Goal Orientated Action Planning Artificial Intelligence

Sam McKay Illiyan Georgiev José Díez Vlad-Eugen Tanase
Aaron Swiss-Hamlet

April 24, 2015

Contents

Chapter 1

Introduction

Project 10 of the 206CDE Real World Projects, namely Goal Orientated Action Planning (GOAP) for Game Artificial Intelligence (AI) was successfully finished by this group. In this project, we will discuss the GOAP artificial intelligence type and its uses within video games. This project was complete in eight months by a team consisting of five members. Due to the majority of the team belonging to the Games Technology course, the overall look and feel of the project is similar to a two-dimensional pixel art style video-game, although no player interaction is possible.

GOAP AI refers to a simplified STRIPS-like planning architecture, designed for real-time control of various character behaviours within video games. A good example of a GOAP AI is the F.E.A.R. video-game AI, which has won many awards and is acclaimed as one of the best gaming AIs. Various programs were used in the making of this project. The coding behind everything was done using the C++ programming language. Photoshop was used to develop various textures and sprites to be used in the game-like presentation. For graphical purposes, the Simple and Fast Multimedia Library (SFML) was used, which is based on the Open Graphics Library(OpenGL). Axosoft and GitHub were used for team organisation and progress tracking.

Chapter 2

Project Brief

Our brief for this project was to write an artificial intelligence for games using Goal Orientated Action Planning techniques. We had to produce an AI agent capable of planning and executing a simple sequence of actions. This should include programming, software engineering and design as its key areas of study in order to complete the project.

To successfully carry out the project certain job roles must be filled, these are to include:

- Project Manager
- Lead Programmer
- Lead Designer
- Programmers

At the end of this project we are expecting to be able to deliver a simple game world containing an AI agent capable of planning simple sequences of actions and executing these in real time.

Chapter 3

Aims and Objectives

The main aim of this project is to fulfil the brief of creating a Goal Orientated Action Planning Artificial Intelligence agent. This agent should be able to asses its current state against its goal state. Our AI agent should then begin to generate an action plan, that it will execute in order to achieve the desired state.

- Design and Create a Survival Game
- Create an Artificial Intelligence Agent capable of making an action plan
- Have the Artificial Intelligence Agent carry out the plan

To demonstrate this we set ourselves the objective of designing and creating a survival game in which the main character is our AI agent. This world should be a desolate environment with multiple biomes and various resources all scattered about it, all of which is randomly generated to ensure our AI agent has a new challenge every time. With this world we envisioned in mind our next objective was to create a list of relevant actions for our AI to carry out. These actions should have pre and post conditions such that our AI could create multi action plans in order to complete a task such as satisfying its need for hunger. To be able to generate these plans we would need to create a planer algorithm, for this we would use similar logic to the STRIPS implementation around which the majority of our research was based. Finally our last objective is that we need to be our able to visualize our AI agent planning and carrying out these plans. To show this we would use the SFML library to graphically represent the aforementioned environment, along with a user interface to show the AI agents "thought process".

Chapter 4

Project Management

In order for this project to succeed we needed to implement an effective project management methodology. For this we chose to implement the agile project management structure commonly used within the software industry. Due to the size and length of the project it was imperative that we implemented good project management from the beginning, without it we could have struggled to complete the project in a timely, well paced fashion. By ensuring we had this project management in place from the start it meant that we could track our progress and continue to work at a reasonable pace throughout the duration of the project.

Agile project management is a highly effective method of managing long term projects, the process can be broken down into many stages. To begin with you collect all of your user stories, or feature requests, into a product backlog. These user stories are mostly generated by members of the team and define the "wish list" of features to include in the project. Once we have all our user stories we can begin selecting the features we will include in the final product into a release backlog (see Appendix 1.01). This release backlog can be broken down into various stages known as sprints, these sprints can be as long as a couple of days up to a whole month. All of these sprints come together to create the full product with each sprint generally expanding upon the work completed in the previous sprint.

In order for teams to easily implement this project management methodology, various different companies have all produced their own software to provide the agile management approach. We chose to use *Axosoft's* Scrum software, this creates a personalised website for the team to use and grants access to all of the previously discussed features. Using this we quickly created all of our user stories and proceeded to arrange these into various different sprints. We decided upon five sprints for the project: Research, Planning, Design and two programming sprints. This helped us to ensure that we kept on track as each task as an estimate for how long it will take to complete. So as work is done on each task we attach a work log to it and can compare estimated time with the actual duration of the task and use that as a measure of when we will finish the project as a whole using a burn-down chart (see Appendix 1.02)

By using Scrum we were able to monitor our progress and even predict release dates, which helped to give us something to aim for. We were also able to track who was working on what and exactly what work has been done on each section as well as what is left to be done. This meant that we didn't have two people each working on exactly the same task and prevented wasting time by doing so. Without Scrum it would have been very challenging to know exactly what stage we were all at and what needed to be done to meet the next deadline. Scrum also helped with our testing process by providing a bug tracker (see Appendix 1.03), allowing us to easily track any bugs and to test new code as it was completed.

We also made use of GitHub, a code sharing repository, in order to store and share our code amongst the team. This allowed us to organise the code and maintain version control whilst also enabling the team to work on various sections at the same time. Throughout the project weekly meetings were attended, Monday with the project supervisor. Team meetings were established by the project manager based on necessity and the availability of every member, minimum once a week. On various occasions, more than one meeting was booked within the library to work, discuss or establish new tasks.

Chapter 5

Methodology

5.0.1 Design Methodology

The design methodology has meant implementing a process whereby the entire team has had some input into the creation of the program's design, both structurally and graphically. In the initial research stage of the project, this meant looking up existing user interfaces (referred to as the UI from now) and determining what kind of art style the program would have to determine the best way to present the information to the screen. Interfaces that were looked at include the World of Warcraft system and how its pre-set setup was not favoured due to a lack of specific information and that of The Sims, where the needs of each sim are covered in the simple design of the UI. Fitting the graphical design to the needs of the project requirements, taking an approach much like The Sims seemed appropriate as this provides a simple and easy to understand system for users to see. The next aspect to approach on the design aspect is that of the world's graphical look. This needed to compliment the UI design. Due to the amount of time available for the entire project as well as the size of our team, a 2D world approach made more sense, which formed the basis of our initial strategy for approaching the project.

Once past the research and initial concept stage, the design implementations need to be coupled with the programming side of the project in the planning and prototyping stage. This meant talking as a group, with each member of our team to come up with the best solution to the design challenges presented. These include showing elements of the AI's thought processes to the screen. Furthermore, aspects of the AI itself that affect its planning and how to represent the items it has on its person to use. Approaching the thought processing aspect, there are many elements that the AI could need to do, for example, obtaining wood or needing to sleep. Therefore, as there are a large variety of different actions that the AI can perform, trying to graphically represent everything in a list is not a good method of display. Therefore the next best method would be to display its thoughts in a console statement window.

The next element to address in the planning stage is the way in which the AI will have elements that affect it displayed to the screen. This means visually representing to the user what exactly is making the AI decide to do that action. As this design aspect includes programming limitations, both sides of the project needed to come together on this segment. Too many elements, which we decided to call "weights", would make programming the AI too hard for the timeframe we had. On the hand too little would not give the AI anything to work on or react to within the environment. Therefore, we decided upon using only the core aspects of a human as the weights which we would include in the program. As such, four of these are included in the final simulation: hunger, thirst, energy and warmth. These were whittled down from a larger list of weights which were the initial criteria we were going to go with however, this makes for a much simpler AI system that fits the expected project outcomes. In order to demonstrate these weights to the user, we chose to use a coloured background that changes under certain conditions. This was favoured over using a bar that changes in size as this is an obvious visual representation that is instantly clear to viewers.

The final element of the UI was to display what the AI has on its person, whether it be food or a material used in creating a fire or shelter. This was much like the appearance of the weights on the UI as they can be used with a simple tile that sticks out from the rest of the UI around it. As evidenced in the final design proposal, there are tiles intended for inventory space with the ability to remove them easily as needed due to not interacting with any other element in the program's environment.

The final graphical design challenge is the creation of the 2D environment that the AI will work within. This requires a simple implementation of images that act as "tiles" in the world to create a randomly generated world where there are a selection of biomes across a fixed space environment. To create these biomes, an xml map generator is created using a sub program that denotes which biome is where, and the image that represents the biome on that tile will be displayed based on the programming implementation. Once the map is implemented, the graphics are placed in the world and then the program generates the world. On top of the terrain tiles that are produced, there are .png files which denote resources in the world, such as trees and stone, that are placed on

top of the terrain under conditions that the resource has to have fulfilled before it spawns in that tile. Keeping the amount of resources in the world was essentially due to the timeframe and also for the implementation of other aspects in the program on the technical side. These tiles are equal in size, being 64x64 pixels and take a uniform art style which, while simple and not professionally impressing, still presents the world in an obvious manner to viewers of the program.

5.0.2 Development Philosophy

At the beginning of the project a group decision was made to maintain modularity and independence between most major parts of the implementation. This was done in order to avoid some common pitfalls associated with working in a team like having to wait for one person to finish his work for another to begin. This philosophy also insured that each component of the final project would function individually and be at a completed state as quickly as possible. The con is that in the case that we lose a team member during development the code / work he has done may be difficult to recover and / or figure out in order to complete implementation. Due to the fact that we had only three programmers for the duration of the project coordinating between them was not that difficult.

5.0.3 Code Structure

The overall structure of the code was based on the system diagram however we did not keep strictly to it. Requirements and limitations that arose during development meant that some of the parts of that diagram became irrelevant, unnecessary or unpractical. The final implementation maintained modularity and plugging in new features was easy and almost effortless. However as the project evolved and changed implementing new functionality into the existing framework and connected components became harder. This was a result of modularity. Interfaces between each component were often defined at the moment when the component was added to the framework. In some cases the contents of the compound data types had to be changed in order to make implementing a specific interface easier.

5.0.4 Navigation

As we decided to not only implement a planner but to create a world for our AI to reside in it had to have a means to move around. The two obvious options explored were pathfinding and steering. The design of the world meant there would be no situations in which the agent will need to navigate negative spaces. Knowing this we decided to avoid pathfinding and used steering instead. A basic collision avoidance steering behavior was implemented. Based on angles it steers around obstacles which are in front of it and in its path. This was an ideal and sufficient movement mechanic for this project however research into different pathfinding techniques was conducted. As with other parts of the project the navigation underwent several changes to accommodate for alterations in the rest of the project.

5.0.5 Simple and Fast Multimedia Library

Using this OpenGL based API we insured quicker implementation of all visuals associated with the project and avoided having to develop basic features ourselves. Basic file type loaders, vector data types, canvas and view port management are just a few of the features we used which sped up creating the visuals of our project. Using this library we had to work around and comply with some of its conventions and concepts. While we did run into issues and setbacks none halted of them were too major and usually got resolved within minutes. Mostly these issues were associated with our unfamiliarity with SFML. Going through the documentation was the most common way we dealt with these. A good example of such a problem is the way SFML draws sprites on the screen. We expected this would be done from the center compensating for the size of the sprite. In reality the coordinates given would be the top left corner of the sprite. This made the collision avoidance look faulty and caused us to look for a solution within the steering code itself. As this was one of the first problems we faced it set a trend for most issues we encountered from that point on and fixing them was much easier and quicker.

5.0.6 Map Generation

For this project, a random map generator has been created. The size of the map in height and width is necessary, as well as the number of biomes to be created. Afterwards, each biome must have an individual, distinguishable character such as "#" or "*" and a limit of spaces this biome can take. The generator sets the starting position of each biome, with a minimum distance between each biome. The map starts with a default empty biome, drawn with the blue colour. After each position is set accordingly, the drawing starts from the middle, using the limit to take the spaces and checking if the spaces are not used by any other biome except the default one.

At the end of the generation, the map is shown using the SFML for graphical representation and an eXtensible Markup Language file is generated, containing the number of biomes, height and width, as well as every tile data type.

The Resource generation program imports and reads the XML file in order to determine the map and biomes associated to it. Each resource has a limit as well, and can only be assigned to one biome type. Once these are set, a random number is chosen between 0 and 3. With this number, the generation of the map will start from a specific corner of the map, such as 0 being top left, 1 being top right. Each resource is constrained to be spawned on an empty space and not have any other resource in its adjacency. A random number is generated every time a suitable position is found, and if it is 1, the resource is placed. A different XML file, containing the resource information of the map is generated at the end. With this imported into the main project, we have a randomly generated map populated with resources for the AI to use.

This was done in order to test the AI in different scenarios, in random maps with randomised resources in order to get different results, actions and plans from the AI. Even though every generation is random, some control is still available through coding if a resource or biome type should fit a certain criteria is needed, as well as the ability to modify the XML file itself to change the positions or types of any biome tile or resource.

5.0.7 STRIPS

The planning algorithm used for this project is STRIPS (see Fikes and Nilsson [1971]), which stands for *Stanford's Research Institute Problem Solver*. It is used to compute the actions that should be taken by the AI agent in the game to successfully achieve a set of goals. The goals are determined by a state machine that keeps track of the anthropomorphic properties of the AI: hunger, thirst, need for warmth, energy.

This state machine can introduce limitations into the planner; for example, it can dictate that the agent cannot move heavy objects while it is tired. All of these parameters are expressed in terms of different states that can be customised by the programmer and tailored to the specific application.

The algorithm works by accepting three pieces of information:

1. **Initial State**: A list of facts about the environment. (such as: "AI Agent is at 0,0", "AI's health is 70%", "There is wood at 100,245", etc).

2. **Actions**: Pure functions that transform the state through predicate manipulation. Each action has a name, a list of parameters, a precondition list (facts about the state which have to be true in order for the action to be applicable) and a postcondition list (facts that will be added or removed from the world state upon executing a given action).

3. **Goal State**: The desired state of the world after executing a plan generated by this algorithm.

When it's asked to solve a problem, the algorithm looks for a relevant action. The list of relevant actions is defined as those whose postconditions modify facts that are present in the goal state.

Then, the algorithm checks if the problem is *trivially satisfiable*: if no action needs to be taken for the initial state to satisfy the goal state, it returns an empty plan.

An action to be tested is then selected from the set of possible candidates. Then, a list of *ground states* is generated. A ground state is an abstraction not present in the original STRIPS paper, but is one we have had to come up with in order to implement a mathematical algorithm such as this in a computer program. Ground states are a mapping from symbolic names to reified values. For example, a ground state could be something like "from: A, to: B".

Each ground state is tested: in order to be able to apply a given operator matched with a ground state, we must first find a plan that solves its pre-conditions: STRIPS tracks back from the solution. For example, if we want to open a door we must first find a key. Therefore, in order to open a door we must first devise the plan to "find a key".

Once a valid plan is found to satisfy the pre-conditions of an operator, the algorithm checks if the result of applying this action to the current world state would result in the satisfaction of the overall goal. If this is true, then we simply need to concatenate the plan to satisfy the pre-conditions of any given action to the application of the action itself and the plan is complete. This recursive definition lends itself to an infinite loop if the desired plan does not exist, but we avoid this issue by terminating the algorithm if the list of candidate actions and ground states has been exhausted.

This plan is then passed to the game, where it is interpreted and executed one frame at a time: if the AI agent needs to move, the hot loop of the game will move it closer towards its target one iota at a time.

The code for the planner can be found in Appendix A, Code Samples.

Chapter 6

Discussion and Conclusion

Overall the workflow of this project went smoothly. A large portion of the work has been done in the early stages, resulting in a base for future endeavours. Due to a team member using the Linux Operating System, various steps were taken in order to make future merging of work much easier. As such, after the first attempt, all other programming merges were done without any problems.

Through the use of *Axosoft* the organisation of the team was made significantly easier. This was updated almost daily by the project manager, and each specific card was updated timely and finished accordingly. A burn-down chart(see appendix 1.01) was available, determining the time at which the project would be done according to our work submitted and number of hours left on the remaining tasks. Effective use of GitHub was in turn used to organise the project from a programming point of view. Commits to individual branches were made regularly (see appendix 4.01) with each new project part or feature implemented. This provided every member with the ability to download and work with the latest version of the project from anywhere, as well as synchronise with the other programmers.

SFML was chosen for graphical representations due to its ease of use and efficiency. It is based on the Open Graphics library and was easy to implement within the project and used to drawn the map, resources and the AI. Visual representation was not highly necessary, yet it is useful, and allowed us to develop other various skills such as creating assets and implementing them accordingly.

Various different actions can be completed, such as gathering resources such as wood and being able to use the available resources. The AI will create a plan and then proceed to execute it. The resources are displayed via the interface to the user in the Inventory section, on the bottom left corner of the screen. Due to the nature of the project, other actions can be programmed with ease within the AI, giving it a much wider array of possible plans. Various assets are available within the project yet not used by the AI, such as a fireplace that can be built using wood and used to either cook food or generate warmth in order to satisfy one of the weights, namely the Warmth weight.

The project was successful in implementing a GOAP AI capable of planning and organising actions in order to complete a goal. The AI can be tested on various maps with different conditions in order to get different results based on the resources available to the AI. We believe that this fulfills our project brief and satisfies our aims and objectives. The planner logic can now be taken and implemented into a variety of scenarios with ease. Overall this task was well managed and mostly ran to the deadlines set by the project manager. The *Axosoft* site was well used and allowed the team to stay on top of their tasks. Overall the development process ran fairly smoothly with most issues being resolved quickly, this meant that we were able to focus upon completing our brief and goals. The Team worked very well with each other and pulled together in order to complete the overall project aims. Each team member completed their assigned work, often completing more to help progress the project. On some occasions the personal deadlines of the team were not met, in these instances certain members of the team undertook other members work so that the project would stay on track.

Bibliography

Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189 – 208, 1971. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(71)90010-5. URL `http://www.sciencedirect.com/science/article/pii/0004370271900105`.

Bulitko, V., Lee, G. 200-. Learning in Real-Time Search: A Unifying Framework. [pdf] Available at: ¡ http://arxiv.org/pdf/1110.4076.pdf¿ [Accessed 23 April 2015]

Lee, W. 200-. DBA* A Real-Time Path Finding Algorithm. [pdf] Available at: ¡ https://people.ok.ubc.ca/rlawrenc/research/S [*Accessed*23*April*2015]

$Sturtevant, N., Buro, M.200-.PartialPathfindingUsingMapAbstractionandRefinement.[pdf]Availableat :< https://skatgame.net/mburo/ps/path.pdf >$ [*Accessed*23*April*2015]

$Anguelov, B.2011.VideoGamePathfindingandImprovementstoDiscreteSearchonGrid-basedMaps.[pdf]Availableat :< https://takinginitiative.files.wordpress.com/2011/05/thesis.pdf >$ [*Accessed*23*April*2015]

$Owens, A.2002.SecretsfromtheRobocodemasters : Anti - gravitymovement.[online]Availableat :< http://www.ibm.com/developerworks/java/library/j-antigrav/ >$ [*Accessed*23*April*2015]

$Sutherland, A.2013.VectorFieldObstacleAvoidance.[online]Availableat :< http://buildnewgames.com/vector-field-collision-avoidance/ >$ [*Accessed*23*April*2015]

$Bevilacqua, F.2012.UnderstandingSteeringBehaviors.[online]Availableat :< http://gamedevelopment.tutsplus.com/ser steering-behaviors--gamedev-12732 >$ [*Accessed*23*April*2015]

$Reynolds, C.199-.SteeringBehaviorsForAutonomousCharacters.[online]Availableat :< http://www.gamesitb.com/path$ [*Accessed*23*April*2015]

$Graham, R., McCabe, H., Scheridan, S.200-.PathfindinginComputerGames.[pdf]Availableat :< http://www.gamesitb.c$ [*Accessed*23*April*2015]

Appendix A

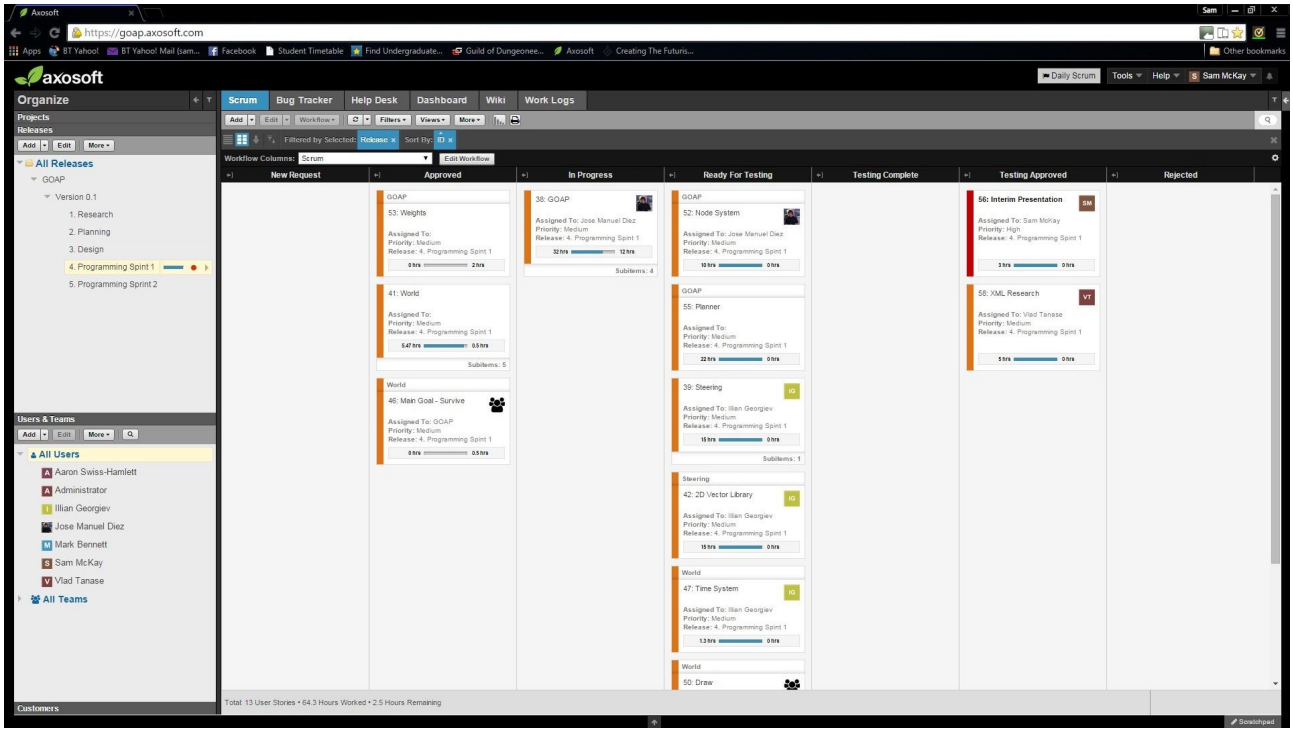
Appendix

A.1

A.1.1

Axosoft Burn-down Chart

A.1.2



Axosoft User Stories

A.2 Code samples

```
#pragma once

#include <algorithm>
#include <iostream>
#include <vector>
#include <map>
#include <assert.h>

#include "goap/printable.hpp"
#include "goap/fact.hpp"
#include "goap/action.hpp"

namespace Planner {
    class Problem: public Printable {
        State state;
        std::map<std::string, Action> actions;
        Goal goal;

        bool satisfies(State state, Goal g, bool weak=true, GroundState* ground=NULL) {
            for(auto it = g.begin(); it != g.end(); it++) {
                // Check if this fact is in the state
                bool found = false;
                for(auto its = state.begin(); its != state.end(); its++) {
                    if(its->name == it->name) {
                        found = true;
                        assert(its->params.size() == it->params.size());

                        // If any of the arguments are different (and not floating), it doesn't satisfy the goal
                        for(int i = 0; i < its->params.size(); i++) {
                            if(weak) {
                                if((its->params[i] != it->params[i]) && it->floating == false && its->floating == false)
                                    return false;
                            } else {
                                if(ground == NULL) return false;

                                // Check with the ground state
                                std::vector<std::string> arguments = matchGrounds(it->params, *ground);
                                if(arguments.size() == 0) // No relevant argument found in current ground, skip
                                    continue;

                                if(its->params[i] != arguments[i]) {
                                    //std::cout << "STRONG MATCH FAILURE: " << it->params[i] << " != " << arguments[i] << std::endl;

                                    return false;
                                }
                            }
                        }
                    }
                }

                if(!found)
                    return false;
            }

            return true;
        }

        std::vector<Action*> choose_operator(Goal g) {
            // TODO: Add weighting/heuristics?
            std::vector<Action*> result;

            for(auto it = this->actions.begin(); it != this->actions.end(); it++) {
                // Check if any of the postconditions in this actions are relevant to some predicate in g
                for(auto itp = it->second.postconditions.begin(); itp != it->second.postconditions.end(); itp++) {
                    for(auto itg = g.begin(); itg != g.end(); itg++) {
                        if(itp->name == itg->name)
                            result.push_back(&it->second);
```

```
                }
69
            }
71      }

73      return result;
    }
75
    std::vector<Fact> update_state(State s, Plan p, GroundState ground) {
77      for(auto it = p.begin(); it != p.end(); it++) {
            // First, find an action with the same name as this fact.
79          auto act_it = this->actions.find((*it).name);

81          // If no action with a matching name is found, ignore this step.
            if(act_it == this->actions.end())
83              continue;

85          // Otherwise, apply it to the state.
            Action act = act_it->second;
87          s = act.engage(s, ground);
        }
89
        return s;
91  }

93  std::vector<GroundState> generateGroundStates(Action* act, State s, Goal g) {
        std::vector<GroundState> result;
95      std::vector<std::string> values;

97      // Extract all values
        for(auto it = s.begin(); it != s.end(); it++) {
99          for(auto itp = it->params.begin(); itp != it->params.end(); itp++) {
                values.push_back(*itp);
101         }
        }
103     for(auto it = g.begin(); it != g.end(); it++) {
            for(auto itp = it->params.begin(); itp != it->params.end(); itp++) {
105             values.push_back(*itp);
            }
107     }

109     std::sort(values.begin(), values.end()); // don't ask

111     do {
            int ctr = 0;
113         GroundState target;

115         // Find all combinations
            for(auto itp = act->params.begin(); itp != act->params.end(); itp++) {
117             //std::cout << *itp << " " << values[ctr] << std::endl;
                target[*itp] = values[ctr++];
119         }

121         result.push_back(target);
        } while(std::next_permutation(values.begin(), values.end()));
123
        return result;
125 }

127 Plan* solve_(State s, Goal g, int depth = 0, GroundState* stepGround = NULL) {
        std::string padding;
129     Plan* p = new Plan;

131     for(int i = 0; i < depth; i++) {
            padding += "  ";
133     }

135     std::cout << padding << "Subproblem state: " << std::endl;
        for(auto it = s.begin(); it != s.end(); it++) {
137         std::cout << padding << " - ";
            it->print();
139         std::cout << std::endl;
        }
141
        std::cout << padding << "Subgoal: " << std::endl;
143     for(auto it = g.begin(); it != g.end(); it++) {
```

```
                std::cout << padding <<" - ";
145             it->print();
                std::cout << std::endl;
147         }

149         if(stepGround != NULL) {
                std::cout << padding << "Ground: " << std::endl;
151             for(auto it = stepGround->begin(); it != stepGround->end(); it++) {
                    std::cout << padding << " - " << it->first << ": " << it->second;
153                 std::cout << std::endl;
                }
155         }

157         // Vector of applicable operators.
            auto candidate_operators = choose_operator(g);
159
            while(true) {
161             bool weakMatch = stepGround == NULL;

163             if(satisfies(s, g, weakMatch, stepGround)) {
                    std::cout << padding << ">> SOLUTION << state: " << std::endl;
165                 for(auto it = s.begin(); it != s.end(); it++) {
                        std::cout << padding << " - ";
167                     it->print();
                        std::cout << std::endl;
169                 }

171                 // Mission accomplished! \o/
                    return p;
173             }

175             if(candidate_operators.size() == 0) {
                    return NULL;
177             }

179             // Get the first candidate operator, and pop it from the list
                auto candidate = candidate_operators[0];
181             candidate_operators.erase(candidate_operators.begin());

183             // Generate possible ground states
                std::vector<GroundState> grounds = this->generateGroundStates(candidate, s, g)
    ;
185             auto ground = grounds.end();
                Plan* subplan = NULL;
187             bool found = false;

189             for(auto it = grounds.begin(); it != grounds.end(); it++) {
                    ground = it;
191                 GroundState* copiedGround = &(*it);

193                 // Get a plan that solves candidate's preconditions with the current
    ground candidate
                    subplan = this->solve_(s, candidate->preconditions, depth + 1,
    copiedGround);
195                 if(subplan == NULL) {
                        //std::cout << "Subplan invalid, continuing to next ground." << std::
    endl;
197                     continue;
                    }
199
                    // Check if this ground state satisifes the subgoal.
201                 // Run all actions in the subplan and update h
                    State h = update_state(s, *subplan, *ground);
203                 // Run the current action candidate and modify the state
                    h = candidate->engage(h, *ground);
205
                    if(satisfies(h, g, weakMatch, copiedGround)) {
207                     found = true;
                        s = h;
209                     break;
                    } else {
211                     std::cout << "Nope, invalid final state." << std::endl;
                    }
213             }

215             if(found == false) // None of the ground states are valid: no solution
```

```
                    return NULL;

                // Add all steps in subplan to the Master Plan
                for(auto it = subplan->begin(); it != subplan->end(); it++) {
                    p->push_back(*it);
                }

                // Add the current action to the plan, matching params to ground.
                std::vector<std::string> arguments = matchGrounds(candidate->params, *ground);
                p->push_back(Step(candidate->name, arguments)); // Housekeeping
            }
        }

    public:
        Problem(std::vector<Action> _actions, State _state, Goal _goal):
            state(_state), goal(_goal) {

            for(auto it = _actions.begin(); it != _actions.end(); it++) {
                this->actions.insert(std::pair<std::string, Action>(it->name, *it)); // Thanks C++
            }
        };

        void print() {
            std::cout << "Available actions: " << std::endl;
            for(auto it = this->actions.begin(); it != this->actions.end(); it++) {
                std::cout << " - ";
                it->second.print();
                std::cout << std::endl;
            }
            std::cout << std::endl;

            std::cout << "Problem state: " << std::endl;
            for(auto it = this->state.begin(); it != this->state.end(); it++) {
                std::cout << " - ";
                it->print();
                std::cout << std::endl;
            }
            std::cout << std::endl;

            std::cout << "Goal: " << std::endl;
            for(auto it = this->goal.begin(); it != this->goal.end(); it++) {
                std::cout << " - ";
                it->print();
                std::cout << std::endl;
            }
            std::cout << std::endl;
        }

        Plan* solve() {
            return this->solve_(this->state, this->goal);
        }

    };
};
```

codesamples/problem.hpp

```
#pragma once

#include <iostream>
#include <vector>
#include <map>
#include <assert.h>

#include "goap/printable.hpp"
#include "goap/fact.hpp"

namespace Planner {
    class Action: public Printable {
    public:
        std::string name;
        std::vector<std::string> params;
        std::vector<Fact> preconditions;
        std::vector<Fact> postconditions;
```

```
19          Action(std::string _name, std::vector<std::string> _params, std::vector<Fact>
    _preconditions, std::vector<Fact> _postconditions):
                name(_name), params(_params), preconditions(_preconditions), postconditions(
    _postconditions) {};
21
            void print() {
23              std::cout << this->name << "(";
                for(auto it = this->params.begin(); it != this->params.end(); it++) {
25                  std::cout << *it << ", ";
                }
27              std::cout << ")";

29              std::cout << ", preconditions: ";
                for(auto it = this->preconditions.begin(); it != this->preconditions.end(); it++)
    {
31                  it->print();
                }
33
                std::cout << ", postconditions: ";
35              for(auto it = this->postconditions.begin(); it != this->postconditions.end(); it
    ++) {
                    it->print();
37              }
            }
39
            State engage(State s, GroundState ground) {
41              // Remove negated postconditions from state, add new facts to the state
                for(auto it = this->postconditions.begin(); it != this->postconditions.end(); it
    ++) {
43                  std::string name = it->name;
                    if(name[0] == '!') { // Remove this from the state
45                      name.erase(0, 1);

47                      auto position = s.end();
                        for(auto stateIt = s.begin(); stateIt != s.end(); stateIt++)
49                          if(stateIt->name == name)
                                position = stateIt;
51
                        if(position != s.end())
53                          s.erase(position);

55                  } else { // Addit to the state
                        if(it->floating == false) { // Trivial case. Not floating? Add it directly
    .
57                          s.push_back(*it);
                        } else { // Otherwise, match it with the ground.
59                          // At this point, we know the name of the fact we want to create,
                            // but we need to match up the arguments with the supplied ground
    state.
61                          std::vector<std::string> arguments = matchGrounds(it->params, ground);

63                          // Got all of this postcondition's params matched up to their ground
    state.
                            // All that's left is to instantiate a matching Fact and add it to the
     state.
65                          s.push_back(Fact(name, arguments));
                        }
67                  }
                }
69
                return s;
71          }
        };
73  };
```

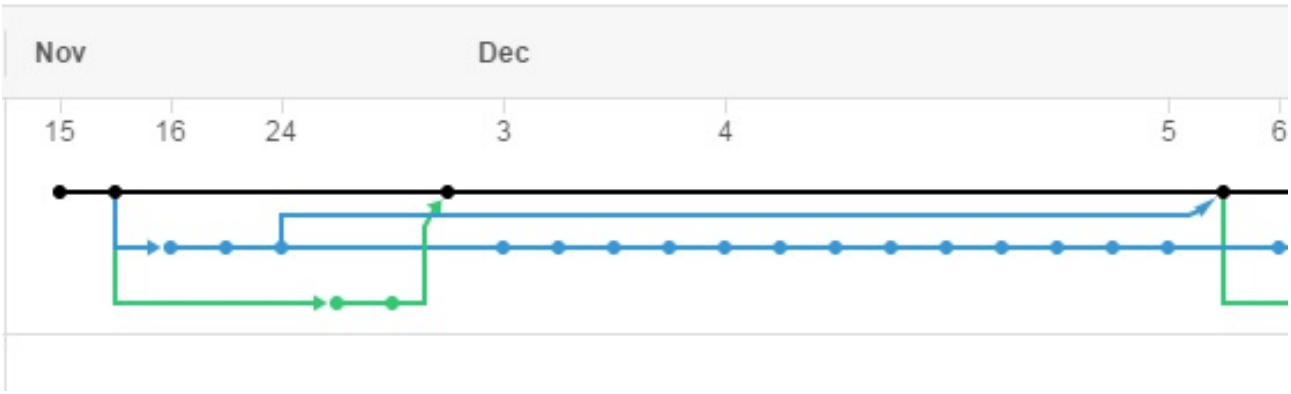
codesamples/action.hpp

```
1   #pragma once

3   #include <iostream>
    #include <vector>
5   #include <map>
    #include <assert.h>
7
    #include "goap/printable.hpp"
9
```

```
namespace Planner {
    class Fact: public Printable {
    public:
        std::string name;
        std::vector<std::string> params;
        bool floating;

        Fact(std::string _name, std::vector<std::string> _params, bool _floating=false):
            name(_name), params(_params), floating(_floating) {};

        void reify() {
            this->floating = false;
        }

        void print() {
            std::cout << this->name << "(";

            for(auto it = this->params.begin(); it != this->params.end(); it++) {
                std::cout << *it << ", ";
            }

            std::cout << ") (floating: " << (this->floating == 1 ? "true" : "false") << ")";
        }
    };

    typedef Fact Step;
    typedef std::vector<Fact> Goal;
    typedef std::vector<Step> Plan;
    typedef std::vector<Fact> State;
    typedef std::map<std::string, std::string> GroundState;

    // Match a list of parameters to their ground state.
    std::vector<std::string> matchGrounds(std::vector<std::string> params, GroundState ground)
     {
        std::vector<std::string> arguments;

        for(auto param = params.begin(); param != params.end(); param++) {
            // Find this parameter in the ground state.
            auto value = ground.find(*param);
            if(value != ground.end()) {
                arguments.push_back(value->second);
            }
        }

        return arguments;
    }

}
```

codesamples/fact.hpp

A.2.1



GitHub Branches

A.3 Literature Review

2015

Literature Review

GOAP

SAM MCKAY, ILLIAN GEORGIEV, AARON SWISS-HAMLET, JOSE MANUEL DIEZ AND VLAD-EUGEN TANASE

Contents

Strips

In order to fully implement a goal orientated action planning system we have researched heavily into the STRIPS methodology, this is one of the original methods of doing this and with slight updates can be utilised within our project.

Strips uses a regressive search algorithm to create an action plan, this works by have a desired state or goal and working backwards to fulfil the pre-requisites of the goal. For example, if the goal is to not be cold then a pre-requisite could be too be near a fire, in order to achieve that it would need wood, to get wood it has cut down a tree and to do that it needs to find an axe. So in this case its plan would be:

Find axe
Cut down tree
Make fire
Sit near fire

These goals can be in the form of a needs system i.e. hunger, energy, warmth, thirst and so on. These would each have a weight which would indicate their importance and the AI would prioritise those with a greater weight. This means that the search algorithm would need to recalculate when a step in its plan has changed in case the goal has changed, this however could make it more efficient in case it has discovered a quicker way of achieving the goal since it calculated the last plan.

Steering

Steering is useful for simplistic applications while still giving the impression of complex actions. The main implementation method involves vectors(math context). Steering model can be applied to collision avoidance, as well as following and tracking other entities and paths.

Steering has many different implementations. From flocking algorithms such as Boids to path tracking, collision avoidance, containment etc. Steering implements vectors to determine the direction in which an AI agent should move. These vectors are subject to change based on the implementation of steering and the current world state.
A good implementation of steering is Anti Gravity movement and Vector Field movement. Both methods work in roughly the same manner. The AI agent has a direction/steer vector which it is trying to keep to.

Vector field/Flow field uses objects around the world and the goal position to create a grid of direction vectors which alter the original direction vector of the AI to have it avoid obstacles. This is useful for avoiding big and difficult to go through terrain.
Vector field method is somewhat limited by the resolution of the grid. An ideal solution is to have a good midpoint. Another possible issue with this method is moving objects.
While static terrain requires only one calculation in the case the grid resolution is high a big moving object that covers multiple grid squares will have to affect all the cells around its perimeter and even a bit beyond.

Similar to "Obstacle avoidance" described by Craig Reynolds, Anti Gravity works on roughly the same principle as Vector Fields however instead of a grid of vectors the AI agent tracks the distance between each object it can collide with and applies a steering vector opposite the direction of that object.
This approach is also similar to what is used in the Boids AI model developed by Craig Reynolds. Except in this case all other "entities" are static and only the Separation and Alignment rules are taken into account.
Path following allows for the use of predetermined paths in the game world. The agent sill uses vectors to steer and implements the use of a future position. The future position usually will have an offset from the path this offset is used to correct the AI direction vector. While not 100% accurate to the path the AI will stay within a certain distance of it.
Steering can also be implemented in the context of moving entities. Following, intercepting, queuing and running away from. These behaviors work on the same principles as the ones listed above with the change that the entities/objects used as a reference for the calculations are not static.

Combining different steering implementations can give the AI agent multiple ways to navigate around the world based on information supplied to it by its sensors.

Pathfinding

Unlike steering method pathfinding can be used to solve complex problems such leave a maze and find the shortest distance between the start and end points (in case there are obsticles).
Pathfinding has several limitations and drawbacks:
It requires a comprehensive grid/graph with well defined connections between cells/nodes
Movement between those is not necessarily described by the algorithm used and usually has to be implemented separately.
Algorithms used for pathfinding are very resource intensive and can take a very long time to compute.
If a path is obstructed by a dynamic object the AI will not register that and possibly get stuck.
The two main approaches are directed and undirected pathfinding.
Undirected pathfinding is not widely implemented and it uses one of two main algorithms, Breadth-first and Depth-first. They both search though the connected nodes until they find a solution with no regard as to the difficulty, time it might take to get to the final goal and while similar the two algorithms work in almost the opposite way.
Breadth-first expands all nodes, at one level, revealing all the nodes they are connected to. Only when all nodes from the current level are expanded will it move down to the next.
This is useful where the node hierarchy has multiple solutions are different depth levels. Breadth-first will find the solution that takes the least steps(node expansions).
As the name with both algorithms suggests Depth-first does not focus on expanding all nodes originating from a single node before moving on. It looks at the children nodes until it reaches a node which has no new ones originating from it. This method is much more efficient than Breadth-first in situations where you have a very uniformly shaped tree of nodes with a lot of solutions.
The directed pathfinding is more common in games since it provides the shortest/fastest route to a target location. However it is much more time consuming to compute than the undirected methods.
Directed algorithms work on a cost system described by the developer. Costs or weights can be anything from the physical distance between nodes in the game, to estimated time of completion or even arbitrary values put in by the programmers/designers to “guide” the algorithm in a certain direction.
Like with undirected pathfinding there are two main strategies to finding the shortest: uniform search and heuristic search.
Uniform search gives an optimal result picking the easiest route possible every time however it is very inefficient and may take a very long time to complete.
As the name suggests heuristic search uses a heuristic to determine the route. This is the much more efficient method however but it does not search through all possible paths and not always is the final solution the most optimal.
The two most used algorithms for directed pathfinding are Dijkstra(uniform search) and A*(combines uniform and heuristic search). Other algorithms exists which are mostly variations of the ones already mentioned.
These variations or evolutions are developed in order to solve some of the issues that are present whenever pathfinding is involved. A* itself is an evolution of Dijkstra.
PRA* is an evolution of the A* algorithm. It speeds up the A* algorithm by first generating a solution for a more abstract version of the search space. This way the A* algorithm has to look through a fraction of the original node/grid mesh.

D*(dynamic A*)this algorithm takes into account changes in the cost of a path after it has been calculated and the AI has been set in motion. This solves the problem with dynamic objects but adds a lot more strain on the CPU in order to accomplish this.

Variations of the A* algorithm go on with each variation improving the original in some way. Here are some of them and how they are influencing the original they are built upon.

Dijkstra – finds all possible paths through a graph/grid.

A* - Uses a heuristic to find the shortest/most optimal path.

IDA* - Iterative Deepening A*

This version tries to speed up A* starting with an approximate answer. With each iteration it expands more and more nodes and tries to match the overall weight/cost to the initial estimate. Somewhat based on the Depth Search algorithm for looking through trees.

D*/LPA* - Dynamic A*/ Lifelong Planning A*

These versions solve the problem of dynamic environments in exchange for requiring a lot of space. These versions keep all details of the last computed path and match it to the current graph/grid state to determine if a recomputed is needed.

PRA* - Partial Refinement A*

Creates a “lower resolution” version of the original graph/grid thus speeding up the algorithm.

LRTA* - Learning in Real-Time A*

substantially lowers the first move delay present in standard A*. It accomplishes this by first doing a very small scale calculation to determine the first step and then it finds paths to the end goal which may be suboptimal. Finding an optimal path is much more expensive compared to A*

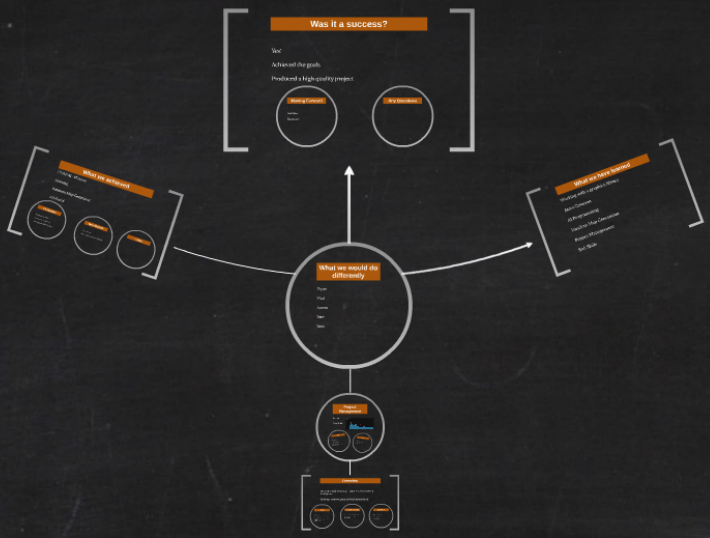
Pathfinding gives the best solutions however it can be very resource intensive based on the implementation. Current algorithms and their variants try and balance speed, resource requirement and quality of result. A specific implementation can be chosen based on the needs of the AI agent.

Map Generation

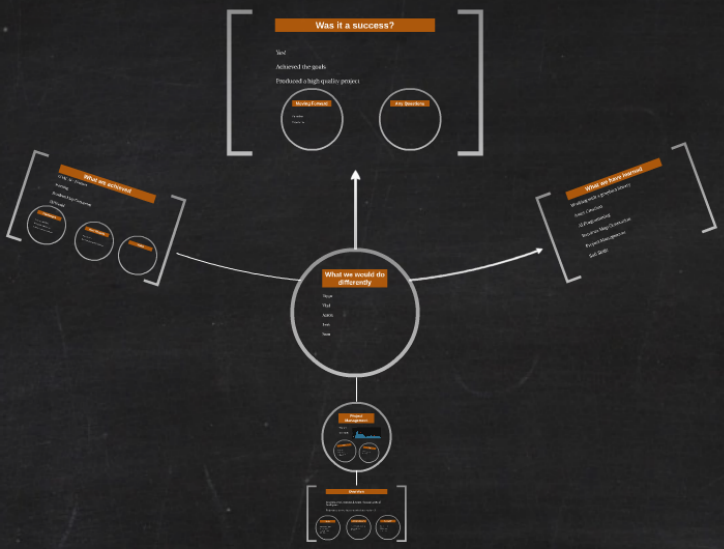
Research conducted into random map generation for 2D games.
Easiest approach is to create tiles, however other options are Height map, Perlin Noise but those are more complicated than what we require for the project and as such we shall be sticking with the tiled approach. This will allow to create plenty of different maps in which to test our AI, we can also utilise the random generator for the creation of objects throughout the scene allowing us to spend more time upon the action planning section.

The maps will need to be exported into an XML format that can be read with a TMX reader that can be used with SFML, the advantage of having them in XML format is that we can also edit them manually should we need to.

A.4 Final Presentation Slides



Goal Orientated Action Planning



Goal Orientated Action Planning

Overview

To create a Goal Orientated Action Planning Artificial Intelligence

To design a survival game in which to test the AI

Goals

Be able to generate a plan

Carry out the plan

Visualize this in the 2D environment

Overall Outcome

Created an AI capable of planning

Demonstrated it in a 2D Environment

Teamwork

Success as a team

Skill development

Communication

Goals

Be able to generate a plan

Carry out the plan

Visualize this in the 2D environment

Overall Outcome

Created an AI capable of planning

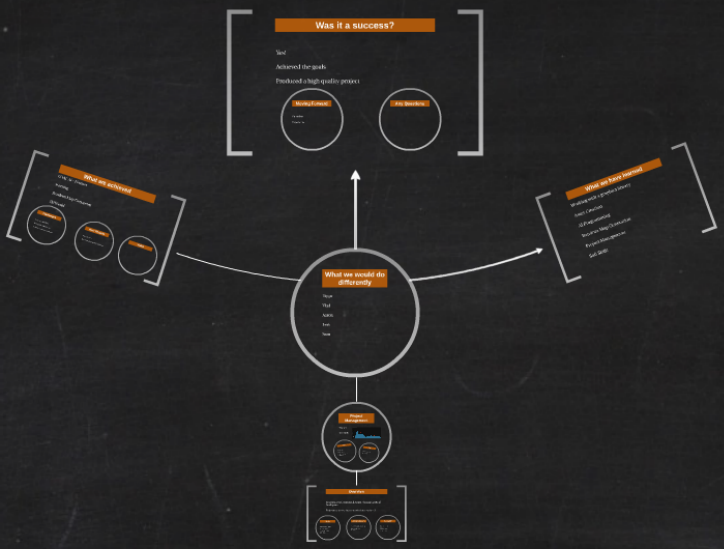
Demonstrated it in a 2D Environment

Teamwork

Success as a team

Skill development

Communication



Goal Orientated Action Planning

What we have learned

Working with a graphics library

Asset Creation

AI Programming

Random Map Generation

Project Management

Soft Skills

What we achieved

GOAP AI - Planner

Steering

Random Map Generator

2D World

Challenges

Keeping to deadlines

Pacing and staying power

Working in a multidisciplinary

New Wizards

Great Success

Interest from potential employers

Video

Challenges

Keeping to deadlines

Pacing and staying power

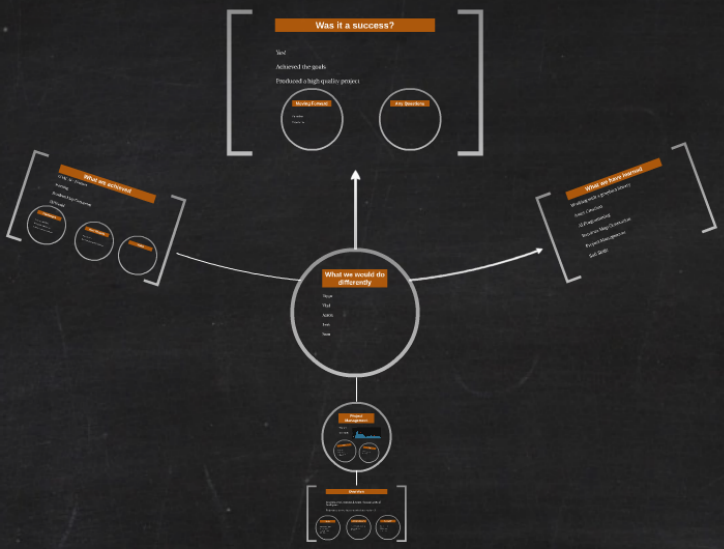
Working in a multidisciplinary

New Wizards

Great Success

Interest from potential employers

Video



Goal Orientated Action Planning

Project Management

Axosoft

New Skills

Pro

Track progress

Task Management

Real world management methodology

Con

Not always up to date

Learning curve

Pro

Track progress

Task Management

Real world management methodology

Con

Not always up to date

Learning curve

What we would do differently

Iliyan

Vlad

Aaron

Josè

Sam

Was it a success?

Yes!

Achieved the goals

Produced a high quality project

Moving Forward

Portfolios

Experience

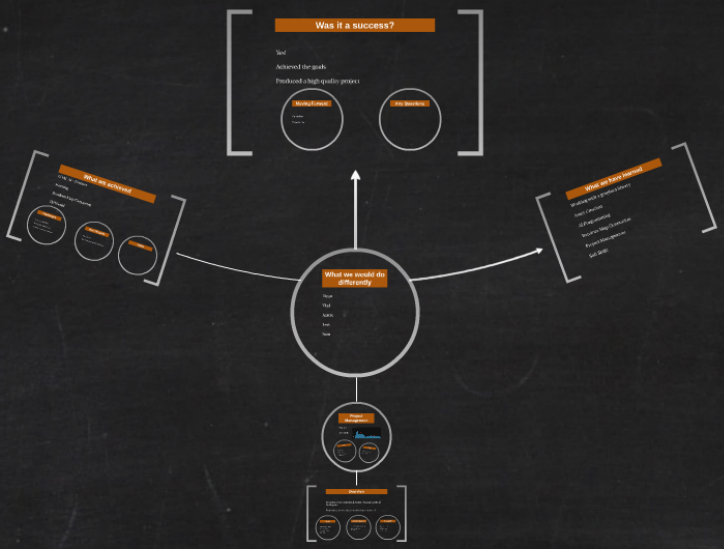
Any Questions

Moving Forward

Portfolios

Experience

Any Questions



Goal Orientated Action Planning