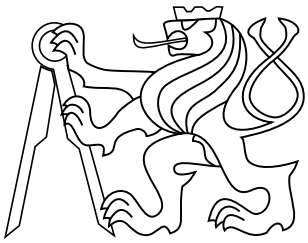
Bachelor Thesis

Modeling and animation of quadrupeds

Julie Trollerová



August 2021

Ing. Ladislav Čmolík, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Computer Graphics and Interaction

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Trollerová** Jméno: **Julie** Osobní číslo: **456196**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Modelování a animace čtyřnožců

Název bakalářské práce anglicky:

Modeling and animation of quadrupeds

Pokyny pro vypracování:

Seznamte se s principy používanými pro modelování a animaci čtyřnožců a porovnejte je s principy pro modelování a animaci postav. Zaměřte se zejména na principy, kdy je výsledný 3D model použit v počítačové hře. Dále se zaměřte na využití jedné animační kostry a jejích pohybů na různé 3D modely čtyřnožců s rozdílnými proporcemi. Detailně zdokumentujte proces úpravy animační kostry a jejích pohybů při aplikaci na 3D model s rozdílnými proporcemi. Na základě analýzy vytvořte alespoň tři modely čtyřnožců s rozdílnými proporcemi. Pro jeden z modelů vytvořte alespoň pět animací (např. chůze, běh, skok). Demonstrujte možnost použití vytvořených animací na zbývajících dvou modelech s jinými proporcemi.

Seznam doporučené literatury:

[1] I. Kerlow, Mistrovství 3D animace, Computer Press, 2011.
[2] L. Skrba, L. Reveret, F. Hétroy, M.-P. Cani, and C. O'Sullivan. Animating quadrupeds: methods and applications. Computer Graphics Forum, 28(6):1541-1560, 2009.
[3] S. Coros, A. Karpathy, B. Jones, L. Reveret, and M. van de Panne. Locomotion skills for simulated quadrupeds. ACM Trans. Graph. 30(4):1-12, 2011.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Ladislav Čmolík, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **02.03.2021** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **19.02.2023**

Ing. Ladislav Čmolík, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studentky

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Assignment

Analyze the principles used for modelling and animation of quadrupeds and compare them with animating humanoids. Focus on such methods when the final product can be used in a video game. Then focus on the application of one animation skeleton and its animations on multiple quadruped models of different proportions. Document the process of the necessary changes made to the skeleton and its animations when applying to the other models. Based on the analysis, create three quadruped models with different proportions. For one of the models, create five animations. Demonstrate the options of using those animations on the other two models.

Acknowledgement

First and foremost, I would like to express my deep appreciation to the supervisor of this thesis, Ing. Ladislav Čmolík, Ph.D., whose expert guidance has helped me immensely through every step of my work. From the bottom of my heart, I would also like to thank my family and my fiancé Barry Hughes for their unwavering support, belief in me, and patience that cannot be underestimated.

Declaration

I hereby declare that this thesis represents my own work. I have referenced all supporting literature and other resources in accord with methodical instructions for writing an academic thesis.

Abstrakt

Tato práce se věnuje otázce modelování a animace čtyřnožců zejména za účelem výroby assetů do videoher. Poskytuje náhled na teorii a metody používané při výrobě takových assetů a navíc prozkoumává možnosti přenosu animací mezi modely čtyřnožců o různých proporcích.

Za použití některých z těchto metod zkombinovaných s prozkoumanou teorií věnující se pohybu čtyřnožců jsem v programu Blender vytvořila tři modely psovitých šelem o různých proporcích se sdílenou sadou animací, které byly vytvořeny na jednom z těchto modelů a následně aplikovány na zbylé dva modely.

Klíčová slova

Animace čtyřnožců, Modelování čtyřnožců, Pohyby čtyřnožců, Přenos animací

Abstract

This thesis is dedicated to modelling and animating quadrupeds, especially to create assets to be used in video games. It provides an overview of the theory and methods used while making such assets. On top of that, it explores the options of transferring animations between quadruped models of different proportions.

Using some of these methods combined with researched theory on quadruped movement, I have created three canine models with different proportions in Blender. These models share a set of animations that I created for one of the models and applied to the other two afterwards.

Keywords

Quadruped animation, Quadruped modeling, Quadruped motion, Animation transfer

Contents

1. Introduction

Animals have been an inseparable part of our lives almost since the beginning of our journey in this world. Archeological discoveries suggest we already had pet dogs as long as 14 000 years ago, with the domestication of many other animal species, such as livestock, following soon after [1]. Humans and animals have lived and evolved side by side for millennia, and therefore it can be no surprise how close we have become. Considering our close relationship with animals, it only makes sense for us to try to bring them to life alongside human characters when telling stories, whether the stories take the form of a painting, a film, a video game, or any other media. The inclusion of animals, or the lack thereof when creating an environment, should be a significant part of the world-building plot. In this thesis, I focus on using animal character in video games.

Over the past decades, we have made massive progress in computer graphics, bringing graphical effects in movies and video games to a level of realism that would have been unthinkable 50 years ago. Thanks to developments in fields such as is motion capture technology, we can create artificial humanoid characters whose every motion seems lifelike and realistic. We are able to animate the way their facial expressions change based on their emotions, the way they walk and move around the environment. We know how to realistically replicate even the subtle movements a person makes when they are idle, to make them feel like actual people on our screens. While it makes complete sense that we have managed to achieve this near perfection for replicating humanoid characters, we still have more to learn when it comes to animating our quadrupedal associates.

Animating quadrupedal animals brings several challenges to the table. Perhaps the most prominent one is gathering data for naturally looking animation. While motion capture is possibly the best way to gather data needed for animating humanoids, it may, for many reasons, not be the way to go with quadrupeds, or at least not all of them. I will further discuss this particular topic in **Section 2.2**

Another major complication of gathering data for the animation of quadrupedal creatures is that many of the creatures we would like to depict in video games do not exist. When building an entirely new organism, we could completely make up how it moves based on its physiology. We could consider various factors such as its lifestyle and what might have been such creatures' evolutionary development bringing it to its current state. Such a method may lead to some unusual, alien-like results. While this may be the way to go when creating strange organisms with no base in any real animals, a different method might be better for creatures that we wish for people to perceive with sympathy and familiarity. In this case, we have the option to base the creature's movement on an existing animal. For example, the motions of the dragon Toothless from DreamWorks *How To Train Your Dragon (2010)* [2] were inspired by owls, eagles, a serval cat, and in certain scenes even a husky dog [3].

The next challenge in creating realistic animations for quadrupeds is how many different quadrupedal animal species there are. Each species's evolution developed based on different factors and chose to deal with their environmental challenges in different ways. That resulted in each species varying in size, anatomy, and behavior, all of which influence how they move, making animating all quadrupeds at once virtually impossible. I will further explore the biomechanics of quadrupedal movement in **Section 2.1**.

1.1. Structure of the Thesis

In order to create realistically animated quadruped models for this project, I will first summarize the physics behind quadruped motion, like the *Dynamic Similarity Hypothesis (DSH)*, different approaches to gathering data necessary for modelling animals, and techniques used to create realistic quadruped models and their animations. Next, I describe the process of creating three quadruped models of varying proportions. To put my research into practice, I create a set of animations for one of these models and explore the possibilities of transferring the animations to the other two models.

1.2. Goals of the Thesis

The focus of this thesis is to find the best approach to modeling and animating quadrupeds. The goal is to uncover the best practices of manual transferal and necessary adjustments of the animated skeleton to other models. I will focus on mainly working manually rather than using automation as this approach gives me the most freedom to explore the best practices. Automating more extensive parts of my work is, while possible, currently above and beyond the subject matter of this thesis.

As mentioned earlier, there is a vast spectrum of quadrupeds. I will focus on canines because there are canines of numerous shapes and sizes. There is also plenty of accessible materials to study their movement.

2. Analysis

In this chapter, I present the current theoretical knowledge and methods used to model and animate quadrupeds in the video game industry. This analysis is necessary for choosing the right approach and the most efficient techniques to help me create realistically animated quadruped models.

Firstly we will look at research done on the subject of the motion biomechanics of real animals. I believe that understanding the anatomy and physics behind quadruped motion is essential for replicating it correctly. The animation methods I will summarize afterward derive from this research at various levels.

2.1. Quadruped Motion Biomechanics

Cavagna et al. [4] attribute the first application of the Dynamic Similarity Hypothesis, or DSH, to quadruped locomotion to the British zoologist Robert McNeill Alexander [5]. According to this hypothesis, the differences between various quadrupedal mammal species' movements depend primarily on their size and body geometry. When the ratios of gravitational force and inertial force acting on the quadrupeds body are equal, the animals will move quite similarly. To describe this, Alexander uses Froude number

$$Fr = \frac{v^2}{g \cdot h},$$

where v stands for velocity $[m \cdot s^{-1}]$, g represents gravitational acceleration $(9.81 m \cdot s^{-2})$ and h stands for characteristic length $[m]$. Cavagna et al. then apply this formula to the quadrupeds movement as follows: the main inertial force is a centripetal force acting on the animal's torso as it moves over its supporting leg (imagine an upside-down pendulum anchored where the quadrupeds foot touches the floor, as displayed in **Figure 1**), h is then equal to the animal's limbs length. Biewener et al. [6] point out that the height of the animal's center of gravity and, therefore, the extent of its limbs determines its potential energy. They also suggests that an animal's kinetic and potential energy ratios are equal at equal Froude number values.

This hypothesis indicates that most cursorial quadrupeds, meaning quadrupeds that have evolved to be good runners, tend to switch between gait types when they reach similar Froude number thresholds [7]. We differentiate five such gait types: walking, trotting, pacing, cantering, and galloping. The first three are symmetrical, cantering and galloping are asymmetrical. While cantering or galloping, left and right limbs are in different phases at one moment instead of mirroring each other. The quadruped favors one side over the other, choosing between left or right lead. Its decision can depend on environmental circumstances or to push back fatigue [8].

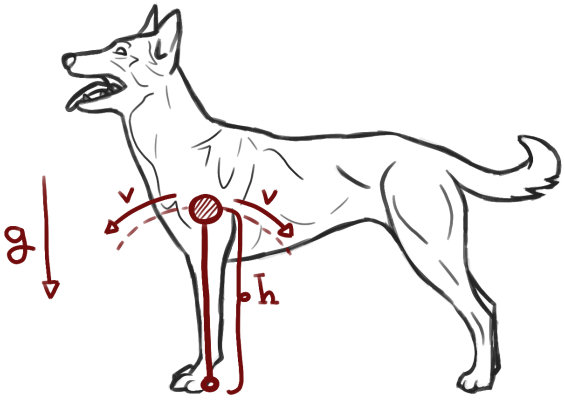


Figure 1 Illustration of Cavagna's description

2.2. Gathering Data

As I mentioned in the **Chapter 1**, collecting data necessary to create realistic animation for quadrupeds can be quite challenging. In my opinion, the most straightforward way of gaining information for animating a quadruped would be merely observing the animal and animating the model based on these observations. Nevertheless, it is not the most precise method. In the rest of this chapter, we will explore a few techniques used in modern computer graphics, especially in the video game industry.

While there have been successes in this area, the use of motion capture on animals is quite limited. If we get over the apparent complications, like attaching markers and required equipment to an animal's, often hairy, body, other complications follow, such as the fact that vast spectrum of creatures displayed in video games that do not even exist in real world. Out of the animals that do exist, only very few species are cooperative enough for us to capture their movement with this technology. It would seem that the only two species that get to act as motion capture actors are dogs and horses. Even then, the way they move around the motion capture studio will still not necessarily be entirely natural as they will walk differently on a concrete floor or a treadmill than they would in nature [9]. That being said, motion capture has proved useful in certain cases, like Riley, a playable dog from *Call of Duty: Ghosts (2013)* [10] played by a German Shepherd Colin, or various canines in *The Last of Us II (2020)* [11]. Meanwhile, in games like *Red Dead Redemption (2010)* [12], we can see the data gathered with motion capture used for animating horses [13].

In an interview [14] at a demo event, Zach Volker, the lead animator behind *Call of Duty: Ghosts*, shared his experience with motion capture a dog named Colin, see **Figure 2**. *"We needed a way to put the markers on [Colin's] body and so we got a spandex suit for him and glued the markers to it,"* Volker explained. *"It worked pretty well, but [Colin] got hot very quickly so we could only work in 20-minute increments and give him a long break to cool off."* He then continued. *"We tried putting these booties on [Colin], with the markers glued on those - they were like these little rubber socks. We were very excited, we put these things on for the first shoot, we let him go on the stage... But then he's walking around like he's on molasses. I thought, 'We are not going to get a single decent shot out of this whole day.'"*

Figure 2 Colin on the set of CoD: Ghosts [14]

Figure 3 Uuno at Remedy [15]

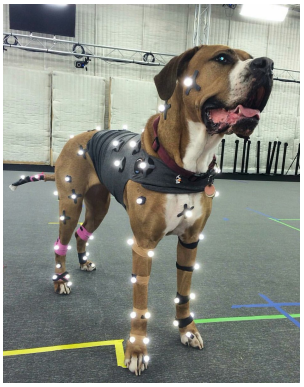


Figure 4 Soba at Ubisoft [16]

From Volker's interview, it is apparent that especially the early stages of using motion capture on animals haven't been exactly smooth, however they managed to push through and create very realistic animations for in-game Special Forces dog Riley. Since then, many other video game studios have also shared photos from their dog motion capture experiences, showing that this method is continually evolving and becoming more comfortable for both sides, animators and quadruped actors. You can see the progress in **Figure 3** and **Figure 4**.

Whenever we require data for animating other quadrupeds, like any wild or non-existent animals, motion capture obviously becomes even more problematic, and we have to look elsewhere. One of those next possibilities, at least when it comes to wild animals that do exist, is capturing motion from video footage. For virtually any animal on this planet, there is a vast amount of documentaries displaying it in its natural environment. We also typically do not have to worry about the animal's well-being, which we can not necessarily ensure for an encaptured animal. Usually, however, they only show footage from one viewpoint at a time, making a straightforward conversion into a 3D motion model impossible [9].

Wilhelms and Van Gelder [17] have come up with a way, how to overcome this problem using active contour model. Active contour model is a technique developed by Demetri Terzopoulos et al. [18] used for tracking familiar objects in a noisy 2D footage. This technique, also called 'snakes,' uses a spline pulled toward lines and edges through external constraint and image forces. They have presented their technique to obtain a quadruped movement on footage of a horse. After creating a 3D model of a horse skeleton, they scaled and aligned it to the horse in the video and attached the active contours of the horse's body to it. As the horse moves, the active contours and skeleton model move with it. Depending on the background behind the quadruped and the footage's quality, it is sometimes necessary to reset the active contours [9].

While acquiring motion data of whichever quadruped we wish to animate can make the animation process a much easier or more complicated task, the data's origin does not necessarily matter. Unfortunately, I do not currently have access to motion capture to obtain the data this way. Therefore I will create my models and animations manually based on freely accessible footage of canines such as photos and videos as well as real-life observations. Fortunately, this does not matter when it comes to the primary topic

of this thesis, which is to study the transferal of animations created for a skeleton of specific proportions on other models of different proportions.

2.3. Modelling Quadrupeds

While the deepest roots of 3D modelling go as far as the third century BC and the "founder of geometry" Euclid, this field's progress has been the steepest over the last decades, thanks to computer graphics [19].

In 1963, an American computer scientist Ivan Sutherland introduced Sketchpad, software that first proved that computers could be a tool for interactive graphical design. Together with his colleague David Evans, they have not only founded the first and famous 3D graphics company, "Evans & Sutherland", but also opened the first-ever department of computer technologies at the University of Utah. Having a department dedicated to computer technologies and graphics has brought together many people that would help develop this industry, like Edwin Catmull and James Clark, to name a few [20].

A 3D model is a mathematical representation of an object in a three-dimensional environment. In computer graphics, we usually represent only the surface of a model, meaning the visible part, rather than the inside. There are several properties models suitable for videogames should have. Usually, we want our models to be closed and manifold. As I already mentioned, the model is technically a surface, which by definition is an orientable two-dimensional manifold existing in a three-dimensional space. A two-manifold mesh is a mesh in which a small enough circular neighbourhood of each of its points is topologically equivalent to a disc. We say a mesh is orientable when every two adjacent faces have consistent orientation, meaning we can tell the inside and the outside of a closed model.

2.3.1. Creating a video game model

While the subject of my thesis is modelling for a video game, I would now like to take a moment and describe the differences between video game models and models used in other media, such as animated movies. At first glance, the differences between 3D models used in movies and video games are not really that obvious. Modelling for video games and movies often share the same or similar techniques and tools. However, the result models could hardly be exchanged between the two media. The first and perhaps most significant difference between the two is their polygon count.

The more polygons a model has, the more detail it can have and, therefore, generally, the better it looks. However, the number of model's polygons also increases the file size and rendering time. Neither of these drawbacks really matter in movies, but both can become significant issues in video games. For example, a frame in the Monsters University [21], an animated movie from the year 2013, took on average 29 hours to render [22]. At the standard framerate of 24 frames per second, that is a lot of rendering, not to mention that Pixar Animation Studios at the time had one of the top 25 "supercomputers" in the world, consisting of 2 000 computers with overall more than 24 000 cores [22]. Granted, with a data center like that, combined with the fact that while creating

a movie, each frame can be re-rendered and edited multiple times to perfection before using it for the final product, the artists working in film industries do not have to worry about many hardships that video game artists must take into consideration.

Unlike movies, the main feature of video games is, without a doubt, their interactivity. This means that within different playthroughs, the crushing majority of frames happening at the same time in the game will never be exactly the same. Therefore, while video games can on a few occasions still use clips that have been pre-rendered and imported into the video game as a finished video clip, such as cutscenes, most of the rendering must happen in real-time, and in today's day and age, it still cannot be outsourced to a supercomputer far away, but it has to be done by the sole machine the video game is running on. Because of this, video game developers have to constantly compromise between creating detailed models and considering hardware limitations, trying to make video games look as well as possible by using breathtaking visuals while still running smoothly.

To ensure a game's scene does not contain more polygons than how many can be rendered in real-time, projects usually have a so-called *polygon budget* defined. This budget depends on the hardware available as well as on the scene's complexity and desired rendering quality. There are multiple ways of lowering the number of polygons in a scene and making the processing easier. The first obvious method to lower the overall amount of polygons in a scene is to create low-polygon models. While there are several extremely successful video games that operate with very few polygons, low-poly environments hardly offer enough opportunities to keep the user entertained and happy with the game.

Thankfully there are ways to make a somewhat low-polygon model seem higher quality than it is. The obvious way to achieve this is to use proper lighting and different kinds of textures, such as normal maps that can be generated from a more detailed version of the model and mapped onto the lower polygon count model. This technique results in the lower quality model at first glance looking like it is a higher quality. However, on its edges, the user will still see it is not as detailed.

Another common strategy to decrease the number of polygons in a scene is changing the level of detail depending on how far from the camera is the object. The closer it is, the more detailed version of it is used. Then as it gets further away and becomes smaller on the screen, we use less and less detailed versions, reducing the number of polygons in the scene.

That being said, with both hardware and software constantly improving, video games can become more and more demanding and models more detailed, making polygon count not quite as restrictive as it had been before. While there still is a limit and a polygon budget that video game developers have to stick to, it is steadily rising, giving developers more room to focus on other things, such as realistic lighting, shaders and physics.

In 1996, a video game developer studio id Software developed Quake [26], which went down in history as the first first-person shooter video game in a fully 3D modelled environment. Running on the Quake Engine, a scene containing whole 200 polygons could

Figure 5 Quake, 1996[23]

Figure 6 Doom Eternal, 2020[24]

Figure 7 Unreal Engine 5 demo[25]

be rendered real-time [27], as seen in **Figure 5**. In 2020, the same studio developed Doom Eternal [28], running on id Tech 7, a game engine capable of rendering 80 to 90 million polygons in one scene as seen in **Figure 6**. In May 2021, Epic games [29] has released Unreal Engine 5 [30] in early access, with a full launch expected in early 2022. Unreal Engine 5 contains several new features, such as Nanite, an engine capable of automatically handling levels of detail and rendering scenes containing billions of polygons, which are often as small as a single pixel, in real-time. This promising technology could let artists working in the video game industry focus on creating breathtaking visuals without having to worry about polygon budgets. In **Figure 7**, you can take a look at a near-photorealistic screenshot taken out of the Unreal Engine 5 reveal demo. However, according to the official documentation [31], Nanite does not currently support mesh deformation and is therefore restricted to translation, rotation and non-uniform scaling of rigid meshes. At the moment, Nanite meshes also cannot be assigned any translucent values.

Such technological progress might prove to be game-changing when it comes to designing and creating video game assets. However, in my opinion, it further extends the room for video game developers to become careless when it comes to optimization, resulting in video games requiring only the newest technologies to run them. While these techniques may grow obsolete in the future, let us look at the two main methods used to reduce the size and complexity of video game assets I have already mentioned - *normal mapping* and *level of detail*, in **Section 2.3.3** and **Section 2.3.4**, respectively. Before that, let us first take a look at the structures video game developers use to represent

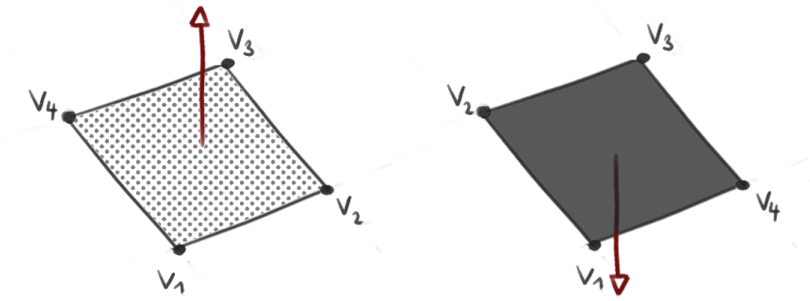


Figure 8 Face normal vectors

models in computer graphics.

2.3.2. Polygonal mesh data structures

The most common method used for model representation in computer graphics is polygonal mesh. It is reasonably intuitive and a favourite among video game developers. As the name suggests, a polygonal mesh is a structure consisting of polygons. Polygon is an n-sided shape, defined by n vertices and edges. In computer graphics, we usually use triangles and quadrilaterals or quads for short. Most of the time, we also call them faces rather than polygons [32]. While this representation is not perfect for all objects, e.g., a sphere, which we can only approximate with a mesh consisting of flat surfaces, it has its benefits. Creating and editing a polygonal mesh is very intuitive and easy to do, especially with the help of one of the many available software, like Blender, Maya, or ZBrush, to only name a few. Another advantage of using polygonal surfaces is that computer graphics hardware manages to process it quite well, particularly if it consists of triangles. Because of this, most software converts models into triangle meshes before rendering them [33].

The direction of a face is a vital piece of information, as it can determine its visibility. To save processing power during rendering, most game engines do not render the back faces unless explicitly told otherwise. The order of vertices defining a face is what determines its direction, and we represent it with a vector, which we call the faces normal vector [32]. As mentioned in the previous section, the normal vector is perpendicular to the face, and it points in the same direction as its front side, as illustrated in **Figure 8**. It also determines the direction of light reflecting off of the face.

When representing a polygonal surface mesh, we have multiple options as well. We can decide between face-based or edge-based data structures as well as a combination of the two. Let us look at the face-based structures first. The most straightforward way is a so-called *polygon soup*. We store the position of each vertex of a face we wish to represent, which causes a significant amount of storing redundant data since the vast majority of faces share vertices. This structure also does not store any information about the faces' connectivity, which is a substantial drawback for the usage in modelling.

To reduce the data replication, we can use an indexed face set, also known as a shared-vertex data structure. In this case, we store an array of vertices and encode the polygon faces as sets of indices of values in this vertex array. Compared to mesh

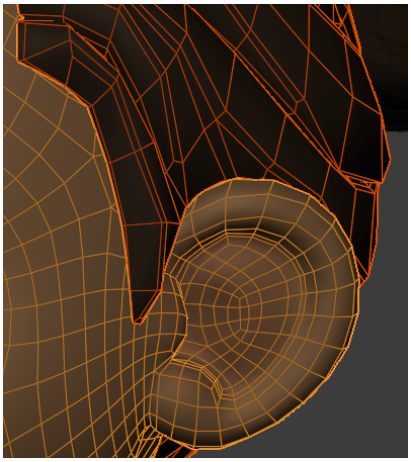


Figure 9 A detail of a model without subdivision surface

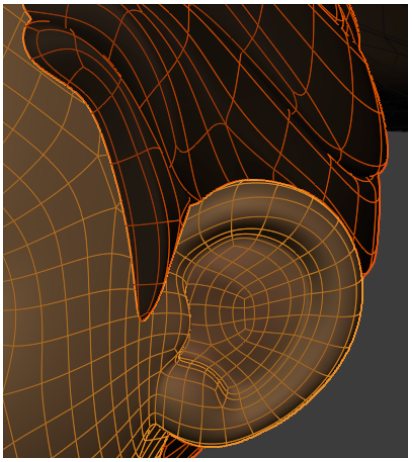


Figure 10 A detail of a model with subdivision surface

soup, this can save on average 1/2 of memory used on storing mesh data. Such storage efficiency, in combination with the structure's simplicity, makes it a viable option used in file formats such as OBJ or OFF. It is also suitable for rendering static objects [34]. If we wish to achieve efficient and quick traversal of a surface mesh using face-based data structures, we have an option to use a structure that for each face stores the information of its vertices, as well as references to its neighbor faces [34].

While face-based structures are suitable for rendering, as I mentioned before, the lack of connectivity data makes them inadequate for modelling and using techniques such as subdivision surfaces, which I will touch on in the next section. Storing mesh connectivity information is easier to do with edge-based data structures, such as winged-edge. In the winged-edge structure, each edge possesses references to the two vertices on its ends, the two faces it connects, and the previous and next edges of those faces. This is an acceptable amount of data stored per each edge, and the mesh traversal is more manageable than with face-based structures. However, it could still be better, with the help of so-called *halfedge-based* data structures [34].

Halfedge structures split each edge into two oriented half edges that head in the opposite directions. Each halfedge has a reference to its ending vertex, its adjacent face, its predecessor, successor, and inverse halfedge. These halfedges are oriented counterclockwise considering the face they are bordering. This way, each halfedge indicates a different face corner and allows for additional data such as normals or texture coordinates to be stored in each corner [34].

2.3.3. Subdivision surfaces

When it comes to parametric surface representations, it may be a complex assignment to generate one. However, it makes some 3D tasks relatively straightforward — for example, adding more vertices directly by sampling the domain and evaluating the parametric function [33]. I will now focus on a parametric surface representation method often used in computer graphics and 3D modelling, Subdivision Surfaces. In 1978 in the journal 'Computer Aided Design', two similar articles came out presenting

subdivision surfaces. E. Catmull and J. Clark wrote one, D. Doo and M. Sabin the other. Both teams were building on an algorithm presented by G. Chaiking [35] in 1974 at a conference at the University of Utah.

Subdivision surfaces have a rough control mesh that defines the surface shape. This control mesh gets repeatedly subdivided, and the vertices get position according to averaging rules [35]. Thanks to subdivision surfaces, we can create smooth objects at a relatively low processing and meager man-hour cost, making it is one of the best methods to use when modelling smooth surfaces. See **Figure 9** and **Figure 10** as an example of a model detail without and with a Catmull-Clark subdivision surface modifier. In the examples you can see a detail of a model Rain v2.0, released by ©Blender Foundation [36].

While using subdivision surfaces can make the modelling process easier for the artist, game engines often require the representation of the models to be polygonal. Fortunately, converting subdivision surface to polygons is no difficult task. In Blender, we can directly apply the subdivision surface modifier. In Maya, we can easily convert the subdivision surfaces to polygons either by using tesselation to create a considerably dense polygon surface matching the contours of the subdivision surface or a simpler polygon mesh matching the subdivision surface control points, rather than its contour [37].

2.3.4. Normal mapping

Normal mapping is a technique allowing us to mimic details in a model that are not there. Using this method can save a substantial amount of polygons that would otherwise have to be processed, consuming valuable processing power and time. The normal vector of a surface is a vector perpendicular to the surface, used to calculate the reflected light and, therefore, the color of the surface. A 2D texture map is an image providing attributes to a models appearance, such as color, transparency, or shininess. A normal map is an RGB texture image containing values that assign the coordinates of the otherwise perpendicular surface normals.

There are different approaches when it comes to encoding the normal coordinations into normal maps [38]. The easy choice would be to use the object space, just like the object's surface geometry does. We can recognize object-space normal maps by their colorfulness - they usually contain all the colors of the rainbow. While relatively simple, the use of the object space to store the normals brings certain shortcomings. The object-space normal maps become tied to the geometry, including its orientation, and they cannot be easily reused for another model, or even a different part of the same model, where the surfaces orientation differs. Another problem arises when the models surface deforms, causing the geometric normals to change. While there are ways to compensate for these shortcomings of object-space normal maps, such as using specialized shaders, most artists choose to use tangent space to store the normal maps, especially when it comes to video game assets that deform, such as animated characters.

Tangent space is a coordinate system that is tangent to the model's surface. To define such space, we require three vectors. One of them is the normal vector of the face. The next vector we need is a tangent, a vector parallel to the face's surface. There is an

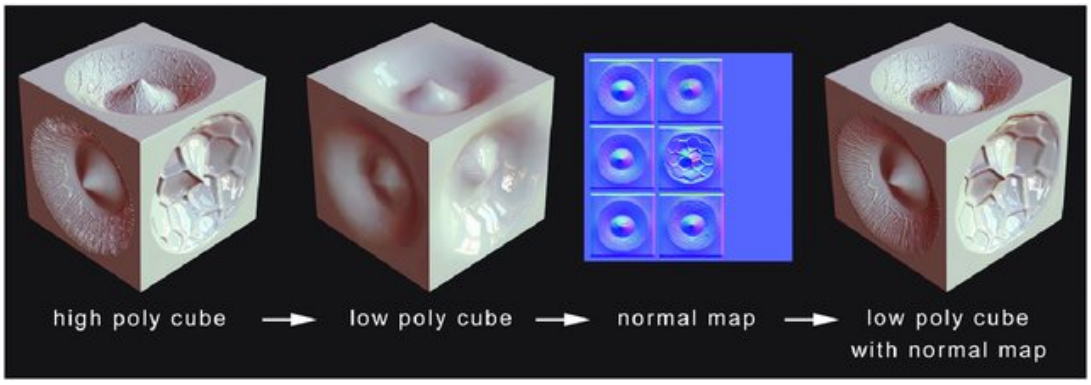


Figure 11 Normal mapping workflow [40]

infinite amount of such vectors, so for the sake of consistency, the tangent is usually chosen to correspond to the directions of texture coordinates. To complete the coordinate system, we also need a bitangent, a third vector that is also a tangent of the surface, generally chosen as the tangent perpendicular to both the normal and the tangent vectors. Tangent-space normal maps have a typical violet color that corresponds to the coordinates of the up-vector. The values saved in the texture are halved, and 0.5 is added to them, which is why the (0, 0, 1) or up-vector is violet in the texture, rather than blue, as it matches RGB(0.5, 0.5, 1) [39].

In other words, the colors of a normal map define how should the normals of the surface be modified and in which direction should the surface reflect light at each point. Using normal mapping is fairly straightforward and can save a lot of time when creating realistic video game assets. As I mentioned, normal mapping is a common technique used when video game developers need to keep the polygon count of the video games assets to a minimum. We can achieve this by generating a normal map from a highly detailed model that looks good but has too many polygons to be used in the videogame and applying this normal map onto a low polygon count version of the same model. This way, we can make the low polygon model look almost as good as its high polygon version, at least from most angles. Since normal maps do not change the model's geometry, the edges of the model remain sharp, and it is usually fairly easy to see where the polygons connect. In **Figure 11**, you can see an example of the workflow used to create a high detail looking low polygon asset.

2.3.5. Level of Detail

Another way to decrease the polygon count of a video game scene is to control the models *Level of Detail*, or *LOD*, based on metrics such as the distance of the model from the camera or its speed relative to the camera. The user will not notice the changes to models that are further away from the camera or those that move fast, and therefore such models do not have to be as detailed as the focus of the scene. Lowering the level of detail of such objects can save a substantial amount of processing power required to render the scene.

Video game developers have the option to choose from two main approaches when implementing the level of detail changes. One is to have several versions of each model varying in levels of detail. We call this method Discrete *Levels of Detail* or *DLOD*. This method is based on switching between the models in real-time as necessary. While easy

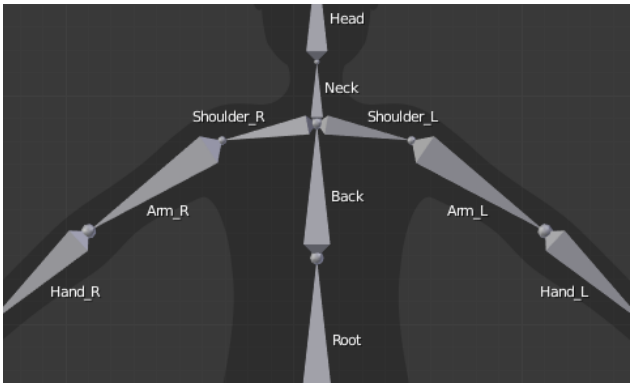


Figure 12 Humanoid animating rig [42]

to implement, this method may produce so-called *popping*, a visual effect that occurs when we suddenly switch between two models with vastly different levels of detail. Popping can be reduced by methods such as alpha blending.

As the name suggests, alpha blending is a method that blends the shift between the two models. As we approach the value at which the LODs should change, the initial version of the model is completely opaque, and its alpha value is set to 1.0, while the new version is fully transparent and its alpha value is set to 0.0. As the transition occurs, the first model becomes more transparent and the second one more opaque until their role switch completely and only the new model end up being rendered. This means that in mid-transition, both LOD models are rendered semi-transparent, which is rather counterproductive when we remember the fact that the main reason we use LOD is to reduce the required processing power. Therefore, it should be avoided during processing power-wise demanding scenes.

2.4. Animating Quadrupeds

When animating quadrupeds, it is crucial to consider both: footage of the quadruped motion we wish to model and research done on its motion biomechanics. However, the result of animating a quadruped will depend on how we decide to approach the creature's animation process as well. In this section I summarize the basics and a few techniques we can use when animating a quadrupedal creature. Combining more of them to achieve a more realistically moving quadruped model is not unusual either.

2.4.1. Rigging

When it comes to animating, virtually all methods used in today's computer graphics build on skeletal animation. Skeletal animation, sometimes called rigging, is an animation technique for which we require our model to have two parts: a mesh representing the model we wish to animate and an animation skeleton we use to deform the mesh [41].

The *animation skeleton*, example of which you can see in **Figure 12**, is a hierarchy of so-called *bones*, and its primary purpose is to make deforming a mesh that can be rather

complex easier to manage during the animation process. Despite the nomenclature, this animation skeleton is usually not as detailed as the actual skeleton of the quadruped we chose as a reference, as most motions do not require such a level of precision. Generally, we try to keep the number of bones to the minimum number needed for the level of detail we wish to achieve. If we use too many bones, it might unnecessarily complicate both the calculations required for animations and the process of creating key poses defining the animation. Each bone has its specific position, orientation, scale, and possibly a parent bone. The parent-child relationship between two bones makes transformations easier, and the overall transformation of a child bone will consist of both its and its parent bone transforms. We set the skeleton's transforms over time using some animation controller [43]. In Bender, bones are vectors defined by head and tail coordinations. If we set the head and tail to the same coordinations, the bone will disappear.

The bones in an animation skeleton are by default rigid. However, most modelling software offers the option to use flexible bones as well. In Blender, we can make bones flexible with the use of Bendy Bones, or *B-Bones* [44]. These flexible, or bendy, bones are helpful, especially when we need a longer flexible segment that we would otherwise have to represent with a long chain of small bones, such as the spine of a creature, or any flexible objects, such as wires. We can set the number of segments we would like the bone to simulate in Bone Properties. Blender then handles the bone as if there was a section of a Bézier curve passing through its segments, influencing their position and rotation to match the curve.

We also have the option to choose between using connected and disconnected bones. While using connected bones may have some advantages in some instances, such as the fact that we can only rotate the child bone rather than translate it. Using connected bones is an easy way to stop accidental translations, and unrealistic deformations stop from happening. Connected bones also make the overall deformation of the mesh more continuous, while using disconnected bones can result in a more rigid look with sharper bends. To achieve maximal realism, the skeletons I use with my models contain both connected and disconnected bones, depending on their placement and purpose.

There have been successful attempts at creating algorithms that automatically generate a skeleton for a specific creature, in our case quadrupeds, making the process easier for the animator. These pre-made skeletons usually contain the finished hierarchy with the appropriate number of bones as well as their settings, such as their relationships and connectivity. The shape of the skeleton and its bones' placement is also generally relatively apparent, making it easy enough for us to scale it to fit the mesh we wish to animate with the help of the skeleton.

2.4.2. Skinning

As I mentioned earlier, the skeleton defines the deformation of its surface mesh. The basic idea is to have every bone affect various vertices differently depending on their position. We bind each bone with the surface mesh vertices and assign it a certain weight to determine how the bone influences the connected mesh, which we'll record into a co-called *weight map*. For example, in **Figure 13** and **Figure 14** you can see the weight maps of two sequential bones in a forearm of Blender's Rain v2.0 model [36].

Figure 13 A weight map of medial part of a forearm

Figure 14 A weight map of the lateral part of a forearm

This process is also called *skinning*. Each vertex can be affected by multiple bones at different levels, making it easier to avoid unnatural looking mesh deformations. While some state-of-the-art graphical engines do this whole process of assigning vertices and weights to bones automatically with the help of a shader program, there are several ways to achieve this manually [41].

The most common way is using so-called *envelopes*. We can imagine envelopes as a field of influence around every bone. For every envelope, we can set a *falloff range*, at which the weight of the bone will decrease to zero, making transitions between bones smoother. Vertices that are inside multiple envelopes get weighted according to all envelopes that contain them. Envelopes are an adequate method for weighting a model, but they are not very precise and usually require touchups in areas influenced by multiple bones.

A technique more suitable for precise weighting is to paint a *weight map* of each bone on the mesh. This method may require a little more artistic skill; however, it gives us better control and makes it easier for us to create weight maps quicker, especially over problematic areas where fields of influences of different bones meet. If we encounter elements that we fail to control using previous methods, most interfaces have an option to enter exact weights for each vertex by number. Using this method for skinning the whole model would be unintuitive and tiresome work.

I have decided to try out Blender's automatic weight map calculations when skinning my models with relatively satisfactory results. Certain problematic areas required manual adjustments, such as the head, shoulders, and hips. Still, I found this option to save me a substantial amount of time I would otherwise have to spend on manual skinning of my models.

2.4.3. Animation methods

The first method used to animate a model skeleton we will look at is *forward kinematics* or *FK*. In forward kinematics, when calculating the position of the last segment of a skeleton after a transform (e. g., the position of front legs paw after rotation in the shoulder), the animator needs to provide the position and orientation of the animal's

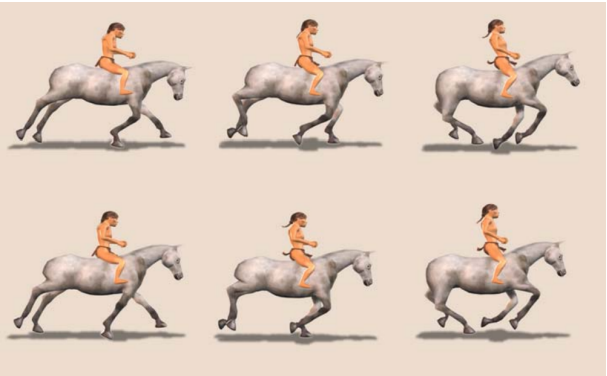


Figure 15 Kuhel's results for galloping [47]

body, shoulder, elbow, and wrist [45]. Presenting all this data can be quite cumbersome. That, as well as the fact that this technique is not incredibly intuitive, is why we usually use *inverse kinematics*, or *IK*, instead [46].

Inverse kinematics does not calculate the position of the skeleton's last segment based on the angles of joints leading to it but computes the joint angles based on the last segment's position. Therefore instead of attempting to predict what angles each joint should have so that the result would look like a natural motion, we can position the last segment where we want it and do not have to worry about values in the joints leading to it. For each joint, we can also set constraints for its range. Constraints corresponding to realistic anatomical angle ranges of joints help prevent strange injury-resembling animation errors we have to otherwise correct manually.

A different approach to computer animation is using physics-based simulation. Unlike the traditional kinematics-based methods, this approach lets all motion of a character result from physics-based simulation rather than manual manipulation of its parts. While physics-based animation of non-living objects, like water or cloth, has been routine for many years, there are several challenges when it comes to using this approach on animating living creatures [48]. Firstly, we cannot control the character we are animating directly, as we do with kinematics-based animations. To get movement, we must alter the forces acting on the model, rather than defining the motion directly, which is more complicated than using kinematics-based animation methods. Also, ensuring that characters move according to physics laws makes their motions physically plausible, however, does not ensure biomechanical realism, and the animations might still require alterations to look natural.

While using purely physics-based techniques to animate quadrupeds may not be worth the trouble, adding physics-based elements to otherwise kinematics-based animations is a common and valuable compromise. Wilhelms and Van Gelder [17] used physics-based animation to achieve realistic shaping of an animal's skin depending on the shapes of muscles contracting beneath it [9]. Rahgoshay et al. [49] introduced *inverse kinodynamics (IKD)*. This workflow defines the character's state as a result of a kinematic state physically simulated over a short time period.

Most animations we see in today's videogames and other media have been manually created by animators using their preferred techniques. Bringing a character to life with

animations is always a lengthy matter, even for veteran animators, and that is why there are significant efforts put into the possibility of automating this process at least partially. Kuhnel [47] managed to automatically generate some predictable aspects of an animation of a person riding a horse. You can see some of her results in **Figure 15**.

2.5. Importing assets from Blender to Unity

2.5.1. Unity

Unity is a cross-platform game engine and *integrated development environment (IDE)*, or integrated development environment, that has been first released in July 2005 by Unity Technologies [50]. Developers can use Unity to create 2D and 3D videogames, simulations and other experiences. Thanks to its support across many platforms, it is one of the most used game engines today. According to their official web page, in 2020, five billion applications that have been built with Unity were downloaded per month. Its software is relatively sturdy, the environment user-friendly, and creating video games in Unity is completely free unless your company made a gross annual revenue of more than 100,000 USD in the previous fiscal year [51]. These are some of the reasons why it is a favourite development tool used by many independent, or *indie*, game developers. They are also the reasons why I have decided to use Unity to demonstrate the results of my thesis as well.

2.5.2. Importing files

It is relatively common for game developers to use the combination of Blender and Unity, and therefore the import of assets from Blender to Unity is quite simple. However, there are a few details that are good to know.

The first obvious question is which format should one use when exporting assets from Blender to Unity, the main two contenders being Blender's native file format *.blend* and *FBX*. Naturally, the .blend file stores everything supported by Blender. However, it can only be opened in Blender. FBX, or Autodesk Filmbox format, is a file format owned by Autodesk [52]. Many video game developers and filmmakers use this format due to its support of all necessary data, such as models, bones, animations, weight maps and much more. Even though Autodesk owns the FBX format, many non-Autodesk applications still support importing FBX files. It also utilizes a binary format for storing data, making it fast and efficient both work and space-wise [53].

Unity supports the import of both of these, and they each have their pros and cons. When importing .blend files, Unity uses Blender's export scripts to create a hidden FBX file before the import itself, which means it requires Blender to be installed on the computer it is running on. Since this may not be the case for all Unity users, it might be a good idea to import assets that have already been converted to FBX, especially when we are unsure who will attempt to run the Unity project. Importing FBX files is also preferable in cases in which a single Blender file contains multiple objects. In such cases, it is easier to export each of these objects as a separate FBX asset. Importing native .blend files, however, can save a reasonable amount of time, as

the asset imported to Unity will update hand in hand with the original .blend file upon further modifications in Blender.

When it comes to preparing the assets for export to Unity, several steps are good to follow. The object's rotation should be set to 0°, and its scale should be set to 1,0 on all three axes. The object's origin should also be set to a consistent, suitable location, like in the middle bottom. Blender offers an easy way to export an asset to an FBX file. The import of the assets to Unity itself is a rather uncomplicated process. We simply need to move or copy the .blend or FBX file to the Assets folder in the Unity project.

3. Modeling and animating a dog

In this chapter, I summarize the process of how I proceeded while modelling and animating three quadrupeds of different proportions. As I mentioned in **Section 2.1**, the way different quadrupedal species move depends on various factors such as their size and proportions. This fact implies that attempting to apply the same animations on very diverse quadrupeds would not result in realistically moving models. Pursuing the idea of creating animations and implementing some sort of algorithm to recalculate the motion based on additional factors might be an interesting subject for further study. However, as mentioned in **Chapter 1**, I have decided to focus specifically on canines for the course of this thesis. I have concluded so for multiple reasons, such as that there is plenty of accessible footage of their movement. There are also many different, real and fantastic, breeds and kinds of canines that differ in sizes and proportions; nevertheless, all of them still move in relatively similar ways.

As my first and primary model, I have decided to create a model of a Belgian Malinois. I picked a Belgian Malinois as a tribute to Cairo, a US Navy SEAL working dog you can read about in the book No Ordinary Dog, written by his handler Will Chesney [54].

To explore the options of transition of the animations I have created on the first model to different models, I made two more models with different proportions. The second canine I modelled is inspired by the Jack Russell Terrier breed. It is a small breed with short legs and a relatively long torso. The third model, on the other hand, represents a warg-like creature. A warg is a large fantastical canine with longer legs and a different posture, its head positioned in front of its torso, rather than above it, as is the case with the first two models. I have picked a warg to represent the fact that in the video game industry, artists often need to create assets depicting creatures that do not exist. With these three models, I think I managed to choose quadrupeds whose proportions vary enough to challenge the process of transferring the animations on various quadrupeds without making it an impossible task.

3.1. Software

The software I have decided to use for modelling and animating my video game assets is Blender, currently in version 2.91. Blender is a free and open-source computer graphics toolset developed by ©The Blender Foundation [55]. I have also already worked in Blender before, and I find it quite user-friendly and intuitive for beginners, but at the same time also scales well over time with the artist's capabilities. While I use Blender to present my workflow, there should not be many differences in the creational process when using different software.

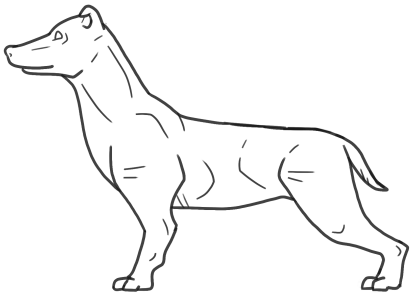


Figure 16 Small canine reference picture

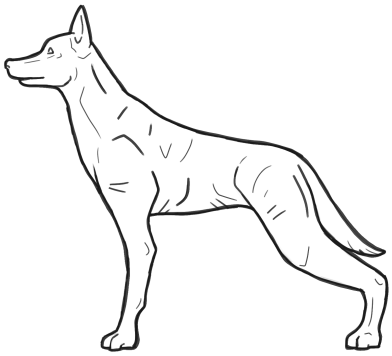


Figure 17 Medium canine reference picture

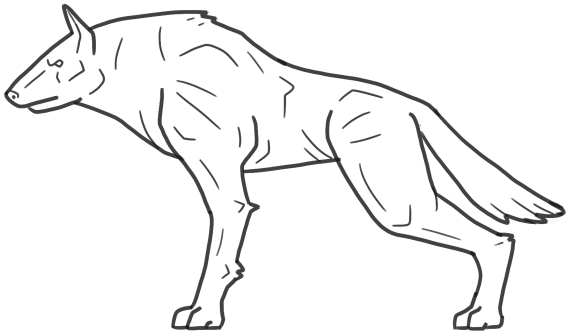


Figure 18 Large canine reference picture

3.2. Modelling workflow

Before beginning the modelling process, I first attempted to find reference pictures of the canines I decided to represent. I could not find any pictures I liked that could be used as reference pictures for my models. Therefore, I decided to draw sketches of the canines based mainly on several different photos, real-life experience and in the third, warg-like case, my imagination. These sketches are represented in **Figure 16**, **Figure 17** and **Figure 18**, please note that they are presented out of scale.

After setting up the desired image as a reference picture in Blender to keep it visible behind my model as I work, it is time to start modelling. I chose to use the *box-modelling* technique. It is a rather simple modelling method based on inserting a primitive shape such as a cube at the beginning and then sculpting it into the shape of the desired model. I have inserted a cube roughly the height of the middle part of the canine's torso. To make my work easier, I have decided to use the mirror modifier. Therefore, before any further changes to the cube, I used a loop cut to cut the cube on the left and right halves, keeping the loop cut in the exact middle by clicking the right mouse button immediately after selecting the direction of the cut. Afterwards, I selected all the faces of the left half and deleted them. Then I proceeded to apply a mirror modifier on this half-cube with the clipping option enabled.

Extruding the front and faces planes repeatedly, I roughly followed the shape of the canine's torso and head on the reference picture. The torso itself does not require a high level of detail, as it is not as flexible as other parts and will not deform too

Figure 19 The finished paw

significantly during animations. Once I had the rough shape of the torso and head, I added the subdivision surface modifier, giving the model a smoother, more organic look. The head and especially the face and ears area, on the other hand, required a slightly denser mesh, not for the sake of animations, but to make the face look detailed and realistic.

To add limbs, I selected the faces in the shoulder and hip areas, deleted them, then selected the edges of the newly created holes and extruded those into sides, creating a cylinder-like shape for each limb. Keeping future animations and mesh deformations in mind, I made the legs a little more detailed, especially in the joint areas such as the elbow, knee, ankle and wrist.

I wanted to make the feet well defined. Since that would be difficult to achieve by continuing the extruding process for each foot, I have decided to create a foot model separately, duplicate it and connect the feet to the rest of the body. I started by inserting a plane and scaling it down roughly to the size of the metacarpal pad, which is the central part of the paw. After subdividing the plane and changing its shape to resemble something between a teardrop and a heart, I added the subdivision surface modifier to this new object to achieve a more detailed look. To create the toes, I duplicated this teardrop-shaped plane, scaled it down, and made it a little more narrow to resemble the digital pad even more. Then I proceeded to create three more copies of this finger plane, reshaping each of them slightly based on their placement and canine anatomy. With the shape of the base established, I began to extrude each of the planes and connect them, creating the model of the foot. I created a copy of the foot for each leg and connected them to the rest of the mesh. You can see the bottom of the foot connected to the rest of the body in **Figure 19**.

To make my work more efficient, I used the mirror and subdivision surface modifiers, which helped me save a notable amount of time. You can see the final model in **Figure 20**, **Figure 21** and **Figure 22**. As I have mentioned earlier in this section, I also created two more models of canines of different proportions. Sticking to the theme of this thesis of working with maximal efficiency, I have used my first model to create the other two, rather than starting from scratch each time. I created a duplicate of the first model and started changing the proportions.

For the smaller Jack Russel terrier-like model, I have firstly scaled the whole model

down to under 50% of the size of the first model. Then I shortened the legs and made the paws smaller. I also enlarged the head a little and made the neck shorter. Lastly, I changed some details, such as the shape of its ears and the size of its tail. You can see the finished smaller model in **Figure 23**, **Figure 24** and **Figure 25**.

To create the larger warg-like model, I scaled the first model up to approximately 200% of the size of the first model. The trickiest part of creating this model was repositioning the head to the front from the chest. I have tried to solely deform the mesh and reposition the head and neck at first, but that resulted in the mesh on the bottom part of its neck being too dense and the other side not detailed enough. Therefore, I have decided to start over, delete the neck part of the mesh completely, position the head where I want it and connect it with the rest of the body with a new neck mesh. I have also changed the shape of the head and snout a little and enlarged the tail. You can see the finished third model in **Figure 26**, **Figure 27** and **Figure 28**.

Figure 20 Front view

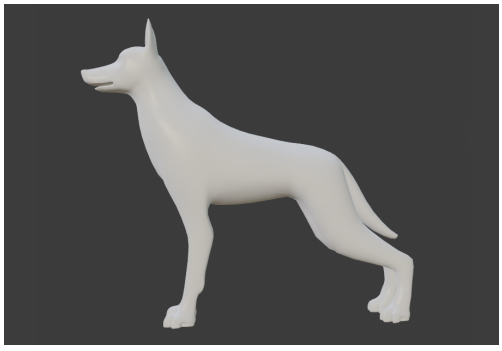


Figure 21 Side view

Figure 22 Back view

Figure 23 Front view

Figure 24 Side view

Figure 25 Back view

Figure 26 Front view

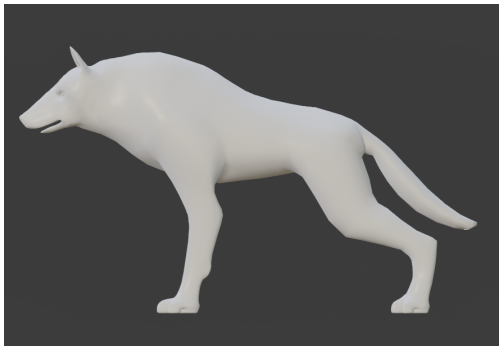


Figure 27 Side view

Figure 28 Back view

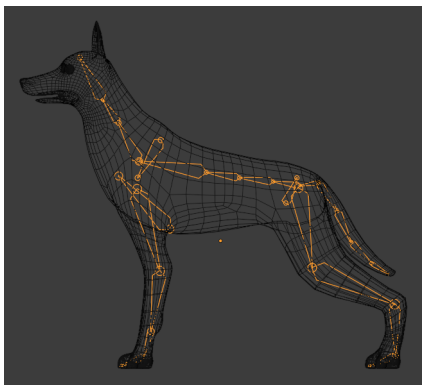


Figure 29 Animation skeleton: Side view

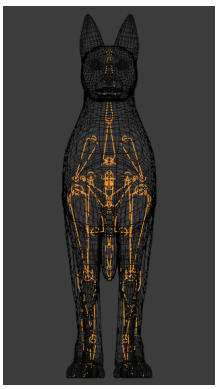


Figure 30 Front view

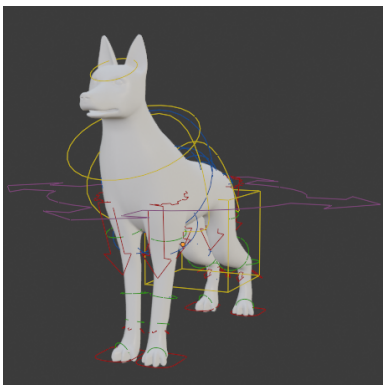


Figure 31 Rigify controls

3.3. Rigging the model

Since one of the purposes of this thesis is to explore the options of modelling and animating quadrupeds in the most efficient ways, I have decided to use a pre-made quadruped skeleton provided by a Blender add-on Rigify [56]. Rigify offers numerous shapes of pre-made skeletons, such as skeletons for basic humanoid and quadruped as well as several skeletons for specific animals, like a bird, or a cat. I have used the Basic Quadruped meta-rig since it had the appropriate topology and amount of detail for my models. After some size and proportion adjustments made on the skeleton to fit my model, I find it a considerably convenient way to save time. You can see the skeleton adjusted to the size and proportions of my surface mesh in **Figure 29** and **Figure 30**.

After aligning the meta-rig to the mesh, I let the program generate a rig based on the meta-rig. This rig also includes intuitive controls you can see in **Figure 31**, which make posing the model well manageable. Blender also offers the option of automatized skinning, which I have decided to use and achieved decent results. However, some of the generated weight maps did require manual adjustments. The head bones area of influence would grow weaker at the snout area, which, naturally, should move with the head and therefore, I had to make sure the whole head would move with this bone, snout and ears included. Another problematic area I had to adjust was where the front limbs connect to the body, so around the chest and shoulders. I have tried to automatically generate multiple different sets of weight maps based on meta-rigs with differently positioned bones in these areas. However, none of them resulted in a set of weight maps I would be completely happy with, so I simply adjusted the weight maps to fit my concept the best.

For each of my other two models, I have repeated this process. I added a basic quadruped armature to each of them, resized and repositioned each of the bones to fit the model's proportions, and had the program generate a rig fitting each of them. During the skinning process, some of the automatically generated weight maps have not been calculated to my likings, so I adjusted them.

For the smaller model, the adjustments mostly included what I have already mentioned, that is, the head and chest areas. For the larger, warg-like model, I also had to adjust the weight maps of the neck and throat areas. Possibly because of its increased

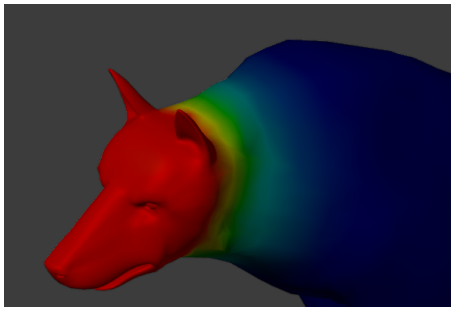


Figure 32 wm - head

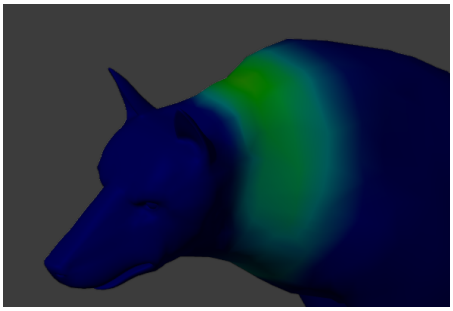


Figure 33 wm - first neck bone

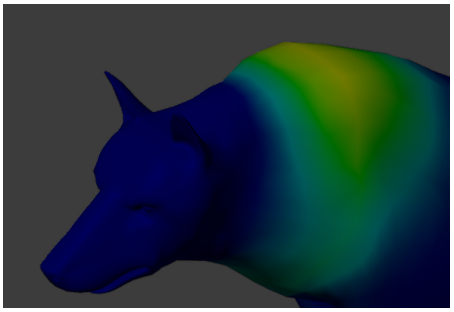


Figure 34 wm - second neck bone

size or different angle, the weight maps generated for the spinal bones of the third model introduced unwelcome effects, such as unnatural deformation that would appear during the animations. In **Figure 32**, **Figure 33** and **Figure 34**, you can see the weight maps of my third model's head and neck bones after adjustments.

3.4. Animating the model

As the source of references for animating my canine models, I have decided to use primarily real-life experience as well as video footage. Blender provides a reasonably straightforward animating environment. The process of animating a model then consists of posturing the model into key poses, letting the program interpolate between those poses, and possibly tweaking those interpolations if they do not look right. This way, I created a simple idle animation as well as animations for walking, galloping, transitioning between walking and galloping, and turning. I have created them for my first model since it is of average size and proportions.

All of the animations have 18 frames, and since all the animations, except the transition one, are designed to repeat smoothly when required, I copied the prose from the first frame to the 19th frame, which is not part of the animation anymore, but it makes it loop well. You can see screenshots of these in animations on my first model in **Figure 35**, **Figure 37** and **Figure 36** and **Figure 38**. The animations I have created are saved as Actions in Blender, which are essentially data blocks containing the animation data. These actions are then assigned to specific objects, such as the rig that control the model.

The main focus of this thesis is to document the possibilities of the application of a set of animations created for one model to other models with different proportions.

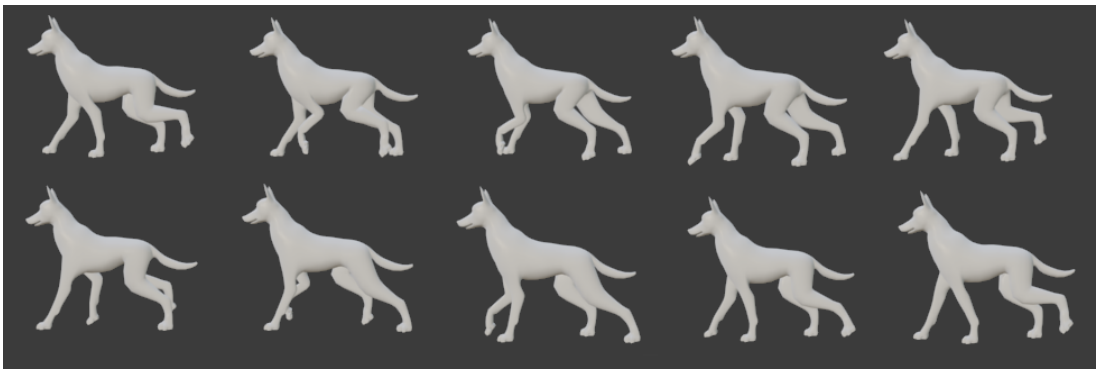


Figure 35 Walking Animation

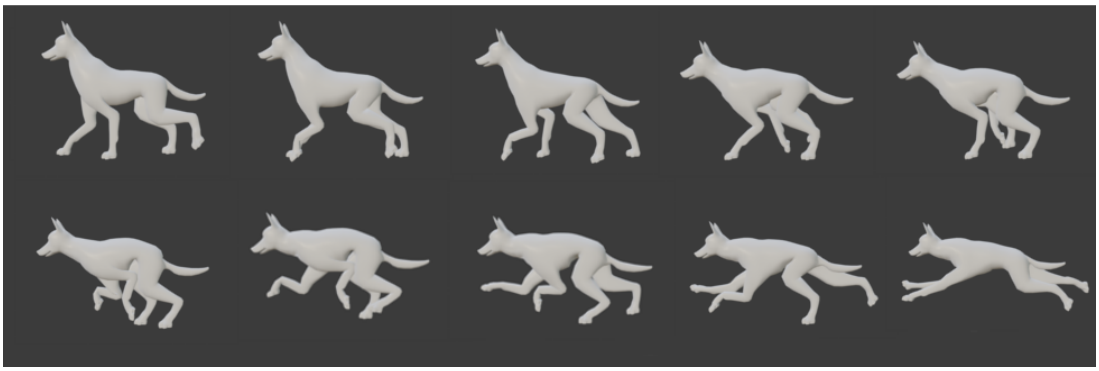


Figure 36 Transition between walking and galloping

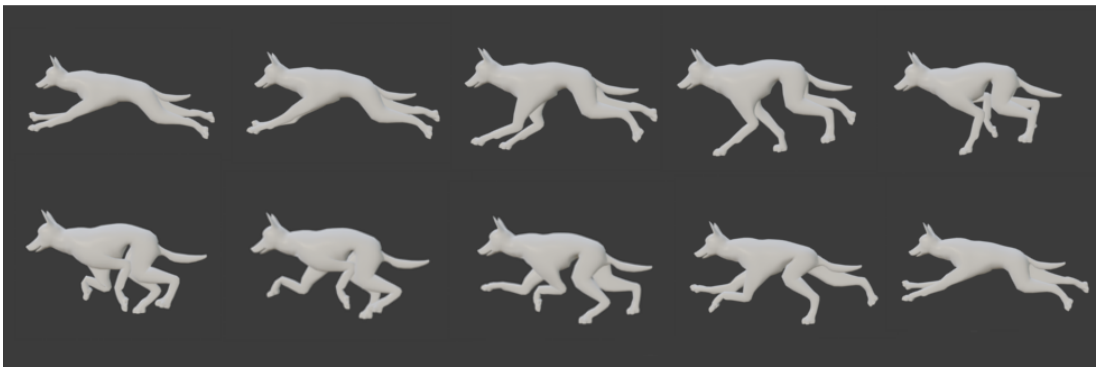


Figure 37 Galloping animation

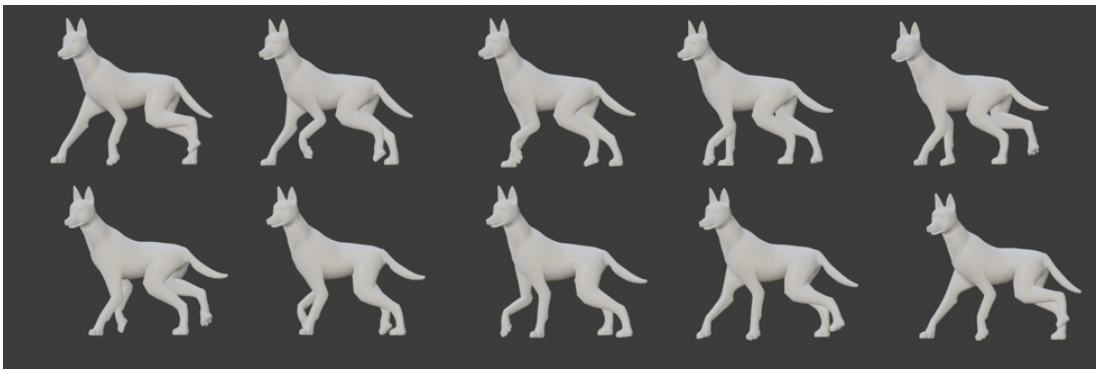


Figure 38 Turning left animation

For an action to be assignable to multiple different objects, the objects must share the same property paths, which are, in our case, bone names. If we try to assign an action, we have created for one object to another object that only shares some of the bone names with the original, the action will be applied to only these bones, leaving the rest motionless. Since I have used the same quadruped armature template for all three of my assets, the number of their bones and their names match, and therefore the animations can be shared between them.

Assigning the actions to my smaller Jack Russell terrier model went relatively well and without complications. Its posture is quite similar to the original model, and its legs are shorter than the first models. Therefore it makes sense that applying the original animations on this model would not cause any limb collisions or unwanted artefacts. The controls I have used for defining the keyframes on the first model scaled relatively well and look decent. However, due to its body being longer than the original models, the animations do not look as realistic and as good as they possibly could be if they were originally designed for this model. Sadly, I could not easily improve this without changing the original animations, but I would still consider the overall look a success. In **Figure 41**, **Figure 43**, **Figure 44** and **Figure 42**, you can see screenshots of some of the action the actions applied to the second model.

The third, warg-like model's posture is considerably different from the original, and further adjustments were required. The increased length of this model's limbs compared to the original caused unwelcome visual complications such as unnatural bending and overlapping parts of the model. Another complication was caused by the different posture and the warg's head position, especially in combination with the galloping animation. Rather than changing the animations, which would go against the focus of this thesis, I have added constraints to the bones causing the problematic behaviour. For the overlapping parts, I added distance constraints to the relevant bones, setting their minimal distance from each other to suitable values. To correct the posture, I have applied several constraints to the head and neck bones to limit the rotation and position on the Z-axis. I have included the pictures with and without constraints in **Figure 39** and **Figure 40**. I have included screenshots of the animations applied to the warg model in **Figure 45**, **Figure 46**, **Figure 47** and **Figure 48**.

Figure 39 Third model mid-gallop without constrains

Figure 40 Third model mid-gallop after adding constrains

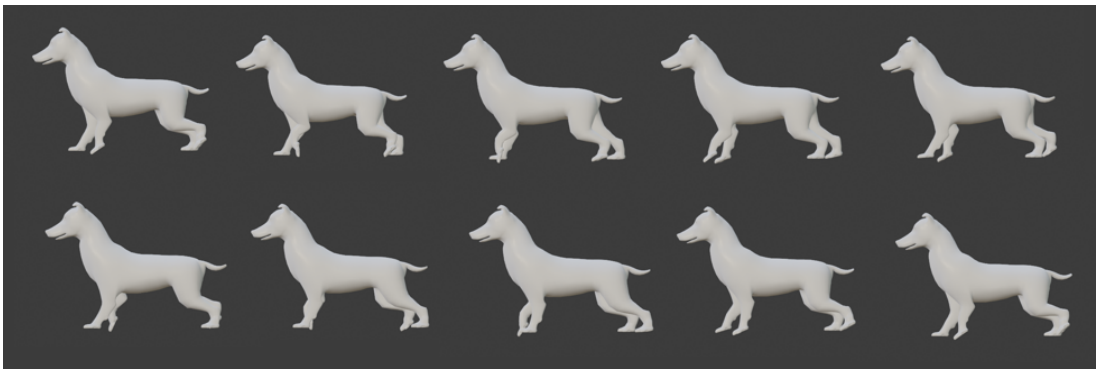


Figure 41 Walking Animation

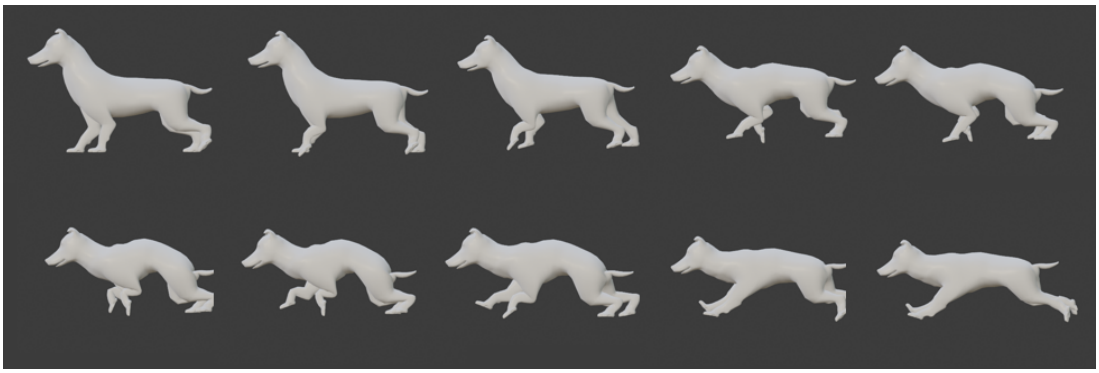


Figure 42 Transition between walking and galloping

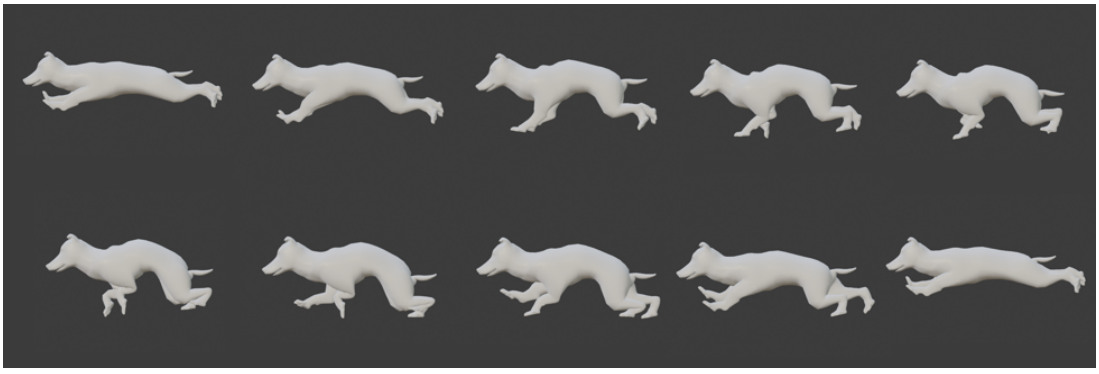


Figure 43 Galloping animation

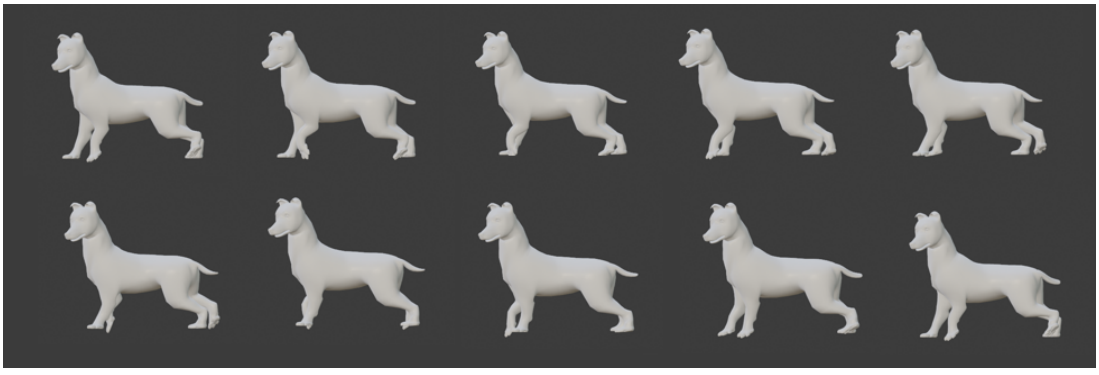


Figure 44 Turning left animation

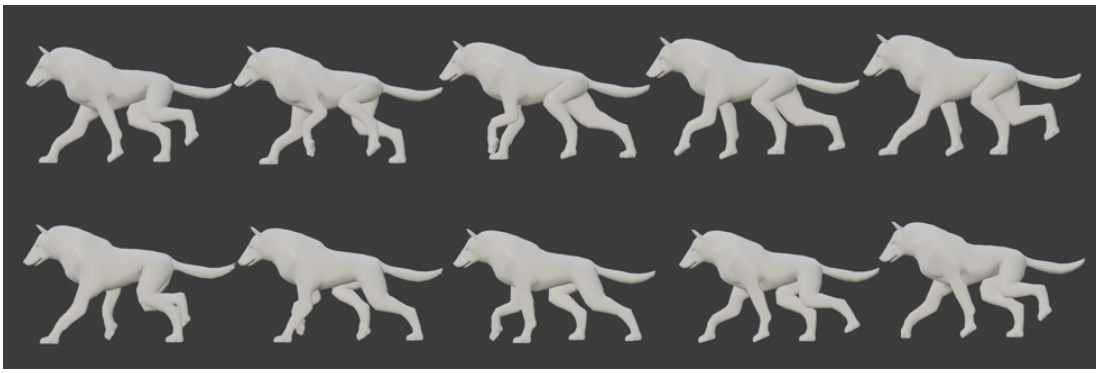


Figure 45 Walking Animation

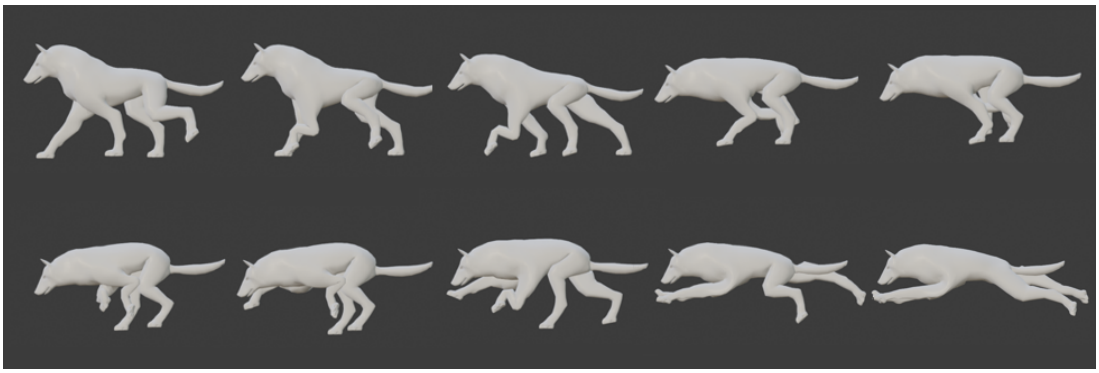


Figure 46 Transition between walking and galloping

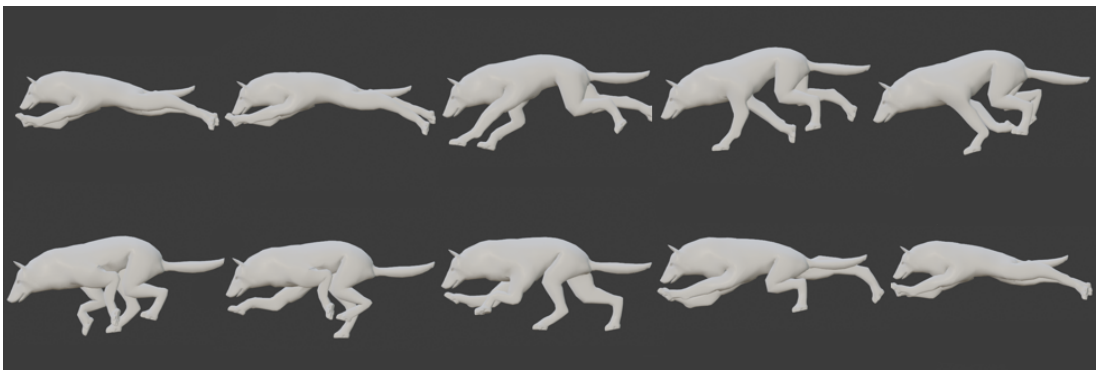


Figure 47 Galloping animation

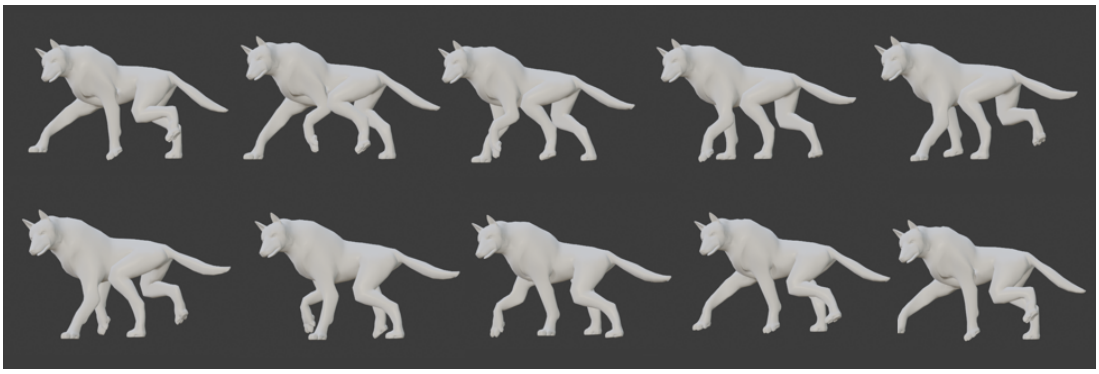


Figure 48 Turning left animation

Figure 49 asset positioning

3.5. Creating a Unity Project

Considering that the subject of my thesis is video game assets, I have decided to create a project in Unity. I have chosen Unity over any other video game engines because of the pros I have mentioned in **Section 2.5** and because I have worked with Unity before and my experience with it has been quite good.

I have exported each of the three models, with their rigs and animations, as separate FBX files from Blender, making it possible for anyone to run the project without the requirement of Blender being installed on their computer. To export each of the assets separately from one .blend file, I; made only one model and its meta-rig visible and selected, chose 'export file as FBX', checked 'Limit to Selected Objects', 'Apply Transform' and 'Bake Animation', exported the FBX file and repeated this process with each of the assets.

In the project scene, I added a plane as the terrain and two sources of simple directional light to make the scene lit to my liking. Afterwards, I imported the three models as assets. Each model already includes six clips, one for each animation. As I mentioned in the previous subsection, I have added a 19th, in Blender ignored, frame to most of the animations to make them loop well. The clips in Unity includes these, making the first and last frame of the loop the same, however, this can be easily fixed by trimming the clips by one frame in the animation inspector to make their length 18 frames. Another change I made to the animations in Unity is speeding some of them up to make them look more realistic, not only as they are, but also in consideration of the speed and scales between the models. Since smaller creatures tend to move quicker, and the movement speed of all three models in my project is the same to make the controlling easier, I set the animation speed rates of all the animation clips of the small canine to 1.7 and the medium canine to 1.5. All animation speeds can be easily changed in the state machine inspector.

To add the models into the scene, I simply dragged them on from the Asset folder. For the sake of simplicity when it comes to comparing the models and animations between each other, I have placed all three models beside each other. You can see the positioning of the assets before running the project in **Figure 49**. To each model, I added several

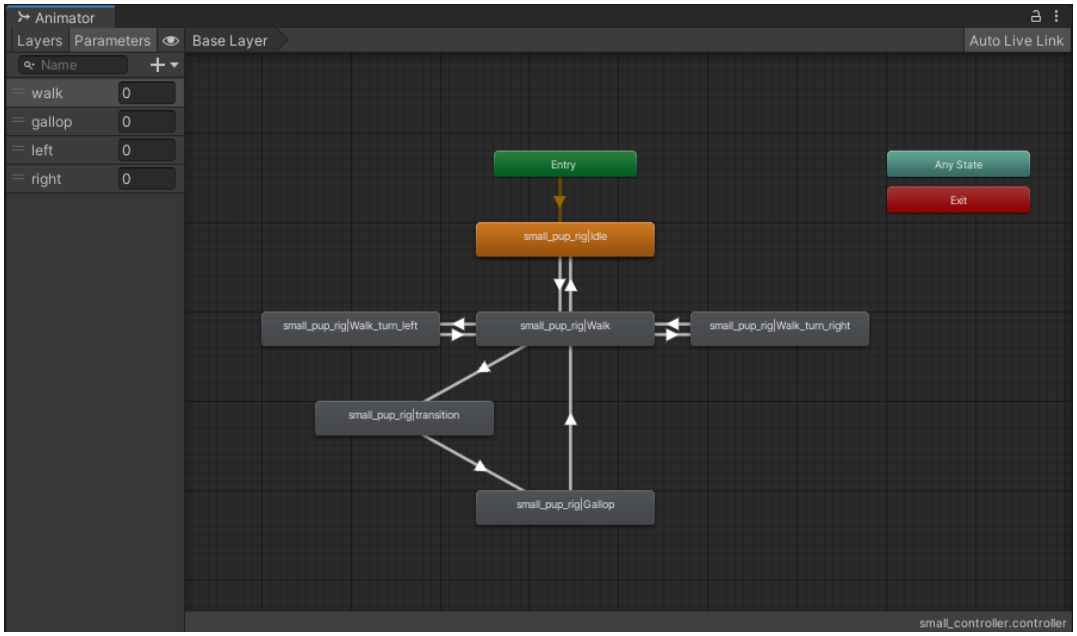


Figure 50 State Machine

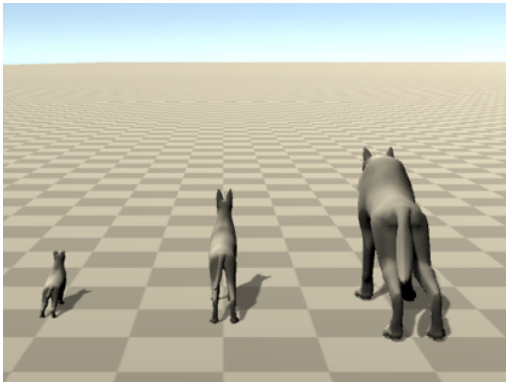


Figure 51 Camera 1

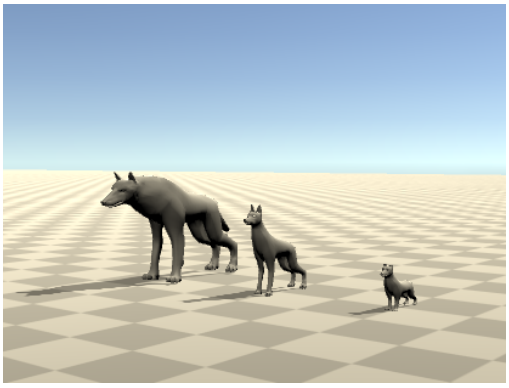


Figure 52 Camera 2

components in the Inspector, specifically *Rigidbody*, *Character Controller*, *Animator* and a script I wrote to control the models and their animations. I had to make minor adjustments to the character controller centre positions to fit the models better.

I made an *Animator Controller* for each model, managing the transitions between animations based on triggers. All three of these controllers have the same structure, but they contain specific clips for each asset. You can see the state machine for the small canine in **Figure 50**, where on the left, you can see parameters that manage how the animation clips change. The default state is the idle animation. When the walk parameter changes from 0 to 1, the idle animation transitions to walking, then back to idle, once it switches back to 0. The other parameters - gallop, left and right similarly manage transfers between the respective clips. A special case is the transition state between walk and galloping. When the gallop parameter changes to 1, the state changes firstly to transition. This state is the only one that does have an exit time and automatically moves on to galloping after the clip finishes.

I have also written a simple script to manage the state machine parameters based on user input. This script switches the values of the parameters depending on pressed keys, and it also controls the speed and direction of the model's movement. The keys I chose for the movement control are often used in video games - W, A and D to move forward, left and right, respectively. Holding Left Shift causes the asset's movement to speed up and their animation to switch to galloping.

Another short script I wrote is to control the cameras. There are two cameras, the default one behind the models and a secondary one in front of them under an angle. The key to switch between the two cameras is set to Q. You can see the view of the two cameras in **Figure 51** and **Figure 52** and I have also included both scripts on the next pages.

4. Conclusion

Through the course of this thesis, I have explored various approaches to modelling and animating quadrupeds, and I firmly believe that all the research that I have done has been not only worthwhile but necessary to do so successfully. The theoretical part on biomechanics that I have researched in **Section 2.1** has helped me better understand the forces behind quadruped anatomy and motion.

While it may be possible to utilize this knowledge while implementing physically-based animations, this path appears to be excessively complex to use in video games, and therefore I find it better to use this as support information for artists and video game developers to have to help them create more realistic and better-looking video game assets than they would without knowing anything about the biomechanics of quadruped motion.

When it comes to the comparison of animating quadrupeds to animating humanoids, there are a few differences, but overall I have found it quite similar. Leaving aside the obvious one, which would be the difference between humanoid and quadrupedal movement itself, one of the main contrasts that I have encountered when it comes to developer environments is that humanoids get a lot more attention and default available options. As I mentioned, I have used some tools specialized for quadrupeds, such as the pre-made quadruped meta-rig from Rigify, as mentioned in **Section 3.3**. However, humanoid-friendly tools are prevalent during most of the process. For example, Unity only supports mirroring animations only when the rig is humanoid. I also had issues attempting to mirror the animations in Blender, which is why I had to manually make different animations for turning to the left and to the right. That being said, I think this difference between the support of making humanoid versus quadruped assets is understandable, considering the vast majority of video game characters being humanoid.

Using the research I have done in **Chapter 2**, I managed to create three canine models of different sizes and proportions that meet the requirements to be used as video game assets. I have also managed to create a set of animations on one of these models and apply it to the other two models, documenting the necessary changes I had to make during the whole process to achieve agreeable results.

I am quite satisfied with the amount of automatization that I have been able to use to create my assets and the amount of time that it saved me. The main example clearly being the focus of this thesis: the possibility of sharing one set of animations between the various quadruped models. Rather than making one quadruped skeleton carrying the animations, I have managed to discover the workflow of sharing animations between models with skeletons of the same hierarchical structure, which I find to be just as much of a success as the first option. The only difference between these two approaches is the order of operations when in my case, I add and scale a skeleton of desired structure to a model I want to animate and then append the animations to it after the skinning

process.

4.1. Future work

The animated assets I have created as a product of this thesis are essentially ready to be used in video games. To make the assets more visually appealing, it might be a good idea to create a set of textures and materials for each of them. Another area worth focusing on is adding fur and animating it to compliment the movement depicted by the animations I have already included. There are also always more animations to be added, such as jumping, sitting and laying down and back up, or turning while galloping, to only name a few.

Bibliography

[1] *Prehistoric Puppy May Be Earliest Evidence of Pet-Human Bonding*. URL: https://www.nationalgeographic.com/news/2018/02/ancient-pet-puppy-oberkassel-stone-age-dog/ (visited on 11/22/2020).

[2] Chris Sanders Dean DeBlois. *How to Train Your Dragon*. DreamWorks Animation, 2010.

[3] *DreamWorks Animator Thomas Grummt Talks 'How to Train Your Dragon: The Hidden World'*. May 21, 2019. URL: https://crookedllama.com/dreamworks-animator-thomas-grummt-talks-how-to-train-your-dragon-the-hidden-world-interview/ (visited on 10/21/2020).

[4] N. C. Heglund G. A. Cavagna and C. R. Taylor. "Mechanical work in terrestrial locomotion: two basic mechanisms for minimizing energy expenditure". In: *American Journal of Physiology* 233.5 (Nov. 1977), R169–R261. URL: https://doi.org/10.1152/ajpregu.1977.233.5.R243.

[5] R. McN. Alexander. "The Gaits of Bipedal and Quadrupedal Animals". In: *The International Journal of Robotics Research* 3.2 (1984), pp. 49–59. URL: https://doi.org/10.1177/027836498400300205.

[6] Sheila N. Patek Andrew A. Biewener. *Animal Locomotion*. Great Clarendon Street, Oxford, OX2 6DP, United Kingdom: Oxford University Press, 2018.

[7] David A. Raichlenand Herman Pontzer and Liza J. Shapiro. *A new look at the Dynamic Similarity Hypothesis: the importance of swing phase*. 2003. DOI: 10.1242/bio.20135165. URL: https://bio.biologists.org/content/2/10/1032?utm_source=TrendMD%5C&utm_medium=cpc%5C&utm_campaign=Biol_Open_TrendMD_0.

[8] DVM Vicki L. Datt (nee Johnson) and PhD Thomas F. Fletcher DVM. *Information About Gaits*. Aug. 2012. URL: http://vanat.cvm.umn.edu/gaits/info.html (visited on 10/17/2020).

[9] Franck Hétroy Ljiljana Skrba Lionel Reveret and Carol O'Sullivan. *Quadruped Animation. Eurographics State-of-the-Art Report*. Hersonissos, Creete, Greece, 2008.

[10] Infinity Ward. *Call of Duty: Ghosts*. Activision, 2013.

[11] Naughty Dog. *The Last of Us*. Sony Computer Entertainment, 2013.

[12] Rockstar Sand Diego. *Red Dead Redemption*. Rockstar Games, 2010.

[13] Petrana Radulovic. *The story behind Red Dead Redemption's horse motion capture*. Oct. 23, 2018. URL: https://www.polygon.com/red-dead-redemption/2018/10/23/17983012/red-dead-redemption-horse-motion-capture-rockstar (visited on 11/22/2020).

[14] Keith Stuart. *Aroooo! The inside story on Call of Duty dog motion capture*. May 24, 2013. URL: https://www.eurogamer.net/articles/2013-05-24-aroooo-the-inside-story-on-call-of-duty-dog-motion-capture (visited on 11/21/2020).

[15] @remedygames. *#YearOfTheDog*. May 18, 2018. URL: https://twitter.com/remedygames/status/996432955802374144 (visited on 11/22/2020).

[16] @UbisoftToronto. *Adventures in MoCap with Soba the dog!* Dec. 15, 2015. URL: https://twitter.com/ubisofttoronto/status/676845103042969600?lang=en (visited on 11/22/2020).

[17] Jane Wilhelms and Allen Van Gelder. *Combining Vision and Computer Graphics for Video Motion Capture*. Computer Science Dept., University of California, Santa Cruz, CA 95064, 2003.

[18] Demetri Terzopoulos Michael Kass Andrew Witkin. *Snakes: Active Contour Models*. Chlumberger Palo Alto Research, 3340 Hillview Ave., Palo Alto, CA 94304: Kluwer Academic Publishers, 1988. URL: https://web.archive.org/web/20160112014330/http://www.vavlab.ee.boun.edu.tr/courses/574/material/Variational%20Image%20Segmentation/kaas_snakes.pdf.

[19] *HISTORY OF 3D MODELING: FROM EUCLID TO 3D PRINTING*. June 14, 2019. URL: https://ufo3d.com/history-of-3d-modeling (visited on 12/31/2020).

[20] Irma Prus. *What Is 3d Modeling? Things You've Got To Know Nowadays*. Aug. 2, 2016. URL: https://ufo3d.com/history-of-3d-modeling (visited on 12/31/2020).

[21] Kori Rae Dan Scanlon. *Monsters University*. Walt Disney Pictures, Pixar Animation Studios, 2013.

[22] Dean Takahashi. *How Pixar made Monsters University, its latest technological marvel*. Apr. 24, 2013. URL: https://venturebeat.com/2013/04/24/the-making-of-pixars-latest-technological-marvel-monsters-university/ (visited on 05/21/2021).

[23] John Davison. *How 'Quake' Changed Video Games Forever*. 2016-06-22. URL: https://www.rollingstone.com/culture/culture-news/how-quake-changed-video-games-forever-187984/.

[24] John Papadopoulos. *Doom Eternal looks absolutely gorgeous in these latest official screenshots*. 2020-01-21. URL: https://www.dsogaming.com/screenshot-news/doom-eternal-looks-absolutely-gorgeous-in-these-latest-official-screenshots/.

[25] Epic Games. *A first look at Unreal Engine 5*. May 15, 2020. URL: https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5 (visited on 06/10/2021).

[26] id Software. *Quake*. GT Interactive, 1996.

[27] inspirationTuts. *Unreal Engine 5*. May 24, 2020. URL: https://inspirationtuts.com/unreal-engine-5/ (visited on 06/25/2021).

[28] id Software. *Doom Eternal*. Bethesda Softworks, 2020.

[29] Epic Games Inc. 2004-2021. URL: https://www.epicgames.com/site/en-US/home.

[30] Epic Games. *Unreal Engine 5*. May 26, 2021. URL: https://www.unrealengine.com/en-US/unreal-engine-5.

[31] © 2004-2021 Epic Games Inc. *Nanite Virtualized Geometry*. URL: https://docs.unrealengine.com/5.0/en-US/RenderingFeatures/Nanite/ (visited on 07/17/2021).

[32] Alias Systems. *Polygonal Modeling*. Silicon Graphics Limited. URL: http://academics.wellesley.edu/MAS/313/sp09/mayaguide/Complete/Polygons.pdf (visited on 05/01/2021).

[33] Eva Eggeling Christoph Schinko Ulrich Krispel and Torsten Ullrich. *3D model representations and transformations in the context of computer-aided design: A state-of-the-art overview*. URL: http://thinkmind.org/download.php?articleid=mmedia_2017_1_20_50022.

[34] Leif Kobbelt Mario Botsch and Mark Pauly. *Polygon Mesh Processing*. 5 Commonwealth Road, Suite 2C Natick, MA 01760-1526: A K Peters, Ltd. ISBN: 978-1-56881-426-1.

[35] Myung-Soo Kim Gerald Farin Josef Hoschek. *Handbook of Computer Aided Geometric Design*. 1st. North Holland, pp. 1–21. ISBN: 978-0-444-51104-1. DOI: https://doi.org/10.1016/B978-0-444-51104-1.X5000-X.

[36] Rain Rig © Blender Foundation. *Rain v2.0*. URL: https://cloud.blender.org/p/characters/5f04a68bb5f1a2612f7b29da (visited on 06/20/2021).

[37] Alias Systems. *Subdivision surface Modeling*. Silicon Graphics Limited. URL: https://courses.cs.washington.edu/courses/cse459/06wi/help/mayaguide/Complete/SubDs.pdf (visited on 05/10/2021).

[38] Steve Marschner et al. *Fundamentals of Computer Graphics*. 4th. A K Peters/CRC Press, Dec. 18, 2015. ISBN: 9781482229394.

[39] © 2019 Unity Technologies. *Normal map (Bump mapping)*. Apr. 15, 2019. URL: https://docs.unity3d.com/2018.3/Documentation/Manual/Standard%20ShaderMaterialParameterNormalMap.html (visited on 07/01/2021).

[40] Nicky L Webster. *High poly to low poly workflows for real-time rendering*. Jan. 2017. DOI: 10.1080/17453054.2017.1313682. URL: https://www.researchgate.net/publication/316435393_High_poly_to_low_poly_workflows_for_real-time_rendering.

[41] George Maestri. *Digital Character Animation 3*. 1st. New Riders, Apr. 12, 2006. ISBN: 10: 0-321-45512-6. URL: https://www.peachpit.com/articles/article.aspx?p=483793%5C&seqNum=3 (visited on 11/13/2020).

[42] Blender Foundation. *An example of left/right bone naming in a simple rig*. URL: https://docs.blender.org/manual/en/latest/animation/armatures/bones/editing/naming.html (visited on 06/20/2021).

[43] Steaphanie Cheng. *Human Skeleton System Animation*. June 2017. URL: https://bib.irb.hr/datoteka/890911.Final_0036473606_56.pdf.

[44] Blender Foundation. *Bendy Bones*. URL: https://docs.blender.org/manual/en/latest/animation/armatures/bones/properties/bendy_bones.html (visited on 06/20/2021).

[45] Rick Parent. *Forward Kinematics*. 2012. URL: https://www.sciencedirect.com/topics/computer-science/forward-kinematics (visited on 11/22/2020).

[46] Intel Game Dev. *Character Animation: Skeletons and Inverse Kinematics*. Aug. 9, 2017. URL: https://venturebeat.com/2017/08/09/character-animation-skeletons-and-inverse-kinematics/ (visited on 11/22/2020).

[47] Jennifer Lynn Kuhnel. "Rigging a horse and rider: Simulating the predictable and repetitive movement of the rider". 22860 N FM 81 Hobson, TX 78117: Texas A&M University, Aug. 2001. URL: http://oaktrust.library.tamu.edu/bitstream/handle/1969.1/490/etd-tamu-2003C-VIZA-Kuhnel-1.pdf?sequence=7%5C&isAllowed=y.

[48] Thomas Geijtenbeek. "Animating Virtual Characters using Physics-Based Simulation". PhD thesis. Utrecht, The Netherlands: Utrecht University, Dec. 2013. ISBN: 978-94-6182-389-2. URL: https://www.goatstream.com/research/thesis/index.html.

[49] Karan ingh Cyrus Rahgoshay Amir H. Rabbani and Paul G. Kry. *Inverse Kinodynamics: Editing and Constraining Kinematic Approximations of Dynamic Motion.* 2012. URL: https://www.cs.mcgill.ca/~kry/pubs/InverseKinodynamics/index.html.

[50] © 2021 Unity Technologies. *Unity.* URL: https://unity.com/our-company.

[51] © 2021 Unity Technologies. *Activation - Personal: Unity licensing.* URL: https://unity3d.com/unity/activation/personal (visited on 08/06/2021).

[52] © 2021 Autodesk Inc. *Autodesk.* URL: https://www.autodesk.com/.

[53] © 2021 THREEKIT INC. *When should you use FBX 3D file format ?* Oct. 28, 2019. URL: https://www.threekit.com/blog/when-should-you-use-fbx-3d-file-format (visited on 08/06/2021).

[54] Will Chesney and Joe Layden. *NO ORDINARY DOG: My Partner from the SEAL Teams to the Bin Laden Raid.* St. Martin's Press, Apr. 21, 2020. ISBN: 9781250240019.

[55] Blender Foundation. *Blender: About.* URL: https://www.blender.org/about/ (visited on 01/03/2021).

[56] Nathan Vegdahl et al. *Rigify.* URL: https://docs.blender.org/manual/en/2.81/addons/rigging/rigify.html (visited on 05/12/2021).

Appendix A.

Attachment structure

The appendix to this thesis includes a .zip file containing the results of my work. In this file, there are two folders as well as a short readme file. The folder named Models carries a .blend file trolljul_quadruped_model_animation.blend containing all three of my assets and their animations. All three assets are in one scene collection.

The folder named Unity carries the Unity project I have created to demonstrate my results. The project includes the three assets added to a scene. After running the project, the user can control the assets with the W, A, D and Left Shift keys, as described in **Section 3.5**, and switch between two cameras in the scene with the Q key.