UNIVERSITY OF TARTU

Institute of Computer Science

Computer Science Curriculum

Aleksei Beljajev

Dwarf Block Game Development - Dwarf Simulation

Bachelor's Thesis (9 ECTS)

Supervisor: Jaanus Jaggo, MSc

Tartu 2020

Dwarf Block Game Development - Dwarf Simulation

Abstract:

The thesis describes the process of creating non-player controlled characters for video game Dwarf Block. An overview of different approaches for implementing an artificial intelligence (AI) in games is given. In order to figure out what features are important for creating a good AI, various tests were conducted and results analyzed.

Keywords:

Computer game, game design, software development, playtesting

CERCS: P170 Computer science, numerical analysis, systems, control

Dwarf Block Mängu Arendus - Tegelaste Simulatsioon

Lühikokkuvõte:

Käesolev töö kirjeldab mängija poolt kontrollimata karakterite loomise protsessi arvutimängule Dwarf Block. Töös antakse ülevaade erinevatest lähenemistest, kuidas mängudele tehisintellekti luua. Selleks, et tuvastada, millised featuurid on hea tehisintellekti jaoks sobivad, viidi läbi kasutajate uuring ning analüüsiti tulemusi.

Võtmesõnad:

Arvutimäng, mängu disain, tarkvaraarendus, mängu testimine

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine
(automaatjuhtimisteooria)

Contents

1. Introduction

Dwarf Block is a first-person multiplayer co-op game in development. The goal of the game is to gather resources while defending from the enemies. To achieve this goal the player is helped by other non-player controlled dwarfs who are the main focus of this thesis. Different methods of simulating their behaviour are explored throughout the thesis. Both theoretical and practical techniques of creating interesting artificial intelligence (AI) are investigated. As a result of this thesis, a number of behavioural patterns are implemented and tested.

The goal of the thesis is to describe and implement a system where entities with different behavioral patterns coexist and interact with each other. Furthermore, another goal is to evaluate what features are important for creating a non player-character, whose behaviour and intentions would be clear to the player. Evaluation is done through conducting a series of experiments, during which players give feedback about the quality and characteristics of the NPC.

Non-player characters in this game must be able to interact with each other, inanimate objects and players. Behavioral patterns of aforementioned entities include:

1. Response to environmental changes. For example, returning to the original position, if followed target gets too far away.
2. Response to actions or states of other NPC's. Fleeing from an overwhelming enemy would be an example of such response.

The result must meet the following non-functional requirements:

1. The system must be modular. Meaning that changing or removing a system's parts must not affect other modules' work. For example, removing the mechanic of following the player must not change how resources are gathered.
2. The system must be scalable. This means that the existing behaviour patterns should not be affected by adding new behaviours. This is especially important because the development of the simulation might be continued by other people in the future.
3. Behaviour of created non-player characters should be understandable to players. This means they have to give at least 3 out of 6 in all categories during the user testing.

The final result of the work can be beneficial to people who are trying to create a game and want to add NPC's. However, work can be of interest to people who want to know more about how in-game characters' behaviour is programmed.

2. Artificial intelligence in games

The main objective of the games is entertainment. The main task of all in-game systems, including the NPC, is to achieve this objective. During the development process game developers can forget the goal and spend time tweaking different settings of the AI which does not make the game better. This may lead to the situation when computer-controlled characters are very realistic or particularly intelligent, but the final result remains the same – they are boring from the player's point of view. In a shooter game NPC's could constantly attack the player without giving him a break or even have a perfect aim. The player in this case would feel that the game is not fair, because it is too hard or impossible to beat.

One of the first games to add "intelligent" characters was Pac-Man[1], which featured enemies with a multitude of behavioral patterns that arguably made the game so popular as it was. However, there are many other reasons to add an AI to a game:

- AI can be added to multiplayer games to replace human opponents. In such cases, actors would generally try to behave like human players would.
- AI actors can control enemies in single-player games to challenge the player. In this case AI can also be used to control the difficulty of the game.
- Actors can also be allies. They help the player and fulfill their respective roles that were assigned by the player or developers. In action games they can defend the player or help in some other way.
- Artificially controlled entities also add a sense of "life" to games. In open-world games, there are usually settlements filled with residents. Player's immersion would immediately break, if residents would stand and do nothing.
- If characters behave in a way that resonates with a player, they can make the player feel certain emotions and be attached to them.

Some creators of computer games think that intelligence does not have to be simulated, but it is enough to have an illusion of simulation (Rabin, 2018). The methods described in the following sections can be used for this purpose.

[1] Pac-Man https://en.wikipedia.org/wiki/Pac-Man.

2.1 Constructing the illusion

Steve Rabin thinks there are three aspects that make illusion believable for the player. First, the players want to believe that computer-controlled actors' thought processes are similar to human ones. Secondly, the people usually try to see human qualities, such as emotions or desires, in computer-controlled figures, even if they do not have them. For instance, in the video game Dark Souls the player faces an opponent, which looks like a giant wolf. When his health points fall below a certain threshold, he does not attack as rapidly as before and the animations change to look slower and as if he is tired. The player might start feeling guilty or sad because of that, but in reality it is just a combination of the animations and behavioural patterns that do not have anything to do with an actual fatigue. And thirdly, players' expectations affect how they perceive the game world and AI characters. This section is focused on these three aspects.

2.1.1 Players want to believe

To investigate an AI in games, a group of experienced game developers conducted an experiment (Wetzel and Baylor, 2017). For the experiment, they created a turn-based strategy game where the player competes against artificial intelligence (AI). Participants played against the computer multiple times and each time the computer used different strategies. People did not know what strategies were used and after each round have assessed how closely AI behaviour resembles human behaviour. Some of the predefined strategies were complex and tried to simulate human behaviour purposefully, but in the end it did not help to convince the players. The results have shown that a strategy of attacking an enemy with the most amount of health was the most realistic.

Players do not know why NPC's take one action or another. Therefore, players themselves deduct how computer-controlled characters come to their conclusions and decisions. This phenomenon allows game creators to stimulate players' imagination and let them fill in the blanks themselves rather than simulate complex behaviours.

2.1.2 Anthropomorphization

Anthropomorphization – a tendency to attribute personal properties to inanimate objects. Anthropomorphization occurs when people see things that resemble human qualities, such as desires or emotions. One of the theories about this is that when people try to understand incomprehensible behaviour, they often apply familiar human behavioural patterns to understand the situation (Waytz, 2010). So if a human-like character in the game does something, the player tries to interpret such behavior with an idea that the character is a person. Thus, if a character in the game looks like a person, it is also easier to make players believe that he behaves like one.

2.1.3 Expectations

Expectations partly control how people perceive the world[2] (Biello, 2019). Players' expectations affect the perception of the game, including the perceptions of the computer-controlled characters. Game developers can use this phenomenon in advertisements by showing certain play-styles or advanced in-game equipment. This way, players do not get into a completely unknown world. They already know the basic mechanics and can orient themselves more easily.

2.2 Applications of illusions

There are different practical ways to make the player believe that an AI in the game is smarter than he really is. This section describes how real world game developers create the illusion of intelligence.

Different approaches are used for different types of games. During the development of the game Halo[3] playtests were carried out. First, players fought against weak enemies with low health points and damage. Only 8% of the participants found the AI "Very intelligent" and 20% thought that the AI is "Not intelligent". Next, the developers increased the amount of health points and damage. People played the same level against the same AI, but enemies

[2] Expectations Influence Sense of Taste - Scientific American https://www.scientificamerican.com/article/expectations-influence-se/.
[3] Halo https://en.wikipedia.org/wiki/Halo_(franchise).

were tougher. This time 43% of the players found the AI “Very intelligent” and nobody marked it as “Not intelligent”.

Making an AI look smarter can be as easy as increasing a couple of variables, but this approach is not universal. There are stealth games like Thief[4] that would not benefit from increasing the amount of health of the enemies because the game genre implies that head-on confrontation would result in losing anyway. One way of making an AI look smart is to telegraph its thoughts. Enemies in games of the Metal Gear[5] series inform the player about their observations by saying them out loud. For instance, they would make a comment if they find an opened door that was previously closed. Even if it does not lead to serious actions like informing other actors, it still has an impact on the player. Same principle can be applied to action games. NPC´s would start to spread if they see a grenade near them and the player would see it as rule if it would be accompanied with shouting something suitable.

It is important to inform the player not only about the observations, but also about intentions, even if they are not real. For example, in shooter games enemies might shout things like “Cover me!” or “Surround him!”, which does not affect the behaviour of their allies. The player, however, might be influenced by these words and start to see the consequences - enemies aim more accurately or change their position, while in reality nothing changed or changed due to other factors.

[4] Thief https://en.wikipedia.org/wiki/Thief_(series).
[5] Metal Gear https://en.wikipedia.org/wiki/Metal_Gear.

3. AI implementations

Game AI is a system that is responsible for decision making of in-game entities. Controlled entity is usually referred to as an agent. Traditionally, agents are characters in the game, but the term is not limited to living creatures. Anything that needs to change behaviour based on current conditions can be an agent: a robot, a vehicle or something more abstract such as country, civilization or a planet.

AI behaviour can be divided into 3 parts (Sizer, B):

1. Perception - the first stage when an actor collects data about the environment. An agent needs to somehow recognise that relevant changes in the environment have occured. In the case of "living" creatures, vision or hearing programmatic counterparts can be used to imitate the sense of surroundings. Due to the fact that games are closed controlled simulations, agents do not need complex algorithms for spotting changes - they can just ask the game. For example, in an action game, NPC's do not need to guess the remaining amount of health of an opponent based on visual aspects such as ripped clothes or facial expression, because the game knows exactly how much health every entity has.
2. Assessment - phase when an optimal way of acting is calculated based on perceived conditions. For example, an agent whose role is mining might consider if it is more optimal to go mining or pick up a resource that was already extracted.
3. Action - stage when an agent executes the action according to the decision in the previous stage.

It may be as simple as to only decide in which direction to move a platform like in game Pong or as complex as controlling different aspects of a whole civilization as in the computer game Civilization. Depending on the complexity of the desired AI behaviour, different approaches are chosen. Continuing the previous example of Pong game, a number of conditional statements would be sufficient to implement a computer controlled character: move up if the ball is higher and move down if the ball is lower.

In more advanced cases, where characters have to consider a lot of changing conditions and have multiple ways to act, simple conditions would not be an effective way of implementing an AI. For example, in the digital card game Hearthstone there are too many variables that

need to be considered to simply put all of them into conditional statements. Each player can have up to ten cards in his hand and on top of that there can be a maximum of fourteen cards on the board. Cards can be used on different targets in different order, which also complicates things. Furthemore, there are 3197 playable cards in the game and this number constantly grows. Writing conditional statements for this number of cards that can be used on up to fourteen cards in unpredictable game states is impossible, therefore some other approach must be used.

There are multiple ways to implement an AI system. Every one of them has its own benefits and drawbacks, so careful consideration must be made before choosing one or another. In this section the most used ways of implementing AI systems are described.

3.1. State machine

Computers we know today were built on the foundation of mathematical models (ref Turing machine). State machine pattern is also connected to previously developed mathematical abstraction - finite-state automata. Finite state machine is a mathematical model of computation, which can be in exactly one state at any given moment. In response to input it can change the state through transitions.

In programming, State pattern is a software design pattern that allows an object to alter its behaviour when its internal state changes. This is one of the first methods for implementing an AI (Gaudl, S).

Figure 1 illustrates a behaviour of a guard implemented as a state machine.

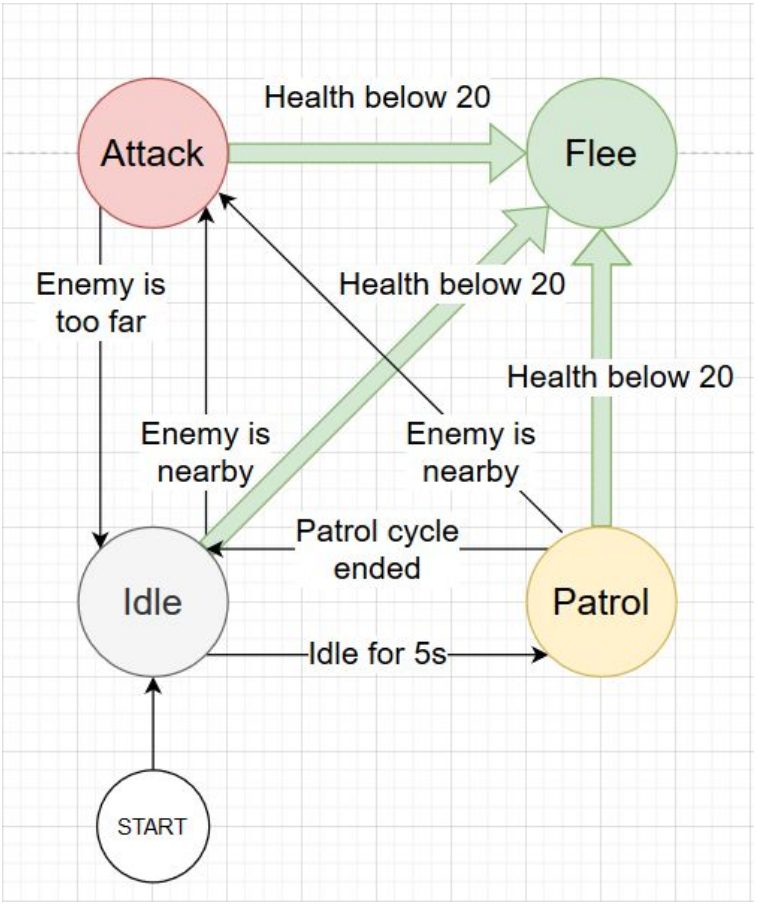

Figure 1. State machine example

There are four states the guard can be in: Idle, Attack, Flee and Patrol. The default state is Idle state, which is indicated by a transition from a special node Start. Arrows represent transitions. For example, Idle has three outgoing transitions and receives two transitions (not counting Start node). Conditions for transitioning to other states are written on the arrows.

One of the benefits of choosing state machines as an implementation of AI is that state machines are easily representable visually and even non-programmers can understand the behaviour. Also, it is easy to control the behaviour by adding or removing transitions.

One of the biggest drawbacks of this method is that transitions might be duplicated multiple times if a state has incoming transitions with the same condition. In figure 1 Flee is the state that has three incoming transitions with the same condition - health is below twenty. The transitions come from three different states: Attack, Idle and Patrol. It means that a controlled character would start fleeing if his health drops below twenty no matter what state he is in. It does not pose an issue now, but it certainly would in the future if the number of states grows. If the intention of fleeing from any state when health is below twenty remains, developers

would have to manually add a corresponding transition every time a new state is created. Furthermore, since the Flee state has incoming transitions from all other states, changing the condition of starting to flee would require modifying all of them one by one. It is time-consuming, not efficient and therefore not suitable for the rapidly changing game development industry.

As a result in the aforementioned features, state machines are more suitable for defining simpler behaviours and more complex ones require different approaches.

3.2. Behaviour tree

Behaviour trees lessen the amount of duplication and make it possible to add transition rules, without adding transitions to every state they apply to.

Just as state machines, behaviour trees are built on the mathematical foundation. Behaviour tree is game development equivalent of arborescence[6] from graph theory[7]. Instead of vertices and edges, behaviour trees have nodes and transitions.

Root is the starting point of the behaviour tree execution. With a certain frequency it sends signals called ticks to its child node, which in turn forwards them to its own children. Nodes that receive the signal must return one of the three responses: “success” if the execution was finished, “fail” if the execution was interrupted or could not be finished and “running” if the execution is not yet finished.

There are two types of nodes: tasks and control flow nodes. In figures 2 and 3 they are marked with green and blue colors respectively. Tasks are terminal nodes, that is, nodes without child nodes. They are a visual representation of a function which gets executed when the task node receives a tick. Control flow nodes are non-terminal nodes that receive ticks and decide which of its child nodes receives them, hence the name. The status they return to their own parent node depends on the status of the child nodes.

A control flow node may be either a selector or sequence. They start sending ticks from the leftmost child and move on to the next child based on returned status.

Selector nodes are used to find and execute the first child that does not fail. It stops the execution with successful status when its child returns “success”. Such case is illustrated in

[6] Arborescence https://en.wikipedia.org/wiki/Arborescence_(graph_theory).
[7] Graph theory https://en.wikipedia.org/wiki/Graph_theory.

figure 2 in subtrees A and B, where successful nodes are marked with a green color, nodes that failed with a red color and grey shows nodes that were not executed. In the A subtree the execution stops right after the first task, because it returns true. Selector from subtree B finishes with a successful status after the execution of the second task, because the first task failed. Selector node fails only if all of its children fail, which is demonstrated in the subtree C in figure 2. All three tasks have failed, therefore the selector also fails.

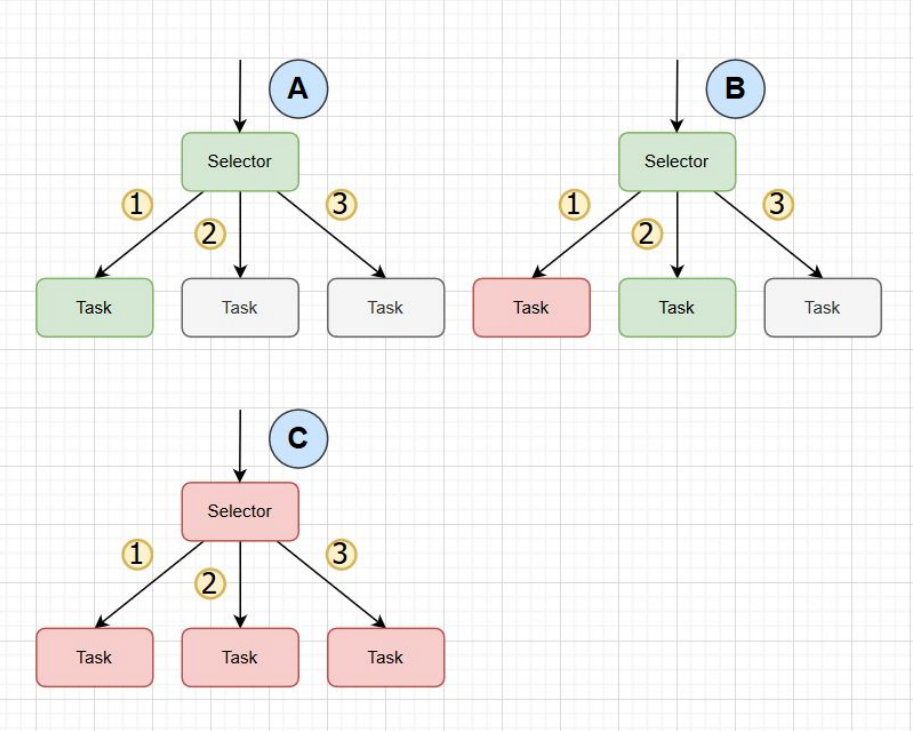

Figure 2. Selector node states

Sequences send ticks to the same child as long as it returns “running” status. Once it returns “success”, the sequence starts to send ticks to the next child. This loop continues until the rightmost child node returns “success”, in which case the sequence also finishes the execution with a successful status code. The execution may stop prematurely if one of the child nodes fails, in which case the sequence node also fails. Subtree A in figure 3 demonstrates the state of the sequence node and its children after executing the first task. The first task was successful, but unlike selectors, the execution continues. One possible outcome of executing the sequence is shown in subtree B in figure 3. All three tasks were successful and therefore the sequence is also successful. And another outcome is illustrated in subtree C in figure 3, where sequence returns “fail” status because the second task has failed.

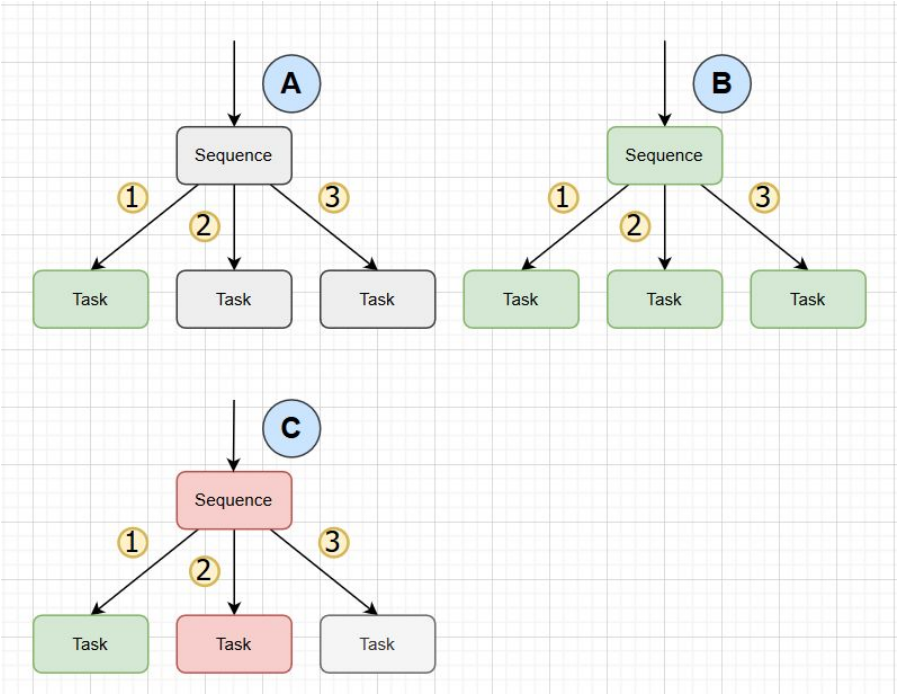

Figure 3. Sequence node states.

Figure 4 shows the behaviour tree equivalent of the state machine illustrated in Figure 1. Control flow nodes are highlighted by blue color and tasks by green color. The numbers in yellow circles show the order of execution of control flow nodes. The execution starts in the root node and after that order number in the following list corresponds to the execution order of the behaviour tree in figure 4.

1. Selector directs the execution to the left sequence. It will try to execute the selector on the right only if the sequence fails.
2. This sequence is responsible for fleeing in case health is too low. This is the left most control flow node after the one connected to the root node, which means it is executed before the other ones. In that case, actions that should be considered in the first place are located here.
    2.1. Task that returns successful status if actor's health is below twenty and would fail otherwise. When failed, the sequence is stopped and the next task responsible for fleeing is not executed.
    2.2. Task of fleeing. Returns "success" if nothing blocks the path and fails if cannot flee for some reason.

3. Selector would try to execute the sequence on the left and move on to the sequence on the right, if the first one fails.
4. This sequence is responsible for attacking an enemy if there is one nearby.
    4.1. The first task checks if there are enemies nearby. It would fail if there are none, therefore the sequence would stop and an actor would not even try to attack.
    4.2. This task is executed after confirming there are enemies nearby, so it would succeed if damage is dealt to them.
5. The last sequence is responsible for patrolling and would be executed only if all other sequences failed.
    5.1. Task that serves as a timer. Succeeds when an appropriate amount of time has passed.
    5.2. The last task is responsible for patrolling.

By grouping the tasks and control flow nodes this way, duplication is avoided. We can add any number of nodes to the right side of the first selector and not worry about the fleeing logic, since it is placed on the left from the first selector and therefore gets executed before anything else.

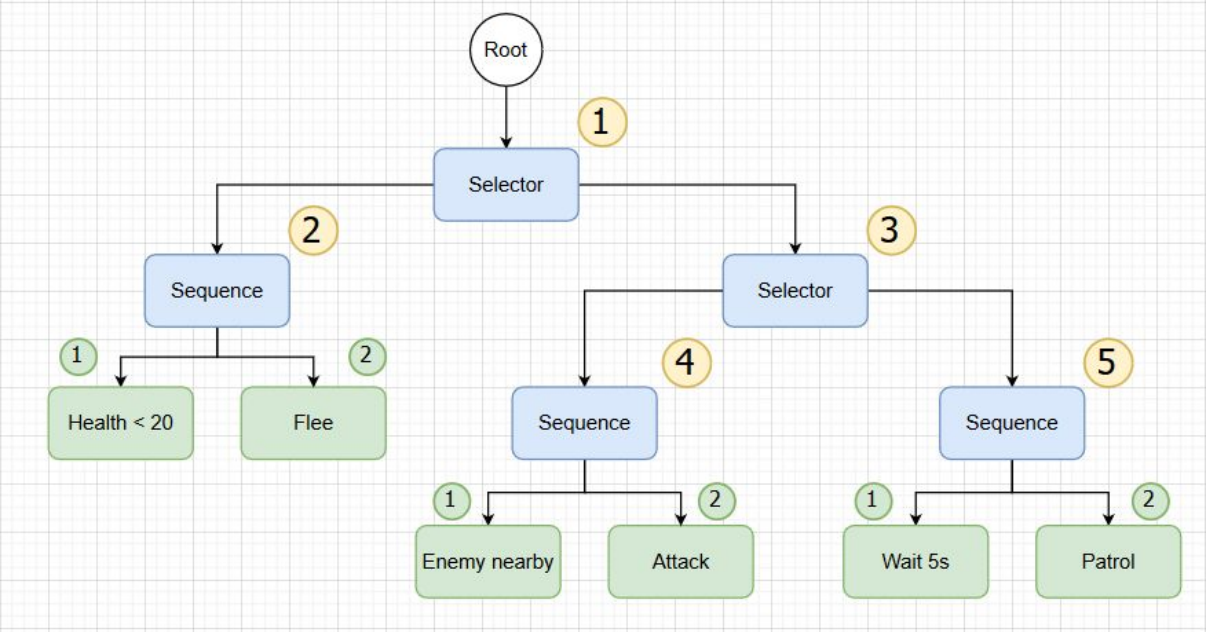

Figure 4. Behaviour tree example

3.3. Utility-based system

In decision-making systems like state machines or behavior trees transition to another state happens when current action is finished or some condition is met. The number of transitions is predetermined by the state machine or behaviour tree configuration. In a utility-based system, on the other hand, transitions are not determined by developers, but created at runtime, while developers only define states.

In a utility-based system AI makes a decision based on the relative value of taking one or another action. Developers only define the options an AI can choose from and how the values are calculated for each of them.

The core idea of the utility-based system is making decisions about transitioning to another state based on calculated relative value of taking available actions (Dill, K). Value for every action is calculated based on relevant data. For example, assume an AI actor whose role is a medic can perform two actions: get to cover or heal an ally. To make a decision, values of taking those actions need to be calculated. The utility of healing an ally can be based on his current health and the medic's own health can affect the utility of getting to cover. How exactly those variables are mapped to utility scores is up to developers.

In this case, figures 5 and 6 represent functions that calculate scores for healing an ally and getting to cover. The importance of healing an ally grows when his health decreases and is equal to zero when fully healed (Figure 5). It is reasonable, because there is no need to heal an ally if he is not injured. The importance of healing is capped at the value of 0.8 so medics would not prioritize healing badly injured allies above all else.

Figure 6 illustrates that the importance of getting to cover decreases when medic’s health grows. The idea behind such function is to make medics heal others only when they are safe themselves.

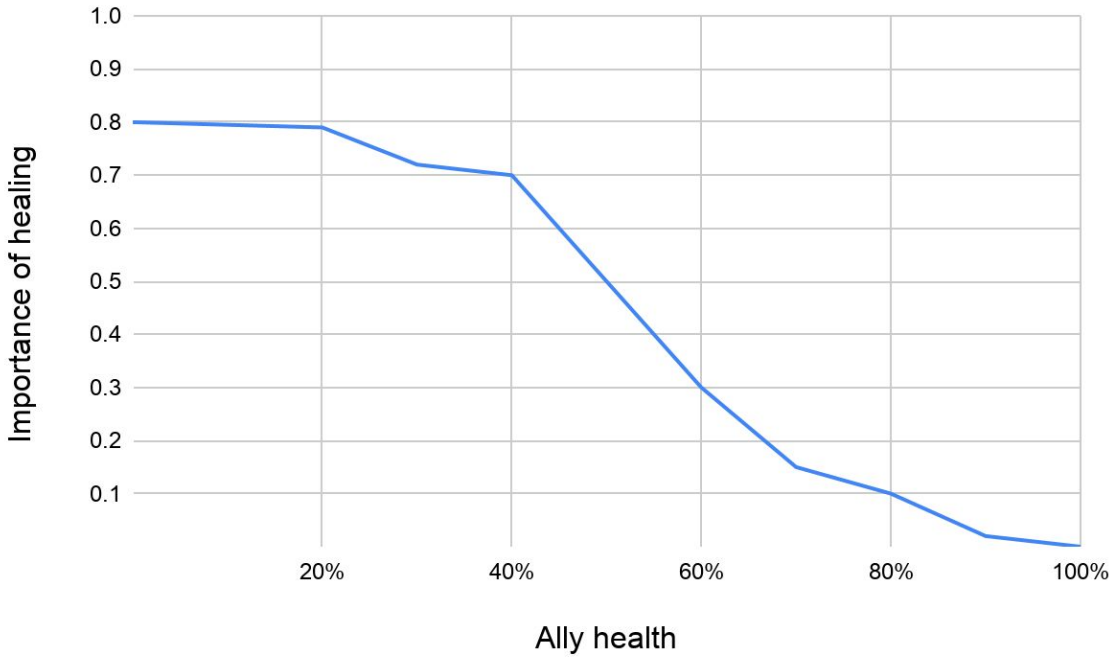

Figure 5. A utility function calculating the importance of healing an ally based on percent of his current health.

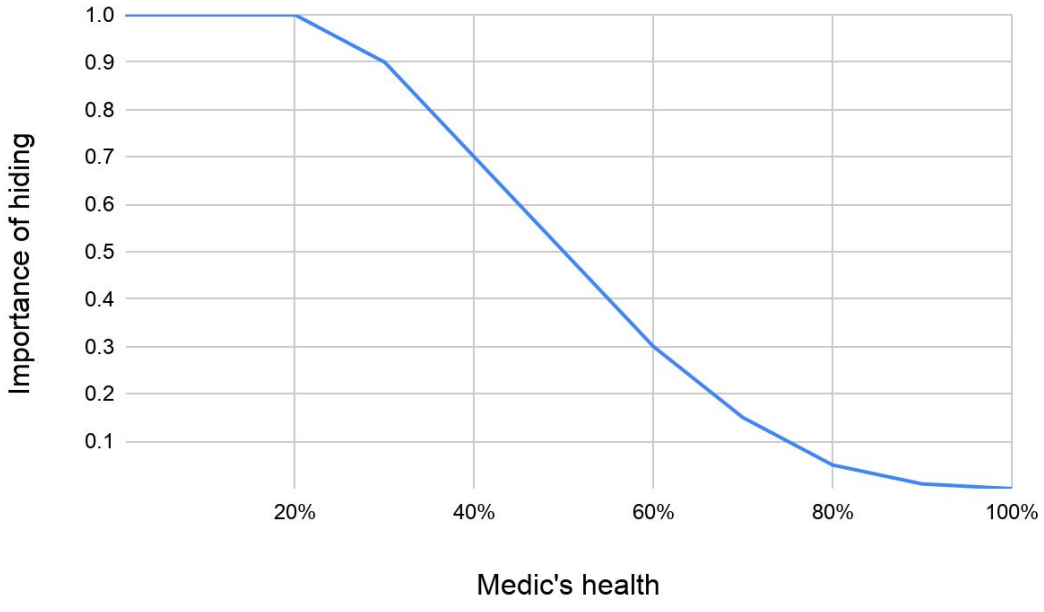

Figure 6. A utility function calculating the importance of hiding based on percent of self current health.

Now, when utility functions are ready, making a decision is trivial. Assuming medic has 60% of health and an ally has 40% of health, we can calculate the utility scores using functions from figures 5 and 6 respectively. The value of getting to cover is 0.3, while the value of healing an ally is 0.7. Second value is higher, therefore medic would decide to heal an ally.
This system is very versatile and powerful because behaviour can be changed at runtime by adjusting the curves. It comes with a cost of time spent configuring the curves to make actor behave in a desired way

3.4. Comparison

Dwarf Block is a relatively complex game so simple conditional statements are not an option for implementing an AI. State machine is a powerful tool, but in the given game there are too many entities which would result in duplication. A utility based system could be a good fit, but it is not practical to implement such a system from scratch, because there is a built-in behaviour tree system in Unreal Engine 4. Behaviour trees are easy to maintain, modify and scale.

4. Unreal Engine 4

Unreal Engine 4[8] is an advanced open-source 3D creation platform. It is widely used for game development, visualizing simulations, filmmaking and much more. Regarding the game development, Unreal Engine has built-in systems for adding lighting, audio, special effects, physics, animation and other features.

This particular game engine was chosen for developing the game because it is free, has support for multiplayer and powerful tools for creating in-game AI. The following built-in components are used for implementing an AI system:

- Blueprint Visual Scripting
- Behavior trees
- Environment Query System
- AI Perception
- Pawn possession

These components and tools will be described in a greater detail in this section.

4.1. Blueprint Visual Scripting[9]

Scripting is usually associated with writing code in a form of text. At its core, scripting is aimed to automate tasks. The problem is that programming languages have different syntaxes and they are mostly not intuitive. Non-programmers cannot understand the code intuitively and certainly cannot make changes without breaking anything. Despite having different syntaxes, programming languages have common structures like conditional statements or loops. To lower the level of expertise needed for automating tasks, visual scripting was created.

Visual scripting tool in Unreal Engine is called Blueprints Visual Scripting system. Objects that were created using this tool are called Blueprints.

[8] Unreal Engine https://www.unrealengine.com/.
[9] Blueprints https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html

Blueprints system lets developers automate multi-step tasks and implement complex logic using the visual editor. Instead of code blocks, blueprint scripting uses nodes that represent functions, events, variables or other programming concepts. Code flow is controlled by transitions from one node to another.

Figure 7 illustrates an example of a program that can be created using blueprint visual scripting. The execution of this snippet starts from the node number one. This node references a special built-in event that gets invoked when the game starts. White outgoing transition shows where the execution goes next. It transitions to a second node that gets two integers as an input and continues the execution based on the relation between two numbers. Node number three is responsible for generating a random number, in this case in range from zero to ten. It does not have an incoming white transition and gets invoked when its output is requested by the second node. Next, a randomly generated number is compared with a constant five and one of the connected nodes gets executed. If the number is bigger, equal or smaller than five then nodes number four, five or six get invoked respectively and an appropriate message gets printed.

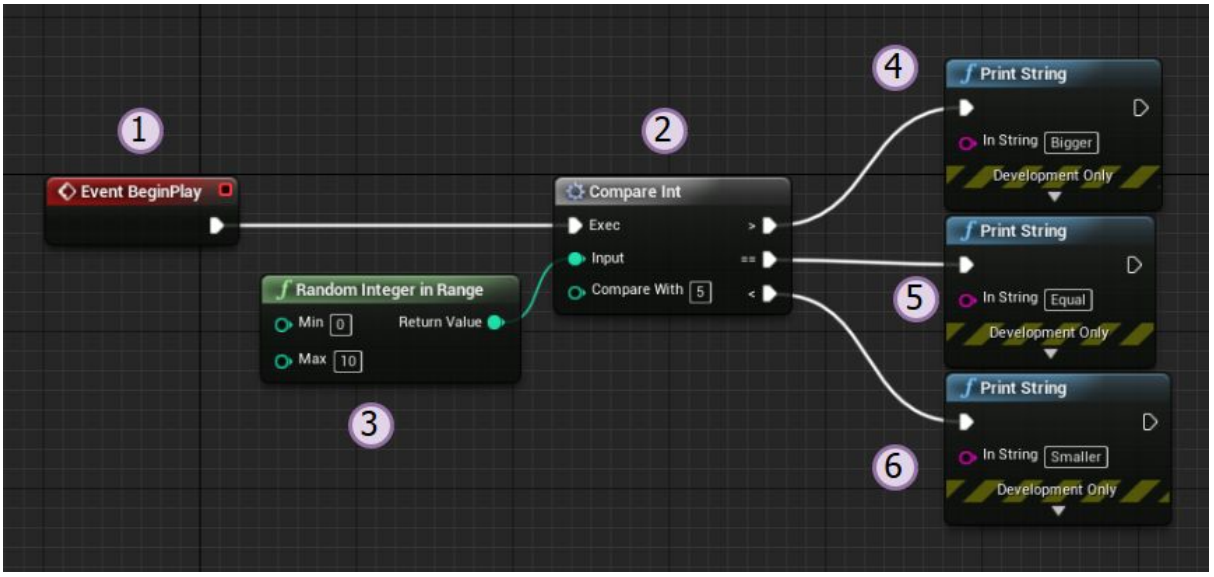

Figure 7. Blueprint scripting example

4.2 Behavior trees[10]

Behavior trees assets can be used to create artificial intelligence for in-game non-player entities. The behavior tree consists of a behavior tree graph and a blackboard. The graph is responsible for handling logic while the blackboard is in charge of keeping track of tree's data.

Behavior tree graph is similar to blueprint system in a sense that it is also a visual editor. Like a normal graph, a tree graph consists of nodes and transitions. The first node is called the root node and execution of the graph starts from there. Execution order in which nodes are executed is set left-to-right.

There are four types of nodes:

1. Composite nodes control the flow of execution. They in turn are divided into three: selector node, sequence node and simple parallel node. Selectors and sequences follow standard rules discussed in Section 3.2. Parallel node has two outgoing transitions: main and secondary. As illustrated in figure 8, the parallel node is labelled with number one, the main branch with the number two and secondary with the number three.. Both main and secondary branches get executed simultaneously while the main branch is active. Parallel node finishes the execution when the node from the main branch either succeeds or fails. It interrupts the execution of the secondary branch and status of the main branch is returned. Figure 8 demonstrates how a parallel node can be used to make an actor run towards a target and at the same time attack the target every second. Main branch succeeds when an actor reaches the target and the secondary branch, namely attacking logic, would be aborted. A parallel node can be configured so that it does not abort the secondary branch, but waits until it finishes. If configured this way, the actor would continue to attack even after reaching the target.
2. Task nodes reference functions they execute. They return one of the three values: *Succeeded* if task was finished, *Failed* if task could not be executed or *In Progress* if task is currently being executed.
3. Decorator nodes are conditional nodes. They can be attached to composite nodes or tasks to check some condition and continue or stop the execution based on the

[10] Behavior Trees https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/index.html

condition. They can also be configured so that execution in child nodes stops if condition changes. Figure 9 illustrates how decorators can be used to simplify the behaviour tree demonstrated in Section 3.2 figure 4. The numbers in circles represent the order of execution. Nodes three and five are tasks with decorators wrapped around them. First decorator in node three is called *IsHealthBelow20?* And it lets execute the underlying node only if condition is met, otherwise it returns *Failed.* Decorator in node number five works the same way, but the condition is different.

4. Service nodes also can be attached to composite nodes or tasks. They are executed as long as the node they attached to or any of its child nodes is being executed. They are executed in parallel with the main branch and can be used for a variety of tasks, for example, to update the destination of an actor so it comes to the right place.

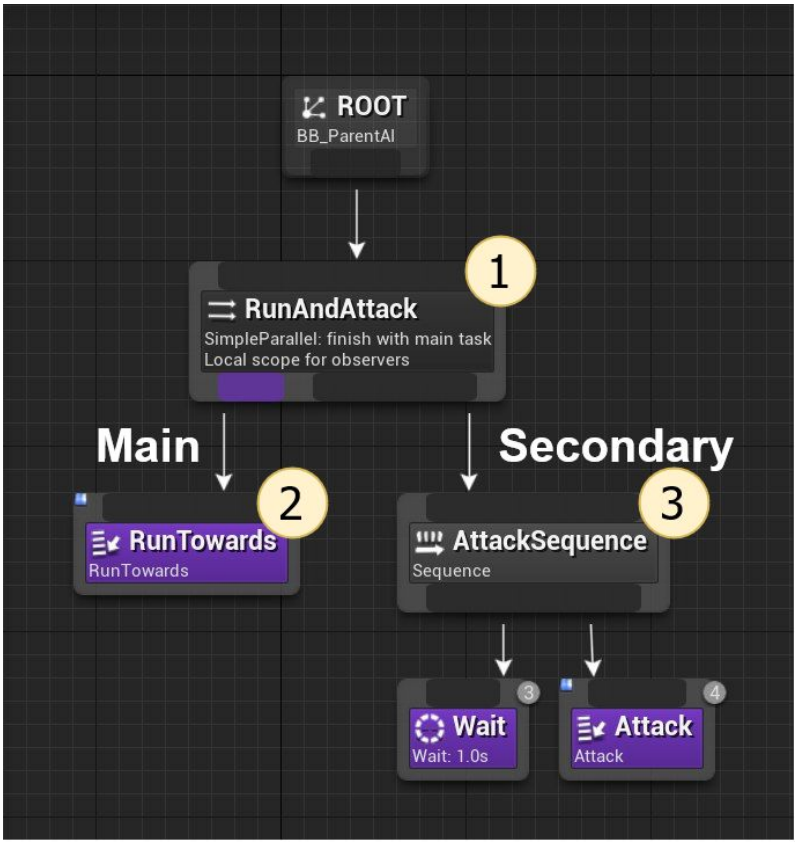

Figure 8. Parallel node example

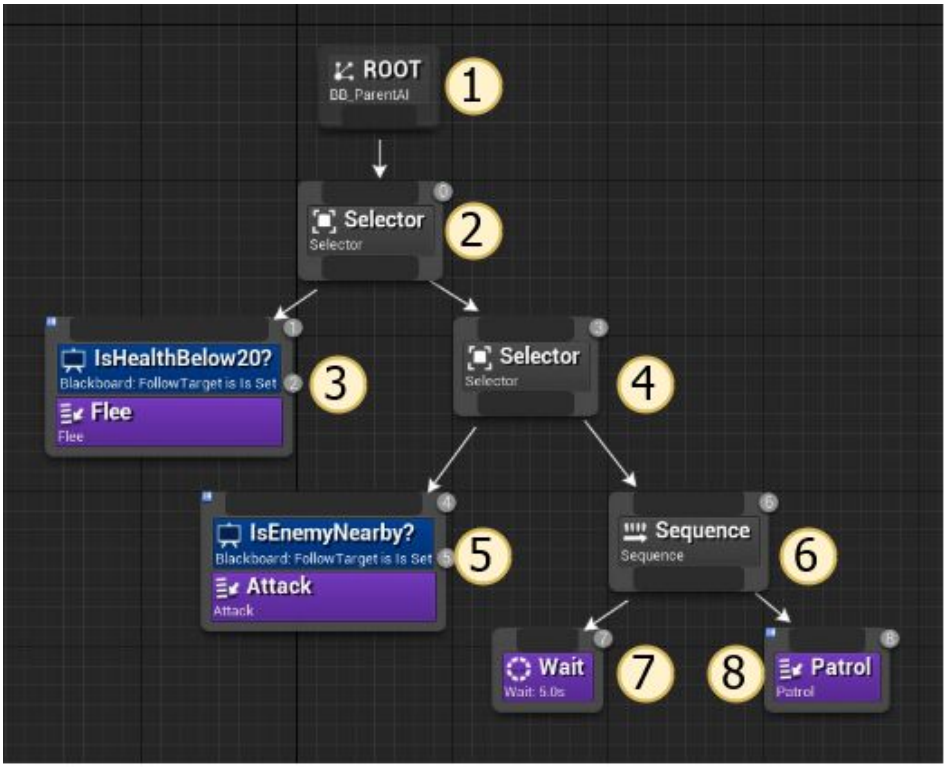

Figure 9. Unreal Engine behavior tree example

4.3 Environment Query System[11]

Oftentimes AI actors need to make a decision based on their surroundings. Actors do not have a direct reference to all objects in the game, so they need a middleware system between them and the rest of the game world that would provide necessary information. Unreal Engine utilizes a tool called Environment Query System (EQS) to address this matter.

AI actors can ask EQS for information about the environment using queries. Query is a number of user-defined tests that return an object or a number of objects that best fit those tests. There are a couple of test templates at developer's disposal, that can be used to query data. Some of the commonly used ones are distance tests that check if an object is in a certain range, visibility test that checks if there is an obstacle between two objects or pathfinding test that checks if a path to a particular point exists.

[11] Environment Query System
https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/EQS/EQSOverview/index.html

4.4. AI Perception[12]

Besides EQS, agents have another tool for retrieving information about surroundings. AI Perception System is a tool that mimics living organisms by sensing the environment. The system combines the equivalents for hearing, vision and touching. Those are implemented using different colliders that detect changes inside themselves and events that are fired when certain changes occur.

4.5. Pawn possession

Objects placed in a game world are called Actors in Unreal Engine. They have position, scale, rotation and other properties. Their another function is replication of named properties across the network. There are a lot of entities and properties in Dwarf Block – from a generated world to dwarfs - and dealing with synchronizing them between different clients would be another huge task.

Actors controlled by a player or AI are called Pawns. Entities under control can be referred to as *Possessed*. Pawns receive input or commands from either PlayerController or AIController classes. No matter what source the commands are coming from, they are executed the same way. It is important because players have the same abilities and are physically identical to AI controlled dwarfs. Designing codebase in such a way lessens duplication, because there is no need to implement logic for executing certain tasks for an AI and player separately.

[12] AI Perception https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/AIPerception/index.html

5. Game implementation

Game cycle includes gathering resources, slaying enemies, building a base and upgrading equipment. One of the key features of the game is an opportunity to control allied characters. This section describes what can characters under control do and technical details about implementation of such a system.

Most of the functionality is implemented using Blueprints, because it takes less time to compile and they are designed to work with behavior trees. Also, combining traditional code and visual scripting increases chances of duplication. It is easy to unintentionally implement the same functionality in both text and visual editors. Furthermore, it is harder to find bugs because places where blueprints reference codebase and vice versa are not apparent.

Figure 10 demonstrates the relationships between main classes that were used during the development. Yellow boxes represent built-in classes, red boxes are custom C++ classes and blue boxes illustrate blueprint classes. Classes on top are parents of the lower ones. *MCharacter* class is responsible for settings that are common for all living entities: instancing camera, configuring meshes, adding a health system component and others. Class *BP_Character* serves as a contact point between *MCharacter* class and blueprints. And finally class *BP_Dwarf* contains information about actual dwarfs. It gets possessed by either a player or an AI. Reactions to the outside world, such as reaction to receiving damage, are implemented in this blueprint. *MPlayerController* and *BP_PlayerController* define what happens when a player possesses a dwarf. And *BP_AIController* controls dwarfs. It is responsible for executing the right behaviours and perceiving the world.

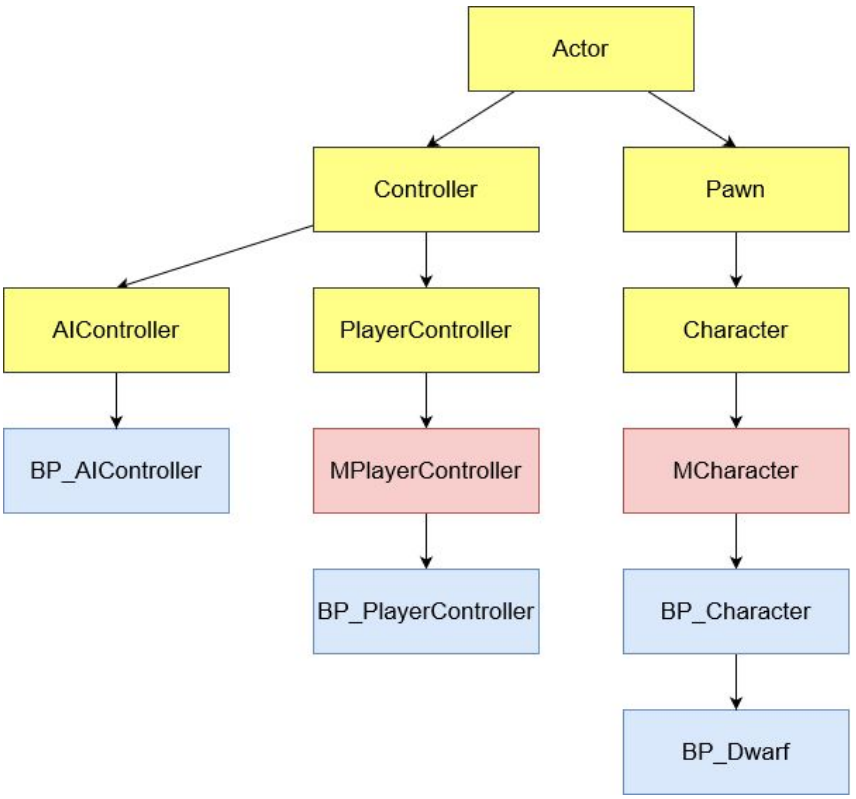

Figure 10. Class diagram

5.1. Pathfinding

Pathfinding algorithms are used for finding the shortest route between two points. It would be a trivial task if there are no obstacles or other constraints, since the shortest route would be a straight line. But in video games algorithms have to consider obstacles blocking the way, environment, slopes and other entities.

Pathfinding system in Unreal Engine is called Navigation. It helps pawns navigate through the environment, avoid obstacles and oriente themselves. By default, pawns cannot navigate in the game world, because surfaces are just models without any information about paths. To define areas where pawns can navigate and move, an actor called NavMeshBounds is used. It can be thought of as an invisible box that determines what parts of the game world are walkable. When added to the world, it analyzes the geometry inside its bounds and generates a simplified version of the specified geometry. The simpler version of original geometry is called NavMesh and is used by pawns for navigating. Figure 11 illustrates how newly

generated NavMesh looks like inside the NavMeshBounds. Green polygon is the NavMesh and pawns can only walk there. This green area is limited by the NavMeshBounds component illustrated as a box with transparent faces. Red boxes are obstacles that pawns would try to avoid when building a path. The purpose of grey areas around them is to account for the pawn's 3D model. The absence of extra space around obstacles would result in pawns getting too close, which in turn would make models of obstacles and pawns overlap. It is impossible to avoid overlapping at all, but minimizing it helps to create an illusion of a real world.

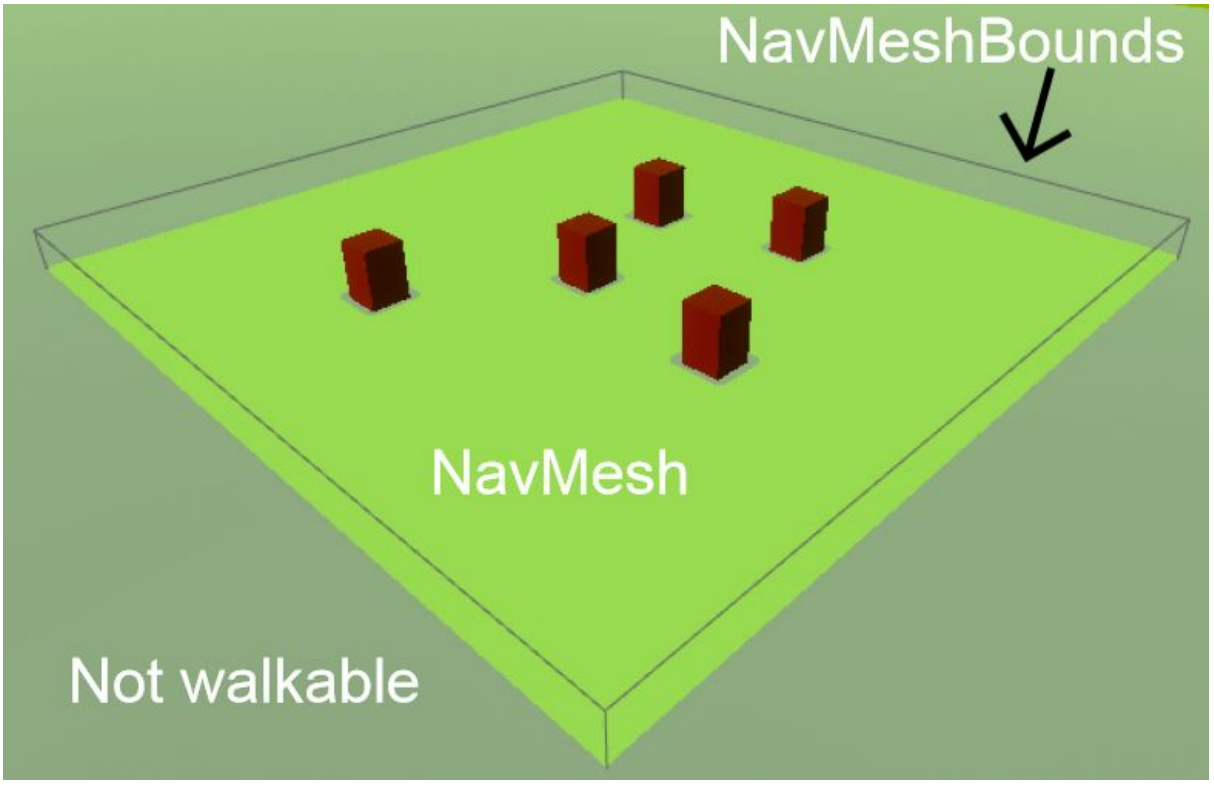

Figure 11. Generated NavMesh.

Environment in games may vary and have different features, so applying the same preconfigured navigation to every situation is not reasonable. Figure 12 shows how lack of configuring leads to undesirable results. The left picture shows a navmesh where pawns can walk around the staircase, but cannot pass the actual steps to climb on top. The right picture illustrates a configured version of the same area which lets pawns climb the stairs. There are around ten settings that define how NavMesh is generated and by adjusting them a broad variety of results can be achieved.

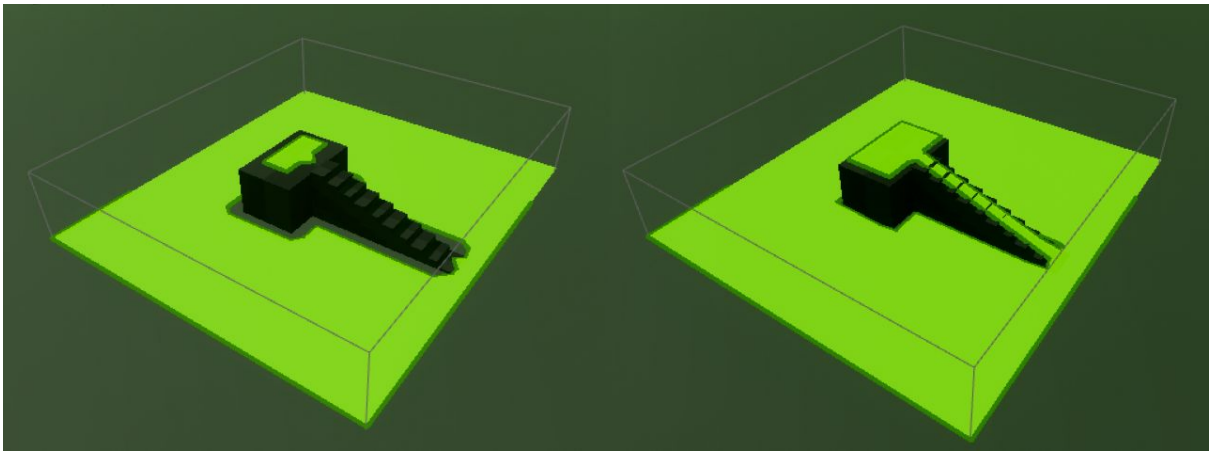

Figure 12. Impassable navigation (left) and configured navigation area (right).

A specific component for creating predetermined routes was created. It was implemented for simplifying the process of setting different paths for actors. It is flexible and allows even non-programmers to define routes that actors may take.

The foundation of such a system are points in space that define points between which characters may move. After reaching such a waypoint, the actor asks it for the next point and moves there. But waypoints that have only one reference to the next point are very restrictive, because all paths are linear and there is no variety. Also, if developers wanted to add crossroads, they would have to at least double the amount of points, because every path would have to have a “in” and “out” point.

In Dwarf Block actors that need to patrol or traverse some path, use a waypoint system with a little twist. Instead of a reference to a single next point, they have an array of such points. Meaning, that if a waypoint has two outgoing points, the actor can go either way randomly. Totally random movement is not always wanted, because it puts certain restrictions on the created paths. To make the system more controllable and add an opportunity to tweak the routes, “weights” were added to each point. Weight is a float number in a zero to one range that represents a probability of choosing the according path. For example, if there are two outgoing points and their weights are 0.3 and 0.7 respectively, then actors would be more likely to choose the second path. It is also possible to change the path dynamically in runtime by adding new or deleting old waypoints and changing weights.

A tool was implemented to help visualize the waypoints and paths. It dynamically shows where the waypoints are, what they are connected to and in what direction the movement is possible.

Waypoints are illustrated on Figure 13. There are five points represented by purple vertical lines. Left bottom point has two outgoing arrows that show the directions of the paths that actors may choose to follow. The left one has a weight of 0.2 meaning, while the right one has 0.8, meaning that on average every fifth pawn that passes this point would choose to go left. Both top points have one outgoing connection each and both weights are equal to one, which means that every pawn will choose the route. Other points are configured in a similar fashion.

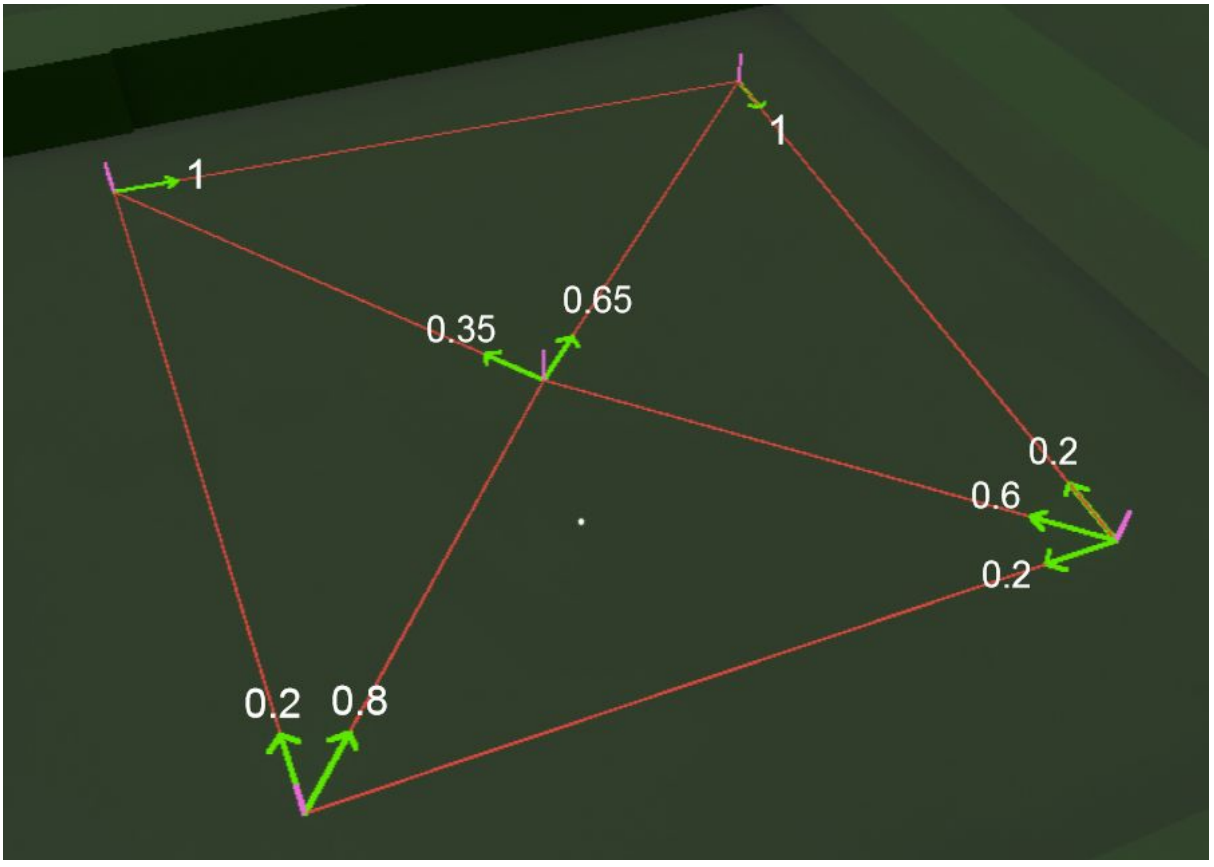

Figure 13. Visualized waypoints and paths.

5.3. Allies

Previously described classes are a foundation for the system the thesis is focused on. Allied dwarfs are implemented in a BP_Dwarf blueprint. Functions common for all dwarfs are

declared there. For example, action that can be executed shows up when the player hovers over the dwarf. Also, it implements an interface that abstracts interactions with the player.

Since all dwarfs have the same controller, changing the behaviour is trivial. Changing a value of one variable is enough to turn a worker into a warrior and vice versa.

Workers are responsible for mining resources, collecting them and bringing them to base. The aforementioned behaviour is illustrated in figure 14. For clarity and simplicity, some nodes were removed from the illustration.

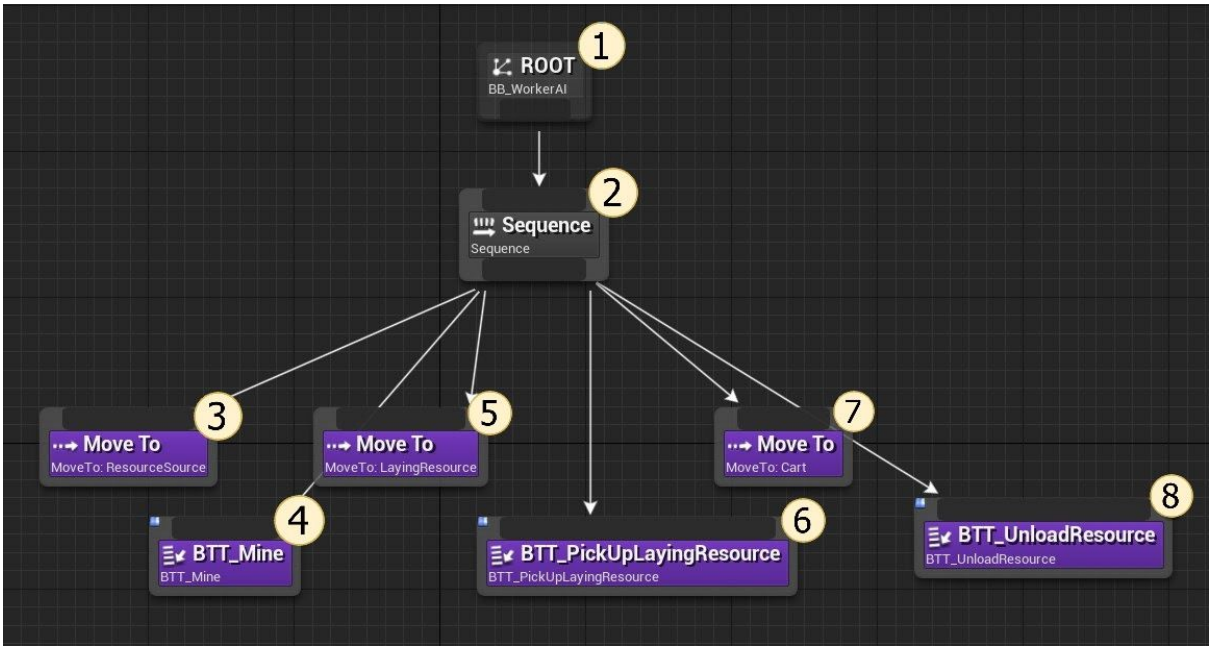

Figure 14. Worker behavior tree

The implementation of this particular behaviour is very straightforward. The execution of this tree can be represented with an ordered list, where order numbers correspond to the numbered nodes in figure 14:

1. Root node executes the sequence number two.
2. Sequence responsible for mining.
3. Task that makes the dwarf move to the resources.
4. Task that performs a mining action. Internally it calls a function of the resource object that spawns a small chunk of resource.
5. Dwarf moves to the newly created chunk of resource.

6. Task that is responsible for taking the resource chunk in hands.
7. Dwarf moves to a cart.
8. Dwarf transfers the resource to the cart.

After that the loop repeats until resources run out or the dwarf gets interrupted by enemies. Handling the interruption is left out of the three illustrated in figure 14 to minimize confusion.

6. Testing

As a part of the thesis five scenarios were created for testing purposes. Each scenario is focused on testing a particular feature that presumably can affect how people perceive artificially controlled characters. Scenarios consist of two to three levels that have similar layout, but slight deviations in visual representation. The demo application is included in Appendix I.

Nine people took part in the testing. Player, or in this case the experiment participant, played a passive role of an observer during the tests. Participants were asked to watch after AI controlled characters and fill a questionnaire in Appendix II

6.1 Scenario 1

The aim of this scenario was to check if it is easier to understand what is happening when actors have human-like models. This scenario consisted of two levels. In both of them AI controlled characters were mining resources and delivering them to the cart. In the first level characters were represented by cubes and the model of resources source was also swapped for a plain cube. In the second level, characters looked like dwarfs and the resources source also looked like it should. The layout of both levels is shown in figure 15.

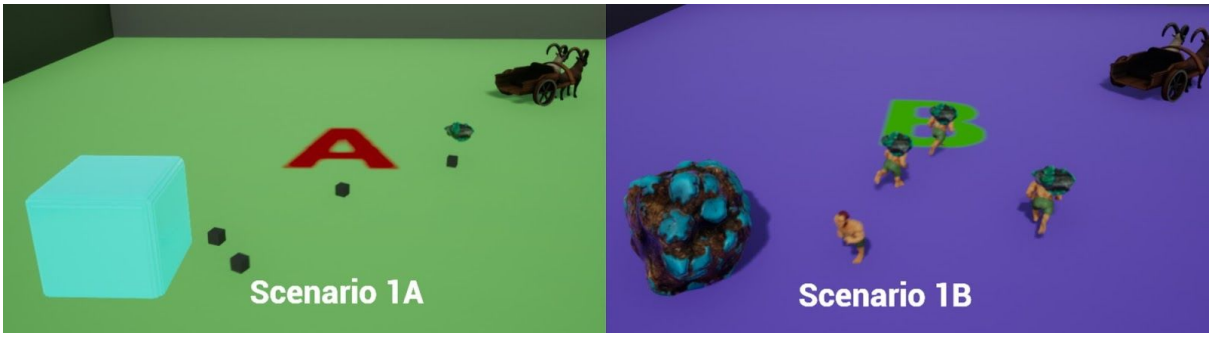

Figure 15. Levels of the first scenario.

Players were asked to rate how easy it is to understand what the cubes are doing. Figure 16 shows that it was easier to understand the intention of actors who had human-like models. It is worth noting that all participants told that they understood the actions of actors in both levels,

but in the first level they were less confident in their opinion.

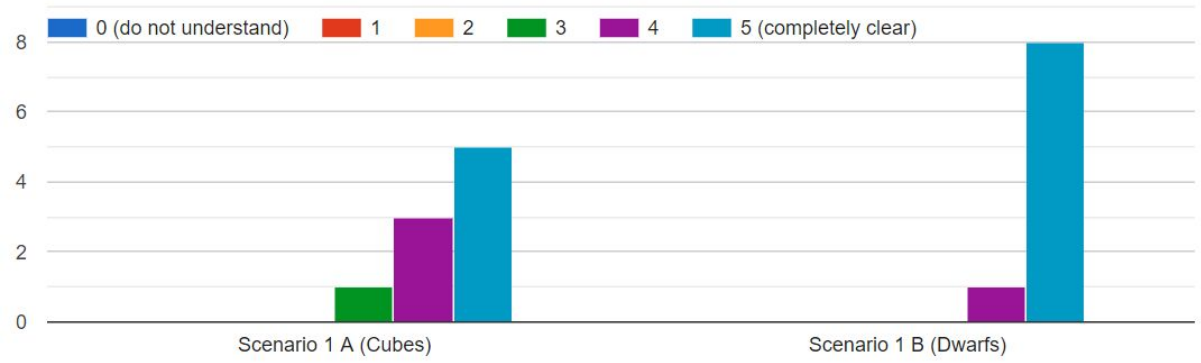

Figure 16. Scenario 1 results.

6.2 Scenario 2

The aim of this test was to check which feature is more important for understanding the intentions of AI characters. This scenario consisted of three levels. There are two teams of entities and they are hostile towards each other. In all three levels members of opposing teams fought against each other. The first level did not have any helping visuals, entities were just cubes. In the second level members of different teams had different colors. And in the third level number representing damage appeared after each attack. Players were not instructed about what changes happen and what numbers are representing. The layout of all three levels is illustrated in figure 17.

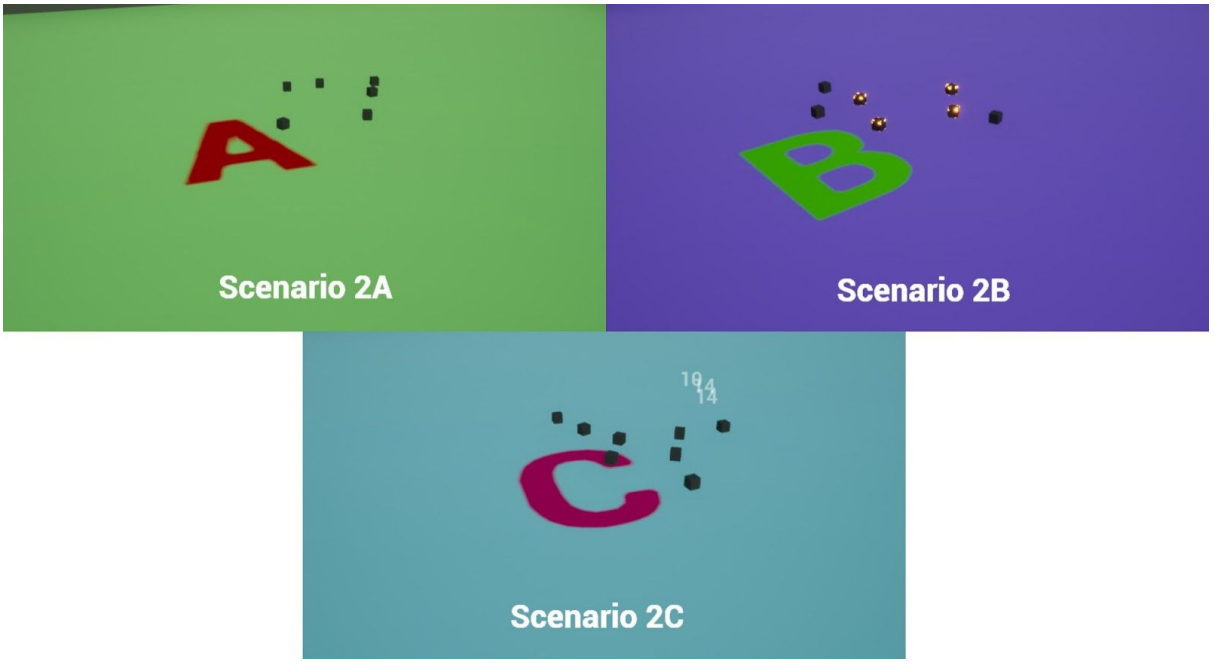

Figure 17. Levels of the second scenario.

Players were asked to rate how easy it is to understand what the cubes are doing. The first level was the most confusing. Opinions greatly differed about this level, but it is clear that the amount of people who found this level hard to understand is bigger than those who thought otherwise. Figure 18 shows that both second and third levels were easier to understand, therefore differentiating actors by color or adding visual representation of attack helps people to spot what is going on. Number of people who thought that the third level is easy to understand is much larger than in other levels, so a conclusion can be made that numbers representing damage are an effective way of telegraphing an act of attacking.

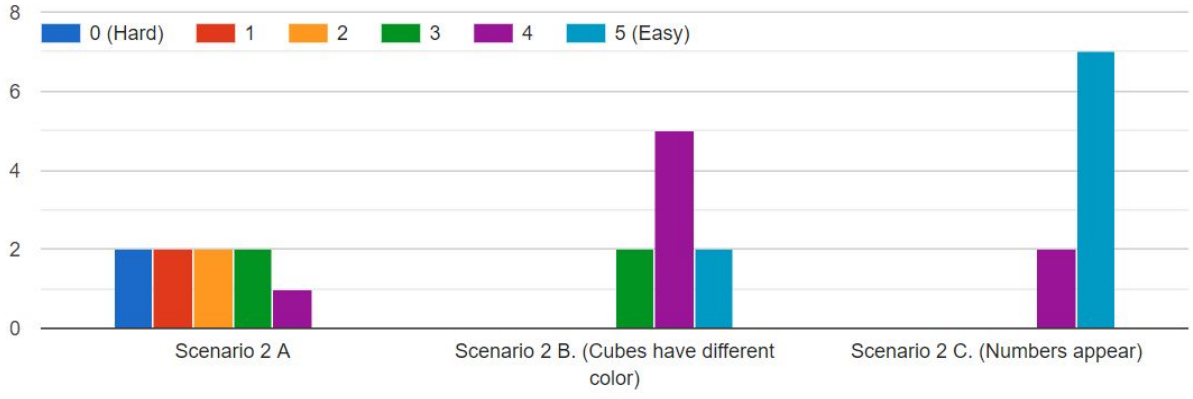

Figure 18. Scenario 2 results.

6.3 Scenario 3

The goal of this scenario was to test which features help people to understand the behavioural patterns of AI controlled characters. Both levels had the same layout and the same number of dwarfs who used the same strategies. In the first level dwarfs had text above them that showed what they are currently doing. In the second level debug lines showing the paths and current destination of the dwarf were used. Dwarfs used three patterns: mining, guarding a certain are and patrolling. The layout of the levels is illustrated in figure 19.

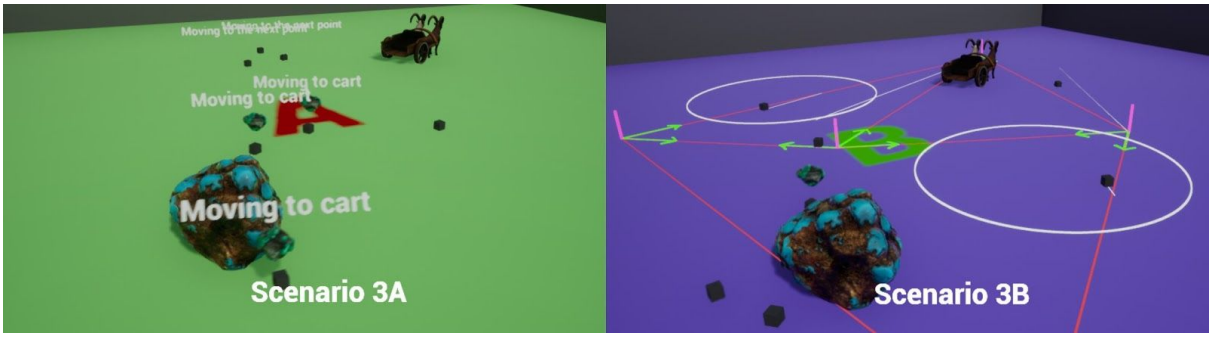

Figure 19. Levels of the third scenario.

Player had to try and count how many behavioral patterns dwarfs were using. As figure 20 demonstrates, in the first level most of the participants counted only two patterns. It can be related to the fact that guarding an area and patrolling are visually very similar. Nevertheless, three players got the number right. One person managed to find a fourth pattern, but there is no way of knowing what this pattern is. In the next level six out of nine players counted patterns right, which indicates that debug lines are more efficient in informing the player about what is happening. There was another question about this scenario and answers showed that 77.8% of players found debug lines to be more helpful, which overlaps with the previous conclusion.

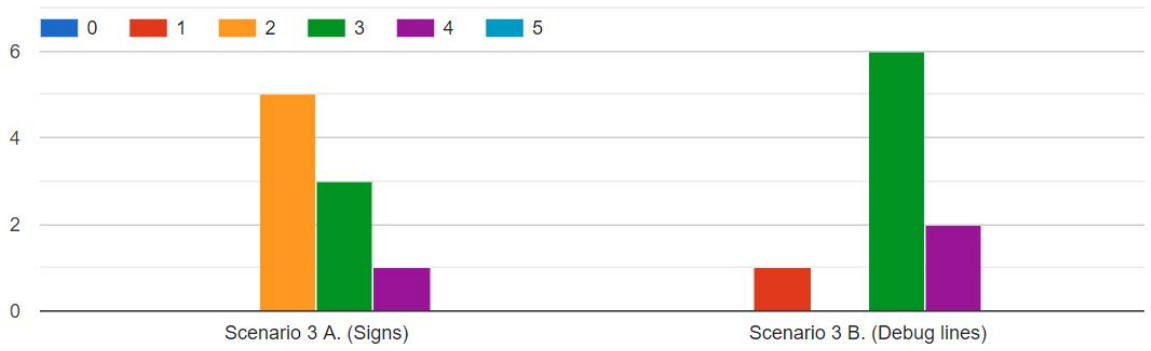

Figure 20. Scenario 3 results.

6.4 Scenario 4

This scenario was created in order to check if human-like characters telegraph emotions more clearly. In the first level cubes with yellow glowing attacked normal cubes and in the second level cubes' models were swapped for dwarfs. The layout of both levels is illustrated in figure 21.

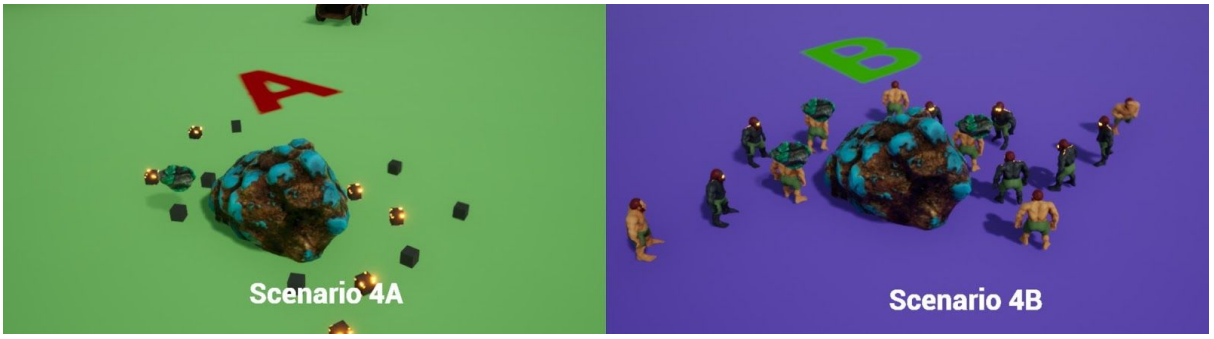

Figure 21. Levels of the fourth scenario.

Players had to rate how aggressive are yellow cubes and black dwarfs. Figure 22 demonstrates that in the first case the most popular answer was two out of five, which can be considered to be not aggressive. Nobody rated the cubes as five in aggression. Dwarfs, on the other hand, were mostly marked as aggressive. Only one person found dwarfs to be not aggressive.

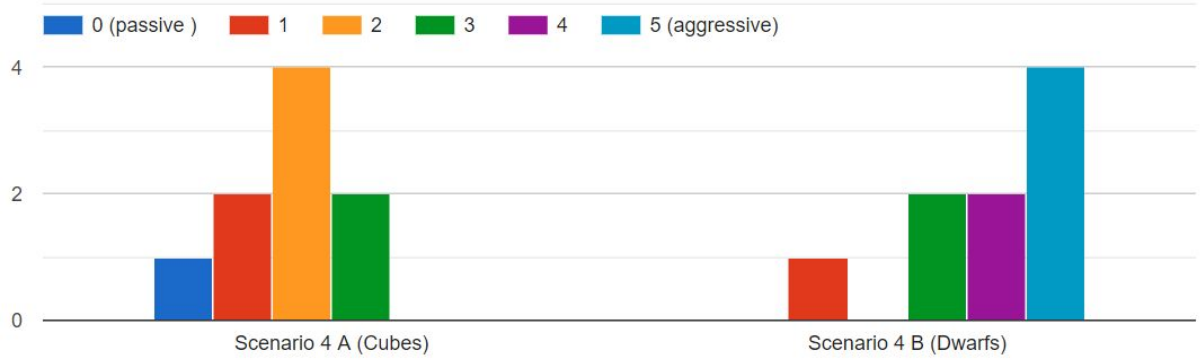

Figure 22. Scenario 4 results.

6.5 Scenario 5

This scenario was implemented in order to check what visual helpers are the most helpful and if players would be able to differentiate strategies. In the first level all visual helpers were turned on: debug lines, damage numbers, signs above the dwarfs, models of dwarfs and color segregation. In the second level only models, color and numbers remained. Also, the confrontation was biased in black dwarfs' favor - they had more health. The layout of the levels is illustrated in figure 23.

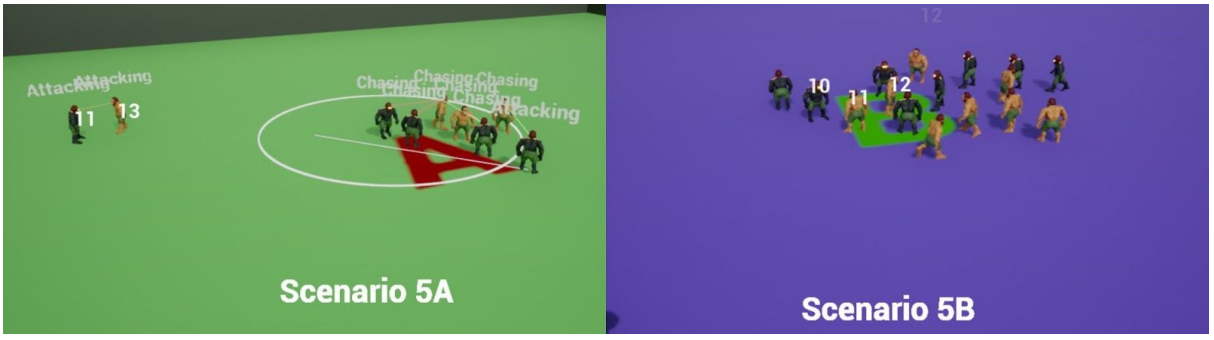

Figure 23. Levels of the fifth scenario

Figure 24 demonstrates that players found the second level to be easier to follow. It is related to the fact that the first level had too many visual helpers, so the player could not focus on what is really going on.

There were two additional questions: which team used more aggressive strategy and which team's strategy is better. Answers to both questions completely overlapped, the majority of players found black dwarf more aggressive and that they had better strategy. In reality, teams had the same strategies and none of them was more aggressive. It appears that stronger

enemies also seem more intelligent, which corresponds to the results of Halo playtests that were discussed in Section 2.2. One fifth of the participants were right and answered that strategies and aggression levels are the same.

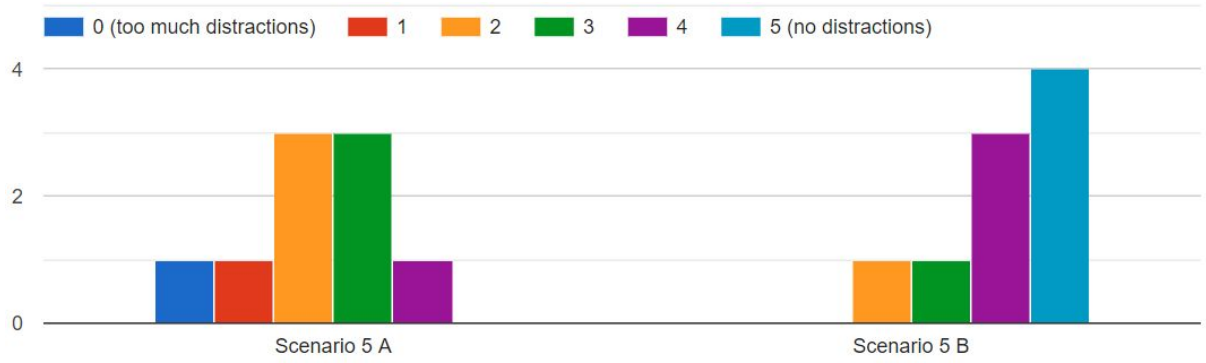

Figure 24. Scenario 5 results.

7. Conclusion

This thesis describes the process of creating artificially controlled characters, namely, dwarfs. Various methods of implementing non-player controlled characters were described and different methods of simulating dwarfs' behaviour are explored.

The thesis also gives an overview of most commonly used techniques and tools that can be used for adding AI controlled characters to a game. A system for simulating Dwarf Behaviour is implemented in Unreal Engine 4 using the behaviour tree system. A number of unique behavioural patterns were also added into the game and a series of experiments were conducted. The results confirmed which features are important for perceiving the characters and which are not. Experiments were unique and each demonstrated something useful that can be used in the future for implementing smarter AI.

The final result contains information that can be useful when designing and implementing an AI. In the future new behavioral patterns, interactions between entities and types of characters can be added to enrich the game world.

References

Rabin, S. (2017). *Game AI Pro 3: collected wisdom of game AI professionals*. AK Peters/CRC Press.

Wetzel, Baylor ja Anderson, Kyle jt. What You See Is Not What You Get Player Perception of AI Opponents. *Game AI Pro*, 2017

Biello, D. Expectations Influence Sense of Taste, 2019
https://www.scientificamerican.com/article/expectations-influence-se/

Dill, K., Pursel, E. R., Garrity, P., Fragomeni, G., & Quantico, V. (2012). Design patterns for the configuration of utility-based ai. In *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)* (No. 12146, pp. 1-12).

Gaudl, S. E., Davies, S., & Bryson, J. J. (2013). Behaviour oriented design for real-time-strategy games. In *FDG* (pp. 198-205).

Rabin, S. (2002). *AI Game Programming Wisdom.* Charles River Media.

Sizer, B. (2018). The Total Beginner's Guide to Game AI, URL
https://www.gamedev.net/articles/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/ (08.05.2020)

Appendix

I. Application

Version of the game that was used for testing can be downloaded from the following link: https://drive.google.com/drive/folders/13JRB60HTWTAUAiPEyC8QNazomhAE9VK6?usp=sharing

Unzip the file “Build.zip” and double click “MultiplayerBase.exe” to start the game.

II. User Testing Questionnaire

User testing questionnaire is located in Appendix I attached archive “AttachedFiles.zip” in folder “Testing”.

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Aleksei Beljajev,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,
   Dwarf Block Game Development - Dwarf Simulation,
   supervised by Jaanus Jaggo.
2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Aleksei Beljajev

08/05/2020