Procedural Animation and Artificial Intelligence for Immersive Horror

Henry Ha u2000231

Supervised by Alex Dixon
Third Year, 2022-2023

March 24, 2024

Abstract

This project aims to develop a procedural animation library which abstracts the management and creation of procedural animations. By combining procedural animations produced by this library with AI techniques, an emergent NPC will be produced alongside a sample game.

Keywords— Procedural Animation, Artificial Intelligence, Behaviour Tree

Contents

Chapter 1

Introduction

In horror games, paramount to the player's immersion is the way it portrays its antagonists. For horror games to be effective, it is crucial that they are portrayed in a believable and immersive manner, consistent with the environment in which they inhabit. In facilitating these qualities, many developers have focused their efforts on high quality animation in order to establish antagonists with a great sense of weight and presence, alongside sophisticated AI to create the effect of realism and emergent behaviour. However, the traditional means of which 2D games are animated often conflict these objectives, commonly requiring a large amount of time and resources which constrains smaller independent game developers.

This project aims to reduce these costs, by developing a library for an alternative form of animation - procedural animation. This library aims to alleviate some of the common limitations associated with the development and management of procedural animations, as well as synthesise AI techniques to create dynamic behaviours for horror antagonists.

1.1 Background

Traditionally, 2D games are animated using keyframe animation. This technique relies on dividing a character's animation into several frames which combine to create the illusion of a smooth motion [1]. Namely, Unity uses this system in conjunction with a state machine, allowing multiple animations to be blended together [2]. However, using such a technique incurs expensive development costs. For instance, resources required to produce the sprite sheets for keyframe animation scales immensely with the number of animations a character has, along with their desired smoothness and quality. Due to this, a single character could easily incur sprite sheets consisting of hundreds of different sprites, which is both expensive and time consuming to produce. Maintaining these animations is especially costly, as character or animation changes in turn require entire revisions of sprite sheets. As a result, using traditional animation can be challenging for developers who lack the expertise or resources to do so.

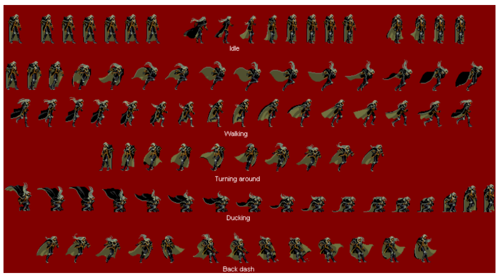


Figure 1.1: Alucard's spritesheet from Castlevania: Symphony of the Night.

Despite these heavy costs, the results of keyframe animation can be lackluster. Due to the constraints of using static frames, keyframe animation can be difficult to coordinate with other game elements without extensive developer involvement. In particular, synchronising both elements can be challenging due to the dynamic nature of video games, ultimately limiting smaller developers to simpler animations. Consequently, games such as Detention [3] suffer due to its static and rigid animations, which often undersell the threat of its antagonists. This is especially problematic for horror games as animation quality closely ties into the believability of an antagonist, ultimately diminishing its effectiveness.

Figure 1.2: A scene from Detention [3] which uses keyframe animation.

1.2 Motivation

Procedural animation is an alternative which relieves some of these limitations. Instead of using a sprite sheet, this technique relies on algorithms to generate motion onto a single sprite [4], thus removing the constraints associated with keyframe production and use. Primarily, this is due to the decoupled nature of this technique, which allows for character designs and animations to be changed independently. This is in vast contrast to the work required to redraw multiple keyframes or, worse, entire sprite sheets. In addition, generating real-time behaviours provide the advantages of unique and dynamic animations. This allows characters to interact more naturally with the environment and other game elements due to its flexibility. Owing to their implementation of procedural animation, games such as Rain World [RAIN] and Carrion have been praised for their satisfying gameplay and dynamic visuals [5], [6]. In particular, Rain World manages to synthesise this technique with AI [7] to generate believable ecosystems filled with impressive creatures.

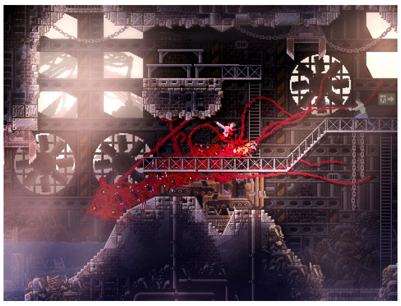


Figure 1.3: Footage from Carrion.

However, procedural animation does not come without its limitations. In order to manipulate sprites, this technique relies on bone-based (or rigged) sprites. Due to this, subtler animations such as facial expressions can be particularly difficult to animate procedurally as they are not commonly rigged components. Moreover, without adequate technical knowledge, managing and implementing procedural animations can be both challenging and unintuitive, potentially dissuading beginner programmers from implementing it into their projects.

As a result, the objective of this project is to produce a Unity library which assists the creation and use of procedural animations. By including various helper functions alongside an animation management system, this library should reduce the extent of these limitations, ultimately streamlining procedural animation development.

Lastly, a small sample game will be delivered alongside the library. This will combine game elements such as behaviour trees with procedural animations build from the library. Primarily, this is done to demonstrate the dynamic behaviours possible from combining such elements, as well as display the possible use-cases of the library.

Chapter 2

Existing Works

Procedural animations are vastly varied in the ways they are implemented [4]. This is particularly true in terms of the ways sprites can be rigged and the algorithms used to animate them. This section aims to break down some of these methods, by discussing the benefits and limitations they bring to implementation and gameplay. Specifically, analysing the structure of these techniques will inform the design of the library and its abstractions.

2.1 Rain World

Rain World is an indie game that achieves a unique physics-based implementation of procedural animation. Through its combination of AI techniques and Unity's physics engine, it delivers a believable ecosystem of NPCs and as a result, is one of the main inspirations for this project. Primarily, the analysis of Rain World's approach to procedural animation and design will be drawn from Joar Jakkobson's GDC talk [7] on the creation of his "Daddy Long Legs" creature.

2.1.1 Physics-Based Procedural Animation

Procedural animation requires a means to manipulate a sprite or model. Traditionally, this is done through bone-based rigging which determines which bones influences what part of the model when manipulated [8]. This opens the model up to skeletal and procedural animations, and remains as one of the most prevalent rigging techniques today, as seen in games such as Rayman Legends [9] and Broken Age [10].

However, instead of using a traditional bone-based rig system, Rain World's creatures unconventionally rigged using Unity colliders. Commonly, these are used to handle collisions between game objects [11]. This rigging method is achieved by forming the creature with multiple colliders, connected via Unity's hinge joint components [4]. As the name suggests, this component allows two colliders to be attached and rotate about each other [12].

Using this rig system, Rain World creates its creature's procedural animations by leveraging Unity's physics engine and hinge joint constraints. Proceeding, will we discuss the way that this animation rig is used in conjunction with procedural animations to create expressive behaviours. As shown Joar Jakobsson his "Daddy Long Legs" example, the creature is split into two different behaviour components: the main body and the tentacles.

Firstly, the tentacles follow a cycle of behaviour which can interpreted as a target selection cycle. During this cycle, two positions are retained: a current grab position and the next desired grab position. This current grab position denotes where the end effector of should be moving towards or latched to. Meanwhile, the next grab position maintains a constantly updated position, depending on the primary body's location and orientation. This update occurs every game cycle, and is done so using a simple point system. In summary, this point system scores random positions in a room based on its distance to some ideal position, relative to the main body's position and orientation. A comparison is then conducted between the position and the current grabbing position, where if it is scored better the position will be adopted as the next grab position in that given cycle. Thus, this technique ensures that tentacles are constantly positioned towards the main body's direction of travel.

To determine whether or not to update the current grab target to the next position, Rain World keeps track of the distance between the two targets. If this distance surpasses an upper bound, the current position simply adopts the next position, at which point the tentacle begins traversing to the newly assigned target.

In transitioning to this new position, Rain World uses A* pathfinding. Instead of having the tentacle arbitrarily moving straight towards the new target, it's trajectory takes on a natural path around the environment to avoid obstacles.

As a standalone component, the main body does comparatively less than the tentacles. Primarily, the main body pathfinds through the environment with a Rigidbody component attached. Rigidbody components in Unity enables physics simulations onto a game object [13], allowing them to experience forces such as gravity and drag. However, to achieve the physics driven reactivity seen in-game, Rain World sets the gravity and speed scales of the main body to be proportional to the amount of tentacles in a “grab” state via the Rigidbody component. This efficiently achieves the illusion of tentacle elasticity.

2.1.2 Analysis

How Rain World dissects it's ”Daddy Long Legs“ by behaviour encapsulates the modular approach taken to procedurally animate its creatures. This presents an intuitive workflow for animating multiple limbs of a sprite, as modularity provides the benefits of more maintainable and atomic behaviours [14].

Furthermore, Rain World's approach to rigging meant that it was highly compatible with Unity's physics tools, and practically removes the need to add colliders later to solve issues such as collision detection. This method of rigging is, however, time-consuming and highly specific to the project in question, due to how it paints sprites on top of the rig. This adoption might have also been due to the lack of first-party rigging packages at the time, as Unity's 2D animation package was only released in 2018 [15].

In terms of animation quality, Rain World's utilisation of A* pathfinding adds a degree of physicality to the environment, as it ensures the monster's tentacles react as consistently to the environment as the primary body does. This continuous motion achieved also work in tangent with the main body's goal and position, allowing the creature to effectively convey intent. This in turn alludes to the player that the creature is intelligent. In combination with the adjusting physics parameters of the body, the animations coalesce to convey reactivity and weight within the creature's movement.

Lastly, Rain World's implementation of its target selection algorithm introduces randomness into the creature's movement. Due to the scoring system, each next target for a given cycle is a random improvement towards the goal. This randomness contributes to the illusion of emergent behaviour. On the other hand, this is at the cost of performance as it runs at every game cycle. Dependant on frame rate, this means it can run as much as 60 times per second.

2.2 The Dark Cave

The Dark Cave is a game jam project which uses a more traditional bone-based procedural animation technique. Despite this, it achieves a relatively adaptable animation which follows a logical cycle not dissimilar to Rain Worlds'. This section follows the tutorial [16] tied to the game to analyse the implementation used to animate its spider enemy.

2.2.1 Skeletal Procedural Animation

Procedural animations can be used to generate natural limb movements. Specifically, with use of Unity's 2D animation package and inverse kinematics management system, The Dark Cave simulates a crawling leg animation.

Inverse kinematics is a popular technique which determines the joint angles required along a limb to achieve a certain position on its end effector. Unity's IK manager maintains the desired end effector location as a solver target game object and adjusts the joint angles accordingly to reach that position. This enables sprites to be more easily manipulated and simulates common limb constraints such as that of real life arms and legs.

Using these IK and rigging solutions, the spider retains two targets: the `currentTarget` and `desiredTarget`. Respectively, they represent the target the solver target is constantly moving towards and the next ideal target to move towards afterwards.

`currentTarget` is simply implemented within the script's `Update()`, which runs each frame. This makes sure the solver target is constantly moving towards the `currentTarget`, which in practise, makes it so that it can stay latched to a target. In contrast, implementing the target selection to update the `desiredTarget` requires a more specific design. One which encapsulates the behaviour of finding a suitable, solid surface within the path of the spider's motion.

Just like Rain World, target selection is performed each game cycle. It's primary difference in this respect in *how* it performs the target selection. Raycasting is a physics technique that directs a ray from a given point, returning collision information such as the game object hit or position of intersection. Appropriately, this raycast can be offset from the spider's body and aimed downward to detect a suitable position on the floor collider. During each frame, this method updates the position stored within `desiredTarget`.

Next, to determine whether the `currentTarget` adopts the `desiredTarget` for a given frame is determined by a condition, which is checked each cycle. For The Dark cave, this condition is based on the distance between these two targets. Once this distance surpasses a set bound, `currentTarget` is simply updated to the current `desiredTarget`. During this transition, as the solver target is moving to its new position, its motion till it reaches the point follows an animation curve. This curve is shaped as a parabola, to mimic the rising motion of each limb as it moves.

Cumulatively, these implementations form a cycle of animation not unlike Rain Worlds'. Between them, both cycles share an implementation of some current position, a constantly updated next position, a condition denoting whether or not a current position is updated and finally, the presence of a transition function when moving to a new target. Therefore, despite vast differences in technique and motion, both cycles are shown to be structurally identical.

2.2.2 Analysis

By isolating single animations to individual scripts, The Dark Cave's tutorial offers a simple and direct design workflow. However, this workflow comes with some caveats.

Firstly, the developer is required to implement and maintain the entire animation cycle themselves. In conjunction with how it positions each individual component discussed into the script's update cycle, this introduces both a lack of modularity and organisation to the workflow. This additional noise makes the implementation less maintainable, especially in terms of collaboration due to a lack of readability.

Moreover, when breaking down procedural animations into these individual components, we can find that animations are often a variation of another. For instance, a humanoid leg may use the same raycasting target selection method used by The Dark Cave's spider, yet require a different transition function or condition. But, due to the lack of modularity, the developer does not have an intuitive way to mix and match specific animation components.

In terms of target selection, The Dark Cave provides a consistent and adaptive use of raycasting. This allows the next desired position to adjust for uneven and varying floor collider shapes, allowing it to adapt to the oncoming environment.

Chapter 3

Methodology

This section breaks down the approach taken towards project development and outlines the development methodology adopted to tackle the various project objectives.

3.1 Development Methodology

During development, the objective timeline has undergone various iterations since the project specification (Appendix B). Under the specification's timeline, it was initially planned to undergo a strictly agile methodology, which delivered an updated version of the game every 1-to-3 weeks. As demonstrated in the progress report (see Appendix A), this approach was inadequate as the initial timeline was difficult to execute. This is because it did not address sudden issues and requirements which can arrive from game development. Key to solving this issue was a flexible development methodology, which assists in the rescheduling and re-prioritisation of tasks. This was essential in maintaining an adaptable and productive development.

Appropriately, a hybrid methodology of Agile and Kanban was adopted in order to meet these needs. Under this methodology, the timeline was adjusted in order to push components of game development into the latter half of the project timeline, instead of developing it in tandem with the procedural library components. This was done to reduce the workload associated with incorporating active library changes into the game.

3.2 Tools

3.3 Project Objectives

Below shows the project objectives, broken up into four major phases of development.

3.3.1 Analysis of Procedural Animation Techniques

Understanding how procedural animations are implemented is essential in informing the direction of the library. During this phase, procedural animation techniques will undergo research and analysis. This analysis will help identify key weak points experienced during development and maintenance, informing the abstractions necessary to streamline development.

1. Research different implementations of procedural animations.
2. Analyse development process of animations with respect to its limitations.

Success Criteria

1. Identify the underlying structure and implementations of two different procedural animation techniques.
2. Identify key limitations and gap in the techniques analysed.

3.3.2 Procedural Animation Testing Environment

Procedural animation is best leveraged through its ability to interact with other game elements. In order to adequately evaluate the compatibility and flexibility required to achieve this, the procedural animation library must be provided a testing environment. This phase of development includes building two primary components: an environment to test these animations and the basic AI elements used to traverse it. This will allow the library to be tested under a wide range of realistic scenarios, encapsulating the basic functionalities of a game.

1. Build the basic geometry of the testing environment.
2. Implement A* pathfinding.
   (a) Implement a Grid Manager.
   (b) Implement A* Search algorithm to form path from Grid Manager.
3. Integrate A* Pathfinding into Behaviour Tree.

Success Criteria

1. An environment exists which contains colliders to mark it as non-traversable.
2. A system exists which integrates the A* algorithm with the Grid Manager
   (a) A system exists which can simplify the level view into a grid form, and mark obstacles in the environment.
   (b) A system exists which utilises the A* algorithm to form a path from the Grid Manager.
3. A behaviour tree exists which incorporates A* pathfinding into a behaviour tree.

3.3.3 Procedural Animation Library

With the necessities of the design approach and testing environment in place, this phase begins development on the main components of the procedural animation library. Previously implemented procedural animations will be used to guide the iterative improvements being made to the library once most functionality is in place.

1. Outline key library objectives.
2. Implement primary functionalities of the library.

Success Criteria

1. Identify key features which align with the objectives of the library.
2. A library exists which implements the key features mentioned.

3.3.4 Game Development

Finally, the last phase of the project entails development of the sample game. At this stage, the main components of the library will be built, however, many of its helper functions will be built ad-hoc during game development. This is done to provide additional functionality to the library.

1. Implement the level layout.
2. Implement the player.
   (a) Implement player controls.
   (b) Implement player animations.
3. Implement the enemy NPC
   (a) Implement the final behaviour tree of the NPC
   (b) Implement the NPC animations
4. Implement a game end condition

Success Criteria

1. A level that is complete with an suitable layout and collider structure exists.
2. A player exists with character animations and controls.
   (a) Controls for the player exist, allowing them to traverse the environment.
   (b) Animations for the player exist, which animate in accordance with controls.
3. An NPC exists, complete with behaviour tree and procedural animations.
   (a) A behaviour tree exists which extends the foundation tree made for the test environment.
   (b) A set of animations exists for the NPC.
4. A goal exists within the game to allow it to end.

Chapter 4

Procedural Animation Testing Environment

To adequately test the procedural animation behaviours on an NPC, the Unity scene requires two main components: the environment and a means to interpret it. To test the robustness of the procedural animations and pathfinding, it is essential that it is tested on a varied environment. The tools used to make this environment should allow for quick iteration, enabling the testing environment to undergo changes.

In addition, a grid manager is required in conjunction with a pathfinding algorithm to identify obstacles within the environment and form a path around them. This is used to evaluate two important characteristics of procedural animations: its compatibility with other game elements and its reactivity to the game environment.

4.1 Environment

The environment was made using Unity's tilemap system, due to its ability to be build environments quickly. Using tilemaps also allow for the use of tilemap colliders [17], which will be used in combination with the grid manager to denote whether an area is traversable or not.

4.2 Grid Manager

The grid manager and A* algorithm is adapted from Unity Artificial Intelligence Programming [18, Ch.7], with changes to adapt it from a 3D context to a 2D context. This implementation sections the desired area in a grid using a 2D array, with each column-row pair represented by a Node. These nodes are initialised with their corresponding position within the grid and stores other variables such as `fScore`, used in the A* algorithm to perform pathfinding.

As the name entails, `ComputeGrid()` (Fig. 4.1) function is responsible for generating this grid. It is called within the grid manager's `Awake()` to ensure that it is generated before the any pathfinding is initiated. It does so by running before any update cycles of the game begin. It does this by first setting up the 2D Node array, depending on the desired number of rows and columns set by the user. Each node within the array is then instantiated and iterated upon, to determine if each node is an obstacle or not. Determining whether a node is an obstacle is done by utilising Unity's `Physics2D.OverlapCircle`, which detects whether the node's position contains a collider [19]. When generated over the tilemap's colliders, it clearly maps out the obstacles in the environment.

4.3 A* Pathfinding

The pathfinding implementation makes use of the A* algorithm, a common AI technique which combines elements of Dijkstra's algorithm and Greedy Best-First Search to find the shortest path between two nodes. At a high-level, it maintains both a candidate and closed

```
    ...
    /*Before the game's initial Update or Start cycle,
     form the grid.*/
    void Awake()
    {
        ComputeGrid();
    }

    /*Compute grid based on desired number of columns and rows.
     The grid is essentially made up of a 2D array of Nodes.*/
    void ComputeGrid()
    {
        //Initialise the 2D grid array
        nodes = new Node[numOfColumns, numOfRows];

        //Initialise each node within the 2D array
        for (int i = 0; i < numOfColumns; i++)
        {
            for (int j = 0; j < numOfRows; j++)
            {
                // Cell position computed as centre of each grid
                Vector2 cellPos = GetGridCellCenter(i, j);
                Node node = new(cellPos);

                // Detects whether there is a collider in each grid
                var collisions = Physics2D.OverlapCircle(cellPos,
    gridCellSize / 2 - obstacleEpsilon, layermask);
                if (collisions != null)
                {
                    node.MarkAsObstacle();
                }
                nodes[i, j] = node;
            }
        }
    }
    ...
```

Figure 4.1: `GridManager`'s `ComputeGrid()` function to generate the grid in the Unity scene.

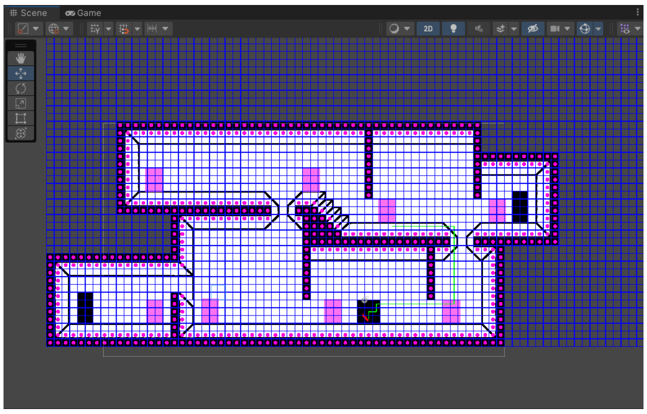


Figure 4.2: A visualisation of the grid manager placed onto the test environment. Pink circles denote grids that are marked as obstacles.

list of nodes, which represent the list of known, but unexplored nodes, and a list of explored nodes respectively. Specifically, the closed list is represented by a priority queue of nodes instead of a traditional list as it allows nodes to be ordered by their `fScore`, which is required by the algorithm. `Enqueue()` achieves this by sorting the node list according to the `fScore` *after* a new node has been added.

Iteratively, the A* algorithm (Fig. 4.3) selects the node from the candidate list with the smallest estimate cost (`fScore`) from the node to the goal. This node is then expanded, by adding it's unexplored neighbours to the candidate list and updating their corresponding `gScore` and `fScore`. This `gScore` represents the costs associated with taking the path from the start node to that node, whilst `fScore` combines this score with a heuristic estimate, which estimates the cost from that node to the goal.

```
...
public List<Node> FindPath(Node start, Node goal)
{
    /*Initialise both opened and closed lists.*/
    ...
    /*While nodes remain in the open list, dequeue
     the lowest fScore node, then add and update
    neighbour values.*/
    while (openList.Length != 0)
    {
        // Get lowest fScore node
        node = openList.Dequeue();
        // If goal is same node, path is found
        if (node.position == goal.position)
        {
            return CalculatePath(node);
        }
        // Gets single instance of GridManager to gain node neighbours
        var neighbours = GridManager.instance.GetNeighbours(node);

        /*Traverse each neighbour, update their scores and then insert
         them into the PriorityQueue, given they are not in the closed list.*/
        foreach (Node neighbourNode in neighbours)
        {
            if (!closedList.Contains(neighbourNode))
            {
                //Total Cost So Far from start to this neighbour node
                float totalCost = node.costSoFar + GridManager.instance.StepCost;
                //Estimated cost for neighbour node to the goal
                float heuristicValue = HeuristicEstimateCost(neighbourNode, goal);
                //Assign neighbour node properties
                neighbourNode.costSoFar = totalCost;
                neighbourNode.parent = node;
                neighbourNode.fScore = totalCost + heuristicValue;
                //Add the neighbour node to the queue
                if (!closedList.Contains(neighbourNode))
                {
                    openList.Enqueue(neighbourNode);
                }
            }
        }
        // Add that node to the closed list
        closedList.Add(node);
    }
...
```

Figure 4.3: `AStar`'s `FindPath()` function to calculate a path using the A* algorithm and the `GridManager` instance.

It is important that the heuristic chosen to calculate the `fScore` is admissible, by not overestimating the actual cost to the goal. The heuristic in this case is simply the distance between the current node and the goal. This ensures that the shortest path possible is provided if one exists. Otherwise, it will return failure.

4.4 Behaviour Tree

Behaviour trees are an increasing popular technique used to program AI-driven NPCs. Games such as Alien: Isolation have used this technique to success in order to portray detailed and adaptive decision-making [20].

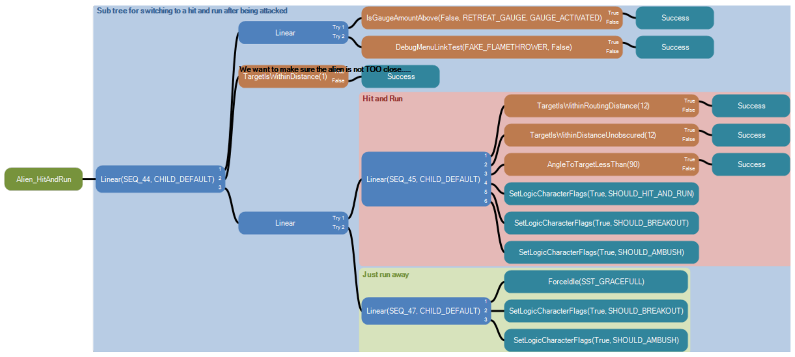

Sub tree for switching to a hit and run after being attacked
Alien_HitAndRun
Linear(SEQ_44, CHILD_DEFAULT)
Linear
Try 1
Try 2
IsGaugeAmountAbove(False, RETREAT_GAUGE, GAUGE_ACTIVATED)
DebugMenuLinkTest(FAKE_FLAMETHROWER, False)
True
False
Success
We want to make sure the alien is not TOO close.....
TargetIsWithinDistance(1)
Hit and Run
Linear(SEQ_45, CHILD_DEFAULT)
TargetIsWithinRoutingDistance(12)
TargetIsWithinDistanceUnobscured(12)
AngleToTargetLessThan(90)
SetLogicCharacterFlags(True, SHOULD_HIT_AND_RUN)
SetLogicCharacterFlags(True, SHOULD_BREAKOUT)
SetLogicCharacterFlags(True, SHOULD_AMBUSH)
Just run away
Linear(SEQ_47, CHILD_DEFAULT)
ForceIdle(SST_GRACEFULL)
SetLogicCharacterFlags(True, SHOULD_BREAKOUT)
SetLogicCharacterFlags(True, SHOULD_AMBUSH)

Figure 4.4: Hit and run behaviour tree of the xenomorph NPC from Alien: Isolation [21].

Particularly, behaviour trees are leveraged through its ability to not only perform decision making, but decision execution. Structurally, a behaviour tree is a data structure which follows the format of a acyclic tree graph [22]. Commonly, through depth-first search, behaviour trees run from its roots down to its leaf nodes. During this, it traverses through behaviour nodes, composite nodes and decorator nodes [23]. Each category of nodes report one of three statuses after execution: `SUCCESS`, `FAILURE` or `RUNNING`. The ways in which a node reports these allow the developer to both organise and order the decision-making procedure of an NPC.

4.4.1 Overview

This section briefly discusses the category of nodes which makes up a behaviour tree and how they communicate between each other. This will provide context for behaviour tree implementations discussed later on in the document.

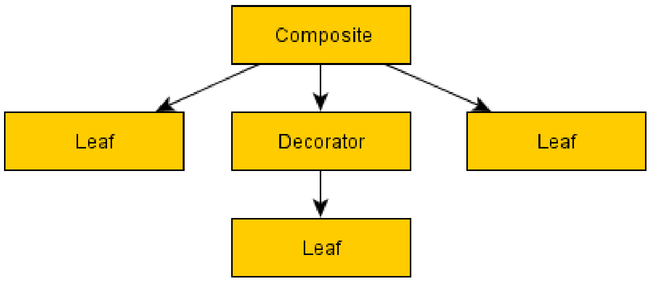


Figure 4.5: The main archetypes of behaviour tree nodes [24].

Behavioural Nodes

Behaviour trees are split into two categories of nodes: action nodes and condition nodes. Action nodes alter the game state by executing some programmed task. After execution, a fixed status is reported denoting the success of its run [23]. Actions allow tasks to be executed in response to decisions made externally from its node.

Condition nodes, on the other hand, are used to test some game property [23] in order to contribute to these decisions being made by the tree. By themselves, they do not execute any tasks as an action node does. Shooter game NPCs, for example, may commonly use an ‘isPlayerSpotted’ condition to determine whether to engage in combat. In this case, the tasks of combat can be represented by various action nodes such as ‘shoot’.

Composite Nodes

Composite nodes need have at least one child node [23], as it controls the overall flow of a behaviour tree. Usual composite nodes include selector nodes and sequence nodes.

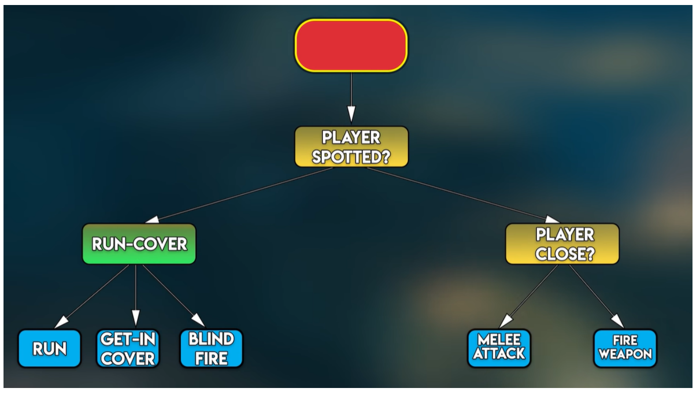


Figure 4.6: Behaviour tree example from [22].

Starting from the leftmost child node, selector nodes reports success when it identifies the first set of successful nodes amongst its children. Due to the order of execution, selector nodes essentially prioritise its children tasks, with the highest priority being the leftmost child.

In contrast, sequence nodes only report success when all of its children also report success. Execution order is also important here, as the nodes must report success in sequence from left to right. This encapsulates operating a series of tasks in order and therefore reports a failure as soon as a task fails.

Decorator Nodes

Decorator nodes represent a sub-category of composite nodes, in that it affects how its children are run. What makes decorator nodes unique to other composite nodes are that they only hold a single child node. Hence, the decorator only controls the execution of that node.

Examples of decorator nodes include repeaters and inverter nodes [23]. Repeater decorators loop the execution of its child, despite the status reported. Inverters differ in that they affect the status reported instead of the execution process. In this regard, inverter nodes invert the status reported by its child node.

Blackboard System

For nodes and behaviour trees to share information, variables are commonly updated or published to a global data structure called a blackboard. This can be useful for tracking node operations. This is important, as the behaviour tree's execution order means that nodes may rely on the operations of others before it. For condition and action nodes, this is particularly useful for accessing information from the game scene, such as NPC health or player location.

4.4.2 Behaviour Bricks

As described in Chapter 3, a behaviour tree will be implemented as part of the sample game using the Behaviour Bricks library. During the test environment phase, only the basis of the NPC's behaviour tree will be made. Incorporating AI at this early stage can help assess how the library can be applied to real world scenarios and movements that are not pre-determined by the developer.

Wander

The primary goal of the wander behaviour is to provide a solid foundation for other behaviours to be added onto in the future. As a result, the wander behaviour is created as a default behaviour cycle for others to fall back on.

As the name suggests, Wander encapsulates the cycle of behaviour of wondering around a level idly. This is meant to be a passive behaviour which initiates when there is no immediate stimulus, such as spotting the player.

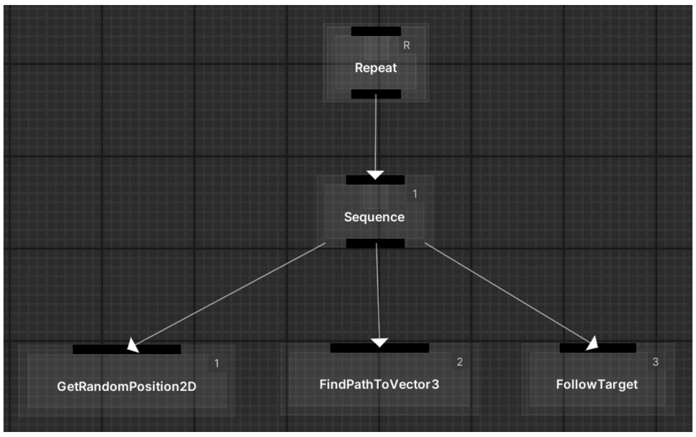


Figure 4.7: Wander behaviour tree.

It is built up using three action scripts: `GetRandomPosition2D`, `FindPathToVector3` and `FollowTarget`. As these actions follow a progression, they are run as children under a sequence node.

Firstly, `GetRandomPosition2D` generates a random 2D vector (`Vector2`) from a given range of x and y values (Fig. 4.8). By editing this range, it can be assured that the generated vector falls within the bounds of the grid manager's grid. This allows the behaviour tree to generate a path from the NPC to the random position in the next node.

```
...
    public override void OnStart()
    {
        // Generate random position given range
        randomPosition = new Vector3(UnityEngine.Random.Range(
xLowerBound,xUpperBound), UnityEngine.Random.Range(yLowerBound,
yUpperBound), 0f);
    }
...
```

Figure 4.8: `GetRandomPosition2D`'s `OnStart()` function to generate a random position.

As `OnStart()` (Fig. 4.8) is mainly used for initialisation, the task status must be delayed to `OnUpdate()` (Fig. 4.9). Hence, the corresponding check of whether a position is an obstacle is also delayed. If the check fails, due to the repeater decorator it is rerun until it can publish a suitable position to the blackboard.

Once it is published, `FindPathToVector3` (Fig. 4.10) takes the target position from the blackboard. Using the previously implemented A* algorithm to formulate a path, it then publishes the path to the blackboard as a list of nodes.

Finally, the `FollowTarget` script (Fig. 4.11) accesses this path and performs traversal. Due to the nature of Unity's update cycles, scripts complete entire runs between frames [25]. Therefore, traversing a path using a `foreach` loop is not suitable as the NPC will appear to go through walls due to how quick the motion appears to be. Therefore, the script is implemented in a way to allow the path to be traversed across several frame updates. Not too dissimilar to Unity's coroutines, Behaviour Brick's implementation of the 'RUNNING' status allows `OnUpdate()` to re-run on multiple frames [26]. This in turn delays the cycling loop of the overall behaviour, prevent the path from constantly updating under traversal.

```
...
    public override TaskStatus OnUpdate()
    {
        int goalColumn, goalRow;
        (goalColumn, goalRow) = GridManager.instance.
GetGridCoordinates(randomPosition);

        Node goalNode = GridManager.instance.nodes[goalRow,
goalColumn];

        if (goalNode.isObstacle)
            return TaskStatus.FAILED;

        //Debug.Log(randomPosition);
        return TaskStatus.COMPLETED;
    }
...
```

Figure 4.9: `GetRandomPosition2D`'s `OnUpdate()` function, which performs obstacle detection and reports status.

```
...
    void FindPath()
    {
        // May want to beam this back up to the public class
variables, KAE
        Transform startPos; //, endPos;
        Node startNode;
        Node goalNode;

        startPos = gameObject.transform;

        // Gets array position form positions
        var (startColumn, startRow) = GridManager.instance.
GetGridCoordinates(startPos.position);
        //Debug.Log("Start: " + startColumn + " " + startRow);
        var (goalColumn, goalRow) = GridManager.instance.
GetGridCoordinates(target);
        //Debug.Log("Goal: " + goalColumn + " " + goalRow);

        // Get nodes for each grids
        startNode = new Node(GridManager.instance.GetGridCellCenter(
startColumn, startRow));
        goalNode = new Node(GridManager.instance.GetGridCellCenter(
goalColumn, goalRow));

        // Find a path using nodes
        pathArray = new AStar().FindPath(startNode, goalNode);
    }
...
```

Figure 4.10: `FindPathToVector3`'s `FindPath()` function.

```
...
    public override TaskStatus OnUpdate()
    {
        // Running is restarted from scratch
        if (pathArray == null)
        {
            return TaskStatus.FAILED;
        }

        if (Vector3.Distance(pathArray[pathArray.Count - 1].position
, gameObject.transform.position) < 0.2f)
        {
            return TaskStatus.COMPLETED;
        }

        if(pathArray.Count <= 0)
        {
            return TaskStatus.COMPLETED;
        }

        if(index >= pathArray.Count)
        {
            return TaskStatus.COMPLETED;
        }

        if(Vector3.Distance(gameObject.transform.position, pathArray
[index].position) < 0.2f)
        {
            index++;
        }

        gameObject.transform.position = Vector2.MoveTowards(
gameObject.transform.position, pathArray[index].position, speed *
Time.deltaTime);
        if(index + 1 < pathArray.Count)
            nextPosition = pathArray[index].position;
        entityScript.emmediatePath = nextPosition;

        return TaskStatus.RUNNING;
    }
...
```

Figure 4.11: `FindPathToVector3`'s `FindPath()` function.

Chapter 5

Procedural Animation Library

This section captures the library development phase of the project. Outlined below is the library's objectives, design and implementation.

5.1 Library Objectives

The following library objectives are drawn from the limitations and gaps seen from the procedural animation techniques previously analysed in Chapter 2. These objectives aim to introduce quality of life changes to the development and design of procedural animations. Proceeding, the objectives will be outlined along with their corresponding success criteria. Mentions of procedural animation components refer to that of the target selection cycle, also analysed in Chapter 2.

5.1.1 Introduce Structure

For beginners not familiar with the overarching structure of target selection, it would be beneficial of the library to guide the procedure of implementing individual animations. By providing a structured means of creating, storing and managing animations, the library should introduce a streamlined workflow.

Success Criteria

1. Abstractions surrounding running the animation cycle should be handled by the library.
2. Class used to store individual animations should be structured with respect to the three primary components.

5.1.2 Simplify Code

Effective abstractions should reduce the amount of code required for procedural animation. This should lead to simpler, less complex code. As a result, leveraging abstractions through helper functions or other means should make animation code to be more readable and maintainable.

Success Criteria

1. Library should contain helper functions commonly used or useful to target selection components.

5.1.3 Streamline Limb Management

Characters in games usually contain multiple limbs which undergo the same animations. For that reason, to prevent developers from manually handling multiple limbs through their own data structures, they should be given the ability to group limbs and control them with ease.

Besides a reduction in workload, this will also allow animation code to remain focused on the primary logic of target selection.

Success Criteria

1. Library includes a system to retrieve solver/animation targets from the scene.
2. Library includes a function to animate all limbs concurrently.

5.1.4 Include Animation Management

Characters which undergo a single animation consistently can be seen as predictable and monotonous over time. Developers should be able to animate limbs with a number of different animations and control which animations are run.

Success Criteria

1. Library includes a class which allows developers to store animations.
2. Library includes a system to manage multiple different animations for a limb group.
3. Library provides a system to transition between animations.

5.1.5 Enforce Modularity

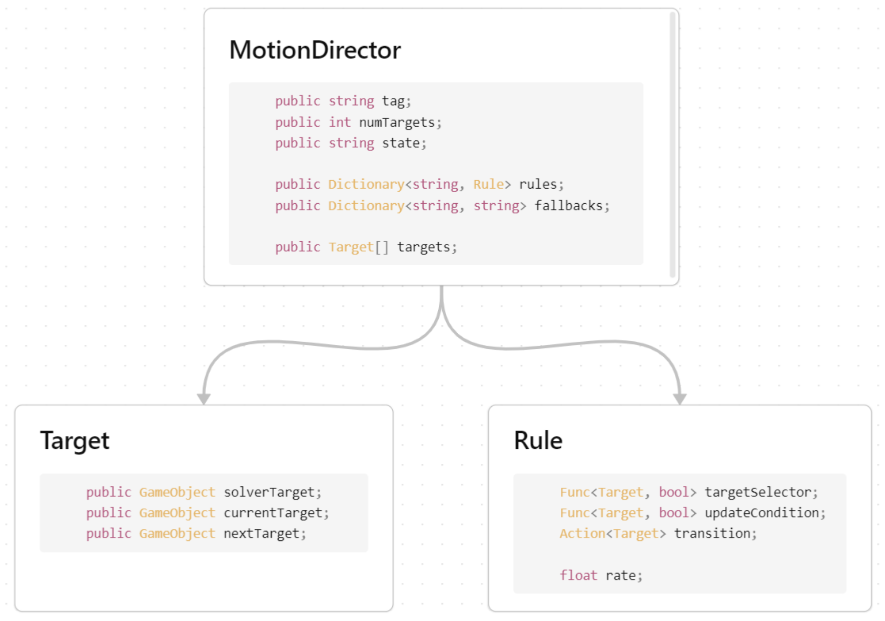
Modularity promotes both re-usability and maintainability [14]. Breaking down animations to their fundamental components extends an animation's re-usability to its components. This would allow new animations to be formed by combining different elements of target selection.

Success Criteria

1. Library stores animations as a collection of modular components.

5.2 Design Overview

Before specifically detailing how the library operates, it is important to first understand the structure of its classes. The library is made up of three primary classes: the `Target` class, the `Rule` class and the `MotionDirector` class.



```
public string tag;
public int numTargets;
public string state;

public Dictionary<string, Rule> rules;
public Dictionary<string, string> fallbacks;

public Target[] targets;
```

```
public GameObject solverTarget;
public GameObject currentTarget;
public GameObject nextTarget;
```

```
Func<Target, bool> targetSelector;
Func<Target, bool> updateCondition;
Action<Target> transition;

float rate;
```

Figure 5.1: Overview of the library's class structure.

The `Target` class retains targeting information, specific to an individual solver target. This involves its `Transform` component, which is used to manipulate its position and rotation within the Unity scene [27]. This class also includes the solver target's corresponding current and next target, which follow the same definitions seen in the target selection cycle in Chapter 2.

Meanwhile, the `Rule` class handles an isolated animation cycle. Each `Rule` contains a specific target selector, update condition and transition function combination to represent a unique animation.

Together, these two classes form a collection of objects which are maintained under an instance of the `MotionDirector` class. In summary, this main class manages a group of target solvers which undergo a common selection of animation behaviour.

5.3 Core Functionality

This section details the core features of the library and how it can help developers implementing procedural animations into their games.

5.3.1 Target Retrieval and Set Up

Following The Dark Cave's method of rigging, this project will use Unity's 2D animation package and IK manager to manipulate sprite models. As such, the library requires a means to access their solver targets from script and generate their respective target game objects. In The Dark Cave's case, it achieves this by manually placing the animation script onto each target solver [16]. This solution is adequate for a small number of limbs, but can quickly become tedious for a larger amount of limbs or characters. Instead, upon initialisation, the `MotionDirector` attempts to automate this process using its `FindSolverTargets()` function.

```
    ...
    public MotionDirector(GameObject parent, string tag)
    {
        ...
        GameObject[] solverTargets = FindSolverTargets(parent, tag);
        ...
        foreach(GameObject solver in solverTargets)
        {
            targets[i] = solver.AddComponent<Target>();
            targets[i].Init(solver);

            // Initiate Animation of each limb -- mostly used to request
     limb states
            OnAnimate += targets[i].Animate;
            // Each Rule is run from this our directory collection,
    instead of saving each Action function to each separate limb
            targets[i].OnTargetSelect += ApplyRules;
            i++;
        }
    }
    ...
```

Figure 5.2: `MotionDirector`'s constructor.

Given a parent game object and a tag, `FindSolverTargets()` traverses the parent's children in order to return an array of child solver targets with the specified tag. Once the array is returned, each solver target is used to initialise a new `Target` object. This in turn creates a new `currentTarget` and `nextTarget`, through the `Target` constructor.

When initialising `Target` objects `MotionDirector` also uses `AddComponent<Target>()` (Fig. 5.2). Essentially, this method adds the `Target` class as a component to the solver target game objects, allowing each script to access the game object's coroutine scheduler. This is important, primarily for the use of Unity's coroutine system to incorporate the `Rule` rates.

5.3.2 Rule Class

The `Rule` class exists to attempt to solve two major problems. Firstly, as the library is expected to manage multiple animations for limb group, there must be an intuitive way

```
    ...
    public GameObject[] FindSolverTargets(GameObject parent, string tag)
    {
        GameObject[] targets;
        targets = parent.GetComponentsInChildren<Transform>().Where(x =>
     x.CompareTag(tag)).Select(x => x.gameObject).ToArray();

        return targets;
    }
    ...
```

Figure 5.3: `MotionDirector`'s `FindSolverTargets()` function to get tagged solver targets.

of storing these animations. Moreover, the creation of these stored animations must be guided in a way which promotes modularity and focus onto the main logical components of procedural animation. These include the target selector, update condition and transition functions.

As discussed, `Rule`s are used to store individual animation cycles. With modularity in mind, the entire animation cycle can be summarised into its fundamental components discussed above.

Each component of target selection is stored as a delegate. Delegates allow `Rule`s to store these components as individual functions as well as structure the development of an animation.

```
    ...
    Func<Target, bool> targetSelector;
    Func<Target, bool> updateCondition;
    Action<Target> transition;
    ...
```

Figure 5.4: `Rule`'s delegate signatures for the target selection components.

Firstly, the `targetSelector` manages how the `nextTarget` is updated during runtime. It returns a boolean denoting the success or failure of a target selection run. Target selection is intended to fail only if a target that is found, if at all, is unsuitable for the `nextTarget`'s adoption (see example in Fig. 5.5). This allows developers to implement fallback animations. Fallback functions will be discussed further on in the document.

Secondly, the `updateCondition` is a conditional which determines whether a `Target`'s `currentTarget` should be updated to the `nextTarget`. Appropriately, this returns a boolean also.

Lastly, the `transition` function denotes *how* the solver target moves towards its new target. Generally, this is intended to run whenever the solver target has not reached the position of the `currentTarget`. For simpler animations, this function could simply move the solver target to the `currentTarget`'s position. However, for more complex animations, this can be greatly leveraged to incorporate methods such as A* pathfinding or animation curves into movement.

Storing animations in this manner allows different `Rule` animations to be formed due to the interchangeability of delegates. Furthermore, the process of `Rule` creation gives developers both a modular and concentrated approach to development, as it asks of them to focus on individual components instead of the overarching animation and how it is managed.

To add a new `Rule` to a `MotionDirector` object, `AddRule(State, Rule)` is used to add a state-rule pair its dictionary of `Rule` objects. States are used as a key to differentiate each `Rule` added. This means of organisation allows the `MotionDirector` to animate its solver target in accordance with what state it is in. The default state upon initialisation is simply the "default" state. This can be changed with use of the `MotionDirector`'s `SetState(State)` function.

```
public bool SpiderCrawl(Target target)
{
    Vector3 dir = (nextPosition - transform.position).normalized;
    Vector3 perpdir = (Quaternion.Euler(0, 0, -90f) * dir).normalized;
    Vector3 offset = transform.position + 4f * dir;// - dir * Vector3.Dot((mouse.transform.position - transform.position), (transform.position - target.GetSolverTarget())); // transform.position, target.GetSolverTarget()

    target.SetNextTarget(offset);

    RaycastHit2D hit = Physics2D.Raycast(offset, Vector2.down, 4f, layermask);
    Debug.DrawLine(offset, offset + Vector3.down * 4f, Color.red, 1f, false);

    if (hit.collider != null && hit.point.y < transform.position.y)
    {
        target.SetNextTarget(new Vector3(target.GetNextTarget().x, hit.point.y, target.GetNextTarget().z));
        isCrawl = true;
        return true; // Collision detected, so is adequate for nextTarget
    }
    return false; // No collider, so target selection fails
}
```

Figure 5.5: Example of a target selector and how it can be formatted to return its corresponding success and failures.

```
public void ApplyRules(Target target, string state)
{
    if (!rules[state].ApplyRule(target))
    {
        if (target.lastRule != rules[fallbacks[state]]) {
            target.isRunning = false;
            target.StopCoroutine(target.tickRate);
        }
        ApplyRules(target, fallbacks[state]);
    }
}
```

Figure 5.6: `MotionDirector`'s `ApplyRule()` function to apply the `Rule` associated to its current state.

5.3.3 Performance

`Rules` are also designed to tackle performance limitations of common procedural animation techniques. As discussed, depending on the implementation, target selection can occur as often as every game update. This scales poorly, because as the numbers of limbs that are animated increase, the more targets that need to be updated regularly. As a result, upon instantiation, every `Rule` must have a specified rate. This denotes how many seconds lapse between runs of target selection.

This implementation of the rate in `ApplyRate` relies on coroutines. Coroutines in Unity allow scripts to continue a task across multiple different game update frames. In particular, these coroutines are used in `Target` to track the last time target selection was performed onto a specific solver target. During a `Rule` run, after applying the `updateCondition` and (if applicable) `transition` functions, if it is detected that the `Target` is still running its `tickRate` coroutine, it will simply return. This skips target selection for that cycle.

5.3.4 Animate

As the `MotionDirector` maintains both the collections of `Targets` and `Rules`, animations can simply be applied to all limbs using the `Animate()` function.

```
    ...
        if (rate != 0f)
        {
            if (target.isRunning)
            {
                //Debug.Log("Delay!!");
                return true;
            }

            // Otherwise target select + apply tick rate
            //Debug.Log("No delay!!!");
            target.tickRate = target.StartCoroutine(target.ApplyRate(
    rate));
            target.isRunning = true;
        }
    ...
```

Figure 5.7: A subsection of the `Rule`'s target selection algorithm, displaying how rates are used to delay target selection.

During initialisation of the `MotionDirector` you may have noticed that each `Target` objects has had their corresponding `Animate()` functions registered to the `MotionDirector`'s `OnAnimate` event and that the `MotionDirector`'s `ApplyRules()` function has been registered to the `Target`'s `OnTargetSelect` event. In summary, this allows the `MotionDirector` to request the `Target` object all of the associated solver targets. Once the class object is received, `ApplyRules()` (Fig. 5.6) applies the `Rule` corresponding to the `MotionDirector`'s state on all of the `Target` objects, allowing them to be manipulated accordingly.

5.3.5 Fallback Animations

As previously mentioned, the boolean return type of the `targetSelector` denotes whether target selection has failed or succeeded. If a `targetSelector` function fails within upon `ApplyRule()` for a given `Target`, the `MotionDirector` attempts to find a fallback animation to apply to the `Target` (Fig. 5.6).

The `MotionDirector` retains another dictionary of state-state pairs. Using `AddFallback(State, State)`, the developer can add a state-state pair, which indicates the fallback state of the key state. For instance, adding an entry using `AddFallback("default","idle")` signifies that the "default" `Rule` falls back onto the "idle" `Rule` animation. This system introduces a hierarchy of fallback states, allowing each animation a back-up animation in the case of failure.

Chapter 6

Game Development

Building off the foundations of the testing environment and procedural animation library, the last phase of development entails building a sample horror game. This section details the development of the primary game components.

6.1 Level Design

Level design plays a crucial role in immersing the player into the gameplay experience of any game, however this is particularly true for horror games. This section describes the design behind the level of the game and how it leverages lighting to incorporate themes of horror.

6.1.1 Level Layout

Continuing the use of Unity's tilemap system, the sample game includes a maze-like level with multiple floors and ladders. Each room is intentionally designed to be varied, to create an overall unnatural level layout. This is done such that the player can feel lost due to the lack of symmetry, making the environment unpredictable. Furthermore, such an environment leverages the real-time adaptability of procedural animations.

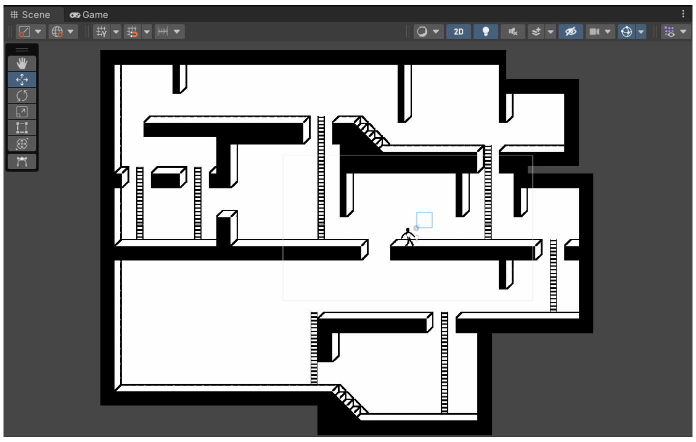


Figure 6.1: The level of the sample game.

6.1.2 Lighting

To create a sense of claustrophobia horror games such as Amnesia: The Dark Descent and Outlast use darkness as a primary proponent of the gameplay. Darkness in these games are intentionally used to obscure the player's view, reducing the amount of information gathered from the environment. Amnesia's creator finds that such reductions can enrich horror, as it allows the gaps to be filled by the player's imagination [28].

Taking inspiration from this, lighting adjustments and tilemap shadows have been incorporated into the environment to limit the player's view using the solution provided by [29].

Specifically, these conditions mean that the player must light their way through the environment in order to traverse and look out for danger.

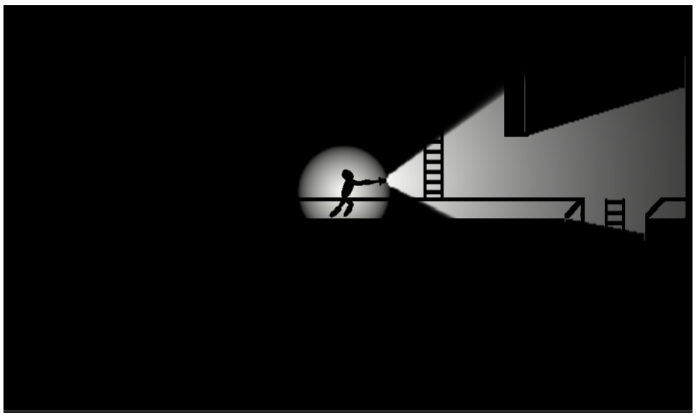


Figure 6.2: Darkness added to the level.

6.2 Player

The animations and controls of a player strongly contribute to the satisfaction felt during gameplay. Animations, in particular, provides the player's actions with feedback and weight. As such, development of the player has been done with respect to these two components.

6.2.1 Controls

When developing the player controls, the core idea was to keep the player restricted. This design choice allows the player to feel helpless and vulnerable. Thus, the platforming controls of the player are kept deliberately basic.

1. `A/D` or ← / → - Horizontal movement
2. `SPACE` - Jump
3. `W/S` or `SPACE` - Climb ladder
4. `MOUSE` - Control flashlight view

6.2.2 Animations

To showcase the procedural animation library, one of the primary objectives of game development was to incorporate satisfying player animations.

Leg

The player's legs undergo the same raycasting target selection which *The Dark Cave*'s spider uses, but with some subtle changes. The distance at which the raycasting takes place from the body varies at each iteration using Unity's `Random.Range()` function. This randomisation prevent the legs from target selecting the exact same position, allowing for more varied and natural movement.

Moreover, the direction in which the raycasting is offset from the player is with respect to the player's velocity. This allows the player's legs to adapt to the direction of movement.

Arms and Body

The arms and body orient themselves according to where the mouse is relative to the player. By attaching a spotlight to the player's arm, the game effectively achieves its flashlight view controls (Fig. 6.5). This makes the mechanic much more immersive as the player seems to adjust its hands in order to hold a torch. Despite following the same orientation behaviour, body motions are not as intense due to the weight adjustments on the solver.

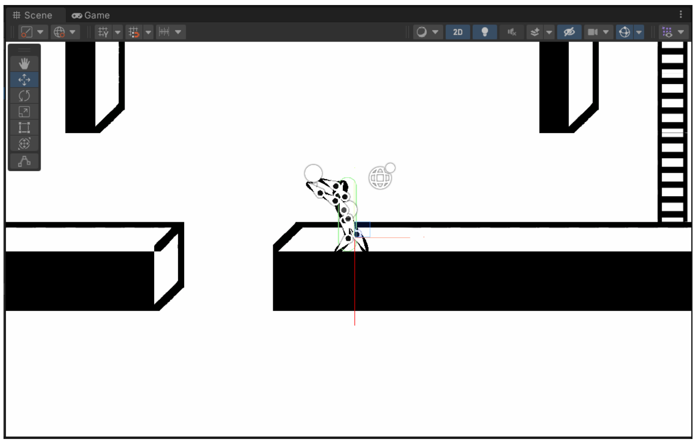


Figure 6.3: Red line displays the ray being cast to find a collider.

```
    public bool Walking(Target target)
    {
        Vector3 dir = rb.velocity.normalized;
        if (dir.x < 0)
        {
            transform.rotation = Quaternion.Euler(transform.rotation.x,
    180, transform.rotation.z);
        }
        if (dir.x > 0)
        {
            transform.rotation = Quaternion.Euler(transform.rotation.x,
    0, transform.rotation.z);
        }
        Vector3 offset = transform.position + Random.Range(-0.5f, 1.5f)
    * dir;
        target.SetNextTarget(offset);
        RaycastHit2D hit = Physics2D.Raycast(offset, Vector2.down, 3f,
    layermask);
        Debug.DrawLine(offset, offset + Vector3.down * 3f, Color.red, 1f
    , false);
        if (hit.collider != null && hit.point.y < transform.position.y)
        {
            target.SetNextTarget(new Vector3(target.GetNextTarget().x,
    hit.point.y, target.GetNextTarget().z));
        }
        //Debug.Log("No Collider");
        return true;
    }
```

Figure 6.4: Target selector for the legs of the player.

6.3 Enemy NPC

This section builds upon the foundations of the wander behaviour tree, A* pathfinding and procedural animation library to build an enemy NPC which chases the player.

6.3.1 Behaviour Tree

As the initial wander behaviour was made to be a fallback behaviour, it was not suitable to simply add to its tree. Instead, the wander behaviour was set as a low priority behaviour under a priority node. Behaviour Brick's implementation of a priority node ensures that the highest priority child node with a true condition decorator runs [26]. This is similar to the behaviour of a selector node, as it returns success as soon as a single child node does.

Building on from this, the NPC needed a means to detect the player character. Following [18, pp. 101], a `Sight` script (Fig. 6.6) was made by adapting the 3D `Sight` implementation into a 2D context. Given a field of view and view distance value, the script allows the NPC to detect the player using raycasting methods. This allows the script to determine if a collider

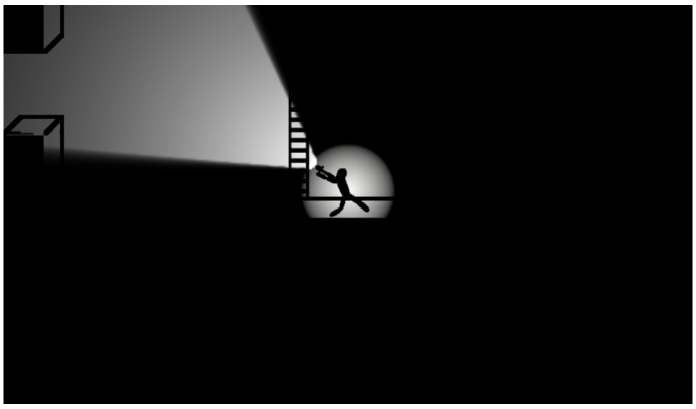


Figure 6.5: Player orients arms and flashlight to shine light the top left.

is in-between the NPC and the player and effectively simulates the sight of the NPC. To incorporate this script into the behaviour tree, the script's variable determining if the player is spotted is simply accessed through the blackboard.

```
    void DetectAspect()
    {
        rayDirection = (playerTrans.position - transform.position).normalized; // Ray is shooting from enemy to player dir

        if ((Vector2.Angle(rayDirection, transform.right)) < FieldOfView)
        {
            RaycastHit2D hit;
            if(Physics2D.Raycast(transform.position, rayDirection, ViewDistance, layermask, -10f, 10f))
            {
                hit = Physics2D.Raycast(transform.position, rayDirection, ViewDistance, layermask, -10f, 10f);
                //Debug.Log("Hitting something..." + hit.transform.gameObject.name);

                if (hit.transform.gameObject.tag == "Player")
                {
                    isPlayerSpotted = true;
                    Debug.Log("Player sighted.");
                    return;
                }
                isPlayerSpotted = false;
                Debug.Log("Player not sighted.");
            }
        }
    }
```

Figure 6.6: `Sight`'s `DetectAspect()` function, used to determine if the player is in sight given the view range and field of view.

Next, this condition is combined with the pre-existing A* pathfinding implemented into the behaviour tree. Together, they form a behaviour where if the player is in active sight, the monster will chase them. Otherwise, it will continue wandering around the level (Fig. 6.7).

6.3.2 Animation

The model is split into five categories of limbs: the arms, legs, head, body and tentacles. The head and arm solver targets are specifically used to orient the NPC according to its direction of motion. This motion is determined by the active path the NPC is undertaking in the behaviour tree. This gives weight to the NPC's movements and allow them to more naturally traverse the environment.

The arms and legs, on the other hand, follow the same raycasting procedures as the player.

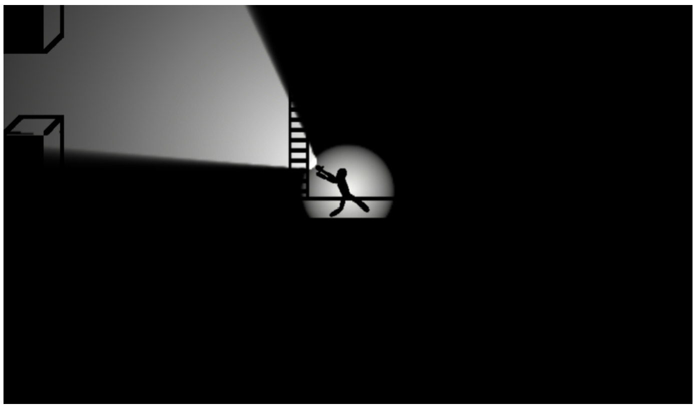


Figure 6.7: The final behaviour tree.

This allows them to grab onto the environment as it traverses the level.

Taking inspiration from Rain World's "Daddy Long Legs", the tentacles of the NPC randomly find an obstacle in the area. If the location is an obstacle and is closer to the current goal than the current target is, it adopts the position as the `nextPosition`. This results in a motion in which shows the NPC dragging itself in the direction of the path.

Chapter 7

Retrospective

This section reflects on the overall completion of the project objectives, addressing problems that plagued development and how they could have been avoided in retrospect. Lastly, this section details the future work possible from the foundations of this project.

7.1 Contribution

The main contribution of this project is the procedural animation library. Despite not being incredibly complex, it provides a unique approach to breaking down and implementing procedural animations. Its approach to animation implementation makes it especially helpful to beginners as it structures the development of such animations into modular components.

7.2 Time Management

Unfortunately, given time constraints the sample game was not completed. Moreover, due to the changing requirements of the project, elements from the original planned were scrapped. For example, sound sensors for the NPC and he macro AI element.

Despite this being due to time constraints, this was also due to a lack of project focus. The project aims to incorporate and combine elements of AI, procedural animation and game development. Regrettably, this lack of focus meant the time spent those elements were thin. Therefore, not as much significant progress was made.

Bibliography

[1] StudioBinder, “What are keyframes in animation?.” https://www.studiobinder.com/blog/what-are-keyframes-in-animation/, 2021. [Online; accessed 2-May-2023].

[2] Unity, “Introduction to sprite animations.” https://learn.unity.com/tutorial/introduction-to-sprite-animations#, 2021. [Online; accessed 2-May-2023].

[3] R. C. Games, “Detention.” http://redcandlegames.com/presskit/sheet.php?p=detention, 2017. [Online; accessed 2-May-2023].

[4] A. Zucconi, “An introduction to procedural animations.” https://www.alanzucconi.com/2017/04/17/procedural-animations/, 2017. [Online; accessed 2-May-2023].

[5] SpookyFairy, “Carrion game review: The horror of being the monster.” https://spookyfairy.com/carrion-game-review/, 2020. [Online; accessed 2-May-2023].

[6] B. Games, “Rain world review.” http://www.brashgames.co.uk/2018/12/30/rain-world-review/, 2018. [Online; accessed 2-May-2023].

[7] J. Jakobsson, “Animation bootcamp: Rain world animation.” https://www.gdcvault.com/play/1023475/Animation-Bootcamp-Rainworld-Animation, 2017. [Online; accessed 2-May-2023].

[8] A. C. Cloud, “Discover the basics of rigging.” https://www.adobe.com/uk/creativecloud/animation/discover/rigging.html, 2022. [Online; accessed 2-May-2023].

[9] HardwareHeaven, “How rayman legends is made!.” https://www.youtube.com/watch?v=y-chi097uV4&t=54s, 2013. [Online; accessed 2-May-2023].

[10] O. Franzke, “Broken age’s approach to scalability.” https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdceurope2013/Presentations/824480OliverFranzke.pdf, 2013. [Online; accessed 2-May-2023].

[11] U. Technologies, “Collider2d class.” https://docs.unity3d.com/Manual/Collider2D.html, 2022. [Online; accessed 2-May-2023].

[12] U. Technologies, “Hingejoint2d class.” https://docs.unity3d.com/Manual/class-HingeJoint2D.html, 2022. [Online; accessed 2-May-2023].

[13] U. Technologies, “Rigidbody2d class.” https://docs.unity3d.com/Manual/class-Rigidbody2D.html, 2022. [Online; accessed 2-May-2023].

[14] TinyMCE, “Modular programming principles: The benefits of a modular codebase.” https://www.tiny.cloud/blog/modular-programming-principle/, 2021. [Online; accessed 2-May-2023].

[15] U. Technologies, “Getting started with unity’s 2d animation package.” https://blog.unity.com/technology/getting-started-with-unitys-2d-animation-package, 2018. [Online; accessed 2-May-2023].

[16] T. Stephenson, “2d procedural animation in unity.” https://www.tomstephensondeveloper.co.uk/post/2d-procedural-animation-in-unity, 2019. [Online; accessed 2-May-2023].

[17] U. Technologies, “Tilemap class.” https://docs.unity3d.com/Manual/class-Tilemap.html, 2022. [Online; accessed 2-May-2023].

[18] D. Aversa, *Unity Artificial Intelligence Programming.* Packt Publishing, fifth edition ed., 2022.

[19] U. Technologies, “Physics2d.overlapcircle method.” https://docs.unity3d.com/ScriptReference/Physics2D.OverlapCircle.html, 2022. [Online; accessed 2-May-2023].

[20] R. Moss, “The perfect organism: The ai of alien: Isolation.” https://www.gamedeveloper.com/design/the-perfect-organism-the-ai-of-alien-isolation, 2014. [Online; accessed 2-May-2023].

[21] OpenCage, “Cathode enums.” https://opencage.co.uk/docs/cathode-enums/index.html, 2019. [Online; accessed 2-May-2023].

[22] AI and Games, “Behaviour trees: The cornerstone of modern game ai — ai 101.” https://www.youtube.com/watch?v=6VBCXvfNlCM&t=3s, 2019. [Online; accessed 2-May-2023].

[23] QualGame, “Behavior trees.” http://qualgame.com/concept/bt/, 2021. [Online; accessed 2-May-2023].

[24] C. Simpson, “Behavior trees for ai: How they work.” https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work, 2017. [Online; accessed 2-May-2023].

[25] U. Technologies, “MonoBehaviour.Update.” https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html, 2021. [Online; accessed 2-May-2023].

[26] P. Games, “Programming overview.” https://bb.padaonegames.com/doku.php?id=quick:program, 2020. [Online; accessed 2-May-2023].

[27] U. Technologies, “Transform.” https://docs.unity3d.com/ScriptReference/Transform.html, 2021. [Online; accessed 2-May-2023].

[28] K. Tremblay, “Evoking emotions and achieving success.” https://www.gdcvault.com/play/1014889/Evoking-Emotions-and-Achieving-Success, 2015. [Online; accessed 2-May-2023].

[29] JustAskQuestions, “Updated tilemap shadow composite.” https://www.youtube.com/watch?v=OPH7ggwPP5M, 2021. [Online; accessed 2-May-2023].

Appendix A - Progress Report