Unity GOAP Tool

Bachelor's Thesis

Video Game Design and Development Degree

Student: Martín Vilà, Ferran

2019

Director: Zuñiga Zarate, Ana Gabriela

Index

Summary

Unity is the most used real-time 3D (RT3D) engine all over the world, with more than 4.5M registered developers. Nowadays Unity engine still don't have an officially supported tool for the development of artificial intelligence (AI) using the goal oriented action planning (GOAP) system. In order to solve this lack, some developers have created its own tools to generate GOAP based AI. Some of this tools are initially developed to create video games or any AI related project and later published on a git platform when the project is released. Some others are directly published on the assets store of unity.

Figure 0.1 Unity Logo

What all these mentioned tools have in common is the limited user interface (UI) interactions. In this kind of tools a limitation of UI elements can be an important usability barrier, because uses need UI elements that let them modify and generate all kind of planning data (conditions, action, effects, ...) and UI elements that display a clear output of the resultant behavior. Otherwise, users will not have enough information about why the AI is not having the desired behavior and will immediately stop using the tool.

The objective of this project is to compete with the existing unofficial tools, with a completely new one that let users generate GOAP based AI easily and fast. This tool will support a worked system of visual editors, with all the necessary UI elements to generate and modify everything related with the AI planning, and a friendly clean code easy to work with.

In order to achieve all the mentioned objectives, we will use an agile methodology that will let us prototype and test all the project functionalities.

Keywords

artificial intelligence (AI), goal oriented action planning (GOAP), Unity

Links

Github <https://github.com/ferranmartinvila/Unity-GOAP_S>

Tables Index

Figures Index

Glossary

UI - User Interface: Composed by all the displayed elements that let the user interact with the computer and backwards. Human use UI elements to send input to the computer and it generates an output from the received information.

AI - Artificial Intelligence: Area of computer science that focus on simulate human behaviours like learning and problem solving on machines. Nowadays artificial intelligence is used in an infinity of fields: Autonomous cars, video games, economics, etc.

FSM - Finite State Machine: Computation model used to represent and control different states execution. Mainly used to generate AI in video games, because this model can generate good behaviours with a basic and optimal implementation of nodes and transitions.

BT - Behaviour Tree: Computation model that describes a structure of hierarchical nodes. Each node represents an action and hierarchy connections are conditions to execute the connected actions. This model provides more flexible results than finite state machines, that is the reason why now a days is the mostly used to generate artificial intelligence in video games.

GOAP - Goal Oriented Action Planning: Refers to a planning architecture that generates autonomous agent behaviors in real-time. Agents follow a dynamically planned sequence of actions, what derives in an interesting number of different ways to reach the same objective, in other words, more randomness on the agents behavior.

RT3D - Real-time 3D: Sub-field of computer graphics that process computer-generated images in real-time. To be considered a real-time system, images must be rendered in less than 1/30 seconds.

Unity - Cross-platform real-time 3D engine published by Unity Technologies in 2005. Now a days is the more used engine for video games development. A curious fact to imaginate how enormously big is the Unity users family is that, during 2017 Unity was installed about 24 million times.

Unreal Engine - Cross-platform real-time 3D engine published by Epic Games in 1998. The actual version named Unreal Engine 4 was launched in 2005 and have more than 6.3 millions users.

1. Introduction

1.1 Motivation

There are two principal motivations that made me decide to work on this project:

- **Deep in AI** and the subsystems that involve it:

  During the degree we have learned about AI working with other Unity tools, like Node Canvas to generate behavior trees (BT) and finite state machines (FSM), that tools show me how useful can be a good AI generation system and the amazing results you can obtain with it. So, I decide to develop an AI generation tool that supports GOAP architecture to learn more about this AI system and the world behind tools development.

- Work and learn with the **Unity engine ecosystem**:

  On a previous subject we develop our game engine in C++. I have a little idea of the basic systems a game engine must support and the hard work necessary to develop them, the second motivation point comes from learn about Unity engine subsystems and all the platforms that involve them, like scripting, user interface (UI) generation, assets store, etc.

1.2 Problem formulation

Actual AI systems have an important barrier related with the performance, mainly because of its unpredictability and highly-variable performance. Till today, the main used AI architectures for real-time projects like videogames are FSM and BT, principally due they relative constant performance.

But what happens with GOAP architecture? GOAP is the most unpredictable of the mentioned methodologies, therefore the less used. The high performance randomness of this structure is due the large planning variants an agent can generate to reach the same objective.

This don't mean that this architecture has been never used to create really interesting AI behaviors in real-time projects, there are several video games that use GOAP architecture, for example: Silent Hill: Homecoming (2008), Just Cause 2(2010), Trapped Dead (2011), etc. But all this projects have something in common, all have been developed by large teams with enough time and resources to produce a GOAP system adapted to its videogame. Therefore, this is a second barrier of using this architecture, the resources that you need to create the AI system.

Figure 1.1 Silent Hill Homecoming portrait

Figure 1.2 Just Cause 2 portrait

Computers performance increase day by day and let developers generate bigger projects with more complex systems that require more resources, AI can benefit of this and be one of this more complex systems.

At the present time we have the necessary performance to create a GOAP system adapted to our project and generate relatively basic agent behaviors in real-time, furthermore we have enough performance to develop a generic GOAP system tool that generate these agent behaviors. This is what AI interested developers have start doing, after a short research we can find several GOAP tools adapted to different formats like Unity, Unreal Engine, C++ code with no dependencies, etc.

If we focus on Unity, there's only two GOAP tools on the assets store, "ReGoap" and "Goal Oriented Action Planning (Artificial Intelligence)", both of them are focused on coding and don't have a complete UI support with all the necessary editors. There's also the option to search on google for other GOAP tools adapted to Unity, we will find some really good tools like the mentioned previously, but the UI limitation problem appears again.

This is not a huge problem that don't let you implement a GOAP system in your Unity project, you can do it perfectly if you have the time and resources to spend on it. Therefore why we are developing another GOAP tool for Unity? Because this project is not only focused on crate a basic GOAP tool, the final objective is to develop a tool with all the UI support that the user needs, and give him the option to develop his AI without spending the time and resources that it will probably spend if uses another tool with a limited UI.

1.3 General objectives

This are the general objectives that we want to reach at the end of this project:

- Solid and functional GOAP logic
- UI support for all the tool elements
- Generate a useful process output
- Provide a complete documentation

1.3.1 Solid and functional GOAP logic

This is the main general objective, from which all the other objectives derive and depend to be reachable. GOAP logic is the base of the tool we aim to develop, without a properly implemented system that handle it the tool functionality is null.

Once we research about GOAP system and implement its architecture to our tool, not all logic work is done. This architecture has a high performance variability and we can't avoid it without limiting several tool fields, like the number of agents or the available actions, but limitation is not an option for a tool that must let the user generate his AI ecosystem freely.

Then, the correct way is optimization. To improve this tool functionality, we need to reach the best performance possible and build a solid GOAP logic able to support the most exigent projects.

1.3.2 UI support for all the tool elements

To get a comfortable experience using the tool we should provide UI support for all the elements that compose it. There are several logic structures inside the GOAP architecture: agents, behaviours, actions, variables, etc; Each of this structures must have an adapted UI that displays the most important information contained in.

A part of logic structures, there are other elements that we should take into account if we want to generate the previously mentioned comfortable experience. This elements are all the different states supported by the tool architecture, for example each logic structure in the tool have at least an edit and a set state. When an element is in edit state user can modify different fields than in set state, therefore UI must adapt to the state of the element displayed.

All the generated UI elements will compose editors and menus that users will use to build their AI system. Finally, to improve user interaction, all the UI displays must follow a similar architecture. This will help users to have a more structured idea of the elements that compose the tool.

1.3.3 Generate a useful process output

A software tool can be perfectly structured and have an extremely optimized performance and keep being a bad tool. This is because in software development the tool is not the only element that must be efficient, the user working with it must be efficient too. In this GOAP AI tool users have to program their own action and behavior scripts to generate custom agents.

We can't transform users with zero knowledge about software development into masters, but we can help them giving all the necessary information about what is happening and why.

Here is when comes out output and debugging. Debugging consists in detect and remove program errors, this is a basic and necessary routine process in software development that all programming tools must support. To let users debug their scripts comfortably and get efficient results what we need to do is, generate a useful output about what is our tool doing each frame and why.

1.3.4 Provide a complete documentation

There's always the possibility that users don't understand properly how to work with one or several tool components, this can generate tool functionality limitation. We need that all the users understand perfectly our tool components and how to use them to avoid that situation.

Document all the information about the tool that we consider important and share it with the users is a good way to deal with that problem.

The generated documentation can't be limited to a theoretical field, it must contain practical examples with comments. Additionally we will develop example scenes where all the tool components are used correctly, this scenes will be also a great test field for the users.

1.4 Specific objectives

1.4.1 Solid and functional GOAP logic:

- **GOAP logic research and implementation:**

  The first step to implement a new system to your tool its research about it, is necessary to know perfectly how the new system works and all the logic elements that compose it. Once we understand how GOAP logic works externally, the next step is deep in architecture. Deep in architecture means understanding all the core elements that compose the GOAP architecture to finally implement your own. The goal is reached when we have a clear idea about how to structure our code and the programming process ends with a solid and well-structured architecture.

- **Well structured code that let users deep in:**

  Programing projects can scale its size and complexity really fast, in this case we will implement the several logic elements that compose the GOAP architecture and the necessary UI components to support them; It is not a short project that reduces to a pair of classes.

  To reach the desired well-structured project we need to keep a set of programming rules throughout all the development process. At the end, we will share this programming rules in the project documentation to all the users that want to deep in code, helping them to have a clearer idea about how is this project structured.

- **Optimize the most heavy processes of the tool:**

  When all the logic elements and the basic interaction components are implemented its time for testing. To reach this optimization goal, we need to do several tests and detect which are the most critical logic processes of our tool.

  Building test scenes is a good practice to do stress tests. Once we detect a non-optimal logic process, the next step consists in research about how to optimize it and modify the necessary code. We have to bear in mind that this tool will be a part of a larger project developed by the user and we can't use the major part of the available computer resources, therefore optimization is a must.

1.4.2 UI support for all the tool elements:

- **Let user access to all the desired tool variables:**

  A GOAP tool can't generate anything without a previous work of developing the desired behaviors, actions and variables. To implement a functional and comfortable tool we need UI support for all the mentioned logic elements in order to let users generate their AI agents properly. A part of the most core parts of the tool, UI will also support a few customization features, like adding a brief description to the actions and placing them in the desired position of the canvas. The objective is to let users generate its own logic structure inside the GOAP architecture.

- **Keep a logic UI structure:**

  Be able to modify any value of a program using UI is a great feature, but if the UI don't follow a logic structure and users can't use it properly UI becomes useless. This is why this goal is really important and we need to keep it in mind during all the UI developing process. Obviously not all the logic elements of the GOAP architecture have the same structure and variables, but we can structure in very similar editors to help users find the desired attributes fast and work efficiently.

1.4.3 Generate a useful process output:

- **Functional debugging feature:**

  As we mentioned before, debugging is a must in a software development tool. This specific goal consists in investigate which are the main logic parts that users need to track during the program execution and then generate all the necessary UI elements to do it. To implement a solid and easy to use debugging system we have to follow a basic set of rules for all the UI elements that compose the system. For example, read highlighted elements are not working properly, yellow ones are idle, etc.

- **Let users customize process output:**

  Sometimes users don't adapt to the UI informative colors and labels that translate the logic state of the program, or simply they don't like it. To avoid it and improve the user experience, we will support a few UI customization features like change the UI debugging colors and alert messages.

1.4.4 Provide a complete documentation:

- **Example scenes:**

  When users work for first time with a tool they can easily lose through all the functionalities and components provided. An example scene can be a really good practice to avoid that situation. We will build that example scene putting on it all the components offered by the tool, in order to show each component and how to use it.

- **Detailed documentation:**

  In any kind of software, documentation is a must. In this case, documentation will provide a complete theoretical explanation of the tool architecture and all the elements that compose it. Additionally, documentation will include several practice examples of the tool components been used properly.

- **Constant acces to the information:**

  To improve the users experience, documentation will be present in all the working platforms supported by the tool. By “working platforms” we mean all the software ecosystems in which the user interacts with the tool or the tool content. Unity engine, the code editor and the web page are the platforms we are talking about. Unity engine will have tooltips in the tool UI elements, code will be commented conscientiously and finally, in the web page you can access to the whole documentation.

1.5 Project range

This tool doesn't pretend to be mainly educative, the main objective is functionality. In the documentation users can find a brief explanation of how GOAP AI works and all the necessary examples to use this tool properly; This is all the educative content shared with the users.

That means that is focused to users with a previous programming and AI education. For the users that don't want to deep in the tool logic structure, the education needed to use it superficially is practically null.

Otherwise, if users need different functionalities than the provided by the tool, they can deep in code and modify it, generating a versioned tool adapted to their project. To do this, users need to have a solid knowledge about GOAP architecture and all the programming techniques used during the tool developing process.

In conclusion, this tool audience is inside a really wide range of developers. From the most beginner software developers with basic knowledge about programming and AI, to experienced developers that know perfectly how GOAP architecture works and the most complex programming techniques.

2. State of the art

2.1 Origin

Goal-Oriented Action Planning is a simplified version of the Stanford Research Institute Problem Solver (STRIPS) planning architecture. It was originally implemented for video games by Jeff Orkin, in order to generate autonomous behaviours in a real-time application. First video games using this system for AI planning were F.E.A.R and No One Lives Forever 2, both implemented by GOAP architecture creator.

In 2005, when F.E.A.R project reach the gold state, its AI system was released. Now the source code is open for any interested developer that wants to use GOAP architecture in its own project. Source Link.

Figure 2.1 F.E.A.R portrait

Figure 2.2 F.E.A.R in game capture

Before the apparition of GOAP architecture, practically all the AI for real-time projects were implemented using FSM and BT architectures. Till today FSM and BT still domain the market, because they are easy to implement and generate good results. Otherwise, the apparition of this award-winning AI system motivates developers to implement its own versions.

Some versions aren't focused to be a fully integrated part of a major project and they can be reused in different software scenes with no dependencies. Other versions are in the mid point between an independent system and a fully integrated one, we are talking about tools integrated in different development platforms that let users generate agent behaviors planned with GOAP. This ones have software dependencies but can be reused to work with any kind of project, meanwhile its supported by the development platform they are integrated in.

Owr tool is inside the last mentioned versions group, has software dependencies with a development platform and can be used to develop any kind of projects supported by it. Therefore, we will mainly focus on research and analyze the existing members of this group. In order to have a clear idea of the actual market state that concerns us.

2.2 Market Study

To develop and offer a competent product we need to study which are our competitors and the features that our product should have if we want to compete with them. It is necessary to keep in mind that our tool is adapted to Unity, therefore the main competitors are other Unity tools or external systems that can be adapted to the engine easily. We also research about GOAP implementations developed for other engines like Unreal Engine, but we finally decide to focus the analisis on Unity related implementations with the objective to know perfectly our most direct competitors.

During the next paragraphs we develop a complete study of the actual market state. In order to have a more clear and well structured idea of it, the analyzed systems are divided in two groups: Non-dependant and Engine-dependant. Obviously, all of them are AI systems that use GOAP for agent planning, but differ in their software dependencies.

The mentioned groups are:

- **Non-dependant:** Independent GOAP implementations that don't have dependencies with other major projects like engines or video games. We don't count code libraries as real dependencies. Not are real direct competitors, because its necessary to adapt them to Unity architecture, but users can consider them if the implementation offers extra features that other GOAP implementations lack.

- **Engine-dependant**: GOAP implementations completely adapted to Unity engine. In this group we find the real competitors that share platform with us. During the engine-dependant implementations we will not only focus on code structure and logic, in this case UI functionality is important. In a full code framework, UI is not extremely necessary because users have programming experience, otherwise in an engine-dependant tool implement a basic UI can help the less code experienced users.

Lets deep in a more concise analysis of the selected GOAP implementations.

2.2.1 Non-dependant systems

In this section we analyze the most interesting GOAP implementations without real code dependences, in other words, can work in any project independently of the software used to develop it. The only barrier in this implementations are the code language; it is obvious that you can't use a set of C# scripts in a C++ project.

GPGOAP:

General Purpose GOAP is a compact and solid C++ implementation of the GOAP architecture by Abraham Stolk, an independent game developer mostly known by his videogame "The Little Crane That Could", played by more than 19 million users.

This GOAP implementation is focused on code and don't have any kind of UI. But in this case UI is not necessary, because the developer has been skilled enough to structure the "how to use" instructions in extremely clear code steps. This GOAP tool internally do all the necessary operations to calculate the final actions planning from all the information provided by the users: available actions, world state, actions conditions, etc.

To understand how this GOAP implementation works and its important features, let's analyze the different "how to use" steps:

- **Initialize action planner:**

  The first step is the most basic, instantiate the planner class to save in it all the information generated during the next steps. This class is basically the brain that uses the information generated by the user to calculate the final actions planning.

```
static actionplanner_t ap; //Allocate planner
goap_actionplanner_clear( &ap ); //Initialize planner
```

Figure 2.3 GPGOAP action planner code

- **Set available actions:**

  Now we can define our set of actions inside the instantiated planner. This GOAP implementation don't works with real defined actions, it only uses preconditions and effects to calculate the final planning. Therefore, in this step users set preconditions and effects and relates them to an imaginary action represented with a string. The conditions and effects values are strings like actions but related with a value. To completely understand this step we have commented a brief code segment

```
//          Planner    Action        Condition      Condition value
goap_set_pre( &ap,     "aim",      "enemyvisible",      true );
goap_set_pre( &ap,     "aim",      "weaponloaded",      true );
goap_set_pst( &ap,     "aim",      "enemylinedup",      true );
```

Figure 2.4 GPGOAP actions code

- **Set current world state:**

The world state is stored in a class and will be used by the action planner allocated in the first step to calculate the planning. To build the current world state we basically add all the desired variables in the world state class, consequently creating our AI scene in which the agents will interact. It is necessary to reference the planner is going to use the world state every time we add or modify a variable.

```
worldstate_t fr; //Allocate world state
goap_worldstate_clear( &fr ); //Initialize world state
//                  Planner   World state       Variable       Variable value
goap_worldstate_set( &ap,        &fr,         "enemyvisible",      false );
```

Figure 2.5 GPGOAP world state code

- **Set goal world state:**

In this step we instantiate and build another world state class to define the goal world state provided by the actions planning. Therefore, this new class have to share one or more variables with the previously created world state; this new instance is basically an evolution of the initial world state.

```
worldstate_t goal;
goap_worldstate_clear( &goal );
goap_worldstate_set( &ap, &goal, "enemyvisible", true );
```

Figure 2.6 GPGOAP goal code

- **Calculate planning:**

Finally, the actions planning is calculated. To calculate it this tool uses A* pathfinding methodology interpreting actions as nodes and actions conditions as walkability conditions. We will see that this is the best known way to generate the planning and all the GOAP implementations use this methodology. Once the planning is calculated, the method that manage the process returns the result in form of string array. Each string is an action name; the agent needs to complete the actions in the specified order to reach the previously defined goal world state.

```
worldstate_t states[16]; //World state at each plan step
const char* plan[16]; //Action plan
int plan_size=16; //Number of actions
const int plan_cost = astar_plan( &ap, fr, goal, plan, states, &plan_size );
```

Figure 2.7 GPGOAP planning code

What we have to keep in mind after analyze GPGOAP is that it is necessary to build a solid code structure with clear steps. In order to let users work focusing on code and avoid UI if they don't like it.

C++GOAP:

As it name says, this is a GOAP architecture implemented in C++. This tool have more features than the previously analyzed one, it also provides a complete example project that shows all the available functionalities.

This tool was developed by Chris Powell, software developer and designer that mainly works with Ruby, Rails and C++.

To analyze this tool, we are going to use the example project code provided by the author. This GOAP implementation is focused on code and don't have any kind of UI that can be analyzed. Therefore, to analyze this project we follow the same previously used structure, based on describe the steps that users should do to work with the tool properly.

Complete step by step analysis of the provided example code:

- **Instantiate actions data:**

  Initially, we have to allocate an array of actions and all the variables that this actions are going to use. Actions are a class defined by the framework and variables are defined using a constant integer that works as variable id. In this case the tool only supports boolean variables.

```
//Actions array
std::vector<Action> actions;
//Variables used by actions
const int target_acquired = 10;
const int target_lost = 15;
```

Figure 2.8 C++GOAP actions data code

- **Build actions:**

  The next step is build the desired actions and save them in the previously instantiated actions array. To define an action, user needs to instantiate an action class and set its preconditions and effects. Preconditions and effects are basically a previously defined variable id and a boolean value. Finally, when the action is defined it is time to store it in the actions array.

```
Action intercept("interceptTarget", 5); //Action name / cost
intercept.setPrecondition(target_acquired, false); //Precondition
intercept.setPrecondition(target_lost, false); //Precondition
intercept.setEffect(target_acquired, true); //Effect
actions.push_back(intercept);
```

Figure 2.9 C++GOAP actions build code

- **Set current world state:**

  This GOAP implementation provides a world state class that contains a map with the variables ids and its boolean values. To build the world state, users can simply enter the previously defined variables ids and its values.

```
WorldState initial_state;
initial_state.setVariable(target_acquired, false);
initial_state.setVariable(target_lost, false);
```

Figure 2.10 C++GOAP world state code

- **Set goal world state:**

  To defined the goal world state, users have to instantiate another world state class and store the desired variables and its goal values. If there is an initial value that don't have to change its value at the end of the planning, it is not necessary to add it in the goal world state. This is an interesting feature non supported by the tool analyzed previously.

```
WorldState goal_state;
goal_state.setVariable(target_acquired, true);
```

Figure 2.11 C++GOAP goal code

- **Calculate Planning:**

  The last step consists in generate the final actions planning, user only need to instantiate a planner class and call its plan method, what will return an actions array if everything is correct. In this case the generated path is not reversed inside the planning method, what means that the array first action it is actually the last action to do. As we mentioned before, all the GOAP implementations use A* pathfinding methodology to generate the planning and this is not an exception.

```
Planner planner;
//              Actions plan              Initial WS    Goal WS  Actions array
std::vector<Action> plan = planner.plan(initial_state, goal_state, actions);
```

Figure 2.12 C++GOAP planning code

Once we have a clear idea of all the features provided by this tool, we can conclude by pointing that this GOAP implementation achieve a clearer and easier to scale structure in relation with the tool analyzed before. This tool let users define real action classes and set its conditions and preconditions inside them, without the necessity to constantly reference the world state like in the GPGOAP provided example. We consider this a structure improvement that needs to be present during our tool development.

2.2.2 Engine dependant systems

In this section, we are going to analyze our real competitors that are actually in the market. The objective is to subtract their more important features, those that invite users to work with them, and their problems, like non supported functionalities, UI editors hard to understand, etc. The research is not limited to the Unity platform, other engines GOAP implementations are commented in this analysis.

First we comment Unity and the other engines cases are commented at the end. In order to compare the different supported features according to the tool development framework.

ReGOAP:

This GOAP library is implemented in C# and can be adapted to Unity. Github is not the only platform to download it, is also published in the Unity assets store. This implementation is not a really engine dependant one, it can work without an engine, but we place it in this group because it has a Unity adapted version with some UI interactions.

The author of this solid GOAP library is Luciano Ferrano, a web and desktop applications developer with more than 10 years of experience and a large list of known programming languages and platforms.

In this case, we focus in the Unity version of the tool because is one of our direct competitors. During the analisis we will treat about the tool core structures, its functionalities and the UI. Time to deep in and start the analisis.

Core structures:

- **Agent:**

  Principal core class that holds all the data related with the action planning; actions, goals, current goal, etc. When the agent class is initialized, it automatically collects all the available goals and calculates which is the current one. To make this possible, users have had to previously set all the agent actions and the world state.

  Agent class does not contain the calculation algorithm, it uses another class named ReGoapPlannerManager to generate the actions planning.

- **Action:**

  This class, aside from define what the action do on execution, it also contains several planning data like preconditions, effects and cost. In this framework, there is a Run method inside the action class to start the action execution. This Run method also sets the previously executed action and the next action to execute.

```
void Run(IReGoapAction<T, W> previous, IReGoapAction<T, W> next, //...
```

Figure 2.13 ReGOAP action code

- **Goal:**

The Goal class is basically a container class for the goal world state that can also store the generated plan and the planner used to calculate it. Provides all the necessary functionalities for planning and execution, for example: getters/setters, check if the goal is reachable, run the plan, etc.

```
protected ReGoapState<T, W> goal;
protected Queue<ReGoapActionState<T, W>> plan;
protected IGoapPlanner<T, W> planner;
```

Figure 2.14 ReGOAP goal code

- **State:**

This framework defines the ReGoapState class, used to store the world variables. In this case, a world variable is a pair of values with undefined type. This world state system provides a lot of flexibility for the users because they can store the desired information in any kind of variable type. In other hand, the world state definition system can become extremely complex and messy if the user does control it.

```
public class ReGoapState<T, W>
{
    //World variables
    private Dictionary<T, W> values;
```

Figure 2.15 ReGOAP state code

UI:

- **Plan debugger:**

This GOAP implementation also provides a basic agent visualizer, useful to debug the agent behaviour during the execution. The agent visualizer displays several information about the agent planning and situation. For example, the agent gameobject, the current goal, the current action with its preconditions and effects, etc.

Figure 2.16 ReGOAP debugger capture

GOAP:

This GOAP implementation was originally created to generate the AI of the videogame Basher Beatdown, a fight game in which you can combat against the AI. When the game was released, the used GOAP implementation was published in github by its creator. Peter Klooster is the author of this implementation, an indie game developer from Netherlands at who we have to be grateful for publishing the source code if his GOAP system.

The mentioned videogame was developed in Unity, what means that the GOAP system is adapted to the engine. Furthermore, the tool implements a GOAP visualizer using Unity UI system. This implementation also uses a tool named Thread Ninja for multithreading, this tool can be downloaded for free from the Unity assets store; We consider that this code dependency doesn't supposes any usability barrier.

To analyze this tool, we follow the same content structure used in the previous analysis. The content is divided in the core structures and the UI.

Core Structures:

- **Agent:**
  Principal class that contains all the necessary data and methods for planning. Actions, goals and world variables are stored in this class that provides methods to add or remove all the desired information from it. The final planned actions path is generated and stored by this class too, what converts the agent class in the main core structure of this GOAP implementation. In this case, the initial structure generated is not an actions path, is a tree with all the possible paths. The agent logic compares all the tree available paths and compares the costs to choose which is the next action to execute.

- **Action:**
  Inside this core element we can find all the information related with pathfinding, like walkability cost, preconditions, effects, etc.

  As we mentioned before, this tool initially structures the action planning in a tree, therefore actions have the information about who are their parent and children nodes.

  Actions also implement a delay system that let users set a wait time before the next action starts, using methods defined in this class we can modify the delay range.

```
get
{
    //                max delay  min delay  current
    return Mathf.Lerp(delaySlow, delayFast, delay);
}
```

Figure 2.17 Peter Klooster's GOAP action code

Another interesting feature is the limited action runs in a concrete time, this improves the system planning randomness because agents can't repeat the same action over and over in a short period of time.

```
// Maximum amount of runtime before it get's blocked for X seconds
protected float maxRunTime = 3f;
```

Figure 2.18 Peter Klooster's GOAP run limit code

- **Goal:**

In this implementation, the goal class is not a desired world state in which, for example, enemies are dead. Here the goal class defines the target action that have to be executed at the end of the planning and all the child actions to reach it.

Goals have a cost and a cost multiplier that gives more randomness to the system. Agents have a list with all the available goals and consider the goal cost in the selection process.

```
// The cost multiplier. Used to create a tendency towards a goal
public float multiplier { get; protected set; }
```

Figure 2.19 Peter Klooster's GOAP goal multiplier code

- **DataSet:**

Basic class that contains all the data about the world state, used by agents to calculate the different action plannings. The world state is defined by boolean values identified with a string, this class implements methods to modify and compare those values during the planning generation.

```
Dictionary<string, bool> data; //World variables
void SetData(string key, bool value); //Set varible value
bool Equals(string key, bool value); //Variables comparsion
```

Figure 2.20 Peter Klooster's GOAP dataset code

UI:

- **GOAP Visualizer:**

  UI canvas that displays all the information about the different action plannings generated by the agents. When we select an agent all its available goals are presented in the same time. The UI structure simulates the real logic structure, goal on the top and all the child actions presented in a node tree under it. Each node presents the most important information about the action, name, cost, preconditions, etc.

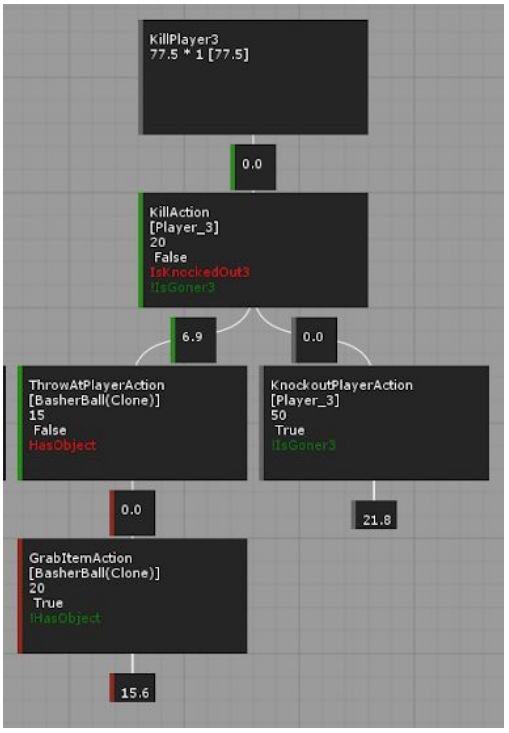

Figure 2.21 Peter Klooster's GOAP debugger capture

To conclude Peter Klooster's GOAP implementation analysis, let's point the most important features found during the research.

- This tool offers a basic but solid planning method really easy to understand and use for your project.
- The UI implemented makes it easier to understand and is functional to debug the agents behaviours.
- There are several variables , like action cost and goal cost multiplier, that make the system more unpredictable, remember that randomness is a great feature for a AI generation tool.

This are some of the several features that we must consider during our tool developing process.

GOAP AI:

The last mentioned GOAP implementation is GOAP AI, a tool published in the Unity assets store with a price of 40$. There is not too much information about this tool and the only way to access it is buying it, we don't consider to do that so this is only a short mention of the public information that we found.

Users report that the GOAP architecture is implemented correctly, following a well-structured and commented code. There is an example scene provided by the developer, it helps users to completely understand the tool, but in some cases can be too much simple for those who want to see the tool in action.

The example scene is only composed by 4 actions and 2 goals, what provides a basic scene to see how planning works in this GOAP implementation. Planning algorithm uses the actions preconditions and effects to calculate the path to the goals, those preconditions and effects are basically strings and values that define the world state.

Finally, is necessary to point that the tool don't supports any kind of UI and all the logic is calculated in the same thread, what probably effects the tool performance in a bad way.

2.3 Conclusion

Now that we know the history behind the GOAP architecture and the current market state, it is time to see if our target tool offers something new to the market.

If we focus in the logic part, we have seen that all tools offer a solid framework that users can modify and adapt to their projects. Each tool uses different classes and methods to provide the GOAP functionality, some of the previously analyzed improve the basic GOAP implementation adding sub-systems, like Peter Klooster's GOAP that implements a goal cost multiplier to increase agents behaviour randomness. Our objective in the logic area is relatively simple, bring a clean and commented code that provides a easy to use framework. As we can see, in the logic part our tool does not really differ from the others, but this is not a problem. Because our differentiation is not in the core logic, is in all the sub-systems and resources that surround it.

From all the tools listed in the market analisis, there is no one that provides a complete example scene, one of our objectives is build an example scene where we properly use all the content offered by our tool. Another important part to compare is documentation, some of the analyzed tools offer a good documentation about how to use the tool and general information about GOAP architecture. We want to surpass the simple documentation and bring a whole system of constant information that users can access at any moment.

To do that, a part of provide an example scene with instructions, all the tool code will be commented in detail and all the UI elements will have tooltips to describe its functionalities. Inside the tool users will also find a toolbar menu to access the GitHub documentation.

Finally, our most differential feature. We are going to implement UI elements for all the functionalities supported by the tool. In other words, users will be able to do any action only using UI. The only actions that obviously can't be done through UI are coding new action and behaviour scripts. From the previously analyzed tools, three implement UI and is only used to display the generated output and practically no UI interaction is supported.

This is why we really believe that our tool will offer a whole new and innovative framework that could be easily used by any developer interested in AI.

3. Planning

In order to have a more clear idea of the project life cycle and generate a solid milestones plan, we divide the project development in different stages. Each stage defines a general development state that englobes several milestones. During the next paragraphs all the different stages and milestones will be commented in detail.

3.1 Initial planning:

R1	Rubric 1
R2	Rubric 2
R3	Rubric 3

Table 3.1 Gantt table legend

	Start	End	Weeks 1	2	3	4	R1 5	6	7	8	9	10	11	R2 12	13	14	15	16	R3 17	18	19	Release 20
			11/02	18/02	25/02	04/03	11/03	18/03	25/03	01/04	08/04	15/04	22/04	29/04	06/05	13/05	20/05	27/05	03/06	10/06	17/06	24/06
Documentation	1	20																				
Research	1	1	■																			
State of the art	1	2	■	■																		
Planning	2	3		■	■																	
Methodology	2	3		■	■																	
Introduction	4	4				■																
Development	5	20					■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	
Conclusions	19	20																			■	■
Tool Wiki	18	19																		■	■	
Vertical Slice	5	10																				
Basic Agent	5	6					■	■														
Basic Action	6	6						■														
Node Canvas	6	8						■	■	■												
Agent UI	9	10									■	■										
Action UI	9	10									■	■										
Alpha	11	17																				
Blackboard	11	14											■	■	■	■						
Action Variables	12	13												■	■							
Agent Behaviour	16	17																■	■			
Planning	15	17															■	■	■			
Blackboard UI	13	15													■	■	■					
Variables UI	14	15														■	■					
Behaviour UI	17	17																	■			
Planning Debug	16	17																■	■			
Beta	18	19																				
Planning Iteration	18	19																		■	■	
UI Iteration	18	19																		■	■	
Instructions	19	19																			■	
Gold	20																					

Table 3.2 Gantt planning

3.1.1 Documentation:

11 february - 22 june

Stage that encompasses all the project duration and focus on research and documentation. The initial objectives of this stage are clarify our objectives and collect and analyze all the important information about other similar tools in the market. Once we know perfectly the market situation and our goals, the next objectives are generate the memory and describe all the development process.

3.1.2 Vertical Slice:

11 march -21 april

Start of the software development process. During this stage we implement the most basic logic classes of the tool, what includes agents and actions. This stage also includes the development of the basic UI support to the mentioned elements and the first player interactions like add or remove agents.

3.1.3 Alpha:

22 april - 9 june

Development of all the logic elements and its functionalities. First we focus in code all the data management classes for the GOAP architecture, like blackboard and variables. Next we develop the required UI elements to support all the data management functionalities, like add preconditions or effects to an action. Once all the core logic elements are implemented, the next step is code the agent behaviour class and a basic planning algorithm. At the end of this stage the tool is able to generate a basic planning using all the data setted by the user.

3.1.4 Beta:

10 june - 23 june

This stage focus on improve all the previously implemented features and add planning output for debugging. First step consists in implement a real-time output system for the agents planning. This output system will display all the important information about the action plans generated by the agents, what includes available actions, preconditions, effects, goal world state, etc. Once this output system is implemented, we can proceed with the next objective, which consist in debug the planning algorithm and improve its performance. Finally the tool is completed, time to publish the gold version, but before we must document all the tool functionalities and generate a detailed wiki to help all the future users.

3.1.5 Gold:

24 june - 25 june

The project is finished and the gold version of the software is published on github, but there is still work to do. Now we need users to keep improving our tool, using the feedback received from them we will be able to detect and solve all the hidden issues. In this project stage we will also write the final project conclusion.

3.2 Estimated Costs

This costs estimation tries to simulate the costs behind a professional project. In order to generate the most accurate numbers, the programmer salary is the average of spain and all the software cost are estimated with the professional versions.

The project development time is about 5 months and two weeks, but in this estimation we decide to round the development time to 6 months, with the objective to cover all the development days.

Costs	Subject	Units	Cost x Unit	Amortization (months)	Used months	Total
Personel						
	Programmer	6 (hour-man)	2.500,0 €	--	--	15.000,0 €
Equipment						
	Computer	1	1.500,0 €	48	6	187,5 €
	Screen	2	350,0 €	36	6	58,3 €
	Keyboard	1	55,0 €	36	6	9,2 €
	Mouse	1	50,0 €	24	6	12,5 €
	Desk	1	200,0 €	36	6	33,3 €
	Chair	1	70,0 €	36	6	11,7 €
Software						
	Unity License	6 (months)	125,0 €	--	--	750,0 €
	Visual Studio	6 (months)	45,0 €	--	--	270,0 €
	Hack n Plan	6 (months)	48,0 €	--	--	288,0 €
	Trello	6 (months)	8,5 €	--	--	51,0 €
Maintenance						
	Water	6 (months)	25,0 €	--	--	150,0 €
	Electricity	6 (months)	40,0 €	--	--	240,00 €
	Ethernet	6 (months)	50,00 €	--	--	300,00 €
Sum Total						17.361,5 €

Table 3.3 Project costs estimation

3.3 SWOT

In order to identify the strengths, weaknesses, opportunities and threats of this projects, we develop the next SWOT analisis.

	Helpful	Harmful
Internal Origin	**Strengths** **- Free product** **- Innovative features**	**Weaknesses** **- First contact** **- Limited resources**
External Origin	**Opportunities** **- New market with few competitors** **- Growing market** **- The platform brings visibility**	**Threats** **- Experienced competitors** **- Small market sector**

Table 3.4 Project SWOT analysis

Strengths:

- **Free product:**

  We offer a completely free product, users can get it from the Unity assets store or github without spending money on it. Additionally, the product is open-source, what means that user can access and modify the source code to adapt the product to their needs.

- **Innovative features:**

  Our tool is not limited to the GOAP logic architecture, also offers a whole functional UI that let users design their AI agents and debug the results easily. As we have seen in the market analysis, our competitors do not offer a complete UI for their tools or directly there is no UI.

Weaknesses:

- **First contact:**

  My experience developing Unity tools is null and I have only worked with GOAP architecture theoretically. When a person do something for first time normally needs more research and iteration than a person with previous experience, that extra resources expenditure is a strong weak for a project with limited time.

- **Limited resources:**

  This project is limited to a development time of five months, in this time we have to generate the tool and all the content that complements it. If we look at the planning, we can see that the entire documentation stage and a part of the beta stage are focused in generate the documentation. In conclusion, more than a month is spend in research and documentation, that can affect the final resulting tool in a bad way. Basically we have less time to invert in the software development and the risk to apply a contingency plan is higher, what results in a less competitive product.

Opportunities:

- **New market with few competitors:**

  GOAP tools for Unity projects is a completely new and non-exploited market. A new market brings to us the opportunity to compete with the few existent products in a relatively easy way, because we can fastly generate a solid market study. Knowing which functionalities we should offer to the users to compete with the other products in our market sector.

- **Growing market:**

  Now a days, GOAP architecture is not the mainstream methodology to generate AI for real-time projects, but that fact does not mean that developers are not interested in using it. The GOAP community is growing and every day more developers use this architecture for their projects, if we bring a functionally and easy to use GOAP tool more users will interest in this architecture.

- **The platform brings visibility:**

  One of the main problems when you publish a product on the internet is the lack of visibility. In this case, the platform brings to our product the necessary visibility to reach the interested users, it is enough to search “GOAP” in the Unity assets store and our tool will appear in the search results.

Threats:

- **Experienced competitors:**

  After the market analisis, we realize that most of the GOAP implementations have been done by really experienced developers. Have previous experience working in big and complex profesional projects provides a huge advantage against the beginner programers, not only in the coding part, also in the task management and milestone organization.

- **Small market sector:**

  It is true that GOAP architecture for real-time projects is an interesting and growing market, but at the end is a very small market compared with the FSM and BT architectures. GOAP market brings opportunities but at the same time limits our project visibility and users range in a little niche market.

3.4 Risks and contingency plans

All projects need to define contingency plans to react properly when something goes wrong. In this case, we are talking about a software development project with a novice programmer as author. In other words, the risks are so high and the contingency plans must be perfectly defined from the beginning. In order to structure the different contingency plans clearly, we are going to divide them in general project risks and specific risky tasks.

3.4.1 Project risks

General project risks that can appear during the whole projects and seriously affect the final product.

- **Lack of time:**

  The tasks planning is generated at the beginning of the project and is impossible to know how many time would you need for each task. Therefore, any of the estimated task times can be miss estimated and we need a contingency plan to properly react when a task is out of time. Not having enough time is not a especial risk of this project, lack of time is normally the main problem of all projects.

  Contingency plan:

  This contingency plan have two variants, depending of the task importance.

  If the affected task is a simple UI implementation and the tool can work perfectly without it, the task will be paused and passed to a stand by tasks list. That list will be readed at the start of the Beta stage, in order to decide which can be done during the time reserved for UI polishing.

  On the other hand, the affected task can be a important core one that the tool needs to work properly. In this case the time reserved for this task will be augmented and the extra time needed will be removed from other less important tasks.

- **Lack of knowledge:**

  This project contains several logic elements with a relatively complex structure, the risk to implement them wrong and need to research for more information is really high. Inside each project programming task, the estimated information search time is included, but sometimes the estimated time can't be enough.

  Contingency plan:

  This problem is directly related with time lack, because inverting more time in research we can solve the lack of knowledge. Therefore, the contingency plan is the same than the described for lack of time risk.

3.4.2 Risky tasks

Risky tasks that appear in specific project stages and can modify some final product features.

- **Blackboard and variables:**

  Blackboard is the class that holds all the variables and its values. In other words, the blackboard defines the world state for the AI agents. In our implementation, variables can store several types of data, like integers, booleans, strings, etc. To support all those variable types, is necessary to build a generic variable class with all the needed functionalities and adapt it to the blackboard structure. The main risk is not being able to implement the mentioned generic class.

  Contingency plan:

  If we are not able to implement variables with multiple data types, we can limit them and partially avoid the complexity of the task. In other words, we can limit the blackboard variables to only boolean values. Therefore, the main risk disappear and we can focus in only implement a basic boolean variable.

- **Use multithreading in planning algorithm:**

  Planning algorithm generates the actions planning using the data setted by the user. Each planning generation requires a lot of operations that must be perfectly implemented to don't affect the tool performance. A good way to optimize that operations is using multithreading, then each operation can be calculated parallely in different threads. This is a complex system and the risk of not being able to implement it is really high.

  Contingency plan:

  If the planning algorithm does not work properly with the multithreading system, the safest way is return to the basic algorithm without multithreading. The tool performance will be affected but the algorithm results and the supported functionalities will be the same.

3.5 Tasks management

3.5.1 Management Methodology

A project is basically a list of tasks that must be done to finally generate the desired product. To complete all the tasks properly, we need to define a workflow to manage all the tasks during the project development.

For this project we have selected an agile methodology named Scrum. Agile methodologies are all those software development methodologies based on iterative development. Scrum is originally designed for teams with several members, but we can adapt it for this individual project simply removing the daily stand ups.

Therefore, owr tasks management methodology basically consist in divide the tasks in weekly sprints. At the sprint start the task is presented and the team starts working in it, at the end of the sprint the task is checked to decide if it is completed or needs more iteration.

3.5.2 Management Tools

To control the project state and keep a clear track of all the tasks, we will use the next tools:

The tool selected for the tasks management is Hack & Slash, this software will let us define different tabs for each project development stage. Inside each tab we will define all the stage tasks and all the necessary information to have a good track of them.

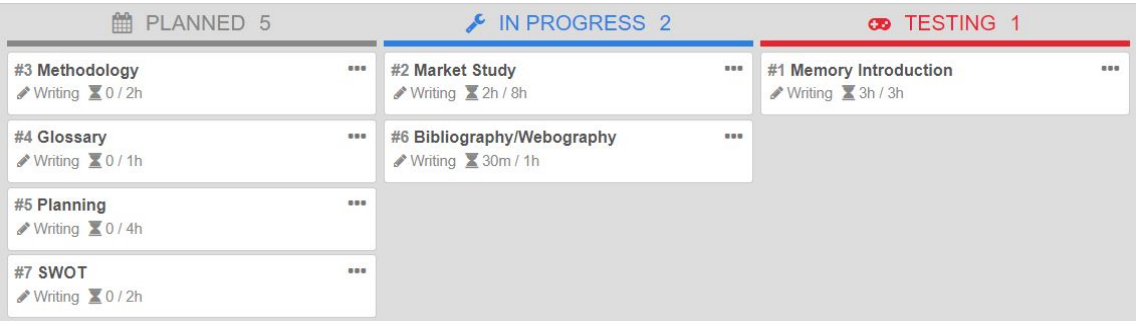

Figure 3.1 Hack n Plan example capture

Github is the selected tool to manage the tool evolution. All the software and documentation modifications will be uploaded to github, like readme updates, releases, issues, etc.

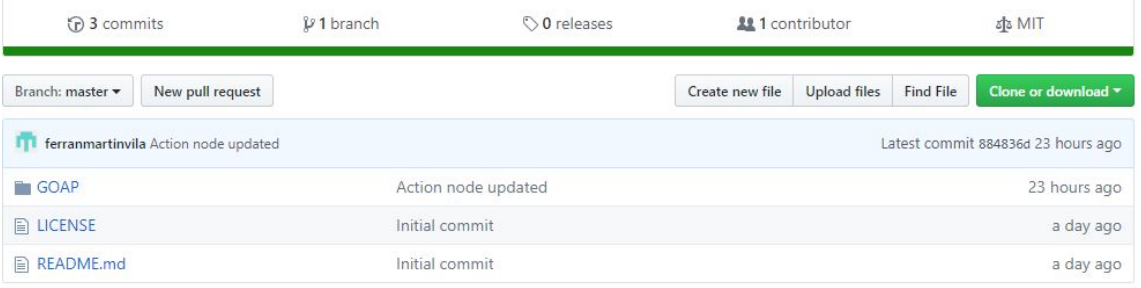

Figure 3.2 Github example capture

3.6 Planning Modifications

After the first six of the development, the vertical slice end date was modified one week forward. The main reason is that the blackboard and variables development was added to the vertical slice stage. Because agent, action and canvas classes were finished faster than we estimate in the project planning.

	Start	End	Weeks 1 (11/02)	2 (18/02)	3 (25/02)	4 (04/03)	R1 5 (11/03)	6 (18/03)	7 (25/03)	8 (01/04)	9 (08/04)	10 (15/04)	11 (22/04)	R2 12 (29/04)
Vertical Slice	5	10												
Basic Agent	5	5					■							
Basic Action	5	5					■							
Node Canvas	5	6					■	■						
Agent UI	7	7							■					
Action UI	7	7							■					
Blackboard	7	9							■	■	■			
Action Variables	8	11								■	■	■	■	
Blackboard UI	8	8								■				
Variables UI	9	9									■			
Planner	10	11										■	■	
Behaviour	10	11										■	■	

Table 3.6 Vertical slice planning modification

In the vertical slice stage we also start the development of the agent planner and behaviour but this implementations are only basic versions that practically work as placeholders for the future versions.

4. Methodology

All projects need to have a clearly defined development methodology to thrive properly, include a solo project like this. The development methodology selected for this project is divided in five stages:

- **Research**: Analysis of the several GOAP implementations in the current market, in order to extract their functionalities and study if they can be implemented in our project.

- **Definition:** Once we know all the functionalities supported by the current market, we define which ones are going to be included in our tool.

Research and definition stages are done only once at the beginning of the project. The next stages are used to develop each of the previously selected functionalities in the definition stage:

- **Design**: In this stage we decide how the functionality will be implemented and all the sub-functionalities that it will provide to the tool. The functionalities design is not only focused in logic, it also includes the design of UI elements for user interaction.

- **Prototype**: Implementation of the previously designed functionalities. A functionality prototype is not a simple mock-up, is basically a non tested version of the functionality. The prototype can pass the test and be part of the final tool.

- **Testing**: User testing to decide if the functionality has been implemented correctly or not. Test include logic and usability metrics. Therefore, a functionality can work perfectly but if users don't understand how to use it the functionality will be tagged as non correctly implemented.

Prototype and testing stages are iterated till reach the desired result for each functionality. During the general project stage Alpha, all the functionalities have to be prototyped and tested at least once. If the Alpha stage ends and some functionalities need a rework, Beta stage is designed to iterate and improve all those non correctly implemented functionalities.

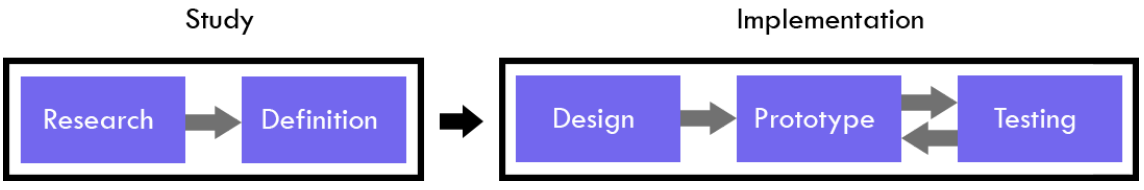

Figure 4.1 Methodology structure graphic representation

5. Development

The development methodology used for this project follows the previously mentioned planning. In order to generate a clear and solid development description, we are going to divide this chapter with the stages defined in the planning structure. The documentation is an external stage that englobes all the project life cycle and describes it. Therefore, this stage will not be described in the development, the only documentation task commented will be the tool wiki generation in the beta stage.

5.1 Vertical Slice:

In this project, the vertical slice is the first step into the production area, and at the end of this stage we release the first version of this tool. An extremely basic and not completely functional version of the tool. But before start producing code we need a pre-production sub-stage, to prepare all the necessary tools and platforms that we will use to work.

5.1.1 Pre-Production

Normally the pre-production of a project is considered an independent previous stage of the first production chapter. In this case we have placed the pre-production as a sub-stage of the vertical slice, because the brief task englobed in it are not enough to be considered as a complete stage.
This sub-stage is only the practical pre-production part of the project, the theoric pre-production with all the information research and memory generation is inside the documentation stage. Let's briefly comment the pre-production tasks:

The main tool for this project is Github. This tool provides an online repository that let us upload our content to it and update all the important changes. We need to configure the repository in order that it works properly with our type of content and it only uploads the necessary data and modifications.

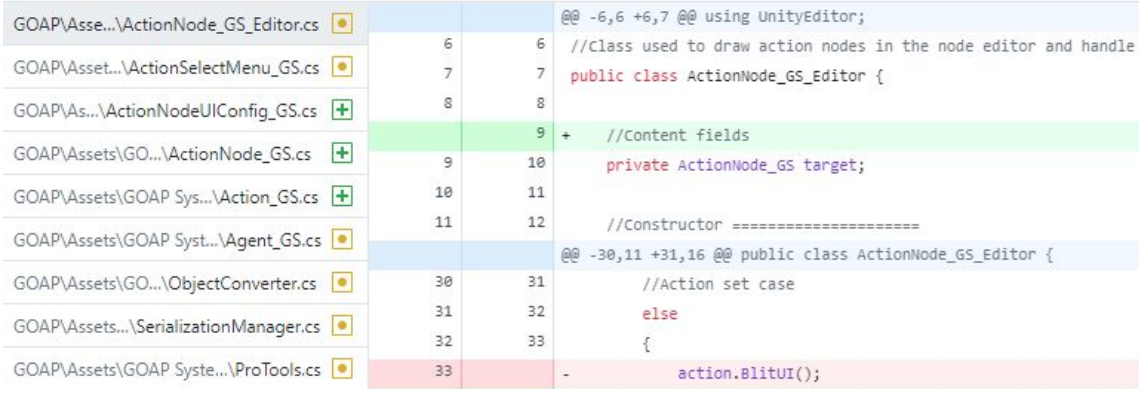

Figure 5.1 Github capture

We will work with Unity engine and Visual Studio code editor. Therefore, in the Github repository we add the Unity **gitignore** that will ignore all the meta files generated by the engine and the Visual Studio cache directory. All those files are locally generated by the engine every time we open the Unity project for first time and the same happens with Visual Studio cache files.

```
### Unity ###
/[Ll]ibrary/
/[Tt]emp/
/[Oo]bj/
/[Bb]uild/
/[Bb]uilds/
/Assets/AssetStoreTools*
# Visual Studio 2015 cache directory
/.vs/
```

Figure 5.2 Gitignore capture

Once the gitignore is setted, the next step is start uploading content to the repository. Previous to the Unity project we upload the project license, which defines how our content can be treated by all the persons that acced to it.
In this case we decide to use the MIT license. This license let users copy, modify and distribute our content freely, meanwhile a copy of the license with our name is located in the product.

```
MIT License

Copyright (c) 2019 Ferran Martin Vilà

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.
```

Figure 5.3 Copyright capture

Now that the repository is ready, it's time to upload our base Unity project. This initial project is basically an empty Unity scene with the required configuration for the tool. For our project we only need to set the Visual Studio as the Unity code editor and the Unity project is ready for being uploaded to the repository as our tool base.
The last pre-production task is configure our Hack n Plan board. As we mention in the planning tasks management part, we will use Hack n Plan to manage our tasks during the project development. We divide tasks in only two groups, programming tasks and writing tasks. At the beginning of every week we place the planned tasks on the board, during the week we update them and when the week finish the board is closed.

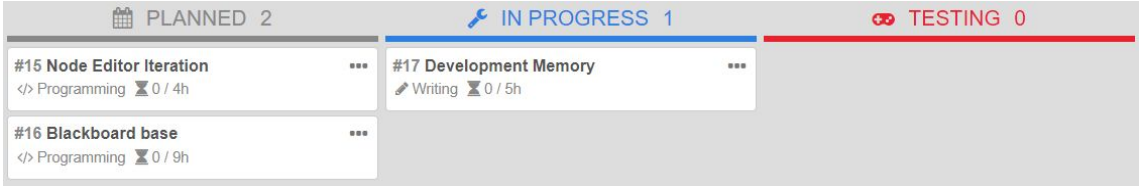

Figure 5.4 Hack n Plan planning capture

5.1.2 Vertical Slice

This stage focus in developing a code base with the most core and necessary logic elements for the final product. Previously to analyze each of that elements individually, we have two basic UML diagrams to describe the code structure reached in the vertical slice tool version. This two UML will evolve during the project stages and will help us to track and understand the tool evolution.

Logic UML:
Describes the current structure of the core logic elements.

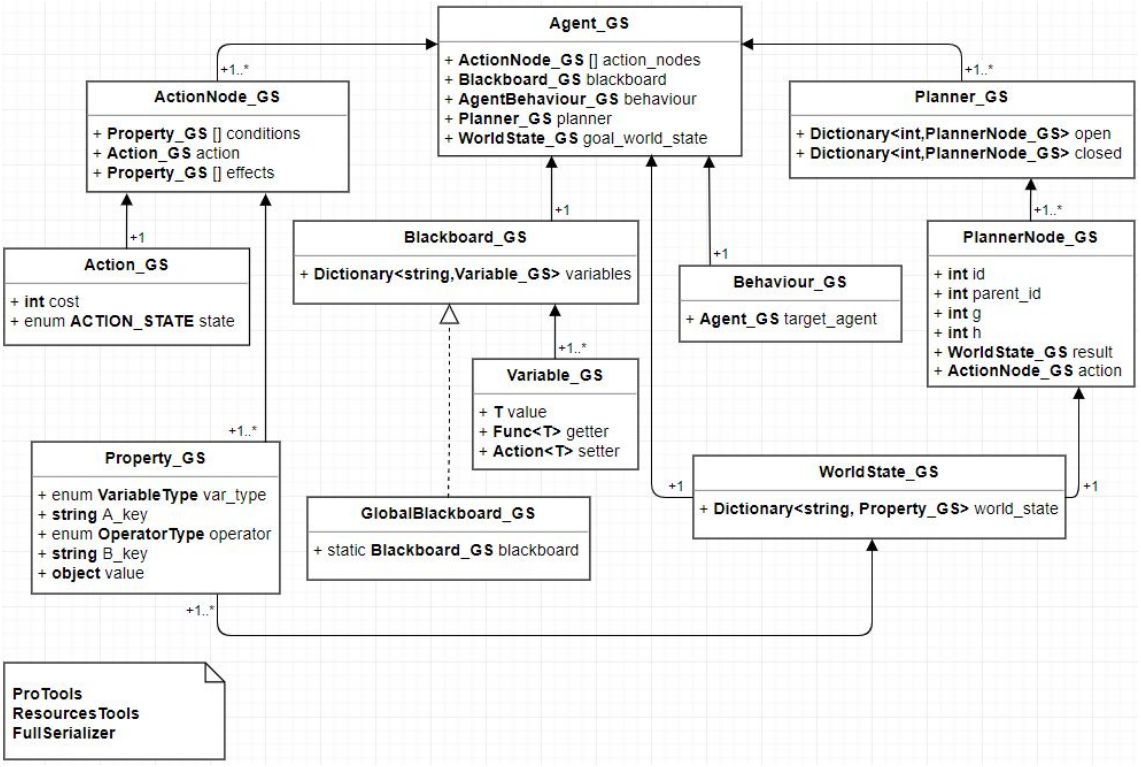

Figure 5.5 Logic UML

Editor UML:

Describes the current structure of the logic elements editors, selection menus and canvas editors.

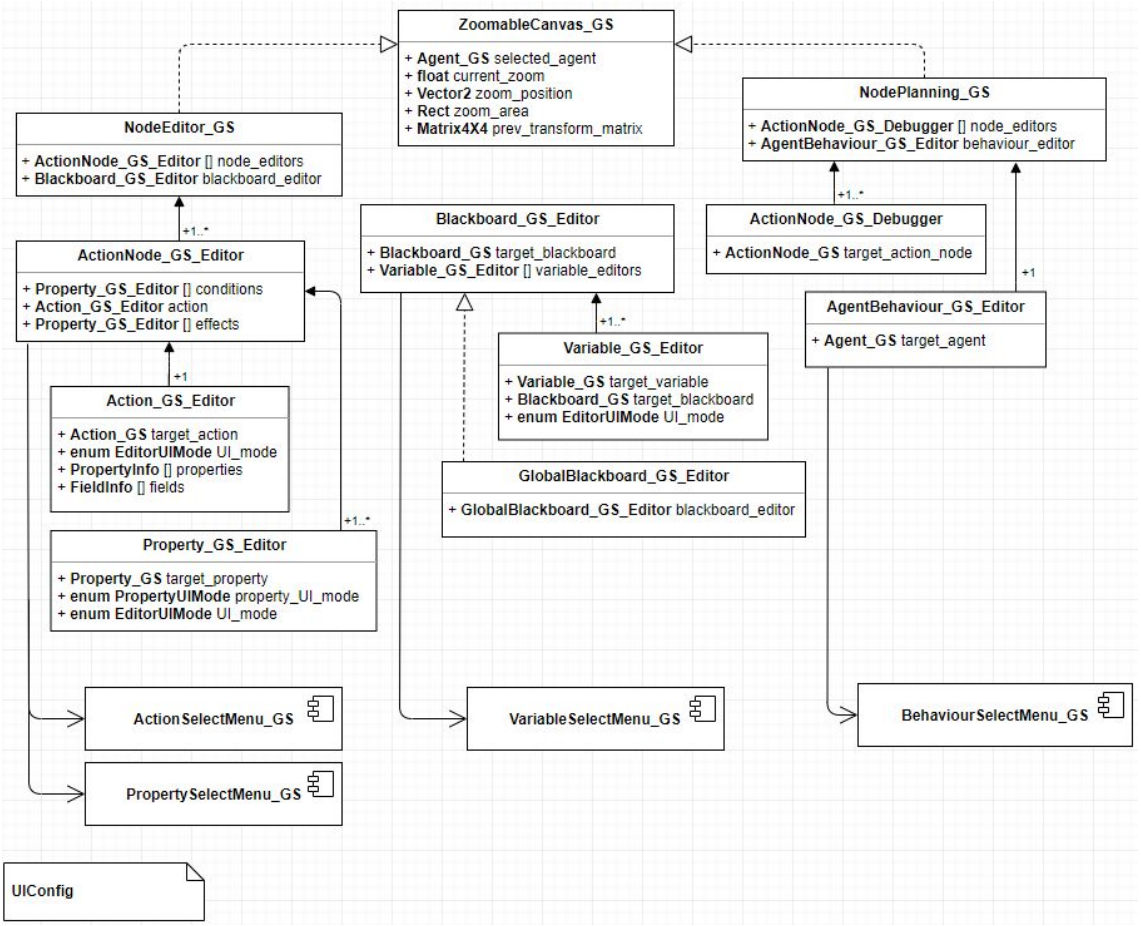

Figure 5.5 Editor UML

Now that we have a general idea of the current logic and editors structure, it is time to individually analyse the different elements that compose the tool.

In order to understand all the current tool elements and functionalities, we obviously need to start commenting the logic elements and next continue with the editor elements. Because editors are basically a visual support of the tool logic elements. During the elements description we will refer to the different UML structures to clarify any doubt about the presented diagrams.

Logic Elements:

Tools and Serialization:
With the objective to work in a professional and optimal way, we implement a pair of tools adapted to our necessities. Those tools are basically a production tool and a resources tool.

- Production Tool: Provide static methods focused on allocate, find and transform data. This static methods are defined inside a static class named ProTools and all the classes can access them. The main objective of this tool is generate clean code and avoid copy pasting code that can be placed in a method.

```
public static T CreateDelegate<T>(this MethodInfo method_info, object instance)...
public static T AllocateClass<T>(this object myobj)...
public static List<T> FindAssetsByType<T>() ...
public static T FindAssetByPath<T>(string path) ...
```

Figure 5.6 Production tool code fragment

- Resources Tool: This tool is basically a warehouse where we store important data that will be used later by the user. Inside the ResourcesTool class we can access to all the actions and behaviours defined in the current project. All this data is loaded every time the Unity project detects a code change and all the assemblies are compiled. This process of iterate all the assets and only store the desired scripts have a high performance cost. Doing it once and storing the result let us optimize the user interaction in the scripts selection menus, in which we need to access to all this previously stored scripts.

```
private static Dictionary<string, Object> _action_scripts = null;
private static string[] _action_paths = null;
private static Dictionary<string, Object> _agent_behaviour_scripts = null;
private static string[] _behaviour_paths = null;
```

Figure 5.7 Resources tool code fragment

A part of generate our tools, we also use an external serialization tool named FullSerializer. This tool is a JSON serialization framework developed Jacob Dufault, an experienced software developer who has worked in very important companies like Microsoft and Google. FullSerializer let us serialize our custom classes into strings and deserialize them when the scene serialization callback is send by the engine.

```
//Serialization process
public static string Serialize(object value,
                               Type value_type,
                               List<UnityEngine.Object> object_references)
```

Figure 5.8 Serialization method code fragment

```
//Deserialization process
public static object Deserialize(Type value_type,
                                 string serialized_state,
                                 List<UnityEngine.Object> object_references)
```

Figure 5.9 Deserialization method code fragment

Agent:

Main class that uses all the other logic classes to generate a behaviour. Users have to define an agent behaviour script, the blackboard variables, and finally the action nodes. The agent class holds all this information and uses the planner to calculate the actions path to the goal world state defined by the behaviour.

The Agent_GS class is a component that can be added to any GameObject, when an agent is instantiated a hierarchy icon marks the selected GameObject.

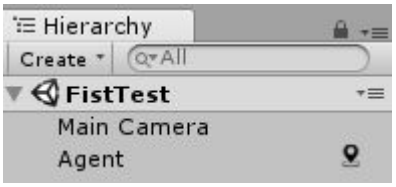

Figure 5.10 Agent hierarchy icon capture

The agent can't work without a blackboard, so it checks if the target GameObject has a blackboard component during the instantiation process, in negative case a new blackboard is allocated and added to it.

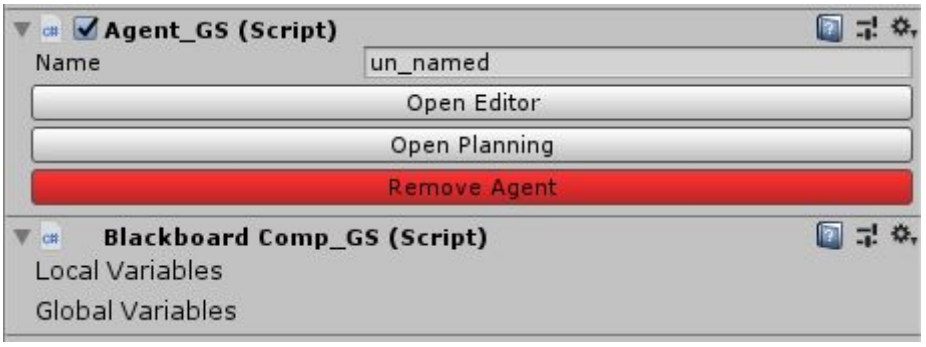

Figure 5.11 Agent and blackboard components inspector

Action node:

All agents have an array of action nodes generated by the user. Each of this action nodes define an action, the action conditions, the action effects, the action cost, and the custom fields that the user wants to add in the action script, like speed, damage, etc. Action nodes are basically the available actions for an agent.

```
private Property_GS[] _conditions = null; //Conditions array
private Property_GS[] _effects = null; //Effects array
private Action_GS _action = null; //Action linked to the action node
```

Figure 5.12 Action node code fragment

Action:

The default action script provided by the tool is only a placeholder that the user has to modify. Base action script only defines a cost integer and the ACTION_STATE enum. The cost is used by the planner during the path generation and the action state is used by the agent during the previously generated path execution.

```
private int _cost = 1;
private ACTION_STATE _state = ACTION_STATE.A_IDLE;
```

Figure 5.13 Action script fields code fragment

A part of this two fields, the action defines iteration methods like Awake, Start, Update, End and Break. Awake and Start are initialization methods that are only called once when the action is going to be executed, Update is called every frame when the action is in execution, and finally End and Break are finalization methods. End method is called when the action ends correctly and Break method is called when the action execution ends with errors.

```
//Awake is called once at first execution loop
public virtual ACTION_RESULT ActionAwake()...
//Start is called once at first execution loop
public virtual ACTION_RESULT ActionStart()...
//Called on the action update
public virtual ACTION_RESULT ActionUpdate()...
//Called when the action ends correctly
public virtual ACTION_RESULT ActionEnd()...
//Called when the action ends with errors
public virtual void ActionBreak()...
```

Figure 5.14 Action script methods code fragment

Property:

Properties are basically two values and a operator that relates them. For example, a property can define an integer named A with a value of 5, an integer named B with a value of 2, and the operator '>'. Therefore, the resultant property defines 5 > 2. This tool uses the property class to generate conditions and effects for the previously mentioned action nodes using the values defined in the blackboard. This property implementation also allows to operate blackboard values with new values defined by the user.

In this tool capture we can see two properties that work like conditions, the first property compares a blackboard value with a random value and the second compares a blackboard value with another blackboard value.

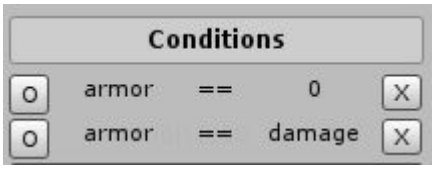

Figure 5.15 Properties used as conditions

As we mentioned before, properties can represent conditions and effects. Therefore, properties operators are divided in two groups, passive and active. Passive operators are all those operators that do not modify the value, like '==', '>=', '<=' and '!='. On the other hand, there are the active operators that modify the target values, like '=', '-=' and '+='.

Blackboard:

The blackboard class stores all the variables defined by the user. There are two types of blackboards: The local blackboard allocated by the agent and the global blackboard, which is static and all agents can access to its variables. This class is extremely simple, only have a dictionary where the variables are stored by their name and methods to add, remove or access to the variables.

```
private Dictionary<string, Variable_GS> _variables;
```

Figure 5.16 Blackboard variables dictionary

```
public Variable_GS AddVariable(string name, VariableType type, object value)...
public void RemoveVariable(string key, bool global)...
public TVariable_GS<T> GetVariable<T>(string name)...
```

Figure 5.17 Blackboard methods

Variable:

This class is basically a container for a value, with the extra functionality that can be binded to another value defined in a script.

For example, imagine that we define a value named ‘damage’ in the agent blackboard, the current value of ‘damage’ is 5 but we want that the damage value is the same as the field ‘attack’ defined in the ‘Combat’ script contained by the agent.

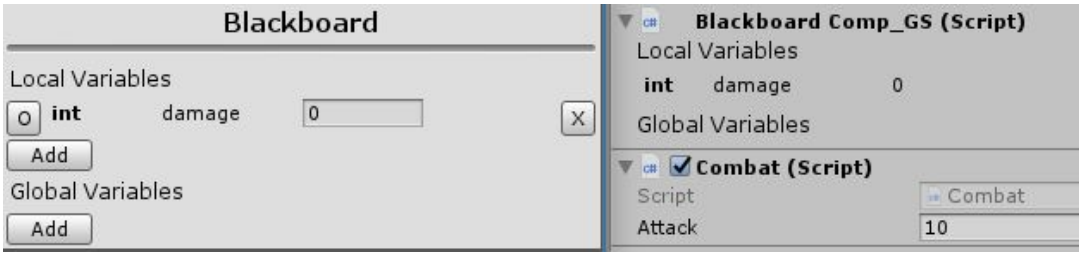

Figure 5.18 Blackboard variable capture

We can bind the ‘damage’ value to the ‘attack’ field adding callbacks on the ‘attack’ get and set methods.

Figure 5.19 Binded blackboard variable capture

Doing this we force that every time the user tries to get the ‘damage’ value we send to him the ‘attack’ value, and every time the user tries to modify the ‘damage’ value he is really modifying the ‘attack’ value.

World State:

The WorldState class contains a dictionary of properties that define a concrete state of the world.

```
private Dictionary<string, Property_GS> _world_state = null;
```

Figure 5.20 World state properties dictionary

A part of the current world state defined by all the scene variables, agents have a goal world state defined by the behaviour. The planner uses this two world states to generate the actions path. In other worlds, the path generated by the planner is the actions combination that transforms the current world state to the goal world state.

The previously commented logic elements are advanced versions that practically offer all the desired functionalities, but this Vertical slice version do not include the behaviour and planner development. Therefore, the next logic elements commented are placeholders with only the most basic functionalities.

Agent behaviour (basic version):

Agents use the behaviour class to generate the goal world state. As we mentioned before, the currently implemented behaviour class is only a basic placeholder for testing. The implemented functionalities are goals definition and the access to the target agent blackboard. Having a direct blackboard access from the behaviour class let users define behaviours that adapt and react to the current world state.

```
void SetGoal(string variable_name, OperatorType operator_type, object variable_value)...
void RemoveGoal(string variable_name)...
```

Figure 5.21 Agent behaviour methods

We have already seen that a world state is basically a dictionary of properties. Therefore, to define a goal we need to define a property. For example, a behaviour can define the next property, 'attack = 9'. If the value of 'attack' in the current world state is different to '9', this will force the planner to find an action that transforms the current 'attack' value to '9'.
If we define a goal with a variable name that don't exists in the current world state, the behaviour will be invalid and the planner will not generate anything.

Planner (basic version):

The planner class is the most important logic element for a GOAP generation tool. This class uses all the inputs sent by the user and stored by the tool framework to generate action paths adapted to the necessities of each agent. The currently implemented planner is uncomplete but we can comment how it works to compare it with the next versions.

To analyze how the planner works we will present an example and comment step by step all the involved functionalities and techniques.

In the exemple scene we have an agent with a local blackboard that defines three variables ‘max_life’, ‘life’ and ‘kills’. In this case all the variables are integers to simplify the example, the implemented blackboard supports several value types, like float, string, char, vector3, etc.
The next image shows the variables current values defined in the blackboard.

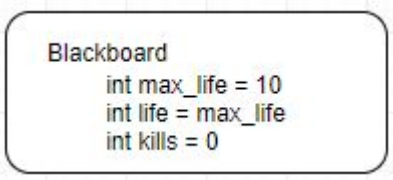

Figure 5.22 Exemple blackboard

The agent behaviour defines the goal world state using the blackboard variables. Therefore, the goal variables values defined by the behaviour are the next:

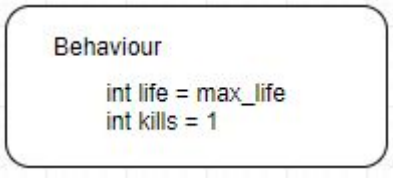

Figure 5.23 Exemple behaviour

We can’t transform the current variables values to the goal ones magically, we need to define actions with conditions and effects that modify the target variables. The actions defined for this exemple are the next:

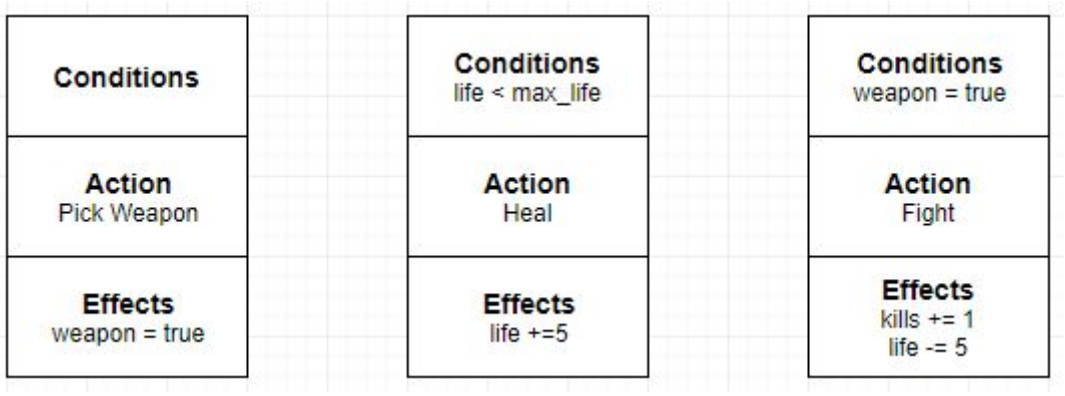

Figure 5.24 Exemple action nodes

Now that all the necessary elements to calculate the path are defined it’s time to analyze the planning process.
The planning generation algorithm is basically a pathfinding algorithm named A* adapted to the GOAP framework. To understand the planning process, first we need to analyze how the A* algorithm works.

Imagine that we have the next map of nodes:

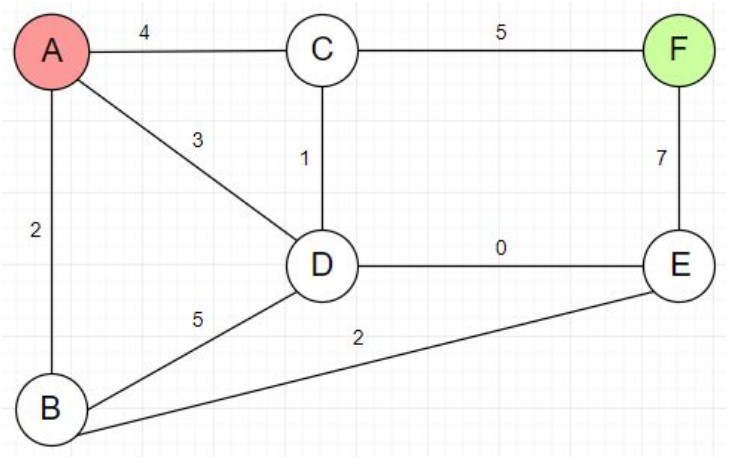

Figure 5.25 Exemple graph

The lines between the nodes are the connections and the numbers are the connection cost, this algorithm also takes in account the distance of each node to the target as a cost. The node A is the start and the node F is the goal. With the A* algorithm we are going to calculate the cheapest path from the node A to the node F.

The first step of the algorithm is to analyze all the A node neighbours and collect the cheapest. The algorithm stores the cost of each node in a list that will be used in the next iterations, the non analyzed nodes have a infinite cost. The cost of the nodes is generated from the connection cost plus the node distance to the target.

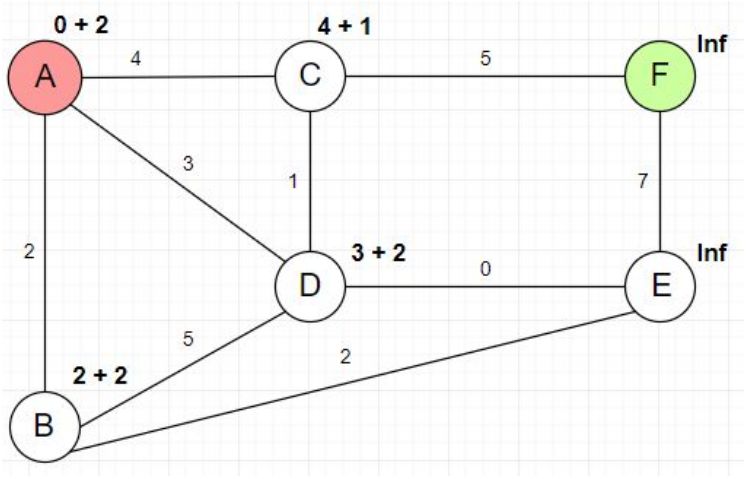

Figure 5.26 Exemple graph first iteration

In the next iterations we do the same process but checking the previously analyzed nodes and their costs. If the algorithm found a cheaper path to a previously analyzed node the node cost is modified. The final result of the presented graph is the next.

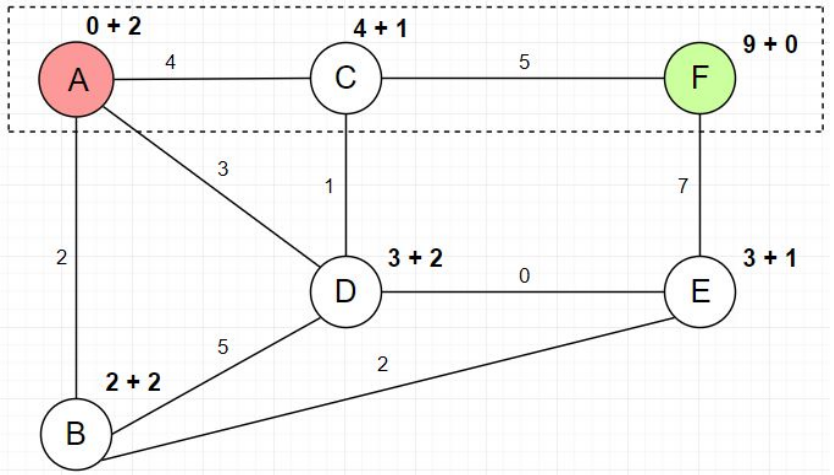

Figure 5.27 Exemple graph last iteration

In conclusion, the final path is A - C - F. Now that we know how A* works, lets apply it to the previously presented action nodes. Each action node is a node for the pathfinding algorithm and the connection cost is the action cost plus the action distance to the target world state.

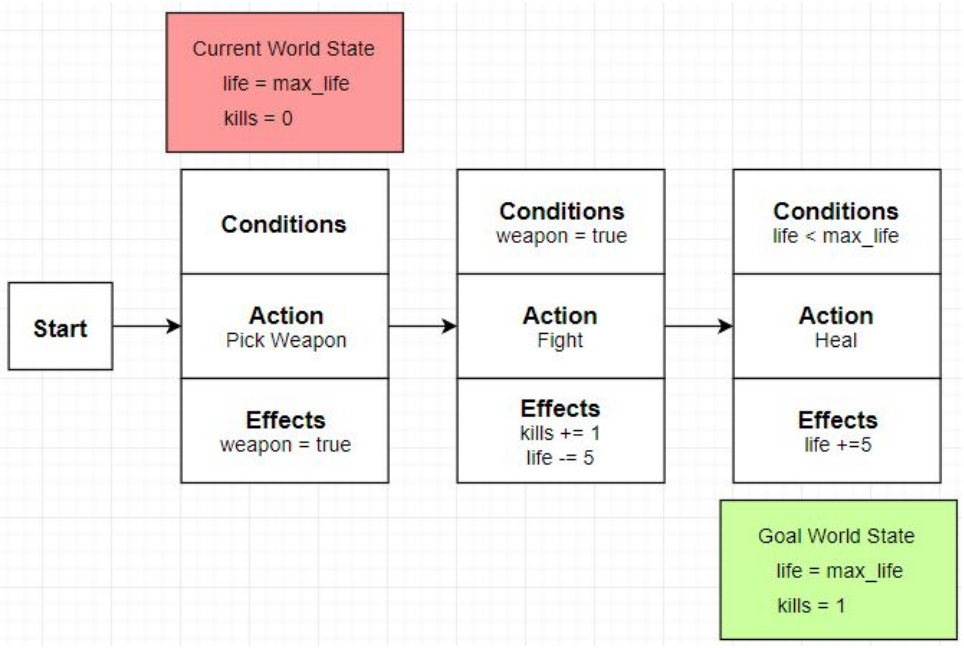

Figure 5.28 Exemple solution

The current world and the goal world state are the A and F nodes of the example, ant the tree action nodes are path nodes with different cost. The only big difference between the nodes example and the real implementation are the connections. In the example the connections are constant but in the GOAP planning the connections depend of the actions conditions. Therefore, every time the algorithm modifies the current world state the node connections are modified. For example, in the start situation the agent doesn't has the healing conditions but after executing the fight actions the healing condition appears as an available in the path.

Editor Elements:

Zoomable Canvas:

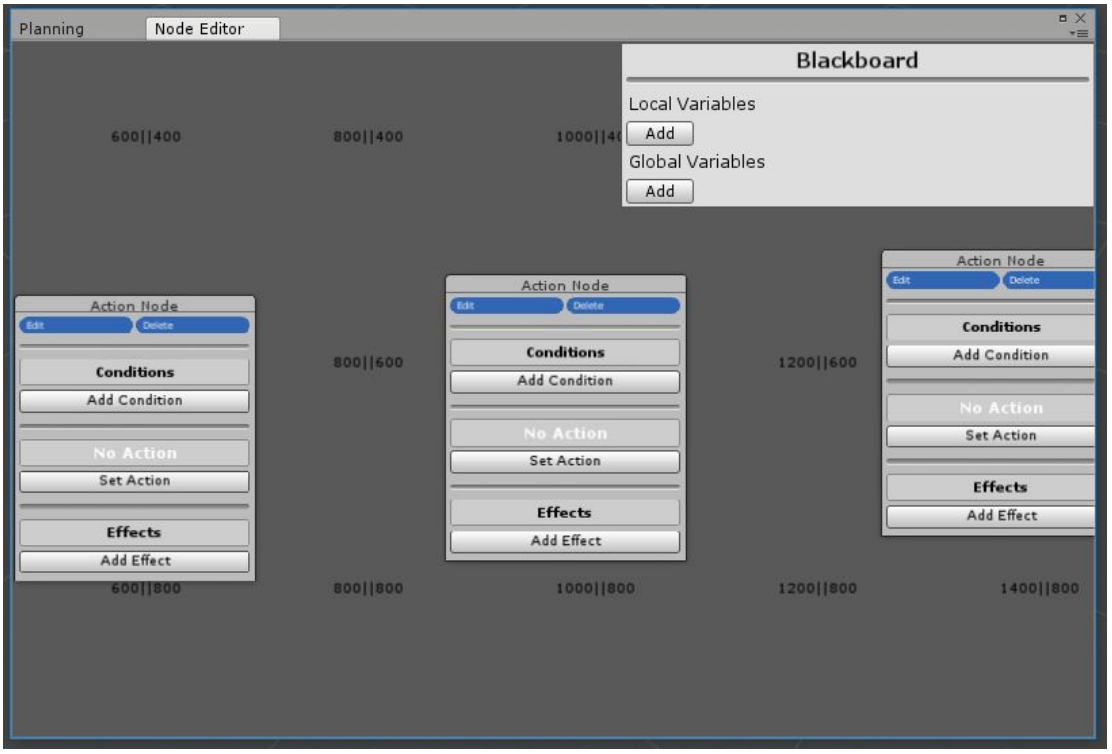

Figure 5.29 Zoomed out canvas capture

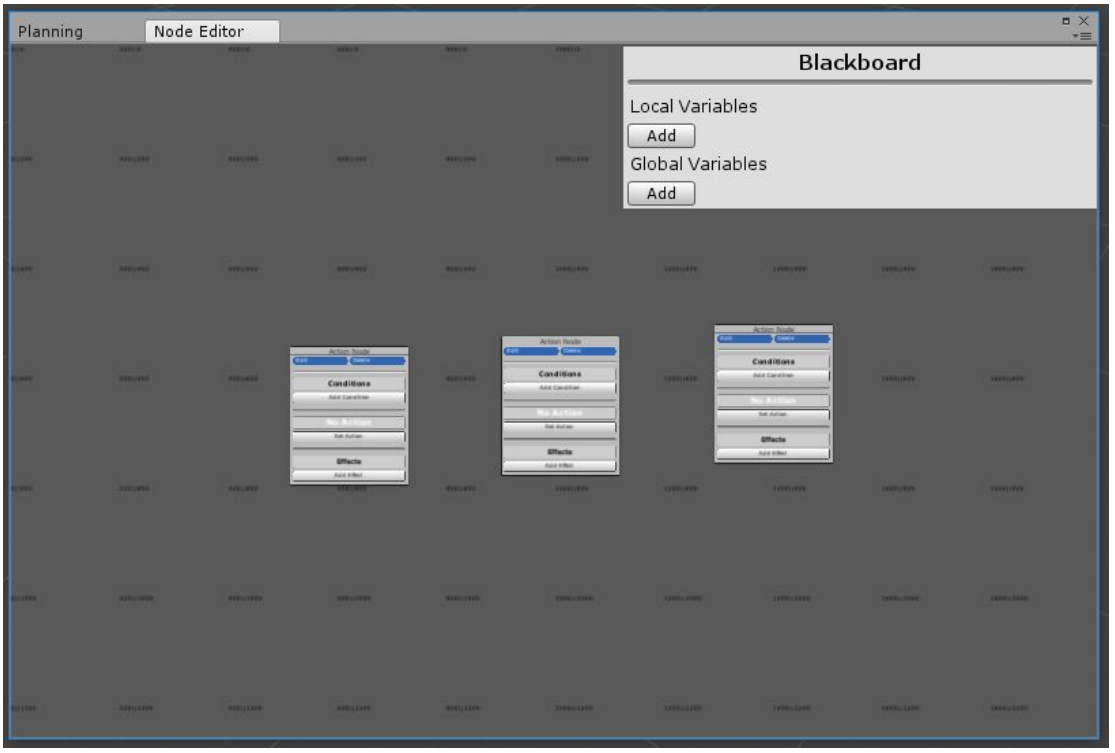

Figure 5.30 Zoomed in canvas capture

The zoomable canvas is an upgraded version of the initially plannified canvas with padding. The objective of adding zoom in the canvas editors is solve a usability problem.

To create interesting agent behaviours we need to define several action nodes and each of this actions nodes is represented by a window in the canvas. The user needs to have a global view of all the defined nodes at the same time he can work closely with a concrete node. Padding functionality is useful but don't gives the general view that the user needs. Zoom in and zoom out provide this point of view change that the user needs to control the node creation system comfortably.

Node Editor:

The node editor inherits from the zoomable canvas class. This canvas is focused in the action nodes anb blackboard definition.

The user can add new action nodes and define its conditions effects and actions, as extra functionality each action node has a name and description that allows to customize the system and made it more readable. The blue edit button of the action node editor changes the editor display mode to customization.

Figure 5.31 Node editor header

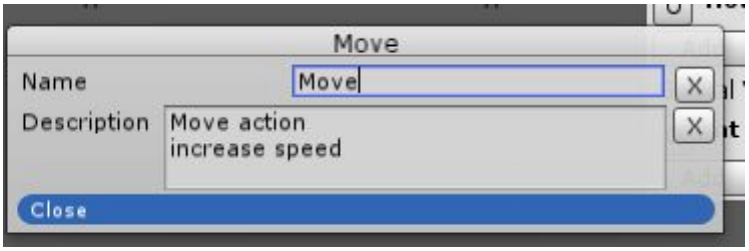

Figure 5.32 Node editor edit mode display

A part of the action nodes the blackboard with the local and global variables is displayed in the top right of the canvas window. We will comment the blackboard editor with more detail in the next paragraph.

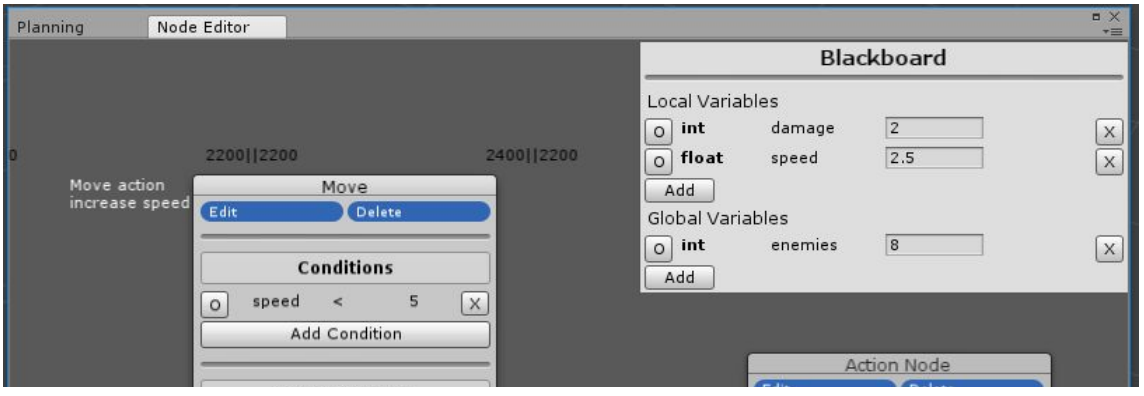

Figure 5.33 Node editor canvas capture

Properties:

The conditions and effects are basically property editors, a property editor displays the property name, operator and value. The editor has edit and delete buttons, they work identically to the variable editor buttons.

Figure 5.34 Properties editors in set mode

When the property editor is in edit mode we can modify the property value and operator. The value can be a new value or a binded value, to swap between the bind or unbind option we have to click the empty right button.

Figure 5.35 Properties editors in edit mode

Blackboard:

Blackboard editor shows the selected agent local variables and the global variables.

A the bottom of the local and global variables list there's the add button, if we pulse it the variables selection menu opens and we can define a new variable. The variable settings are the name, value type and value. We can choose between a new value or bind the variable with a value defined in a script. When the variable is defined we can pulse add and the variable will appear in the local or global variables, depending of which add button we have previously selected.

Figure 5.36 Blackboard editor

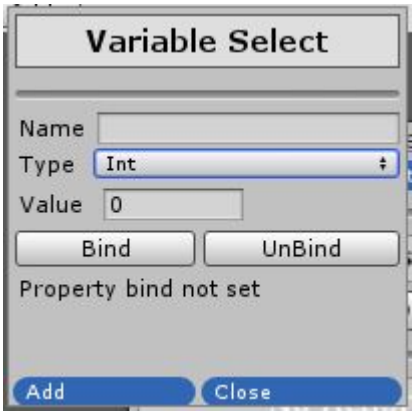

Figure 5.37 Variable selection menu

The variables have a edit and delete button, the edit button allows to change the variable name and the delete button removes the variable.

The previously commented editor elements are advanced versions that practically offer all the desired functionalities, but this Vertical slice version do not include the behaviour and planner development. Therefore, the next editor elements commented are placeholders with only the most basic functionalities.

Planning Editor (basic version):

This editor only displays the behaviour editor and shows the action nodes of the current path. For the moment there is no interaction a part of the behaviour selection or creation.

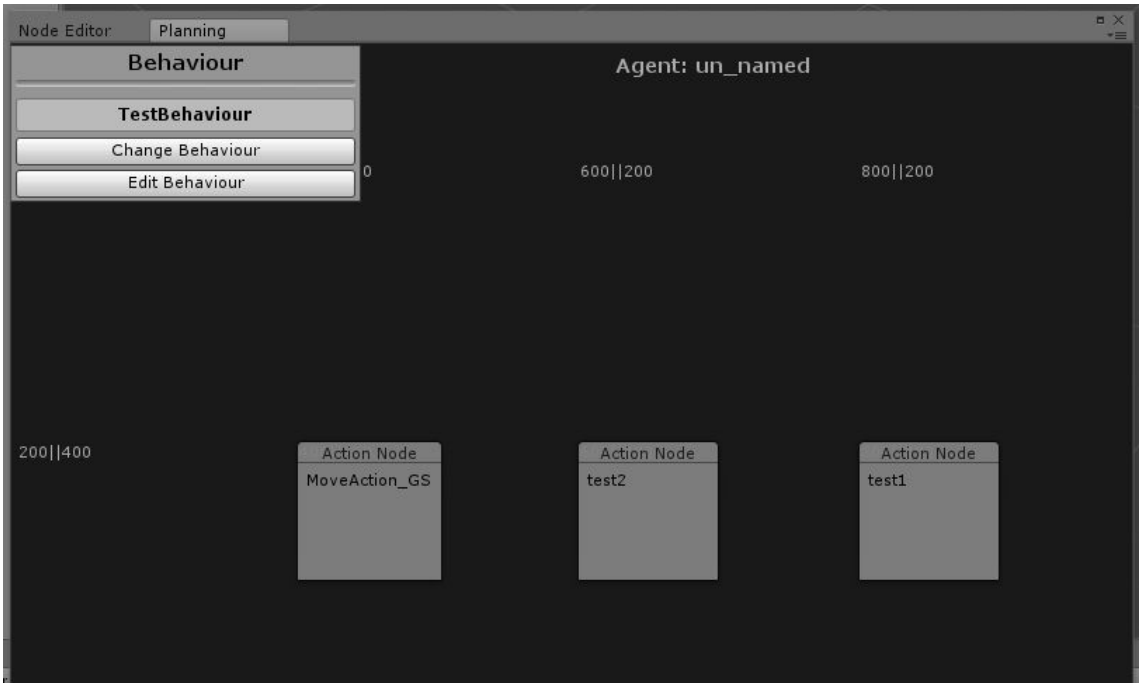

Figure 5.38 Planning editor canvas

Behaviour (basic version):

The behaviour editor let you set the behaviour that will follow the agent selected. When the behaviour is not set we can open the behaviour select menu and select a defined behaviour or create a new one. Once the behaviour is set, we can select between edit the current behaviour or select a new one. The edit behaviour button opens the selected behaviour with the engine code editor, the change behaviour button opens the behaviour select menu.

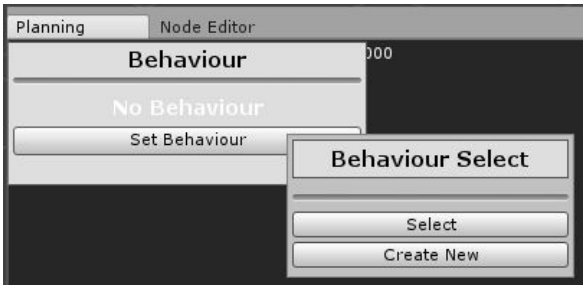

Figure 5.39 Non defined behaviour editor

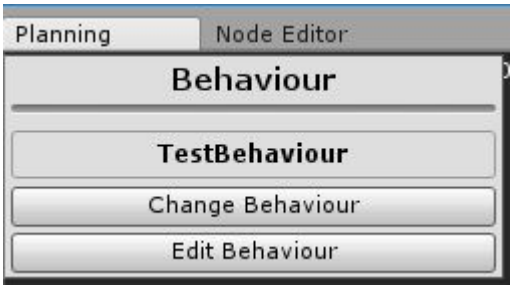

Figure 5.40 Defined behaviour editor

5.2 Alpha:

The alpha stage involves the development of all the logic elements and editors. Therefore, at the end of this development point the tool is practically finished. Beta stage is only polish and testing.
Time to briefly comment the upgrades developed in this stage. As we have done in the previously commented stage, the content is divided in logic and editor elements.

5.2.1 Logic Updates

Global blackboard:

The variables management system has been upgraded with the implementation of a global blackboard that all agents share and modify freely. A global blackboard allow agents communication using the variables defined in it. In other words, now users can generate grupal behaviours that react to the variables defined in the global blackboard.
The global blackboard uses the same structure as the agents local blackboard and is displayed in the node editor canvas at the bottom of the blackboards window.

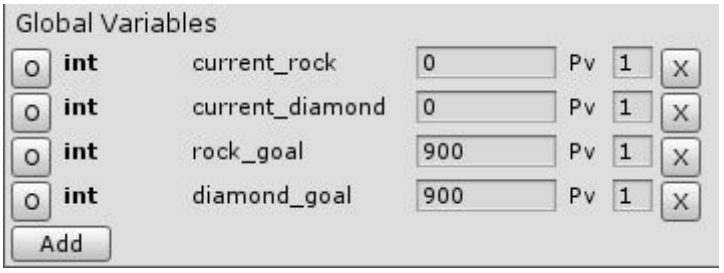

Figure 5.41 Global blackboard capture

Method Binding:

Variables now can be binded to methods defined in the scene. To understand the potential of this new implementation we need to remember the variables binding system commented in the vertical slice stage.

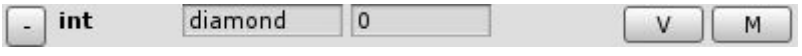

Figure 5.42 Variable editor with variable and method binding option

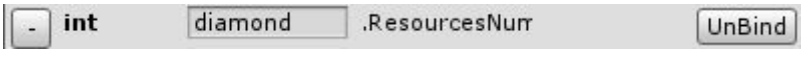

Figure 5.43 Method binded variable editor

Variables binding means that the variables defined in the blackboard can be binded to variables defined in scripts instantiated in the scene. When a blackboard variable is binded, it has the same get and set methods of the linked script variable. Therefore, when you are interacting with the blackboard variables this interaction is applied to the linked script variable.

The new method binding system allow users to bind blackboard variables to methods defined in instantiated scene scripts. This binding system only supports getters, that means that when a blackboard variable is binded with a method you can't modify its value, you only can get it.

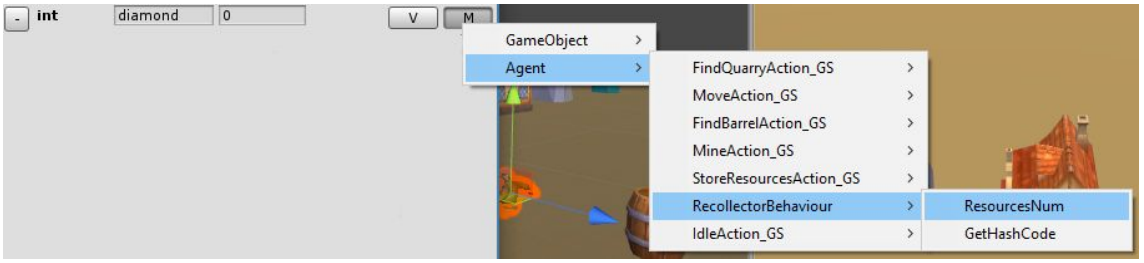

Figure 5.44 Variable editor with the method binding selection system

The important and useful part of this implementation, is that a method is a combination of operations that can use input to generate an output from it. The input of the method can be simple values like integers or floats, but what happens if we use blackboard variables as input for a variable binded with a method?
Here we can see the potential of this new logic implementation.

Using a simple method definition menu, we allow users to define the binded method input using values or other blackboard variables. This means that a method binded variable can use another method binded variable as input at the same time that this method binded variable uses another blackboard variable binded to another script variable... and all the binds and combinations that we can imagine.

Behaviour:

The behaviour scripting system now supports all the global blackboard implemented functionalities. In other words, agents can define global goals using the variables of the global blackboard. Users can access to the global blackboard using a property of the base behaviour script named "global_blackboard" that returns the instance of the current global blackboard.

In order to define global or local goals, the "SetGoal" method has been improved with a local/global enum. Depending of the enum input value, the property defined inside the "SetGoal" method will interact with the local blackboard or the global one.

```
SetGoal("current_rock", OperatorType._equal_equal, 5, VariableLocation._global);
```

Figure 5.45 SetGoal method defining a global goal

The last behaviour update is the definition of a idle action. In the previously commented vertical slice version, the agent behaviour stays in no-action state when all the planned actions are completed and there's no new goals to generate new plans. Now the agent no-action state has been replaced by the agent idle state, which means that users have to define a idle action that will be executed when the agent is in this new state.

Figure 5.46 Behaviour editor with idle action editor

This idle action system gives a more natural behaviour to the entire system, because an intelligent entity never stops completely, there's always a little bit of movement/reaction meanwhile it is alive.

Planning value:

The planning algorithm has been upgraded with a new variable named planning value or Pv. This new value is used by all the local/global blackboards variables during the actions path generation. Inside the planning algorithm is used to calculate the logic value of the distances between different world state properties.

For example, imagine that in the world state A we have a boolean value named X with a value of false and in another world state named B we have the same boolean property but with a value of true.

World State A [X = false]
World State B [X = true]

The distance between different world states is the sum of the distances of all the properties defined by the two world states. During this operation we use the planning value to define the distance value of each property multiplying the initial distance by the defined planning value.

In this case the absolute distance between X in world state A and X in world state B is 1.

Distance = Abs(X(false) - X(true)) = 1

Looking at this formula we can see that without the planning value all the boolean values are always limited to a max distance value of 1, because boolean values only can be 0 or 1. Now that we multiply the distance by the defined planning value the distance limitations disappear.

*Distance = Abs(X(false) - X(true)) * Pv = y*

But which is the real use of the distance modification functionality?

The planning algorithm always execute the action that generates the nearest world state to the goal. Therefore, the goals that involve variables with high planning values will be the first ones to be executed. In other words, when we change the planning value of a variable we are changing the priority of the goals that use it. High planning value means high priority meanwhile low planning value means low priority.

5.2.2 Editor Updates

Action editor:

Action scripts are defined by users and can contain all type of fields and properties. In order to make the action scripts more customizable, we have developed a visual editor to modify the public values defined in the action scripts.

Action editors show all the action script public fields and properties. The value types supported by this editor are the next ones: Integer, float, boolean, char, string and vectors. This means that all the script variables with a non supported value type appear in the editor but the variable value is not editable.

Figure 5.47 Action editor with properties and fields configuration

This visual editors are displayed in the node editor canvas when the user press the show config button of a node with a defined action script.

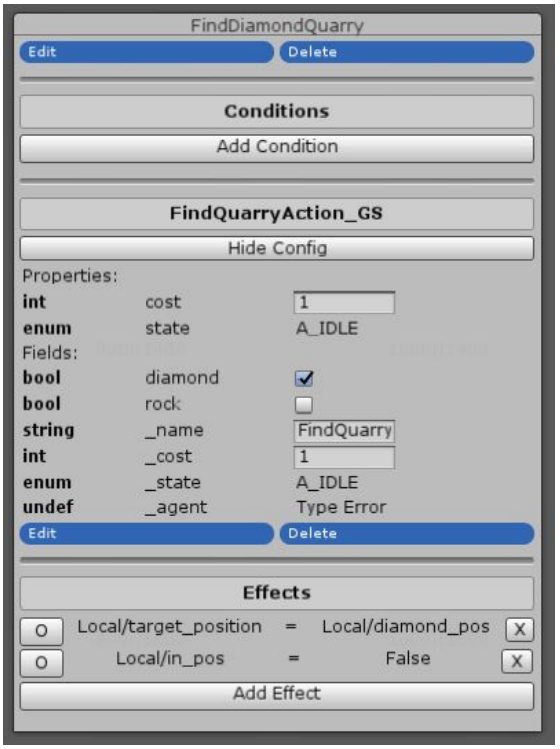

Figure 5.48 Complete action node editor capture

Debugging:

The planning canvas now displays the generated action planning and all the necessary information about the actions involved in the action plan.

Action scripts can be modified by the user and it is impossible to know which variables are going to have and which ones have to be displayed in the debug window. Therefore, the action script has a debug method that can be overwritten in order to let users customize the action debuggers.

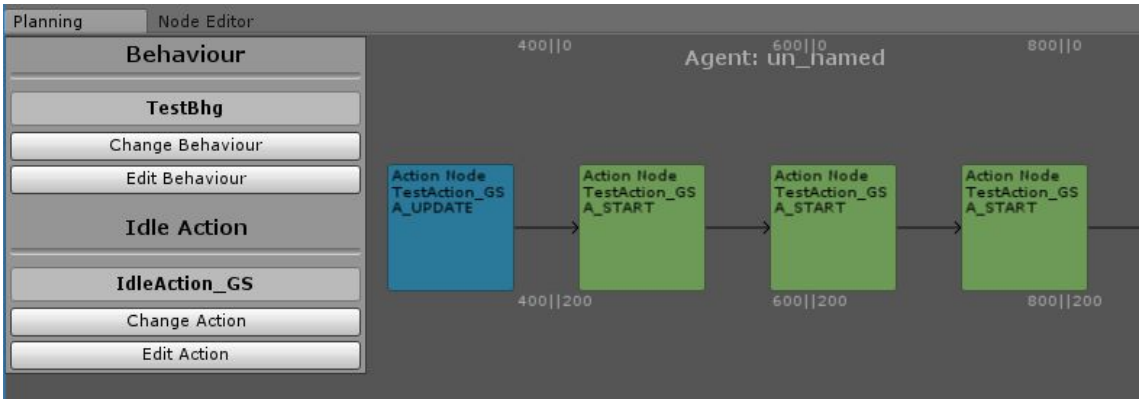

Figure 5.49 Planning canvas with plan actions debuggers

A part of the important variables, the action debuggers also change the window color depending of the action state. The implemented action states are the next:

- Start: Green
- Update: Blue
- Complete: Highlighted green

5.3 Beta:

Beta stage focus on iterate and test all the implemented functionalities meanwhile we develop the exemple scene. Let's briefly comment the developed scene structure and all the mechanics that we use in it.

The exemple scene is about a team of recollector agents that have to get all the materials that a mayor agent requests.

Figure 5.50 Exemple scene screenshot

The mayor agent behaviour is extremely simple because it only defines the global goals for the recollector agents and when the goal values are reached the agent resets the blackboard values to start the recollection cycle again.

On the other hand, we have the recollection behaviour and the recollection actions system, that use all the tool functionalities.

Recollector behaviour update gets the previously defined resources goal from the global blackboard and checks if the goal is completed or not. In negative case it defines a global goal to equal the current resources number to the goal resources number.

The next screenshot shows a general view of the recollectors node editor action set. In other words, all the actions, variables and properties that the recollection agents use to generate the final behaviour. It is not possible to appreciate all the details from a general photo so we are going to describe the most important ones in terms of tool functionality.

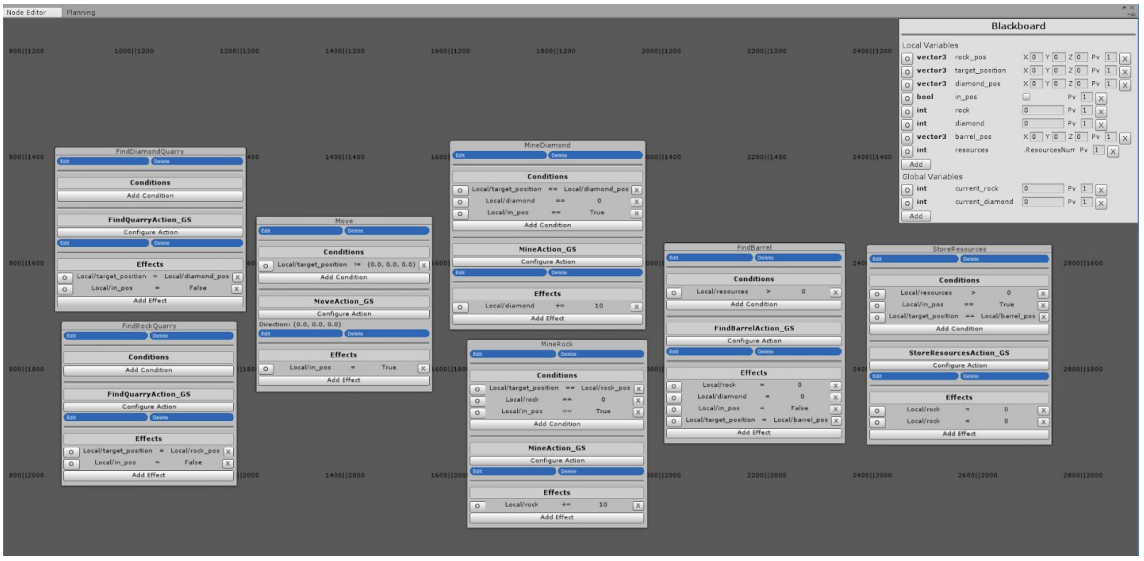

Figure 5.51 General view of the recollector agent action set

Action configuration:

As we have seen before, the action editor let us configure any action changing its fields and properties values. In this case, the recollector agent uses this functionality to change the "FindQuarry" action behaviour. This action has two boolean values, "diamond" and "rock", to define which type of quarries is the agent looking for. If we set "diamond" to true the action will look for diamond quarries and if we set the "rock" value to true the action will look for rock quarries, we can also set both values to true and get all the diamond and rock quarries in scene.

Figure 5.52 Exemple scene find quarry action configuration

This is a simple example of how useful can be the action configuration functionality to adapt actions to different agent behaviours and avoid the generation of practically equal scripts.

Multiple Properties operators:

Sometimes actions interact with several blackboard variables and we need to modify all of them when the action ends. In this scene we have implemented a "StoreResources" action that gets all the resources collected from the local blackboard and sums them to the desired global blackboard variables. After doing that the local resources and position variables are set to zero in order to reset the agent for the next resources recollection cycle.

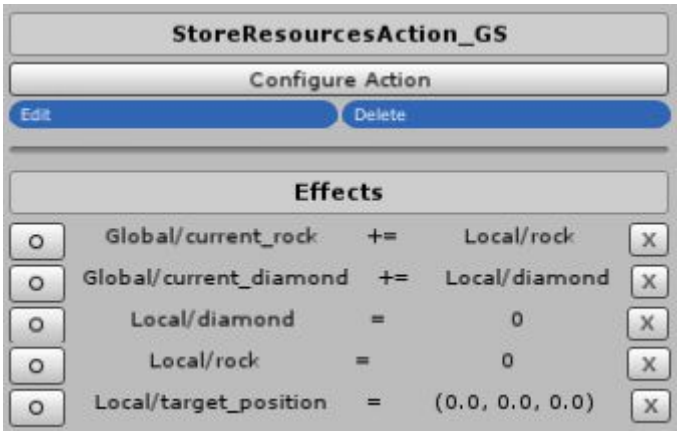

Figure 5.53 Exemple scene case of multiple property operations

Method binding:

In this exemple scene we have used the method binding functionality to execute a basic operation every time the planning system needs to know how many resources is carrying the targeted agent.

Figure 5.54 Exemple scene method binding variable

```
public int ResourcesNum()
{
    return blackboard.GetValue<int>("rock") + blackboard.GetValue<int>("diamond");
}
```

Figure 5.55 Exemple scene method binding method

Inside the agent planning, this variable is used by the “FindBarrel” action in order to check if the agent really has resources to store in the barrel. In other words, it checks if the sum of rocks and diamonds is bigger than zero.

Action debugs

The action scripts used in this example scene are very simple and some of them are only timers that simulate the execution of a real action. In order to really use all the tool functionalities, we have defined the customizable debug methods of the most complete actions. Like “MoveAction” and “FindQuarryAction”. On the first case the custom UI shows the current agent position, the target position and the distance to the target. On the second case our custom UI simply displays if the action is looking for diamond quarries or rock quarries.

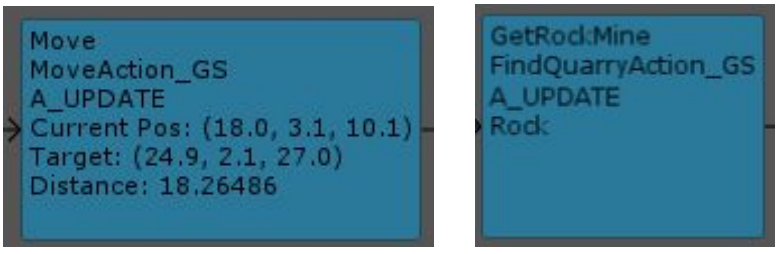

Figure 5.56 Action custom debuggers of the exemple scene

5.4 Gold:

This is the last stage and all the project tasks are practically done. At this point the tool is completely developed and the memori documents all necessary information about it.

The last task consist in publish our tool on the unity assets store and github, and wait for users feedback. We need feedback to check if we really have reached the final objective of develop an extremely easy to use and functional tool.

Before publishing the tool we will briefly comment the implemented functionalities. In order to know if we have reached all the objectives defined at the beginning of this project. The functionalities completely implemented will be only mentioned meanwhile the non-completely implemented will include a brief explanation.

Initial objectives:

Solid and functional GOAP logic:

- Well structured code with instructions ✓
- Blackboard system that supports several variable types ✓
- Variables and methods binding system ✓
- Planner class that uses A* algorithm to generate action plans: The planning algorithm is implemented correctly but not supports multithreading, which means that if we have several agents in the scene and some of them have to calculate the actions paths at the same time we can see how the framerate drops down during the needed milliseconds for the operations. In order to notify this performance issue to the users, we display an error message when the frame rate is under 30 fps when an agent starts the action planning process.

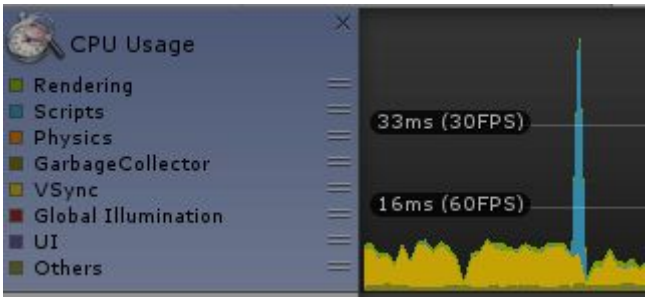

Figure 5.57 Tool CPU performance graphic from Unity profile

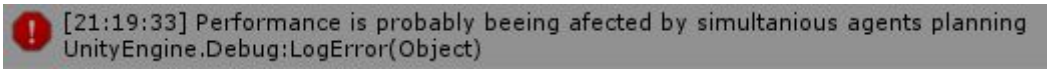

Figure 5.58 Tool performance alert on Unity console

UI support for all the tool elements:

- Zoomable canvas that provides a smooth navigation ✓
- Several sub-menus to build all the implemented logic elements like variables, methods, scripts and properties ✓
- Action nodes editor that let users configure all the desired actions adding conditions and effects ✓
- Easy to use UI with tooltips ✓

Generate a useful process output:

- Planning canvas that displays the generated plan and all the important information about it ✓
- Customizable action debuggers ✓

Provide a complete documentation:

- Tool wiki that shows all the elements and functionalities ✓
- Constant access to the tool wiki ✓
- Exemple scene that uses all the implemented functionalities: The exemples scene is extremely simple and there's only one real agent behaviour developed. The scene is enough to show how the tool works but with more time we could have developed a more complete scene with more agent behaviours.

Now that we know the current situation of the tool and all its characteristics is time to release it and wait for feedback.

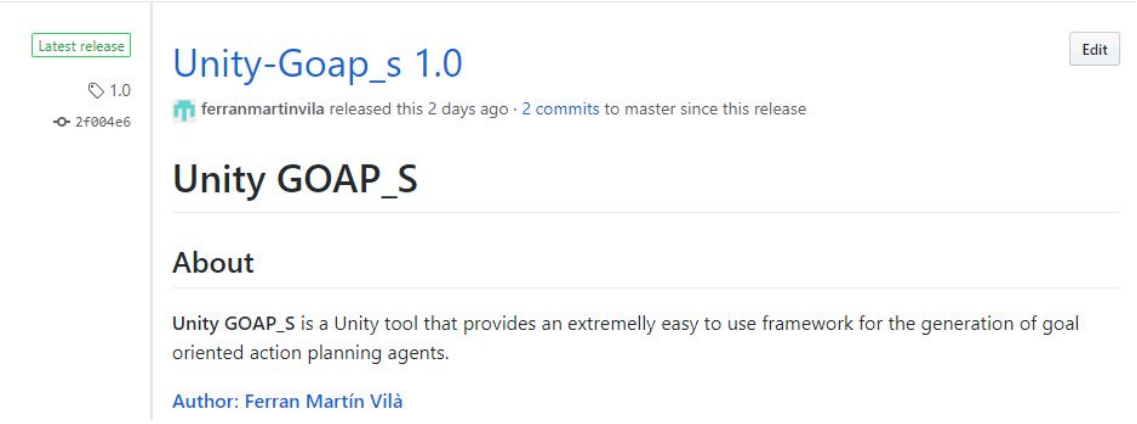

Figure 5.59 Tool github release capture

Github releases <https://github.com/ferranmartinvila/Unity-GOAP_S/releases>

Unity release is under Unity team approvement process. Therefore, is uploaded to the assets store but not appears publicly.

6. Conclusions

This project starts with an intense research about GOAP architecture and the planification of all the tasks needed to implement the mentioned architecture in a Unity framework. We aim the objective of develop a competent tool that provides all the GOAP functionalities with the facility of visual editors and debugging.

After the research process we start the first development stage named vertical slice, during this stage we notify that some tasks like the basic agent generation were far shorter than we initially planned. This planning error helps us to end the first development stage with a more complete framework than we expected.

The next development stage named Alpha focus on develop all the logic elements of the tool and all the necessary UI elements to support the implemented functionalities. During the development of this logic elements we realize that to generate a framework that supports all the GOAP functionalities we must implement more logic elements than we previously planned. One big example of implemented logic element that was not initially planned is the global blackboard. The extra work of this new logic elements activates the contingency plan and we decide to not implement multithreading in the planning algorithm. The final result of this contingency plan is that with several agents the app performance can be affected by the agents planning process.

During the last development stage named Beta we basically iterate all the implemented logic elements. We also generate an example scene to show users how to use the tool at the same time we test it with real implementations.

Finally, the gold stage focus on finish the memory and publish the tool in the desired platforms. In this case the desired platforms were github and the Unity assets store. Right now the tool is uploaded to both platforms, but in the Unity assets store when you upload an asset it must be revised and approved by the Unity team before it becomes public.

After this general resume commenting the most remarkable experiences, we can proudly say that all the objectives presented at the beginning now are completely implemented. There is only one important feature that has not been implemented, we are talking about multithreading.

Therefore, multithreading and general performance optimization are two objectives that we aim as future work. Changing the planning algorithm and adapting it to the multithreading architecture is a really interesting work that we will cheerfully do during the next months after this project release.

Github release under MIT license
<https://github.com/ferranmartinvila/Unity-GOAP_S/releases/tag/1.0>

7. Bibliography

Nils J.Nilsson. 1998. Stanford University. Artificial Intelligence: A New Synthesis. pp. 1-6
<https://books.google.es/books?id=GYOFSd6fETgC&printsec=frontcover&hl=es&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false>
Consultant 12 Feb, 2019

David-Pittman. 2005. University of Nebraska-Lincoln. GOAP-Masters-Thesis. pp. 12-20
<https://www.dphrygian.com/bin/David-Pittman-GOAP-Masters-Thesis.doc>
Consultant. 14 Feb, 2019

Philip Bjarnolf. 2008. Högskolan Skövde. Threat Analysis Using Goal-Oriented Action Planning, Planning in the Light of Information Fusion. pp. 11-19, 26-35
<http://www.diva-portal.org/smash/get/diva2:2228/FULLTEXT01.pdf>
Consultant 17 Feb, 2019

8.Webography

Unity Public Relations.
<https://unity3d.com/public-relations>
Consultant 11 Feb, 2019

Quora. How many game programmers use Unity.
<https://www.quora.com/How-many-game-programmers-use-Unity>
Consultant 11 Feb, 2019

MIT Media Lab. Jeff Orkin. GOAP.
<http://alumni.media.mit.edu/~jorkin/goap.html>
Consultant 11 Feb, 2019

EnvatoTuts+. GOAP for a smarter AI.
<https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793>
Consultant 13 Feb, 2019

Unreal Engine. Unreal Engine Marketplace.
<https://www.unrealengine.com/en-US/blog/epic-announces-unreal-engine-marketplace-88-12-revenue-share>
Consultant 14 Feb, 2019

Gamefront. F.E.A.R SDK.
<https://www.unrealengine.com/en-US/blog/epic-announces-unreal-engine-marketplace-88-12-revenue-share>
Consultant 14 Feb, 2019

Wordpress. Chris Powell.
<https://cbpowell.wordpress.com/about/>
Consultant 15 Feb, 2019

Github. cpowell. cppGOAP.
<https://github.com/cpowell/cppGOAP>
Consultant 15 Feb, 2019

Stolk.org. Abraham Stolk.
<https://stolk.org/>
Consultant 17 Feb, 2019

Github. Stolk. GPGOAP.
<https://github.com/stolk/GPGOAP>
Consultant 17 Feb, 2019

LinkedIn. Luciano Ferraro.
<https://www.linkedin.com/in/luciano-ferraro-9046997a/#experience-section>
Consultant 20 Feb, 2019

Github. luxkun. ReGOAP.
<https://github.com/luxkun/ReGoap>
Consultant 20 Feb, 2019

Github. crashkonijn. GOAP.
<https://github.com/crashkonijn/GOAP>
Consultant 21 Feb, 2019

Unity. Assets Store. GOAP AI.
<https://assetstore.unity.com/packages/tools/ai/goal-oriented-action-planning-artificial-intelligence-72912>
Consultant 21 Feb, 2019

Software Testing Help. Agile Scrum Methodology.
<https://www.softwaretestinghelp.com/agile-scrum-methodology-for-development-and-testing/>
Consultant 25 Feb, 2019