original video. After all the processing, you will receive a sequence of frames in the new style that can be used in the project.

Chapter 3

Storytelling and script writing tools

Game creation requires programs and tools to create sounds, images, 3D models, etc. The development of such programs is a laborious and long task, which is suitable only for large companies. But most often, all the necessary programs for development already exist. You need to pick the right one and use it.

However, choosing the wrong technology at the planning stage can cause many problems during development. For example, the game will not be optimized or will not be completed at all [37]. Therefore, I would like to tell about the most suitable tools that will allow you to create a 2D adventure game.

3.1 Tools for Unity

Unity Asset Store is a library of free and commercial assets created by Unity Technologies or members of the community. It contains a huge number of different assets - from textures and models to add-ons and tools. You can get to the Asset Store using either a browser or Unity, which has a separate window for the store.

Assets greatly facilitate the development of the game, so you can focus on other parts of the project. A good example is toolkits that extend the functionality of Unity and can be used to create 2D or 3D games without coding.

3.1.1 Fungus

Fungus is free and open source tool for creating interactive games in Unity [38]. This tool allows to create a lot of different games, like visual novels, hidden objects of RPGs.

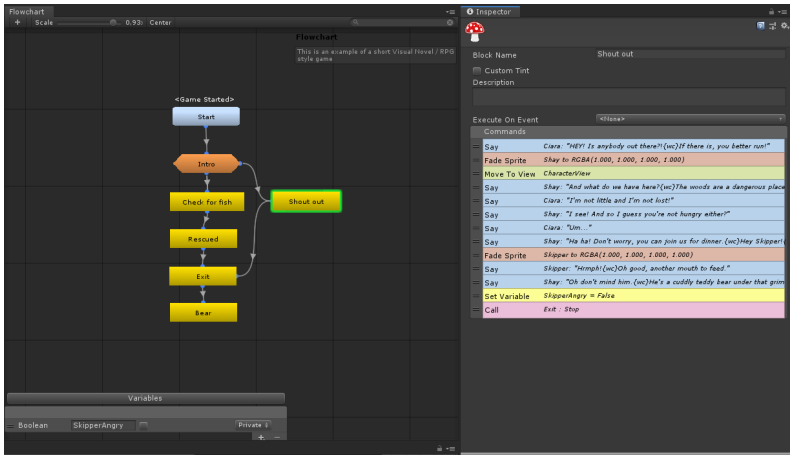

Figure 3.1: Flowchart window with Blocks (left) and Commands (right)

A fundamental concept of Fungus is the **Flowchart**. Flowchart in the game represents a specific sequence of actions that make up the game logic. It is a component which is controlled through the window. Flowchart window is a node-based editor where each node is a **Block**. Blocks are another fundamental part of the Fungus.

Exist 3 types of Blocks:

1. **Event Block** is a blue node that triggered by an event. There are different types of events (such as Game Started, Key Pressed, Message Received and others).

2. **Branching Block** is an orange node which can transfer control to other Blocks.

3. **Standard Block** is a yellow node that represents a simple Block with no events and can pass control only to the next Block.

When the Block starts, are sequentially launched **Commands**. Each Command can be considered as an action that occurs during the game. Therefore there are a large number of Commands to operate the camera, Blocks, events, sprites and so on.

For sharing data between Flowcharts, Commands and Blocks are used **Variables**. This concept is familiar with scripted variables which contain different data depends on the type. Variables don't save and restore values between game sessions, however, they can be used for this mechanism. Types of Variables can be both: standard (like in the C#) and Unity types (Vectors, Image, Texture, GameObject etc.).

Fungus is simple to use tool and ideal for teaching game development or using in the game jams. With it, you can quickly create interactive dialogue

system (with characters, effects and localization) or drag-and-drop system for interacting with game items. Also, Fungus Asset contains a large number of demo scenes where you can take a closer look at using this tool.

3.1.2 Adventure Creator

Adventure Creator, or "AC", is a toolkit that provides foundation and instruments for creating 2D, 2.5D and 3D adventure games [39]. AC contains ready-made solutions for inventory, dialogues, movement and other systems without the need to coding them.

All the main elements of development using AC are controlled by managers which control various aspects of the project. At the beginning should be created own managers of the project with **New Game Wizard** feature. It consists of a few simple steps with the basic options which will set up the game.

After determining the project settings, they can be modified through eight managers.

Scene Manager This manager shows unique settings for each scene and allows to create different AC objects. It organises a "regular" Unity scene into an "AC" with new hierarchy. Here it's possible to change default scene settings, set up cutscenes, create attributes and add AC prefabs to the scene.

Settings Manager Here are defined major settings of the project. This manager consists of 15 sub-sections that cover almost every aspect of the project, like Interface, Movement, Audio, Inventory, Cutscenes, etc.

Actions Manager It provides a list of **Actions** of the project. Action is the main block of AC's visual scripting system where each Action has a different task. Game logic and cutscenes are formed of a sequence of Actions. There are over a hundred basic Actions and can be more by creating custom Actions.

Variables Manager Variables have the same function as in Fungus – keep track of progress and share data between game scenes. In the Variables Manage can be defined different variables.

Inventory Manager During adventure games, the player can find items and use them depending on the game mechanics. Through this manager, it's possible to create game items and define categories, item's properties or crafting recipe.

Speech Manager It used to control the representation of the speech in the game and how it sounds. Also, the manager handles translations of the languages. It even has **Lip syncing** that can help to synchronize animation of the lips with the speech.

Cursor Manager The manager will define the graphics of the cursor while the player is interacting with NPCs, inventory or other active objects. The cursor can also be turned off during cutscenes.

Menu Manager In this manager can be constructed game's UI. Default interface provides a list of menus for inventory, game menu or options, dialogues and other. An interface can be rendered with AC's system or with Unity UI.

Adventure Creator is a very powerful toolkit with a lot of different features that can be used in the game. In addition to the mentioned features, it contains others that allow working with game characters, cameras or different interactions. However, some tools can be used only with a specific game (2D, 3D or 2.5D). For better understanding, AC has demos where it's possible to see how these tools are using in the 2D or 3D games. AC also provides API, so its capabilities can be expanded.

3.1.3 Game Creator

Game Creator is a set of tools that provides different features to create any game of any genre [40]. This toolkit represents a base that can also be supplemented by **Modules**.

Game Creator consists of different components that are used together and implement the main logic of the game. The following components are responsible for interactive events in the world:

- **Actions** are a list of commands that run sequentially. They allow to work with the camera, animation, game status and so on..
- **Conditions** have a similar function as **Actions**, but they also check execution conditions. Whole process is based on the **if/else** statement.
- **Triggers** are run **Actions** nad **Conditions** when player perform a specific action, like pressing a key or entering a certain area.

Among other things, other components are used in the game world. Game Creator offers **Character** and **Player** components with fully animated humanoid 3D model. Components allow simply change default model to custom character model or add new animation states. Like the previous toolkits, that one has **Variables** for processing and storing data during the game.

Game Creator have special systems for more complex games, like **Event System** that can control **Triggers** and **Actions** for better flexibility. It is a translator that keeps a list of objects which can be notified from another game object. But most of them complements Unity systems, like UI or Timeline (using for creating cutscenes).

But the biggest advantage of this toolkit are **Modules** that are distributed separately from the Game Creator. These are ready-to-use mechanics that add new features to the basic tools. Can be found next modules: Dialogue, Inventory, Quests, Stats, Behaviour, Shooter and Melee. One of the interesting modules is **Behaviour**. It helps create and organize AI systems in the node-based **Behaviour Graph** where each node is an object's state. Or **Shooter** module. It adds new animations, **Actions** and allows to create a top-down or third-person perspective shooters. **Melee** module expands Game Creator with new melee combats and new cold weapon assets.

Game creator can offer a wide selection of mechanics and systems, but some are not included in the base version. This toolkit is in the developing, so it will be evolved and supplemented. Game creator and its modules have open API that allows to extend all functionalities.

3.2 Script writing tools

From the beginning, games were created just for fun. They had no thoughtful characters or dialogues [41]. The plot was indicated by the simple goal of the game - to kill everyone, not die, collect all the parts and so on.

However, in our days it's hard to imagine a big game without thoughtful characters and plot. And some of them can contain more than a million words in the game [42]! And for writing complex dialogues are using special tools that make their creation easier.

3.2.1 Twine

Twine is an open-source tool for making interactive stories [43]. The program allows creating hypertext that leads to another hypertext. And the main feature is the ability to choose between them. The main element of the Twine editor is a **passage**, where the main elements are **text** and **link** to other passages [44]. Twine supports different types of displaying links, however, each link should be placed in square brackets and contain the name of another passage.

In Twine, you can create your own variables and write macros to work with them. This allows making the story more complex and nonlinear. It doesn't require knowledge of any programming languages for such things. However, newer version **Twine 2** supported *JavaScript* and *CSS* to expand the functionality of working with story. Another main feature of this version is that it's web-based and can be launched on different systems through a browser.

Twine offers several **story formats** through which the story will be published to a file. Each format has its own syntax and user interface but using one excludes the ability to switch to another [45]. There are 4 standard story formats in Twine 2:

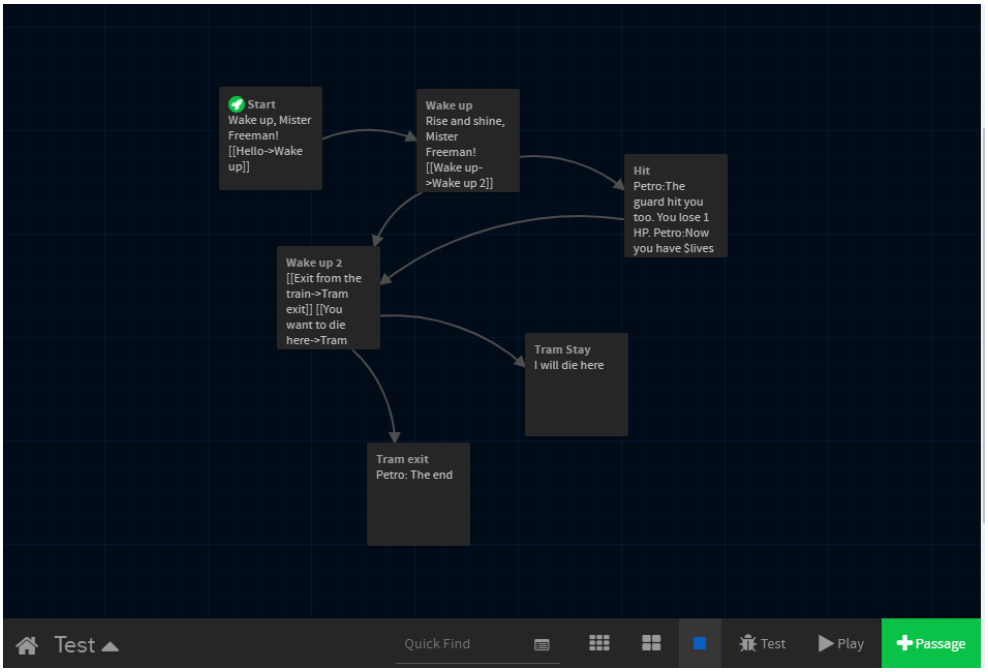

Figure 3.2: Twine Story "Test" with passages

- **Harlowe** The default format in Twine 2 that is is easy to learn.
- **Snowman** More advanced format aimed at creating story using *JavaScript* and *CSS*. It doesn't use macros and uses variables **window.story** and **window.passage**.
- **SugarCube** It comes from Twine 1. It has more functionality but also requires some programming skills.
- **Chapbook** A format that is divided into **inserts** (cause text to appear) and **modifiers** (affect story structure or passage text)

Also, to the Twine can be imported another story formats with which will be changed the resulting file [46].

3.2.2 Yarn Spinner and Editor

Yarn Spinner and Editor are tools for creating interactive dialogues in games [47]. Dialogues are written using the Yarn language. It's a simple text-based format that is designed to make dialogue in games a snap [48]. Yarn Spinner read the dialogues line by line and provides lines to the game, which decides what to do with each line. Just like Twine, Yarn Spinner allows to use **commands**,

options and **variables** to create a complex dialogues. Besides, there are also **shortcut options**, which let you add answer options for the player without creating separate dialogue.

All dialogues are stored in the Yarn Editor where they can be conveniently edited. Yarn Editor is a node-based visual tool where each dialogue is a node. Text in the dialogues is divided by characters, so at the beginning of each phrase is indicated the name of the character who pronounces it.

An important feature of this tool is the ability to integrate it with the Unity. Main parts of it are:

- **Yarn Programs** It is the compiled representation of the Yarn file.
- **Dialogue Runner** Unity component that takes Yarn Programs and read its contents.
- **Dialogue UI** Another component that deliver content from **Dialogue Runnder** to the player.
- **Variable Storage** It is responsible for storing variables of the dialogues.

Chapter 4

2D adventure game design workflow

Game development always begins with its design. You should select and accurately determine the work of game mechanics. Also, it's important to identify the tools and programs with which these mechanics will be made.

In this chapter, I would like to tell about the basic game design workflow, the main components of the 2D adventure concept game and used tools for their making.

4.1 Main concept

The main objective was to create a proof-of-concept 2D adventure game. For this purpose was used *Unity engine 2019.2*. This technology is enough to create all the basic components of an adventure game. As a graphic style have been chosen hand-drawn style. Animation of the game was created using the *EbSynth* technology in another thesis [2] and the results were used in this 2D adventure game.

The main elements of the game are dialogues, interactive story and interaction with the environment (more in section 4.2). To design simple dialogues and game plot, the *Twine 2* program was used. To transfer dialogues from Twine to Unity were developed a tool for Unity called *Twine Parser*. For manipulations with dialogues in Unity was created another tool *Dialog Editor* (more in section 4.3).

Based on these elements, was created an "escape the room" game where the player should leave the level solving the simple quest.

4.2 Game mechanics and systems

Game mechanics are an important aspect of the game. For their proper operation inside the game, were created managers and entire systems. Also, such systems can control the game logic to launch certain actions in the right time.

Dialogue system

Dialogues in the game look like comics, where a bubble with words appeared above the character's head. During player responses, a small menu appears on the screen that provides various responses options for the player. Communication between different components in the system can be seen in the image 4.1.

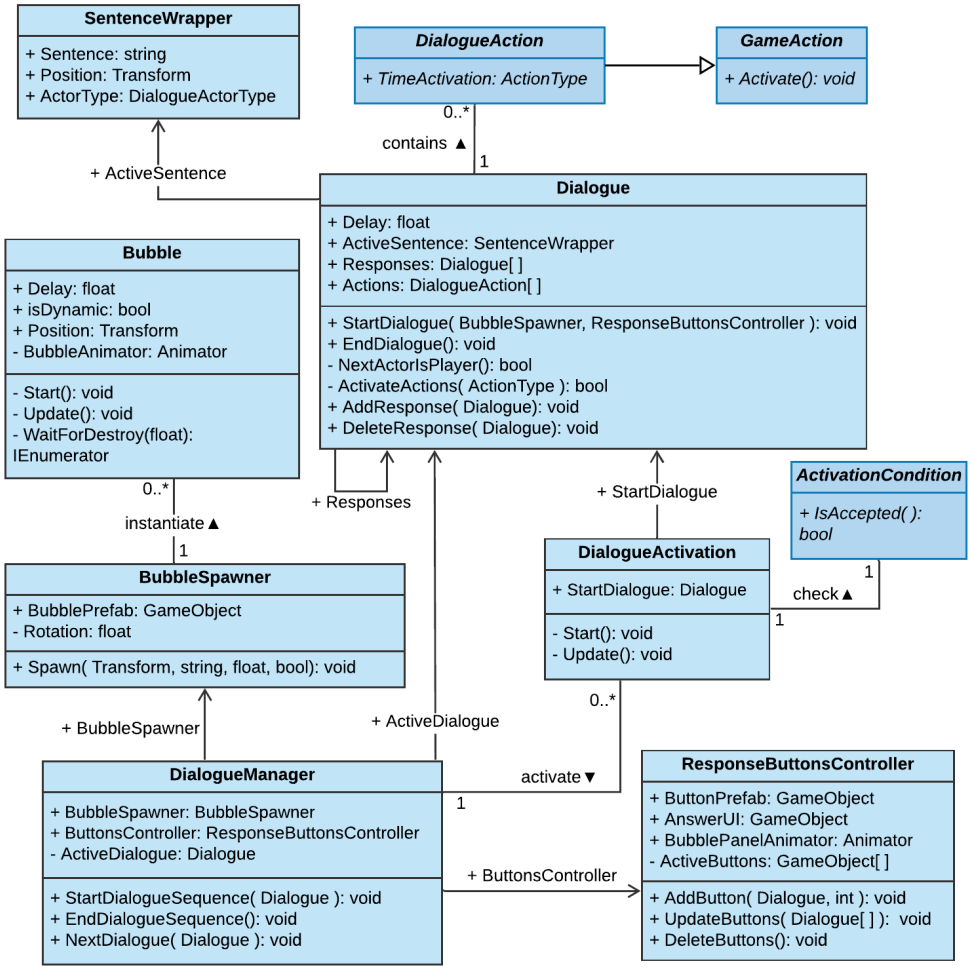

Figure 4.1: UML Class diagram of the Dialogue System

The important component is the **Dialogue Manager**, which launches the dialogue sequence and stores the object of **Bubble Spawner** and **Response Buttons Controller**. **Bubble Spawner** is responsible for the appearance of **Bubbles** with the text. **Response Buttons Controller** displays an answer menu on the screen with answer option buttons.

The **Dialogue** itself contains the necessary information in the **Sentence Wrapper**, as the text, place of appearance and type of actor (NPC or Player). Each Dialogue leads to the next Dialogue or to several Dialogues at once. Also, at the beginning or ending of the Dialogue can be activated **Actions**.

To activate any Dialogue, should be accepted **Activation Condition** of the **Dialogue Activation**. After which **Dialog Activation** will send a start Dialogue to **Dialogue Manager**.

Stage Manager

A game may have a large number of elements that must be launched in a given sequence after certain conditions are satisfied. In my concept game, it is used for loading between scenes or to control the states of sheep and called **Stage Manager**.

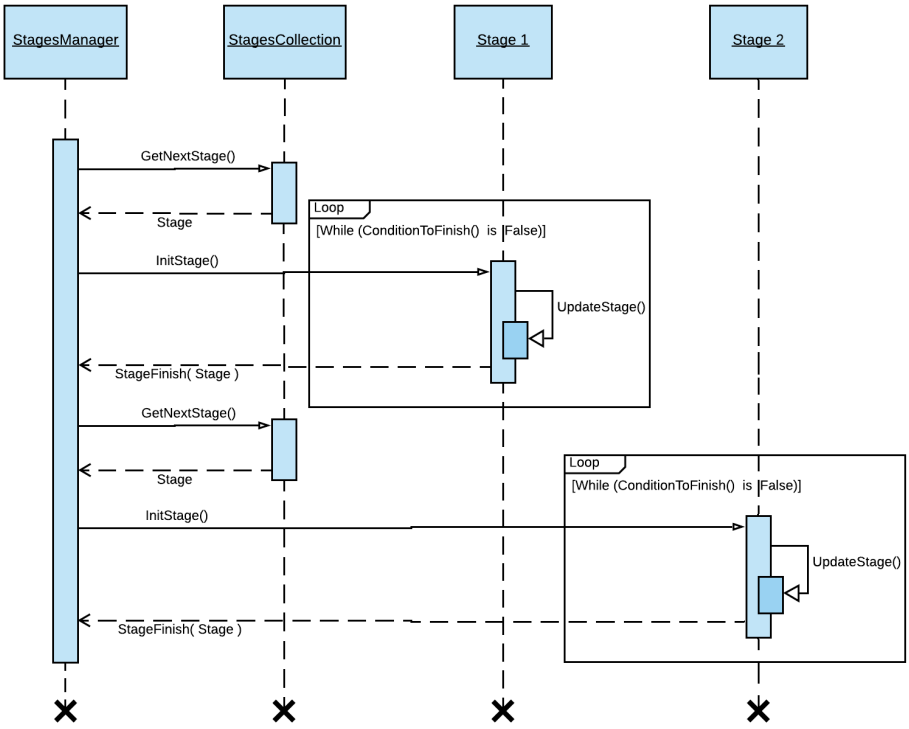

Figure 4.2: UML Sequence diagram of the Stages Manager

The system starts with the **Stages Manager**, who receives the first **Stage** from the **Stage Collection**. After that, **Stage** is initialized and each frame is updated. When the condition for the end is satisfied, Stage will return control to the Stage Manager which immediately starts the next stage. This sequence takes place until the manager reaches the last Stage (see image 4.2).

An example of such a system is **Sheep Stage Manager**. It contains two stages: **Wait** and **Walk**. The manager is supplemented with the "Repeat" function, so after the last stage is completed, the sequence will start from the beginning again.

4.3 Dialogue tools for Unity

Dialogues in games can be very long, complex and variable, with a large number of endings. And for easier creation and editing of Dialogues, for Unity was created separate tools for working with them.

Dialogue Editor

Dialog Editor is an node-based editor where every node is a **Dialog** (see image 4.3.

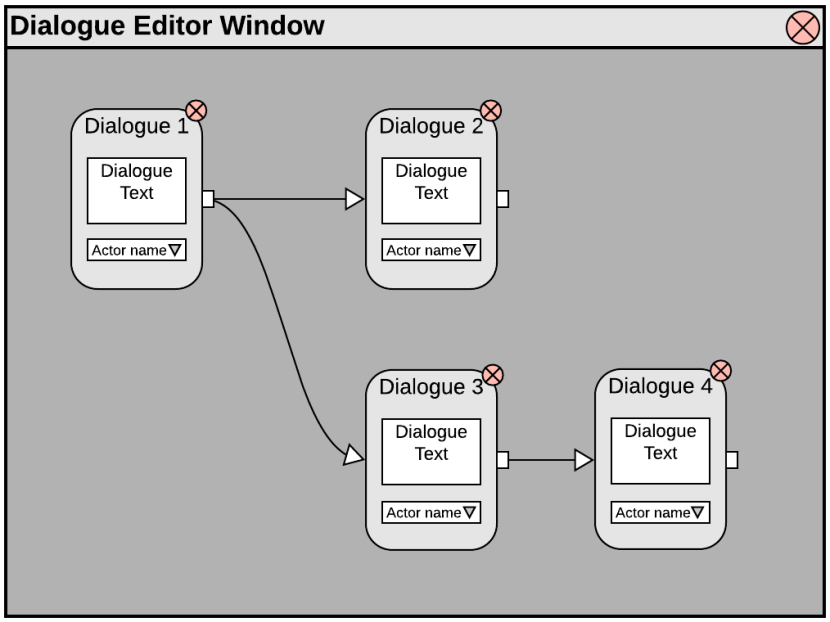

Figure 4.3: Design of the Dialogue Editor

Dialogue node has a text box and a drop-down menu with actors. In the text box will be dialogue's phrases which will pronounce the actor from the drop-down menu. Each node can be freely moved, connected with other nodes or delete unnecessary connections. All actions that can be launched in the **Dialogue Editor** window are located in the mouse context menu. The window is opened through another component called **Dialogue Handler**. This component will store a list of all active actors for these Dialogues.

Dialogue node data will not be stored as a file but will be saved directly in the scene, so after creating or editing Dialogues, they can be immediately used in the game.

From Twine to Unity

Since the Twine program is used to create dialogues, a special tool has been developed that allows transferring the Twine Story to Unity in the form of the **Dialogue System**.

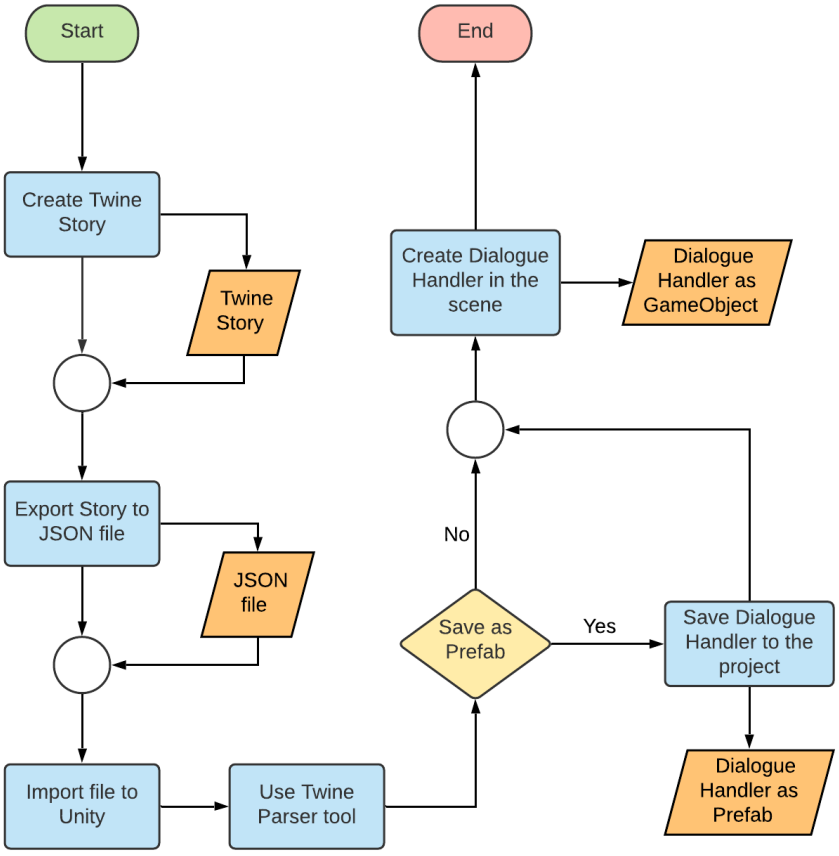

Figure 4.4: Flowchart diagram of transferring Twine Story to Unity

To do this, each passage in Twine is a text with links to the next passage. The text is divided into a phrase, where at the beginning is indicated the name of the actor who pronounces this phrase. Twine Story should be exported as *JSON* file using special Twine Story Format [49]. After adding the file to Unity, it could be parsed using the tool **Twine Dialogue Parser**. This tool will separate each phrase and link into separate Dialogues, connect them together and add a finished dialogue sequence to the scene. As a result, will be obtained the GameObject with a Dialogue Handler component in the current scene. Also, these Dialogues can be immediately saved as Prefab to the project (see image 4.4).

4.4 Gameplay

The game focused on two main mechanics: interaction with the items or objects and interactive dialogues with different NPCs.

Dialogues begin only when certain conditions are met. During dialogues, the player may be given the opportunity to select an answer or different actions may be activated. Interaction with objects works in a similar way. After successful activation, a game action sequence is launched that changes the state of the game (see image 4.5). Interactions and Dialogues can influence each other through actions and because of it, player's manipulations change a game state.

The main character have the ability to walk freely in four different directions: left, right, back and front. And can be in two different states:

1. **Idle** The player stands and no animation is played.

2. **Walking** The player walks around the scene and the movement animation is played, depending on the chosen direction of movement.

4.5 Gameflow

The game consists of one level, where the player appears immediately after launch. He sees trees, sheep and NPC around him. He can go to the NPC and activate the Dialogue. The player will receive information about the world from the NPC and the main goal – to find the Item to exit the island. After searching for it by level and finding, the hero can return to the NPC and speak with him again. The NPC will notice the player's item and offer to leave the level. The player will be able to choose whether to stay on the island or not. If the player will choose to leave the island, the game ends. The main scenario of the game can be seen in the image 4.6.

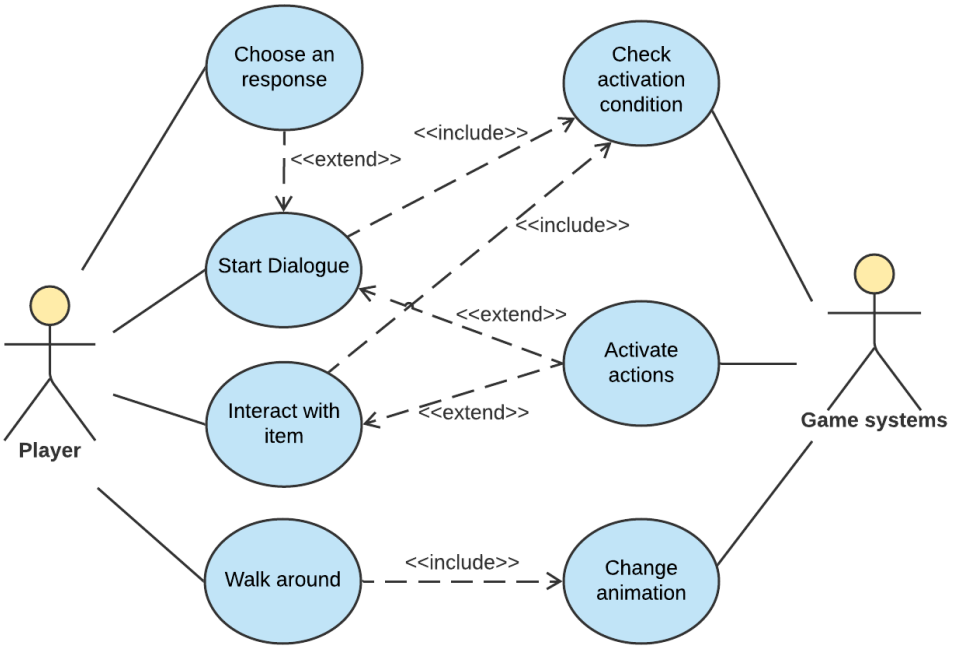

Figure 4.5: Game mechanics and reaction to them

4.6 Game objects

The following types of objects exist in the game:

1. **Environment objects** These objects fill the game scene and the player cannot interact with them (except restricting movement in the space of these objects). They can be either static (such as trees or houses), or dynamic (moving objects or characters).

2. **Interactive objects** The player can interact with these objects and will be activated certain actions. In this concept, these can be NPCs with which can be activated a dialogue, or interactive objects.

3. **Visual objects** These objects don't affect the game scene but simply complement it. They are out of the player's moving range.

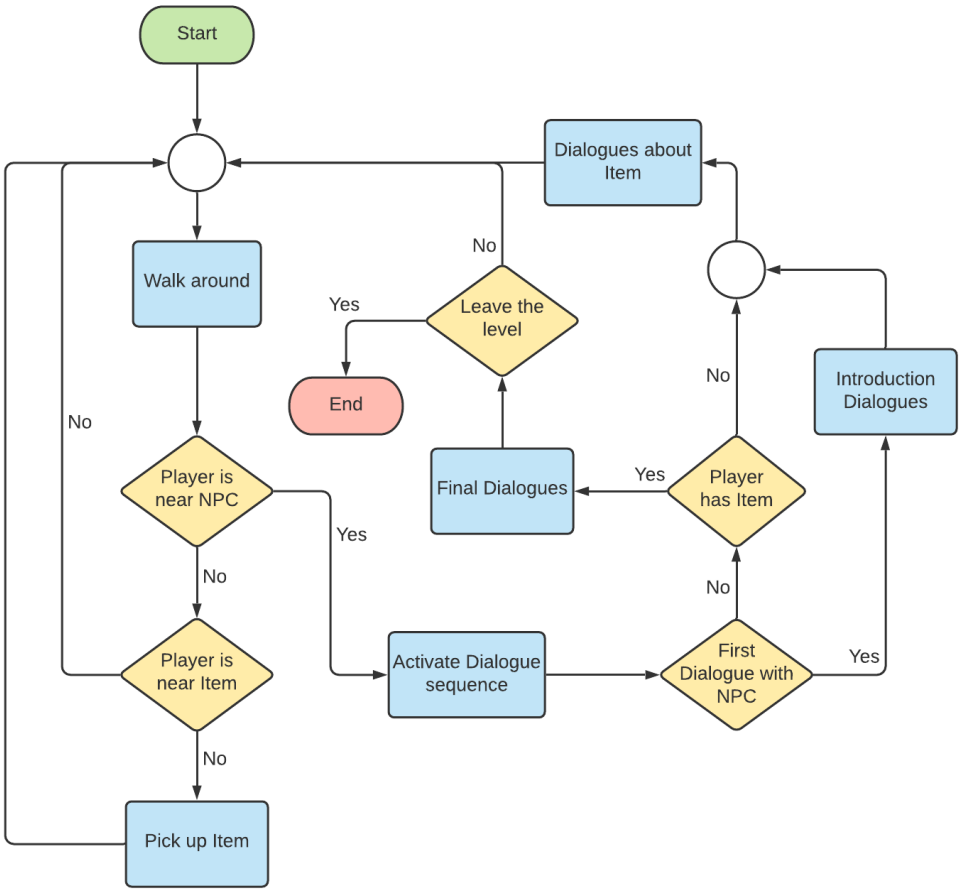

Figure 4.6: Flowchart diagram of the concept game scenario

Chapter 5

Proof of Concept

In this chapter, I would like to tell about the implementation of the Dialogue Editor and Twine Parser tools, as well as the creation of some elements of the 2D adventure concept game.

5.1 Used packages

Unity offers a large number of **Packages** that are not included in the Unity program but can be added to each project individually. Each package contains features that improve and make easier development of different aspects of the project.

Were used the following ready-made solutions:

- **Navmesh** The navigation system allows controlling a character who can find the path to a given point in the scene. Used to navigate the sheep in the scene.
- **Cinemachine** Package of modules for working with the Unity Camera.
- **TextMesh Pro** Multifunctional text solutions with lots of tools for styling and texturing text. Used for dialogue bubbles text and responses text.

5.2 Dialogue Editor

To create any window in Unity, you need to inherit the **EditorWindow** class and implement the static method with the **MenuItem** attribute (see code 5.1). Due to this attribute, the window can be opened via the context menu Window or other.

The first implementation of the node editor was class **NodeBasedEditor** based on creating **Windows** as nodes using **GUI.Window** function. The

```
public class NodeBasedEditor : EditorWindow
{
    [MenuItem("Window/Node editor")]
    static void ShowEditor()
    {
        NodeBasedEditor editor =
                EditorWindow.GetWindow<NodeBasedEditor>();
    }
}
```

Code 5.1: Example of creating Node editor Window

class has **OnGUI** method where all elements are drawn. To create a scroll view used **GUI.BeginScrollView** function. However, due to the interaction of **Windows** with scrolling view, the mouse position is incorrectly registered.

Therefore, based on the achievements of the first implementation, was created the **NodeEditor** generic class. To register various events (like a mouse click or mouse drag) is used the **HandleEvent** method. This time, the main element of the window is the **Node** class, which stores all the necessary information about the position in the window, style and various actions. The class has methods for **Moving**, **Drawing** and **Holding events**. Scrolling view was implemented using the dragging mouse where all Nodes moves in the same direction as the mouse. Also, Node is constraining the generic parameter to the NodeEditor class.

Each Node has two points with which can be connected with different Nodes. The class **NodeConnectioPoint** implements this point. Points can be of two types – **input** and **output**. And only two points of different types can be connected. As soon as a point-button is pressed, an action **OnClickAction** is called.

For the connection between two points is responsible **NodeConnection** class which contains information about two interconnected NodeConnectioPoints. All connections are stored in the NodeEditor. To draw the connection between these points was used the function **Handles.DrawBezier**.

NodeEditor's method **OnGUI** calls only one time per event. Because of it all elements of the window don't update correctly and occurs "freezing" effect. To solve this problem was used the function **Repaint**. Each time when the user manipulates with the window, this function will update it.

To create the **Dialogue Editor** were implemented classes **NodeEditorDialogue** and **NodeDialogue** that inherit the classes **NodeEditor** and **Node** respectively.

NodeDialogue has the Dialogue property to which it refers. A field of text was implemented using **GUI.TextArea** function and popup menu with

current Actors using **EditorGUI.Popup** function. All changes to the node are immediately saved in the Dialogue.

After closing the window, the object is immediately deleted and all information is lost with it. To avoid this, were added **DialogueEditorData** and **DialoguesHandler** classes.

DialogueEditorData stores information about nodes and the connections between them. Since the base classes used with the NodeEditor do not save data after closing the window, were created new serializable classes **Node** and **Connection**. When the window is opened, all information is read and added to the editor window. And when the window closes - all relevant information is overwritten to the DialoguesHandler object.

```
public abstract class DefaultEditor<T> : Editor
        where T : MonoBehaviour
{
    bool showDefaultInspector = false;

    private void OnEnable()
    {
        OnCustomEnable();
    }

    public override void OnInspectorGUI()
    {
        showDefaultInspector = EditorGUILayout.Toggle(
                "Show the default editor",
                showDefaultInspector);

        if (showDefaultInspector)
            base.OnInspectorGUI();
        else
            OnCustomInspectorGUI();
    }

    public abstract void OnCustomInspectorGUI();

    public abstract void OnCustomEnable();
}
```

Code 5.2: Default Editor class

DialoguesHandler class stores DialogueEditorData object and has **List** of all active **Actors**. Actor has **Name**, **Actor Type** and **Object Transform** where will be spawning Bubbles. Also, for DialoguesHandler was created

Custom Editor which replaces the default layout. **DialoguesHandlerEditor** inherits **DefaultEditor** abstract class (see code 5.2) and adds new features for DialoguesHandler: opening the Dialogue Editor window as well as adding or removing new Actors. When this component is enabled, it checks if the dialogue sequence already exists for the active GameObject and loads data about Dialogues.

If the project is restarted, then all data will be lost without any warnings. Therefore, during every user action with the Dialogue Editor are called functions **EditorUtility.SetDirty** and **EditorSceneManager.MarkSceneDirty** that mark the scene as unsaved.

For the appearance of Dialogues in the game, should be added **DialogueActivation** component to the NPC and chosen required DialogueHandler and **Activation Condition**.

5.3 Twine Dialogue Parser

After creating a new Twine Story and saving it to the *JSON* format, it should be imported to the Unity. The class **TwineParser** is responsible for translating this file into DialoguesHandler. TwineParser creates a new window "Twine Parser Dialogue" where should be chosen new Dialogue GameObject name and required *JSON* file.

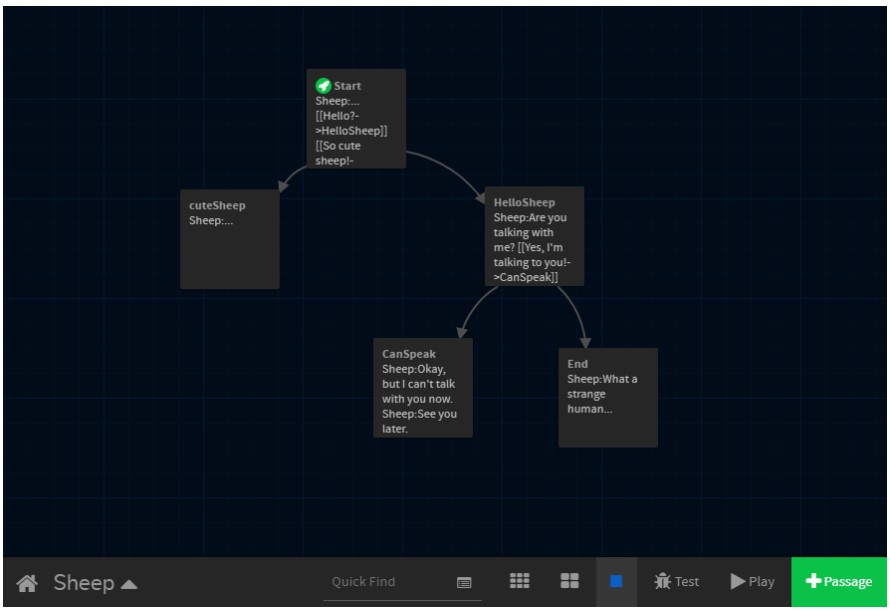

Figure 5.1: Created Twine Story called Sheep

As soon as the "Complete" button is pressed, will be created an object **TwineDialogue** from its *JSON* representation. After that, based on each

Passage, will be created a separate Dialogue GameObject with the text from the Passage. Then the text is divided into more Dialogues using class **DialogueTextParser** which separates all links and phrases. Regular expressions were used to identify key words in the text. **DialogueTextParser** returns **ParsedDialogue** object which has two **List** properties: **NPCText** and **PlayerText**. And based on this data, are created additional Dialogues.

The main idea is that the Passages always have a similar text structure, where NPC phrases go first, and then player's answers in the form of links to other Passages.

In the end, should be properly configured the actors in the DialogueHandler and positions for the Bubbles to appear. And Dialogues will be ready to use in the game.

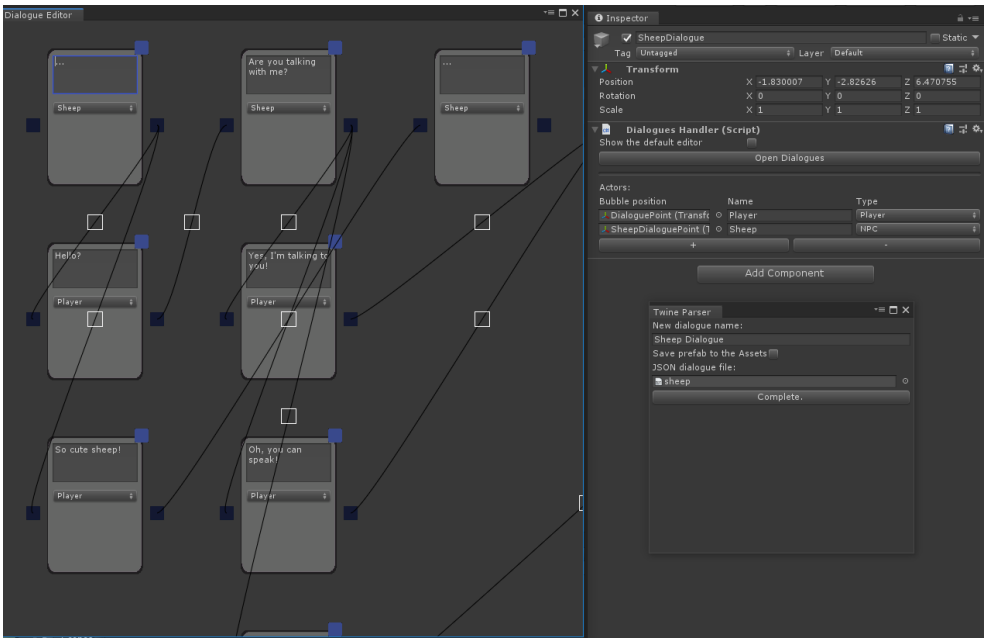

Figure 5.2: Result of the parsing *JSON* file

5.4 EbSynth animation

The **PlayerMovement** class is responsible for the player's movement. Movement occurs using the Rigidbody component. Based on the pressed keys, it calculates the **Direction** of movement and transfers it to the **MovePosition** function. Since the movement is using physics, it is constantly called from the **LateUpdate** method to avoid incorrect behaviour. Also, instead of LateUpdate, could be used **FixedUpdate**.

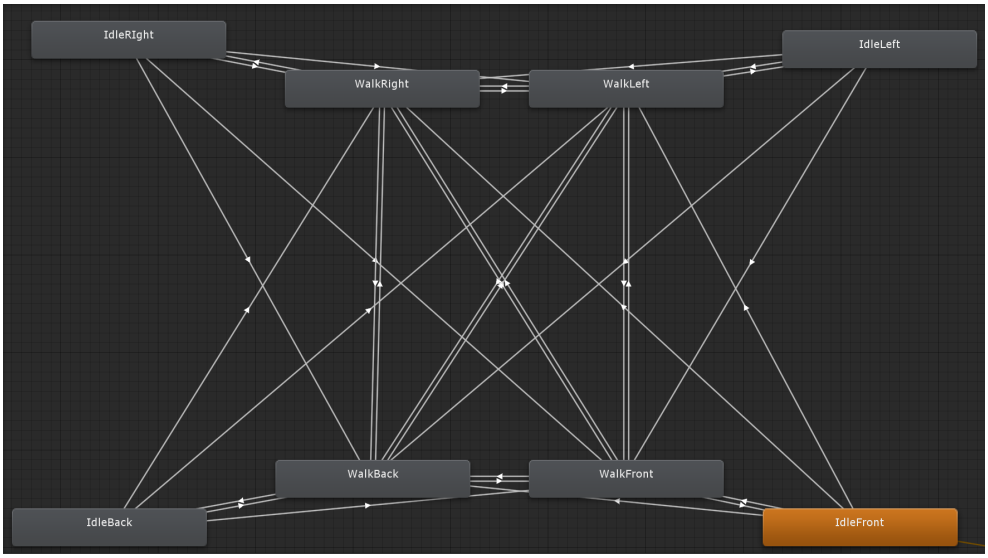

Figure 5.3: Animation states of the Player

The **PlayerAnimationManager** class controls the animation of the player based on its movement. The player has four directions for animation: left, right, back and forward. And as soon as the player stops moving, the Idle animation will be played based on the previous animation. It is implemented using the **Animator** (see image 5.3). Each animation state is a sequence of images that were previously made using **EbSynth**.

Movement and animation are launched through the singleton class **PlayerManager** .

5.5 Sorting image layer

In the scene, all game objects are Sprites that are rotated 45 degrees to the camera. Each sprite has a **rendering order** in a particular type of layer. Since some objects are dynamic, their rendering order changes. For such purposes was created the class **SortingLayerController**.

Each GameObject that has **DynamicSpriteSortLayer** and **SpriteRenderer** components is added to the **List** of the SortingLayerController. And every 0.01 seconds, this controller changes their rendering order based on the invert value Z of the position. Static objects must be configured in the same way but rendering order for them is static. To do this, **StaticSpriteSortLayer** component is added to environment objects.

5.6 Testing

Were created automated unit tests to make sure that the game works properly. I have created unit tests that check individual parts of the code that run during the game. Other tests check the certain actions that a player can perform during the game. All tests work during the *PlayMode*.

To verify that the game can be completed were created unit test **CompleteLevelSuit**. The goal of the game is to find the item and give it to the NPC. Therefore, the unit test checks if the item exists in the scene using `"Assert.NotNull(apple)"`. Then the test activates item actions and begins required dialogue with the NPC. When the player has a choice of answer, a positive answer is selected and the final scene is loaded. In the end, it checks if the correct scene was loaded using code 5.3.

```
Assert.AreEqual(SceneManager.GetSceneByName("EndScene"),
                SceneManager.GetActiveScene());
```

Code 5.3: ”The correct level is loaded” assertion

Conclusion

The main objective of this thesis was to design and create an open-source proof-of-concept 2D adventure game. The game was developed in the Unity game engine and for creating the animation was used the program EbSynth.

I have researched all the basic components of the Unity engine. Therefore, I quickly resolved problems during the implementation of the game components. I've also explored various artistic style transfer techniques and programs for image synthesis, especially EbSynth. I reviewed various storytelling and game script creating tools with which can be facilitated the game development.

Based on the selected technologies were designed and implemented major game mechanics of the 2D adventure game: Dialogue System and Interaction with the environment. For easier and more clear work with the Dialogue System, were developed practical tools: Dialogue Editor and Twine Parser. Dialogue Editor allows changing dialogues in the game quickly and comfortably. Twine Parser gives the opportunity to transfer dialogues from the Twine program to Unity in Dialogue System format.

In the end, based on the mechanics, I created the concept of a simple 2D adventure game. And in this game, I managed to demonstrate all advantages of EbSynth technology with which was made wonderful and unique animations. This game, along with created tools and source code, can be found in the online repository [50].

The purpose of this project was to create a concept game. However, based on the developed mechanics and tools, it will be possible to develop a full-fledged 2D game. To do this, should be written a game plot, added additional components for working with sound and music. Dialogue Editor needs to be supplemented with new elements, like Actions, so all manipulations with dialogues would be possible through the Dialogue Editor only. Twine Parser should be expanded with other Twine commands.

Bibliography

1. *THE SHEEPLESS ONE* [online]. 2019 [visited on 2020-05-30]. Available from: `http://sheepless.one/`.
2. KLICPERA, Jan. *Sheepless – An Open-source 2D AdventureGame in Unity*. 2020. Bachelor's Thesis. Czech technical university in Prague. Supervisor: Ing. Marek Skotnica.
3. BADRONOV, Robert. *Sheepless – An Open-source 2D AdventureGame in Unity*. 2020. Bachelor's Thesis. Czech technical university in Prague. Supervisor: Ing. Marek Skotnica.
4. STEWART, Samuel. Video game industry silently taking over entertainment world. *EJINSIGHT* [online]. 2019 [visited on 2020-03-17]. Available from: `http://www.ejinsight.com/20191022-video-game-industry-silently-taking-over-entertainment-world/`.
5. OPPENHEIMERFUNDS. Investing in the Soaring Popularity of Gaming. *Reuters* [online]. 2018 [visited on 2020-03-17]. Available from: `https://www.reuters.com/sponsored/article/popularity-of-gaming?utm_source=reddit.com`.
6. NAKAMURA, Yuji. Peak Video Game? Top Analyst Sees Industry Slumping in 2019. *Bloomberg* [online]. 2019 [visited on 2020-03-17]. Available from: `https://www.bloomberg.com/news/articles/2019-01-23/peak-video-game-top-analyst-sees-industry-slumping-in-2019`.
7. *Video Games – Thematic Research* [online]. GlobalData UK Ltd., 2019. Available also from: `https://store.globaldata.com/report/gdtmt-tr-s212--video-games-thematic-research/`. Technical report.
8. TOFTEDAHL, Marcus. Which are the most commonly used Game Engines? *Gamasutra* [online]. 2019 [visited on 2020-03-17]. Available from: `https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/Which_are_the_most_commonly_used_Game_Engines.php`.

9. ADAMS, Ernest. *Fundamentals of Adventure Game Design*. Peachpit Press, 2014. ISBN 9780133812329.

10. *Plans and pricing* [online]. Unity Technologies, 2020 [visited on 2020-04-30]. Available from: `https://store.unity.com/#plans-individual`.

11. WONG, Nichole. Introducing the Unity Student plan: Start creating like a pro. *Unity Technologies Blog* [online]. 2020 [visited on 2020-03-17]. Available from: `https://blogs.unity3d.com/ru/2020/02/25/introducing-the-unity-student-plan-start-creating-like-a-pro/`.

12. *Unity User Manual* [online]. Unity Technologies, 2020 [visited on 2020-04-30]. Available from: `https://docs.unity3d.com/2019.2/Documentation/Manual/index.html`.

13. *Learning C# and coding in Unity for beginners* [online]. Unity Technologies, 2020 [visited on 2020-04-30]. Available from: `https://unity3d.com/learning-c-sharp-in-unity-for-beginners`.

14. *About the Universal Render Pipeline: Universal RP: 7.1.8* [online]. Unity Technologies, 2020 [visited on 2020-04-30]. Available from: `https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@7.1/manual/index.html`.

15. NORTON, Terry. *Learning C# by developing games with Unity 3D beginner's guide : learn the fundamentals of C# to create scripts for your GameObjects*. Birmingham: Packt Publishing, 2013. ISBN 9781849696586.

16. GEIG, Mike. *Sams teach yourself Unity 2018 game development in 24 hours*. 3rd ed. Indianapolis, Indiana: Sams, 2018. ISBN 978-0-13-499813-8.

17. LINTRAMI, Tommaso. *Unity 2017 Game Development Essentials*. 3rd ed. Birmingham: Packt Publishing, 2018. ISBN 978-1-78646-939-7.

18. JACKSON, M. A. *Principles of program design*. London New York: Academic Press, 1975. ISBN 978-0123790507.

19. CHONG, Andrew. *Digital animation*. Lausanne, Switzerland New York, N.Y: AVA Academia Distributed in the USA & Canada by Watson-Guptill Publications, 2008. ISBN 2-940373-56-6.

20. LAYBOURNE, Kit. *The animation book : a complete guide to animated filmmaking–from flip-books to sound cartoons to 3-D animation*. New York: Three Rivers Press, 1998. ISBN 0517886022.

21. CULHANE, Shamus. *Animation from script to screen*. New York: St. Martin's Press, 1988. ISBN 978-0-312-05052-8.

22. BRET, Michel. *Image Synthesis*. Dordrecht: Springer Netherlands, 1992. ISBN 978-94-010-5133-0.

23. HEMANTH, D. *Deep learning for image processing applications*. Amsterdam, Netherlands: IOS Press, 2017. ISBN 978-1-61499-821-1.

24. WANI, M. A. *Advances in deep learning*. Singapore: Springer, 2020. ISBN 978-981-13-6793-9.

25. ALEXANDER MORDVINTSEV, Christopher Olah; TYKA, Mike. DeepDream - a code example for visualizing Neural Networks. *Google Research Blog* [online]. 2015 [visited on 2020-05-14]. Available from: `https://web.archive.org/web/20150708233542/http://googleresearch.blogspot.co.uk/2015/07/deepdream-code-example-for-visualizing.html`.

26. CHANDLER, Nathan. How Google Deep Dream Works. *HowStuffWorks* [online]. 2015 [visited on 2020-05-14]. Available from: `https://computer.howstuffworks.com/google-deep-dream.html`.

27. GOOGLE. *deepdream* [online]. GitHub, 2015 [visited on 2020-06-04]. Available from: `https://github.com/google/deepdream`.

28. *Deep Dream Generator* [online]. Deep Dream Generator, 2020 [visited on 2020-05-31]. Available from: `https://deepdreamgenerator.com/`.

29. KARRAS, Tero; LAINE, Samuli; AILA, Timo. A Style-Based Generator Architecture for Generative Adversarial Networks. *CoRR* [online]. 2018 [visited on 2020-05-16]. Available from: `http://arxiv.org/abs/1812.04948`.

30. AHIRWAR, Kailash. *Generative adversarial networks projects : build next-generation generative models using TensorFlow and Keras*. Birmingham, UK: Packt Publishing, 2019. ISBN 978-1-78913-667-8.

31. YIN, Kayo. How to Train StyleGAN to Generate Realistic Faces. *Towards Data Science* [online]. 2019 [visited on 2020-05-16]. Available from: `https://towardsdatascience.com/how-to-train-stylegan-to-generate-realistic-faces-d4afca48e705`.

32. *thispersondoesnotexist.com* [online]. Philip Wang, 2019 [visited on 2020-05-16]. Available from: `http://thispersondoesnotexist.com`.

33. *thisartworkdoesnotexist.com* [online]. Michael Friesen, 2019 [visited on 2020-05-16]. Available from: `https://thisartworkdoesnotexist.com`.

34. FIŠER, Jakub; JAMRIŠKA, Ondřej; LUKÁČ, Michal; SHECHTMAN, Eli; ASENTE, Paul; LU, Jingwan; SÝKORA, Daniel. StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2016)*. 2016, vol. 35, no. 4, pp. 92.

35. JAMRIŠKA, Ondřej; SOCHOROVÁ, Šárka; TEXLER, Ondřej; LUKÁČ, Michal; FIŠER, Jakub; LU, Jingwan; SHECHTMAN, Eli; SÝKORA, Daniel. Stylizing Video by Example. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2019)*. 2019, vol. 38, no. 4, pp. 107.

36. NEMIRC. Turn your footage into digital paintings with EbSynth. *Renderosity Magazine* [online]. 2019 [visited on 2020-05-17]. Available from: `https://magazine.renderosity.com/article/5386/turn-your-footage-into-digital-paintings-with-ebsynth`.

37. SCHREIER, Jason. *Blood, sweat, and pixels : the triumphant, turbulent stories behind how video games are made*. New York: Harper, 2017. ISBN 978-0062651235.

38. *Fungus* [online]. Fungus, 2020 [visited on 2020-06-04]. Available from: `https://fungusgames.com/`.

39. *Adventure Creator* [online]. ICEBOX Studios, 2020 [visited on 2020-06-04]. Available from: `https://www.adventurecreator.org`.

40. *Game Creator* [online]. Catsoft Studios, 2020 [visited on 2020-06-04]. Available from: `https://docs.gamecreator.io/`.

41. MILDORF, Jarmila. *Dialogue across media*. Amsterdam Philadelphia: John Benjamins Publishing Company, 2017. ISBN 978-90-272-1045-6.

42. WRIGHT, Landon. Disco Elysium's Script Is Over A Million Words Long. *GamingBolt* [online]. 2020 [visited on 2020-05-17]. Available from: `https://gamingbolt.com/disco-elysiums-script-is-over-a-million-words-long`.

43. *Twine is an open-source tool* [online]. Interactive Fiction Technology Foundation, 2020 [visited on 2020-06-04]. Available from: `https://twinery.org/`.

44. SMALLWOOD, Carol. *Teaching technology in libraries : creative ideas for training staff, patrons and students*. Jefferson, North Carolina: McFarland & Company, Inc., Publishers, 2017. ISBN 978-1-4766-6474-3.

45. MAYER, Brian. *Create iInteractive stories in Twine*. New York: Rosen YA, Rosen Publishing Gorup, 2020. ISBN 9781725340183.

46. DEMARCO, M.C. *A Catalog of Twine Story Formats* [online]. 2019 [visited on 2020-05-18]. Available from: `http://mcdemarco.net/tools/hyperfic/twine/catalog/`.

47. *Yarn Spinner tool* [online]. Secret Lab and Yarn Spinner Contributors, 2020 [visited on 2020-06-04]. Available from: `https://yarnspinner.dev/`.

48. *Yarn Spinner* [online]. Secret Lab, 2016 [visited on 2020-05-18]. Available from: `https://www.secretlab.com.au/blog/2016/2/10/yarn-spinner`.

49. AZERWALKER. *Twison* [online]. GitHub, 2017 [visited on 2020-05-27]. Available from: `https://github.com/lazerwalker/twison`.

50. MUSTIATS, Ian. *Adventure concept game* [online]. GitHub, 2020 [visited on 2020-06-04]. Available from: `https://github.com/mustiian/Adventure-concept-game`.

Appendix A

Acronyms

3D Three dimensional

2D Two dimensional

AI Artificial intelligence

API Application Programming Interface

JSON JavaScript Object Notation

GUI Graphical user interface

NPC Non-playable character

UI User interface

Appendix B

Contents of enclosed USB

- `readme.txt` ..... the file with USB contents description
- `game` ..... the directory with the Unity game build
- `src` ..... the directory of source codes
  - `unity` ..... the directory with Unity implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format