How to combine the strengths of current Artificial Intelligence techniques and extend them to support non-interruptible actions.

Student name	**Silvan Josua Ott**
Student number	**101402**
Course	**BSc (Hons.) Games Programming**
Assignment	**6FSC1PE102**
Submition date	**11.08.2023**
Word count	**6156**

Confirmation of Authenticity

I hereby declare that this is my own work, and does not use any materials other than the cited sources and tools. All explanations that I copied directly or in essence are marked as such. This work has not been previously submitted.

[illegible]

……………………………………………..

11.08.2023, Liverpool

Abstract

During this project the strengths and weaknesses of current Artificial Intelligence (AI) techniques were determined using several reputable sources. A new AI structure was then created, based on the AI agent model (Russel and Norvig, 2020) and with an additional fourth layer between decision and actuation as discussed in the GDC (2017) talk. Each layer was assigned one AI technique based on which strength best suited the needs of the layer. To test the created structure, an environment was created with a basic AI in it. Observation, white-box and unit testing confirmed the correct functionality of the AI in the environment. The structure's value in industry use was determined through qualitative analysis.

Building on above work, a structure has been created that consists of an outer framework to define how each layer interacts with each other. The inner structure contains the implementation of each layer by one AI technique. Each layer implementation can be viewed as a module and be easily exchanged with a different AI technique. This emphasises the single responsibility principle and allows the developer to make a wide range of individual adjustments by using the framework of the structure. By separating the actuation from the AI and putting it on the agent, the system also supports non-interruptible actions without interfering with the AI logic.

Table of Content

Table of Figures

List of Tables

List of Abbreviations

Abbreviation	**Meaning**
AI	Artificial Intelligence
GDC	Game developer conference
BT	Behaviour Tree
GOAP	Goal oriented action planning
UML	Unified modeling language

Acknowledgements

I am extremely grateful for the academic support I received from my supervisor Farhan Ahmed and Zahra Muhsin. I would like to extend my deepest gratitude to everyone who took the time to read and help my improve my work, especially my aunt Monika Harter. Special thanks to my parents who supported me throughout my whole study and game me the opportunity to be here. I also would like to recognize the help I received from Alex Hannah and Firduas Khalid during this final year.

Introduction

This research will analyse some of the currently used AI techniques. It will determine their strengths and how to utilise them as well as their weaknesses and how to avoid them. A new AI structure will then be created with the aim to focus on the strengths of the individual AI techniques. As an additional requirement the new structure should support non-interruptible actions and be easily interchangeable between AI and player control. The created architecture will not be compared to other structures in terms of performance and memory usage since these factors are very dependent on the actual individual AI created in a game. Nonetheless, attention will be paid to improving performance and optimising memory usage wherever possible during the implementation of the project. Finally, the structure will be assessed by its strengths and weaknesses. The results will allow a developer to determine if this structure is suitable for the implementation of his AI. To test the structure, a test environment with a basic AI agent will be created. The AI executes simple tasks such as: gather food, chop wood, build houses and attack. Pathfinding, even if part of an AI, is not inside the scope of this project and not implemented.

This project will be executed with the Unity framework version 2020.3.41f1. All code is written in C# if not stated otherwise. During this project, when referencing an Agent, it refers to the visual representation. Therefore, the character that can be seen in the game executing actions. When talking about the AI, the non-visible part is referred to, basically the equivalent to the player of a game. The AI includes everything from the perception of the environment over the decision-making what action should be executed up to the command to execute a certain action. This project focuses on AI in real time games not turn-based.

The target audience of this work are primarily game AI developers, but also AI developers in other fields such as robotics. It might also be of interest to people who use AI development techniques like behaviour trees and state machines.

Problem definition

Many AI techniques have been developed. Behaviour Tree, State machine, Fuzzy logic, Rule-based and Goal oriented Action planning (Millington, 2019), to list only a few. All of them have their strengths and weaknesses. However, according to GDC (2019) and Anguelov (2020), some techniques are utilised without making best use of their strengths. In contrast, their usage may bring out their weaknesses, which then must be overcome with much additional work.

External Resources

The whole project is available on GitHub: https://github.com/Morchul/MajorProject.

Ethics

This project does not involve or affect the life of any person. Other people who contribute to the project apart from the researcher are the supervisor and professionals in the industry who review and evaluate the utility of the created AI structure. They are referred to as reviewers. If permission is granted by reviewers, their name, past or present experience and workplace are recorded to show credibility of their industry experience.

Method

In the GDC (2019) talk by Anguelov and in his subsequent paper (Anguelov, 2020), the author discusses the common misuse and disadvantages of BTs in the decision-making layer of AIs in video games.
He explains that the decision-making process is in nature cyclical and that a BT works in an acyclical way. This is one of the reasons the author argues that BTs often do not shine in the process of decision making. He also points out the difficulties that arise through the static prioritisation of nodes in BT. Proposed in the GDC talk is the creation of a new layer in a standard AI agent model. Instead of only the three layers: sensor, decision making and actuation a fourth layer should exist between decision making and actuation where the BT would be of best use.
Based on Anguelov’s argument, this project aimed to create a new structure where the BT would not be used for decision making, but instead be used in the third layer of the four-layer system described by Anguelov. As an additional requirement for the newly created structure, it should be able to support non-interruptible actions. This was achieved by putting the actuation layer on the agent and by separating it from the actual AI. To determine how each layer should be implemented, several reputable sources were revised and watched to collect information about the strengths and weaknesses of current AI techniques. By comparing the tasks of each layer and the strengths of AI techniques, each layer was assigned an appropriate technique. To verify that this technique is applicable in real life, a test environment with a few AIs executing simple tasks was created.
After the initial research, the implementation was split into three parts. Firstly, the creation of the outer framework. The base layer contains mostly interfaces, which represent the four layers and define how they interact. Secondly, the AI techniques chosen for each layer were implemented and provided an internal structure for the framework. Thirdly, the test environment was created with a few objects and the specific logic for an AI agent that can perform basic tasks. The functionality of the agents and AIs was confirmed through observation, white box, and unit tests. Finally, with a qualitative study approach and data collected in discussions with people who have experience in game AI programming, the usability of the structure in the industry was determined.

AI Techniques Research

Behaviour Tree

A behaviour tree is a hierarchical tree structure which is used to define the process of actions of an AI. It consists of a root object, branches, and leaves. While the root and branches can have child objects and determine the flow of the behaviour tree, the leaves contain the actual logic the AI should execute. With every tick the AI controller traverses through the tree from the root to find a leaf to execute. Information throughout the behaviour tree is communicated over a blackboard where all nodes can write and read values. The tree structure enables simple expansion and offers a modular way to build the AI (Rabin, 2017, pp.115–124).
A technique to not go through the tree at every frame is the event driven behaviour tree. There, every node which is visited but not executed registers a monitor node. If one of the monitors triggers, the controller reevaluates the tree to find a new active node. Thus, a new evaluation of the tree happens only if a monitor triggers, and not with every tick.
A disadvantage of behaviour trees when using them for game AI is their nature of being acyclic and having a static prioritisation of tasks. They are also ineffective in handling interruptions and transactions (GDC, 2019).

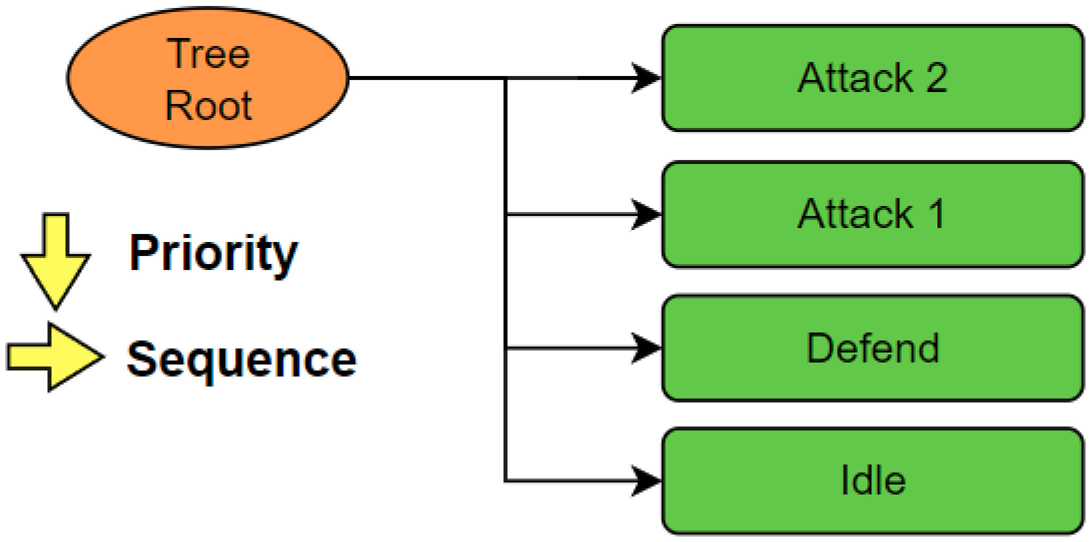


Figure 1 BT example

The above Figure 1 shows the problem of static prioritisation. If “Attack 1” is executed, the defend option is not even considered even if it might be a better option. To allow the AI to consider defending, “Attack 1” needs to know when it is preferable to choose “Defend” and to skip itself. This leads to a large increase of dependencies and destroys the principle of single responsibility. Especially higher priority actions need to know many special conditions for the lower priorities.

Utility theory

Utility theory is a way for AI to handle decision making. When using utility theory every possible option gets assigned a value, the utility. The higher the value the more useful the decision is in the current situation. One important factor when calculating the utility is that the results must be comparable between each other. A good way to do this is by normalising all the resulting values in a certain range. (Rabin, 2023, pp.113–126). The calculation of a decision can be influenced by several factors as shown in Figure 2:

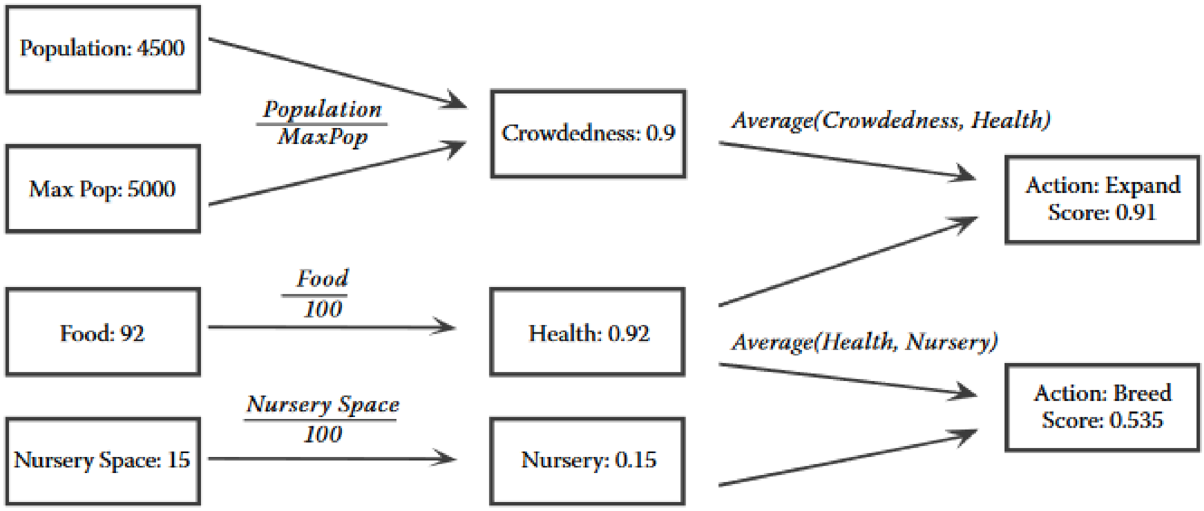


Figure 2 Utility calculation (Rabin, 2023, pp.116)

While in the above example the calculations are mostly linear, any function can be used to calculate the utility e.g. exponential or logarithmic.

After calculating each utility one option must be picked. The simplest way is to just pick the one with the highest utility, this approach is called absolute utility. But there are other methods to use the utility value. Another approach is the weighted random, in which case the utility determines the likelihood that this option will be chosen. The probability is calculated by dividing the utility of the option by the total utility of all options (Dill et al., 2012):

$$P_O = \frac{U_O}{\sum_{i=1}^{n} U_i}$$

Figure 3 Weighted random formula (Dill et al., 2012)

A more modular approach is to create several considerations. Each consideration independently analyses one single aspect of the current situation and returns a value which can be combined or compared with other considerations. A decision then consists of several considerations that together create a final value for this option (Lake, 2011).
Utility AI is a great tool for decision making as it considers every option and can compare these to come to a reasonable decision. In addition, utility AI is very scalable and with a few weighting values every AI can get its own personality.
Utility theory is often used in combination with other AI techniques rather than on its own, there is no execution logic.

Finite State Machine

A finite state machine is a set of states connected to each other by transitions. The AI occupies one state at a time and executes the logic defined by that state. Each state contains at least one transition to another state. If one transition triggers, the state machine will change the current state of the AI to the one connected by the transition and a different state is executed. One problem with state machines is that they are not well scalable. One new state can lead to several new transitions, which may quickly result in unreadable, massive state machines. One approach to minimise transitions is the hierarchical state machine, where states can act like parent states to combine states and limit transitions. However, this will not help if the state machine reaches a certain size (Millington, 2019). The strength of the state machine lies in describing states and in reacting to external influence with transitions to adapt the state. On the other hand, the state machine is not useful in the description of a sequence of action, and not ideal in handling interruptions (GDC, 2019).

Goal oriented action planning

In GOAP the AI selects one of multiple valid goals. This goal has a required world state. The AI then must pick an action which results in the required world state. The selected action also has a required world state for which the AI must pick a suitable action. This continues until the required state of an action is the same as the current state of the AI. Then all actions will be sequentially executed until the goal is achieved.
How this goal is selected is open to the developer. Utility theory would be an example, but GOAP itself does not make decisions, it rather finds the quickest path from one world state to another by executing actions.
To find the correct actions to get as quickly as possible from the current AI state to the desired goal state, a planner is used that holds all the possible actions. One algorithm for the planner to find the best actions is the A* algorithm since the chain of actions can be seen as a path and A* finds the shortest path between the current and the desired state (GDC, 2017).
With GOAP AIs can solve problems in a dynamic way.

Non-interruptible actions

One essential part of an AI is the ability to react to events or changes in the environment. Apart from turn-based games, which are not considered in this research, change in the environment or an event can happen at any time. This means that the AI needs to be able to react at any moment. However, the core property of a non-interruptible action is that it cannot be interrupted or stopped at any moment. Its execution will take a certain time. Therefore, a conflict will exist between the cycle of an AI and an uninterruptible action. There are ways to support these kinds of actions, but they all come with a lot of additional work, increasing complexity and dependency while decreasing readability and modularity (GDC, 2019). To support non-interruptible actions without overengineering the AI, the actuation layer must be independent of the AI. With the independence of the actuation layer, the controller of the agent would be able to use it as an interface, behind which the abilities of the agent lie. This would open a way to simply swap between AI and player controlling the agent.

Structure

A basic AI agent model is divided into three main parts: Sensor, Decision, Actuation. Sensor is the process of gathering information about the environment through inputs like sound or visual feedback, and of reacting to a change in them. Information is gathered about nearby objects, units and other important states of the environment that influence the actions and decisions of the AI. The actual execution of an action by an agent is part of the Actuation layer and includes but is not limited to animation, change of state of the AI himself or another entity, sound, or physical motion (moving, jumping). The decision part of the AI will decide which action is going to be executed depending on the input gathered by the sensors. The decision logic can vary for each individual AI and can range from being simple to being extremely complex (Russell and Norvig, 2020).

In this project an additional layer will be added to the structure of Sensor, Decision and Actuation. As discussed before, many commonly used AI techniques in decision making do not exceed this task and their strength lies more in planning to achieve a set goal. This idea is also discussed in the GDC (2019) talk with the behaviour tree added in between the decision making and the actuation layers. The misuse of BT in decision making is also discussed in Anguelov (2020). The additional layer will be called Planning or Planner and is located between decision and actuation.
It should also be mentioned that the actuation layer is not located on the AI but on the Agent, since the abilities of the Agent are not determined by the AI. The AI merely decides how to use the possible actions of the agent to achieve a desired goal.
The next figure shows the base structure and life cycle of the four-layer system:

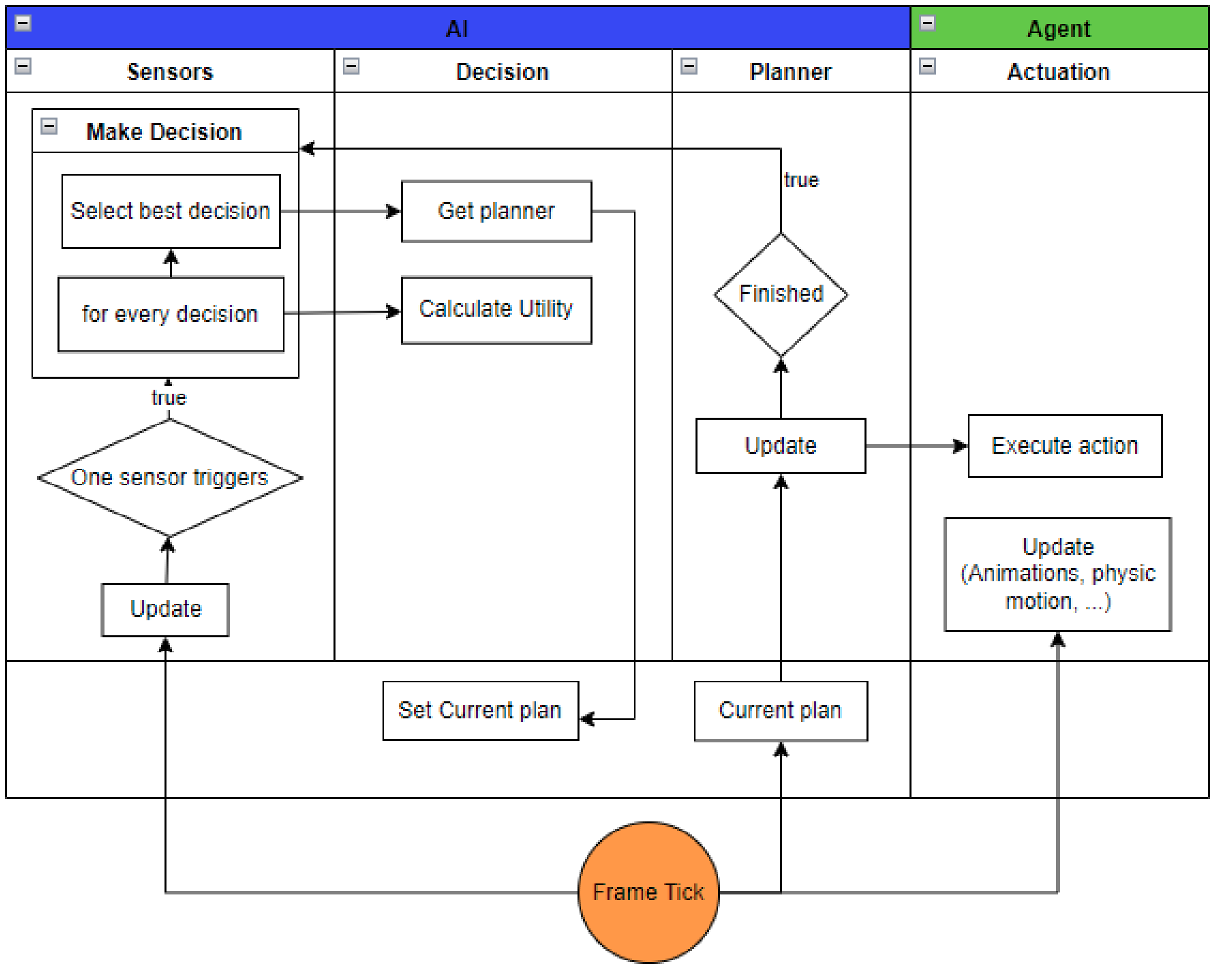


Figure 4 AI structure

The sensors will be updated every frame. If one sensor triggers because something happened in the environment around the AI, a decision-making process is started to determine what plan should be executed next. The current plan will then be updated every frame. The plan will execute actions on the agent. Once the plan finishes, a new decision-making process will be executed to determine the next plan. This is a very simplistic representation of the structure but gives a first overview.

Implementation

Framework

The outer structure should provide an interface and offer more of an outline to the four layers and not force the developer to use one specific technique like BT or GOAP. The structure can be seen as a framework that supports combining several known AI techniques in the form of the four layers: sensor, decision, planner, and actuation. It is up to the developer to decide which technique is used in each of those layers. Figure 5 Framework UML shows the base implementation of those classes, enums and interfaces.

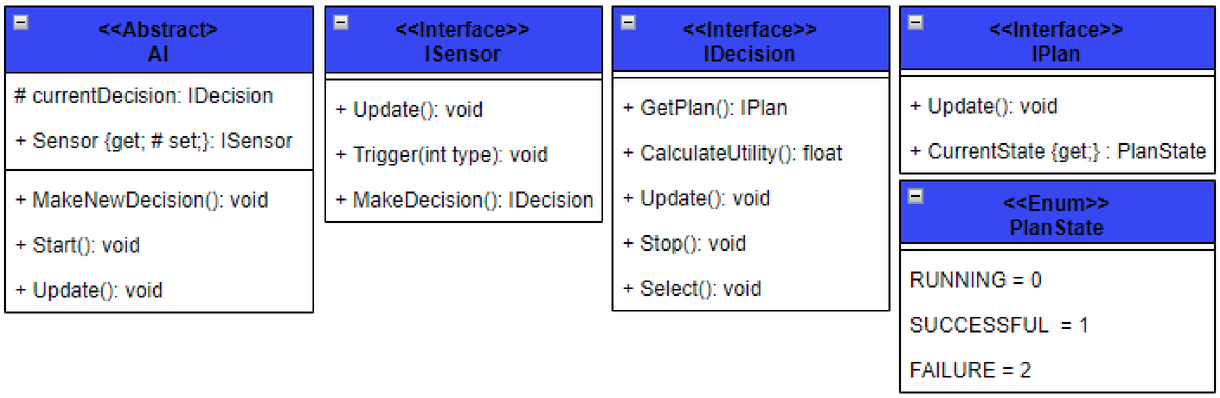


Figure 5 Framework UML

Sensor

In this project Finite state machines will be used for the sensor layer to react to the environment and to enter an appropriate state.
The finite state machine was chosen because it is not responsible for decision-making nor for the execution of a series of actions, as those would lead to big state machines. Which is preferably prevented. It was chosen because of its ability to group things in a higher hierarchy state and because, with its transitions, it is also able to react quickly to external events.
Each state consists of a set of possible decisions and transitions that are viable in the current situation. This will allow the developer to group decisions into appropriate situations and prevent the AI from testing every decision even if clearly invalid.
There will be two kinds of transitions. Continuous transitions which will be checked every frame and trigger transitions which must be triggered from an outside source like a physics trigger or a nearby event. If one transition triggers, there are two possible outcomes: either

the state changes and a new decision will be made in the new state, or the state stays the same, but a new decision process will be executed. This allows for the creation of simple or complex transitions.

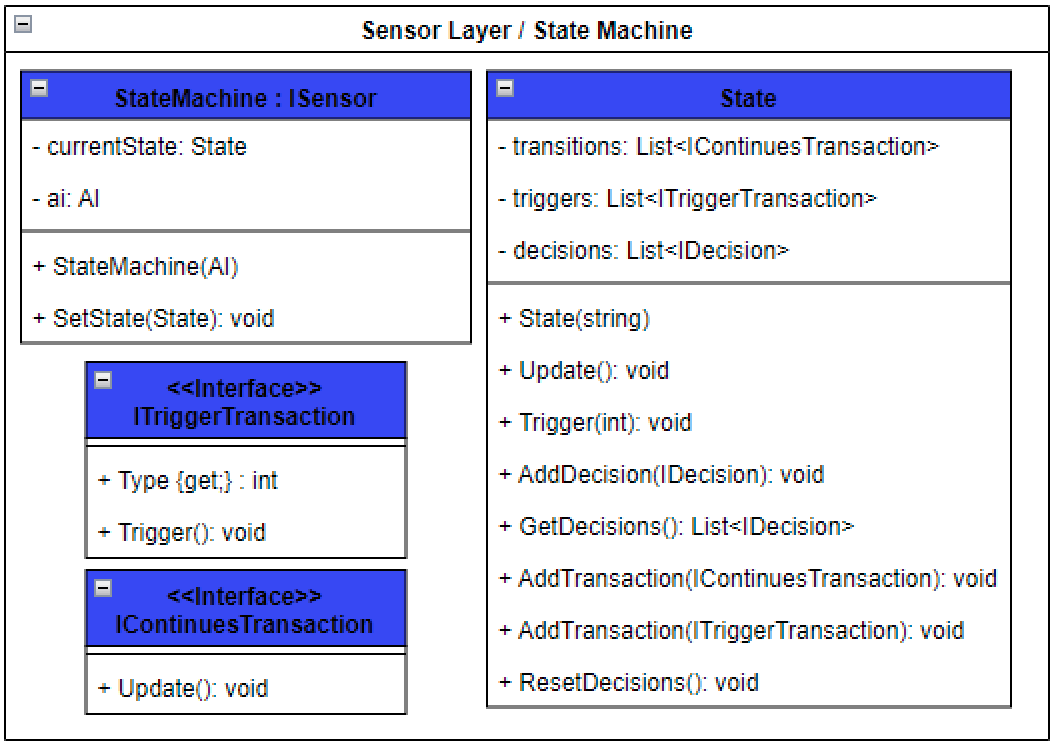
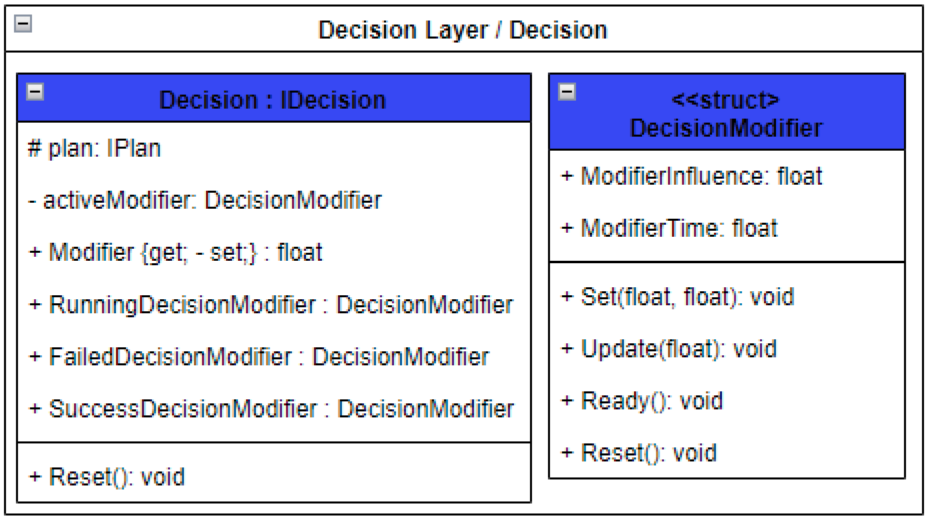

Sensor Layer / State Machine
StateMachine : ISensor
- currentState: State
- ai: AI
+ StateMachine(AI)
+ SetState(State): void
<<Interface>> ITriggerTransaction
+ Type {get;} : int
+ Trigger(): void
<<Interface>> IContinuesTransaction
+ Update(): void
State
- transitions: List<IContinuesTransaction>
- triggers: List<ITriggerTransaction>
- decisions: List<IDecision>
+ State(string)
+ Update(): void
+ Trigger(int): void
+ AddDecision(IDecision): void
+ GetDecisions(): List<IDecision>
+ AddTransaction(IContinuesTransaction): void
+ AddTransaction(ITriggerTransaction): void
+ ResetDecisions(): void

Figure 6 Sensor layer UML

Decision

As for the decision-making layer, this project introduces decisions. Decisions are based on utility theory and contain a plan and a method to calculate the utility of this plan. If a decision is chosen, the attached plan will be selected. One problem that might occur with utility theory is that the same option is chosen repeatedly because it simply has the highest utility. But maybe this option cannot even be executed now, or it might become repetitive if an AI always does the same, as discussed by Dill et al. (2012). To prevent this from happening, every decision will, after its completion or when interrupted because of a new decision-making process, save their last state which is either SUCCESS, FAILURE or RUNNING. Each decision then has three decision modifiers, one for each state. These modifiers will affect the calculation of the utility for a defined duration. Through this an option can be emphasised to be continued if it was running, but it can also be used to lower the score if this option failed to tell the AI that this option is currently not executable. Utility theory is the perfect choice for this task. Its strength lies in the determination of the value of a certain option, and through normalisation allows comparison with other options. This enables it to make good, believable decisions.



Decision Layer / Decision
Decision : IDecision
# plan: IPlan
- activeModifier: DecisionModifier
+ Modifier {get; - set;} : float
+ RunningDecisionModifier : DecisionModifier
+ FailedDecisionModifier : DecisionModifier
+ SuccessDecisionModifier : DecisionModifier
+ Reset(): void
<<struct> DecisionModifier
+ ModifierInfluence: float
+ ModifierTime: float
+ Set(float, float): void
+ Update(float): void
+ Ready(): void
+ Reset(): void

Figure 7 Decision layer UML

Planner

The BT is used for the planner layer. Another possibility would be to use GOAP instead of BT. This is open to the developer. As discussed in the talk GDC (2019), BTs are an excellent option to describe a set of actions that, if executed, lead to a desired outcome. While BT already knows the sequence of the actions it must execute, GOAP will determine the actions at runtime. The BT was implemented by Aversa, Sithu Kyaw and Peters (2018, pp.172–189). The base classes are BTRoot, BTNode and BTComposite. Each BT object except the root inherits BTNode, which contains a tick method and returns the status, which is either RUNNING, FAILURE or SUCCESS. The BTNode will not contain any execution logic of the agent, but references to the action the agent should execute.

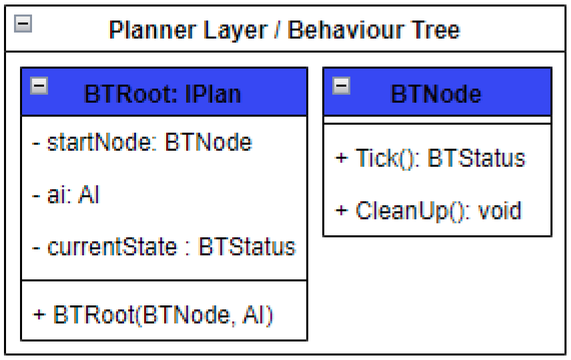


Figure 8 Planner layer UML

Actuation

As opposed to the three other layers, the actuation layer will not be a layer of the AI but will be located on the Agent. Due to this independence, it does not matter to the agent if it is controlled by an AI or by a player. As discussed, this separation will allow the handling of non-interruptible actions without interfering with the AI. In RPG games there is a common skill called animation cancelling, which aims to interrupt actions that would go on for a longer duration, to save time, as shown and explained in this video by Isth3reno1else (2022). Animation cancelling can be prevented with a structure that supports non-interruptible actions.

The agent will utilise the command pattern and an adjusted ring buffer to achieve the handling of non-interruptible actions. With the ring buffer the active actions can be monitored and updated. The ring buffer is described further below. The command pattern allows it to not execute an action immediately, because it might need to wait until a non-interruptible action has finished and execute it at a later point (Nystrom, 2014).

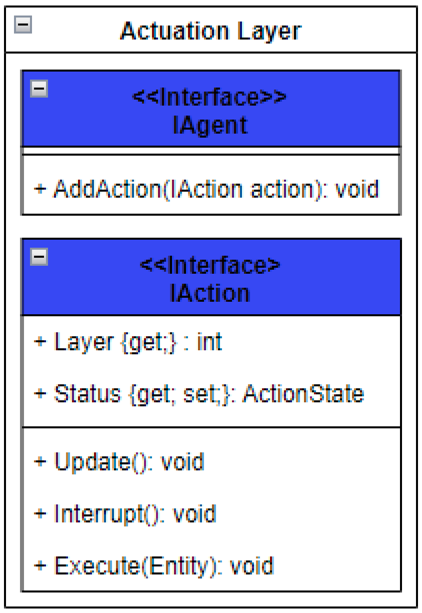


Figure 9 Actuation layer UML

Actions

Each Action has an Execute, Update and Interrupt method, which handles the basic life cycle of it. Actions also contain a status, layer, ID and a name. The name is for debugging purposes. The layer tells if an action intersects with other actions or if actions can run alongside each other. E.g. you can walk forward and turn your head, but you cannot swim and jump at the same time. The parameter of Execute should be a base class of all agents. The parameter represents the agent that executes the action.

These are the possible different statuses:

Table 1 Possible action statuses

Status	**Description**
WAITING	When an action is added to an agent its status is WAITING until it is added to the ActionRingBuffer.
ACTIVE	If an action is added to the ActionRingBuffer its status changes to ACTIVE
INTERRUPTED	If the Interrupt method is called on an action in state WAITING or ACTIVE, its status is changed to INTERRUPTED. This status needs to be handled inside the action itself.
FINISHED	If the actions logic is finished the status must be set to FINISHED by the action.
SLEEPING	The ActionHandler will change a FINISHED action to SLEEPING. This indicates that it should not be run anymore, but that it is still in the ActionRingBuffer. It can be reactivated.
INACTIVE	The ActionRingBuffer will set the status of an action to INACTIVE if it is removed from the buffer. The action cannot be reactivated and must be re-inserted in the buffer again.

ActionRingBuffer

A data structure inspired by a ring buffer is used to keep track of the current actions. The ring buffer was chosen because of its performance, static size, and there is no need to copy data around. Simple pointers keep track of the head and tail of the buffer (Ward, 2020).

Changes made to a default ring buffer were necessary because the ring buffer is a FIFO (First in first out) data structure. However, it might be that the first action that was added is not the first to leave. Therefore, instead of increasing the tail by one if one item was removed, the action at the tail will be checked if it is still active. If so, the tail stays the same. Since knowledge must exist in the ring buffer to test if the action is active, it is not generic anymore. The implementation of the adjusted action ring buffer is in Appendix A.

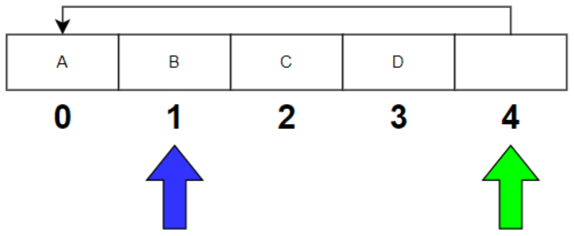


Figure 10 Ring buffer example before

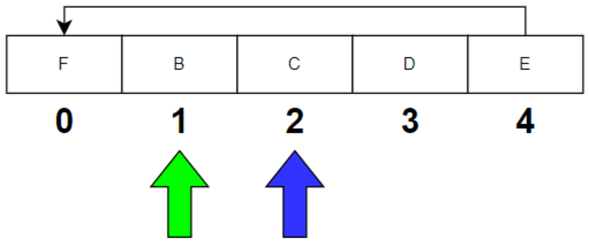


Figure 11 Ring buffer example after

In this example of a normal ring buffer, if element E is added, it will be at position four. The tail (green arrow) will be moved to the next position, which is position zero. If an element is removed, the head (blue arrow) will move to position two without the need to remove element B. If element F is added next, it simply overwrites A on position zero and the head moves to position one. One risk is that the tail will reach the same position as the head. If this happens, the ring buffer basically clears itself since the data structure now thinks it is empty. To prevent this, it is possible to not allow new items to be added if the tail would go to the same position (Ward, 2020).

In the adjusted ActionRingBuffer a few changes needed to be made and edge caches had to be handled. Next are four cases to show how the ActionRingBuffer works.

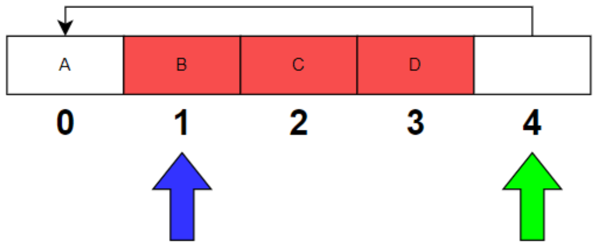


Figure 12 Action ring buffer, case 1

Case 1: In this example of the adjusted ring buffer the red elements are active actions, whites are inactive. Adding a new action is the same as before: simply add the element at the position of the tail (green arrow) and move the tail to the next position.

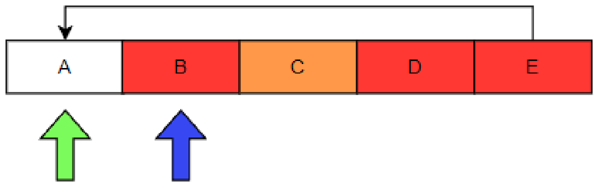


Figure 13 Action ring buffer, case 2

Case 2: If the status of an action changes to sleeping (shown in orange), a remove action is triggered. In this example C changes to sleeping. Now, instead of moving the head (blue arrow) to the next position it checks if the action at the head (B) is sleeping. It is not, therefore the head stays.

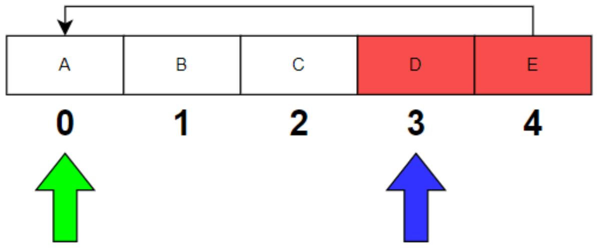


Figure 14 Action ring buffer, case 3

Case 3: To get all the active actions, loop through from head to tail, but only return non-sleeping actions. This creates a bit of an overhead since we must check if the action is still active or not. But thanks to the ring buffer head and tail, we do not have to loop through every element, since they set the scope of possible active actions.

If now action B changes to sleeping, the head will move to the next active action, which is on position 3. Every sleeping action that the head moves across will be set to inactive.

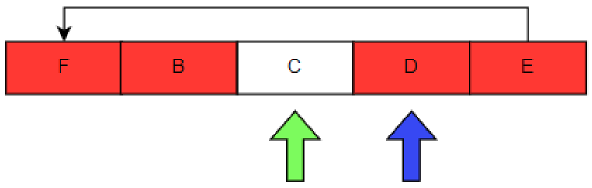


Figure 15 Action ring buffer, case 4

Case 4: Let's start from the final stage of case 2 and assume that the tail will reach the same position as the head because the action stays active for so long. In this case B is still active and action F is added. Now head and tail are at the same position. In theory the buffer would now think it is empty. Instead of just removing the action, both the head and tail will be moved forward by one position and a remove action is executed. This will bring the still active action from the head to the tail and the ring buffer can continue. In the unlikely event that all actions in the ring buffer are active and a new one is added, a deadlock counter will discover it and throw an error with the message that the buffer is too small.

One possible problem that might occur is that actions are classes and agents have only a reference to it. If a smart object now gives the same object to multiple agents, this will probably result in unexpected behaviour because one agent will change the status of the whole action and affect the other agent as well. To prevent this, it must be analysed if an action might be shared between several agents. If yes, the smart object should have an object pool to handle multiple instances of the action and distribute each action only to one agent (Nystrom, 2009).

Test environment

Up to this point the AI structure and the four-layer system were introduced. The following chapters will present a concrete example of a test environment to see the structure in action.
The main parts of this test environment are entities, agents, and components.

Entity

Entity is a base for everything that can be, in some way, interacted with or has values. It consists of several components, which determine what this entity is and what actions can be performed with it.

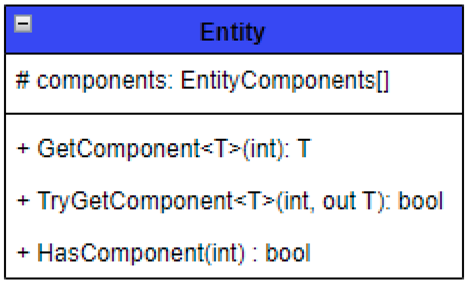

Entity
# components: EntityComponents[]
+ GetComponent<T>(int): T
+ TryGetComponent<T>(int, out T): bool
+ HasComponent(int) : bool

Figure 16 Entity class diagram

Entity Component

An entity component is a data container and can be accessed and changed by actions. Entity components determine what an entity is and what it can do. Components can be either public to allow actions to change them or private with methods to determine the way values are adjusted. To quickly access the right component, it has an integer as an identifier. A component can also determine actions that can be executed with the object having this component.

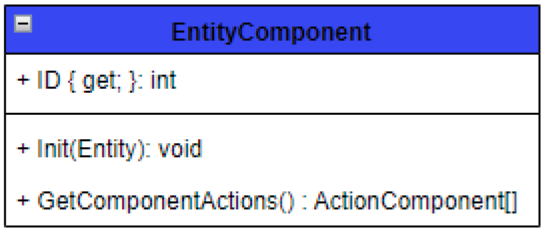


Figure 17 EntityComponent class diagram

Agent

Agents are entities but are additionally able to execute actions. Agents contain the action ring buffer introduced in the actuation chapter and represent objects which can be controlled by AI, player input or other sources.
Objects with the IHasActions interface contain actions that can be executed by an IAgent. Agents too can have actions like ‘move forward’.

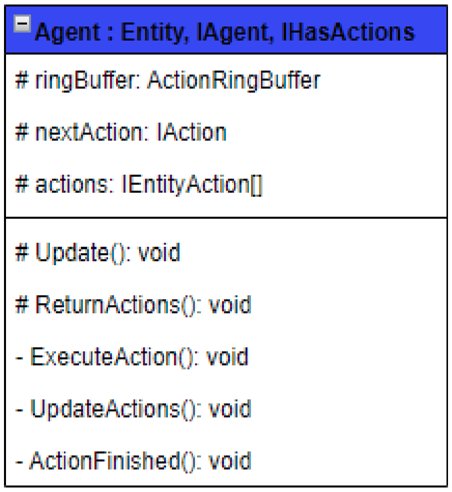


Figure 18 Agent class diagram

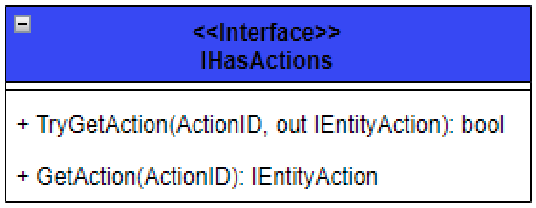

<<Interface>>
IHasActions
+ TryGetAction(ActionID, out IEntityAction): bool
+ GetAction(ActionID): IEntityAction

Figure 19 IHasAction interface diagram

The actuation layer workflow

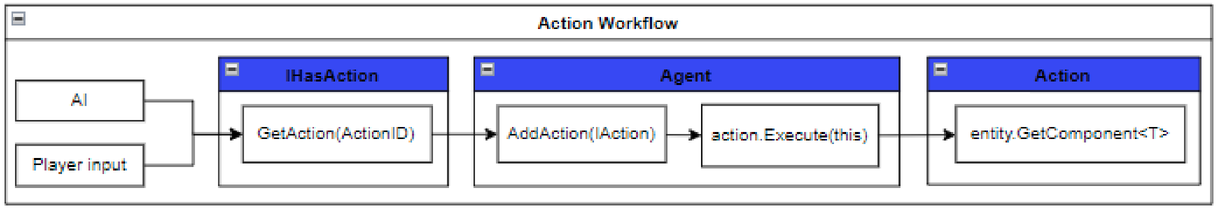


Figure 20 Actuation workflow

The main workflow of the actuation layer looks like the above figure 21. Agents are entities, which can execute actions and consist of several components. These components are accessed by the actions and changed. Actions can come from different places, for example agents themselves or smart objects. Through AI or player input these actions can be accessed and added to the agent for execution.

The final logic as to which animation should be played and how the value is changed can be either in the components or in the actions. The advantage of defining it in the action is that every entity will execute it in the same way. The logic of the action is centralised. An action might be executed in slightly different ways, in which case the action can call abstract methods on components that contain the final executions and every entity can have its individual version of the component. These both can also be mixed of course and allow a wide variety of possible actions, either unique per agent or centralised for everyone.

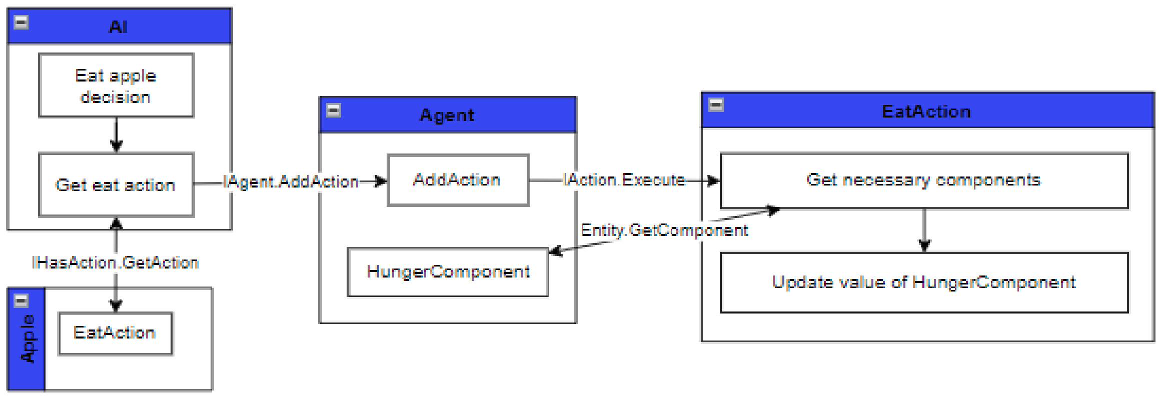


Figure 21 Actuation workflow example

In this example an Entity has the hunger component, which consists of a value to remember how much food he has eaten. The AI picks up an apple and selects the EatAction to be executed. The EatAction will be added to the Agent, which calls the actions Execute method with himself as the executor. The Action then accesses the HungerComponent of the Agent and increases the value, signalling that the entity has eaten something. The logic of how the hunger value is adjusted is on the EatAction. This would mean that this action has a centralised logic.

Unity or entity component implementation

Everyone who is familiar with Unity might find the current structure familiar. With components on entities, it is close to components on game objects that can be accessed with GetComponent. During the first part of implementation this structure used an individual approach. Instead of using Unity's component system a new entity component class was created and components would not be scripts on game objects, but directly initialised through code in the entity. To reduce the time of finding a component, each component has an ID. To find the correct component, integers are compared instead of object types. But the question still was open, is the new implementation faster?
With the help of a performance testing extension of Unity (Unity, n.d.) some tests were created.
A game object was created with six unity components on it. One of them was an Entity that contained an equal amount of the new entity components. The test performs 1000 GetComponent calls, once on the Entity with the entity components and once on the gameobject of Unity.
Three test cases were written. In the first test case the sixth component was seleted. In the second test case the first component and in the last test case the third component was the serach parameter of the GetComponent method.
The results are as follows:

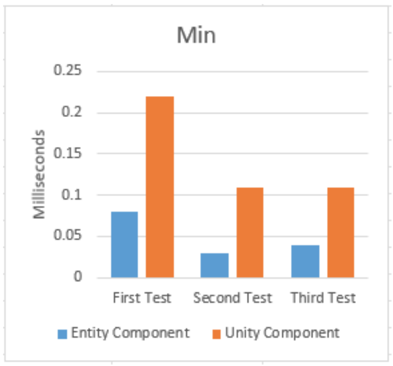


Figure 22 Component test min

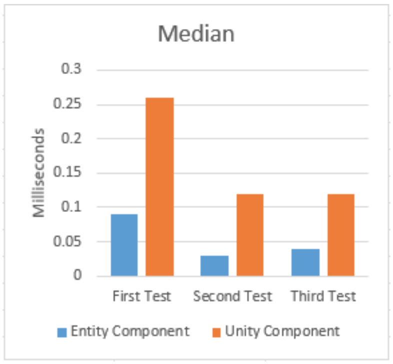


Figure 23 Component test median

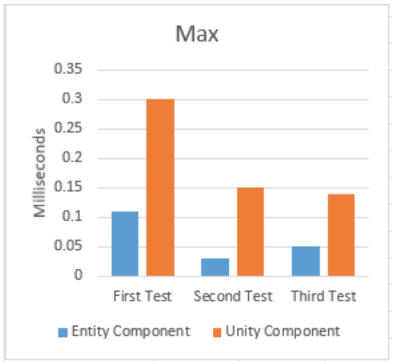


Figure 24 Component test max

In every test the median of the entity component is between 2.8 and 4 times faster than the unity component. (See Appendix B for raw data and test setup)

Table 2 Component Pros/Cons comparison

	Entity component	**Unity component**
Pros	Performance	Build into Unity Simple add / remove and edit directly in Unity Editor Easier to debug. Finds child classes and works with interfaces
Cons	Initialisation must go through Entity if value should be editable in Unity. This increases the size of entities.	Performance

As seen in the above table the most important pros and cons considered in this project are performance and simplicity to edit components. While the entity component clearly has a performance advantage, the unity components are a core feature of Unity with many benefits.

The final solution is to make every entity component a unity component (MonoBehaviour). At the start each unity component that is an entity component will be loaded by the entity and put into a list. During runtime the entity component version of GetComponent will be used. With this approach the performance benefit of the entity component can be used, but every component can be added, removed, and edited, like a unity component, directly in the editor. This will result in a longer load process at the start but will result in better performance during the game, which is preferred.

Smart objects & Smart items

A smart object is an entity that contains actions in addition to components but is not able to execute actions itself. It contains information as to what actions can be performed with this object and allows the agent to execute actions based on the object as introduced in Sims (Champandard, 2017).

A smart item is a reference to a smart object that does not exist in the world.

In video games there is the possibility to pick up objects in the world and place them in the inventory. If an object is picked up, it disappears from the world and is not visible anymore nor does it interact with the world. However, the owner of the item could still use it. The only thing that exists is a reference to the object. It is mostly displayed as an item in the inventory. If an item gets thrown on the ground, the object appears again and interacts with the world.

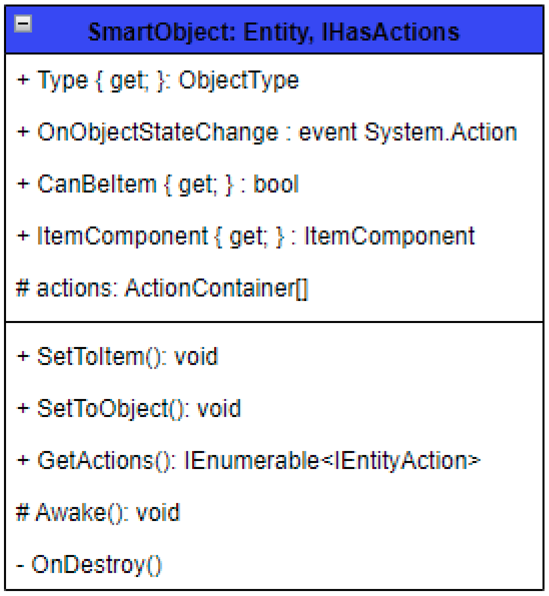

SmartObject: Entity, IHasActions
+ Type { get; }: ObjectType
+ OnObjectStateChange : event System.Action
+ CanBeItem { get; } : bool
+ ItemComponent { get; } : ItemComponent
# actions: ActionContainer[]
+ SetToItem(): void
+ SetToObject(): void
+ GetActions(): IEnumerable<IEntityAction>
# Awake(): void
- OnDestroy()

Figure 25 SmartObject class diagram

EntityActions

EntityActions are an extension to actions and contain additional methods like Init, CanBeExecuted check and properties to allow further functionalities.
The code of an Action is completely independent of a specific object. Each action has to be initialised with the object that it is executed on, and then be executed with the object that actually performs the action. E.g. an EatAction would be initialised with an apple and the human who eats the apple would be the parameter in the Actions execute method. There are exceptions like a MoveForward action where the executioner is the same as the initialiser. Therefore, the action is not chained to a specific object and can be reused by others. To take advantage of this, pooling is used to minimise the creation of new actions and to reuse existing ones (Nystrom, 2009).
The action can be in three different states, first, uninitialised, next, initialised, and finally, executed. During the first, the action does not know on what or by whom it is executed. Once initialised it knows on what or where it is executed, and during execution it also knows who executes the action. Each state except execution will be pooled.
If an AI now wants to eat an apple, it requests the EatAction from the apple. First the apple will check if an EatAction exists in its local pool, which is already initialised. If not, it requests one from the factory. The factory as well will check if it has uninitialised EatActions. If so, it will return one of those and the apple will initialise it with itself. Only if the factory also does not have any EatAction in the pool, a new one will be created. Through this there will never be more actions created than necessary. A step to improve this would be a cleanup of the pools after an action was used very often but hardly any more now.
The internal structure looks like in Figure 28

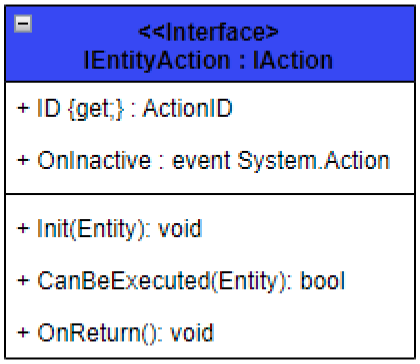


Figure 26 IEntityAction interface diagram

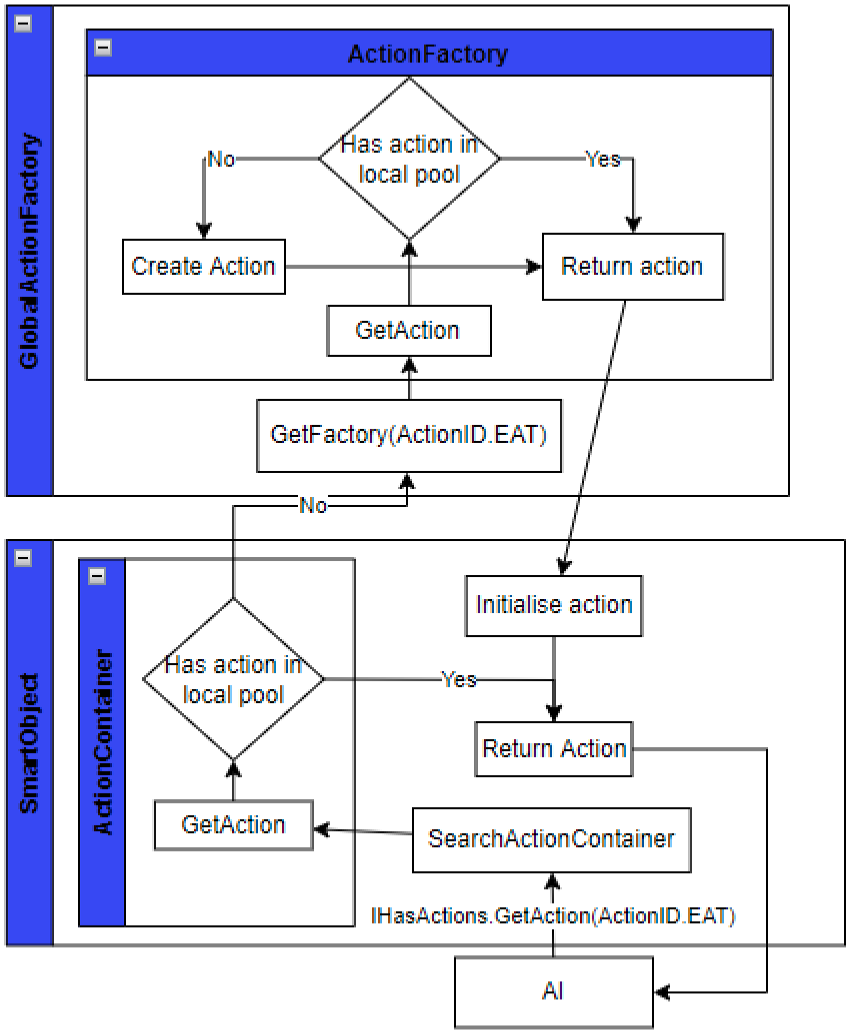


Figure 27 Action factory workflow

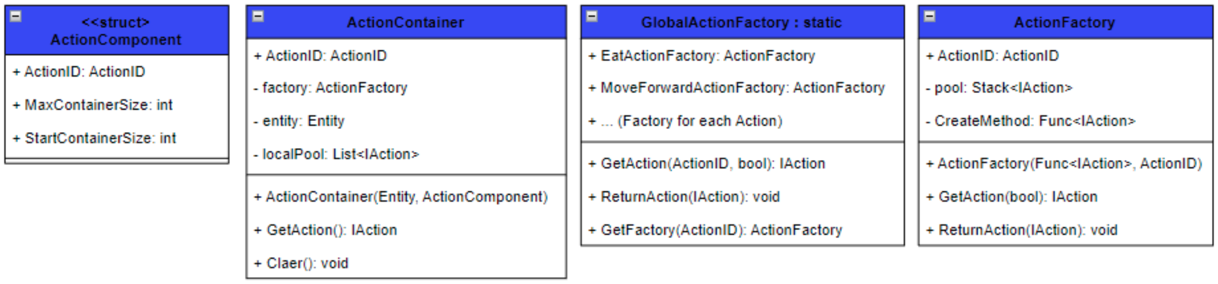

<<struct>> ActionComponent
+ ActionID: ActionID
+ MaxContainerSize: int
+ StartContainerSize: int
ActionContainer
+ ActionID: ActionID
- factory: ActionFactory
- entity: Entity
- localPool: List<IAction>
+ ActionContainer(Entity, ActionComponent)
+ GetAction(): IAction
+ Claer(): void
GlobalActionFactory : static
+ EatActionFactory: ActionFactory
+ MoveForwardActionFactory: ActionFactory
+ ... (Factory for each Action)
+ GetAction(ActionID, bool): IAction
+ ReturnAction(IAction): void
+ GetFactory(ActionID): ActionFactory
ActionFactory
+ ActionID: ActionID
- pool: Stack<IAction>
- CreateMethod: Func<IAction>
+ ActionFactory(Func<IAction>, ActionID)
+ GetAction(bool): IAction
+ ReturnAction(IAction): void

Figure 28 Action factory UML

Test environment description

The test environment consists of two agents, an enemy called Blob and a few objects to interact with. (See Appendix C for a list of all the objects and properties)

Testing

To verify the functionality of the test agent, a series of white box tests are performed in which the result is evaluated by observation and variable checks. For internal code structure validation, a set of Unit-tests are created whose output is a simple Boolean value to check if it succeeded or failed (Homès, 2013). To verify the value of the structure a qualitative study is performed in which the structure is discussed with people who have experience in AI programming in games.

All white-box test cases were successful (See Appendix D for documentation). Unit tests were written to test the ActionRingBuffer and the ActionPooling. All tests run successfully.

Based on the data from the qualitative study the structure presents a very powerful base tool for AI programmers and because of very efficient implementations as the action ring buffer presents a great tool or plugin for large scale mobile applications. There might be a downside when marketing the structure since it is niche and with the vast amount of AI plug and play plugins in current storefronts there are simpler solutions when implementing a basic game AI. Nonetheless, it has the potential to be a very powerful AI tool. (Appendix E for raw data)

Conclusion

How to combine the strengths of current Artificial Intelligence techniques and extend them to support non-interruptible actions. As discussed by Anguelov in his paper (Anguelov, 2020) and talk (GDC, 2018), BTs are often misused for decision-making. He proposes an extension of the standard AI agent model, where the BT is placed between decision-making and actuation layer.

This project follows up on this idea and creates a new structure with four layers instead of three. To find a suitable technique for each layer, several current AI techniques were analysed, and their strengths compared to the needs of each layer. The actuation layer was separated from the AI to support non-interruptible actions. As a result, uninterruptible actions do not get in conflict with the AIs need to be responsive to changes in the environment.

This project found a way to modularise the different parts of an AI and emphasises the single responsibility principle by studying different AI techniques and finding a suitable layer based on the strengths of each technique. It avoids the common misuse of BTs and can handle non-interruptible actions without interfering with the AI logic. The structure contains an outer framework to represent the layers and their interaction without defining specific techniques to use and can be the base for every AI. Each layer then contains an AI technique appropriate to its needs. The AI techniques are interchangeable depending on the requirements of the AI or developer, which enables this project to work as a base for various AIs in games.

This project questions the current implementation of AIs with one single monolithic BT or state machine. It motivates people to think about combining techniques to create something more powerful and with a focus on the single responsibility principle. This will hopefully lead to an examination of current AI techniques and how to improve the use of them, as well as how to avoid pitfalls and weaknesses.

However, the structure created in this project is in no way the solution for everything and might be overkill in some instances or BT might not be appropriate for the planner layer. As different as games are, an individual solution is needed for each one. This structure can help point developers in a different direction or provide a framework. It might not work or be suitable in the exact way it is presented and implemented in this project. This project also only looks at five AI techniques. With further research other techniques can be analysed and placed in one of the layers.

Reference list

Anguelov, B. (2020). *Behavior Trees – Breaking the cycle of misuse*. [online] Taking Initiative. Available at: https://takinginitiative.wordpress.com/2020/01/07/behavior-trees-breaking-the-cycle-of-misuse/ [Accessed 29 May 2023].

Aversa, D., Sithu Kyaw, A. and Peters, C. (2018). *Unity artificial intelligence programming : add powerful, believable, and fun AI entities in your game with the power of Unity 2018!* Fourth ed. Birmingham, Uk: Packt Publishing, pp.172–189.

Candy, L. (2006). *Practice Based Research: A Guide*. [online] Available at: https://www.creativityandcognition.com/wp-content/uploads/2011/04/PBR-Guide-1.1-2006.pdf.

Champandard, A.J. (2017). *Living with The Sims' AI: 21 Tricks to Adopt for Your Game | AiGameDev.com*. [online] web.archive.org. Available at: https://web.archive.org/web/20190401215457/http://aigamedev.com/open/review/the-sims-ai/ [Accessed 6 Jun. 2023].

Dill, K., Pursel, E., Garrity, P., Fragomeni, G. and Martin, L. (2012). *Design Patterns for the Configuration of Utility-Based AI*. [online] Available at: https://course.ccs.neu.edu/cs5150f13/readings/dill_designpatterns.pdf.

Game Maker's Toolkit (2018). *The Rise of the Systemic Game | Game Maker's Toolkit*. *YouTube*. Available at: https://www.youtube.com/watch?v=SnpAAX9CkIc.

GDC (2017). *Goal-Oriented Action Planning: Ten Years of AI Programming*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=gm7K68663rA&t=2209s [Accessed 27 May 2023].

GDC (2019). *AI Arborist: Proper Cultivation and Care for Your Behavior Trees*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=Qq_xX1JCreI&t=1844s [Accessed 14 May 2023].

Girard, S. (2021). *Postmortem: AI action planning on Assassin's Creed Odyssey and Immortals Fenyx Rising*. [online] Game Developer. Available at: https://www.gamedeveloper.com/programming/postmortem-ai-action-planning-on-assassins-creed-odyssey-and-immortals-fenyx-rising- [Accessed 8 Jun. 2023].

Homès, B. (2013). *Fundamentals of Software Testing*. London: Wiley.

Isth3reno1else (2022). *Different Kinds of Animation Cancels and How to do Them (For Beginners)*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=1mcGyiYvUOE [Accessed 7 Jul. 2023].

Lake, A. (2011). *Game programming gems 8*. Boston, Ma: Course Technology/Cengage Learning.

Millington, I. (2019). *AI for Games, Third Edition*. CRC Press.

Nystrom, R. (2009). *Object Pool · Optimization Patterns · Game Programming Patterns*. [online] Gameprogrammingpatterns.com. Available at: https://gameprogrammingpatterns.com/object-pool.html.

Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.

Rabin, S. (2017). *Game AI Pro 3 : collected wisdom of game AI Professionals*. Boca Raton, Fl: Crc Press, Taylor & Francis Group, pp.115–124.

Rabin, S. (2023). *Game AI Pro*. A K Peters/CRC Press, pp.113–126.

Russel, S. and Norvig, P. (2020). *Artificial Intelligence : A Modern approach.* 4th ed. Prentice Hall.

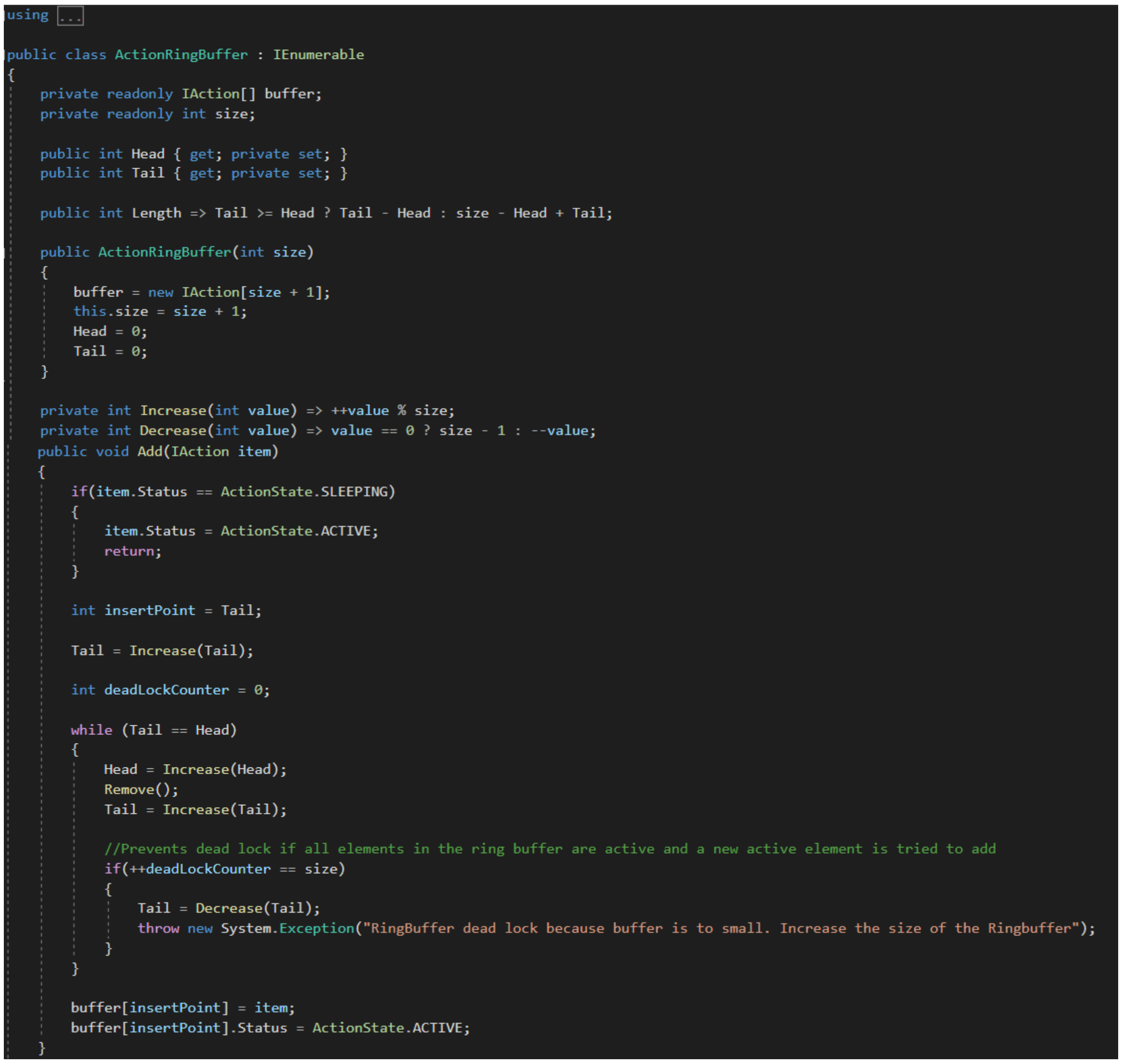
Unity (n.d.). *Performance Testing Extension for Unity Test Runner | Package Manager UI website*. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Packages/com.unity.test-framework.performance@1.0/manual/index.html [Accessed 10 Jun. 2023].

Ward, B. (2020). *Circular Queue or Ring Buffer*. [online] Medium. Available at: https://towardsdatascience.com/circular-queue-or-ring-buffer-92c7b0193326.

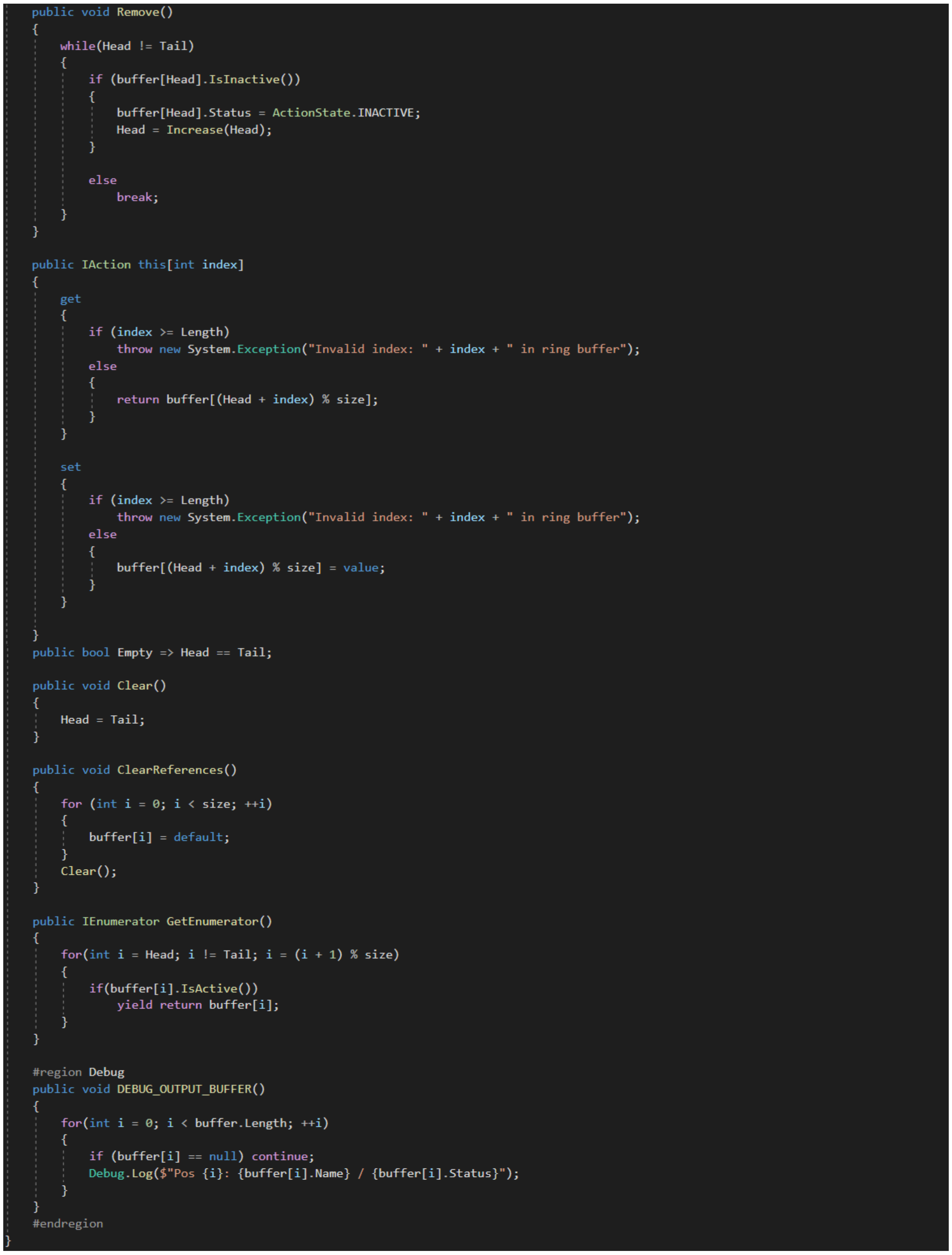
Zubek, R. (2010). *Needs-Based AI*. [online] Available at: http://robert.zubek.net/publications/Needs-based-AI-draft.pdf.

Appendices

Appendix A

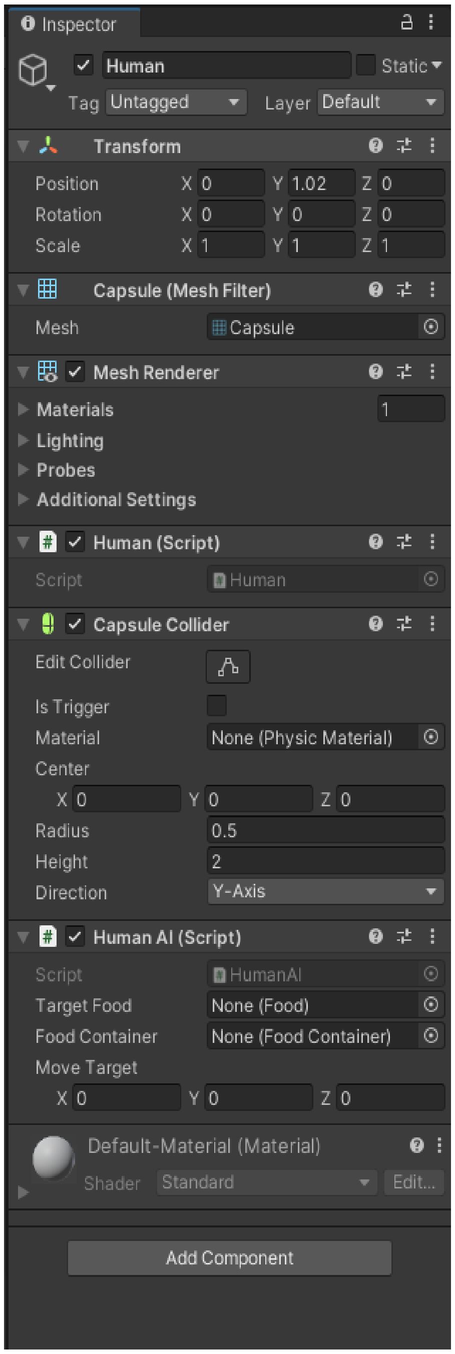


```
using ...

public class ActionRingBuffer : IEnumerable
{
    private readonly IAction[] buffer;
    private readonly int size;

    public int Head { get; private set; }
    public int Tail { get; private set; }

    public int Length => Tail >= Head ? Tail - Head : size - Head + Tail;

    public ActionRingBuffer(int size)
    {
        buffer = new IAction[size + 1];
        this.size = size + 1;
        Head = 0;
        Tail = 0;
    }

    private int Increase(int value) => ++value % size;
    private int Decrease(int value) => value == 0 ? size - 1 : --value;
    public void Add(IAction item)
    {
        if(item.Status == ActionState.SLEEPING)
        {
            item.Status = ActionState.ACTIVE;
            return;
        }

        int insertPoint = Tail;

        Tail = Increase(Tail);

        int deadLockCounter = 0;

        while (Tail == Head)
        {
            Head = Increase(Head);
            Remove();
            Tail = Increase(Tail);

            //Prevents dead lock if all elements in the ring buffer are active and a new active element is tried to add
            if(++deadLockCounter == size)
            {
                Tail = Decrease(Tail);
                throw new System.Exception("RingBuffer dead lock because buffer is to small. Increase the size of the Ringbuffer");
            }
        }

        buffer[insertPoint] = item;
        buffer[insertPoint].Status = ActionState.ACTIVE;
    }
```



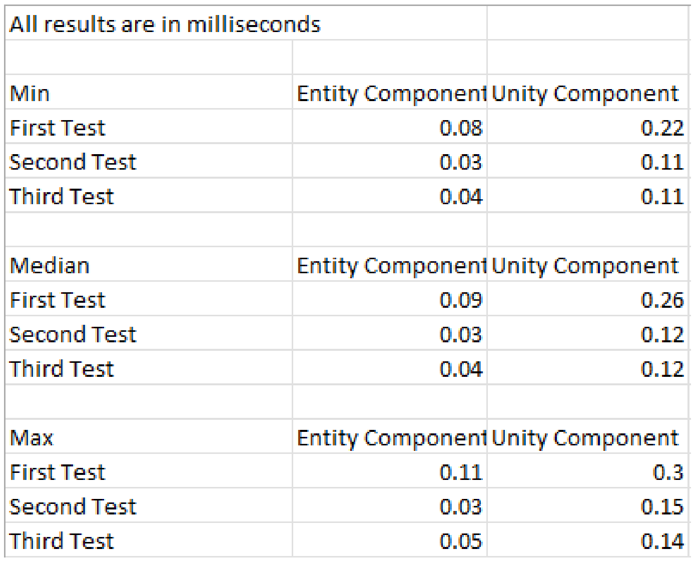
```
    public void Remove()
    {
        while(Head != Tail)
        {
            if (buffer[Head].IsInactive())
            {
                buffer[Head].Status = ActionState.INACTIVE;
                Head = Increase(Head);
            }

            else
                break;
        }
    }

    public IAction this[int index]
    {
        get
        {
            if (index >= Length)
                throw new System.Exception("Invalid index: " + index + " in ring buffer");
            else
            {
                return buffer[(Head + index) % size];
            }
        }

        set
        {
            if (index >= Length)
                throw new System.Exception("Invalid index: " + index + " in ring buffer");
            else
            {
                buffer[(Head + index) % size] = value;
            }
        }

    }
    public bool Empty => Head == Tail;

    public void Clear()
    {
        Head = Tail;
    }

    public void ClearReferences()
    {
        for (int i = 0; i < size; ++i)
        {
            buffer[i] = default;
        }
        Clear();
    }

    public IEnumerator GetEnumerator()
    {
        for(int i = Head; i != Tail; i = (i + 1) % size)
        {
            if(buffer[i].IsActive())
                yield return buffer[i];
        }
    }

    #region Debug
    public void DEBUG_OUTPUT_BUFFER()
    {
        for(int i = 0; i < buffer.Length; ++i)
        {
            if (buffer[i] == null) continue;
            Debug.Log($"Pos {i}: {buffer[i].Name} / {buffer[i].Status}");
        }
    }
    #endregion
}
```

Appendix B

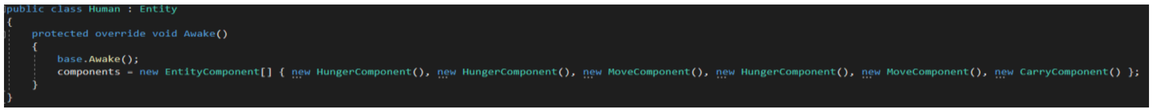


```
Inspector
Human                         Static
Tag Untagged          Layer Default

Transform
Position    X 0    Y 1.02   Z 0
Rotation    X 0    Y 0      Z 0
Scale       X 1    Y 1      Z 1

Capsule (Mesh Filter)
Mesh        Capsule

Mesh Renderer
Materials                 1
Lighting
Probes
Additional Settings

Human (Script)
Script      Human

Capsule Collider
Edit Collider
Is Trigger
Material    None (Physic Material)
Center
  X 0    Y 0    Z 0
Radius      0.5
Height      2
Direction   Y-Axis

Human AI (Script)
Script          HumanAI
Target Food     None (Food)
Food Container  None (Food Container)
Move Target
  X 0    Y 0    Z 0

Default-Material (Material)
Shader   Standard      Edit...

Add Component
```

In the first test the last component in the list was asked for. So CarryComponent and HumanAI. In the second test the first component: HungerComponent and Transform and in the third test the third component: MoveComponent and MeshRenderer.

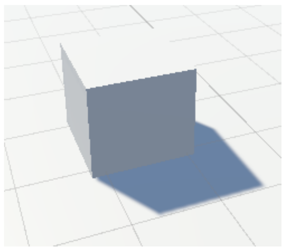


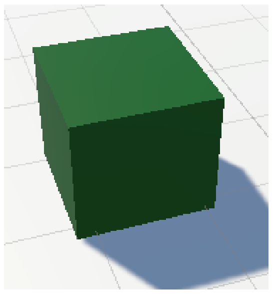
All results are in milliseconds		
Min	Entity Component	Unity Component
First Test	0.08	0.22
Second Test	0.03	0.11
Third Test	0.04	0.11
Median	Entity Component	Unity Component
First Test	0.09	0.26
Second Test	0.03	0.12
Third Test	0.04	0.12
Max	Entity Component	Unity Component
First Test	0.11	0.3
Second Test	0.03	0.15
Third Test	0.05	0.14



```
public class Human : Entity
{
    protected override void Awake()
    {
        base.Awake();
        components = new EntityComponent[] { new HungerComponent(), new HungerComponent(), new MoveComponent(), new HungerComponent(), new MoveComponent(), new CarryComponent() };
    }
}
```

Appendix C



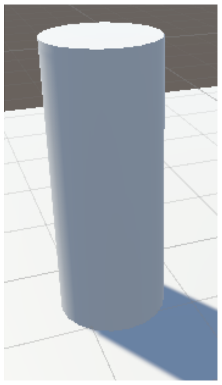


Food container

Food can be put into and taken out of this container.

Food in the food container is also safe from blobs.

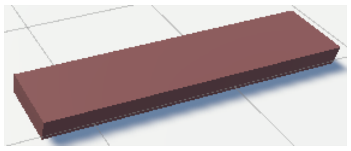
The colour will change from black to light green the more food is in there. The maximum capacity of the container is ten.



Tree

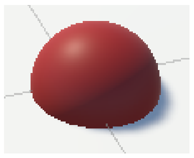
A tree can be cut down to get wood.

A tree has 20 health points and will drop two pieces of wood.



Wood

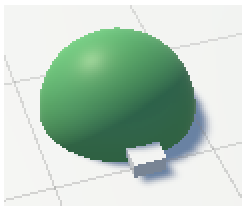
Wood can be picked up and is used to build a house.



Food

Food can be picked up or eaten. When eating food, the food metre of an agent is increased by 15.

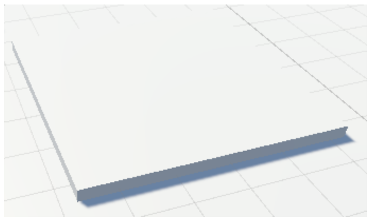
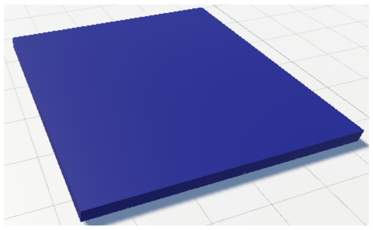
It can also be put into the food storage.



Blob

The blob will eat away food that lies on the floor.

The blob can be killed and has 20 health points.

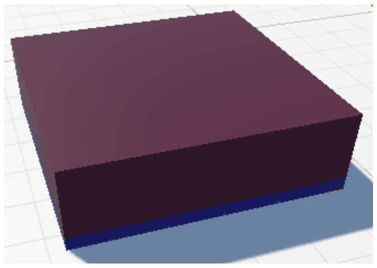
House construction site

Wood can be transported to the construction site, the more wood is put into the construction site, the bluer the object becomes. If four pieces of wood are placed, the house can be built. After four build actions the house is finished.

Agent

The agent will eat when he is hungry. He eats either food from the floor or from the food container. If he is not hungry, he will pick up the food and bring it to the container. If there is no food, he will work on the house by first gathering enough wood, which might require him to cut a tree, and afterwards build the house. If he spots a blob, he will immediately chase and try to kill him.

The model and animations are from Mixamo: https://www.mixamo.com/#/



House

The house after it is built. It has no additional purpose or properties.

Appendix D

White-box test cases

Each white box test was performed in a separate scene which exists in the project and is named: TestCase + test case number. The title of the test case consists of the number, name, and the background functionality it is tested with.

Test case	1) Agent eats food, Decision modifier
Preparation	Only two decisions are active. One to do nothing with a static utility value of 0.5 and one for eating with a linear function of $1 - \frac{hunger}{100}$ and success modifier of 0.15 for 0.5 seconds.
Expected result	The agent stays still until his hunger metre falls below 50. He then moves to the nearest food and eats it. By eating the food his hunger metre increases by 15 (the value set in the food). He then immediately goes to the second food and eats this as well. After eating the second food he will stay still again until his hunger metre falls below 50 and then eat the third food.
Test result	The agent eats one food and stays still until the hunger metre falls below 50.
Evaluation	The eat decision function did not include the modifier. The function was changed to $mod + 1 - \frac{hunger}{100}$ the time of the success modifier was increased to 1.5 seconds as its influence was too weak. After adjusting these values and rerunning the test, the test results were equal to the expected ones.

Test case	2) Agent builds house, Behaviour tree
Preparation	Set the strength property of the agent to 0.5
Expected result	The agent picks up two pieces of wood and brings them to the construction site. After that he cuts down the nearest tree and brings the two-wood spawned by the tree to the construction site. The agent hits the tree four times: $\frac{tree\ health}{agent\ attack\ damage} = \frac{20}{5}$. He then executes four build actions before the house is finished: $\frac{needed\ build\ progress\ by\ house}{build\ progress\ per\ build\ action\ by\ agent} = \frac{20}{10 * agent.strength}$
Test result	The test result is exactly as expected.
Evaluation	The test was successful. But it might fail if the hunger metre falls below 60 before the house gets finished since the build decision has a static utility of 0.4. To remove this thread, enable only the build decision. This action was not carried out because the test case did show that the agent is fast enough.

Test case	3) Blob appears, State transaction
Preparation	Same as test case 2. Add a spawner that spawns a blob after 8 seconds.
Expected result	When the blob spawns, the agent should immediately walk in its direction and attack it. The blob will vanish after four hits by the agent. $\frac{health\ of\ blob}{agent\ attack\ damage} = \frac{20}{5}$. After defeating the blob, the agent should return to building the house.
Test result	The agent immediately stops and starts to move towards the blob and attacks it. After defeating the blob, he returns to building the house.
Evaluation	Test successful.

Test case	4) Interrupting blob, Non-interruptible actions & action layers
Preparation	Same as test case 2. Add a spawner that spawns a blob after 20.7 seconds. The time is set exactly to the moment when the agent is executing a build action that is non-interruptible
Expected result	When the blob spawns, the agent should immediately turn in the direction of the blob but not move until the whole build animation is finished. After the animation is finished, he should move to, attack, and kill the blob. After the blob is gone, he returns to building the house.
Test result	When the blob spawns, the agent turns in its direction but does not move until the whole animation is finished. He then kills the blob and returns to building the house.
Evaluation	Test successful. It also shows that the action layer system works since turning and building actions are on different layers but moving and building share at least one layer.

Appendix E

Feedback from Farhan Ahmed

Actions depending on layers is great, also the separation between actions and AI (BT) seems efficient in terms of workflow and replication. The actionringbuffer customised to the needs seems to be something to be applicable to a lot of different kinds of projects, characters, and behaviour. This whole project seems to be efficiently applied to large-scale mobile games when thinking as a plugin or AI base. It is quite niche, which could work badly in terms of marketing, especially in the current storefronts for developers e.g. plug and play plugins, although, I have yet to see a plug and play plugin/project that does this. This has the potential to be a very powerful AI tool.