FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Sheepless - An Open-source 2D Adventure Game in Unity
Student:	Jan Klicpera
Supervisor:	Ing. Marek Skotnica
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter 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.

References

Will be provided by the supervisor.

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

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

Prague September 19, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

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

Jan Klicpera

Department of Software Engineering
Supervisor: Ing. Marek Skotnica

May 20, 2020

Acknowledgements

I would like to thank my supervisor, Ing. Marek Skotnica, for his patience, guidance and persistent positive attitude. I would also like to thank Poli Akhmetzhanova for her beautiful artwork, which was used throughout the project. I am also grateful to the creators of *EbSynth* from DCGI at FEL CTU for publicly sharing their great results and also answering all my questions regarding it. My thanks also goes to everyone who has been working on the *Sheepless* project. Last, but not least, my deep gratitude goes towards my family and my partner Káťa for their endless support and encouragement.

Declaration

In Prague on May 20, 2020 …………………

Citation of this thesis

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

Abstract

This thesis is dedicated to exploring the possibilities of creating a video game in the *Unity engine* with a heavy emphasis on utilizing artistic style transfer technology for all in-game 2D animations. A method to record and process live-action footage is described and custom tools in the *Blender* editor, to more effectively process the footage, are proposed and created. The processed footage is then stylized using the *EbSynth* algorithm. A movement and animation system is created for the *Unity game engine*, utilizing the stylized animations. The game components are then demonstrated in a proof-of-concept implementation. The solution utilizes free software only and is fully available in an online repository.

Keywords 2D animation, Blender, EbSynth, game development, style transfer, Unity engine

Abstrakt

Tato práce prozkoumává možnosti využití moderní technologie pro převod uměleckého stylu v herním průmyslu. Zvolená technologie pro převod stylu, *EbSynth*, je využita pro efektivní uměleckou stylizaci živě natočených záběrů. V práci je nejprve popsán postup natočení, zpracování a stylizace záběrů. V editačním programu *Blender* jsou vytvořeny nástroje, které částečně automatizují proces zpracování záběrů. V herním engine *Unity* je následně vytvořen animační a pohybový systém, využívající tyto stylizované animace. Vytvořené herní komponenty jsou poté demonstrovány na jednoduché ukázce herního světa. Veškerá výsledná řešení využívají výhradně bezplatný software a jsou plně dostupná v online repozitáři.

Klíčová slova 2D animace, Blender, EbSynth, herní vývoj, převod stylu, Unity engine

Contents

List of Figures

Introduction

Until quite recently, creating authentic animations in a specific art style was possible only by painting every frame manually, which is very time consuming and costly. According to the CBS news [1], a full feature hand-painted film *Loving Vincent* took a team of 120 artists over 4 years to complete. The movie consists of 64,000 frames, each one painted by hand. In the interview, the director of the film, Hugh Welchman, has stated: "*computers could never replicate this kind of authenticity*". While that statement might stand true for the foreseeable future, the image synthesis research field is evolving at an incredible pace. It is already possible to greatly expedite the animation process, whilst preserving a large degree of authenticity. Using *EbSynth*, an artistic style transfer software, the number of frames needed to be drawn by hand can be significantly reduced – description of *EbSynth* and other approaches to artistic style transfer can be found in chapter 1.

The aim of this thesis is to utilize *EbSynth* in game development using the *Unity game engine*. That includes recording the live-action footage and creating tools to efficiently process the footage in bulk. The next step is to authentically control and display the stylized animations in the game engine. These approaches and tools are then demonstrated in a proof-of-concept *Unity* project. The main goal is to provide a general solution that is both efficient and utilizes free software, not to develop a fully fleshed-out playable game.

The first three chapters introduce the reader to the software and tools used in the project. Chapter 1 explores the current state-of-the-art style transfer techniques, which could to some degree be utilized in game development. In chapter 2, the tool used for processing the footage, *Blender*, is introduced. The *Unity game engine* is presented in chapter 3. In chapter 4, the process of recording and transforming the footage using the available tools is analyzed and new tools to aid in the process are proposed. The method chosen to import the processed footage is also described. Lastly, the movement and animation systems in *Unity* are designed. In chapter 5, the features and tools, implemented based on the analysis, are presented.

The outcome of the thesis (or parts of it) will be utilized during the development of a video game called *Sheepless*. Two of my colleagues, Ian Mustiats and Robert Badronov, have also been developing other features for the game in their theses [2, 3].

Chapter 1

Artistic Style Transfer

Style transfer algorithms are used to synthesize artistically stylized images based on a set of input parameters. These input parameters vary based on the approach of the specific implementation, but they generally consist of a *source style* (the style example) and a *source content* (the image the style of the exemplar gets applied to). In this chapter, certain types of state-of-the-art style transfer algorithms, that could potentially be useful in game development, are explored.

1.1 StyLit

StyLit is an approach to example-based stylization of 3D renderings introduced in [4]. It focuses on preserving the rich expressiveness of hand-created artwork, achieving this by being able to distinguish among context-dependent illumination effects, rather than being guided only by colours and normals. The algorithm also tackles disruptive artifacts that have been common in the results of the previous approaches to guided texture synthesis.

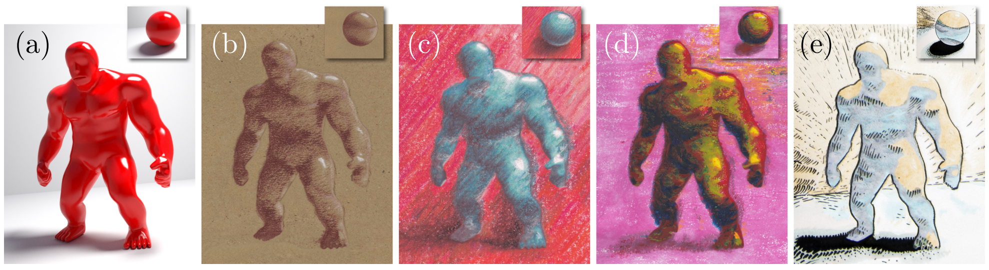


Figure 1.1: An example of images synthesized using *StyLit* [4]. Exemplar images: © Daichi Ito (b), Pavla Sýkorová (c, d), and Lukáš Vlček (e).

The most common stylization workflow is to first create a simple 3D scene, containing all important illumination effects. Typically, “a sphere on a table” (see top insets in figure 1.1) is used as the reference scene. The scene is then rendered and can be painted over by the artist in the desired style, either digitally or physically by printing it onto a paper with special alignment marks. The algorithm is then able to transfer the style from the exemplar image onto a different rendered scene. If an exemplar image already exists, but the reference 3D model does not, an approximate 3D reconstruction can be created to roughly resemble the shape and illumination effects of the original image.

StyLit can also be used to create animations or to auto-complete an image. The user can stylize only a portion of the model and then use the algorithm to transfer the style to the rest of the model.

While technically this specific implementation may no longer be considered as state-of-the-art, it is an important milestone in example-based style transfer, providing a foundation for the approach described in the next section.

1.2 StyleBlit

StyleBlit, introduced in [5], is an example-based style algorithm, which is able to produce results similar to the previously described *StyLit* (see figure 1.2 for a comparison), whilst being able to produce the result orders of magnitude faster.

The algorithm allows to stylize a one-megapixel image at 10 frames per second on a single-core CPU (Core i7, 2.8GHz), while also being able to achieve more than 100 frames per second at 4K resolution on a GPU (GeForce GTX 970). That creates a possibility to render stylized images in real-time, which can be directly applied in game development.

Such approach has been explored in [6], with interesting results. A plugin for the *Unity game engine* has been created, directly utilizing a modified version of the *StyleBlit* algorithm. The plugin is capable of creating fully stylized scenes, when provided with a sample texture of the desired style. Complex models can also be seamlessly stylized using multiple sample textures. One of the visual limitations is stylizing flat monotonous surfaces – the style texture is applied in the form of small patches, creating a visibly repeating pattern map. Due to the form of style input, the artist is also unable to precisely control the stylization of more detailed parts of a model, as they can only control the overall texture of the model.

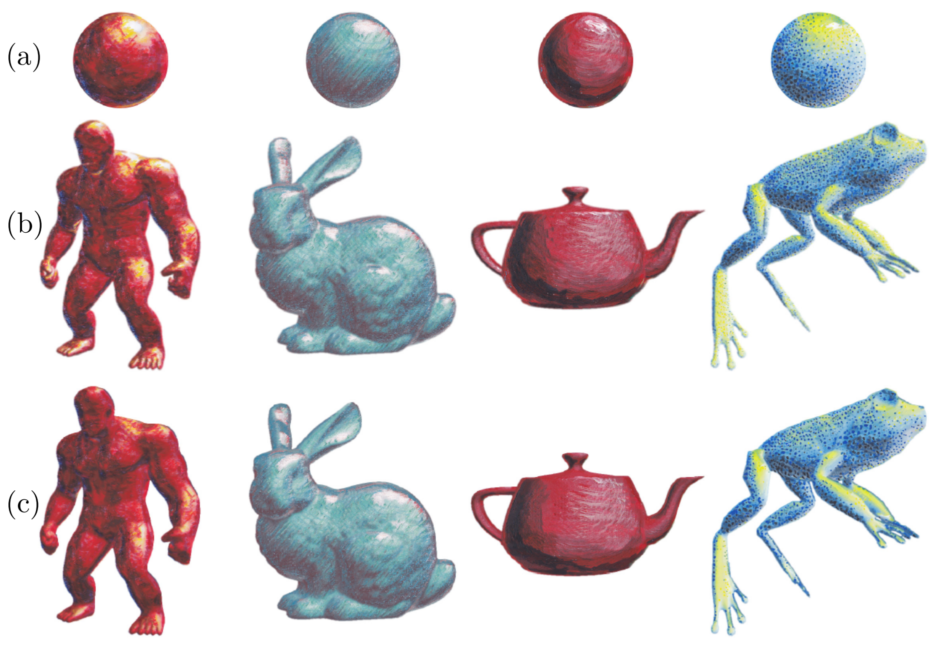


Figure 1.2: A comparison between *StyLit* and *StyleBlit*: original style exemplar (a), the result of *StyleBlit* (b), and the result of *StyLit* (c) [5]. Style exemplars: © Pavla Sýkorová.

1.3 EbSynth

EbSynth, as proposed in [7], is a powerful video synthesizer that is able to apply an artistic style to a video sequence based on one or more keyframes, which serve as a stylistic example for the synthesis. Because these keyframes are a direct stylization of the source content, the artist can very precisely control the overall stylized look and details of the synthesized video.

The synthesis is especially useful for stylizing live-action sequences, as no tool that would provide such wide artistic control over the resulting video previously existed. Using *EbSynth*, it is now possible to create smooth stylized 2D animations utilizing live-action footage, without the need to draw every frame by hand, which was the most time consuming and expensive part of the process.

1.3.1 Working With EbSynth

In its current version (Alpha as of writing this thesis), *EbSynth* allows to synthesize images using a free GUI application for the *Windows* operating system (see figure 1.3 for a screenshot of the GUI). The only mandatory inputs are the original image sequence and at least one keyframe, based on which the synthesis operates. The name of the keyframe must be named to numerically match with the corresponding original frame and, for the best result, should structurally match the original video frame.

One keyframe is usually sufficient for simple shots, but if new information is introduced into the scene, for example by out-of-plane rotation, multiple keyframes are necessary. Additional keyframes might also be needed, when sudden illumination changes happen in the scene, as the steep colour shift might mislead the synthesis.

It is also possible to supply a mask, which defines the area to synthesize and must be defined for every single input frame. The mask must be in a standard black and white format (see figure 1.4 b).

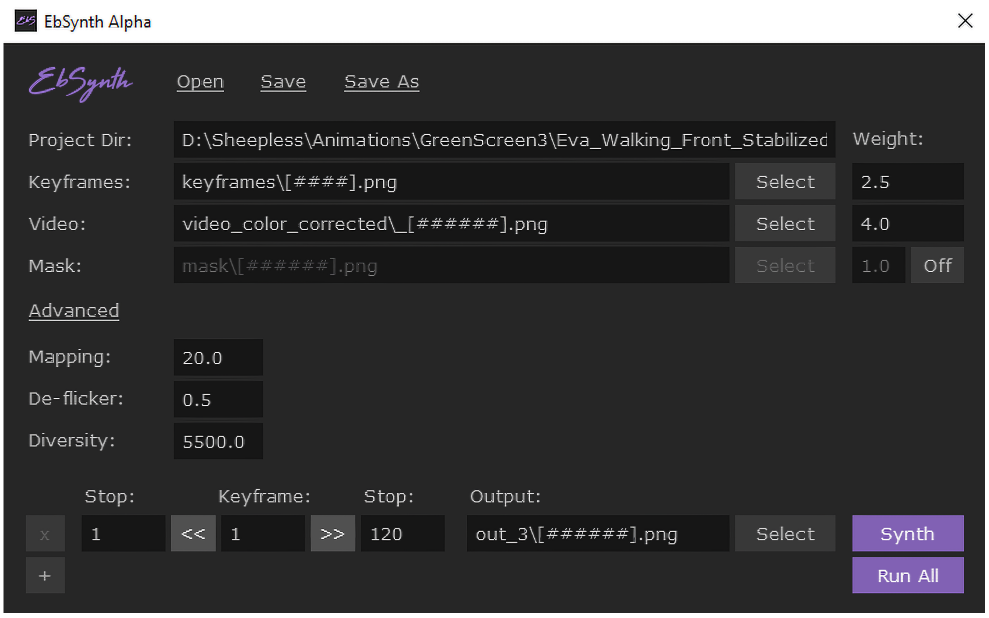
Finally, the user can alter the following settings:

Keyframe and video weight The ratio of these two settings determines how similar the final image will be to each respective input. The higher the keyframe weight is, the more stylized the output will be. Intuitively, going the other way will produce images more similar to the original sequence.

Mapping If set to a low value, the synthesis can exchange sets of pixels between regions. Whilst useful at times, it can also lead to some unwanted and unpredictable behaviour.

De-flicker Determines how similar a newly generated frame should be to the previous one. If set to zero, each new frame is synthesized independently to the previous ones and the final animation can flicker.

Diversity Controls how much diversity of the reference keyframe the synthesis tries to transfer. The higher the value is, the more the synthesis process tries to not highly reuse small portion of the reference, which would create large monolithic blurry areas.



Project Dir: D:\Sheepless\Animations\GreenScreen3\Eva_Walking_Front_Stabilized
Weight:
Keyframes: keyframes\[####].png 2.5
Video: video_color_corrected_[######].png 4.0
Mask: mask\[######].png 1.0 Off

Mapping: 20.0
De-flicker: 0.5
Diversity: 5500.0

Stop: 1
Keyframe: 1
Stop: 120
Output: out_3\[######].png

Figure 1.3: *EbSynth* Alpha for *Windows* [8].

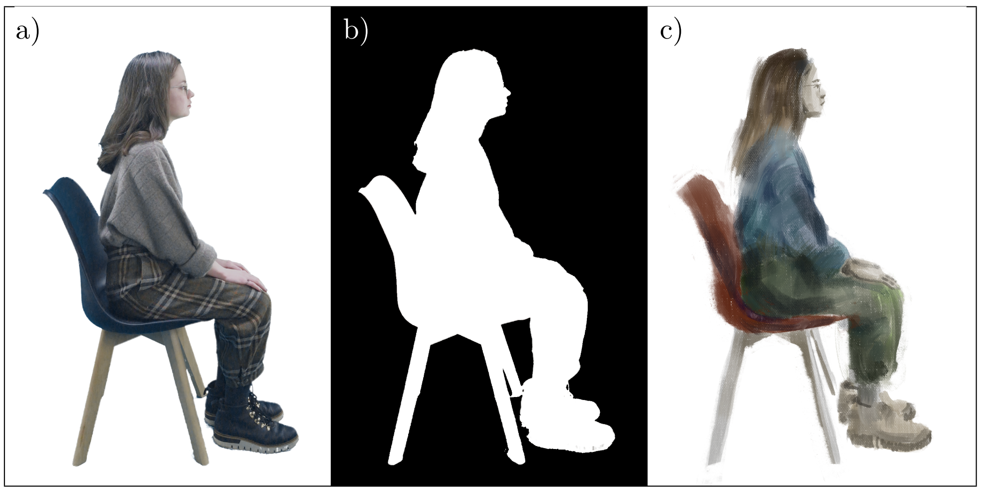


Figure 1.4: An example of an *original frame* (a), *mask* (b), and *keyframe* (c). *Keyframe (c) courtesy of* © Polina Akhmetzhanova.

Chapter 2

Blender

Blender [9] is a free open-source software primarily used for 3D modelling and animation. It also provides support for video editing, 2D animations, VFX (Visual effects), tracking and more. Whilst being well over two decades old [10], it is still being regularly updated and provides modern tools that are more than adequate for the purposes of this project. If not explicitly stated otherwise, the source of information for this chapter is the official *Blender reference manual* [11].

2.1 User Interface

The UI can be divided into three parts (also shown in figure 2.1):

Topbar Located at the very top. Contains all option menus, open workspaces and scenes.

Areas Located in the middle. Contains editors belonging to the currently active workspace.

Status bar Located at the very bottom. Is used to display general information, such as what action will mouse keys perform for the currently chosen editor, or the current memory usage.

Key features of *Blender* are split up into *editors*, which are used for displaying and editing different aspects of data. Some editors are further divided into *views*. The layout of the editors on the screen is called a *workspace*. The user is free to customize the workspace by adding editors to the desired place on the screen. *Blender* also provides a group of predefined workspaces, which are fully customizable as well.

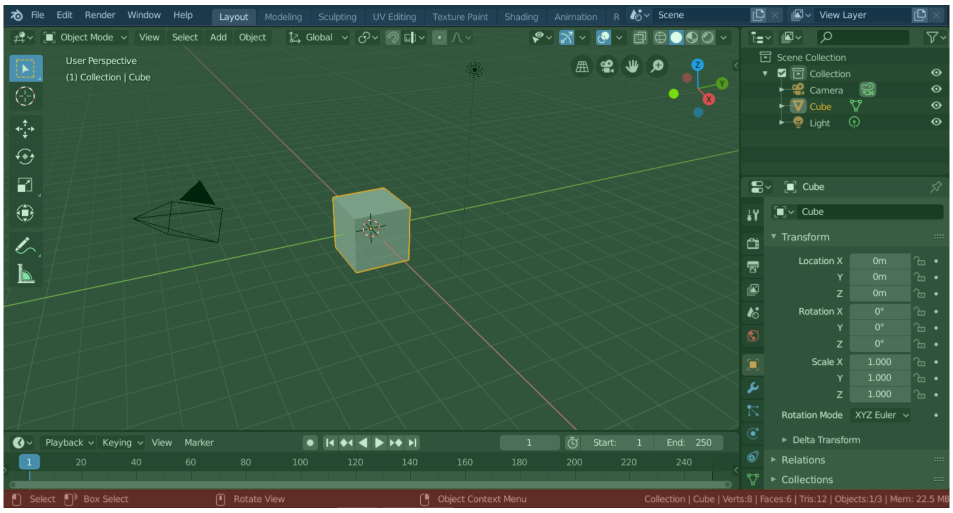


Figure 2.1: The default screen layout of *Blender*, split into highlighted UI parts: Topbar (blue), Areas (green) and Status bar (red) [12].

2.2 Scenes

Scenes are a way to organize data inside of a project. By default, all of the data stored from each editor belongs to a scene, which means that the data in a scene can be changed without affecting the other scenes. It is possible to create data *links* between scenes, which means the linked data will always be identical in those scenes, as they are "linked together" – they share the same data source.

Creating new scenes and switching between them can be done using the *topbar*. There are four options when creating a new scene:

New Creates an empty scene with default settings.

Copy settings Creates an empty scene with all of the editor settings copied from the previous scene.

Linked copy Creates a new scene with all of the data contents being linked from the previous scene (also known as a *shallow copy*). All the contents of the new scene are linked to the objects in the source scene, which means changes in one scene will result in changes in the other scene as well.

Full copy Creates a new scene with all of the contents and settings being fully copied from the previous scene (also knows as a *deep copy*). The contents of the source scene are fully copied over, which means changes in one scene will not result in changes in the other scene.

2.3 Editors

As described in section 2.1, features of *Blender* are split up into editors. This section covers the editors that are relevant for the purposes of this project.

2.3.1 Properties Editor

Displays and allows editing of data relevant to the current scene – for example the render settings or settings of the currently selected object. The settings are divided into *tabs* based on the category the edited data belongs into. These tabs are then further divided into *panels* which contain various UI elements suitable for displaying and editing the given data type.

2.3.2 Composition Editor

Allows the user to assemble or edit movie clips using a set of *control nodes*, which are sorted into categories by their functionality. Each node has a defined number of *input and output sockets*, which are used to link nodes between each other. The most common type of socket is an *image socket*, which is used to transfer image data. The nodes can also contain inner parameters that can be configured by the user. Below is a list of nodes useful for the purposes of this project.

Movie clip node Used to load a movie clip into the compositor. It Contains an inner *clip parameter* which is used to select the desired movie clip.

Keying node Is a node specialized for green/blue screen removal by containing parameters for multiple useful chroma keying techniques, such as defining a mask, despill balance, etc.

Mask node Allows to select a *mask data-block*, which can be then passed to other nodes to selectively edit parts of the image. It can be for example be connected to the *Garbage Matte* socket of the *keying* node to exclude the area defined by the mask from the node's output.

Stabilize 2D Allows to stabilize a clip using tracking data obtained from the *motion tracking* editor (described in the next subsection).

Crop node Used to remove unwanted parts of the image by defining a rectangular region.

Transform node Allows to move, scale and rotate the image based on the defined input parameters.

Composite node Serves as an output point of the compositor – image data routed into this node will be rendered.

2.3.3 Motion Tracking

Motion tracking is a technique used to track the motion of objects and/or camera in a 2D video sequence. The tracking data can then be applied to 3D objects, or to stabilize clips using the composition editor. An especially important use case of the technique for this project is centering and stabilizing a moving actor. It is possible to stabilize a moving object, so that its position and scale stays fixed throughout the playback of the clip.

Most of the motion tracking features of *Blender* are located in the *Movie Clip Editor*. The editor is further split up into three views, which each serve for editing and viewing different aspects of the tracking data.

Clip View

It is the main part of the editor, as it contains almost all of the tools to edit and create tracking data. First, the user must define the pattern to be tracked by placing *tracking markers* over it. *Blender* then tries to track the position of the patterns defined by the trackers throughout the playback of the clip. The user can step in at any frame and adjust the tracker if it was not tracked properly, or not tracked at all.

Each tracker has a defined *pattern area* and *search area*. The pattern area determines the area on the screen, which will be tracked. The search area determines an area, which will be searched for the tracked pattern during each frame update. The search area should be set according to the approximate velocity of the tracked pattern. If the search area size is too low, it will result in the pattern not being tracked properly, as it will often move out of the search bounds. On the other hand, if the search area size is unnecessarily high, it will take a long time to compute the new positions of the trackers.

Another important parameter of a tracker is the *motion model*. It defines in which ways can the tracked pattern be deformed. The options are:

Loc Only searches for translation (location on the screen) changes in the pattern between frames.

LocRot Searches for rotation and translation changes of the pattern between frames.

LocScale Searches for scale and translation changes of the pattern between frames.

LocRotScale Searches for scale, rotation, and translation changes of the pattern between frames.

Affine Searches for affine transformation changes (see figure 2.2) of the pattern between frames. It can be used to approximate changes in perspective relative to the camera.

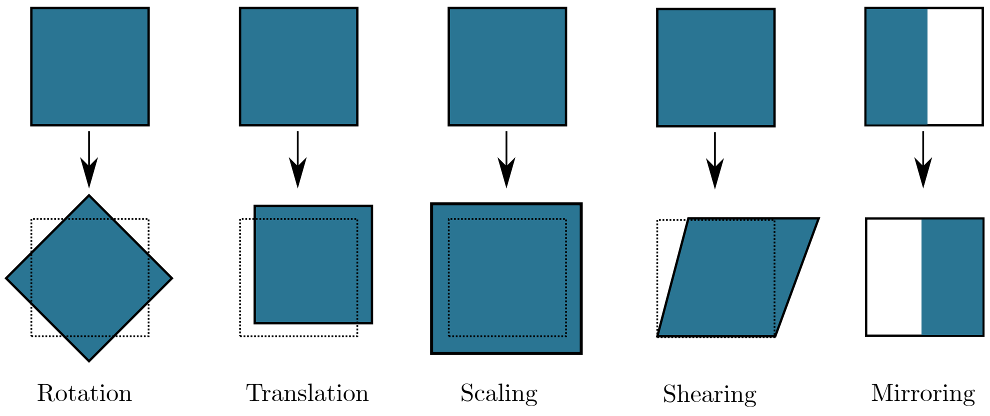


Figure 2.2: A visual example of the affine transformations on a square. Image reconstructed based on [13].

Perspective Searches for perspective transformation changes (homography) of the pattern between frames. Enables the pattern to be tracked more precisely in all dimensions, especially useful when the camera is freely moving.

Each tracker also has a *correlation value*, which determines the minimal correlation between matched pattern and reference (in percentage), for the tracking to be marked as successful. The reference pattern can be set to be used either from the *keyframe*, or the *previous frame*. Keyframe marker is always defined by the user, whereas the marker of the previous frame might be automatically generated. Setting it to compare with the previous frame can reduce the tracking precision, as the tracker might get deformed and less accurate over time.

If an object undergoes lighting changes during the playback (enters a shadow for example), the *normalize* setting can be used, to normalize the patterns by their average light intensity. This makes them invariant to illumination changes at the cost of lower tracking computation speed.

Graph View

The graph view displays the speed of the tracking markers over time using a *line graph*. Each tracker has two lines – green for vertical, red for horizontal movement speed, with the currently selected tracker being highlighted. A blue line can also be displayed after pressing *camera solve*, which displays the average per-frame error.

The view is especially useful for reviewing automatically generated tracking data. If any major tracking error occurs a spike will be visible in the graph

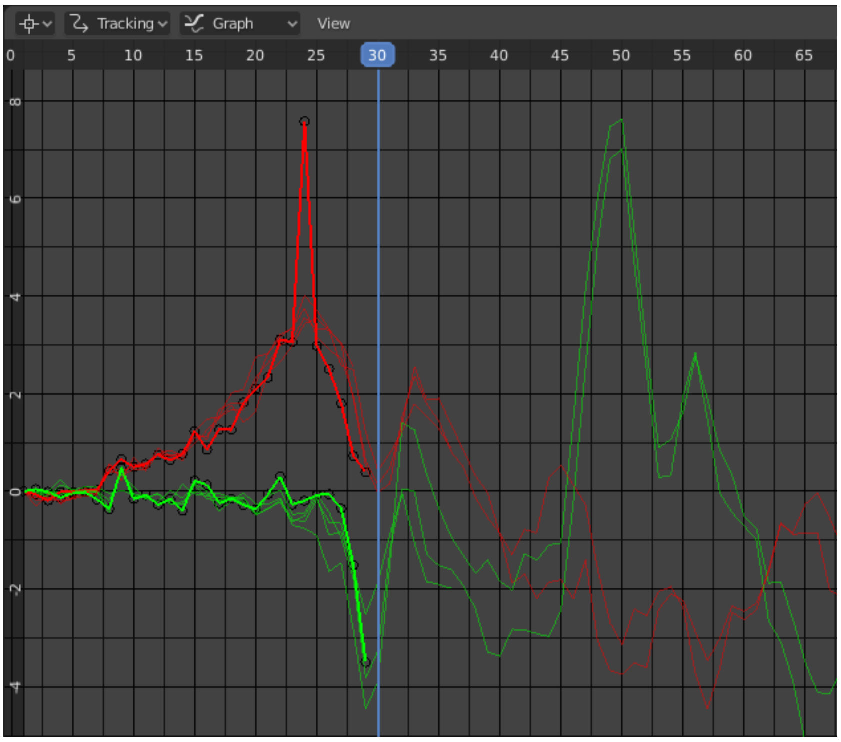


Figure 2.3: An example of a graph view for multiple trackers. The selected curve peaks high above the other curves between frames 20–25, which means a tracking error has probably occurred.

(an example can be seen in figure 2.3). It is possible to edit the tracking data directly in the graph view, by dragging the points of the graph.

Dope Sheet View

The dope sheet view provides an area to view and select all of the trackers in a concise manner. The visualization allows the user to see how many frames each tracker covers and where its keyframes are located. The view also allows to sort the trackers based on their properties. The trackers can be selected, but cannot be directly edited in this view (the user must switch to the graph or clip view).

2.4 Scripting

Blender provides a versatile way to extend its functionality, by utilizing the *Python* programming language. To interact with *Blender*, *Python* scripts can make use of the tightly integrated *Blender API* (Application Programming Interface).

Using this API, it is possible to edit data, create new tools, run existing ones, create user interface elements and much more. The scripts can be run directly from the console, but *Blender* also provides a built-in editor.

2.4.1 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It offers a wide variety of high-level libraries, which can be used to speed up development. It is also often used as a way to easily extend the functionality of a product, by implementing scripts. [14].

For the past years, Python users have been split between using two major versions – Python 2 and Python 3. Python 3 was released in 2008 and it focused on fixing a broad range of issues present in Python 2. However, due to the nature of these changes, Python 3 is backwards incompatible with Python 2, which means the code from the older version has to be rewritten in order to be Python 3 compliant. In 2020 the version 2 officially reached the EOL (End of Life) status and will no longer receive further updates. *Blender* has been using Python 3 since 2011 (version 2.5). [15, 16]

2.4.2 Operators

Most buttons and key-strokes in *Blender* call an *operator*, which are written in C, Python or macros. These operators can be directly called in scripts as well. The API provides a base class to create a custom operator. The inherited class can override a number of methods, the most important of them being `execute`, which is the method that gets executed when calling the operator.

2.4.3 User Interface

The API also provides base classes, which allow insertion of custom UI elements into the existing layouts. The most common UI container elements are *panels* (sidebar) and *menus* (dropdown). These can be placed into any editor by overriding the inherited class properties defining its placement.

These container elements can then contain *property elements* or another containers. In the module `bpy.props` a range of properties is provided, which allow storing and displaying different data types. Depending on the datatype, an appropriate UI element is displayed in the container (checkbox for boolean property, slider for integer property, etc.).

A custom button can be placed by referencing an *operator*. When the button is clicked by the user, the `execute` method of the operator is called.

2.4.4 Data Access

All of the *Blender's* internal data can be accessed and modified using the `bpy.data` module. While it is possible to access all data by name or by collection, it is also common to access data by the current selection of the user. The module `bpy.context` allows to access the currently active data members (e.g., the currently active scene). The context is read-only, so for data modification, the data module must be used.

Chapter 3

Unity

Unity [17] is a powerful multi-platform game engine, first introduced in 2005. It is available completely for free, without limiting any essential features, as long as the revenue generated from the project is not higher than $100,000 in a year, making it a great choice for small games and learning projects. It offers a large number of built-in features, as well as an asset store, which is used for sharing community-made features. Thanks to its popularity, countless learning resources are available, many of them free of charge. *Unity* mainly focuses on being a versatile tool – it offers development of both 2D and 3D games, supports deployment to all major platforms (including internet browsers), and allows to create fully customizable editor extensions. [18]

In this chapter, the basic elements of the game engine are briefly described, as well as features and editors, which could be useful for the purposes of this project. If not explicitly stated otherwise, the source of information for this chapter is the official *Unity manual* [19].

3.1 Main Concepts

This section provides information about the key concepts of *Unity*, which are essential for understanding the workflow of the game engine.

3.1.1 GameObjects & Components

GameObjects are the basic building stones of the game world. They serve as containers for *components*, which determine all the logical behaviour and properties of the object. They can be either pre-instantiated using the editor or created and configured using scripts at runtime. *GameObjects* can also be organized by nesting – defining a *parent* and *child GameObjects*. If a *GameObject* does not move at runtime, it can be marked as *static* in order to save on runtime computational resources, as many systems in *Unity* can precompute the information about a static object.

The only mandatory component for every *GameObject* is the *transform* component, as it is used to represent the object's location, rotation and scale in the game world. *Components* can be configured using the public properties, which can either be *values*, or *references* to other structures (for example other *GameObjects*, files, etc.). These properties can be edited either using the user interface in the editor or by code at runtime using the scripting API.

3.1.2 Assets

An *asset* is an item, typically a file, that can be used in a project. It can be imported from outside the project (3D models, audio files, video files, images, etc.), or it can also be created inside of *Unity* (scenes, animation controllers, etc.). *Unity* provides an official *Asset store*, which allows to easily share and trade assets created by the community.

3.1.3 Scenes

Scenes are a way to logically separate all pre-instantiated *GameObjects* in the game world. They could be thought of as levels of the game, although it is possible to have the entire game in one scene, meaning they are more of a tool for the designers to easily organize the *GameObjects*.

3.1.4 Prefabs

Prefabs allow to store *GameObjects* with all of their components as *assets* and reuse them anywhere in the project as *instances*. The main benefit of using prefabs is that all instances of a prefab can be synchronized. If changes are done to the prefab asset, those changes propagate into all of the instances of that prefab. If a specific instance is edited, then the changes done are called *overrides*. Those edited fields will no longer be synchronized with the prefab, which means that the instance of a prefab does not have to be fully synchronized with the prefab.

For more complex objects, it is also possible to nest prefabs – place a prefab into a prefab. A prefab can also be a *variant* of some other prefab, which means all unedited properties will be inherited from the *base* prefab.

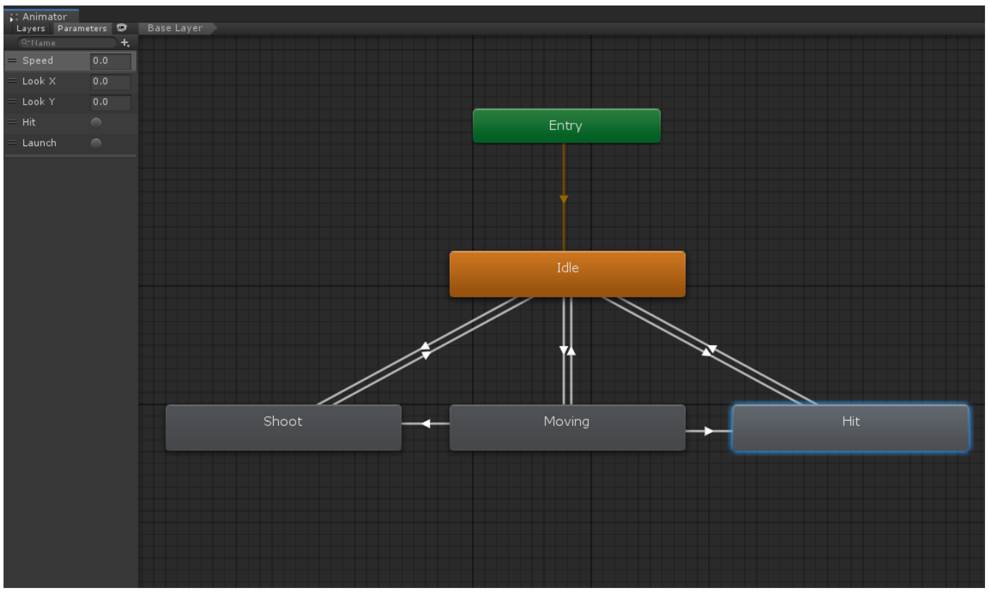


Figure 3.1: An example of an animator controller.

3.2 Animation System

Unity provides a sophisticated animation system called *Mecanim*. It is mainly designed for animating 3D objects but 2D sprite (PNG image format sequence) and 2D skeletal animations are also supported.

3.2.1 Animation Clips

The basic elements of the system are *animation clips*, which should represent a single linear action. The clip can be either imported from an external source or created in the editor. Various properties of the clip can be edited either using the editor, or the scripting API.

3.2.2 Animation Controllers

Animation controllers are used to create connections between *animation clips* using transitions, forming a *state machine*. During runtime, the controller keeps track of the current state and transitions to other states when the conditions are met. The controllers are created visually inside of the editor (see figure 3.1 for an example).

Parameters can be defined inside of the controller, which can serve as conditions for the defined transitions. These parameters can be then edited from outside the controller using the scripting API to control the flow of the state machine.

3.3 Physics System

This section briefly describes the essential components that allow physical interactions between objects.

3.3.1 Colliders

Collider components are used to define the shape which should be used for collision detection for a given *GameObject*. It is possible to use mesh colliders which are able to exactly match the shape of the *GameObject's* mesh, however, those are very computationally expensive and often not necessary. Primitive shape colliders (box collider, capsule collider and sphere collider in 3D) can be used to roughly approximate the shape of the mesh. A number of colliders can also be added to a single *GameObject* to create a *compound* collider.

3.3.2 Rigidbody

Rigidbody is the main component of the physics system, which enables physical behaviour of a *GameObject* when added to it. It allows the object to respond to gravity and forces being added to it in a realistic way. If the object contains a *collider*, then it can physically interact with other objects with colliders. The behaviour of the rigidbody can be adjusted using its parameters (e.g., its mass and drag).

3.4 Pathfinding

Unity offers a pathfinding system, which allows the game characters (agents) to navigate around the game world by building a navigation mesh based on the geometry of the scene. An overview of the system's components can be seen in figure 3.2.

3.4.1 NavMesh

NavMesh covers the area accessible by the agent and is used to compute the navigation path. It is built according to the geometry of the scene by testing where the agent can stand. The area is represented as a set of convex polygons. Such representation is useful because there are always no obstructions between two points of a convex polygon. The system stores the location of these polygons as well as references to neighbours of the polygons and applies the *A* graph traversal algorithm* to find the shortest paths.

3.4.2 NavMesh Agent

NavMesh agent components are added to the characters that supposed to move and navigate around the game world. It is represented as a cylinder with a

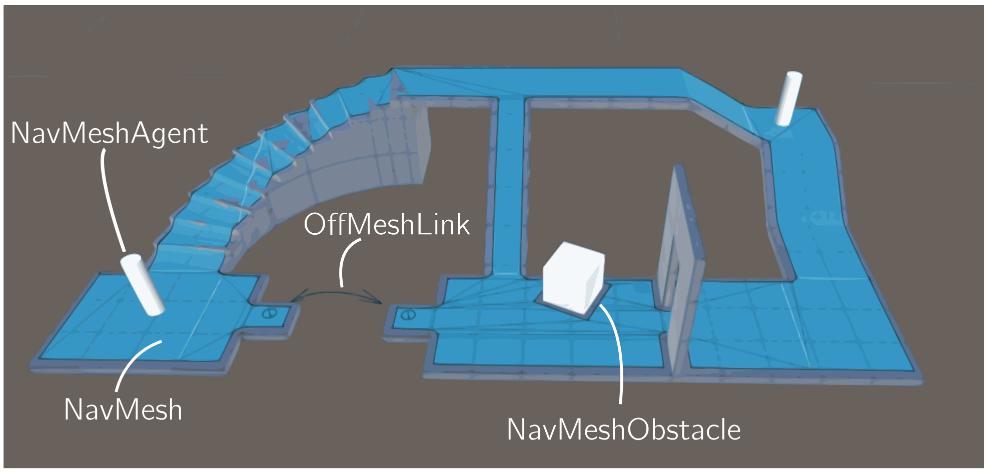


Figure 3.2: An example of the navmesh components. [21]

variable radius. When a valid target point is set to the agent, the system finds a path using the navmesh and moves the agent towards the target. It also attempts to avoid other agents and dynamic obstacles along the way by adjusting the path if needed.

3.4.3 NavMesh Obstacle

Allows to define an obstacle for the agent to avoid. If the obstacle is static, it gets carved directly into the navmesh. If it is dynamic, it is no longer computed alongside the navmesh, as it would be too expensive to recalculate the navmesh during every frame update. Instead, the agent tries to steer away from the dynamic obstacle, if it gets too close to one.

3.4.4 OffMesh Link

Allows to define path points, which could not be represented by walkable surface, for example jumping over a hole.

3.4.5 High-level NavMesh Building Components

Whilst not being officially built into *Unity*, refined navmesh building components are available at [20]. These components provide a better UI for navmesh building and allow to easily construct navmeshes at runtime.

3.5 Video Player

Unity allows to import and play video files inside the game using the *video player* component. It supports a wide variety of video formats, including formats with alpha channel (transparency support). It contains a number of configurable parameters (e.g., playback speed, aspect ratio, etc.).

3.5.1 Video Clip Playback

The supported video formats can be directly imported as assets and used as video clips for the video player component. *Unity* offers an option to transcode (convert) the video file if the imported video file format is not supported on the target platform. The only platform, where playback of video clips isn't supported is the *WebGL* platform, URL playback must be used instead.

3.5.2 Video URL Playback

The component also supports playback from URL (Uniform Resource Locator), which can be used to play video directly from the filesystem or web resources. *Unity* provides a special folder, *Streaming Assets*, which can be used for storing video files for direct streaming, as *Unity* does not modify files in this folder in any way and is also able to access it on any specific target platform.

Chapter 4

Analysis and Design

The first part of this chapter analyzes the necessary steps of recording the raw footage and the current editing process in *Blender*. Based on the findings of the analysis, a semi-automated custom tool to effectively process the footage is proposed and designed. In the second half of the chapter, animation and movement systems utilizing the synthesized animations are designed inside the *Unity* engine, with an emphasis on preserving the authenticity and high quality of the stylized footage.

4.1 Video Capture

As described in section 1.3, one of the greatest strengths of *EbSynth* is its ability to very precisely stylize live-action sequences. This section contains all of the technical requirements for producing the raw footage for the synthesis. The recording setup used for recording the footage utilised in this project is shown in figure 4.1.

4.1.1 Recording Device

The recording device does not have to be a fancy camera, any modern mobile phone will do just fine, as long as it can record in Full HD (High Definition) and has a decently wide colour range. The recording device should ideally be placed on a tripod, as the footage needs to be as stable as possible.

For best results, the codec of the output movie file should be set to the least compressed one available. If the footage were heavily compressed, the compression artifacts would make it more difficult to properly remove the background. For dynamic scenes, it is also recommended to increase the shutter speed, in order to reduce the motion blur. [22]

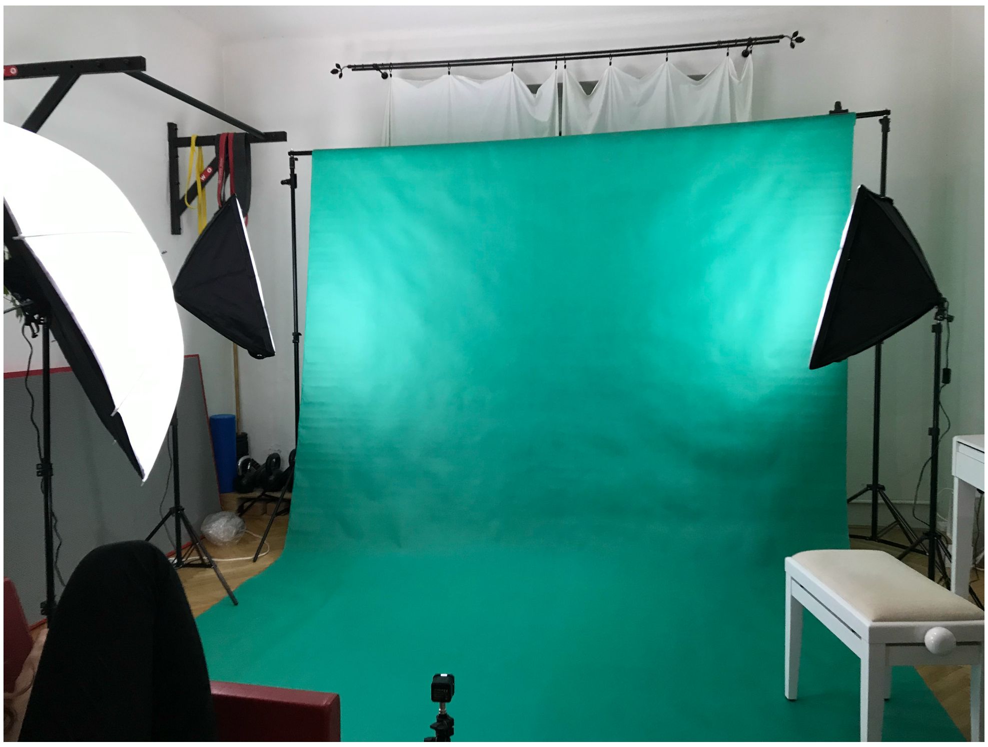


Figure 4.1: The setup used for recording the footage for the project.

4.1.2 Green Screen

In order for the recorded footage to be usable in game development, the background needs to be easily removable in post-production. The most common method to do so is using a green screen. By placing a green background behind and below the actor (a green sheet or wall), the background can be easily removed using an editing software. Technically, other solid colours would work, but green is most suitable because it is far from human skin tones in the colour spectrum (most camera sensors also capture more information on the green channel). The actor should stand as far away from the background as possible, to minimize colour reflection from the screen onto the actor. [23]

4.1.3 Lighting

Another important step is to properly light the scene. For best effect, the background should be uniformly lit, so that it does not contain any distinctly darker or lighter patches. It is recommended to use at least two lights to illuminate the background, in order to eliminate shadows cast by the actor. One light should also be illuminating the actor.

4.1.4 Motion Capturing

Recording animations that require motion, for example walking, can be tricky because in the processed clip, the subject must stay centered at all times. Not doing so would make it impossible to loop and connect the animations. There are three possible approaches:

Walking in place The actor can create an illusion of walking without actually moving by sliding their feet on the ground. Whilst *Michael Jackson* makes this look very easy, it is actually quite tricky to pull off. Having green cloth on the ground also does not make the sliding any easier.

Motion tracking Using an editing software, it is possible to automatically track the movement of an actor and transform the footage so that the actor stays centered. It is even possible to scale the actor when they move closer or further from the camera. For good tracking results, it is recommended to place small *trackers* onto the actor, as they help to produce consistent and accurate tracking. There are no specific requirements for the trackers, they can, for example, be a piece of coloured tape. It is only important for it to be non-reflective and contrasting with its surroundings.

Even with good tracking results, the animation will not loop perfectly, because the angle of the actor in relation to the camera changes as they move. The green screen also has to be wide enough, so that the actor can take at least three steps, otherwise there would be no way of looping the animation. This is the option chosen for this project.

Treadmill Having the actor walk on a treadmill would produce the best results out of the three methods. However, it is not very budget-friendly and it would also have to be completely painted green. For those reasons, it was not used when recording the footage.

4.2 Processing the Footage

In order for the animations to be usable in *Unity*, they need to be processed first. That means removing the background, cropping and aligning the footage. If the actor physically moves, they also need to be stabilized, so that their size and position is fixed throughout the animation. Keyframes for the *EbSynth* algorithm are then selected and stylized by the artist. Using the available *EbSynth* tool, the clips are then stylized (see figure 4.2 for an example). In this section, the workflow using the current tools of *Blender* is analyzed and an add-on is proposed, which simplifies and automates steps of the process.

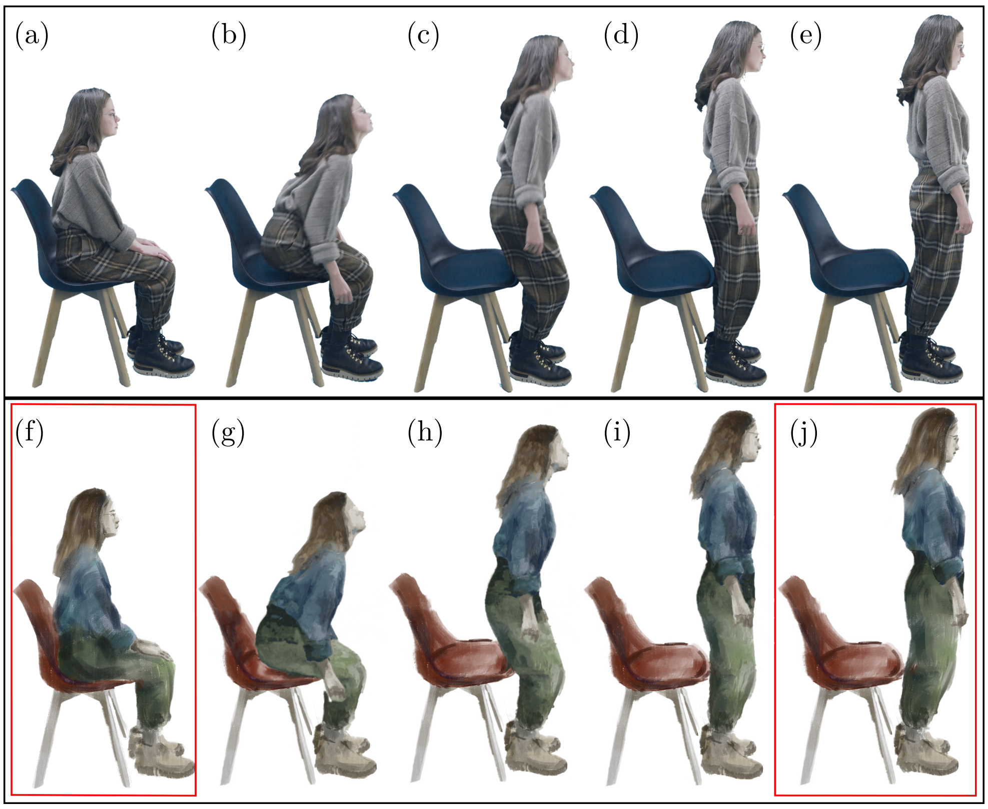


Figure 4.2: An example of the processed images (a–e) being used to create synthesized images (g, h, i) based on two exemplar keyframes (f, j). Stylized keyframes (f, j) courtesy of © Polina Akhmetzhanova.

4.2.1 Tracking and Stabilization

As described in subsection 2.3.3, *Blender* provides extensive tools for tracking an object and stabilizing it. Using these tools, the footage of moving actors can be adjusted, so that their position and scale relative to the camera stays fixed throughout the playback. If the actor does not move in the footage, this step can be skipped.

Exporting the Tracking Data

The data obtained from the tracking can be useful to synchronize the animation with the velocity of the actor. *Blender* does not provide a way to export the tracking data in the user interface. However, by using the *Python API*, the data can be extracted and exported into a `csv` (comma-separated values) file format.

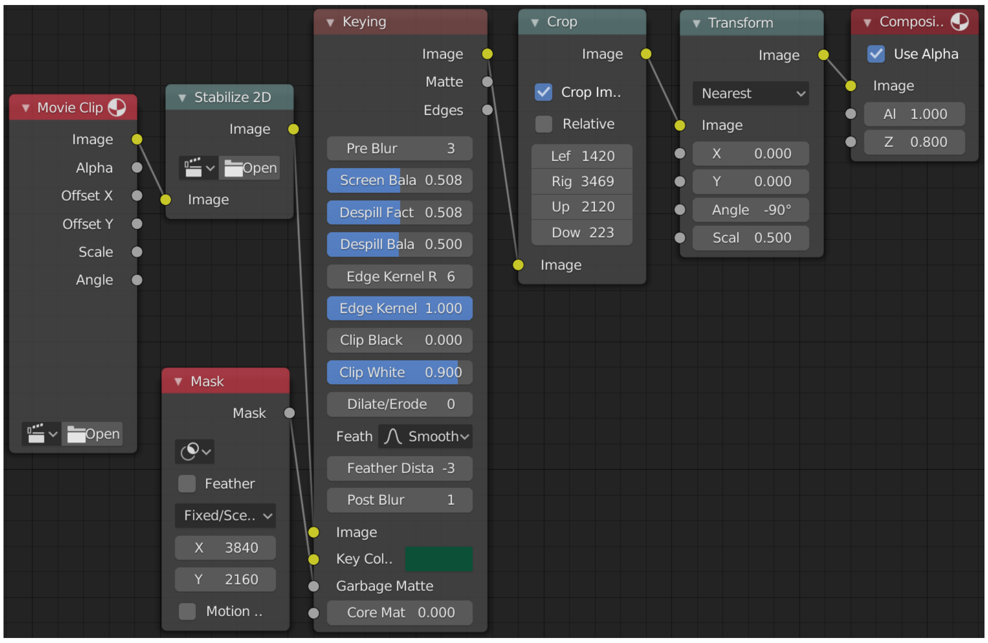


Figure 4.3: The node setup for basic chroma keying in *Blender*.

Rendering the Stabilized Clips

After tracking the actor, the footage needs to be further processed. The tracking data can be forwarded into the *compositing editor* and used to stabilize the footage. It can then be further processed, as described in the next section.

4.2.2 Chroma Keying

Chroma key compositing is a technique of layering two images together based on colour hues, removing a defined colour spectrum from the first image and replacing it with the second image [22]. For the purposes of this project, the colour will be replaced with full transparency, effectively creating a cutout of the actor.

Preparing the Editor

Inside of the *compositing editor*, *Blender* provides extensive tools for this task (described in subsection 2.3.2). Satisfactory chroma keying results can usually be achieved easily using only the *Keying* node by appropriately adjusting the node's parameters. Additionally, the footage usually needs to be cropped and scaled to the required size, which can be done using the *Crop* and *Transform* nodes. Unwanted static objects in a scene (for example markers) can also be removed with a mask. The mask is created in the *Movie Clip Editor* and then input into the compositor using the *Mask* node. If the actor was moving and has to be stabilized, the *Stabilize 2D* node can be used to utilize the tracking data and stabilize the image. Finally, the *Composite* node is used for rendering the sequence. The connections between the nodes are shown in figure 4.3.

Workflow Using Only the Built-in Tools

After preparing the nodes, the workflow using the *Blender's* built-in tools can be summarized as:

1. The user sets the *clip* parameter in the *Movie Clip* node to the next clip file to be processed. If the actor is moving, the user must also manually define the clip in the *Stabilize 2D* node.

2. The user edits the control nodes, so that the footage is correctly chroma keyed, cropped and scaled. A mask can also be defined, to remove unwanted objects.

3. The user sets the number of frames to be rendered, selects the output folder and waits for the footage to finish rendering.

4. If a mask is also to be rendered, the user connects the *Matte* output socket of the *Keying* node to the *Image* input socket of the *Crop* node. The user must also edit the output image colour format to *BW* (black and white) and specify the mask output folder. The user then waits for the mask to finish rendering.

5. If there are more clips left, the process repeats from item 1.

The workflow is also shown in an activity diagram shown in figure 4.4.

Task Automatization

Whilst *Blender* provides all of the tools necessary to fully process the footage, several tasks are repetitive and could be automated or simplified:

- Defining the input file for every individual clip could be reduced to defining a folder containing all the clips to be processed.

- Defining the output folder for every footage and mask clip can be reduced to defining a general folder, where the output folder structures will get created automatically.

- Switching to the mask output (rerouting the node connections) could be fully automated.

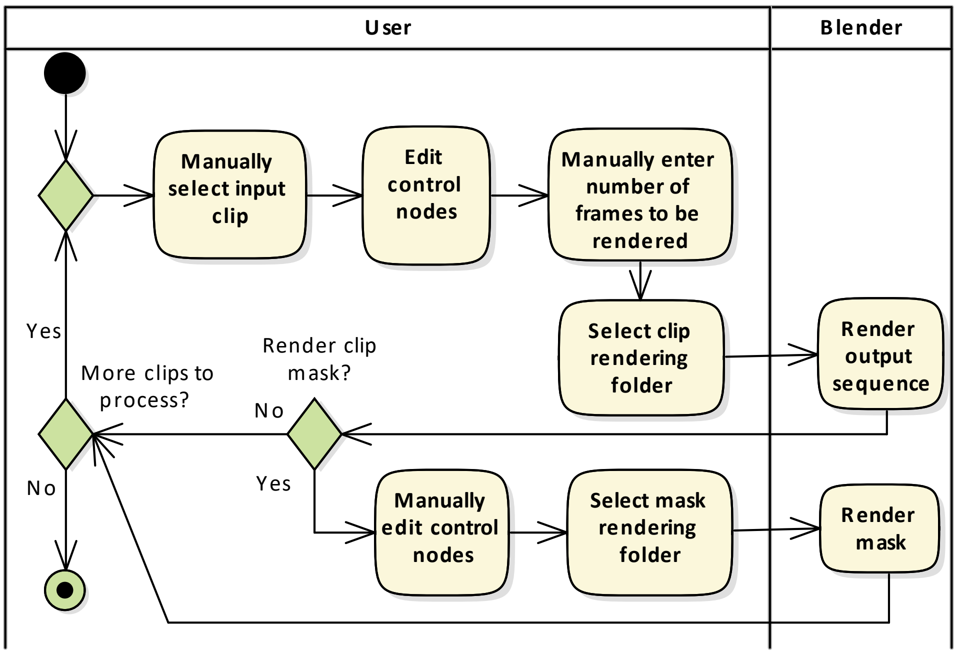

User
Blender
Manually select input clip
Edit control nodes
Manually enter number of frames to be rendered
Select clip rendering folder
Render output sequence
Yes
More clips to process?
Render clip mask?
No
Yes
No
Manually edit control nodes
Select mask rendering folder
Render mask

Figure 4.4: Activity diagram describing the chroma keying workflow without the use of custom scripts.

- The number of frames to be rendered could be set automatically to match the clip's length.

These tasks are not very time consuming on their own, but they have to be done every time when processing a clip. When processing large amounts of footage, these repetitive tasks can take up a sizeable portion of the total time spent.

Probably the biggest problem with this approach, however, is the fact that the user needs to wait for each clip to finish rendering before they can begin processing the next clip. When rendering, the functionality of the editor is limited, so the user cannot prepare the next clip in the meantime. This could be bypassed by opening another instance of *Blender*, it would still, however, cause performance issues, as rendering is very CPU intensive. Depending on the hardware *Blender* is running on and the length of the clip, rendering can on average take several minutes to complete. This issue could be solved by sequentially rendering all of the clips automatically after the user is done editing all of the footage.

Workflow Using a Custom Add-on

If the addressed tasks were automated by implementing an add-on with custom functionality, the improved workflow could be summarized as:

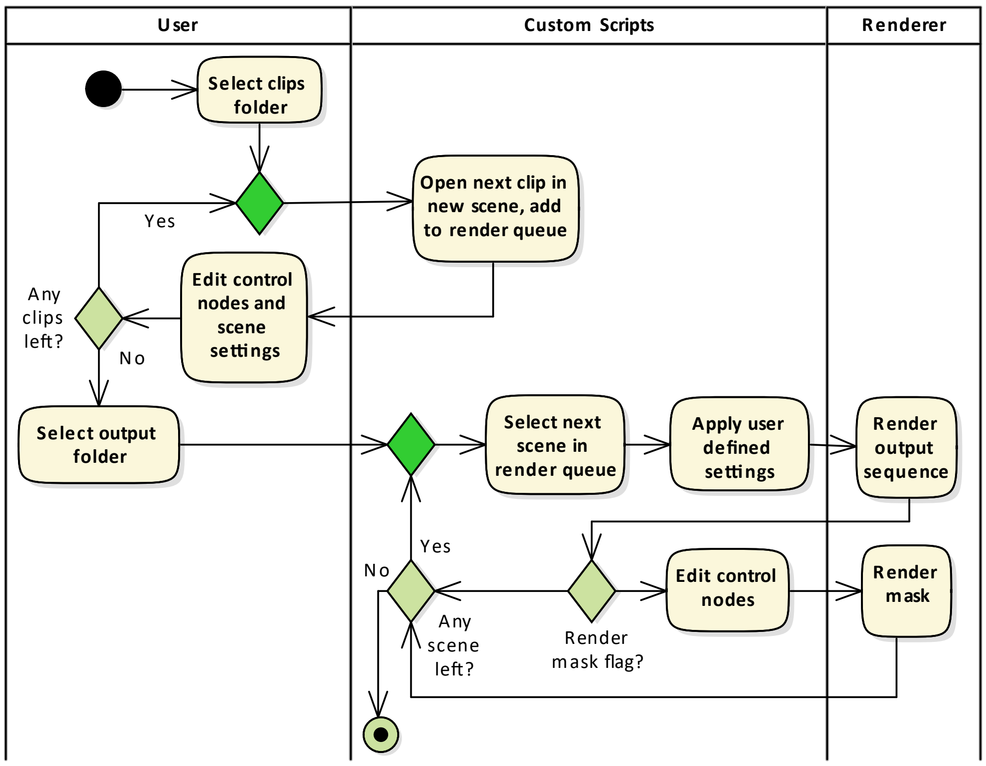


Figure 4.5: Activity diagram describing the workflow utilizing the proposed custom functionality.

1. The user defines a folder, which contains all of the clips they want to process. They also define a namespace of the current batch.

2. The user selects the next clip from the list.

3. The user edits the control nodes and custom script settings.

4. If there are more clips left, the process repeats from item 2. Otherwise, the user defines an output folder and using the tool renders the clips as a batch.

A simplified diagram describing the new workflow can be seen in figure 4.5. The diagram includes a summary of the steps taken by the custom script when rendering the batch. It is apparent from the diagram, that the user no longer directly interacts with the *Blender* renderer, which eliminates the problem of waiting for the individual clips to finish rendering.

4.3 Custom Blender Addon

Based on the observations in the previous section, a custom tool (add-on) can be designed, which would automate or simplify the repetitive time-consuming tasks. The extension can be integrated directly into the *Blender* user interface using the *Python API* described in section 2.4.

4.3.1 Functional Requirements

1. **Tracking data export** The user should be able to export the data of any tracker into a `csv` file.

2. **Mass import** The user should be able to import multiple video clips to be processed by specifying only the folder they are located in. These clips should then be displayed in an interactive list, which should allow the user to switch between the clips. It should be visually indicated, whether the user has already visited a clip in the list. Switching to a clip should automatically set it as input in the *Movie Clip* and *Stabilize 2D* nodes.

3. **Scene system** When switching between clips, the changes done to the compositing nodes should be saved independently. When switching to a new clip, the settings of the nodes should be copied from the clip the user switches from.

4. **Namespace** The user should be able to define a namespace for the current batch, which would serve as an identifier between multiple batches of clips having the same name.

5. **Mask system** The user should be able to specify, whether they want to render a mask of the current clip. They should also be able to switch to the mask preview, without having to directly modify the connections between the nodes.

6. **Automatic frame range** The user should be able to specify, whether they want the frame range set automatically when rendering. If they do, then the frame range should be set to cover the duration of the entire clip.

7. **Output folder structure** The user should be able to specify the output folder, into which the rendered footage should be saved. The footage should be further divided into folders named as the corresponding input clip.

8. **Batch render** When the user is done editing the clips, the tool should be able to render all of the edited clips as a batch, applying the user-defined settings to each clip and rendering it into the output folder.

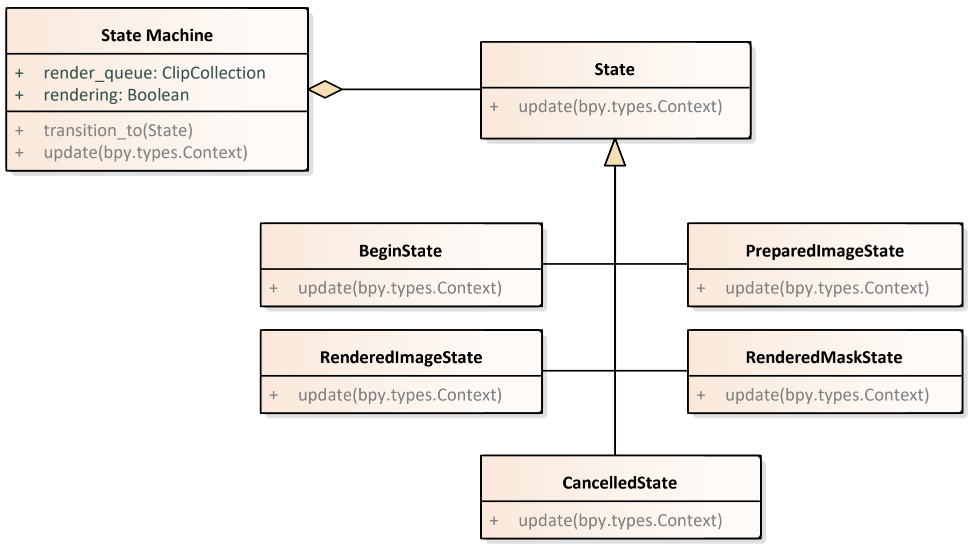

State Machine
+ render_queue: ClipCollection
+ rendering: Boolean
+ transition_to(State)
+ update(bpy.types.Context)
State
+ update(bpy.types.Context)
BeginState
+ update(bpy.types.Context)
PreparedImageState
+ update(bpy.types.Context)
RenderedImageState
+ update(bpy.types.Context)
RenderedMaskState
+ update(bpy.types.Context)
CancelledState
+ update(bpy.types.Context)

Figure 4.6: Class diagram of the state pattern implementation.

4.3.2 Batch rendering

When rendering the clips, they each have to go through several stages of preparation. In order to avoid a large block of if-else statements, *state* design pattern described in [24] can be used for the decision-making logic. The pattern has been slightly modified for *Python*, inspired by [25]. A class diagram describing the pattern is shown in figure 4.6. The update function is called from the modal operator each timer cycle when not rendering.

The states have the following functionality:

Begin State Gets the next scene in the *render queue* and switches to it inside the editor. It also transitions the state machine to *Prepared Image State*. If the *render queue* is empty, then the update cycle is stopped.

Preparing Image State Configures the scene settings, sets the `rendering` flag to true and launches the rendering of the image.

Rendered Image State If the user wishes to render mask for the given clip, then it switches to the mask view, changes the output settings, sets the `rendering` flag to true and launches the rendering of the mask. Otherwise it transitions the state machine to the *Begin State*.

Rendered Mask State Changes the scene settings back to the previous configuration and transitions the state machine to the *Begin State*.

Cancelled State transitioned to when the user cancels the rendering, stops the update cycle.

It might seem that the *Begin State* and *Preparing Image State* should be merged together, as their functionality could be run consecutively. The reason for them being split is that when switching scenes, the context sometimes does not update immediately, which might create problems when rendering. With the "pause" between switching the scene and rendering, the context has time to update properly.

4.4 Asset File Formats & Compression

This section explains why the animation formats supported by the built-in *Unity* animation system are insufficient for the purposes of this project and proposes a more space-efficient format. It also describes a compression method suitable for static assets.

4.4.1 Animation File Format

The two 2D animation types supported by the *Mecanim* animation system (described in section 3.2) are sprite animations and skeletal animations. The skeletal system cannot be used, as it would produce an unnecessary step in animation processing and would render the animations less authentic. The animations could be imported as a sequence of sprites (images) without any extra work and compromise of authenticity, however, that type of animation is only viable for low-resolution, low-framecount animations due to the inefficient compression of image sequences.

Video Compression

Unlike image compression, video compression can utilize the "extra dimension" – progression in time, to encode parts of an image and reference to that information if it does not change in the following frames.

Video encoders use *I-frames* as points of reference – those are fully encoded images, compressed in a similar way as regular image files. The algorithm then predicts the frames in between the *I-frames*. *P-frames* are predicted based on how the data changes from the last *I-frame*, *B-frames* are bi-directionally predicted based on both the last *P-frame* and next *P-frame* (see figure 4.7 for a visual example). [26]

The efficiency and quality of the predictions then depend on the compression algorithm used, which is called a *codec* (coder and decoder). The encoded information is then stored in a *container* (also called the video format), which determines how the encoded data and metadata should be structured in the file. [27]

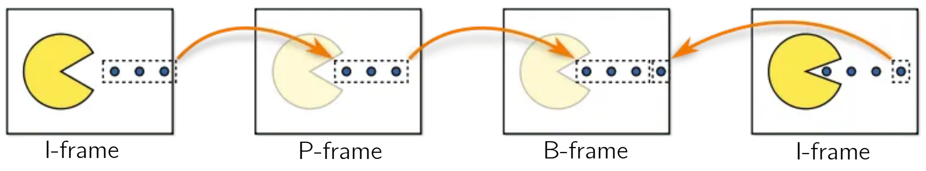


Figure 4.7: A visual example of the compression frame types and flow of data used for predictions. [26]

Supported Video Format

Currently, the only combination of video *container* and *codec* compatible with all *Unity* platforms, that also supports alpha channel, is the *Webm container* with the *VP8 codec* [28]. The video file can be then played using the *video player* component described in section 3.5.

Unfortunately, *Blender* does not allow to render using this combination of container and codec. For that reason, the animations will be exported as a PNG image sequence and converted using a conversion tool *FFmpeg* [29].

4.4.2 Image File Compression

In order to preserve the artistic look, many of the image assets are in very high resolution, which means that they are quite large in terms of storage space. In order to reduce the size of the images, it is possible to use image compression tools. The compression should however not directly alter the patterns of the image, in order to preserve the fine artistic details. The compression method must also support alpha channel (transparency).

Pngquant [30] is a command-line utility for lossy compression of images in the PNG format. It is able to significantly reduce the file size (on average by around 70%), whilst preserving the alpha channel. It uses a modified version of the *Median cut quantization* algorithm. The algorithm is able to compress the image by reducing the size of its colour palette, effectively lowering the number of bits needed to represent each pixel. This does mean that some finer colour details of the image are lost, however, those changes are in most cases indistinguishable by the human eye when viewing the image as a whole.

4.5 Animation System For Video Files

As described in section 4.4, the video format is the most suitable for importing the animations in terms of optimal file size (the animation sets are on average 100 times more compact in video format). Unfortunately, the built-in animation system does not support video files. In this section, a new simplified video animation system is designed.

4.5.1 Animating Moving Characters

In the traditional animation system, the animations of dynamic characters are driven by the movement system of the character – they get adjusted and transitioned using the incoming velocity data and have no control over the movement of the character in the game world. That can be well executed for 3D models, as they can be directly controlled and blended to create seamless transitions.

The video animations, however, do not have this advantage and have to be fully played out. Otherwise, the transitions will be abrupt and noticeable. The speed of the character must also proportionately match the frames of the animation. Otherwise, the character will appear to be sliding when accelerated at an incorrect rate. As all characters walk a bit differently, it would be very difficult to correctly configure the acceleration and movement speed values for all of them. The rest of this subsection proposes, how these issues could be practically solved.

Animation Continuance

In order to be able to extend the movement animations, they can be split into three parts:

Walking begin Contains the beginning of the animation and the first step.

Walking loop Contains the next two steps. This animation can be played repeatedly, in order to expand the duration of the walking animation.

Walking end Contains the last step.

Animation Clip Velocity

In order to better synchronize the animations with the character's speed, the direction of the velocity data flow can be reversed. Using the motion tracking data obtained from processing the dynamic clips, the animations can have a defined distance travelled for each frame and directly control the speed of the animated character in the game world. That way, the speed of the character in the game world will be always correctly synchronized with the animation.

Adjustable Clip Velocity

As described earlier, the animation clips should fully play out for smooth transitions, which means that with a fixed playback rate, the travel range of the animation would not be very flexible (could only be increased discretely by increasing the number of loop clips played). This can be solved by adjusting the velocity by a multiplier, so that the total distance travelled matches the distance from the target. First, the closest base animation distance is calculated:

$$nLoops = \lfloor \frac{dTarget - dBegin - dEnd}{dLoop} \rceil$$
$$dAnim = dBegin + nLoops \times dLoop + dEnd$$

The *dBegin*, *dLoop*, *dEnd* are the distances travelled during each respective clip and *dTarget* is the distance from the target point. Knowing the closest base animation distance, the ratio between the target distance and animation distance can be obtained:

$$dRatio = \frac{dTarget}{dAnim}$$

By multiplying the base animation velocities and playback speed by this ratio, the distance travelled can be adjusted to match the specific travel distance, whilst maintaining the same speed proportions between the individual frames.

4.5.2 Components of the Video Animator

Similarly to the built-in *Unity* animator, the custom video animator will use the *state pattern* [24] to structure and control different stages of animation. This subsection describes the main components and concepts of the video animator.

Clip Containers

The video clips are stored in an encapsulation class `VClip`, because the *WebGL* platform only supports playback of videos using *URL* source, whilst the *VideoClip* source type is more appropriate for other platforms. The `VClip` class contains both these fields and selects the source type based on the current deployment platform.

For directional animations, the clips are stored in a `VClips4D` class, which outputs a `VClip` based on the chosen direction. For directional movement animations, the velocity data files are also stored in `VClipsVelocity4D`, which allows to store and output velocities for specific clips. The structure of these classes can be seen in figure 4.9.

States

States represent the current animation phase and control the playback of video files. For the more complex dynamic animations, they can also contain logic to determine the animation speed multiplier and the number of loop animations, before switching to the other phase.

The current animation state is stored and controlled in `VStateMachine`. The basic structure with an example of the states used for character movement can be seen in figure 4.8.

Controllers

Controllers serve as a control point for the animation – they keep track of the current animation state and allow to switch to different animation states (their base class is always the `VStateMachine`). They also store the *clip containers* (to make it easier to define the clips from the editor UI). Every controller must inherit from the abstract class `VAnimController`, which contains all of the necessary logic to initialize and control video playback. The base controller class has one generic parameter, which defines the owner of the animation controller when specified. Figure 4.9 describes the abstract controller and two examples of non-abstract controllers.

The `VAnimCharacterController` allows to control the more complex dynamic animations, split into three parts with the distances defined for every clip frame. The animation playback directly controls the speed of the animated *GameObject*, using the `CharacterMovement` owner class (described further in the next section).

The `VAnimSimpleCharacterController` can be used for simpler dynamic animation, for example when the velocity data is not available, or the animation cannot be split into three parts. The animation playback does not control the speed of the animated *GameObject*.

4.6 Navigation and Movement

This section describes which form of navigation and movement has been chosen to control the in-game characters. The built-in pathfinding system described in section 3.4 can be utilized to automatically navigate the characters through the game world.

4.6.1 Simple Movement

Simple character movement can be chosen, when the animation clips of the character cannot be split into three parts, or when the animation distance data is not available (or not necessary). This movement system directly controls the playback of the animation by supplying the animation controller with the current `velocity` value of the *GameObject*. For moving the controlled *GameObject* towards the target point, the navigation system is directly utilized.

The simple movement can be for example used for sheep movement, as the current sheep animations are not complex enough to benefit from the more advanced animation controlled movement. See figure 4.10 (right column) for the detailed class structure.

4.6.2 Animation Controlled Movement

The more complex movement system utilizes the distance information available from the animation playback. The movement class supplies the animation controller with the location of the target point. The velocity of the animated *GameObject* is then directly controlled by the animation controller during animation playback.

The class structure can be seen in figure 4.10 (left column). The base class for this movement type is `CharacterMovement`. It contains all of the necessary logic and methods to communicate with the animator.

`CharacterNavMeshMovement` is more specialized, as it utilizes the pathfinding system to obtain the path points towards the target (however, the *GameObject* itself is not controlled by the navigation system). The child of that class is the `PlayerMovement`, which allows setting the target point using player input.

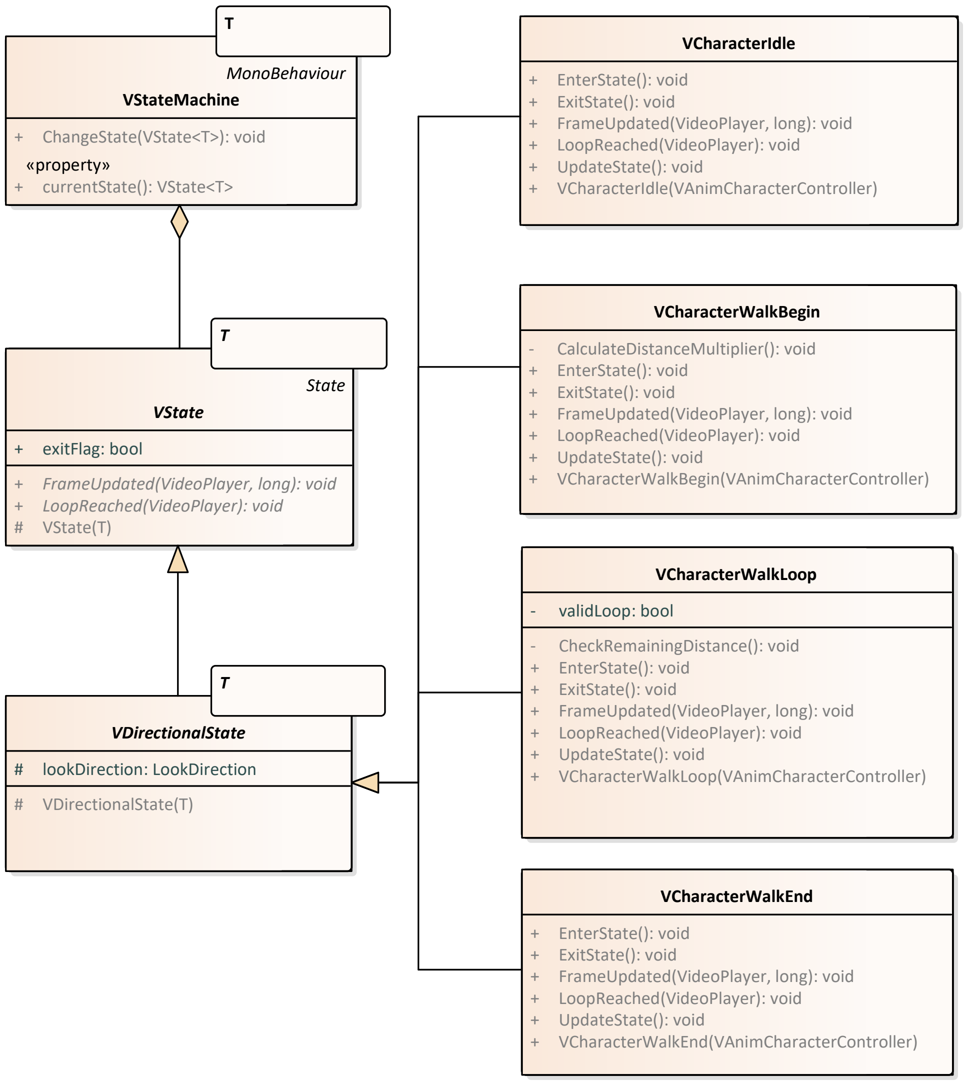

T
MonoBehaviour
VStateMachine
+ ChangeState(VState<T>): void
«property»
+ currentState(): VState<T>
T
State
VState
+ exitFlag: bool
+ FrameUpdated(VideoPlayer, long): void
+ LoopReached(VideoPlayer): void
# VState(T)
T
VDirectionalState
# lookDirection: LookDirection
# VDirectionalState(T)
VCharacterIdle
+ EnterState(): void
+ ExitState(): void
+ FrameUpdated(VideoPlayer, long): void
+ LoopReached(VideoPlayer): void
+ UpdateState(): void
+ VCharacterIdle(VAnimCharacterController)
VCharacterWalkBegin
- CalculateDistanceMultiplier(): void
+ EnterState(): void
+ ExitState(): void
+ FrameUpdated(VideoPlayer, long): void
+ LoopReached(VideoPlayer): void
+ UpdateState(): void
+ VCharacterWalkBegin(VAnimCharacterController)
VCharacterWalkLoop
- validLoop: bool
- CheckRemainingDistance(): void
+ EnterState(): void
+ ExitState(): void
+ FrameUpdated(VideoPlayer, long): void
+ LoopReached(VideoPlayer): void
+ UpdateState(): void
+ VCharacterWalkLoop(VAnimCharacterController)
VCharacterWalkEnd
+ EnterState(): void
+ ExitState(): void
+ FrameUpdated(VideoPlayer, long): void
+ LoopReached(VideoPlayer): void
+ UpdateState(): void
+ VCharacterWalkEnd(VAnimCharacterController)

Figure 4.8: Class diagram describing the state animation structure.

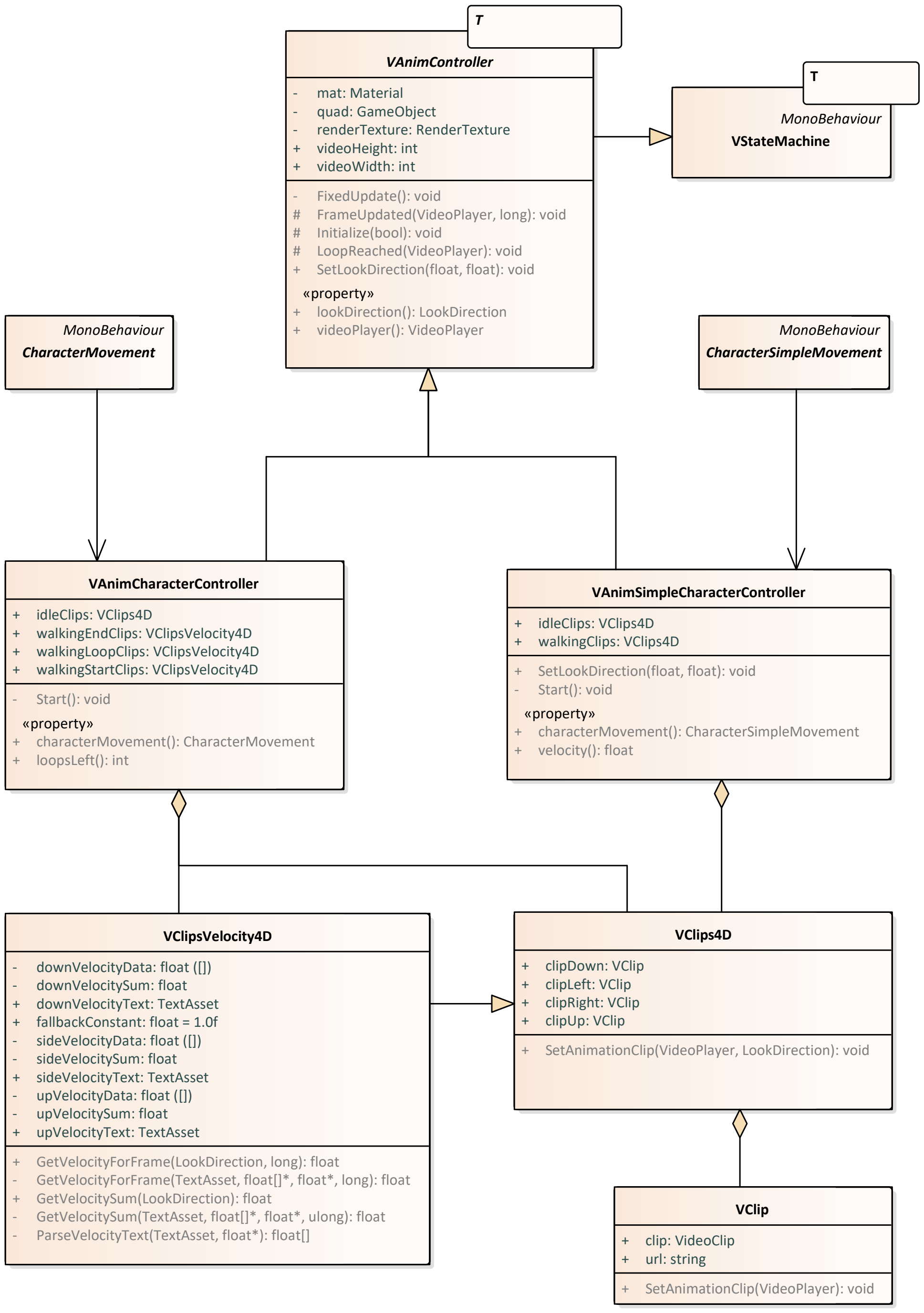

T
VAnimController
- mat: Material
- quad: GameObject
- renderTexture: RenderTexture
+ videoHeight: int
+ videoWidth: int
- FixedUpdate(): void
# FrameUpdated(VideoPlayer, long): void
# Initialize(bool): void
# LoopReached(VideoPlayer): void
+ SetLookDirection(float, float): void
«property»
+ lookDirection(): LookDirection
+ videoPlayer(): VideoPlayer
T
MonoBehaviour
VStateMachine
MonoBehaviour
CharacterMovement
MonoBehaviour
CharacterSimpleMovement
VAnimCharacterController
+ idleClips: VClips4D
+ walkingEndClips: VClipsVelocity4D
+ walkingLoopClips: VClipsVelocity4D
+ walkingStartClips: VClipsVelocity4D
- Start(): void
«property»
+ characterMovement(): CharacterMovement
+ loopsLeft(): int
VAnimSimpleCharacterController
+ idleClips: VClips4D
+ walkingClips: VClips4D
+ SetLookDirection(float, float): void
- Start(): void
«property»
+ characterMovement(): CharacterSimpleMovement
+ velocity(): float
VClipsVelocity4D
- downVelocityData: float ([])
- downVelocitySum: float
+ downVelocityText: TextAsset
+ fallbackConstant: float = 1.0f
- sideVelocityData: float ([])
- sideVelocitySum: float
+ sideVelocityText: TextAsset
- upVelocityData: float ([])
- upVelocitySum: float
+ upVelocityText: TextAsset
+ GetVelocityForFrame(LookDirection, long): float
- GetVelocityForFrame(TextAsset, float[]*, float*, long): float
+ GetVelocitySum(LookDirection): float
- GetVelocitySum(TextAsset, float[]*, float*, ulong): float
- ParseVelocityText(TextAsset, float*): float[]
VClips4D
+ clipDown: VClip
+ clipLeft: VClip
+ clipRight: VClip
+ clipUp: VClip
+ SetAnimationClip(VideoPlayer, LookDirection): void
VClip
+ clip: VideoClip
+ url: string
+ SetAnimationClip(VideoPlayer): void

Figure 4.9: Class diagram describing the animation controller structure.

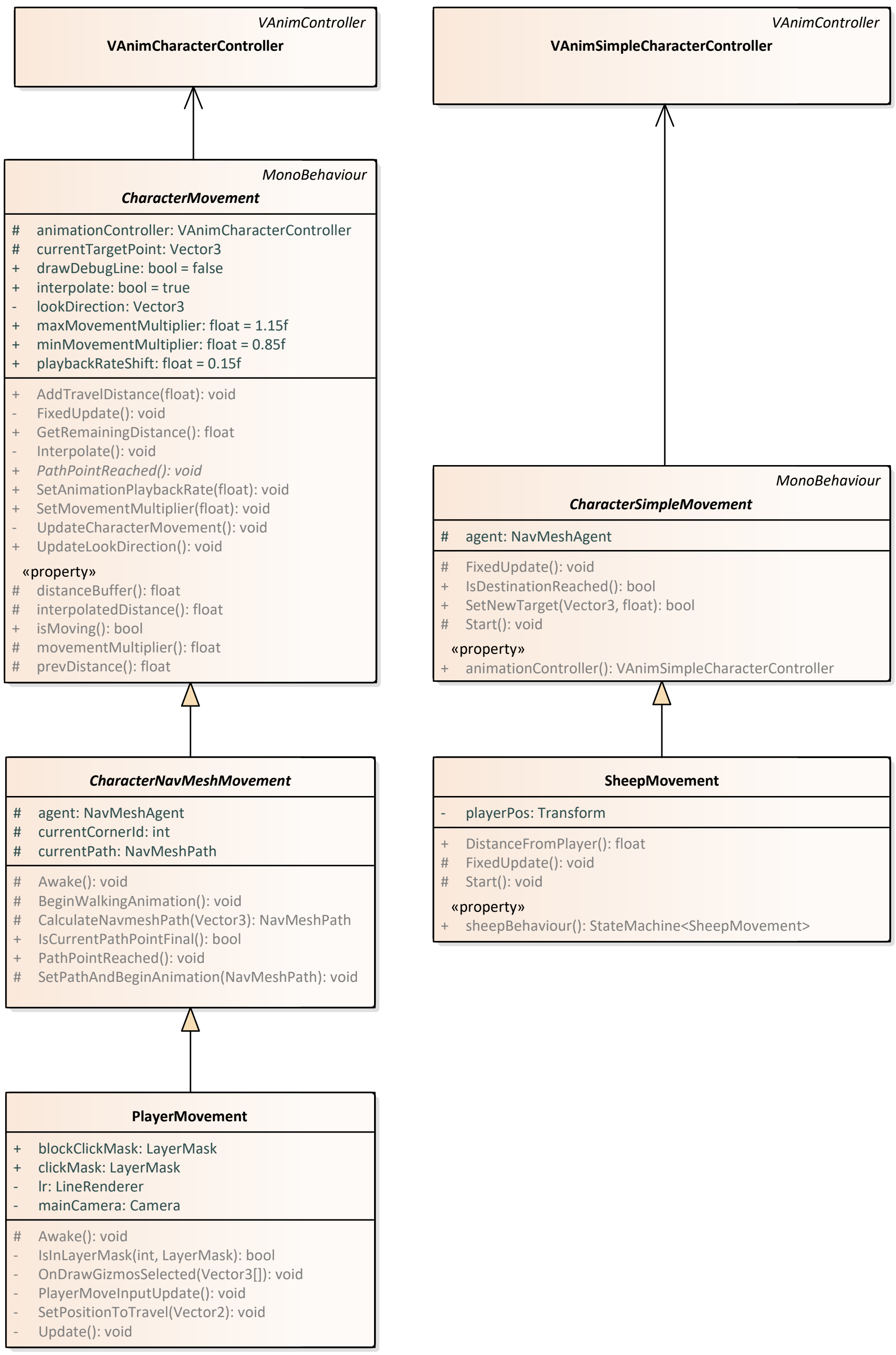

VAnimController
VAnimCharacterController
VAnimController
VAnimSimpleCharacterController
MonoBehaviour
CharacterMovement
# animationController: VAnimCharacterController
# currentTargetPoint: Vector3
+ drawDebugLine: bool = false
+ interpolate: bool = true
- lookDirection: Vector3
+ maxMovementMultiplier: float = 1.15f
+ minMovementMultiplier: float = 0.85f
+ playbackRateShift: float = 0.15f
+ AddTravelDistance(float): void
- FixedUpdate(): void
+ GetRemainingDistance(): float
- Interpolate(): void
+ PathPointReached(): void
+ SetAnimationPlaybackRate(float): void
+ SetMovementMultiplier(float): void
- UpdateCharacterMovement(): void
+ UpdateLookDirection(): void
«property»
# distanceBuffer(): float
# interpolatedDistance(): float
+ isMoving(): bool
# movementMultiplier(): float
# prevDistance(): float
MonoBehaviour
CharacterSimpleMovement
# agent: NavMeshAgent
# FixedUpdate(): void
+ IsDestinationReached(): bool
+ SetNewTarget(Vector3, float): bool
# Start(): void
«property»
+ animationController(): VAnimSimpleCharacterController
CharacterNavMeshMovement
# agent: NavMeshAgent
# currentCornerId: int
# currentPath: NavMeshPath
# Awake(): void
# BeginWalkingAnimation(): void
# CalculateNavmeshPath(Vector3): NavMeshPath
+ IsCurrentPathPointFinal(): bool
+ PathPointReached(): void
# SetPathAndBeginAnimation(NavMeshPath): void
SheepMovement
- playerPos: Transform
+ DistanceFromPlayer(): float
# FixedUpdate(): void
# Start(): void
«property»
+ sheepBehaviour(): StateMachine<SheepMovement>
PlayerMovement
+ blockClickMask: LayerMask
+ clickMask: LayerMask
- lr: LineRenderer
- mainCamera: Camera
# Awake(): void
- IsInLayerMask(int, LayerMask): bool
- OnDrawGizmosSelected(Vector3[]): void
- PlayerMoveInputUpdate(): void
- SetPositionToTravel(Vector2): void
- Update(): void

Figure 4.10: Class diagram describing the movement system structure.

CHAPTER 5

Implementation

This chapter briefly describes the implementational details of both the *Blender* extension and the *Unity* animation components proposed in the previous chapter. It also explains solutions to issues, which have arisen during the implementation phase. Exemplar screenshots of the implementation products are provided. Lastly, the results of the implementation are summarized.

5.1 Blender Addon

All of the features proposed in section 4.3 are implemented to work directly inside of *Blender* using the *Python API* described in section 2.4. These features are fully integrated into the user interface and can be easily installed into *Blender* as an add-on.

5.1.1 Motion Tracking Extension

The extension for exporting the tracking data is located in the *misc* tab of the *movie clip editor*. It allows to export the distance a defined tracker travels during each frame. The distance is in pixels by default, but it can be modified using the *multiplier* input. When exporting, the file is saved into the *output folder* (defined in the compositor extension panel) in `csv` file format.

5.1.2 Compositor Extension

This subsection explains how the requirements for the compositor extension have been met. The user interface is located in *Render properties* tab of the *Properties* editor, split into two panels. The interface is shown in figure 5.2.

Mass Import

The input folder is stored inside of a `StringProperty` with the class variable `subpath` equal to `DIR_PATH`. This property is then displayed in the panel as a folder input field.

When the user selects a folder, all valid movie clip files in that folder are loaded into a custom collection and displayed using the `UIList` type. Each row of the list contains the *id*, *name* and *state* (*unvisited, visited, active*) of the clip record. The state of the clip is visually represented by icons.

The user can freely switch between the clips either using the *Next clip* and *Prev Clip* buttons, or by selecting a clip inside of the list and clicking on the *Switch to Clip* button. Internally, these button call operators which alter the selected clip id and switch to the appropriate scene.

Scene System

When switching between clips, each clip has its own dedicated scene. If switching to an *unvisited* clip, a new scene is created by making a *Full copy* of the scene the user was previously on. The user can then check, whether the previous settings apply for the new scene and if not, they can make adjustments without affecting the settings in the other scenes.

Namespace

The user-defined property *namespace* serves as a batch identifier. The scenes are named in the following format: `namespace/clip-name`. This makes it easier for the user to distinguish between saved scenes and also allows them to create experimental batches without altering the main batch's settings.

The user can enter it in the *namespace* subpanel. The user can also delete all the scenes of a given namespace using the *Delete Namespace Scenes* button, when they are no longer needed. As the button might be considered "dangerous", the user is prompted to confirm the action when clicking the button.

Mask System

Mask of the image can be obtained from the *Matte* socket of the *Keying* node. The user can switch between the image view and mask view using the *Show Mask* button in the *Scene Settings* panel box, which reroutes the connections into the output node.

Using the *Render Mask* checkbox, the user can also define whether they want the mask to be rendered. Internally, the information is stored as a `BoolProperty` inside of every scene independently.

Automatic Frame Range

Another setting stored inside of each scene independently is the *Auto Frame Set* checkbox. If checked, the range of frames to be rendered is set automatically to correspond with the length of the clip. The information about the length of the clip in frames is obtained using *context*, by accessing the `clip` property inside of the *Movie clip* node.

Batch Render

When the user is done editing the clips, they can define the output folder and click the *Render All* button. The script then sequentially renders all of the clips previously visited by the user, using the defined settings.

The script utilizes *callback functions* and an *event timer*, being inspired by [31]. The callback functions are responsible for raising a flag inside of the operator when rendering is done or cancelled. To do so, they are registered to `render_complete` and `render_cancel` handlers.

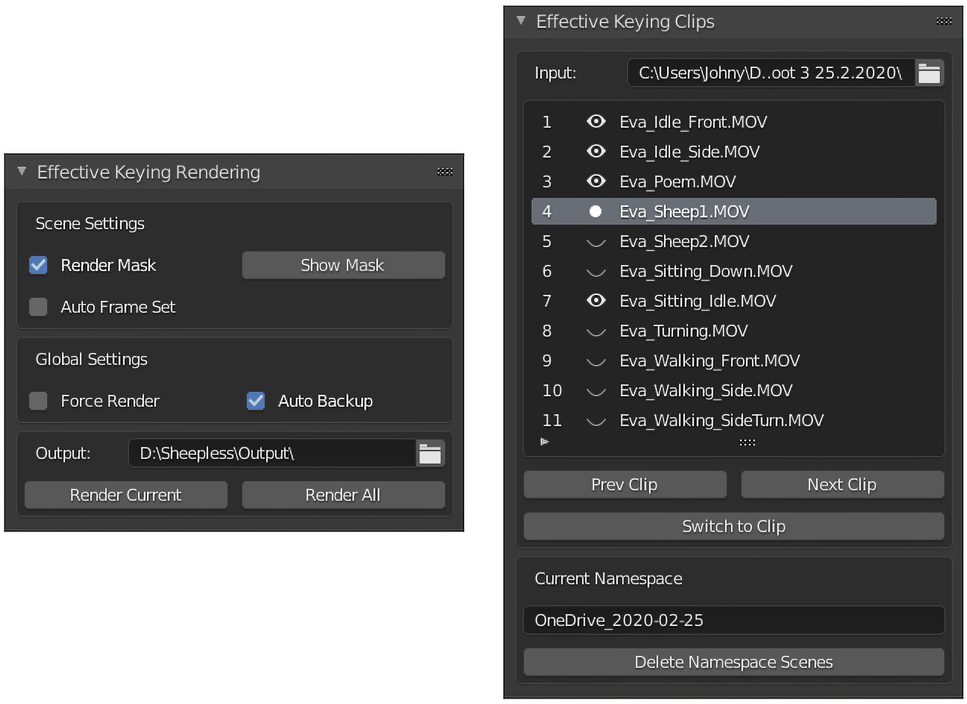
The batch render operator is registered as *modal*, which means it will run in the background and that the *Blender UI* will not wait for it to finish (otherwise it would become unresponsive while waiting). The `modal` function is run every second by being registered as a timer event. It checks for flags being raised by the callback functions and updates the inner logic accordingly.

Output Folder Structure

When rendering a batch of n clips, a folder structure is created inside of the folder defined in the *Output* folder field. The structure is shown in figure 5.1. An empty *keyframes* folder is created for every clip folder, to make later insertion of keyframes easier for the user. By default, if a clip folder with a given name already exists, the rendering for that clip is skipped. If the user wants to instead overwrite the existing files, they can check the *Force Render* checkbox.

- `output-folder-name` .... The output folder defined by the user
  - `clip-name`$_1$ .... Folder created for the 1st clip
    - `video` .... Folder containing the video clips
    - `mask`* .... Folder containing the mask clips
    - `keyframes` .... An empty folder for the keyframes
  - ...
  - `clip-name`$_n$ .... Folder created for the nth clip
    - ...

Figure 5.1: The output folder structure. *Mask folder is created only if *render mask* is checked.


Effective Keying Rendering

Scene Settings

☑ Render Mask — Show Mask

☐ Auto Frame Set

Global Settings

☐ Force Render — ☑ Auto Backup

Output: D:\Sheepless\Output\

Render Current — Render All

Effective Keying Clips

Input: C:\Users\Johny\D..oot 3 25.2.2020\

1	👁	Eva_Idle_Front.MOV
2	👁	Eva_Idle_Side.MOV
3	👁	Eva_Poem.MOV
4	●	Eva_Sheep1.MOV
5		Eva_Sheep2.MOV
6		Eva_Sitting_Down.MOV
7	👁	Eva_Sitting_Idle.MOV
8		Eva_Turning.MOV
9		Eva_Walking_Front.MOV
10		Eva_Walking_Side.MOV
11		Eva_Walking_SideTurn.MOV

Prev Clip — Next Clip

Switch to Clip

Current Namespace

OneDrive_2020-02-25

Delete Namespace Scenes

Figure 5.2: The custom UI panels created for the compositor editor.

5.2 Unity Components

The implemented animation and movement system are showcased in a proof-of-concept game demo. Animation-wise, the demo contains a playable character (controlled by left mouse button), sheep (controlled by simple AI) and a simple stationary animated character (controlled by key input). See figure 5.3 for a screenshot from the game demo.

The playable character is controlled using the more complex animation controller `VAnimCharacterController`, combined with the `PlayerMovement` movement class. It uses the tracking data obtained from *Blender* to directly control the speed of the character. The sheep are controlled using the simpler animation controller `VAnimSimpleCharacterController`, combined with `SheepMovement` movement class.

New animation controllers can be easily implemented by inheriting from the `VAnimController` and defining appropriate *clips* and *states*. The movement system can be also easily expanded to support different control schemes by extending the appropriate movement abstract class. In the rest of this section, a few interesting code highlights are shown.

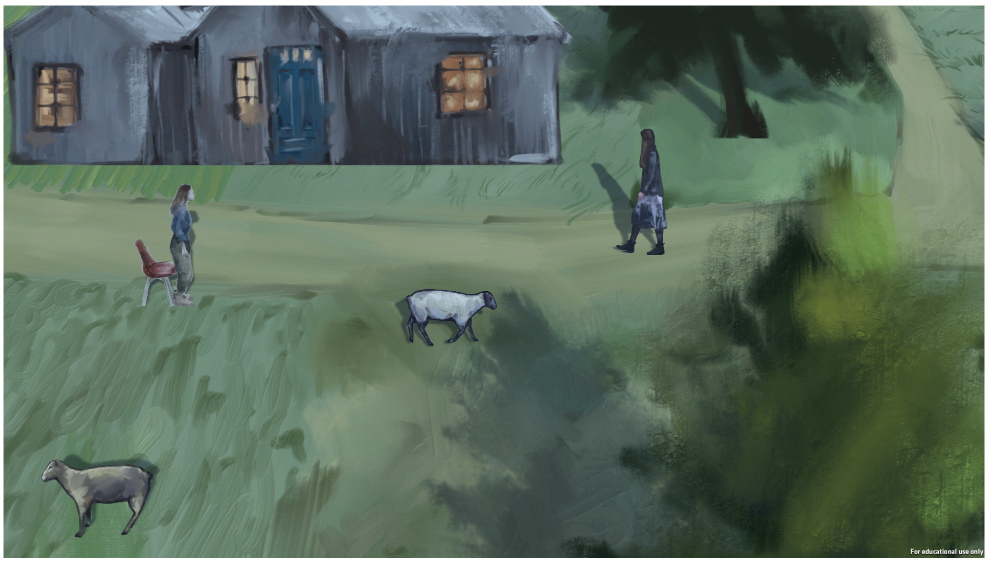


Figure 5.3: Screenshot from the game demo.

5.2.1 Movement Interpolation

An unexpected problem has arisen when implementing the movement system – the animated *GameObject* was noticeably jittering when it was being moved. The reason for the jitter was the frequency of movement update calls. As the more complex movement system is linked directly to the animation playback, the update rate was dependant on the playback rate of the video (one update per frame of the video). *GameObjects* are however expected to be moved consistently during each `Update` or `FixedUpdate` call (update rate of the game loop). To solve this issue, a simple interpolation method has been implemented, which more evenly distributes the movement increments (see listing 1).

5.2.2 Inputting Velocity Data

The `VClipsVelocity4D` clip encapsulation class also contains the logic necessary for parsing the velocity data and saving it to memory. The expected format is a `csv` file with a single column of data containing the velocity data for each frame (the *Blender* extensions outputs the data in this format). The `csv` file is loaded in as a `TextAsset` and parsed using the method described in listing 2.

```
void Interpolate()
{
    var updateRatio = Time.fixedDeltaTime
        * animationController.videoPlayer.frameRate
        * animationController.videoPlayer.playbackSpeed;
    var interpolateVal = prevDistance * updateRatio;

    if (distanceBuffer == 0.0f)
    {
        distanceBuffer += interpolateVal;
        interpolatedDistance += interpolateVal;
    }
    //if the distance buffer is not empty, then a part is
    //taken to balance out the interpolated value
    else if (interpolateVal < distanceBuffer)
    {
        var valueToTake = distanceBuffer - interpolateVal;
        if (valueToTake > interpolatedDistance)
            valueToTake = interpolatedDistance;
        distanceBuffer -= valueToTake;
        interpolatedDistance -= valueToTake;
    }
}
```

Listing 1: Method used for smoothing out the character movement.

```
float[] ParseVelocityText(TextAsset textAsset, ref float sum)
{
    try
    {
        var lines = Regex.Split(textAsset.text, "\r\n|\r|\n")
            .Where(s => s != string.Empty);
        var parsed = lines.Select(line => float.Parse(line))
            .ToArray();
        sum = parsed.Sum();
        return parsed;
    }
    catch (System.FormatException)
    {
        throw new VelocityDataBadFormatException();
    }
}
```

Listing 2: Method used for parsing the velocity data.

5.3 Results and Future Work

All of the results of this thesis are fully available on *GitHub*, split into two repositories, including a user manual – the *Blender* add-on [32] and the *Unity* components [33]. They will also be further utilized and developed during the development of an adventure video game *Sheepless*.

5.3.1 Future Work

In the future, the character animation system could be further improved to support a wider range of movement (for example turning). An editor extension, similar to the official *Unity* animator, enabling visual creation of the animation controllers could also be developed.

5.3.2 Testing

Due to the nature of the project, thorough user-testing was the main form of testing. As the features of the *Blender* add-on are heavily UI-based and tightly bundled with the *Blender* API, the resulting code coverage of the unit tests would be low anyway. This applies to the *Unity* scripts as well, the unit test code coverage would be low due to the nature of the game development platform.

5.3.3 Limitations

One of the drawbacks of the chosen approach to the animation system in *Unity* are the limited navigation possibilities. The characters animated using the more complex animation system (in which animations drive the movement) can utilize the navmesh components and navigate around static obstacles. They are, however, unable to respond to dynamic obstacles, due to their pre-computed animation speed when going towards a target, in order to fully play out the animation. That is linked with the second limitation, which is the difficulty of creating smooth transitions between animations, due to the chosen animation format.

The video animations are also quite resource-intensive, if a high amount of them is played in very high definition. That can be circumvented by playing them at lower definition when viewed from far away and switching to higher resolution when viewed up close.

Conclusion

The aim of this thesis was to create a proof-of-concept game implementation to see if modern image synthesis tools can be used in game development. The process of preparing and transforming the video footage has been analyzed and partly automated using custom *Blender* scripts and other tools. The image format for animations typically used in the *Unity game engine* has been replaced with video files, due to the much more efficient compression of the format. That has introduced a number of problems since the built-in *Unity* animator does not support video file animations. A new solution has been proposed and implemented using the *video player* component. The system has been designed in a way, where the animation playback drives the velocity of the animated object, in order to best preserve the authenticity of the animations in the game world. The animations and movement system are demonstrated in a proof-of-concept *Unity* project.

The techniques and features described will be used for the development of an adventure artistic game *Sheepless*. The source code and created tools along with a user manual are also freely accessible on GitHub [32, 33] for any developers who might be interested.

Going forward, the animation system could be improved to be more easily programmable using UI elements, similar to the built-in animator. A better compromise between performance and file size for the animations could also be designed, which would however introduce far greater complexity to the project.

Bibliography

1. CBS NEWS. *“Loving Vincent” Van Gogh: How the world’s first hand-painted film was made* [online]. CBS Interactive, 2017 [visited on 2020-03-17]. Available from: `https://www.cbsnews.com/news/loving-vincent-entirely-hand-painted-film-about-vincent-van-goghs-life/`.
2. MUSTIATS, Ian. *Sheepless – An Open-source 2D Adventure Game in Unity*. Praha, 2020. Bachelor thesis. Czech Technical University in Prague, Faculty of Information Technology, Department of Software Engineering. Supervisor: Marek Skotnica.
3. BADRONOV, Robert. *Sheepless – An Open-source 2D Adventure Game in Unity*. Praha, 2020. Bachelor thesis. Czech Technical University in Prague, Faculty of Information Technology, Department of Software Engineering. Supervisor: Marek Skotnica.
4. FIŠER, Jakub; JAMRIŠKA, Ondřej; LUKÁČ, Michal; SHECHTMAN, Eli; ASENTE, Paul; LU, Jingwan; SÝKORA, Daniel. StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings. *ACM Trans. Graph.* 2016, vol. 35, no. 4. ISSN 0730-0301. Available from DOI: `10.1145/2897824.2925948`.
5. SÝKORA, Daniel; JAMRISKA, Ondrej; LU, Jingwan; SHECHTMAN, Eli. StyleBlit: Fast Example-Based Stylization with Local Guidance. *CoRR*. 2018, vol. abs/1807.03249. Available from arXiv: `1807.03249`.
6. BURÝŠEK, Jiří. *Example-based Non-photorealistic Rendering using Game Engine*. Praha, 2019. Available also from: `https://dspace.cvut.cz/handle/10467/82729`. Master thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Graphics and Interaction. Supervisor: Daniel Sýkora.

7. JAMRIŠKA, Ondřej; SOCHOROVÁ, Šárka; TEXLER, Ondřej; LUKÁČ, Michal; FIŠER, Jakub; LU, Jingwan; SHECHTMAN, Eli; SÝKORA, Daniel. Stylizing Video by Example. *ACM Trans. Graph.* 2019, vol. 38, no. 4. ISSN 0730-0301. Available from DOI: `10.1145/3306346.3323006`.
8. SCRTWPNS. *EbSynth Alpha* [software] [visited on 2020-03-30]. Available from: `https://www.ebsynth.com/`.
9. BLENDER FOUNDATION. *Blender* [software]. Version 2.81.16 [visited on 2020-02-20]. Available from: `https://www.blender.org/`.
10. ROOSENDAAL, Ton. *How Blender started, twenty years ago…* [online] [visited on 2020-02-09]. Available from: `https://code.blender.org/2013/12/how-blender-started-twenty-years-ago/`.
11. BLENDER FOUNDATION. *Blender 2.81 Reference Manual* [online] [visited on 2020-02-10]. Available from: `https://docs.blender.org/manual/en/latest/index.html`.
12. BLENDER FOUNDATION. *Window System Introduction* [online] [visited on 2020-02-10]. Available from: `https://docs.blender.org/manual/en/latest/interface/window_system/introduction.html`.
13. MATHWORKS. *Linear mapping method using affine transformation* [online] [visited on 2020-04-26]. Available from: `https://www.mathworks.com/discovery/affine-transformation.html/`.
14. PYTHON SOFTWARE FOUNDATION. *What is Python?* [online] [visited on 2020-02-10]. Available from: `https://www.python.org/doc/essays/blurb/`.
15. PYTHON SOFTWARE FOUNDATION. *Should I use Python 2 or Python 3 for my development activity?* [online] [visited on 2020-04-28]. Available from: `https://wiki.python.org/moin/Python2orPython3`.
16. BLENDER FOUNDATION. *Blender 2.57 Release Notes* [online] [visited on 2020-04-28]. Available from: `https://archive.blender.org/wiki/index.php/Dev:Ref/Release_Notes/2.57`.
17. UNITY TECHNOLOGIES. *Blender* [software]. 2019.2.9f1 [visited on 2020-05-03]. Available from: `https://store.unity.com/`.
18. PETTY, Josh. *What is Unity 3D & What is it Used For?* [online] [visited on 2020-05-06]. Available from: `https://conceptartempire.com/what-is-unity/`.
19. UNITY TECHNOLOGIES. *Unity User Manual (2019.3)* [online] [visited on 2020-05-06]. Available from: `https://docs.unity3d.com/Manual/`.
20. UNITY TECHNOLOGIES. *Components for Runtime NavMesh Building* [online] [visited on 2020-05-09]. Available from: `https://github.com/Unity-Technologies/NavMeshComponents`.

21. UNITY TECHNOLOGIES. *Navigation and Pathfinding* [online] [visited on 2020-05-07]. Available from: `https://docs.unity3d.com/Manual/Navigation.html`.

22. YEAGER, Charles. *Everything You Need to Know About Chroma Key and Green Screen Footage* [online]. 2019 [visited on 2020-04-04]. Available from: `https://www.premiumbeat.com/blog/chroma-key-green-screen-guide/`.

23. YEAGER, Charles. *Blue Screen Vs. Green Screen: Which One Do You Need?* [online] [visited on 2020-03-02]. Available from: `https://www.premiumbeat.com/blog/blue-screen-vs-green-screen/`.

24. FREEMAN, Eric. *Head first design patterns*. Beijing: O'Reilly, 2004. ISBN 9780596007126.

25. REFACTORING GURU. *State in Python* [online] [visited on 2020-03-23]. Available from: `https://refactoring.guru/design-patterns/state/python/example/`.

26. FOX, Alexander. *How Modern Video Compression Algorithms Actually Work* [online]. 2019 [visited on 2020-05-08]. Available from: `https://www.maketecheasier.com/how-video-compression-works/`.

27. TECHSMITH. *Video File Formats, Codecs, and Containers Explained* [online] [visited on 2020-05-08]. Available from: `https://www.techsmith.com/blog/video-file-formats/`.

28. UNITY TECHNOLOGIES. *Video Transparency Support* [online] [visited on 2020-05-08]. Available from: `https://docs.unity3d.com/Manual/VideoTransparency.html`.

29. FFMPEG. *FFmpeg* [software] [visited on 2020-05-08]. Available from: `https://www.ffmpeg.org/`.

30. PNGQUANT. *PngQuant – lossy PNG compressor* [software] [visited on 2020-02-10]. Available from: `https://pngquant.org/`.

31. KURZEMNIEKS, Gatis. *User defined render queue* [online] [visited on 2020-01-22]. Available from: `https://www.rendereverything.com/blender-multi-render-script/`.

32. KLICPERA, Jan. Effective Image Processing for Blender. In: *GitHub* [online] [visited on 2020-05-20]. Available from: `https://github.com/HonzaKlicpera/Effective-footage-processing-Blender`.

33. KLICPERA, Jan. Unity Video Animator. In: *GitHub* [online] [visited on 2020-05-20]. Available from: `https://github.com/HonzaKlicpera/Unity-Video-Animator`.

Appendix A
Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
CSV	Comma-separated Value
EOL	End of Life
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HD	High Definition
UI	User Interface
URL	Uniform Resource Locator
VFX	Visual Effects

Appendix B

Contents of Enclosed SD Card

```
readme.txt ............ file describing the SD card contents
├─ manual.pdf ......... file containing the user manual
├─ binaries ........... directory containing the binary files
│  ├─ blender ......... directory containing the Blender add-on starter project
│  └─ unity ........... directory containing the Unity game builds
├─ src ................ directory containing the source codes
│  ├─ blender ......... the Blender add-on implementation sources
│  ├─ thesis .......... directory containing the LaTeX source codes of the thesis
│  └─ unity ........... the Unity implementation sources
└─ text ............... directory containing the thesis text
   └─ thesis.pdf ...... the thesis text in PDF format
```