PRIFYSGOL
BANGOR
UNIVERSITY

School of Computer Science
College of Physical & Applied Sciences

Real-time simulation of a Myriapod, including movement and behaviour

---

Benjamin David Winter

Submitted in partial satisfaction of the requirements for the
Degree of Bachelor of Science
in Computer Science

Supervisor Dr Llyr Ap Cenydd

April 2018

Acknowledgements

I would like to thank my supervisor Llyr Ap-Cenydd for his continued guidance throughout this project and for providing incredibly useful documentation of his own previous work for research. I would also like to thank Cameron Gray for his detailed lectures and support in the planning stages of this dissertation.

Statement of Originality

The work presented in this thesis/dissertation is entirely from the studies of the individual student, except where otherwise stated. Where derivations are presented and the origin of the work is either wholly or in part from other sources, then full reference is given to the original author. This work has not been presented previously for any degree, nor is it at present under consideration by any other degree awarding body.

Student:

Benjamin David Winter

Statement of Availability

I hereby acknowledge the availability of any part of this thesis/dissertation for viewing, photocopying or incorporation into future studies, providing that full reference is given to the origins of any information contained herein. I further give permission for a copy of this work to be deposited with the Bangor University Institutional Digital Repository, and/or in any other repository authorised for use by Bangor University and where necessary have gained the required permissions for the use of third party material. I acknowledge that Bangor University may make the title and a summary of this thesis/dissertation freely available.

Student:

Benjamin David Winter

Abstract

Keyframe and data-driven animations are constantly used in game development for complex characters as a way to ease transitions between certain actions or movements. This dissertation looks at ways emergent characters in games can be created without the use of keyframe animations. The goal of this project is to create a fully emergent centipede in a virtual environment with no keyframe or data-driven animations, using only programming techniques and algorithmic animations to recreate a centipedes behavioural system. Using these techniques allows for more intelligent and reusable game characters and reduces the time and money it would cost to make and use data-driven animations and motion capture technologies. The end result is a real-time simulation of a centipede including all characteristics and behaviours they would normally show. Development from basic navigation with a primitive sphere to a working 3D model of a centipede roaming around a 3D environment.

Contents

List of Figures

Chapter 1

Introduction

This study will discuss some of the ways you can create a real-time lifelike simulation of a centipede as well as discussing various techniques to simulate a centipedes behaviour and movement, the centipede was chosen to test these techniques because of their uniquely aggressive behaviours. As well as the fact that, the centipede has incredibly dexterous and agile movement patterns, compared to other insects, animals and humans.

Centipedes are one of the most dangerous and aggressive creatures in the animal kingdom. Centipedes can be so dangerous in fact that, even Charles Darwin was frightened of some species, with one species being named the *Darwin's Goliath Centipede*. The Scolopendra gigantea will be the centipede this project will focus on, it is known as one of the most aggressive families of centipede, some of whose have a length of up to twelve inches and is generally found in northern South America. however throughout this dissertation there will be a number of times the Scolopendra gigantea is compared to other classifications of centipedes.

1.1 Aims

The aim of this project is to recreate a centipede's movement and behaviour in Unity [16], by creating a range of algorithms and techniques to mimic a centipedes actions, as well as designing and programming accurate behavioural and movement patterns, whilst remaining fully autonomous with no human inputs or animations.

This project will cover a series of physics and computational geographical algorithms to perform autonomous movement in a 3D environment, as well as creating a 'brain' for the centipede, in which it will choose to hunt, hide, find shade or simply roam.

1.2 Objectives

- Investigate which technique would be most suitable for a centipedes autonomous movement, either; NavMesh agents, gravity, neural networks and/or state machines
- Create a 3D model of the centipede
- Design and create a suitable test environment
- Create a behavioural system for hunting prey
- Create a behavioural system for finding shade
- Create a 'Cobra mode', which will cause the centipede to raise the front third of its body, used often in the animal kingdom when centipedes hunt their prey or climb obstacles

1.3 Hypothesis

A real-time virtual centipede will be able to navigate through a virtual environment with various pathfinding techniques and lifelike speed. The centipede will be able to demonstrate naturally, various behavioural categories of the Centipede, such as finding shade, wandering and also hunting for prey.

1.4 Dissertation Overview

The remainder of this dissertation is as follows: Chapter 2 will look at the background to this work, including three distinct sections, the biology of the centipede, which talks about the predatory behaviours of the centipede, the common habitats of the centipede and their taxonomy. The next section of Chapter 2 involves centipedes being used in Robotics, and discusses studies that use the centipede's unique movement for the locomotion of robots and why their movement could be used widely in the world of robotics for more robust and resilient movement. Section 2.3 and 2.4 of Chapter 2 discusses studies of different simulations of 3D movements, including insects as well as humans. Chapter 2 also discusses the different software this project used and why they were necessary.

Chapter 3 discusses the design of the test environments, the designing and rigging of the centipedes body as well as its legs. Chapter 4 goes over the alpha and beta implementation of the centipede and the various problems faced with different techniques, especially in the alpha implementation stages. This chapter also examines the disparate methods of achieving movement as well as the efficiency of some techniques used. Chapter 5 covers all this projects testing as well as the results of divergent tests. Chapter 6 concludes this dissertation as well as discussing potential future works for this project. The next chapter contains all of the references used in creating this dissertation. The final chapter of this dissertation is the Appendix that includes all of the code stubs of this project.

Chapter 2

Background

The background chapter of this dissertation includes three sections, that discuss various studies that relate in some way to this project. This chapter is split up into three sections: Biology, that covers the Guizze et al's [15] study on the predatory behaviours of certain centipede species, as well as Akkari et al's [12] work on the taxonomy, distribution and habitats of the Scolopendromorpha family of centipedes. This section also has independent research discussing the habitats of the centipede further.

The next section looks at studies regarding how centipedes can be used in the study of Robotics for robotic locomotion as well as developing them for rescue operations.

The next section of this chapter contains a study as well as an artificial intelligence system used in a video game that was sold commercially. These studies show in detail how artificial intelligence can be used to display real-time simulations of creatures, and different techniques that can be used to demonstrate behaviours and movements of such creatures.

The final section discusses the various applications and technologies used in creating this centipede and what benefit they had over other technologies that are commonly used.

2.1 Centipede biology

This section reviews some studies and thesis' of various biologists and zoologists that talk about the biology of different species of centipede, including some traits such as predatory behaviour and movement, as well as their taxonomy and habitat preferences.

2.1.1 Predatory behavior of three centipede species of the order Scolopendromorpha (Arthropoda: Myriapoda: Chilopoda)

Guizze et al [15] produced a study, showing the different predatory behavioural patterns of the order Scolopendromorpha. This study concluded a lab experiment using various specimens of Scolopendra as well as thirteen different type of prey, to map and describe their predatory behaviour. They analyzed video images of around 65 hours of recordings, and recorded 15 separate behavioural categories that describe foraging, prey capture, feeding, and cleaning habits. According to this study, 95% of all observations concluded with the centipede killing its prey. They witnessed that some stimulus was triggering the centipedes movement towards its prey in all observations. Their experiment also showed that centipedes are actively forage to seek food. Even showing that some of the centipedes would actually feed on plants if no prey is available despite being thought of as carnivores. This study also found that the size and aggressiveness of the prey actually affected the centipedes capture process as seen in figure 2.1. Not only this, but this study also found that some genus of Scolopendromorpha raise the first third of the body while the rest of the body remains adjacent to the substrate beneath them, as well as restraining prey in the front third of the centipedes body with the aid of locomotory legs, simultaneously confirming that some genus of centipedes hold prey between their ultimate pair of legs as seen in figure 2.2.

The study goes on to create a process diagram of how centipedes hunt their prey whilst defining separate categories that directly label each behavioural pattern and cross analyze each one.

Behavioral categories	*Scolopendra viridicornis*	*Otostigmus tibialis*	*Cryptops iheringi*
Antenna movement (M) - the extended antennae are raised and lowered repeatedly without touching the substrate or the prey.	X	X	X
Groping the substrate (Ts) - the extended antennae are raised and lowered to touch the ground repeatedly, during locomotion or when stationary.	X	X	X
Groping the prey (Tp) - antennae are raised and lowered to touch the prey.	X	X	X
Posture 1 (X1) - the ultimate pair of legs is raised vertically. The elevation angle is directly proportional to the intensity of the stimulus that triggered it.	X	X	X
Raising (E) - centipede raises the first third of its body as the rest of the body remains adjacent to the substrate.	X		
Capture (C) - insertion of forcipules into prey.	X	X	X
Containment (G) - prey restraint along the ventral region of the first third of the body, with the aid of locomotory legs. Legs of other segments are used as anchorage in the substrate.	X		
Onslaught (I) - displacement of the centipede toward the prey faster than the speed used in locomotion.	X	X	X
Pinch (P) - lateral movement of the head and the ultimate pair of legs simultaneously converging toward the prey.	X	X	X
Transport (S) - the centipede restrains the prey along the ventral region of the first third of the body, assumes the raising posture (E), and moves to another area of the terrarium.	X	X	X
Feeding (A) - scraping movement of mouthparts on the prey.	X	X	X
Scissor (Z) - prey restraint with the ultimate pair of legs.	X	X	X
Locomotion (Ç) - orderly movement of the legs, in which the animal's body is kept above ground and driven in a certain direction.	X	X	X
Antennae cleaning (La) - scraping movement of mouthparts on the antennae (from base to distal end).	X	X	X

Figure 2.1: Behavioural Categories of the Centipede [15]

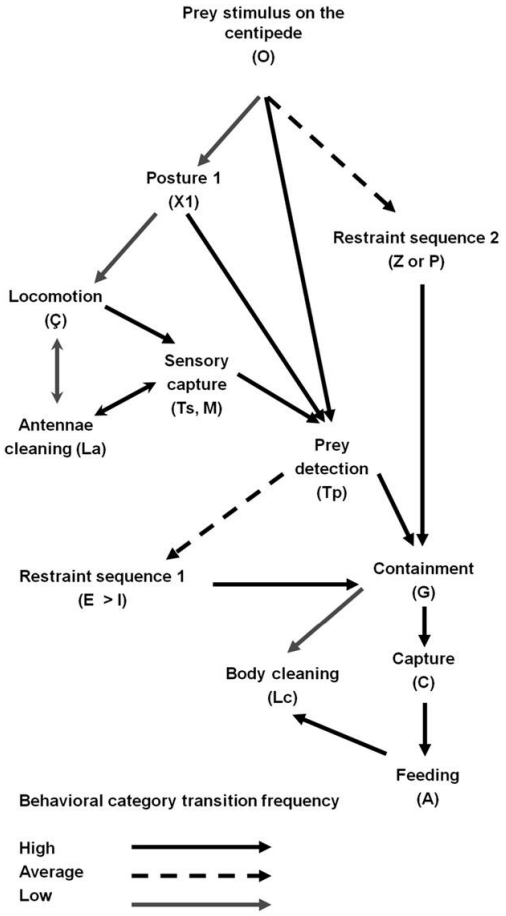


Figure 2.2: Predatory behavioural process [15]

2.1.2 The Scolopendromorph centipedes (Chilopoda, Scolopendromorpha) of Tunisia: taxonomy, distribution and habitats

Akkari et al [12] proposed this study to document more information about the habitats and taxonomy of the Scolopendromorph centipedes living in Tunisia. They found that of the six species, three had adapted to living in arid and semidesert biotopes and had much wider ranges compared to the other three species which live in more humid parts of the country, perhaps offering a correlation between humidity and distribution in centipedes. They studied the various habitats of each of the species, writing that the species who live in humid areas often took shelter in: overturned logs made of oak and different kinds of mixed heteregeneous woods, they were also found prodominantly in shrubs and underneath stones. The study even suggested it could be very possible to find some cave dwelling species of centipede in the near future.

2.2 Habitats of the Centipede

Centipedes, specifically the Scolopendridae of the phylum Arthropoda, mostly live in neotropical Biogeographic regions, specifically in savanna, grassland, forest or scrub forest terrestrial biomes [5]. Scolopendridae typically live in moist microhabitats, often burrowing in soil to make shelter. They also find shaded areas such as leaf litter, the underneath of stones or logs in order to keep moist, as they have no waxy covering on their cuticle.

2.3 Robotics

This section discusses some studies involved in creating centipede like robots, for both better locomotion across rough terrain, as well as for potential rescue operations across rubble and uneven surfaces.

2.3.1 Decentralized control scheme for myriapoda robot inspired by adaptive and resilient centipede locomotion

Yasui et al [9] produced an engineering study in order to create a mobile robot to mimic centipedes, as they said they potentially have the capability of producing highly stable, adaptive, and resilient behaviours. They set out to create a robot that had the same behaviour of locomotion even under unusual conditions. They observed changes in movement when part of the terrain was removed or even when the centipede is missing legs. They determined that the ground reaction force is significant for generating rhythmic leg movements, where they proposed that the movement of each leg is likely affected by a sensory input from its neighbouring legs. The study went on to create various 2D and 3D models to test various reflexes and reactions. Yasui et al [9] then proceeded to perform simulations using this model and demonstrated that the myriapod robot could move adaptively to changes in its own environment as well as its own body properties. They concluded that their findings could greatly help to aid designing adaptive and resilient robots under various circumstances.

2.3.2 Semiautonomous centipede-like robot for rubble - development of an actual scale robot for rescue operation

Ito et al [8] developed a multi-legged robot that mimics a centipede in order to adaptively traverse through complex environments. They achieved this by creating a robot with multiple rubber joints connecting the body to the legs to compensate for bumps on the ground. Ito et al [8], also mimicked the centipedes legs by moving them repetitively. They controlled the movement direction of this robot by pulling a series of wires installed through various links. Their study concluded that the robot they developed could move over rubble and uneven terrains and surfaces without damaging itself to the desired position.

2.4 Simulation

Simulation is the imitation of a situation or a process. This section discusses a study that created a simulation of an arthropod, adapted for dynamic and ambulatory movement. This section also looks at an artificially intelligent zombie, specifically looking at vertical climbing in real-time situations.

2.4.1 The Dynamic Animation of Ambulatory Arthropods

Teahan et al [6] set out to explore further possibilities into realistic real-time simulations of creatures with the ability to scale arbitrary surfaces as well as exploring their environment fully. Ap-Cenydd and Teahan presented a system where ground based arthropods would be dynamically animated in real-time, the system they created was capable of traversing realistically across complex environments, enabling complex emergent animations to form. The system they created used arbitrary path Gait adjustments as well as step adjustments across uneven terrain, this was accomplished by using three modular components, allowing for curved path as well as uneven terrain locomotion using sagittal elevation angles and an inverse motion interpolation algorithm. They used a rigid body to obtain the creatures absolute position and orientation, the system would then apply a virtual push force to the rigid body, pushing the creature in the eventual direction of the step. The Gait system they used would instruct each leg to take a step in sequence, even making the order in which the steps are taken depend on which arthropod subphyla.

2.4.2 Left 4 Dead zombie AI

This artificial intelligence system created by Valve [11] whilst not being animal or centipede related, provided great insight into how to make agents climb vertically whilst remaining fully in tact and looking realistic. The system did this by following six steps: The first step they called the *Approach* step, this involved the agent periodically testing for obstacles immediately ahead as it is following an already given path. The second step called *Find Ceiling* works after the obstacle has been detected, a hull trace determines the available vertical space above the agent. The third step or the *Find Ledge* step involved scanning the vertical space above the agent via a series of hull traces from lowest to highest to find the unobstructed hull trace, this trace would then mean that there was a gap in the obstacle in front or it would show the top of the obstacle. The next step is the *Find Ledge Height* step, this step used another hull trace downwards from the unobstructed trace to find the precise height of the ledge. The *Find Ledge Forward Edge* step is next and that involved a final series of downward hull traces that step backwards to determine the forward edge of the ledge. The final step is the *Perform Climb* step in which the system would pick the closest animation that matched the obstacles height.

2.5 Unity and C#

Unity [16] is a cross platform game engine, used mainly to develop three-dimensional video games and simulations. Unity supports both 2D and 3D graphics as well as having the ability to script C# as well as JavaScript within Unity. The game engine primarily uses the following graphical API's: OpenGL, Direct3D, OpenGL ES, WebGL and other specific API's that are used for private consoles. Unity is used widely in the video game development field as not only is it free for indie users assuring they don't make over $200k from the game they develop. It is also popular as it is incredibly simple to export games to either mobile, desktop, web or even gaming consoles.

C# is used primarily by Unity developers, as C# is an object-oriented programming language which has easy to lean syntax, whilst also being extremely versatile in offering powerful features such as nullable value types, enumerations, delegates, lambda expressions and having direct memory access. These features lead to C# being the most popular choice for developers programming in Unity, over other languages such as JavaScript [7].

2.6 Blender

Blender is an open-source, cross platform, free, 3D computer graphics software toolkit. Blender is mainly used for creating animations, 3D printed models, 3D applications and visual effects. Supporting 3D pipeline-modelling, animation, rigging, simulation, rendering, compositing and even motion tracking. Blender is well suited to individuals and small studios who benefit from its unified pipeline and responsive development process [2].

2.6.1 Bendy Bones

Blender defines Bendy Bones or B-Bones as "... an easy way to replace long chains of small rigid bones. A common use case for curve bones is to model spinal columns or facial bones." [3]. They work by treating the bone as a section of a Bézier curve passing through joints. Each segment can roll and bend to follow a point on the Bézier curve. The shape of these bones can be controlled using various properties or indirectly through neighbouring bones.

Chapter 3

Design

The design chapter consists of three sections that considers different design methodologies as well as talking about how they were implemented in this projects final centipede design and its environment.

3.1 Environment

Knowing the environment of the centipede as seen in figure 3.1, the environment of this centipede had to resemble something similar. Rainforests were chosen to be the most common environment for centipedes in the wild.

In order to create something similar to a rainforest, this virtual environment had to have certain objects that could be used for traversing over, navigating around, hiding in, etc. The first object created was the ground, this was to be the surface that the centipede walks on, a large cube was used roughly 90x the size of the centipede, this is due to the ground having to have enough room for the centipede to wander around. The cube was coloured a dark brown(R = 72, G = 63, B = 57) to represent the soil and twigs on the ground, such as the ones in figure 3.1. This material created in Unity was set to not include any *Specular Highlights* to create a dull material, and to ensure the ground had no metallic look to it.

Figure 3.1: Rainforest in Hawaii [14]

In order for the centipede to be able to find shade out of the light, there first must be some shaded areas for the centipede to find. Half logs as seen in figure 3.2 were created in Blender [2], they were made by cutting a cylinder in two, taking one half of the cylinder, hollowing it out and increasing the width of the log, an increased depth of the vertices groups were made to bulk the log out.

The half logs were a dark green(R = 14, G = 64, B = 31), this was to recreate something similar to a mossy log, also similar to the ones shown in figure 3.1. The log had to be squashed down in order to agree with the *Max Slope* being 60°, in order for the centipede to be able to walk up and down the log just like it would in the wild.

However, in order to make sure that the centipede wouldn't seek shade in the open, for example: If a tree was in between the light source and the centipede, it would be classed as a shaded area, even if the centipede had no cover. For this reason, it was best to have no tall trees or obstacles in the centipedes environment as it could have messed with the *Finding shade* function 4.2.3.

Ground obstacles were created out of spheres and cubes of various sizes and random rotations to mimic different obstacles centipedes might find in the wild, such as twigs, rocks and trees as seen in figure 3.3.

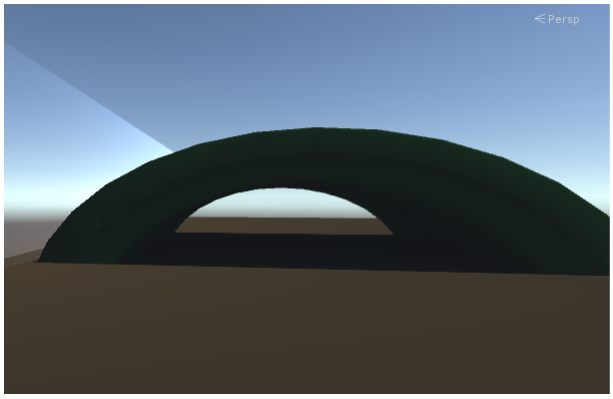


Figure 3.2: Half log in Unity

The last part of the environments design process was trying to get the right lighting for the environment. Even though rainforests are in hot and sunny countries, the grounds of rainforests are typically very dark as the rainforests trees block the suns light. Taking this into account, the centipedes environment had to be quite dark, but not too dark that the centipede was hidden as the fact that centipedes have been seen in rainforests means that they of course must be visible to man. The lighting for the centipedes environment was created by using the default skybox that Unity [16] offers but reducing the *Intensity Modifier* to zero. As well as having a single directional light above the test environment at a slight angle, this gives the centipedes environment a relatively dark look, whilst also creating shadows around the environment. In the wild the centipede often lives in humid areas [2.1.2], in order to recreate this humidity, *Fog* was added to the environment at a very thin density as to not impact vision.

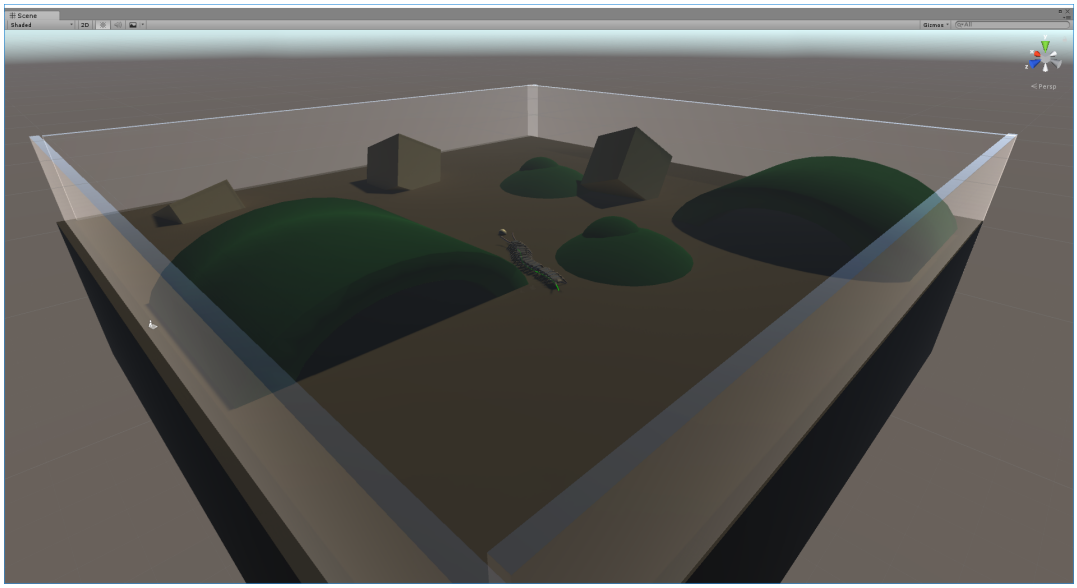


Figure 3.3: Centipedes environment whilst the centipede is wandering

3.2 Legs

The legs were designed in Blender [1] to not only look similar to the centipedes, but also be functional in finding the distance between each leg as to not bump into each other or even glitch through each other.

The shape of the legs were based of a Bézier curve, these curves are already shaped like legs, which made them very easy to slightly morph them into the correct shape so they fitted perfectly with the centipedes body. A Bézier curve is a parametric curve that uses the Bernstein polynomials as a basis [13].

At first when created in Blender [1] a Bézier curve is just a small thin line (see figure 3.4), mainly used for animations. Animators can attach an object to a curve and the object will move whilst following the path of the curve. However, an increased depth of the bevel of the curve creates a half tube with 3 distinct edges , the left of the tube, the right of the tube and the middle of the tube as seen on figure 3.5a. The half tube as it stands is very primitive, however Blender by default fills Bézier curves only half way so instead of filling only half of the vertices, fill all of the vertices (see figure 3.5b). This creates a hollow tube which will start to look like a leg. The vertices at the end can be sealed off (see figure 3.6a) as to not look hollow from the outside whilst not wasting memory by rendering pixels inside the leg which will never be seen. The leg will also need to be converted from the object mode *curve* to *mesh* in order to change some of the vertices. For this project one end of the leg off was sealed off to form a point, which will act as the tip of the centipedes leg. In Blender [1] it is easy to create armatures and bone structures, for the centipede two bones were needed for each leg, an upper bone that connects the upper part of the leg to corresponding body segment, and the tip bone which controls the tip of each leg (see figure 3.6b). This technique can be used for every leg the centipede has.

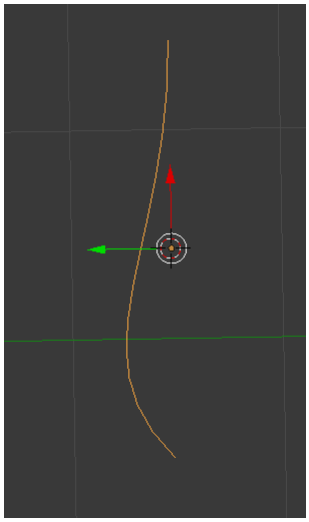
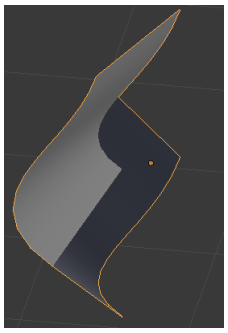
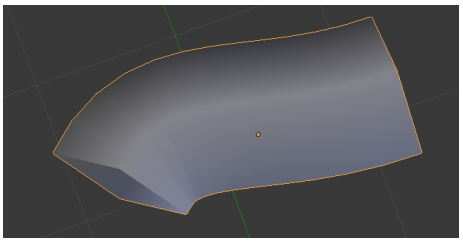


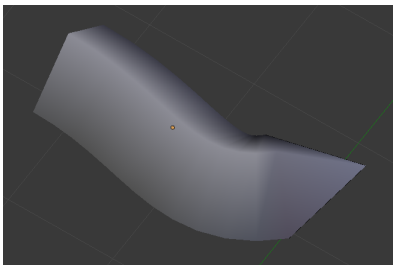
Figure 3.4: Bézier Curve in Blender



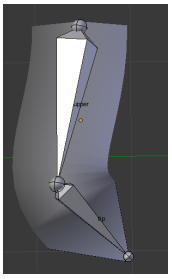
(a) Bézier curve with increased bevel depth



(b) Hollow Bézier curve with increased bevel depth



(a) Bézier curve with a tip at the end resembling the tip of a centipedes leg



(b) Bézier curve with two bones attached(Upper and Tip)

The colour design was based on figure 3.7 of the *Scolopendra Polymorpha*, the RGB colour was picked using Blenders Colour picker tool whilst clicking on the back legs of the picture below, the RGB values are: R = 0.195, G = 0.125, B = 0.053. This colour gives the legs a somewhat brown/orange look which was exactly what the centipede needed.

In order for the legs to be affected by code, they must first have objects that can be transformed and manipulated in Unity [16]. As stated previously, bones in Blender [1] are a modelling tool used to move limbs in various ways without destroying meshes or

Figure 3.7: Common Desert Centipede or *Scolopendra Polymorpha* [10]

having to change the shapes of certain vertices. In this project two bones were used for each leg, one for the upper part of the leg and one bone to control the tip of the leg. In Blender [1] it is very easy to control how much each bone controls by using the **Weight Paint** tool. Vertex Groups can potentially have a very large number of associated vertices and thus a large number of weights (one weight per assigned vertex). Weight Painting is a method to maintain large amounts of weight information in a very intuitive way. It is primarily used for rigging meshes, where the vertex groups are used to define the relative bone influences on the mesh [4]. These leg bones will be connected to a parent bone which will be in the centre of the corresponding body part.

3.3 Body

The centipedes body is segmented 21 times, 20 body segments and one head segment, the reason the head segment is separated is because the head will be the first parent bone and also will be manipulated by the coding separately, for transformations such as rotating upwards. The reason the body is segmented 21 times is because the genus *Scolopendra* typically has 21 segments and in order to provide more realism, 21 segments was best.

Each segment has a bendy bone that is connected to the previous bone(if that's possible), the bone ahead(if that's possible), as well as being connected to two leg bones on either side of the segment. The body's bone has minimal weight on the legs, only affecting the legs movements in the most minor way. The bones do however affect itself as well as the bone behind it, this is done to increase the fluidity of movement, as if the bone only affected its own segment; each segment would be detached from each other and would make the centipedes slither movement impossible. Having said this, the weight this bone carries of its neighbour bone is nowhere near as strong as the segments own bone has on it, allowing individual movement and certain transformations possible. The bendy bones of the spine are also segmented five times each to provide easy and smooth rotations along the x and y axis.

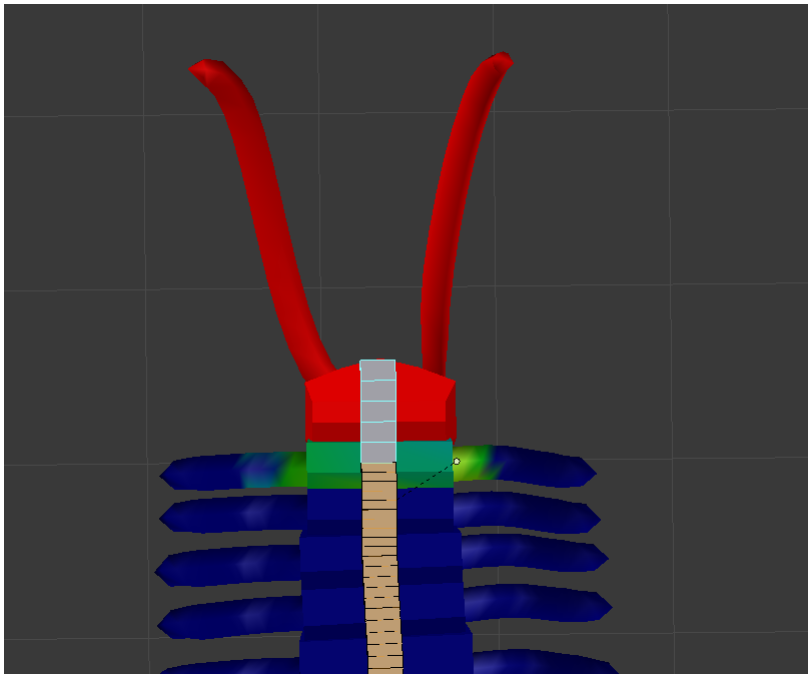


Figure 3.8: The weight of the head bone, seen in **Weight Paint** mode in Blender, Red indicates strong weight, Blue indicates no weight

The colour of the body segments is also based on figure 3.7, Using the colour picker again on the darkest section of segment, the RGB colours present where: R = 0.105, G = 0.065, B = 0.034, the colour arrangement is similar to the legs however, the body is much darker and less orange due to the decrease in the R value.

Chapter 4

Implementation

4.1 Alpha implementation

This section will be discussing the early thoughts of different potential options for movement and navigation as well as creating an environment for the centipede.

4.1.1 Environment

The environment that the centipede walked on needed not only to look real but also have functionality, the centipede must be able to crawl around as well as climb. Through the alpha implementation the idea was to create an assault course for my centipede, consisting of various objects in various sizes and rotations to determine which objects/obstacles the centipede could navigate through or climb over, the reason this was done was to test the heights of objects that would look real for the centipede to climb etc. The alpha implementation of this assault course consisted of creating a large 'stadium' which would act as the floor and add random primitive shapes to the floor in random order. The centipede was going to be some form of agent for movement and pathfinding therefore it was imperative that the floor and the obstacles have Navigation Meshes attached [4.1.2].

4.1.2 Navigation Meshes

A navigation mesh is an abstract data structure used to aid autonomous entities in various path finding techniques, based on a simplified geometrical plane these meshes are designed to ultimately find a path between objects over a given space. Navigation meshes are created automatically from your scenes geometry after baking. *Baking* is the process of creating a navigation mesh into a scene.

4.1.3 Navigation

The second section was to create some 'object' that would act as the centipede's head. This 'head' object would then be used for testing the centipede's movements as well as to navigate past obstacles around the centipede's environment. To make sure that the object created could navigate autonomously around the environment [4.1.1] a method was created that would use Unity's **Random.insideUnitSphere** method to set random Vector3 points on the stadium, that the head would navigate towards by returning a random position every time it is called.

This was attained in the project by passing the method *RandomNavSphere* [A.1.2] 3 parameters, first of which was the **origin** of the object that would be calling the method, this origin would act as the centre of the sphere, the second parameter is the **distance** which would act as the distance between the origin and the radius of the sphere. The third parameter would be the **layermask** which would tell Unity [16] which layer to test against. Once these parameters are given, the method will then pick a random Vector3 position somewhere between the origin and the radius of the navigation sphere, it would then sample the position, which returns the nearest point from that position where a navigation mesh lies. This position is then returned as a viable Vector3 position for the object to traverse to.

4.1.4 Gravity

Gravity is the force that attracts bodies to the centre of a physical body as long as it has some mass. In Unity [16] gravity works exactly as it does in real life, however Unity [16] gives the option of turning gravity off for chosen objects, meaning you can pick and choose which bodies are affected by those bodies that have a gravitational pull. Not only this, but you can also manipulate how much a body is affected by gravity using certain scripts.

At first, it would be thought that gravity would be the perfect option for obtaining vertical rises for this projects centipede, as the centipede would be able to walk up walls that were greater or equal than 90°. The hypothesis was that gravity would be added to some object, in this case lets say a tree, the centre of gravity would be in the middle of the tree. Then as soon as the centipede would reach a certain distance to the tree around 1-5 metres, the centipede would be drawn up the tree by the force of the trees gravity, whilst retaining the same speed it had previously, before it was within reach of the tree as well as, being able to walk back down the tree and carry on roaming about the land.

However, in reality when gravity is applied with some movement translation, objects rotate towards the centre of gravity. As well as this, when objects get close to the centre of gravity, objects speed up unnaturally as the force of gravity decreases with distance. The force of gravity can be found by using the equation $F = \frac{GMm}{r^2}$.

F = force of gravity

G = gravitational constant(6.67 x $10^{-}11$)

M = mass of one object

m = mass of other object

r = distance between the two objects

In this project the force of gravity can be worked out by: The gravitational constant multiplied by the mass of the body of gravity, multiplied by the mass of the centipede, divided by the distance between the two squared.

4.2 Beta implementation

This section goes into detail how the centipede was finally implemented into this project as well as explaining the benefits and drawbacks of the methods used.

4.2.1 Movement

The beta implementation involved the use of navigation meshes [4.1.2] on all terrain/obstacles in the scene. This is to make sure that the centipede does not encounter anywhere it cannot walk across. All navigation meshes in the scene are set to *static* meshes. Static meshes are meshes that are included during the navigation mesh baking process. With static meshes on all obstacles the centipede can walk freely over all terrain if the object has fitted the navigation meshes baking settings of course. The settings this project used include: An agents radius = 0.3, which defines how close the centre of the agent can get to a wall or a ledge. The agents height = 1, which defines the height at which the agent can reach to. The max slope = 60°, which defines how steep ramps are that the agent can walk up. And finally the step height which is set to 1, which defines the maximum height that the agent can walk over or climb on obstructions.

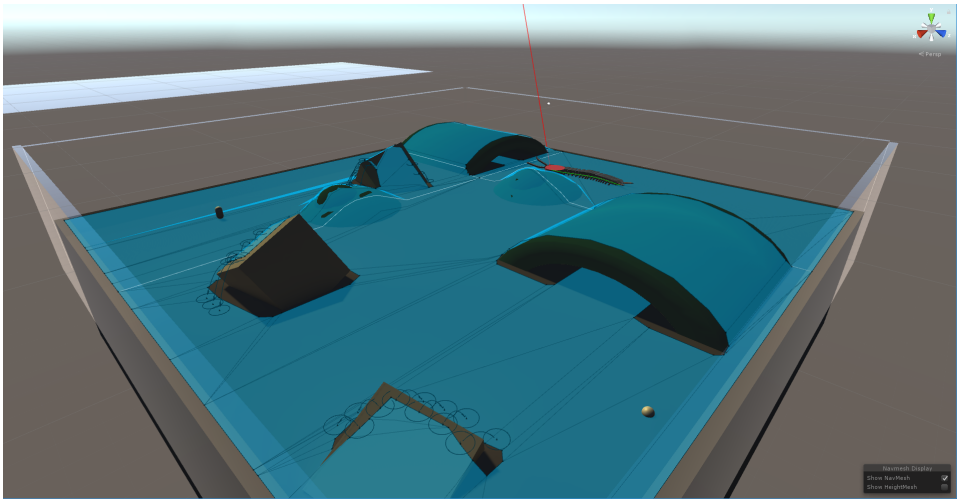


Figure 4.1: The Navigation meshes in this projects test environment

During the creation of the movement process, certain steps were needed to attain fluid like movement as well as being able to make use of the other effects such as the slither [4.2.2] effect and cobra mode [4.2.5] effect. First, the centipede needed an invisible target that would traverse through the scene using the *randomNavSphere* [4.1.3] method which would choose a random available position in the scene in which the target would follow. However, if left like this method would be called once and only one position would be found, which is not viable.

Therefore, the *randomNavSphere* [A.1.2] had to be called more than once. The target calls the method to find a new random position every five seconds, this provides a new viable position for the agent to traverse to without the target staying stationary for too long [A.1.4].

However, this also threw up a problem in that the centipede would not be able to occasionally prioritize finding shade [4.2.3] over randomly wandering. So, to fix this the target *rolls a 10 sided dice* 'figuratively speaking' every 6 seconds [A.1.9], if a random number is between 0-4 inclusive, the target will find shade instead of wandering, if the random number is between 5-9 inclusive, the agent following the target will wander disregarding finding shade/shelter [A.1.9].

The next step was to have another invisible agent follow the target around the scene, the reason for this agent is to be able to define properties such as the acceleration and speed of the centipede to keep movement smooth and not traversing the scene with rogue rotations etc. The head of the centipede then follows this agent around the scene with the ability of adding the slither [4.2.2] effect or cobra mode [4.2.5] effect.

In order for the body segments to follow the head, two sets of invisible body parts are created [4.2]: The *Virtual Body Parts* are prefabs that follow the path of the head whilst staying level to the ground [A.1.20]. The *Body Parts* prefab are almost identical to the *Virtual Body Parts* however they take into account elevation, these body parts are the reference points to the centipedes mesh [A.1.21]. During runtime, the bones of the centipede follow their corresponding *Body Part* [A.1.14]. Using this technique, the centipede can achieve all of its movement goals: the ability to slither, find shade over wandering, go into cobra mode, as well as the ability to find completely random locations in the scene to wander to.

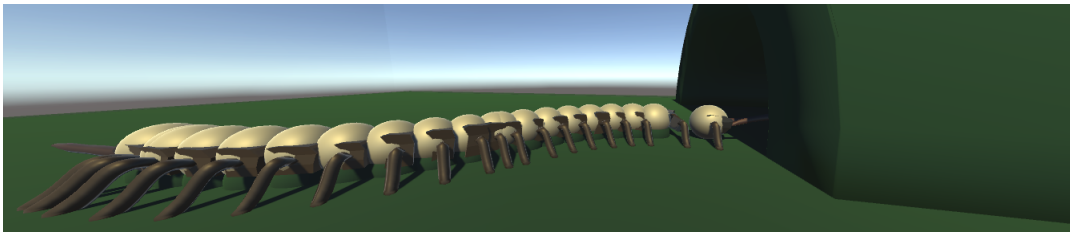


Figure 4.2: Virtual Body Parts(green) and Body Parts(white) with corresponding bone segments attached in debug mode

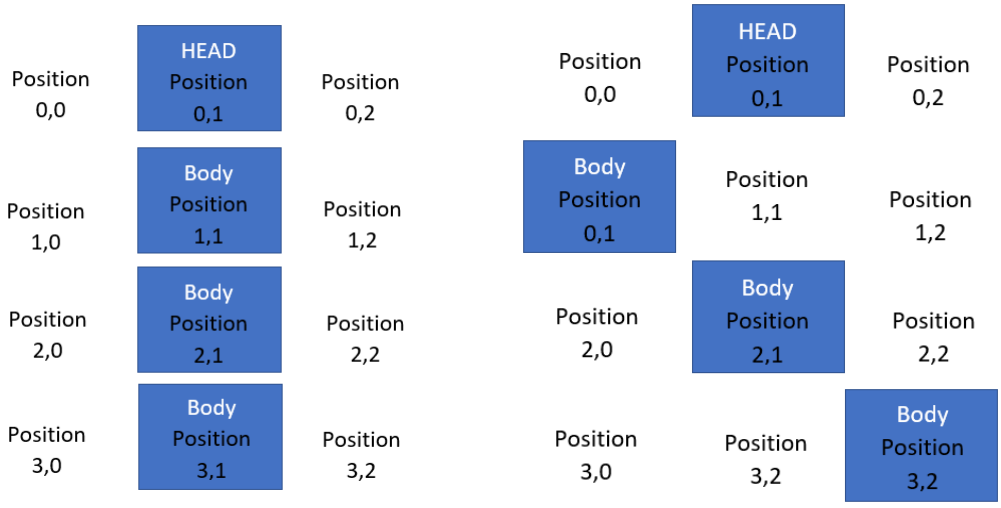
4.2.2 Slither

Most centipedes movements are similar to snakes in that they slither through their environment, there is two ways to implement this slither movement into this project, they are:

1. Create offsets for each section of the centipede

2. Place a Sine function directly on to the centipedes movements at runtime

The first option was to treat the centipedes body like a grid where each quadrant could be moved based on the quadrant in front. The method would go something like: The first node(the head) would remain in a constant position at (0,1) on the grid, the first section of the body looks at the section in front of itself and moves left, it then would set the **bool** variable *goLeft* to false and the variable *goRight* to true. From there, the second body segment would look ahead of itself, if there is no body parts there it stays where it is, the section behind then looks ahead of itself, if there is something in front it checks which variable out of *goRight* or *goLeft* is true, it then moves one grid space in that direction of the segment ahead, and so on. This technique would have to be used each frame as otherwise the motion would only happen once, see figures 4.3a and 4.3b.



(a) Centipede at frame 0

(b) Centipede at frame 1

The problem with this technique is that it would make the centipede look incredibly 'blocky', another problem is the fact that all of the segments would move at once each frame as opposed to a smooth slither motion.

The second option is to create a function that moves the head based on a Sine manipulation of the current path it is traversing. While the head follows the agent [4.2.1] the line between the centipedes head and the agent will be called the 'original route'. Using this route the centipede would move based on a **Mathf.Sin** function which computes the sine wave on the x axis along the route, resulting in a curved continuous path between the centipedes head and the agent it follows, from here the rest of the body will simply follow the head along, this can be visualized by figure 4.4.

A *Slither turn rate* was created [A.1.9] as well as the *Swim Amplitude Range X* which increases the amplitude of the Sine waves. The most realistic turn rate found for the centipede is a Slither turn rate of four with a Swim Amplitude Range of 5 [A.1.9].

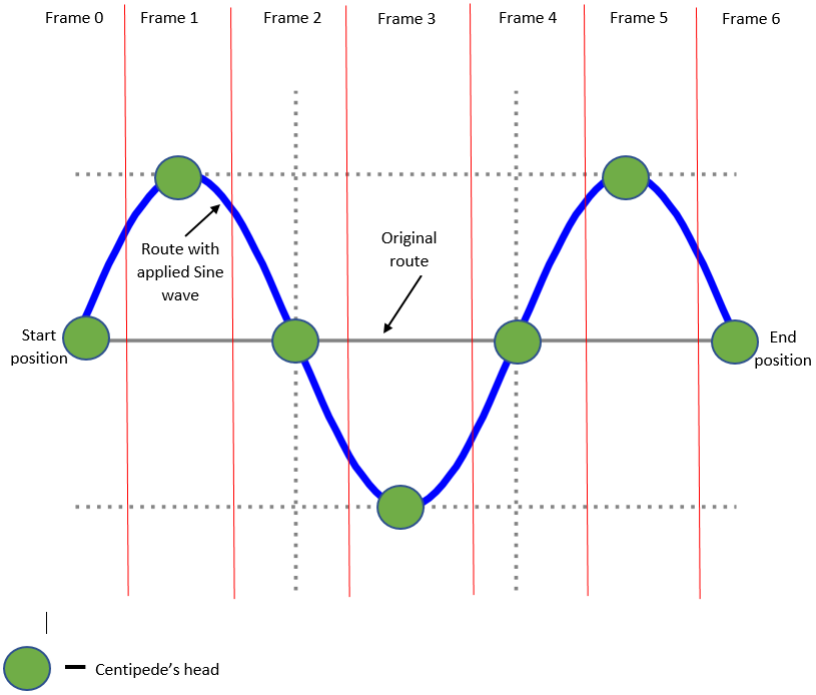


Figure 4.4: Diagram of the centipedes head following a path with a given Sine wave

4.2.3 Finding shade

The centipede finds shade in this environment by shooting rays from the centipedes head directly towards the light source of the environment, if this ray reaches the light source without being blocked then the centipede knows it is not in shade, however if the ray is intercepted or blocked by an object then the current area of the centipede is a shaded area, if this is the case it will store the current position of the centipede in a singly linked list [A.1.6]. If the user clicks the button *Find shade* on the scripts **Inspector** tab. The centipede will run to a randomly chosen position in the list. The rays will also stop firing as the centipede has already found shade, the list will therefore not have any more Vector3 positions added to it. If the button is clicked again, the list will be erased and the centipede will start firing rays again.

The finding shade methods use the function of *Time.deltaTime*. A timer is set to 0 += *Time.deltaTime*, if that timer is >= to 4.83 seconds which is 0.17 milliseconds less than the movement timer, it will store the position. The reason the time has to be so specific is due to the way *Time.deltaTime* works. *Time.deltaTime* works by returning the time in seconds it takes to complete the last frame. If *Time.deltaTime* is applied with an operator such as '*' or '/' you change moving an object by meters per second to meters per frame. If this method is not used, the list will store hundreds of the same shaded position. This is a lot more efficient as slower computers that have a lower frame rate will still view the movement in the same speed as someone with a more powerful computer.

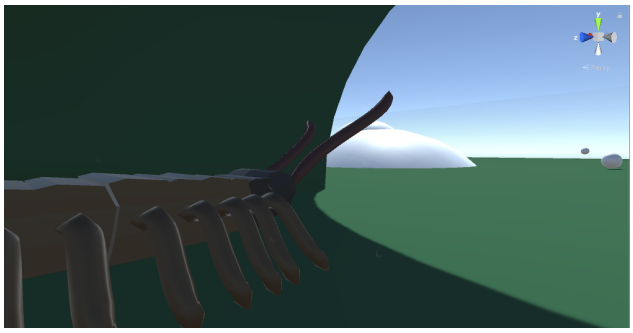


Figure 4.5: Centipede hiding in shade in a half log

The reason a singly linked list has been chosen to store these Vector3 positions over other data structures such as arrays, trees, and dictionaries is due to the time complexity of a Linked list. When choosing shade, the program will be choosing a random item anyway therefore having an ordered data structure would be pointless.

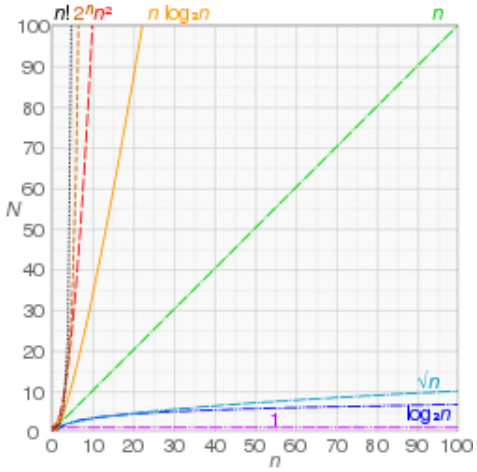


Figure 4.6: Number of operations N versus input size n for each function [18]

The time complexity when storing objects into the end of a singly linked List is O(1) meaning it takes the minimum amount of time to insert an item into the list. This is due to the fact that all of the data objects that will be inserted into the list are inserted at the end of the list therefore the program only needs to look at the last node and add it on to the end.

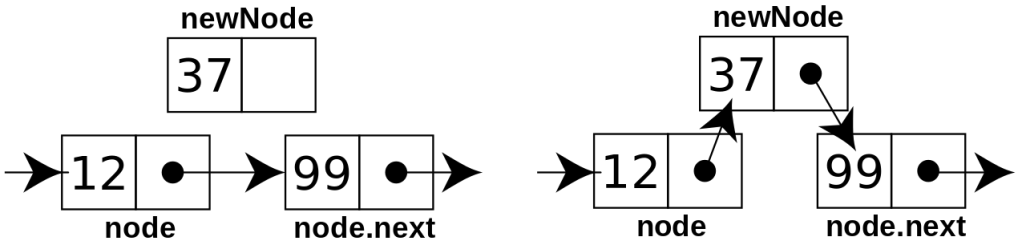


Figure 4.7: Insertion of a node into a singly linked list [19]

Another reason the finding shade program uses linked lists is due to the time complexity of deletion, and in this projects case, **all** positions will be deleted when the user clicks the *Find shade* button. The most efficient deletion time complexity is again O(1), this would normally not be achievable in a singly linked list as the deletion time is O(n) in this structure, this is because in order to delete a node in a singly linked list you need to know the node before and after the node you are deleting in order to link them together. However, considering this fault, singly linked lists were definitely the best option for storing the centipedes Vector3 positions.

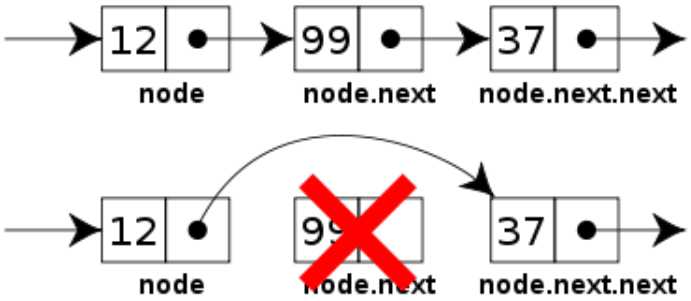


Figure 4.8: Deletion of a node into a singly linked list [20]

4.2.4 Attacking

Centipedes are known as incredibly aggressive creatures. This simulation needed to accomplish at least some level of aggression. Firstly, the centipede needed an enemy. Centipedes in the wild will attack nearly anything that threatens them or that is common prey. Their diet usually consists of invertebrates such as insects, spiders, millipedes, scorpions as well as lizards, frogs, snakes and bats [17]. The prey used in this project was a simple Unity [16] capsule. A public **bool** called *'Attack'* was created to give the user the choice to let the centipede attack or not. If the user selects *'Attack'* it triggers a method that computes the distance between the prey and the centipedes position. It then checks to see whether the prey is within twenty metres of the centipede. If the prey is within twenty metres of the centipede, the centipedes speed doubles and the slither turn rate [4.2.2] doubles also, this can be seen in figure 5.9.

The agents stopping distance is also decreased to one, in order to ensure the centipede is within eating distance of the prey [A.1.13]. The capsule however, runs in the opposite way ensuring a chase will occur. When the prey runs further than twenty metres away from the centipede, the centipede will give up chase and continue to wander around its environment normally, see figure 5.11.

4.2.5 Cobra Mode

As discussed earlier [2.1.1], the Scolopendra genus of centipedes have the ability to raise the front third of their body, in order to provide better vision or even climb adjacent objects that are of a greater height to the ground they are currently on. This ability is crucial to the centipedes survival as it improves: sight, dexterity, aids in escaping from predators as well as helping them ambush prey.

In this project we refer to this ability as *Cobra Mode* this is due to the similar movement the King Cobra performs. The King Cobra as well as some other species of snake can raise up to 35% of their body during this stance. This percentage is in fact also equal to their maximum strike distance [4.2.4].

The way *Cobra Mode* was created in this project had to be able to mimic this ability whilst also retaining the original centipedes mesh fully. Using a public *AnimationCurve* the user can choose how much of the centipede will be raised. This could be useful for testing out other species of centipedes as they often differ. However, in this projects case, the program mimicked the Scolopendra genus.

The *Animation Curve* is a curve with a number of arbitrary animation keyframes. Figure 4.9 references the curve this centipede used, the start of the curve is untouched with only the front third raised. The curve has two axis', the horizontal axis being the time, and the vertical axis being the value of the keyframe at any given point.

This curve can then be used when updating the body parts of the centipede by first checking if *Cobra Mode* is activated, if it is activated the elevation [4.2.1] of each body part is worked out. This is done by looping through the body segments, then dividing the body segments one by one starting at the back of the centipede by the total number of body segments the centipede has, this will result in a number between zero and one with the back end body segments of the centipede being close to the zero and the very front segment of the centipede (the head) being 1. This result is then used as the time variable for a method called *AnimationCurve.Evaluate*, this method evaluates the time passed through it and evaluates where each body segment will lie on the curve. The evaluation is then applied with a *Vector3.Up* translation resulting in the vertical raising of the centipede [A.1.14].

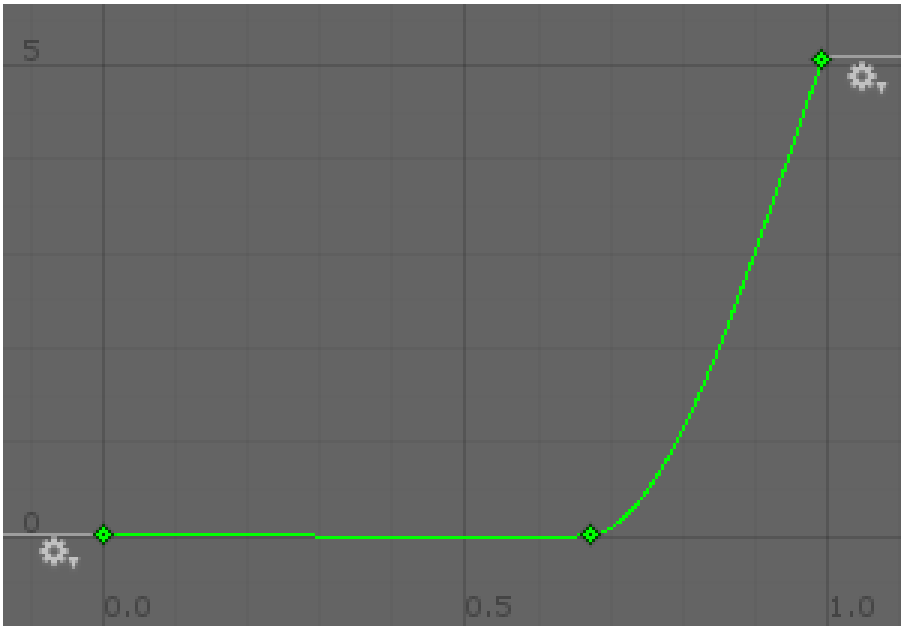


Figure 4.9: The curve of the centipede between the time frame 0 and 1 and with a value of 5

The finished product of the beta implementation consists of a dynamic 3D centipede, that can hunt, find shade, wander its environment and can achieve the phenomenon of what this project coined 'Cobra Mode', the finished centipede can be seen wandering its environment in figure 4.10.

Figure 4.10: The finished centipede in its environment

Chapter 5

Evaluation

This chapter involves two rigorous tests to test the full functionality of each section of the implementation and design process of this project. The first test was a 'Play test' which is the process of testing for bugs and design flaws. Objects tested included, the centipede, the centipedes target, various obstacles, as well as the centipedes abilities to climb, hunt and find shade.

The second test was a visual test, to determine that the centipedes behaviour and movement patterns acted and seemed realistic. This test gave better feedback about small enhancements to this project, as opposed to the play test which looked at bugs and glitches in the project.

5.1 Play Test

5.1.1 Climbing vertically using gravity

Involved subjects: Sphere and a large cube with 90° angle. Force of gravity on cube set at 10.

Expected results: Sphere should be able to walk up the cube as well as walk back down with no change to speed

Actual results: Sphere correctly walked vertically on the cube, however sphere was drawn to the centre of the cube(centre of gravity) and will not traverse back down, remained stationary in the centre of the cube.

5.1.2 Agent following target

Involved subjects: One agent, one target and a virtual terrain to walk on.

Expected results: When 'Play' the agent should follow the target sphere and remain stationary until the target sphere is moved.

Actual results: Agent correctly followed the target sphere to its location and remained stationary until the sphere was moved.

5.1.3 Head following agent

Involved subjects: Centipede's head, one agent, one target and a virtual terrain to walk on.

Expected results: The centipede's head should follow the exact same path as the agent and traverse normally to the target sphere.

Actual results: Centipede's head correctly followed the agents path until the agent reached its destination.

5.1.4 Body following head

Involved subjects: Centipede's head and body, one agent, one target and a virtual terrain to walk on.

Expected results: Centipede should reach the target sphere with the mesh remaining in tact.

Actual results: Centipede followed the path of the agent to the target sphere in full and remained stationary when it got there, whilst the mesh remained in tact.

5.1.5 Slither

Involved subjects: Centipede's head and body, one agent, one target, a virtual terrain to walk on and a slither turn rate of 5 with both swim amplitude x and y being set to 5.

Expected results: Centipede should follow the path of a Sine wave with the end point being the agents current position. The wave should have an amplitude of 5.

Actual results: The centipede successfully followed the path of a Sine wave with the end point being the agents current location, until it reached the target sphere and remained stationary.

5.1.6 Climbing

Involved subjects: One full centipede, one agent, one target, a virtual terrain to walk on, and a sphere with various maximum angles.

Expected results: Centipede should walk over the sphere while retaining its mesh and have no legs/body parts going through the mesh of the floor.

Actual results: Centipede successfully traverses over obstacles as long as the angle of the movement is equal to or below 60°.

5.1.7 Obstacle Avoidance

Involved subjects: One full centipede, one agent, one target, a virtual terrain to walk on, and an obstacle with an attached nav mesh in the way of the agent and the target.

Expected results: Centipede should walk around the obstacle to the target sphere and not walk through the obstacle and have no legs go through the obstacle also.

Actual results: Centipede successfully walked around the obstacle to the target sphere until it reached it and then it remained stationary.

5.1.8 Finding Shade

Involved subjects: One full centipede, one light source, a timer with a limit of 5 seconds and a ray between 0 and infinity on all layermasks.

Expected results: Centipede should after 5 seconds of wandering, walk to a shaded Vector3 position in a linked list.

Actual results: Centipede successfully wandered over to a shaded area it had previously walked through after 5 seconds.

5.1.9 Cobra Mode

Involved subjects: One full centipede, one agent, one target, a cobra height of 5 and a curve moving the final third of the centipedes body.

Expected results: Centipedes front third of its body should raise, must be able to work even whilst walking.

Actual results: Centipede successfully raises its front third of its body even whilst walking.

5.2 Functionality Testing

5.2.1 Wander

Conducted with a one minute timer to review positions every ten seconds.

Centipede should wander freely with no preferred position, not looking for shade or attacking any target.

Bugs Found: None

Figure 5.1: Centipede wandering after ten seconds.

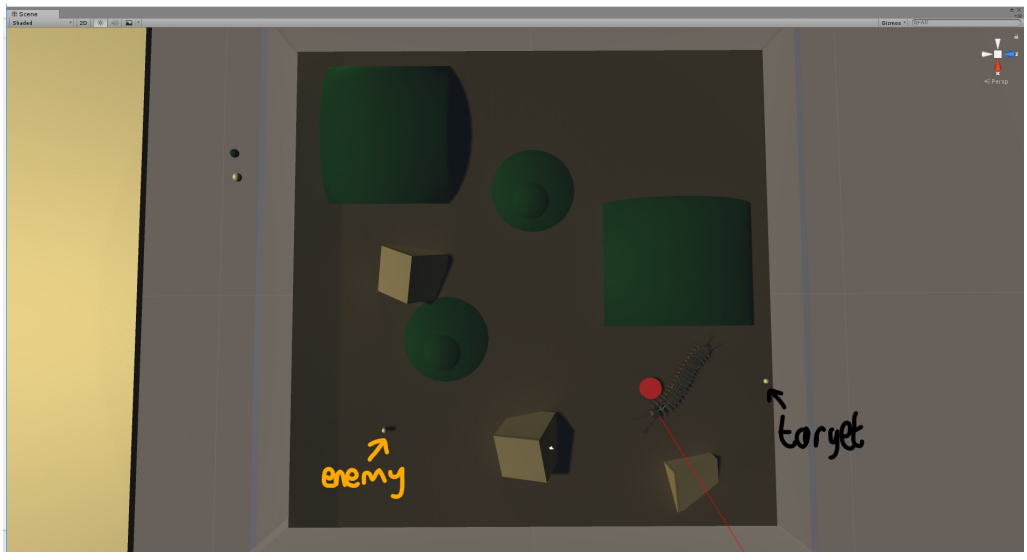


Figure 5.2: Centipede wandering after twenty seconds.

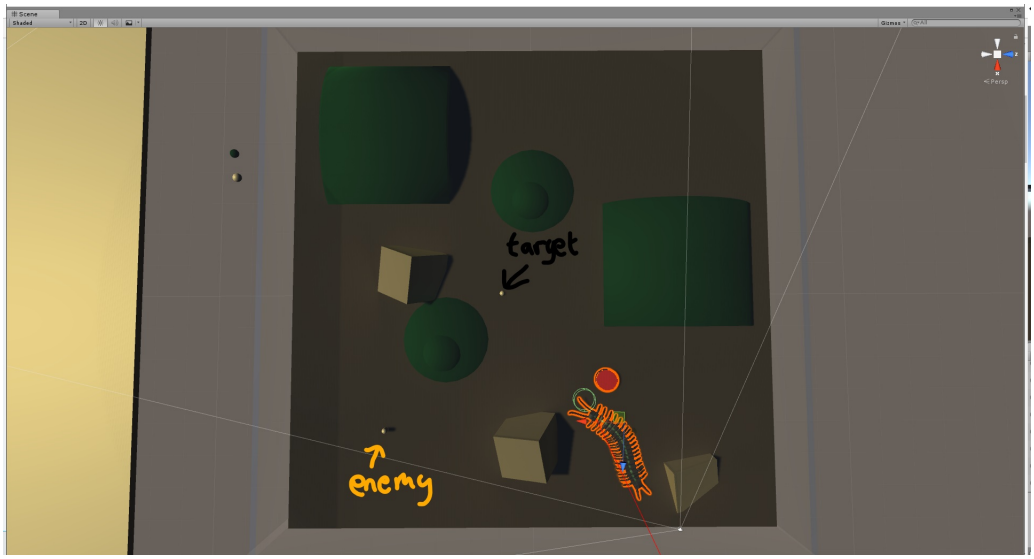


Figure 5.3: Centipede wandering after thirty seconds.

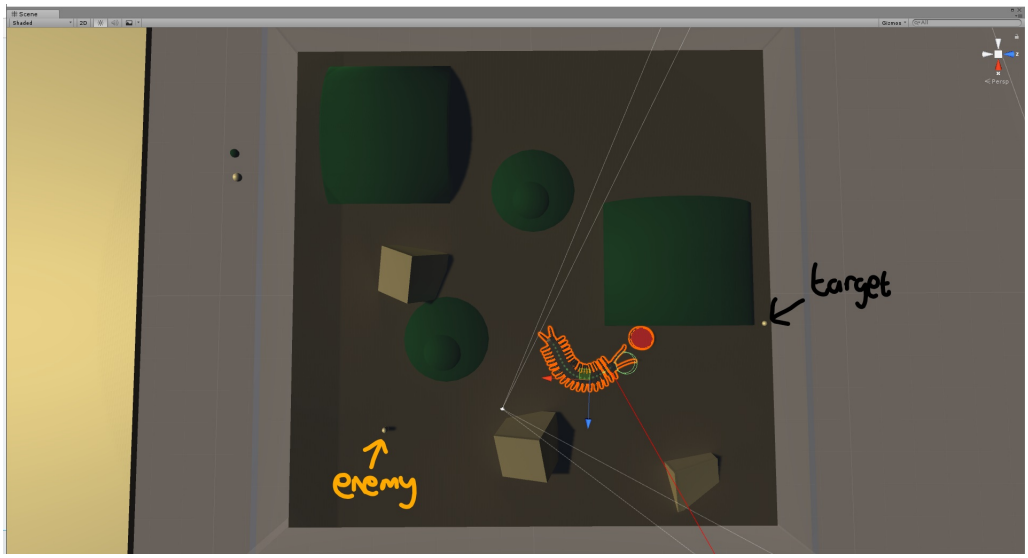


Figure 5.4: Centipede wandering after fourty seconds.

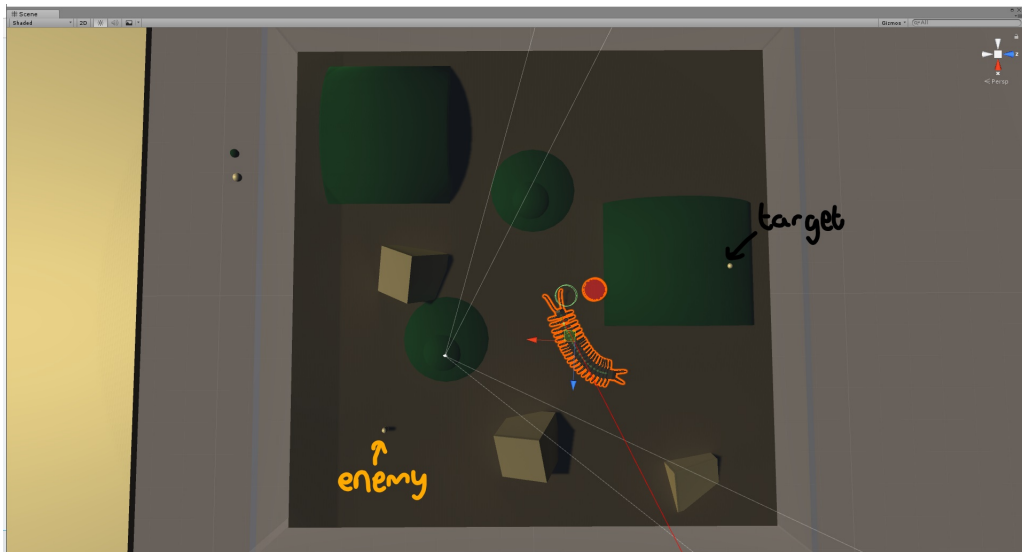


Figure 5.5: Centipede wandering after fifty seconds.

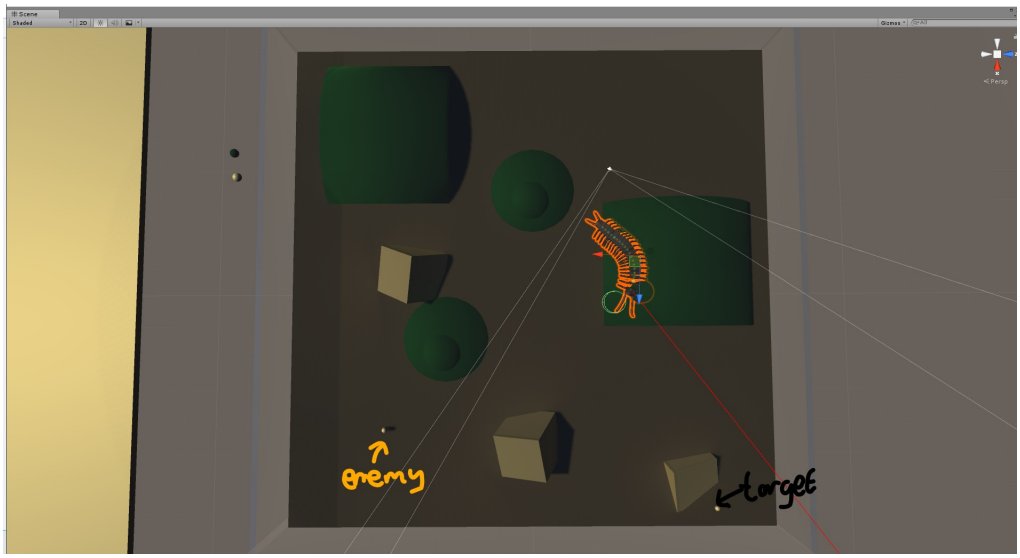


Figure 5.6: Centipede wandering after sixty seconds.

5.2.2 Hunting

Conducted with a no set timer, images were taken in steps of the centipedes attack.

Centipede should wander freely with no preferred position, until it is within twenty metres of the *enemy*. From here, the centipede should increase speed as well as decreasing the agents *stoppingDistance* from ten to one. This will ensure the centipede is within range to attack the target. The enemy should run in the opposite direction of the centipede. When the enemy runs twenty or more metres away from the centipede, the centipede should continue randomly wandering.

Bugs Found: None

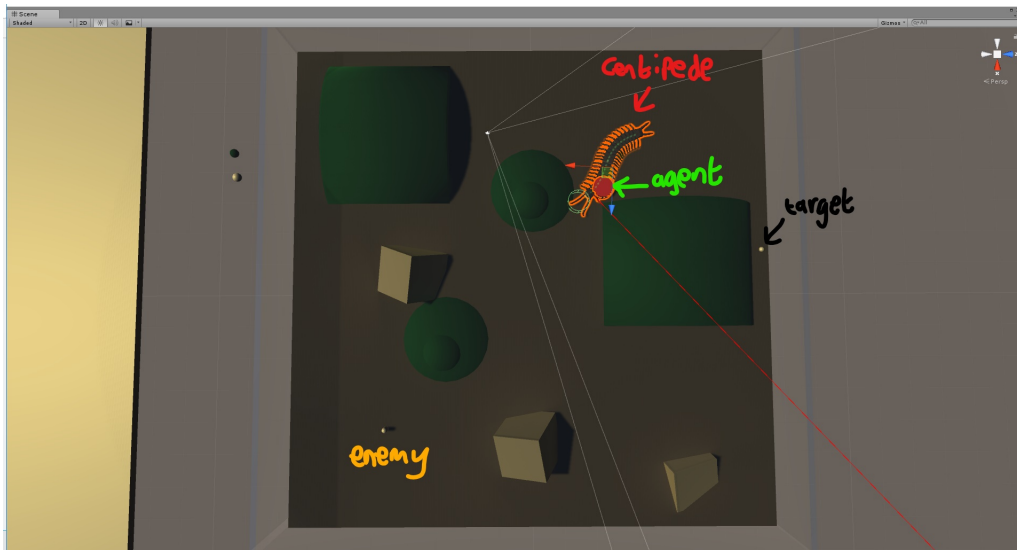


Figure 5.7: Centipede has not seen the enemy.

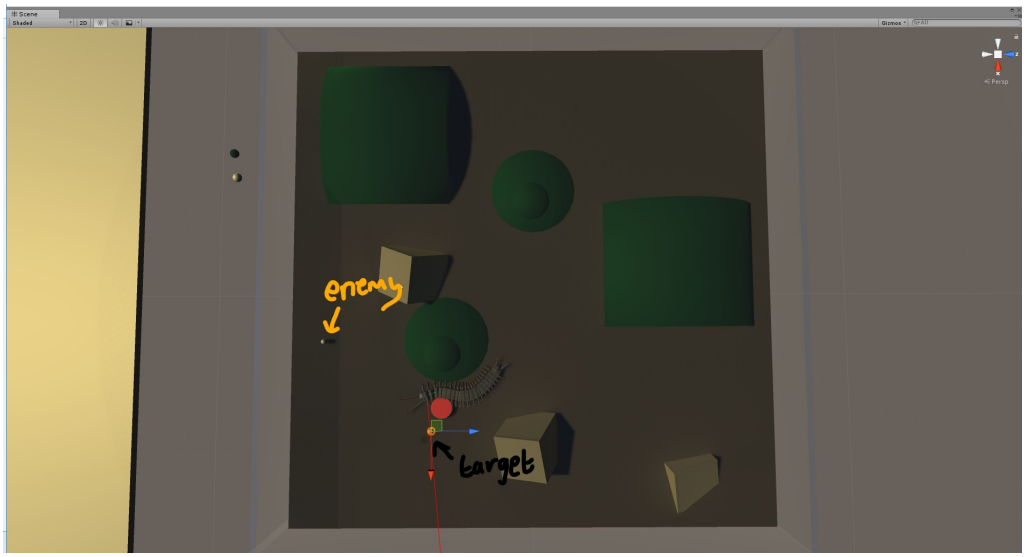


Figure 5.8: Centipede has spotted its enemy.

Figure 5.9: Centipede starts chasing the enemy, speed increases and stopping distance decreases.

Figure 5.10: Centipede is about to go out of range of the enemy.

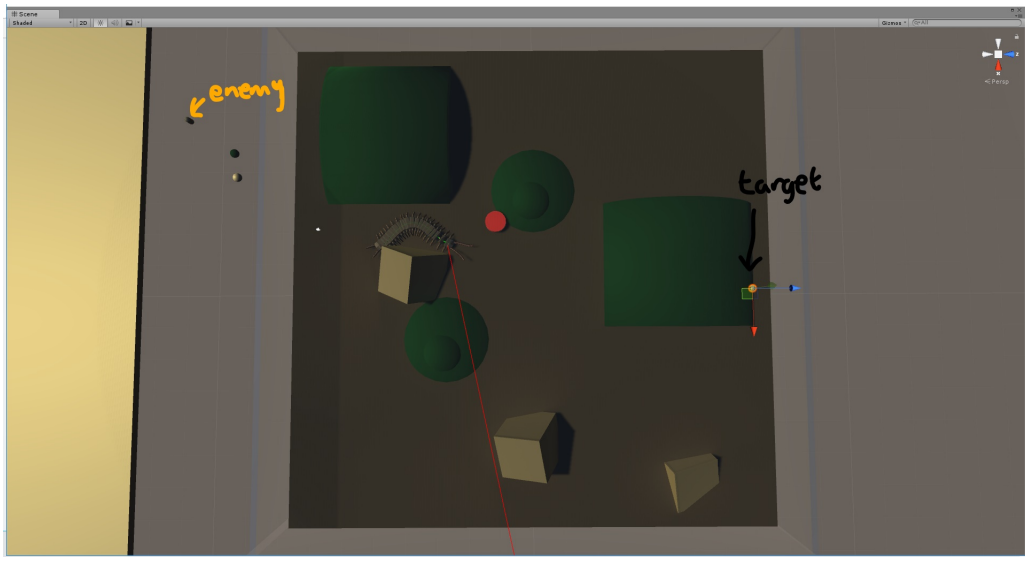


Figure 5.11: Centipede is out of range from the enemy, continues to wander.

5.3 Finding Shade

Centipede should wander freely, if the centipede is in a shaded area, position should be added to the list. A random number should be picked between zero and nine if the list is longer than one, if the number is less than or equal to four, centipede should stop wander to one of the shaded areas in the list. Every five seconds a new number is picked. The centipede should remain in the shaded area until the random number is greater than 4, in which case it should continue wandering as normal.

Bugs Found: Instead of fleeing to a random shaded position, the centipede stays where it is.

Bug Fixes: After giving it some thought, this bug actually makes more sense, as if the centipede is already in shade it wouldn't need to find another shaded area just because its shaded. This 'bug' remains in the code purposefully.

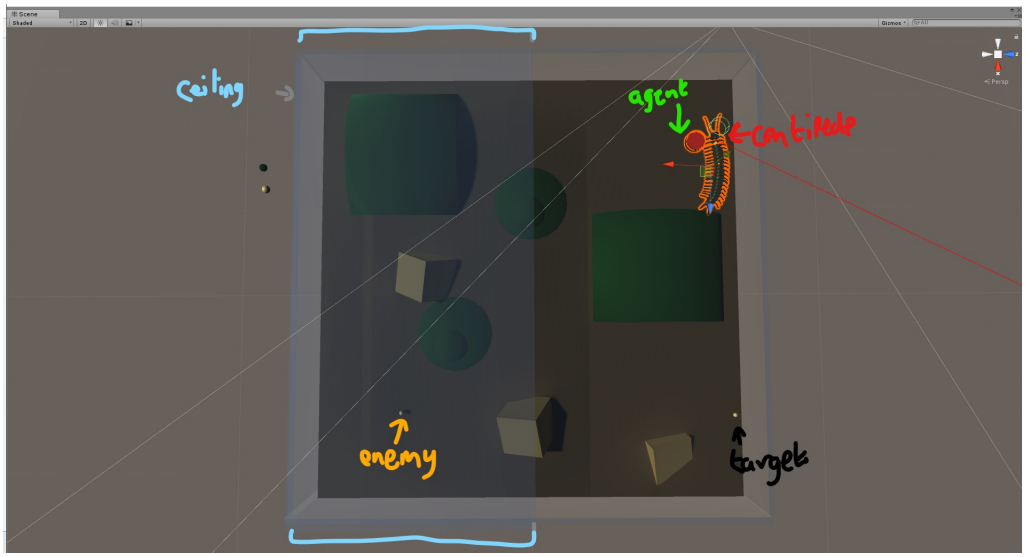


Figure 5.12: Centipede wanders freely *Glass ceiling is purely for testing, think of it as an opaque ceiling.*

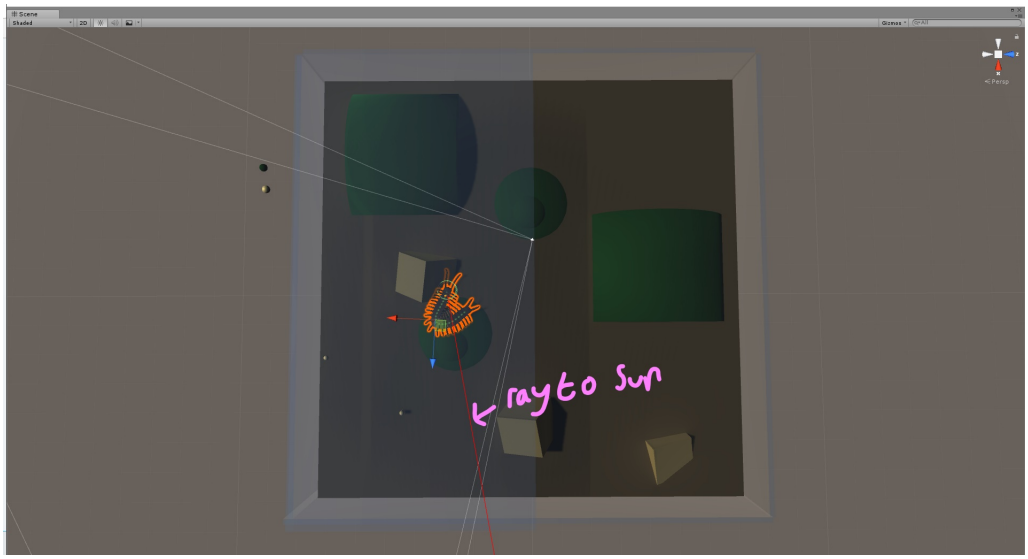


Figure 5.13: Centipede is in shade so stores position.

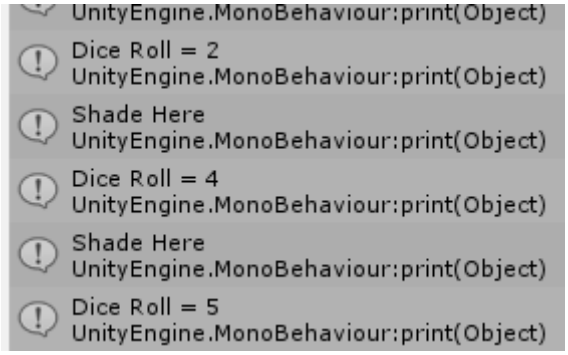


Figure 5.14: Centipede rolls higher than a 4 so continues to wander.

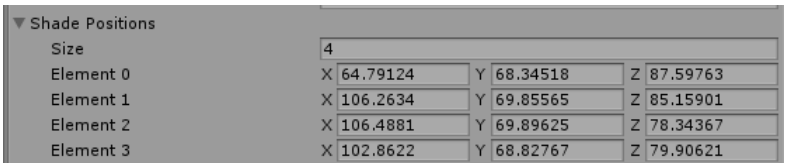


Figure 5.15: Centipedes list as it continues to roam.

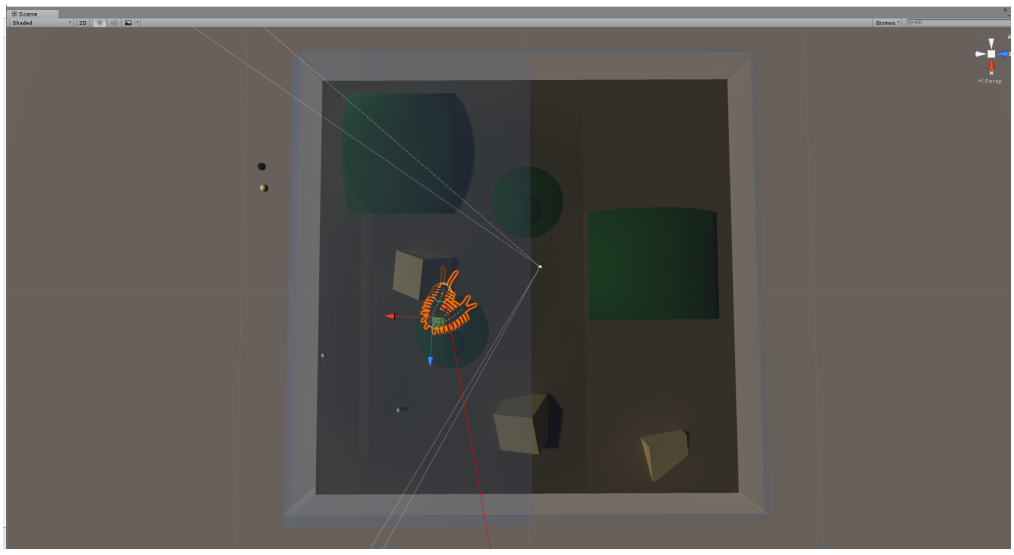


Figure 5.16: Centipede is in shade and rolls a 0, remains here until it rolls higher than a 4.

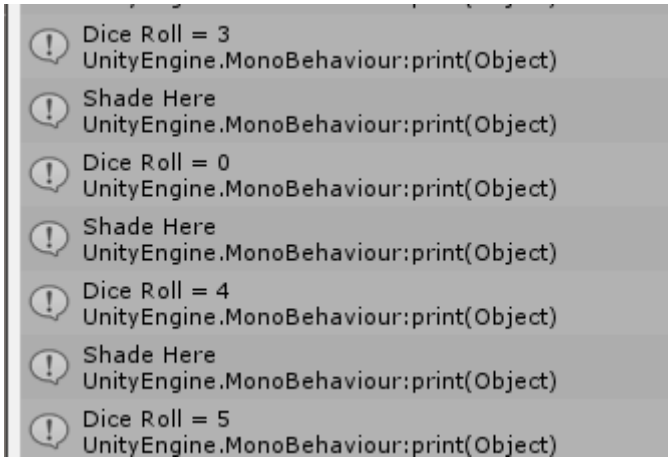


Figure 5.17: Centipede rolls a 5 so begins to wander.

Figure 5.18: Centipede continues to wander its terrain.

Chapter 6

Conclusion

6.1 Analysis

This section discusses the length of completion of each of the aims and objectives that were set at the start of this project.

6.1.1 Investigating the most suitable techniques for a centipedes autonomous movements

The techniques that were considered for this project included: Gravity, Unity's [16] NavMesh agents and Transform Translations.

Transform Translations are forms of primitive movement, they work by moving a transform object in a set direction and distance. This could have worked for very simple movement but, the piece of code responsible for the translation would have to be changed every time the transforms target moves, and since the distance is set by a distinct number(e.g. 10, meaning the transform will translate 10 metres in the direction chosen), it would make movement incredibly hard and really, unusable for any transform with more than one movement.

Gravity was looked at, as a way to create a route for the centipede to walk directly up obstacles as well as even travelling at obtuse angles upwards. This technique was also unusable as the centre of gravity was so strong that even at the minimum force, the centipede would be drawn to the centre of gravity.

Finally, NavMesh agents as well as static Navigation meshes were used as way to easily set an agents destination as well as defining which obstacles and objects the centipede could walk on/over.

6.1.2 Creating a 3D model of the centipede

A 3D model was created in Blender [1] consisting of a centipede cut into sections defining each section of the centipedes body and head, as well as implementing bendy bones [3] for each of the centipedes segmented body parts, legs were also created with two bones in each that would later on be able to detect the presence of other legs and make sure they were two legs were never touching each other. The functionality of the model worked sufficiently, however the model had a very blocky design feel to it and with more time could have looked a lot more realistic.

6.1.3 Designing and creating a suitable test environment

An environment consisting of virtual objects that resembled a rainforest was created, with numerous shaded areas as well as obstacles for the centipede to walk over. The design of the environment was based on figure 3.1 and consisted of green and brown colours to resemble trees and grassland. This test environment successfully resembled a centipedes natural habitat as well as providing difficult obstacles and tests for the centipede to traverse through.

6.1.4 Creating a behavioural system for hunting prey

A system was created to increase the centipedes speed and swim swim amplitude when in close proximity to prey, this prey was resembled simply by a capsule in the centipedes environment, when the centipede gets within a certain distance of the prey, it will chase it and the prey will flee. This system worked well as it showed just how much centipedes increase their speed when in the presence of prey and showed how aggressive they are.

6.1.5 Creating a behavioural system for finding shade

The finding shade behaviour was a very important one and a somewhat complex one, the centipede had to be able to wander naturally until it had found enough shaded areas(>=2) and from here it was important that the centipede didn't just run to shade and stay there for the duration, a system was created to implement a *dice roll* in which the centipede after finding a sufficient number of shaded areas would have a 50% chance of being chosen to find shade instead of wandering. This system worked well as the lists in which the shaded areas positions were stored was efficient therefore not slowing down the program. Furthermore, the *dice roll* function worked smoothly with the other centipede behaviours.

6.1.6 Creating a method to represent the phenomenon of the centipedes front third raising to a 90° angle

Cobra mode was created in order to resemble this phenomena, the front third of the model would rise to a set point by the user on an animation curve. The reason behind this was to give the user the flexibility to increase/decrease the maximum height of the centipede when in cobra mode, based on the users environment. This was important as otherwise the centipedes head could potentially go through low objects.

6.2 Limitations

This section considers the limitations of this project as well as any parts of the project that simply don't work or had to be removed due to clashes with other methods.

The main limitations of this project were based around design as opposed to functionality. The design of the centipede could have been more realistic to go along with the realistic movements and behaviours of the centipede.

Moreover, there could have potentially been added emphasis on gamification, improving the users experience when witnessing the centipede. As well as giving the centipede more purpose. In addition to this, an attack behaviour could have been built to be used at the end of hunting the prey.

6.3 Future work

The project as it stands offers a logical way to create clever artificial intelligence. Without using animations it makes it a lot easier to tweak certain behaviours and/or movements. However, work could be done to make a 'Feasting' function where not only does the centipede catch its prey but also eats it, this could be useful if the centipede was in a game that used a 'health' or 'hitpoints' system, Feasting could replenish health perhaps. Another possibility would be to create a 'Laying eggs' function, where to centipede lays eggs which will eventually spawn and become small versions of the parent centipede, these centipedes could then grow, the methods created in this project could easily be replicated and reused for each of these child centipedes. The model of the centipede could also have much more detail as well with the legs having some function to be able to know where the front and previous ones are and avoid them. One more big improvement could be to have the lighting dynamically move around the environment every so often, this would mean the centipedes shaded areas would change drastically, it would also provide more realism as in the real world, the centipedes shelters would depend on the time of day. Another improvement for future work could be done on the environment, potentially paid Unity Assets could make the scene look more realistic. Finally, more theoretical work could be done to research more about how the centipede attacks and specific move sets the centipede commonly uses when capturing prey.

References

[1] Blender, *Blender*, 2.79a, 2002. [Online]. Available: https://www.blender.org/ (visited on 17th Apr. 2018) (pp. 15–17, 47).

[2] ——, (2013). About, [Online]. Available: https://www.blender.org/about/ (visited on 17th Apr. 2018) (pp. 11, 13).

[3] Blender Manual. (2017). Bendy bones, [Online]. Available: https://docs.blender.org/manual/en/dev/rigging/armatures/bones/properties/bendy_bones.html (visited on 17th Apr. 2018) (pp. 11, 47).

[4] M. Blender Reference Manual. (2013). Weight paint mode, [Online]. Available: https://blender-manual-i18n.readthedocs.io/ja/latest/modeling/meshes/vertex_groups/weight_paint.html (visited on 17th Apr. 2018) (p. 17).

[5] Catherine Meshew and University of Michigan. (2001). Adw: Scolopendra gigantea: Information, [Online]. Available: https://animaldiversity.org/accounts/Scolopendra_gigantea/ (visited on 15th Apr. 2018) (p. 7).

[6] Cenydd, Llyr-Ap and William Teahan, 'The dynamic animation of ambulatory arthropods', Dynamic Animation, Computer Science, University of Wales Bangor, Bangor, United Kingdom, 2007 (p. 9).

[7] JavaScript, *Javascript*, 1.8.5, 1995. [Online]. Available: https://www.javascript.com/ (visited on 19th Apr. 2018) (p. 11).

[8] Kazuyuki Ito and Yasunori Ishigaki. (2015). Semiautonomous centipede-like robot for rubble - development of an actual scale robot for rescue operation, [Online]. Available: http://www.inderscience.com/offer.php?id=70709 (visited on 18th Apr. 2018) (p. 8).

[9] Kotaro Yasui, Kazuhiko Sakai, Takeshi Kano, Dai Owaki and Akio Ishiguro. (2017). Decentralized control scheme for myriapod robot inspired by adaptive and resilient centipede locomotion, [Online]. Available: http://journals.plos.

org/plosone/article?id=10.1371/journal.pone.0171421 (visited on 17th Apr. 2018) (p. 8).

[10] Matt Reinbold and Critter Getter Pest Control and Wildlife Services. (2010). Centipedes, [Online]. Available: http://azcrittergetter.com/centipede.html (visited on 17th Apr. 2018) (p. 17).

[11] Michael Booth and Valve. (2009). The ai systems of left 4 dead, [Online]. Available: http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf (visited on 17th Apr. 2018) (p. 10).

[12] Nesrine Akkari, Pavel Stoev and John Lewis, 'The scolopendromorph centipedes (chilopoda, scolopendromorpha) of tunisia: Taxonomy, distribution and habitats', Zoology, Research Unit of Biodiversity and Biology of Populations, Institut Supérieur des Sciences Biologiques Appliquées de Tunis, Tunis, Tunisia, National Museum of Natural History, Sofi a, Bulgaria and Somerset County Museum, Taunton Castle, Taunton, Somerset, UK and Entomology Department, Th e Natural History Museum, London, UK, Tunis, Tunisia, 2008 (pp. 4, 7).

[13] Nicholas M.Patrikalakis, Takashi Maekawa, Wonjoon Cho and Michigan Institute of Technology. (2009). Definition of bėzier curve and its properties, [Online]. Available: http://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/node12.html (visited on 17th Apr. 2018) (p. 15).

[14] PhotoEverywhere. (2011). Photo: Hawaiian rainforest background, [Online]. Available: http://photoeverywhere.co.uk/west/usa/hawaii/slides/hawaii-rainforest-national-parkDSC_3739.htm (visited on 19th Apr. 2018) (p. 13).

[15] Samuel P.G. Guizze, Irene Knysak, Katia C. Barbaro, Manoela Karam-Gemael, Amazonas Chagas-Jr and SciELO. (2016). Predatory behavior of three centipede species of the order scolopendromorpha (arthropoda: Myriapoda: Chilopoda), [Online]. Available: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1984-46702016000600300&lng=en&nrm=iso&tlng=en (visited on 17th Apr. 2018) (pp. 4–6).

[16] Unity, *Unity*, 2017.4.0f1, 2004. [Online]. Available: https://unity3d.com/ (visited on 19th Apr. 2018) (pp. 2, 10, 14, 16, 20, 21, 29, 46).

[17] Wikipedia Contributors. (2006). Scolopendra gigantea — Wikipedia , the free encyclopedia, [Online]. Available: https://en.wikipedia.org/wiki/Scolopendra_gigantea (visited on 20th Apr. 2018) (p. 29).

[18] ——, (2018). Time complexity — Wikipedia , the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/%20Time_complexity (visited on 17th Apr. 2018) (p. 28).

[19] Wikipedia Contributors and Derrick Coetzee. (2014). File:cpt-linkedlists-addingnode.svg — Wikipedia , the free encyclopedia, [Online]. Available: https://commons.wikimedia.org/wiki/File:CPT-LinkedLists-addingnode.svg (visited on 17th Apr. 2018) (p. 28).

[20] ——, (2014). File:cpt-linkedlists-deletingnode.svg — Wikipedia , the free encyclopedia, [Online]. Available: https://commons.wikimedia.org/wiki/File:CPT-LinkedLists-deletingnode.svg (visited on 17th Apr. 2018) (p. 29).

Appendix A

Appendix

A.1 Code stubs

This section will display all of code for every method created in this centipede project. All functions in this section are from the script Centipede.cs which is the main script controlling most of the behaviours and movements for the centipede.

A.1.1 Global variables

```
1    public bool showDebug = false;
  bool prevShowDebug = false;
3
    public bool attack = false;
5  public bool updateBonePositions = false;
    public bool findShade = false;
7    public Transform enemy;
  public List<Transform> VirtualBodyParts = new List<Transform>();
9
  List<Transform> TailBones = new List<Transform>();
11
  List<Transform> BodyParts = new List<Transform>();
13
  public Transform firstTailBone;
15 public Transform meshRoot;

17 public float acceleration = 1.0f;
  public float damper = 10.0f;
19 public float maxSpeed = 5;
```

```
21  public int numSegments = 10;

23  public GameObject virtualBodyPrefab;
    public GameObject bodyPrefab;
25
    float speed = 0;
27
    LinkedList<Vector3> path = new LinkedList<Vector3>();
29
    float[] distancesToHead;
31
    public float pathResolution = 1.5f;
33
    float followDistance = 0;
35
    public LayerMask terrainMask;
37

39  Vector3 attackVector;

41
    private UnityEngine.AI.NavMeshAgent agent;
43
    public float targetHeadHeight = 0.1f;
45

47  public Transform target;

49  Vector3 actualTargetPos;

51  public Vector2 swimFrequencyRangeX = new Vector2(2,2);
    public Vector2 swimAmplitudeRangeX = new Vector2(2,2);
53
    float swimFrequencyX;
55  float swimAmplitudeX;

57  float slitherTimer = 0.0f;
    public float slitherTurnRate = 10.0f;
59
```

```
    float phaseOffset = 0.05f;
    float headHeight = 0.0f;

    Vector3 allNoise;

      public float timer;
      public float wanderTimer = 5;
      public Vector3 newPos;
      public GameObject Light;
      public RaycastHit hit;
      public float shadeTimer;
      public List<Vector3> shadePositions;

      float distanceToLastPathNode = 0.0f;
    float percentage = 0.0f;
      private float randomNum = 6;
    public bool cobraMode = false;
    public AnimationCurve cobraCurve;
    public float cobraHeight = 3;
```

A.1.2 RandomNavSphere

```
public static Vector3 RandomNavSphere(Vector3 origin, float distance
    , int layermask)
  {
     Vector3 randomDirection = UnityEngine.Random.insideUnitSphere *
    distance;
     randomDirection += origin;
     NavMeshHit navHit;
     NavMesh.SamplePosition(randomDirection, out navHit, distance,
    layermask);
     return navHit.position;
  }
```

A.1.3 Start

```
//------------ Runs once at the start ------------

  void Start () {

    followDistance = pathResolution;
    agent = transform.Find("Agent").GetComponent<UnityEngine.AI.
   NavMeshAgent>();

    FindAllTailBones();
    CreateVirtualBodyParts();
    CreateBodyParts();

        //Setting the spheres position to a random one, that has the
    correct y coord
        newPos = RandomNavSphere(target.position, 200, 1);
        target.position = newPos;


        shadeTimer += timer;

        UpdateDebug();

    StartCoroutine(changeTargetPosition());

  }
```

A.1.4 changeTargetPosition

```
//------------ Enumerator method to change the centipedes target position ------------

    IEnumerator changeTargetPosition()
  {

    while(true)
    {
      agent.SetDestination(target.position);
      yield return new WaitForSeconds(1);
    }


  }
```

A.1.5 findLightRay

```
//------------ Debug method to fire rays to the source of light from the centipedes position ------------

    void findLightRay()
    {
        Debug.DrawRay(meshRoot.position, Light.transform.position, Color.red);
    }
```

A.1.6 foundShade

```
//------------ Check if the centipede is in shade ------------

    public bool foundShade()
    {
        if(Physics.Raycast(meshRoot.position, Light.transform.
    position, out hit, Mathf.Infinity, -1))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
```

A.1.7 walkToShade

```
//------------ Sets the destination of the centipede to one with
    shade ------------

    void walkToShade(Vector3 pos)
    {
        agent.SetDestination(pos);

    }
```

A.1.8 isMoving

```
//----------- Update the main (head) movement of the creature
    ------------


     void isMoving()
     {
          timer += Time.deltaTime;
          if (timer >= wanderTimer)
          {
               newPos = RandomNavSphere(meshRoot.position, 100, 1);
               target.position = newPos;
               timer = 0;
          }
     }
```

A.1.9 UpdateMovement

```
//----------- Update Centipedes movement  ------------

void UpdateMovement()
  {
          int randomPosition = 0;
          if (foundShade() == true)
          {
               if(timer >= 4.98)
               {
                    print("Shade Here");
                    shadePositions.Add(meshRoot.position);
                    randomNum = Random.Range(0, 9);
                    timer = 0;
                    print("Dice Roll = " + randomNum);
               }
          }
          timer += Time.deltaTime;
          if(findShade == true)
          {
```

```
            if ((shadePositions.Count > 1) & (randomNum <= 4))
            {
                randomPosition = Random.Range(0, shadePositions.
Count);
                walkToShade(shadePositions[randomPosition]);
                //fix this
            }
            else
            {
                isMoving();
            }
        }
        isMoving();



        findLightRay();



        //------------ Update Agent ------------
        float targetSpeed = maxSpeed;

    //agent.SetDestination(newPos);
    //agent.SetDestination(target.position);
    //agent.SetDestination(target.position);
    //------------ Update Slither side-to-side motion ------------

    float speedPerc = speed / maxSpeed;

    float targetSwimFrequencyX = swimFrequencyRangeX.x + (speedPerc
* (swimFrequencyRangeX.y -
            swimFrequencyRangeX.x));
    float targetSwimAmplitudeX = swimAmplitudeRangeX.x + (speedPerc
* (swimAmplitudeRangeX.y -
            swimAmplitudeRangeX.x));

    targetSwimFrequencyX = swimFrequencyRangeX.x * speedPerc;
    targetSwimAmplitudeX = swimAmplitudeRangeX.x * speedPerc;

```

```
        swimFrequencyX = Mathf.Lerp (swimFrequencyX,
    targetSwimFrequencyX, Time.deltaTime * 3.0f);
58
        float angleX = Mathf.Sin(slitherTimer);
60
        Quaternion swimComponent = Quaternion.Euler (Vector3.up * Time.
    deltaTime * angleX * targetSwimAmplitudeX
62              * slitherTurnRate);

64      //------------ Move the head (+ rest of worm) to follow agent
    ------------

66      Transform head = VirtualBodyParts[0];

68      Vector3 vectorToAgent = head.position - agent.transform.position
    ;
        Vector3 dirToAgent = vectorToAgent.normalized;
70      float distanceToAgent= vectorToAgent.magnitude;

72      float slowingDistance = 5.0f;

74      speed = maxSpeed * (distanceToAgent - 2.0f / slowingDistance);


76
        //------------ Update head rotation to follow terrain and target
    ------------
78
        Quaternion lookQuat;
80
        Vector3 vectorToPlayer = (BodyParts[0].position - target.
    position);
82      Vector3 dirToPlayer = vectorToPlayer.normalized;
        float distanceToPlayer = vectorToPlayer.magnitude;
84
        lookQuat = Quaternion.LookRotation(dirToAgent);
86
        Quaternion rot = Quaternion.Slerp(head.transform.rotation,
    lookQuat, 3.0f * Time.deltaTime);
88      head.transform.rotation = rot * swimComponent;
```

```
90          //Changing this bottom line --------
            //      meshRoot.transform.rotation = Quaternion.Lerp(meshRoot
    .transform.rotation,
92          //  Quaternion.LookRotation(dirToPlayer) * Quaternion.Euler
    (0,180,-90), Time.deltaTime * 15.0f);
     meshRoot.transform.rotation = Quaternion.Lerp(meshRoot.transform
    .rotation,
94              Quaternion.LookRotation(dirToPlayer) * Quaternion.Euler
    (0,180,-90), Time.deltaTime * 15.0f);

96      headHeight = Mathf.Lerp(headHeight, targetHeadHeight, Time.
    smoothDeltaTime * 0.2f);

98      //----------- Update Translation ------------

100     head.Translate(-VirtualBodyParts[0].forward * speed * Time.
    deltaTime, Space.World);

102     //----------- Timers ------------

104     slitherTimer += (Time.deltaTime * swimFrequencyX);

106 }
```

A.1.10 FindAllTailBones

```
//------------ Find all the tail bones (will have "Tail" string in
    the name ------------

  void FindAllTailBones()
  {

    Transform[] temp = firstTailBone.GetComponentsInChildren<
    Transform>();

    foreach(Transform bone in temp)
    {
      if(bone.name.Contains("Tail"))
        TailBones.Add(bone);
    }

    print("Found " + TailBones.Count + " tail bones");

  }
```

A.1.11 UpdateBones

```
//------------ Update the bones of the model to rotate and follow
    body parts ------------

  void UpdateBones()
  {
    meshRoot.position = BodyParts[0].position - new Vector3(0,0.5f
    ,0);

    for(int i=0; i<TailBones.Count-1; i++)
    {

      if(updateBonePositions)
        TailBones[i].position = BodyParts[i].position;

      TailBones[i].LookAt (BodyParts[i+1].position, BodyParts[i].up)
    ;
            //Problem was bottom row was ... Quaternion.Euler (-90,
    90, 0);
            TailBones[i].rotation *= Quaternion.Euler (0, 90, 0);
    }
  }
```

A.1.12 OnDrawGizmos

```
//------------ Draw the Gizmos in editor ------------

  void OnDrawGizmos()
  {
    Vector3 prevPoint = Vector3.zero;

    foreach(Vector3 point in path)
    {
      if(prevPoint != Vector3.zero)
      {
        Debug.DrawLine(prevPoint, point, Color.green);

        //DebugExtension.DrawCircle(point, Vector3.up, Color.red,
   0.25f);

      }
      prevPoint = point;
    }
  }
```

A.1.13 isAttacking

```
void isAttacking()
    {
        if(attack == true)
        {
            if(Vector3.Distance(meshRoot.position, enemy.position)
<= 20.0)
            {
                speed += speed;
                slitherTurnRate = 8;
                agent.stoppingDistance = 1;
                agent.SetDestination(enemy.position);
                enemy.transform.Translate(new Vector3(-meshRoot.
position.x, 0, -meshRoot.position.z) * Time.deltaTime);
            }
        }
        else
        {
            //print("Dist = " + Vector3.Distance(meshRoot.position,
enemy.position));
            slitherTurnRate = 4;
            agent.stoppingDistance = 10;
            swimFrequencyRangeX = new Vector2(1, 1);
            swimAmplitudeRangeX = new Vector2(5, 5);
        }
    }
```

A.1.14 UpdateBodyParts

```
//------------ Update the reference body parts ------------

  void UpdateBodyParts()
  {
    for(int i=0; i<numSegments; i++)
    {
      Vector3 tempPos = VirtualBodyParts[i].position;

      if(i == 0)
            {
                Debug.DrawLine(BodyParts[0].position, BodyParts[0].
    position + attackVector, Color.red);
            }

      Vector3 elevation = new Vector3(0,headHeight,0);


      if(cobraMode)
      {
                agent.height = agent.height + 10;
        elevation += Vector3.up * (cobraCurve.Evaluate( 1.0f - ((
    float)i/(float)numSegments)));
      }

      BodyParts[i].position = tempPos + elevation;
    }

  }
```

A.1.15 DrawDebugPath

```
//------------ Draw the debug path ------------

  void DrawDebugPath()
  {
    Vector3 prevPoint = Vector3.zero;

    foreach(Vector3 point in path)
    {

      if(prevPoint != Vector3.zero)
            {
                //Debug.DrawLine(prevPoint, point, Color.green);
            }


            prevPoint = point;

    }
  }
```

A.1.16 Update

```
//----------- Update called once per frame------------

  void Update () {

    //Update movement of creature
    UpdateMovement();
        isAttacking();

    //----------- Update the path on ground ------------

    distanceToLastPathNode = (path.First.Value - VirtualBodyParts
    [0].position).magnitude;
    percentage = distanceToLastPathNode / pathResolution;


    if(distanceToLastPathNode > pathResolution)
    {
      path.AddFirst(VirtualBodyParts[0].position);
      path.RemoveLast();  //remove track behind snake

      distanceToLastPathNode = (path.First.Value - VirtualBodyParts
    [0].position).magnitude;
      percentage = distanceToLastPathNode / pathResolution;

      //print(percentage);
    }


    //Move all the virtual body part segments along path on ground
    MoveVirtualBodyPartsAlongPath();

    //Move the Body Part segments (virtual body part position +
    elevation)
    UpdateBodyParts();

    //----------- Update Debug ------------

    if(showDebug != prevShowDebug)
```

```
        UpdateDebug();
37
    prevShowDebug = showDebug;
39
    DrawDebugPath();
41  }
```

A.1.17 LateUpdate

```
1 //------------ Late Update (happens last) ------------

3   void LateUpdate()
    {
5     UpdateBones();
    }
```

A.1.18 MoveVirtualBodyPartsLongPath

```
//----------- Calculates where virtual body parts should be on the
    path head has previously created ------------

  public void MoveVirtualBodyPartsAlongPath()
  {

    AlignToTerrain(VirtualBodyParts[0]);


    for(int i=1; i < VirtualBodyParts.Count; i++)
    {
      Transform curBodyPart = VirtualBodyParts[i];
      Transform prevBodyPart = VirtualBodyParts[i-1];

      Vector3 vectorToParent = prevBodyPart.position - curBodyPart.
    position;
      float distanceToParent = Vector3.Distance(prevBodyPart.
    position, curBodyPart.position);

      float distanceLeft = distancesToHead[i];

      Vector3 segmentStart = path.First.Value; //current front of
    path

      //print("-----");

      int k=0;

      float segmentLength = 0.0f;

      foreach(Vector3 point in path)
      {

        if(k++ == 0)
        {
          //     continue;
        }

```

```
        //see how far this point on path is away from current
    segment

        Vector3 segmentEnd = point;
        Vector3 segmentDiff = segmentEnd - segmentStart * 1;

        segmentLength += segmentDiff.magnitude;

        if(segmentLength >= distanceLeft)
        {
          float percentageAlongSegment = distanceLeft /
    segmentLength;

          Vector3 newSegmentPos = segmentStart + (segmentDiff * (1.0
    f - percentage));

          VirtualBodyParts[i].position = Vector3.Lerp(
    VirtualBodyParts[i].position, newSegmentPos, Time.deltaTime *
    10.0f);

          break;
        }

        segmentStart = segmentEnd;
      }
    }

}
```

A.1.19 AlignToTerrain(Transform segment)

```
//------------ Align the body parts to the terrain ------------

  float segmentHeightAboveTerrain = 0.15f;
  float terrainLerpSpeed = 100.0f;

  public void AlignToTerrain(Transform segment)
  {
    RaycastHit lr;

    Vector3 bonePos = segment.position;
    Vector3 offset = new Vector3(0,5,0);


    if(Physics.Raycast(bonePos + offset, -Vector3.up, out lr, Mathf.
    Infinity, 0))
    {
      float height = 0.0f;

      Debug.DrawLine (bonePos + offset, lr.point, Color.green);

      segment.position = Vector3.Lerp (segment.position, lr.point +
    new Vector3(0,segmentHeightAboveTerrain + height,0), Time.
    deltaTime * terrainLerpSpeed);
    }
  }
```

A.1.20 CreateVirtualBodyParts

```
//------------ Creation of the Virtual Body Parts ------------

  public void CreateVirtualBodyParts()
  {
    distancesToHead = new float[numSegments];

    //Create all the segments

    for(int i = 0; i < numSegments - 1; i++)
    {
      Transform newpart = (Instantiate(virtualBodyPrefab,
    VirtualBodyParts[VirtualBodyParts.Count -1].position + (Vector3.
    forward * followDistance), Quaternion.identity) as GameObject).
    transform;

      newpart.SetParent(transform);
      VirtualBodyParts.Add(newpart);
    }


    //Store each segment's ideal position from head

    for(int i=0; i<numSegments; i++)
    {
      distancesToHead[i] = (VirtualBodyParts[i].position -
    VirtualBodyParts[0].position).magnitude;
    }

    //Create the initial path

    for(int i = numSegments-1; i >= 0; i--)
    {
      path.AddFirst(VirtualBodyParts[i].position);
    }


  }
```

A.1.21 CreateBodyParts

```
1  //------------ Creation of the Body Parts ------------
2
3
4    public void CreateBodyParts()
5    {
6
7      foreach(Transform part in VirtualBodyParts)
8      {
9        Transform newpart = (Instantiate(bodyPrefab, part.position,
       Quaternion.identity) as GameObject).transform;
10
11       newpart.SetParent(transform);
12       BodyParts.Add(newpart);
13     }
14
15   }
```

A.1.22 UpdateDebug

```
//------------ Switch debug on and off as required ------------


  public void UpdateDebug()
  {

    MeshRenderer[] renderers = transform.GetComponentsInChildren<
    MeshRenderer>();

      print("there are " + renderers.Length + " renderers");

    foreach(MeshRenderer renderer in renderers)
      {
        if(showDebug)
          renderer.enabled = true;
        else
          renderer.enabled = false;
      }

  }
```

A.2 Poster

This section displays the poster presented at the 2018 School of Computer Science expo at the University of Bangor, Wales.

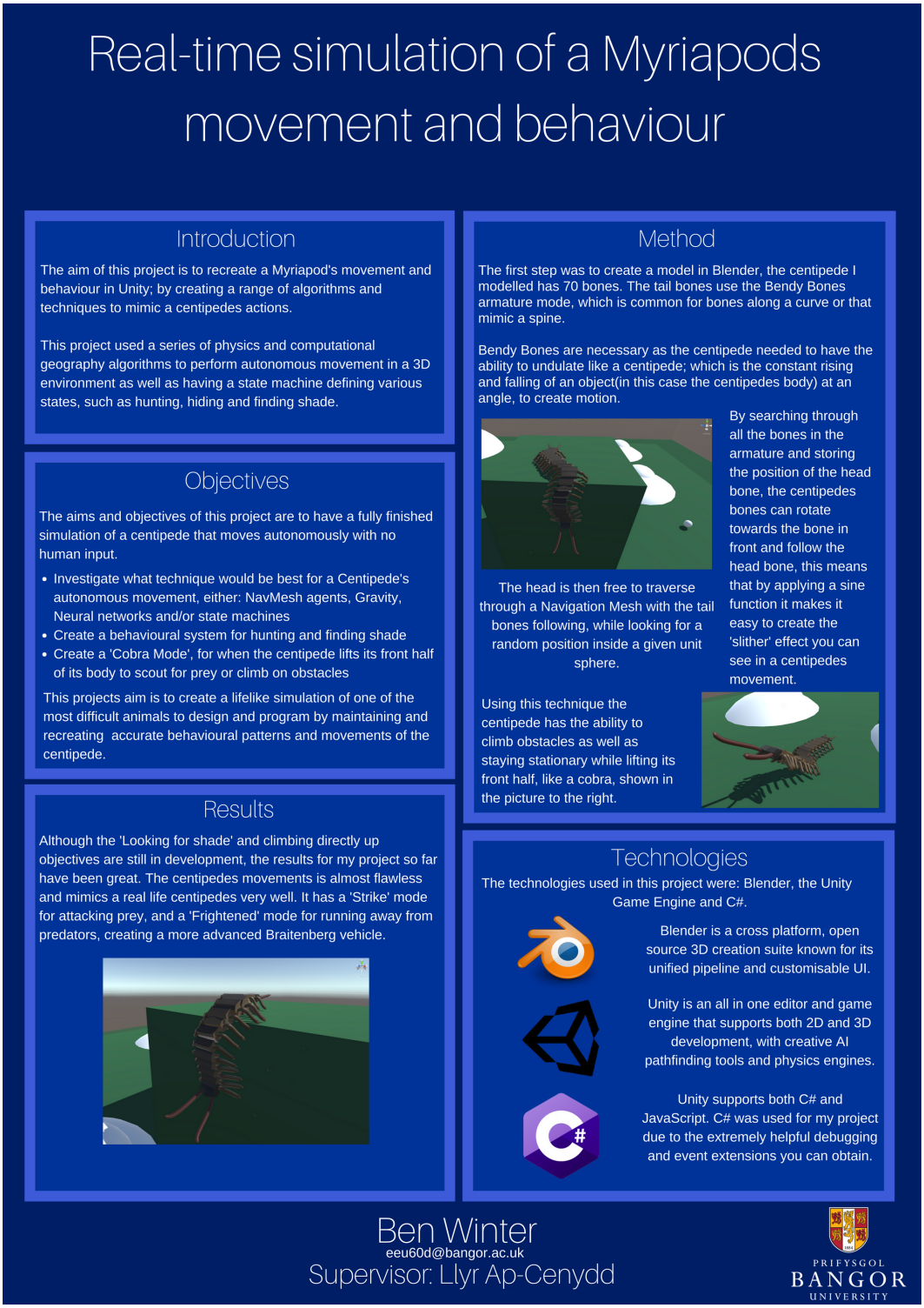


Figure A.1: Real-time simulation of a Myriapods movement and behaviour poster, 14/03/2018