Procedural Walking Animations for Non-humanoid, Multi-limbed Characters

MSc Computer Games Development (Computing)

Author: **Marco Vincenzi** - SID: **1923786**
Supervisor: **Ian Brown**

September 2021

Table of Contents

List of Figures

Acknowledgements

I would like to thank the following people that have supported me in my efforts to develop this project, and without whom I would have not been able to complete this research. The entire team at Anglia Ruskin University, the Arts, Humanities and Social Studies faculty and, in particular, my research supervisor Ian Brown. My colleagues in the Computer Games Programming MSc course, as well as those in the Games Art MA course, with whom I have had the opportunity to gather feedback and new ideas about my work. Finally, I would like to thank my family, whose love and support have accompanied me and my studies throughout the past year and the unprecedented challenges it brought with it.

Abstract

This thesis presents a framework of systems that allow the creation of procedurally generated walking animations for 3D characters within the Unity Engine. The system takes in input a rigged model and dynamically handles its locomotion as it moves, producing a realistic walking animation of its limbs. The system is designed to accommodate any multi-legged character model, but its strength lies in animating bug-like creatures. It can adjust feet placement and body rotation to account for different terrain conformations, such as hills and slopes. At runtime, the algorithm dictates which limb to move and which trajectory its stepping animation should follow, appropriately positioning each leg to maintain a state of simulated stability. The algorithm was tested on different terrains and its performance was analysed by comparing it against a traditionally animated solution.

Digital Material

This study presents a series of methods for the automatic generation of procedural character animation, and would therefore not be complete without a tangible demonstration of its abilities. The entirety of the project files and source code is freely available online and it can be accessed from a public repository hosted on GitHub at:

github.com/ZykeDev/procedural-locomotion

The application project can be downloaded and experienced first-hand. The project requires the installation of the Unity3D Engine distributed by Unity Technologies ApS.

Introduction

The following study revolves around the development and analysis of a programming tool capable of dynamically animating the movement of characters in a virtual 3D space, using information gathered from its surroundings in real-time. More specifically, this research aims to find the set of components necessary for the creation of a procedural locomotion system for non-bipedal, non-humanoid, multi-legged avatars, that is, characters with more than two lower limbs used for locomotion. While the most basic examples of this are quadrupedal creatures such as cats and dogs, the system should be able to comfortably accommodate an indefinitely high number of limbs, appropriately animating arachnids and insect-like creatures. The proposed algorithm does, however, require the limb distribution to follow a symmetric or semi-symmetric structure, with each limb classifying as either left, or right, relative to the main body.

This tool could be used by developers and animators, and thus, a list of customizable parameters will need to be exposed to the end-user. These can range from the number of legs that can be raised at a time while maintaining a stable posture, to the arcing trajectory the tip of a limb should follow during each step animation. While the developed tool is primarily aimed at game development on the Unity Engine (Unity Technologies, 2005) and was designed to play at runtime within a game engine, its only objective is to focus on a character's walking animation, meaning it could be used for any similar task that requires a dynamic locomotion system.

At the end of this study, it will be shown how the developed Procedural Locomotion algorithm is capable of animating the walking animation of multi-legged characters in a realistic manner.

Key Terms

End Effector — The end component of a kinematic chain or apparatus. In the field of robotics, it is typically a device designed to interact with the environment, while in the field of procedural animation, it simply identifies the end node of a kinematic chain.

Gait — The pattern of steps of a creature at a particular speed. For humans, this is defined as the manner of walking a person displays as it moves.

Inverse Kinematics — The mathematical process of calculating the variable joint parameters needed to place the end effector of a kinematic chain in a given position and orientation relative to the root of the chain.

Kinematic Chain — A sequence of rigid bodies connected by joints.

Keyframe — An animation element that defines the starting and/or ending points of any smooth transition.

Normal Vector — In Geometry, a Normal Vector, or simply Normal, is a unit vector that is perpendicular to a given plane.

Procedural Animation — A type of computer animation designed to automatically generate animations in real-time.

Raycast — In 3D design and game development, a raycast is a ray that is "cast" from a specific point in space towards a given direction. It is often used to compute the lighting values between a light source and a three-dimensional object.

Rig — A rig, or armature, is a hierarchical set of interconnected parts (bones) that represent the internal skeleton of a model. 3D animation is normally achieved by manipulating their coordinates.

Secondary Animations — Motions that are generated as a reaction to the movement of primary motion by a character, making its animations seem more natural.

1. Literary Review

The traditional character animation workflow in game development is typically tackled at a per-character basis, with animators manually adjusting a 2D or 3D model's pose at every keyframe to match the desired motion. Walking animations, for instance, tend to require animators to describe where each limb and each joint should be positioned at any given time throughout the animation. This process, while often made easier with the help of specifically designed tools and software, can be expensive in both time and resources (E.M. Rawes, 2018). Furthermore, it is also a procedure that is generally carried out in the early phases of development. Unforeseen design changes that often arise midway through production may result in artists having to go back to the drawing board. It is this research's aim to develop a system capable of dynamically dictating where each of a three-dimensional model's components should be located while moving. Such a system would allow developers to simply *rig* a creature and delegate its walking animation to the computer, granting them the option to shift their focus onto other areas of the creative process. This is known as procedural animation. Within the specific context of this study, the developed system is a Procedural Locomotion algorithm capable of describing the movement of non-bipedal, non-humanoid entities in real-time.

1.1. Procedural Animation

Procedural animation is an automatic, algorithm-driven method of generating computer animations in real-time. This is normally achieved by dynamically animating objects in a virtual space, whereas the traditional, handmade approach relies on predefined animations. Applications can be quite varied, ranging from movie production to physics simulations, but the most common area where they are utilised is the videogame development field. Procedural animations can be employed to design not only the motions of a character, but also particle systems and secondary animations like a cloak's fluttering or a person's hair blowing in the wind. It is noteworthy to point out how these two solutions are not mutually exclusive. Semi-procedural animation can complement handmade animations by dynamically blending the transitions between them. This can be incredibly useful in situations where, for example, a character needs to transition from an idle animation to a walking one, a type of transition that may appear quite abrupt unless the *end* and *start* keyframes are properly interpolated.

In the context of procedural locomotion, developers have proposed a plethora of different systems. Some are data-driven, relying on positional values obtained through motion capture, while others are entirely physics-based and can programmatically simulate a creature's movements in a very realistic manner (Multon et al, 1999). Others chose to instead take advantage of the most recent strides in Artificial Intelligence and Machine Learning, training a series of frameworks and using these to imitate real-life behaviour. As the need for more realistic 3D animations has grown over the past decade, so has the efficiency and sophistication of AI solutions. A great number of companies have created their own products, often specializing in animated movies or character rigging for video games. Some of these include *Mixamo*, an online tool for automatic machine learning-piloted rigging and animation of 3D characters (Adobe Systems, 2015), or *Cascadeur*, a fully-fledged animation software specialised in physics-based motion (Nekki, 2020). Both of these examples provide several preset animations, not only for walking and running, but also for a myriad of other types of movement and interaction developers may need for their projects. Unfortunately, products like Mixamo are only able to rig and animate humanoid-like models, forcing developers to rely on other solutions when their target creatures are animals or other mythical creatures. A big proponent of Machine Learning-driven methods is none other than Disney and its movie production studio, which developed a Deep Neural Network method capable of generating novel human faces and their corresponding expressions, an incredibly valuable tool in creating 3D characters for their virtual worlds (Chandran, 2020). The recent push for more affordable and automatic solutions comes not only from big studios with large budgets, but also from smaller, indie companies, aiming to optimize workflow without compromising on quality (G. Katragadda, 2019). These tools can, however, pose a few risks. Many professional studios have run into efficiency and efficacy issues while working with procedural animation toolsets. The lack of a streamlined approach is thought to be one of the main causes (Isikguner, 2014).

While many distinct techniques for procedurally animating locomotion have been developed over the years, they are found to be either proprietary – with their specifications not often divulged to the general public – created *ad hoc* for specific projects, or are solely focused on describing bipedal motion, typically targeted at humanoid characters in First Person Shooters or Third Person Shooters. A notable exception can be found in Maxis Studios's generalized procedural animation systems for arbitrarily legged creatures (Hecker et al, 2008). Their exhaustive research process ultimately culminated in the development of a custom 3D animation tool the Californian studio utilised in-house while working on their 2008 game *Spore* (Maxis Studios, 2008). Other researchers in this field opted to instead tackle annexed control strategies for "realistically simulating periodic locomotion patterns", which are recurrent in quadruped animals, such as a horse's trotting or galloping. The main objective of these such studies was to analyse their stable movement, replicate it and apply it to robotic armatures (Raibert, 1990).

A lack of baseline solutions at the engine level has prompted developers to build custom tools able to satisfy their needs. As recently as April of 2021, a third-party plugin specifically targeted towards procedural locomotion (Misultin Studios, 2021) was developed for the Unreal Engine (Epic Games, 1998), one of Unity's greatest competitors (Šmíd, 2017). Taking all of this into account shines a light on a niche, but present, gap in this corner of the Unity market space. Filling such a gap is one of the aims of this research.

1.2. Locomotion Control

While the underlying mechanism for procedurally animating a model may be the same, the way the locomotion is controlled can vary drastically between solutions. The three most common forms of control are *Foot Placement*-driven locomotion, *Motion*

Capture Data-driven locomotion and *Pelvis*-driven locomotion (Karim et al, 2012). The Foot Placement method analyses how each foot is positioned and forces the upper part of the body to follow the resulting structure. In doing so, the main body tracks the trajectory dictated by its feet and is able to maintain a stable and realistic form. The Motion Capture method relies instead on *mocap* data and overlays its coordinates on top of the model. Then, similarly to how it is done in the foot placement method, each limb follows the predefined set of steps by aligning its feet with the overlaid footprint below. The Pelvis method, as opposed to the other two modes, gives full movement control to the upper body and uses it to define the locomotion trajectory. As a character moves, its pelvis, or any element that may serve the same function, imposes a list of target positions for the legs beneath to try and follow. This is the behaviour this project's algorithm implements.

2. Methodology

This project was developed, tested and built using the freely available Unity 3D Engine by Unity Technologies (version 2020.3.12f1) and the Animation Rigging package (version 1.0.3) on a Windows 10 PC equipped with an AMD Ryzen 7 3700X and 16 GB of DDR4 RAM. Given the stable environment offered by Unity, the system is assumed to work on other similarly-specced machines. In order to facilitate both development, testing and deployment, alongside the locomotion system, a simple movement controller and camera-tracking system are included. These were implemented using Unity's *Character Controller* and *Chinemacine* respectively. The study involved the creation of a group of algorithms and tools, written mainly in the C# programming language. Attaching these to an arbitrary creature-like three-dimensional model allows it to follow simple directional commands and display a procedural walking animation. For the sake of simplicity, within the following document, the words "limb" and "leg" will be used interchangeably to identify the body parts that are used for locomotion.

2.1. Constraints

There are a few points that differentiate the methods proposed in this study from the solutions presented in the section above, one of which is the limitation this project imposes on itself. By deliberately constraining its scope to a game engine, the described approach does not require the use of a separate, standalone software for animation design. Another strength of this approach is the tool's ability to define different weights for each limb, adjusting the speed and gait of a character without the need to rely on the engine's physics simulation system. The presented solution is partially based on Unity's Animation Rigging package, as it utilizes its Two-Bone Inverse Kinematic solver (TBIK). In order to abide by the rules and requirements of the TBIK, the described algorithms also impose some constraints onto the imported 3D models, requiring them to respect a list of simple criteria:

1. The entirety of the model needs to be inside a parent object.
2. The model needs to have a hierarchical bone structure where all limbs are *children* of the *parent* rig element, and each limb must contain its sub-limbs in a cascading child-parent relationship. This type of structure is already common in most commercially designed models. All other parts within the model, such as the head and torso, if present, are not subject to these restrictions.
3. The main “body” element of the model, where the joints and limbs typically start from, needs to be manually selected and flagged as such when setting up the character.
4. The model needs to incorporate an *armature* skeleton for its limbs. Its leg bones need to have been therefore rigged in advance.
5. The main body element also needs to have a *Rig* component attached to it. The Rig component is part of the Animation Rigging package. Its default behaviour already presents a slider to adjust a rig’s weight and how strictly its movement should follow the animation target. Said component is, however, ignored, in favour of a more specific implementation that is described further below (Section 3.4).
6. The default model pose has to be a *stable* one, meaning if a creature needs to stand on a minimum of 4 feet in order to maintain balance, its model will also need to have at least 4 of its feet laying on the ground surface.

In order to make the implementation process easier for a possible end-user, an option to automatically set up some of the necessary requirements is included, making sure these constraints are adhered to. However, due to the highly variable nature of model hierarchical structures and the components that form them, this approach is not guaranteed to work correctly in every case. How this automatic setup system is implemented and how it behaves is illustrated in Section 3.7.

2.2. Inverse Kinematics

One of the fundamental aspects of this research is the way in which limbs behave during locomotion, as it hinges on the concept of *Inverse Kinematics*. In order to make the character's legs move in a realistic fashion, many examples of animation software implement an Inverse Kinematic (or simply "IK") system, a mathematical process used to compute the optimal position of each limb element and the optimal orientation of each joint in an articulated structure. Given a chain of connected nodes, the IK process tries to find a valid position for all nodes. Starting from its "leaf" up until it reaches the "root", the procedure attempts to move the leaf node, known as the *end effector* in the field of robotics, towards a given target location. In a limb IK chain, the leaf represents the "foot", while the root is simply the main joint attached to the pelvis and from where the limb starts.

This approach mirrors the Forward Kinematics methodology, where positional values are instead calculated starting from the root joint and descend towards the leaf, which is much less challenging to compute. An algorithm for finding a valid, and ideally optimal, set of coordinates for a given chain is called an Inverse Kinematic Solver. Countless IK Solver implementations have been proposed and developed over the years, with varying levels of efficiency and efficacy, many of which present hybrid solutions. The *Jacobian* IK Solver is a purely numerical technique able to work with chains that concatenate multiple joints. It and its derivations are popular in the field of robotics, but can quickly grow in computational overhead as the number of joints and degrees of freedom increases. Another set of popular algorithms are those of the Cyclic Coordinate Descent family. CCD processes are renowned for their inherent simplicity, reliability and speed, offering a good but performant solution (Meredith et al, 2004).

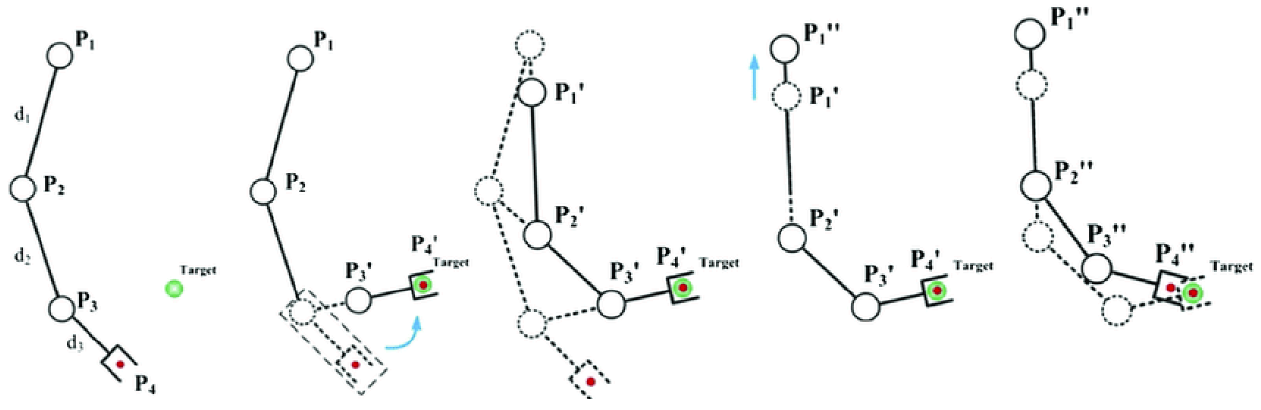

Figure 2.1: Visualization of the FABRIK algorithm (Aristidou, 2011). From left to right, the chain tries to move its Tip (red) towards the Target (green).

In order to limit the project's complexity and not deviate from its initial scope, it was decided to use a solver of the CCD class that is already included in the engine. Unity conveniently provides its own IK Solver implementation within the Animation Rigging package: the FABRIK algorithm (Aristidou, 2011). Not only is it an easily implementable solver, it is also approximately ten times faster than other CCD algorithms and requires a thousand times fewer operations than comparable Jacobian methods. Furthermore, it can handle chains in both two-dimensional and three-dimensional spaces. As a matter of fact, by its very design, FABRIK is always able to take a 3D problem and simplify it into a 2D case, reducing its complexity and required processing time. Additionally, the Animation Rigging package enables the use of a few different types of IK constraint manipulators. Two of the most powerful are the *Chain IK* and the *Two-Bone IK*. The first is capable of modelling a kinematic chain with any number of joints, while the latter only allows for two bones at most, but also guarantees faster compute times. The procedural locomotion algorithm developed required multiple of these IK methods to be solved at runtime (one for each limb), and the choice thus fell onto the faster, more reliable, Two-Bone IK.

2.3. Limitations

The main objective of the project was to design an algorithm solely focused on handling the walking animations of creatures. The final result, therefore, lacks the compatibility and integration with other systems that would otherwise be present in commercially available products. For instance, it completely ignores the engine's physics and gravity systems. This could no doubt be useful for games that do not make use of these features, but the walking algorithm is not guaranteed to work in environments that have them enabled. Furthermore, this walking algorithm may behave unpredictably when required to walk up or downstairs and along other step-like structures, as it does not present any specific behaviour when confronted with upwards or downwards stepping motion. Finally, only the locomotion animation is the one being procedurally generated, the rest of the character model can simultaneously follow other, more complex primary and secondary animations designed by the user, but these do require to be implemented separately.

3. Development and Analysis

3.1. Base Locomotion Mechanism

The term Locomotion is derived from the Latin words for “place”, *locus* and “to move”, *movere*. It is defined as the process of moving from one place to another. Humans and other quadruped animals normally achieve locomotion by advancing forward one limb at a time, maintaining stability while their centre of mass shifts. The Procedural Locomotion algorithm aims to simulate this behaviour, applying it to 3D models. Before any movement can take place, however, a character model needs to first be imported into the Unity project. From here, once it has been correctly set up inside of the “scene”, a *target coordinate* is automatically assigned to each of its legs. This coordinate is where the *tip* of the limb (known as “leaf” in the context of a Kinematic Chain, or “foot” when referring to a leg) will “want” to move towards. It can be seen as a location where, if positioned there, a leg would leave the rest of the body in a structurally *stable* configuration. At runtime, the Character Controller allows the model to be moved around by pressing the W, A, S, or D keys on a keyboard. As this happens, the Locomotion mechanism algorithmically generates a walking animation by constantly executing the following sequence of actions:

1. As the body’s centre of mass shifts, so do the targets, following its overall trajectory. While this takes place, all feet remain anchored to their original position touching the ground: these restrictions are imposed by their *Tip Constraints*, which serve the function of simulating how a foot remains on the ground while the opposite one advances. A step can therefore take place only if the opposite limb is not already in the process of doing so. In order to save on computational times, each limb holds an internal reference to the limb on the direct opposite side of the model.
2. At every engine tick – typically 50 to 60 times a second on sufficiently modern hardware – the algorithm computes the IK Solver for each of the leg chains,

rotating and displacing each part in order to fulfil the conditions imposed by the Tip Constraints.

3. During the locomotion process, if at any point in time the distance between the target coordinate and the tip becomes greater than a user-controlled *step size threshold* parameter, the tip temporarily deactivates its constraint, the leg chain moves towards the target and updates its anchoring position.
4. Once the tip is in its new, stable position, its corresponding tip constraint is activated once again, locking it into place until the next step is ready to start.

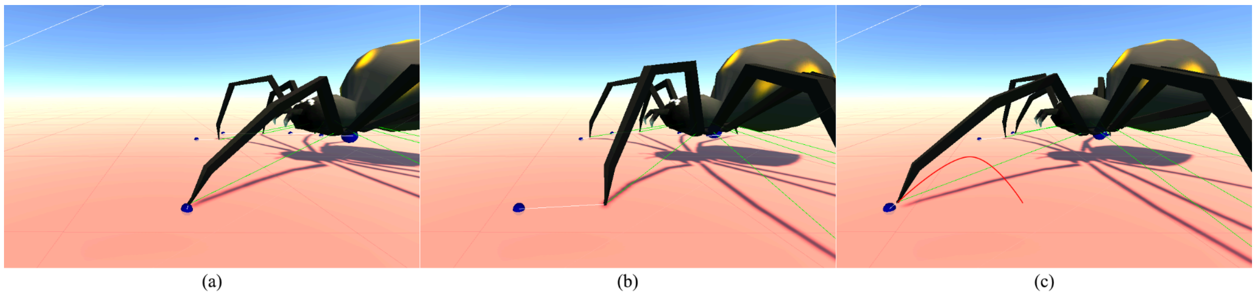

Figure 3.1: A spider model walking forward. The limb target is shown as a blue sphere. (a) The initial target is placed under the foot. (b) The body and target move forward. (c) The limb steps towards the target.

Notably, while the stepping motion described in the third step takes place, it would be possible for the body to continue to move. If this was the case, the target position would end up changing during the stepping process, forcing the tip to land on a different location than the one planned at the start of its transit. This location could potentially violate one or more of the constraints that are necessary for the limb structure to remain stable. For this reason, the stepping motion uses as its target, not the target itself, but rather, its coordinates are recorded at the start of the stepping motion, and those values are used as the final destination of the transit.

3.1.1. Target Anchoring

The previous section discussed how the leg tips move in accordance with the targets, but it is also necessary to understand how the targets themselves are placed and how they move. When a creature first spawns, a target object is also spawned for each of its limbs, and it is positioned at the same position as its corresponding tip. From each target, an invisible *ray* is then fired pointing downwards. If such a ray detects a walkable surface underneath, the point at which the ray intersects with the plane becomes the new target coordinate. This method is constantly called alongside the locomotion mechanism, and it is therefore used to keep the target object "anchored" to a walkable area at all times. This does, however, mean that, if a creature is placed too far above the ground, its targets will move downwards until they find a walkable surface, thus instantly teleporting the rest of the entity.

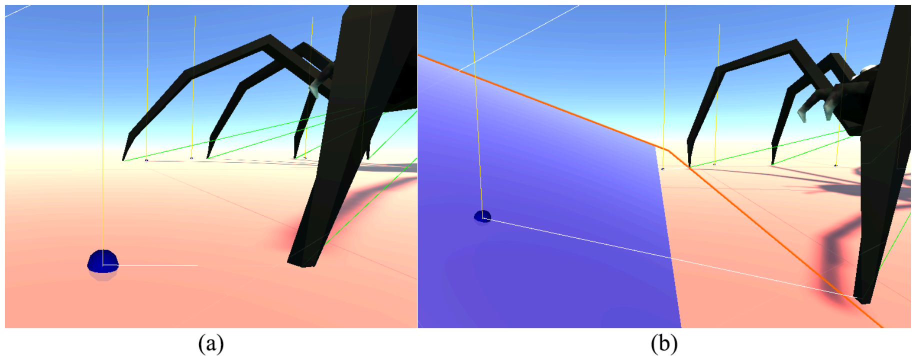

Figure 3.2: Target anchoring on the ground. (a) The downwards ray (in yellow) finds the correct position. (b) The ray hits a slope (in blue) and finds the correct position.

3.2. Arcing Trajectory

Contrary to how they would normally behave in nature, virtual limbs don't require the stretching and contraction of muscles to move. For additional realism, the gait changes and arcing motions each leg follows while walking needed to be simulated.

This was achieved through the use of a simple parameterized parabola, which the chain tip follows when moving towards its target. The used formula can be written as follows:

$$y = (-x^2 + d * x) * m$$

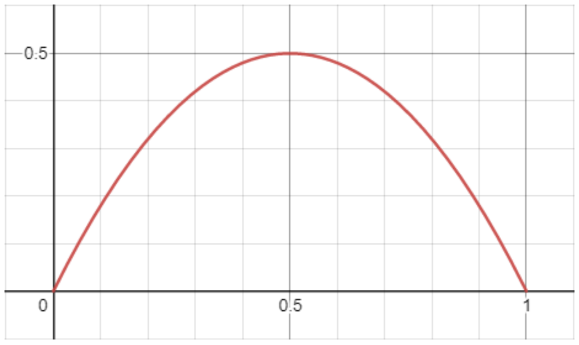

Figure 3.3: Example of the step function used, plotted using the Desmos free online calculator.

Where *y* is the desired height coordinate along the parabola, *x* is the position along the trajectory (normalized between the range 0 to 1), *d* is the distance in 3D space between the starting coordinate and the target coordinate, and *m* is a conversion factor used to determine the highest point of the curve. The *m* variable is itself computed through the given formula:

$$m = \frac{4h}{d^2}$$

Where *d* is, again, the distance between the two points in space, and *h* is the height (in world units) of the peak of the parabola. This conversion factor *m* was specifically added so that the *h* variable would map 1-to-1 with the world coordinate system used by Unity, and make it easier for users to adjust its value.

The given formulas may only seem to work in 2D space, but because the axis along which the upwards-downwards transit occurs is always only one – the *y*-axis if the creature is walking on a perfectly horizontal plane – the same function can be used in 3D space by simply *linearly interpolating* the values along *x* and *z*. This means the function needs to be computed once for the values of *x*, and once for the values of *z*.

Afterwards, both results are used to construct a 3D vector representing the desired position. It is also important to note how, even as the limb moves along the arching trajectory, the IK solver finds an appropriate set of values for its rotation, resulting in a seamless stepping animation.

The method illustrated above can take the *upwards axis* as one of its parameters. By default, this value is always the *y*-axis, but it is possible to specify a different one, choosing from *x*, *y*, and *z*. In doing so, the function finds the parabola values using the remaining two axes and subsequently interpolates the found coordinates. This process, however, lacks the ability to seamlessly transition from vertical steps on a horizontal plane to horizontal steps on a vertical wall. Improvements in this aspect could be attained by computing both parabolas and interpolating the target position between them. Another possible option could be to discard this mathematical-based approach entirely and instead rely on Unity's built-in Animation Curve datatypes.

3.3. Locomotion Patterns

As Earth's lifeforms evolved throughout the years, their walking patterns adapted to the changes in size and mass, sometimes opting for unique locomotion solutions. Some animals walk by keeping three of their feet on the ground at all times, while others can sprint and remain airborne for a few seconds in between steps. In order to replicate a portion of these different styles of motion, the initial position of the limb targets is asymmetrically displaced: using a simple quadruped creature as an example, the front-right and hind-left targets are moved further up ahead, or behind, by a *target displacement* factor *k*. As the creature starts moving, the resulting pattern will approximate real-life behaviour, albeit not necessarily in a perfectly faithful manner. The value of *k* is inversely proportional to the number of limbs present on the creature to maintain a plausible and convincing distance between them and from the main body. Figure 3.5 shows a top-down view of a spider model, where its targets have been shifted.

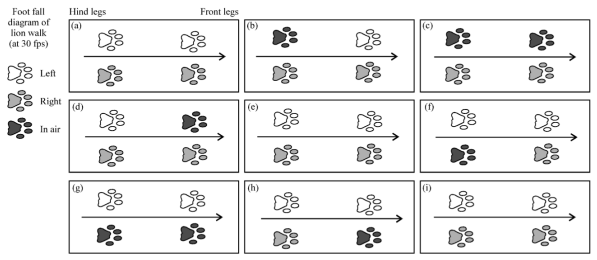

Figure 3.4 (above): Footfall phases of a lion walking (Bhatti et al, 2015).

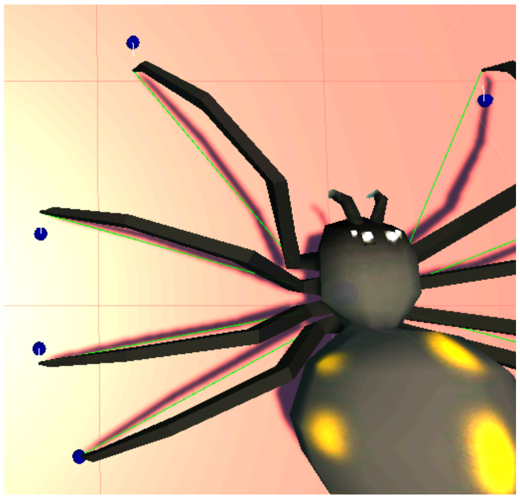

Figure 3.5 (right): Top-down view of a character whose limb targets have been shifted.

The method in charge of determining a plausible displacement presumes the limbs are internally ordered in an alternating fashion, with each index number indicating whether they are to the left or right of the body. The procedure then cycles through all limbs and shifts them. A *disparity* counter is increased by one after moving every even-indexed limb in order to move forward only those leg tips whose neighbours have been moved backwards. The opposite is also true for odd-numbered indexes, resulting in a configuration where opposite limbs are always displaced by different amounts. A portion of the code used in this procedure is illustrated in Appendices I and II.

3.3.1 Faster Motion

Running locomotion patterns, such as trotting and galloping, have not been directly implemented or tested. However, given how they can be replicated by changing the limb and target positions, they should be easily implementable through the simple manipulation of the adjustable variables, such as *step distance threshold*, *step duration* and *target displacement*. A potential issue regarding faster motion animations can arise from the way the main body element behaves during locomotion. Animals in real life change their gait depending on their pace, moving their bodies upwards and downwards both for increased aerodynamics or due to changes in velocity and momentum. The presented approach partially ignores the speed at which these creatures move, and instead focuses simply on adapting the position and rotation of their bodies to match leg distribution and the centre of mass coordinates, as discussed in Section 3.4. Within the engine, the project does, nonetheless, give an option for increased movement speed through the use of a *sprint* command, accelerating *step duration* by a *sprint multiplier* factor while the “shift” key is held on the keyboard.

3.4. Weight Distribution

Section 2.3. illustrates how this algorithm does not make use of the gravity and physics engine provided by Unity, nor does it employ any other third-party libraries that implement it. Instead, some of their effects, such as balance, are simulated in a simplified mathematical manner, strategically displacing the starting locomotion targets (Section 3.3) and adjusting gait and limb coordinates (Section 3.6.1) in order to realistically replicate a living creature’s behaviour. Weight is instead represented by a series of components that can be attached to each body part. These components contain a *weight* value, ranging from 0 to 10, that users can adjust, and which indicates how impactful each specific part of the body is when computing the creature’s movements. The maximum value of 10 is also arbitrary and can be easily changed from within the project’s codebase. Deciding where to attach these weight components and what values to assign to them is left to the end-user. Using the weight system is, in fact, entirely optional. By default, even without a weight component coupled to it, each segment of the model is given an internal weight value of 1, meaning it does not influence the locomotion calculations, neither negatively, nor positively.

3.4.1. Weight and Speed

The weight values described above behave like multipliers, adjusting the speed variables before they are used in the movement and rotation functions. The higher the speed, the faster the animations are resolved. More often than not, the speed values are floating-point numbers, representing the number of seconds a given action should take.

As one might expect, higher weight values have the opposite effect, slowing down the locomotion and turn animations.

Figure 3.6 shows a list of all the ways *weight* affects movement and speed. Along the Formula column, the *s* represents the default, user-controlled, speed value, and the variable *w* represents the weight of the corresponding component. The term w_b serves to specifically indicate the weight of the body element alone.

Interaction	Formula
Movement Speed	$M = s * \frac{1}{w_b}$
Step Speed	$S = s * \frac{1}{w}$
Rotation Realignment Speed	$R = s * \frac{1}{w}$
Turning Speed	$T = \frac{w_b}{s}$

Figure 3.6: Table showing how each speed value is influenced by weight.

It may seem surprising to see how the Turning Speed formula does not follow the same pattern as the other ones, using *s* as the negative multiplier rather than *w*. This is because the resulting number is not actually used as a value for speed, but rather for the *smoothness* of rotation. This number is then passed onto a *SmoothDampAngle* method, the objective of which is to smoothly rotate the creature along the *y*-axis. On the other hand, the movement speed is solely dependent on the weight of the main body unit, thus ignoring how heavy limbs are. Finally, the relationship between these three factors (initial speed, weight and final speed) is a linear one, meaning a steady increase in weight implies a steady decrease in the resulting velocity. This correlation is not necessarily representative of a real-world example. The general movement speed results do appear to approximate it to a sufficiently high degree, but this is not the case for the stepping speed, which changes dramatically when the character sprints. The stepping speed thus passes through a limiting function which reduces the direct impact the sprint multiplier has on it.

3.5. Multiple Body Sections

When modelling the 3D versions of most quadrupedal mammals and reptiles, it is often easier to design a single body component from where the head and limbs spring forth from. When designing arachnids or other insect-like creatures, it may be more straightforward to instead create different body sections, each with a pair of legs for both sides. By merging these together into a shape that mimics an arthropod's thorax (or abdomen) it would be possible to construct a creature resembling a centipede or a millipede. The Procedural Locomotion algorithm may be able to accommodate such creatures by applying some minor additions to their structure. For instance, each body segment could be organized in a *spine* kinematic chain, separate from the limbs' one. Starting from the main section, which is often the one closest to the head, each body segment would need to first adjust itself in accordance with the neighbouring elements in the spine. Only afterwards would the IK solver of each limb be calculated.

3.6. Adapting to the Environment

After having developed a simple movement algorithm it was necessary to give both the limbs and the body a set of positional and directional constraints. This was done in order to limit the number of possible edge cases the locomotion system could stumble into and subsequently fail, leaving its limbs in an unrealistic and unstable configuration. This section describes a few of the ways the system behaves against these situations and how they are implemented.

3.6.1. Centre of Mass

Along with other necessary data, each creature also internally stores the coordinates of its Centre of Mass. This point is however not found through a standard physics formula, but rather it is computed by averaging the positions of each of the character's components. The location vector of every body element with a weight component is collected and a weighted mean is calculated. The result is, therefore, more akin to a geometric centre, or centroid, than an actual centre of mass, but it can nonetheless serve as a useful variable when computing movement trajectories or gait and rotation changes. This is one of the reasons why the main body should, in most cases, be given a weight slightly higher than that of a single leg. As always, the weighted average is also re-calculated at every engine's tick to update its position as the locomotion process takes place.

3.6.2. Gait and Rotation Changes

A walking creature's gait is determined by how all of its components change while moving. In this study, however, this feature is incredibly simplified for the sake of computational time, at the expense of realism. The only two factors that are taken into account when designing the gait movement are the body's overall position and its rotation. Whenever a character moves, the Gait method first determines how its rotation should change. A simple and often employed way to find the desired spin values is by casting an invisible, virtual *ray* from the body's centre downwards. Whatever other object the ray intersects with is then considered the "ground" on top of which the creature is walking. The algorithm then analyses the normal vector of the ground's surface, also known as the *ground normal*. The difference between the creature's current angle and the ground normal angle would then be used to calibrate the body's target rotation. This approach does, however, pose a few disadvantages, one of which is the inadequacy displayed when confronted with rough terrains. Over a perfectly smooth surface, ground normals computed very near to one another would result in a perfectly smooth vector gradient (within computational feasibility); above uneven terrains, however, ground normal vectors in the general vicinity may end up showing an excessive level of variance. A possible solution to this issue could be reached through the use of *spherecasts*, rather than the *raycasts* described above, as proposed by Schofield in his work on Procedural Wall Walking Spiders (Schofield, 2020), where they are also used for anchoring, wall detection, and wall climbing.

The solution this locomotion algorithm uses is, instead, achieved by comparing the height differences between the front and hind sets of legs. The Gait method reads the positional values of 2 limbs and constructs a right triangle from three points in 3D space: the front tip, the hind tip and the intersection of their respective *y*-axis and *z*-axis below. By applying the arcsine function to the lengths of its two catheti, it is possible to obtain the values of the two acute angles in the resulting triangle. Only the smallest of the two is however necessary. This process is repeated for every *front-hind* leg pair present in the creature. A weighted average is then calculated from all found angles, and the obtained value is used as the body's rotation along the *x*-axis. The *z* component of the rotation is computed similarly, with the sole difference being the analysis of *left-right* limb pairs rather than *front-hind* ones.

Whether the *sign* of each angle value is positive or negative is also necessary to know if the creature needs to be rotated forwards or backwards, to the right or to the left. These signs can be determined through a simple list of operations. The first step in this process is to find, for every tuple of front-hind limbs, exactly which one is in front and which one is behind relative to one another. This can be done by taking both of their coordinate vectors, namely *a* and *b*, and multiplying them by the *x* component of the creature's *global forward* vector, a vector that describes the direction in which the character is moving towards. This operation results in two new applied vectors, which can be called *A* and *B* respectively, that start from the center of the creature and end at the tip of each limb. Finally, if all components of the *A* vector are greater or equal to those of the *B* vector, then it can be said *a* is ahead of *b* and *b* is behind *a*. The opposite also holds true if *B* is greater than *A*. Having said that, if a creature were to be climbing on a perfectly vertical surface, the given tuple of tip coordinates would lie vertically on top of one another. The global forward vector would therefore read 0 on its *x*-axis, there would be no appreciable difference between the *A* and *B* vectors, and the algorithm would simply default to the first one as the ahead vector. Nonetheless, if that was actually the case, determining which limb precedes the other would be unnecessary, as the target rotation of a vertical body would be either 0 or 180 degrees, both of which remain invariant with respect to their sign.

Once the forward-behind relationship has been established, the two resulting vectors are compared. If, for instance, the front vector is higher along the *y*-axis, its relative angle value is multiplied by -1, rotating the object backwards. Otherwise, the angle maintains its original sign. To find the *z*-axis rotation sign, both the ahead & behind and the sign processes are computed on the left & right set of pairs. It's important to note how these rotations along *x* and *z* are relative to the *local* set of coordinates of the creature, and do not directly modify the quaternion values used by the *global* vector space the engine uses.

Rotation changes are computed at every engine tick, but they are not applied until all legs are both stationary and in a stable position. If this was not the case, the limb targets would rotate mid-step, forcing the creature into an unstable configuration. On the engine's following tick, the algorithm would try to find new rotation values relative to the new target coordinates, and would keep trying to do so on every tick after that, without ever finding a stable configuration, thus remaining stuck in an infinite cycle. This solution has the unfortunate side effect of adding a slight delay between each step-rotation cycle. In extreme cases, the body may not align correctly until the entity has stopped moving. Once the body has been correctly rotated, its correct position is determined by simply ensuring the entity's base lies at the same height of the ground.

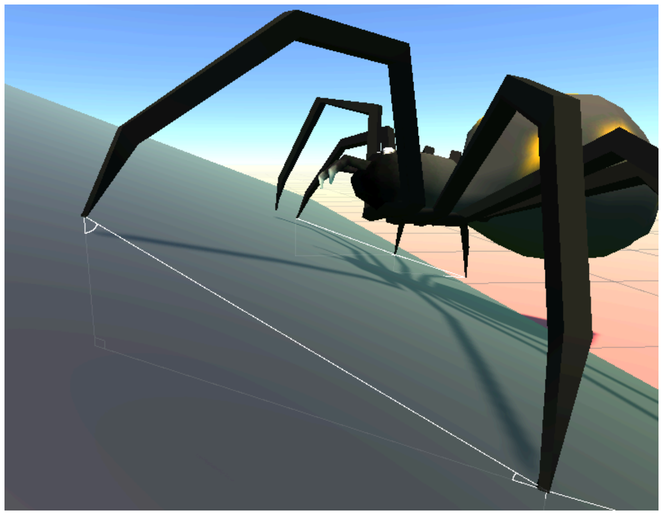

Figure 3.7: Spider character on a slope. The front left limbs form a right triangle (white).

Similarly to most of the procedurally animated components in the project, for additional realism, both the rotation and positional adjustments undergo a linear interpolation distributed throughout a number of frames, smoothing their changes over a given time. The speed at which these occur is determined by a *realignment speed* variable. This speed is also influenced by the weight of the body, which slows it proportionally.

3.6.3. Directional Limits

When a rigged creature is confronted with a wall or cliff where the ground's height difference is higher than the *maximum chain length* of its limbs, rather than fall off or teleport to the ground below, the Movement Controller component denies any further movement in the direction of the drop. This is achieved by a combination of the *raycast* system described in Section 3.1.1 and an *awareness circle* that wraps around the creature, running parallel to the ground surface. A limb's maximum chain length is calculated by simply adding up the distances between each pivoting joint along the chain. This algorithm computes a list of tuples, consisting of a pair of floating-point values: the starting and ending degrees of an arc running along the awareness circle, as depicted in Figure 3.8. These values are then normalized and used to highlight a number of arc segments around a unit circle. Once the delimiting arcs have been found, the Movement Controller reads the directional vector of motion, a unit vector provided by Unity whose centre is the same as the one of the unit circle. This vector then passes through a 2-argument arctangent function, converting it into a value in degrees. If this number is found to be within one of the limiting arcs, the Movement Controller does not allow the creature to continue to move in that direction. Similarly to the other movement and position control methods, this algorithm is called at every engine tick. If the Movement Controller takes in an 8-way input, the Arc Limit algorithm will find a maximum of 8 arc segments. When choosing which range to consider between (*a*, *b*) and (*b*, *a*), the algorithm always selects the lesser one, that is, the one with the lowest absolute difference between *a* and *b*.

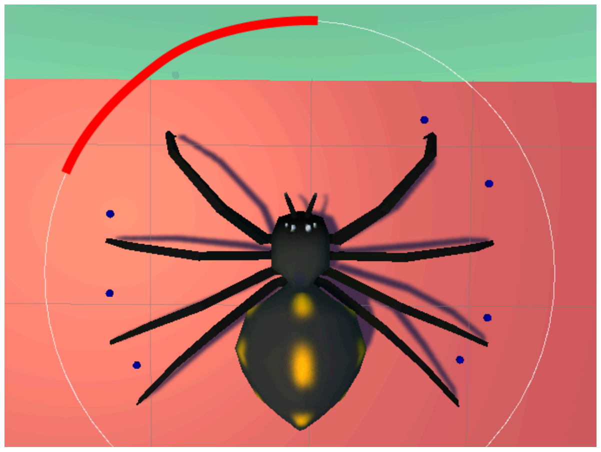

Figure 3.8: Top-down view of an awareness circle with an arc limit (red segment).

3.6.4. Walls and Obstacles

A walking system may also need to be able to recognize which areas it can traverse and which walls it cannot go through. To achieve this, the locomotion procedure of Section 3.1. also implements a *traversability* check. If this check is not passed, the leg does not start its transit towards the target, even if all other constraints are satisfied and all other checks are successful. Whether a step is considered traversable or not is decided by the casting of a ray from the body's Center of Mass, towards the target anchoring position. The ray is then “trimmed”, a step that cuts off the rest of the ray from the target point onwards. Afterwards, the end extremity of this ray segment is shortened by 2%, effectively scaling its length by 98%. This allows the system to ignore any obstacles that may reside in the immediate vicinity of the target location, which often produce false-negative results whenever an anchor precisely lies on a surface. If the line segment that was just described interests any game object that holds the tag “Untraversable”, the target location is deemed *untraversable*, and the stepping motion does not begin. Notably, this does not apply to the motion of the body, which is able to move unimpeded in any direction. In order to prevent this behaviour, this system is typically expected to run in conjunction with the directional limiter of the previous section (Section 3.6.3) but is not required to do so.

3.7. Automatic Set up

As specified in the Constraints paragraph of Section 2.1, a specific hierarchy should be present in any and all 3D models that are imported into the engine for the algorithm to behave as intended. These specific requirements and guidelines can be difficult to fulfil, especially since each imported creature needs to be set up in a very specific manner. For this reason, the “Entity” component class also enables the majority of these prerequisites to be automatically satisfied by the simple press of a button. Said button triggers the “Setup” procedure, which executes the following instructions:

1. Instantiates the IK Manager, attaches a Rig component to it, and sets it up.
2. For every limb, a Tip Target is generated and anchored to the ground, and it is aligned to the position of its corresponding tip.
3. Both the Two-Bone IK and the Constraints Controller are attached to each leg.
4. Each leg chain is then subdivided into 3 joint segments, which are necessary for the TBIK to work. These are the Root, the Tip, and the Middle, the section closest

to the midpoint of the chain. In a 3-joint leg, they correspond to the hip, the ankle and the knee respectively.

5. Each leg is given a reference to its opposite, linking every left-right pair of limbs.
6. Finally, the Animator, Movement Controller, and Character Controller components are attached to the parent object that houses the imported model.

Unfortunately, despite how powerful the Setup method can seem, there are still a few steps that have to be executed manually. Namely, the second step is unable to progress unless all limb objects have been properly added as references to the Entity controller's list. As such, a list of strictly user-controlled steps needs to be followed before the tool can be used:

- The Entity class needs to be attached to the parent object.
- A reference to the model object has to be added in the "Body" field of the Entity component. This object should contain the armature as one of its direct children.
- From the armature hierarchy, all limb roots need to be added as references to the Limbs Objects list. If necessary, the number of limbs of a creature can be set by writing it as the length of the Limbs Object list in the inspector.
- Walkable surfaces need to be on a layer called "Ground".
- Untraversable game objects need to be tagged with the "Untraversable" keyword.

3.7.1. Collider Generation

A group of issues that might arise when designing gameplay elements within a project are the possible physical and virtual interactions between its game objects. For this reason, the project also provides a simple utility for the automatic coupling of 3D colliders to the attached creature. If the given model does not already implement its own set of mesh colliders, the Entity component provides three options to achieve this:

- *Don't Generate Colliders*, where no colliders are generated.
- *Generate General Collider*, where only one Box Collider is added. Its size and centre are adjusted to match the object's general shape by approximating a box encapsulating it.
- *Generate Limb Colliders*, where a capsule collider is generated for each limb segment. These collider shapes are not designed to approximate the shape of the limb. Rather, they are simple capsules that follow the general length and diameter of the segment.

4. Discussion

An integral part of this study was the testing process, the developed program underwent a series of tests, aimed at analysing and assessing its capabilities and performance. These trials were conducted on the Unity engine itself, supplying a range of 3D models with varying numbers of limbs, and checking how the algorithms behaved when confronted with different terrain types.

4.1. Testing Multiple Models

The procedural walking algorithm was applied to a total of 4 creatures, consisting of a 4-legged mechanical arachnid, a 4-legged alien creature, an 8-legged spider, and a centipede-like creature with 14 limbs. This last character was constructed by concatenating 7 sections of the mechanical spider, each of which had 2 of its limbs removed. The Spine IK features that were suggested in Section 3.5 are, however, not included in it. All of its sections instead behave as one entity. All the models used were freely available and had been downloaded from Unity's Asset Store. Links to their respective pages can be found in the References section of this study. This range of creatures was chosen in order to test the system's flexibility, observing whether it was able to reach the same results and behave in the same manner, even when faced with entities with unusual shapes.

4.2. Environmental Testing

The next step in the testing phase was the analysis of the creature's interaction with the surrounding environment. For this purpose, a range of different terrains was constructed and placed into the engine's world space. More specifically, a series of platforms presenting different shapes and ground conformations were laid out for the tested creature to traverse through. These terrain platforms may, however, contain an inherent bias where it could have been subconsciously designed to cater to the algorithm at hand, ignoring any potential shortcomings. In order to reduce this possibility, a more complex environment is also employed, which presents a slightly more authentic use of this algorithm in real-life situations. A link to this asset can also be found in the References section below.

4.3. Comparing Solutions

Using the same model, the study compared the procedural walking animations of the project against other available solutions. Unfortunately, while a quantitative comparison of Animated versus Procedural can be easily achieved by analysing their capabilities and their performance, a qualitative comparison can be tricky to carry out. Nevertheless, a list of advantages and disadvantages each of the two presents can be observed.

One of the strengths of the illustrated algorithm-piloted locomotion lies in its ability to both adapt to the environment and seamlessly change movement direction. A basic character with just one walking animation would require a list of transitions to animate it veering to its left or to its right. Those without the ability or resources necessary to do this often instead resort to a simple rotation along the character's vertical axis, which often looks unnatural and may ruin a player's immersion. The same can also be said about the transitions between different movement speeds or between idle, walking, and running forms. A procedurally animated character would instead adapt to any speed, as its limbs are able to move at any velocity and independently from one another. A polished character with hand-made animations will no doubt be able to smoothly shift from walking to running, but others may not be able to afford such luxury. It can obviously be argued how, due to the simplicity of free, publicly available assets such as the ones used for testing, simple models are by their very design not expected to display a high level of polish. But this was precisely one of the main reasons this procedural locomotion tool was developed, granting even the most bare-bones models the ability to easily move in 3D space.

4.3.1. Performance

As this animation process hinges on the computer deciding how to move a creature, there is bound to be a non-zero amount of computational overhead that arises when comparing it to predefined animations. Premade animations usually rely on a simple interpolation of the position and rotation of a 3D model along a list of predetermined trajectories. These paths and their keyframes are often designed with external software and come already packaged inside of the model's file. The engine's job is to then simply apply these transformations within its world space over a specified period of time. How each game engine implements these processes can vary, but it is expected to be a more efficient procedure than the one described in this research.

In order to test these differences, two perfectly identical scenes were set up inside of Unity, containing both a simple walkable terrain and a spider creature. The only difference between the two was the animation component that was attached to the spider: one used the algorithm implemented in this study, while the other simply followed the animations that came prepackaged with it.

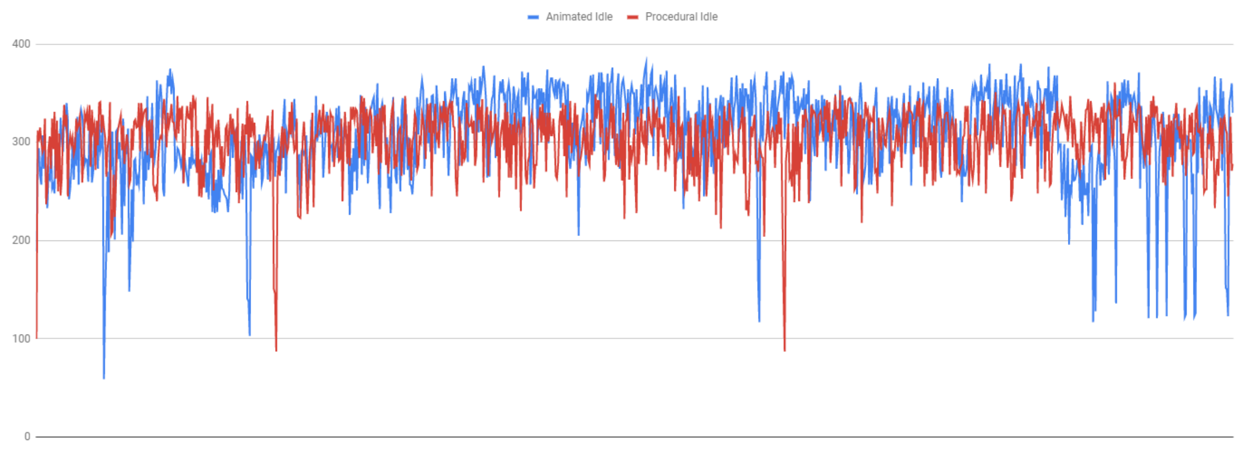

Figure 4.1: FPS timing plotted over a 30 second period of an idle animated spider (blue) and an idle procedurally animated spider (red). Higher is better.

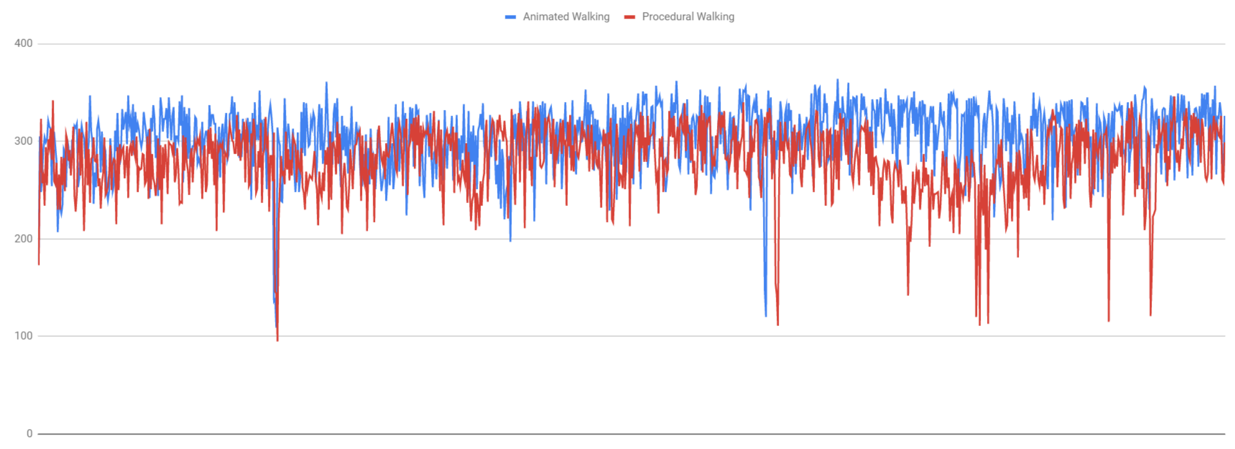

Figure 4.1: FPS timing plotted over a 30 second period of a walking animated spider (blue) and a walking procedurally animated spider (red). Higher is better.

The level of performance of a specific algorithm can be calculated by comparing many different aspects, looking at both the software and the hardware it runs on. In the context of video games, the number of *frames per second* (FPS) is usually considered a fairly good estimate of general performance. If two graphics-intensive tasks are compared, a higher average number of frames per second typically indicates how a certain task is more performant than the other, or rather, that said process does not significantly impact the various underlying processes of a machine. Average FPS was therefore chosen as the only metric in the experiment's measurements. A total of 4 tests were conducted, all on the same model: the 8-legged spider introduced in Section 4.1. The first two analysed the performance of a "traditionally animated" spider and compared it against a "procedurally animated" method that implements the Locomotion System, both in their respective idle state. The other two tests were instead carried out by juxtaposing the two spiders as they walked forward. All tests were recorded over a period of 30 seconds. Their results are plotted in the graphs of Figures 4.1 and 4.2 respectively. From the gathered data it can be seen how the idle animated spider displayed a mean FPS value of 323, while its procedural counterpart only averaged 307 FPS. The walking comparison displayed the same kind of results, with the animated spider exhibiting slightly lower values than its idle version and averaging around 309

FPS. The procedurally generated walk timings instead decreased by 15 frames, lowering the average down to 294 FPS. Overall, the locomotion algorithm presented a drop of approximately 3% in average framerates.

	Idle	**Walking**
Animated	323	307
Procedural	309	294

Figure 4.3: Table depicting the average FPS recorded in all 4 tests.

Interestingly, the animated spider displayed lower, more frequent minimums throughout the idle experiment, whereas the procedural one presented more frequent drops in the walking test. These troughs throughout the graph could be attributed to a specific behaviour the Unity engine exhibits when dealing with objects moving in the scene, but the lack of a clear periodicity in their frequency may point towards a different cause.

Supposing the relative difference between the Animation and Procedural approaches can be linearly projected onto a more complex environment, the 3% difference in performance could be considered a worthwhile tradeoff. This could, however, not be the case and the performance hit could scale exponentially, a much more likely scenario. Future developments will undoubtedly require a separate experiment with a higher number of concurrent animated characters, akin to a stress test able to record the algorithm’s performance in its worst-case scenario.

5. Conclusions

Throughout this study, it has been shown how a Procedural Locomotion algorithm can be used to animate non-bipedal characters in a realistic manner. Sections 3.1 and 3.2 illustrated how this can be achieved by combining Inverse Kinematic leg chains, a pelvis-based motion control, and a smooth arching step transit system. Moreover, Sections 3.4 and 3.6 showed how characters that employ such systems can also greatly benefit from simple obstacle avoidance and user-controlled weight components. Their usefulness is, nevertheless, heavily dependent on the kind of project they would be applied to. This project is however undoubtedly simplistic and rough, but it can function as a stepping stone for the future development of cheaper, more user-friendly solutions to procedural animation, at least for what concerns multi-limb creatures.

5.1. Shortcomings

The completed project has its fair share of flaws and shortcomings, part of which can be attributed to the inherent complexity of procedural algorithms, a lack of polish, or simply due to time mismanagement issues encountered throughout its development. The term “non-humanoid” itself, which was used in the initial proposal, is actually an umbrella term that can encompass an incredibly varied range of creatures of all shapes and sizes. While this research project was initially conceived to be able to work with any presented model, it quickly became apparent just how tricky it would be to visualise and account for every possible combination of body and limb forms. Consequently, the main focus shifted, veering slightly towards the procedural locomotion of bug-like creatures. Given more time, it would have certainly been interesting to test how well the same algorithms would adapt to other types of fictional animals, like centaurs and dragons likely to be used in video games set in a high fantasy setting, or even the extravagant kinds of robotic creatures found in futuristic works of fiction.

Another group of issues can be found in all components that handle the locomotion control. The Character Controller used was a fairly basic component that the Unity engine readily offers that can be implemented without too much difficulty. Its simplicity can however be the cause of a few problems, which is especially evident when a character is made to walk sideways. The procedure that takes care of turning suffers from the same challenges, as it does not take into account the positions of each leg, and just simply pivots the entire model around a centre point behind it. On top of that, the scripts themselves do not properly handle foreseeable errors and would greatly benefit from better exception handling. Many possible optimizations also became evident only after the project's conclusion. One of the most glaring examples of this is how the *front-hind* and *left-right* positional calculations are executed at every frame. Unless a creature is able to change its shape or limb conformation at runtime, these kinds of computations should be performed only once, either at the start of the initialization process or during the model's setup stage discussed in Section 3.7. Once these variables are found, they can be directly assigned and stored within each limb as references, similarly to how the "opposite" reference is handled in Section 3.1.

Finally, with regards to the testing methodology, perhaps it would have been more appropriate to design both quantitative and qualitative measurements, proposing a questionnaire to determine how realistic the locomotion animation is perceived by a varied number of viewers. Models could also have been tested by generating a Navigation Mesh for an obstacle course and have the model run programmatically through it.

5.2. Future Prospects

The project's in-engine constraint can be regarded as a double-edged sword. On one hand, it allows developers to forgo external software. On the other, it demands a significant amount of features for the tradeoff to be worthwhile. Many of the distinctive tools that multiple animation software offer could be ported into the system. The most obvious examples are Rotation Limiters, which allow users to define minimum and maximum rotation values on a per joint basis, and would certainly make a great addition to the toolset. Other Inverse Kinematic related features include *IK Hints*. When a limb goes from a straight configuration to a bent one, the joints in between the root and the end effectors can have a very large number of possible arrangements for the algorithm to choose from. A Hint target helps reduce this amount, as it can give a preferable direction for the middle nodes to move towards. Despite their usefulness, an overabundance of tools may run the risk of overwhelming users that are not familiar with these kinds of programs, deviating from the initial proposition of an "easy to use" system.

Other future developments the Procedural Locomotion system could benefit from include: better feet placement detection to avoid holes, a more thorough analysis of the imported models upon set-up to automatically detect their limbs, and a *temporal* component to account for speed when generating the gait animations. Likewise, further improvements could be applied to the limb IK systems. In Section 2.2 it was noted how the Two-Bone IK approach is only capable of handling a maximum of three joints, ignoring a portion of the intermediate nodes in models that display four or more. Rather than ignoring the in-between joints, limbs with more than 3 sections could use multiple TBIKs, one for the general Root-Mid-Tip articulation, and other, smaller ones for the sections in between. The first would help maintain the overall shape of the limb, regulating its stiffness. The intermediate ones would work on their own. Differently from the Chain IK method, computational complexity should only increase linearly since each IK Solver would be solved separately.

A more robust "limb decision system" could also be programmed in the future. Such a method could choose which limbs have the right to start their stepping transit by storing their requests on a priority queue, either in an FCFS manner (First Come, First Served) or through a more sophisticated methodology. The body rotation component could also be incorporated into this queue-based solution, allowing it to realign itself with the ground after every completed step.

Finally, it would be interesting to explore a material-based interaction system between the feet and the ground. By adjusting parameters like surface friction and speed, it may be possible to simulate slipping and sliding on ice, or even trudging through snowy or sandy terrain.

References

UNITY TECHNOLOGIES, 2005. UNITY 3D. [computer program] *San Francisco, U.S*. Available at: <unity.com> [Accessed 30 June 2021].

RAWES, E.M., 2018. The Pros & Cons of Being an Animator. *Career Trend* [online] Available at: <careertrend.com/pros-cons-being-animator-19729.html> [Accessed 12 July 2021].

MULTON, F. et al., 1999. Computer Animation of Human Walking: a Survey. *Journal of Visualization and Computer Animation*, John Wiley & Sons, 1999, Volume 10, pp.39–54. DOI: 10.1002/(SICI)1099-1778(199901/03)10:1<39::AID-VIS195>3.0.CO;2-2.

ADOBE SYSTEMS INCORPORATED, 2015. Mixamo. [computer program] *Adobe Labs*. Available at: <mixamo.com> [Accessed 2 August 2021].

NEKKI, 2020. Cascadeur [computer program] *Nekki*. Available at: <cascadeur.com> [Accessed 2 August 2021].

CHANDRAN S. et al, 2020. Semantic Deep Face Models. *International Conference on 3D Vision (3DV)*, 2020, pp. 345-354, DOI: 10.1109/3DV50981.2020.00044.

KATRAGADDA, G., 2019. Deep Learning can democratize Animation and VFX. *Myelin Foundry.* [online] Available at: <myelinfoundry.com/deep-learning-can-democratize-animation-and-vfx> [Accessed 7 August 2021].

ISIKGUNER, B., 2014. Procedural Animation: Towards Studio Solutions for Believability. Ph.D. Nottingham Trent University.

HECKER C., et al. 2008. Real-time motion retargeting to highly varied user-created morphologies. *ACM Transactions on Graphics, Volume 27, Issue 3*, pp. 1-11. DOI: 10.1145/1399504.1360626.

MAXIS STUDIOS, 2008. Spore. [computer game] Redwood Shores, California, U.S: Electronic Arts.

RAIBERT M.H., 1990. Trotting, pacing and bounding by a quadruped robot. *Journal of Biomechanics, Volume 23,* pp. 79-81, 83-98. DOI: 10.1016/0021-9290(90)90043-3.

MISULTIN STUDIOS, 2021. Simple Procedural Walk. [computer program]. Misultin Studios Available at:

<https://www.unrealengine.com/marketplace/en-US/product/simple-procedural-walk> [Accessed 3 September 2021].

EPIC GAMES, 1998. Unreal Engine [computer program]. *Cary, U.S*. Available at: <unrealengine.com> [Accessed 3 September 2021].

ŠMÍD, A., 2017. *Comparison of Unity and Unreal Engine*. Czech Technical University of Prague. Available at: <dcgi.fel.cvut.cz/theses/2017/smidanto> [Accessed 9 September 2021].

KARIM, A. A., et al, 2012. Procedural Locomotion of Multi-Legged Characters in Dynamic Environments. *Computer Animation and Virtual Worlds*, Wiley, 2012, Volume 24 (1) pp. 3-15. DOI: 10.1002/cav.1467.

MEREDITH, M., et al. 2004. Real-Time Inverse Kinematics: The Return of the Jacobian. *University of Sheffield*. [online] Available at: <staffwww.dcs.shef.ac.uk/people/S.Maddock/publications/MeredithMaddock2004_CS0406.pdf> [Accessed 11 August 2021].

ARISTIDOU A., 2011. FABRIK: A fast, iterative solver for the Inverse Kinematics problem. *Graphical Models 73* (5) pp. 243-260. DOI: 10.1016/j.gmod.2011.05.003.

SCHOFIELD, P., 2020. U*nity Procedural IK Wall Walking Spider*. GitHub [online], Available at: <github.com/PhilS94/Unity-Procedural-IK-Wall-Walking-Spider> [Accessed 14 August 2021].

PRISM BUCKET, 2015. Animated Spider [computer program] *Unity Asset Store*. Available at: <assetstore.unity.com/packages/3d/characters/animals/insects/animated-spider-22986> [Accessed 1 September 2021].

DMYTRO, 2020. Spider Orange [computer program] *Unity Asset Store*. Available at: <assetstore.unity.com/packages/3d/characters/robots/spider-orange-181154> [Accessed 1 September 2021].

PRISM BUCKET, 2015. Animated Spider [computer program] *Unity Asset Store*. Available at: <assetstore.unity.com/packages/3d/characters/animals/insects/animated-spider-22986> [Accessed 1 September 2021].

LB3D, 2021. Insectoid Crab Monster: Lurker of the Shores [computer program] *Unity Asset Store*. Available at:

<assetstore.unity.com/packages/3d/characters/insectoid-crab-monster-lurker-of-the-shores-20-animations-107223> [Accessed 1 September 2021].

TRIFORGE ASSETS, 2019. Fantasy Forest Environment - Free Demo [computer program] *Unity Asset Store*. Available at: <assetstore.unity.com/packages/3d/environments/fantasy/fantasy-forest-environment-free-demo-35361> [Accessed 1 September 2021].

Bibliography

ZUCCONI, A., 2017. An Introduction to Procedural Animations. *Alan Zucconi* [online]. Available at: <www.alanzucconi.com/2017/04/17/procedural-animations> [Accessed 29 July 2021].

BERMUDEZ, L., 2017. Overview of Jacobian IK. *Medium* [online]. Available at: <medium.com/unity3danimation/overview-of-jacobian-ik-a33939639ab2> [Accessed 12 July 2021].

GILANI, M., 2018. Animating Beasts using the Procedural Way in Unity. *80Level* [online]. Available at: <medium.com/unity3danimation/overview-of-jacobian-ik-a33939639ab2> [Accessed 16 July 2021].

WEAVER DEV, 2019. Bonehead Procedural Animation. *Weaverdev* [online]. Available at: <weaverdev.io/blog/bonehead-procedural-animation> [Accessed 16 July 2021].

Appendices

Appendix I

```
int disparity = 0;
for (int i = 0; i < limbs.Count; i++) {
    limbs[i].DisplaceTarget(i, disparity, limbs.Count, transform.forward);

    if (i % 2 == 0) disparity++;
}
```

Appendix II

```
public void DisplaceTarget(int index, int disparity, int numberOfLimbs,
                           Vector3 forwardDirection) {

    float forwardDistance = stepSize / (numberOfLimbs * 2) +
                            stepSize / (numberOfLimbs * 4) * index;

    // Converts the disparity into a number sign (-1 or +1)
    int sign = Convert.ToInt32(disparity % 2 != 0) * 2 - 1;

    // Shift the target position by its direction and distance
    Vector3 targetPos = target.position +
                       (forwardDirection * forwardDistance * sign);

    // The initial target distance must be <= the max range of the limb
    Vector3 rootPos = root.transform.position;
    float distFromRoot = Vector3.Distance(targetPos, rootPos);
    int iterations = 10;

    while (distFromRoot > maxRange && iterations > 0) {
        targetPos -= (forwardDirection * 0.005f * sign);
        iterations--;
    }

    target.position = targetPos;
}
```

Marco Vincenzi - 2021