

DO MINION!

A Reflective Practice



illustration by Jeremia

Dominion

A Reflective Practice
03/03/2016

Author: **Hien Quy Tran**
Student Number: 543800

Semester: 5th Semester

Lecturers:
Prof. Susanne Brandhorst
Prof. Thomas Bremer
Oliver Langkowski
Sven Thomas Gorholt

Lecture: B 25 Praxisphase 1
Project: Development of a game with focus on agent based game mechanics

Content

1. Introduction to our Project	1
1.1 The Development Team	2
1.2 Concept	3
1.3 Initial Idea	4
1.4 Art Style	5
1.5 Background Story	6
2. Personal Goals and Milestones	8
3. My Task Area	9
3.1 Developing Gameplay Concepts	10
3.2 Prototyping	11
3.3 Making Changes to the Concept	12
4. Setting up the Foundation	13
4.1.1 <i>Main Base</i>	13
4.1.2 <i>Game Entities</i>	14
4.1.3 <i>Level Design</i>	16
4.2 Scaling Down	18
5. Understanding Architecture	19
5.1.1 <i>Dual Cursor System</i>	20
5.1.2 <i>AI Performer</i>	22
5.1.3 <i>Animation</i>	23
5.1.4 <i>Artificial Intelligence</i>	28
5.2 Giving Power to the Player	31
6. Graphical User Interface	32
6.1 Icons	33
6.2 Reaction Screens	34
7. Recap	38

1. Introduction to our Project

This project marks the last and greatest project of our game design studies.

Our task is to create a game within a three month period. The minimum time budget per student is 600 hours. Making this project being the biggest project with estimated working hours of 1800 in total.

Having this in mind; we were able to stretch our scope a little bit further than in previous projects. We were eager to finish the project with a overall polished game, which we would consider as “shippable”.

Other than the previous projects, we had 5 topics to choose from: *Agents, Extend 4, No Screen Game, Racing and Walden*

The topic we chose for this group project is: “**Agents**”.

1.1 The Development Team (Quyrelina)



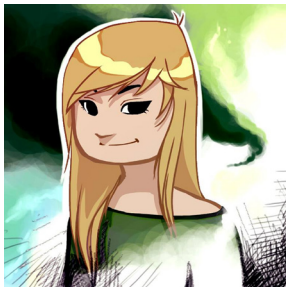
Hien Quy Tran

Game Design
Level Design
Programming
Character Animation Implementation
Graphical User Interface Implementation



Jeremia Oelschläger

Visual Development
Character Design
Environment Design
Graphical User Interface Design



“Lina” Yvonne Röser-Buchkremer

Visual Development
2D Animation
3D Animation

1.2 Concept

The focus of this project was to design a game with main emphasis on “agent based game mechanic”.

An agent represents an entity, which is able to operate by itself without constant player input.

It can be represented as an actual visible character as in games, like for example “Lemmings”. It can also be visual less apparent while managing the whole economical system of a game, like for example in games as “Sim City”.

While agents could be game entities which are operating independently, they also could be game entities which are operating in dependence to the player or towards each other. This would be the case with for example the flocking behavior of ants.

One of the main advantages of agent based systems, is that the player can maneuver a high amount of units without the need of steering each unit individually, like for example in Real Time Strategy games as “StarCraft”.

We decided on a system in which the player has to handle a small amount of units, but in which each of the units would have its own behavior and a rather complex artificial intelligence system, resulting in each unit to have individual personality.

(background image:
StarCraft II by Blizzard Entertainment in 2010)



“Lemmings“ by DMA Design (Rockstar North) in 1991



“SimCity“ by game designer Will Wright in 1989



ants using flocking behavior to bridge a gap in real life

1.3 Initial Idea

Our game was aimed to be a top down isometric 4-player, battle arena in which each player is in charge of a little squad.

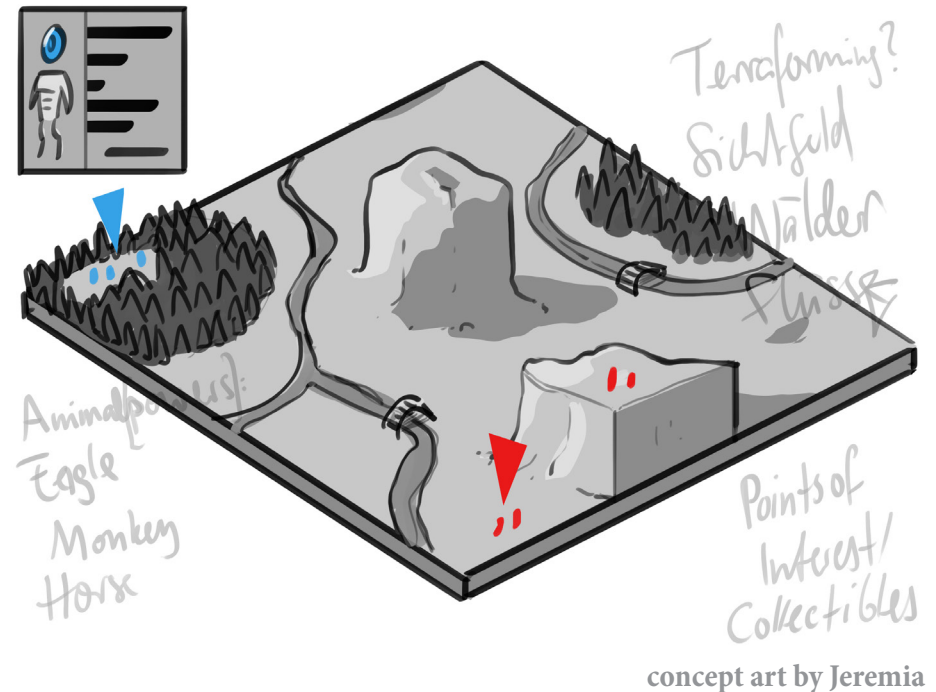
Each squad member has its own statistics, which would result in strengths and weaknesses, this again would ultimately influence their decision making and therefore their behavior.

The player will not be able to steer the characters directly, instead the player will be able to set the statistics, but also the class and assign tasks to each squad member which will also affect their behavior.

There will be two classes each squad member has to be assigned to: worker or fighter.

A worker will be given tasks like scavenging for resources or building an expansion, while a fighter will be given tasks like searching for enemy units in order to attack them or defend a specific area from enemies.

The player will be able to change the assigned tasks of a squad member whenever needed, but the squad member will be forced to return to the base or expansion whenever a change of class is required. Meaning, that changing tasks within a class is free of expenses, but changing tasks which requires to change the units class may be uneconomical and will cost time in which the unit will be especially vulnerable.



Winning Condition

There will be several ways to win a game. A player can win by being the first one to collect 3 out of 5 targets, a player can outplay the other players by being the first one to reach the target amount of resources or a player can be the last surviving race by destroying enemy bases while maintaining the own base intact.

ZOMBIE

AGENTS

CLONES

1.4 Art Style

One of our preferences was to develop a game which can be played against other players locally on a shared screen.

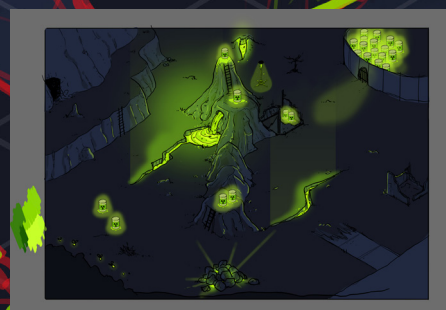
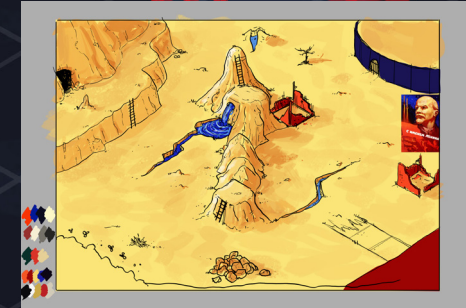
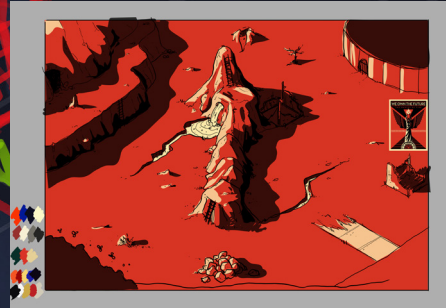
The advantages of a shared screen are: good overview over the whole map, the full usage of the entire screen for every player and the convenient and easy set up for gaming sessions and presentations.

Setting our game up as a local multiplayer was important to us, because we wanted to add a diplomacy system to the game mechanics. We hope this diplomacy system would leverage the players social interaction on the meta game layer.

We were eager to not only develop a fun, but also good looking game, with great assets and animations.

One of the main challenges was to find an art style, which not only looks good on 2K HD resolution, but would also provide a good overview over the whole action for all 4 players on one single shared screen.

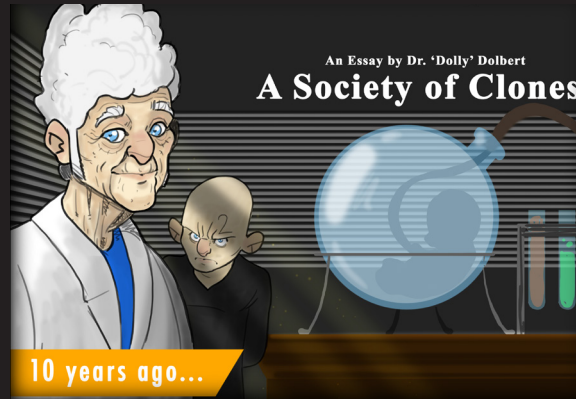
(concept arts by Jeremia)

INTERFACE
EINKLAPPBAR

ROBOTS

1.5 Background Story

The background story of the game is about four scientist, who went mad because of their dedication to science.



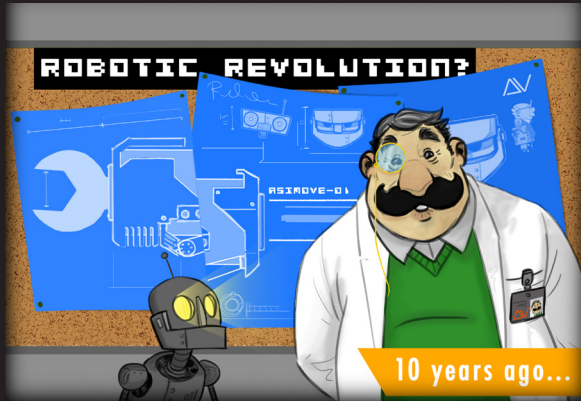
Dr. 'Dolly' Dolbert, who created unsatisfied clones of himself.



Dr. Vicki Frankenstein, who tried to keep people alive for too long.

These mad science, are now creating their little army of minions to conquer the world.





Dr. Robert Asimov, who did not see any harm in researching robotics.



Lady Bugoni, who studied bugs and experimented with cross species.



2. Personal Goals and Milestones

The last project “inVert”, was developed with Unity 5, partly because I wanted to refresh my text based scripting capabilities. The language we were using was C sharp. I was very satisfied with the result and was able to prove myself, that I am able to use the Unity engine with its text based scripting still efficiently after having used visual scripting for a while.

The visual scripting has helped me a lot to understand things, which also applies to text based scripting and my plan is to switch back to text based scripting eventually. (Preferably C++) But for now, I think, that visual scripting helps me to build up general scripting knowledge, which will be applicable for both, either visual or text based scripting.

While programming the behavior of the enemies for the third last project “cubed”, I got very excited about artificial intelligence in games. The project was developed with Unreal Engine 4 and I had the chance to get a little bit more familiar with its components like the behavior tree and the environment query system. But the time schedule was too short to explore its full potential.

That is why, I chose to switch back to the Unreal Engine 4 for this last project and dive into the AI components. My goal is to understand the general principles of Artificial Intelligence in games and eventually get good in programming them.



screenshot of a previous project: “cubed”



3. My Task Area

After our team have agreed on the topic “Agents”, we decided that it would be most efficient, if everybody is developing multiple gameplay concepts by her/himself. We would then have a pool of concepts and ideas, which we could share, combine or redefine.

After having defined a gameplay concept, my task as the only programmer was to start prototyping.

In order to be able to test the AI, I realized, that I would need a consistent designed level, which I started to develop and to block out in the engine with brushes.

Then, I had to start programming all necessary items like for example the agents and a cursor as an interface to select and interact with the agents. Also It was necessary to program an item, which represents the base and enables the player to spawn agents.

Also, I had to create several items, which the AI could interact with.

Having this all setup, the next big task was to implement their behavior.

Things like GUI implementation and animation implementation had still to be done yet.

image: me working on the prototype in our studio

3.1 Developing Gameplay Concepts

Right at the beginning, we decided, that each one of us should develop gameplay concepts for the topic: “Agents” by her/himself.

My idea was to develop a top down isometric 4-player, battle arena game in which each player is in charge of a squad.

Each squad member would have individual configurable statistics like e.g.: speed, strength and intelligence. Different priorities in the statistics would result in different strengths and weaknesses, which would again result in different individual personalities and behavior. A general fast unit, would maybe have less stamina or strength and vice versa.

The player would not be able to steer the characters directly, but instead set the class and assign the tasks for each unit, which ultimately also will affect its behavior.

While the player can assign the statistics, class and tasks of an unit, the unit will maintain independence to a certain degree and decides on how to execute a certain task or when to flee from dangerous situations.

There would be two classes the player can choose from: worker and fighter, with each having choice of two assignable tasks.

The worker would be given tasks like scavenging for resources or building expansions, while a fighter would be given tasks like searching for enemy units in order to attack them or defend a specific area in order to keep enemies away. This would allow the player to perform strategic actions and to protect strategic locations on the map.

The player would be able to change the assigned tasks of an unit whenever needed, but the unit will be forced to return to the base or expansion whenever a change of class is required. Meaning, that changing tasks within a class is free of expenses, but changing tasks which requires to change the units class may be uneconomical and will cost time in which the unit will be especially vulnerable.

There would be several winning conditions: A player could for example win by being the first one to collect 3 out of 5 targets, a player could outplay the other players by being the first one

to reach a target amount of resources or a player could be the last surviving race by destroying enemy bases while maintaining her/his own base intact.

Another important feature would be the diplomacy system. This way players could truce or betrayal each other, which would leverage the social interaction within the players and create some more emotional gameplay.

Our main challenge is to get the right balance between luck and strategy. I want the player to watch her/his agents and cheer about lucky events, while still thinking to have enough control over the game progress to feel responsible for the outcome.

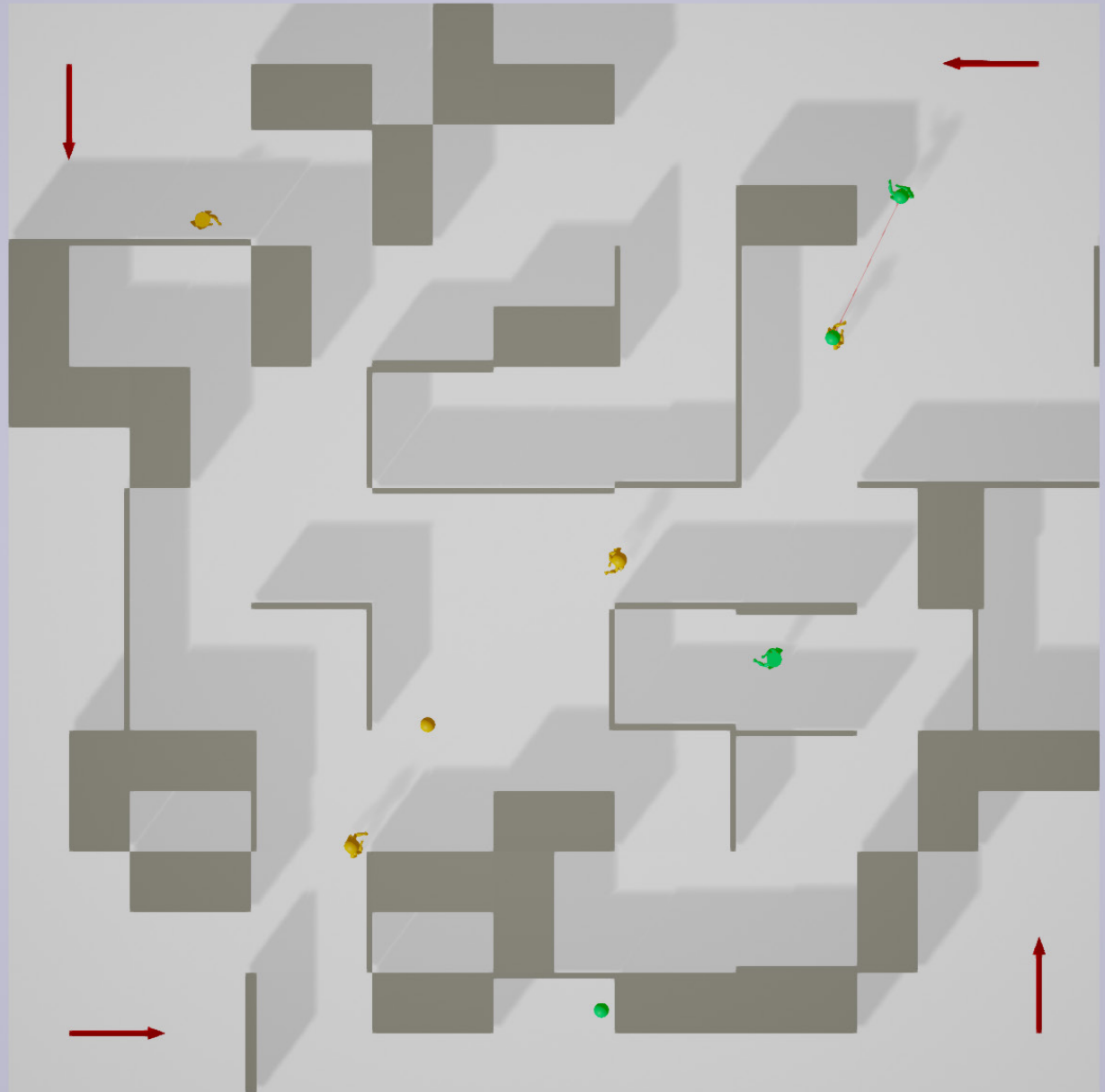
I think with the right balance between luck and skill, the game could stay exciting between experienced and unexperienced gamers. Also it could turn out to be a good causal game, in which players do not need to be extremely skilled or concentrated to exhaustion in order to stay competitive and have fun.

3.2 Prototyping

Having a game concept which strongly relies on Artificial Intelligence, was very new to all of us and it seemed to be very important to have a prototype as fast as possible. We needed to figure out, whether and what makes a game fun, in which the player's dominant activity is mainly watching, not interacting. Also, I was very unfamiliar with AI systems and needed to understand its capabilities as soon as possible.

I started with a level generator, which would provide me levels with random rooms and corridors. I was hoping to be early confronted with several level structures for prototyping. Soon enough I realized, that it was too difficult to optimize behavior in an ever changing environment. This made me abandon the level generator and start blocking out a new persistent level.

It took me several attempts; I needed to discard a lot of prototypes and had to rewrite a lot of functions, in order to achieve a state, where I finally had a basic structure, which would provide all different requirements to start prototyping with the AI.



the first prototype with the level generator

3.3 Making Changes to the Concept

Our initial gameplay concept strongly relied on the player to watch her/his units and make few little changes in their statistics. The player would not give direct orders or steer her/his units anyhow directly.

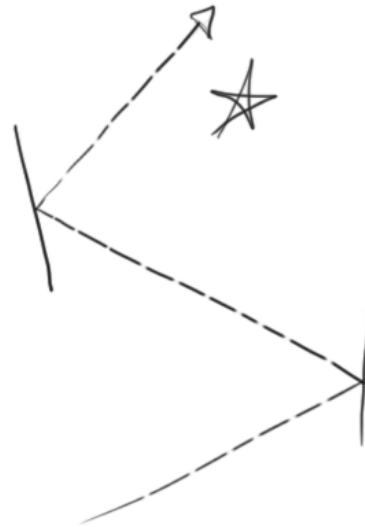
The difficulty with observing as the core gameplay element is, that unintended behavior will be noticed straight away.

My conclusion is:

If a game is simulating simple behavior, the player does not get unsatisfied as long the behavior is consistently behaving in the same pattern. This way the player can comprehend the logic and the conditions, which led to this behavior in certain situations.

On the other hand If a game is trying to simulate smart and complex behavior, the player might not be able to follow how the decisions for the behavior are made. In order to not unsatisfy the player, the behavior should always be considered as reasonable in any situation.

comprehensible:



And this is where our game did not deliver. Because the behavior was tightly tied into the statistics of the unit, it became very difficult for the player to comprehend the decision making part of the units behavior.

Because the player cannot predict the unit's behavior, the player is putting up expectations, which the units behavior has to fulfill. As long the behavior is keeping up with those expectations, the player would take it as guaranteed, but anytime it does not, the player would get unsatisfied.

not comprehensible:



To always guarantee a reasonable behavior, the AI system has to be very complex. Very complex! I was not aware of.

In order to avoid situations, in which the player is unsatisfied about the units behavior, we implemented the feature to assign tasks to units specifically. This way, the player can overwrite some of the units decision making part.

4. Setting up the Foundation

I wanted to make sure, that all required assets are prepared, before I would tackle the big task of setting up the AI. I wanted to finish up all these assets, so that I can later completely focus on AI programming without needing to setup other assets.

Additional to that, I felt more comfortable to deal with the more familiar task first. I have changed to Unity 5 Engine for the last project, and therefor have not been using Unreal Engine for about 4 months.

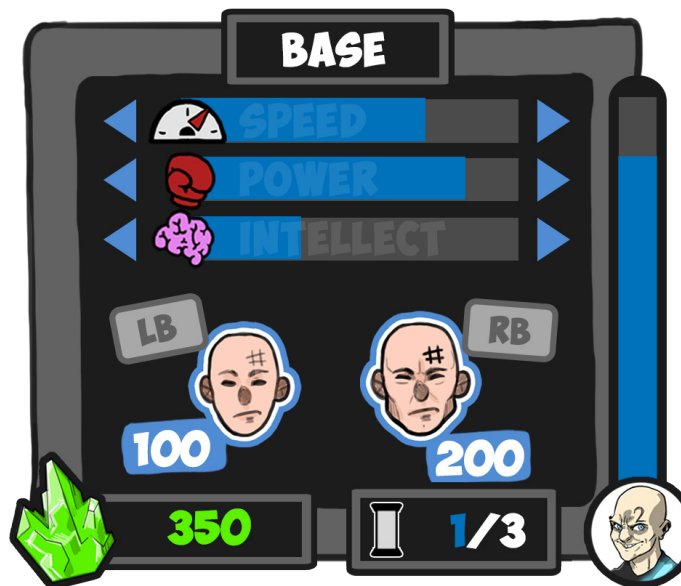
4.1.1 Main Base

Before any unit is able to perform any action in the game, the unit itself needs to be created and placed into the world. That is why I needed some sort of character creator and placement tool for each player.

Setup Statistics

As mentioned before, every unit has three different character traits: *speed*, *strength* and *intelligence*. The effect of each trait will be explained at a later stage of this document. There are three sliders (one for each trait) in the base menu. The player can adjust those settings by using the right stick. The higher an unit is going to be skilled, the more resources it will cost to be created.

Main Base GUI Panel (graphics by Jeremia):



Creating Units

The player is then able to create a worker with the press on the left shoulder button or a fighter with a press on the right shoulder button.

This character creation tool is representing the main base. It also has 800 health points and can be attacked by enemies. If the health drops down to 0, the owning player is out of the game and all her/his units die. This makes it crucial to protect the main base.

4.1.2 Game Entities

While almost everything are game entities, this chapter is all about the little ones. Those, which usually only contain some variables and a little amount of functions. Those functions are being called by the unit which interacts with them, they do not fire them by themselves. In order to interact with those Game Entities, I have created a Interface Blueprint which allows me to specify these objects as “interactables”. The units can then call an unified event, named “interact”, when interacting with them.



Collectible

As mentioned before, the winning condition will be to collect 3 out of 5 collectibles, which are scattered around the map.

As with all objects in this chapter; when an unit interacts with this object, he will call the “interact” function inside the object. This function will set the parameters of the interacting unit to make him carry a collectible and then destroy itself.



In order to drop the collectible, the unit is simply setting his own parameters back to normal and spawns a new collectible in front of him.

Each newly spawned collectible will call out a function on the main bases, which counts the amount of collectibles sitting in front of the main base. If the number reaches 3, the game will end.

Resources

Resources are having the same functionality as collectibles, with only one difference, that upon delivering to the main base, the unit does not spawn a new one in front of the main base, but rather increases the amount of resources in the bank.

That is why resources, other than collectibles, cannot be stolen upon successful delivery, but will be cashed in immediately.

Mining Location

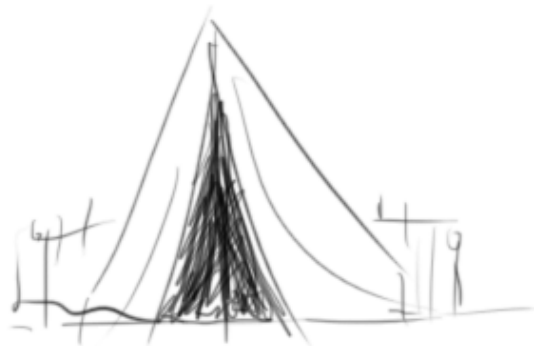
Resources are the only currency in the game and need to be highly disposable. Because we do not want them lying around, overlapping each other and covering half of the map, we decided to create locations, on which the units can produce those resources by themselves. Just as in almost every real time strategy game, the units can see mining location, go to those mining location and mine them.

The mining location holds a variable of the information on how many resources are left. On event “interact”, a function is called out, which subtracts the amount of resources the unit is taking and applies it to the parameters of the unit. It also changes the appearance of the mining location in order to display how depleted the mining location is. If there are no resources left to be taken, the mining location will destroy itself.

Expansion

Expansions can be built by workers anywhere on the map. They represent little stations, in which units can cash in resources.

This makes them very strategical assets, which the player can use to gain advantages on territories, which are further away from the main base.



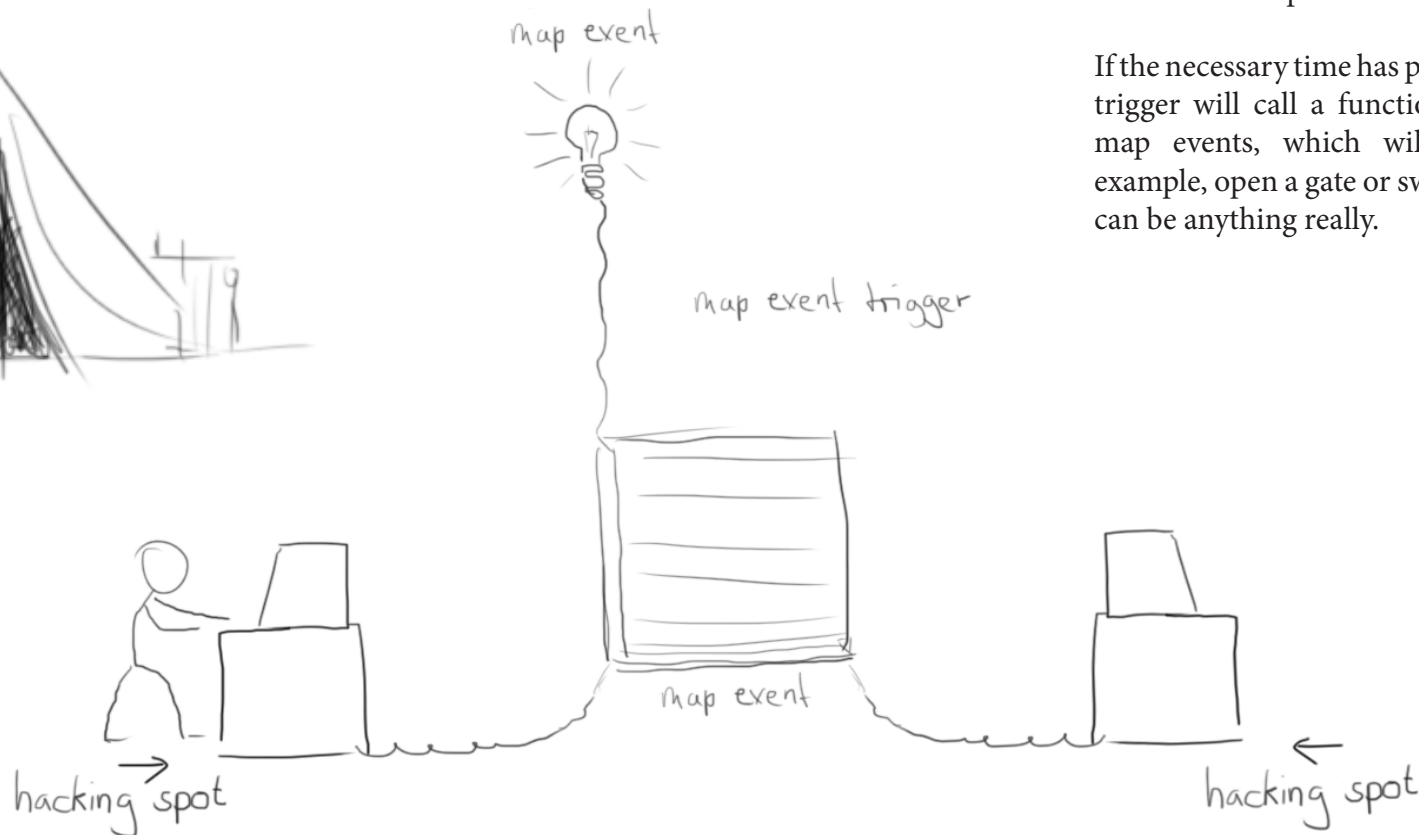
Map Event Triggers

Map event triggers are totally functional. They are not visible to the player, but can be sensed by the cursor. Their only task is to activate the functions of the other referenced game objects. This way, they are multi-functional and can be placed next to several different map events.

The information they need to be fed with are: the referenced map events(, like for example a gate and a light) and invisible hacking spots.

Upon interaction with the map event trigger, the unit will walk to the nearest hacking spot and align its orientation with the orientation of the hacking spot, which potentially faces a representative mesh(, like for example a terminal). The hacking spots contain a variable about the default operation time to activate the map events.

If the necessary time has passed, the map event trigger will call a function in all referenced map events, which will make them, for example, open a gate or switching on a light. It can be anything really.



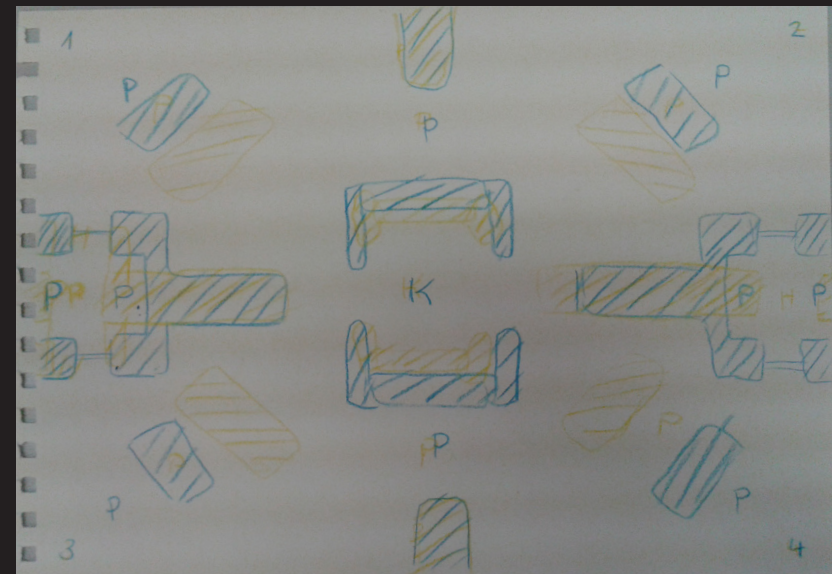
4.1.3 Level Design

Because I figured, that it is rather difficult to evaluate the behavior of the units on a random generated map, I decided, that I would need a consistent environment in order to be able to draw conclusions about their decision making.

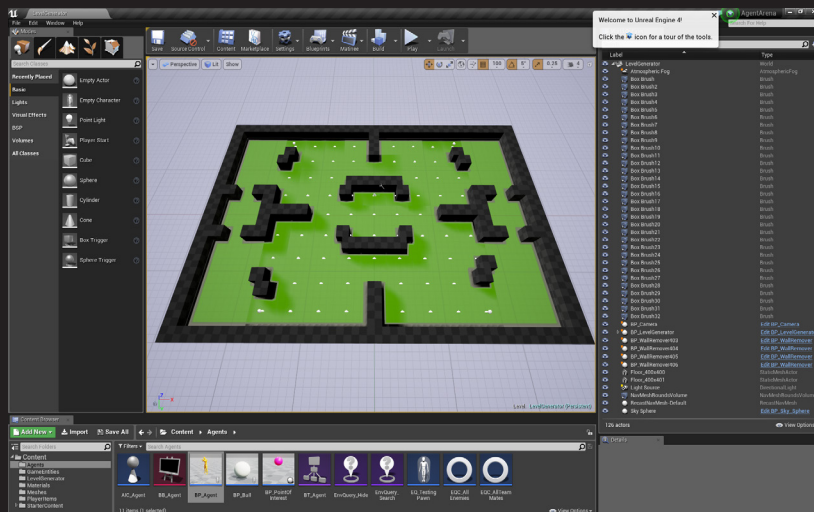
That is why I started to sketch out a level on paper and used this sketch to block out the level in the game engine.

This blocked out level is giving me the opportunity to test the behavior on a consistent environment and helps Jeremia to continue develop the visual appearance of the level, by replacing my blocked out areas with his Assets.

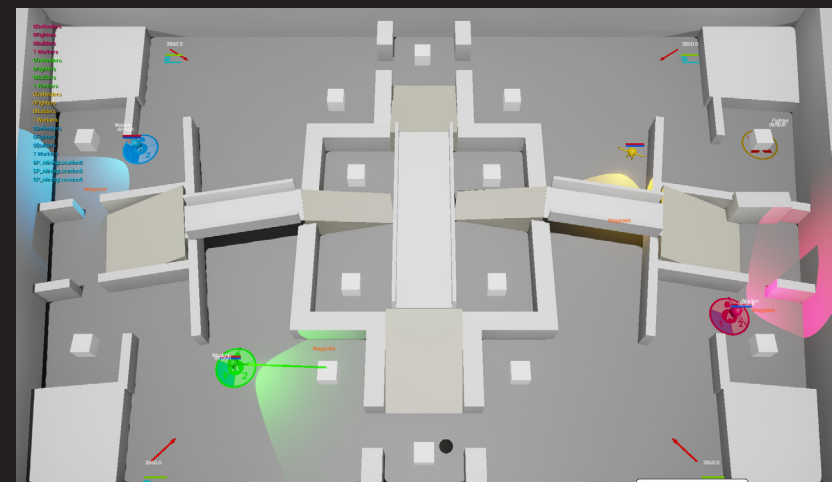
The plan was to keep a consistent level design and to use just a little bit of randomness for the target placements etc., in order to raise the replayability.



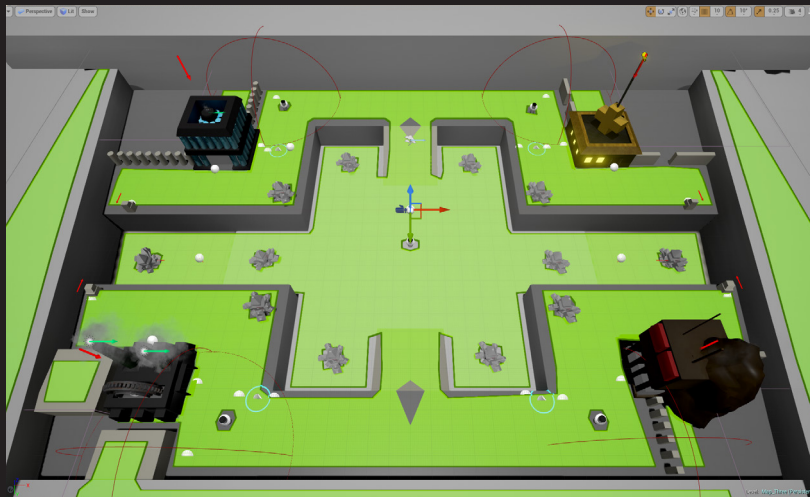
an early sketch of the Level Design on paper



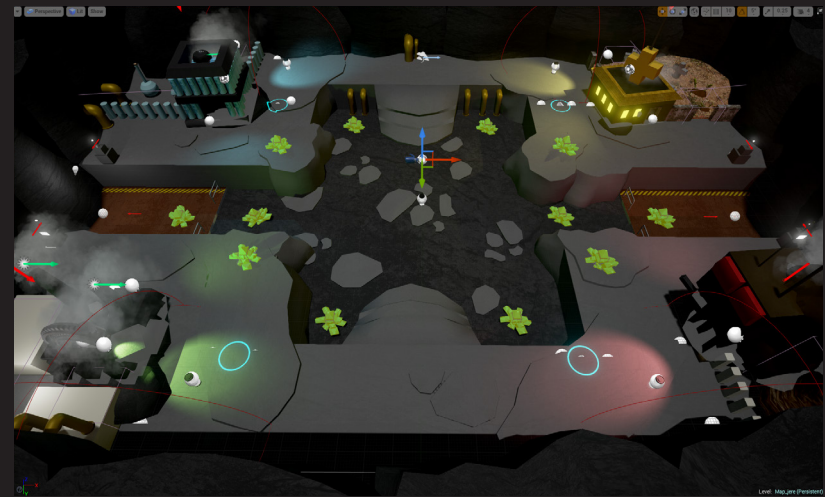
the blocked out level in the engine



a redefined version of the first level



the blocked out new level layout



the final layout of the map



the new level layout after Jeremia has been adding his assets

The first level layout helped me a lot to draw valuable conclusions through testing and redefining.

Because we were aiming for a shared screen game, every bit of unused space had to be avoided. Even the placement of walls would result in unused space, depending on how thick they are. Using planes to divide the space would not look right.

One of the major solutions for the final level layout was to use differences in heights in order to separate areas from each other. This way it was not necessary to place walls in between those areas.

4.2 Scaling Down

The first part of the project, represents the part within my comfort zone. The main base and the game entities did not represent a challenge to me.

But due to several gameplay changes during development, I had to discard several features of the game. One of the biggest changes was the introduction of assignable tasks.

The level design part was a huge problem to be tackled. The whole level had to fit onto one single screen with the maximum resolution of 2K HD.

The major problem was the question of scaling. To fit a large map onto a single screen, we need to scale everything down.

But how large is large enough? And how small can an unit be scaled down?

This was a huge problem, which made us reconsider our concept of the shared screen from time to time. We were not able to predict whether we would be able to fit everything onto one single screen in a reasonable manner and whether this concept would work at all.

Furthermore, it was quite difficult to predict, what kind of level design would work, without having the AI ready to interact on it.

Although I had to redo a lot of work and make frequent changes to my code, I am very satisfied with the result of these tasks.

I think it was necessary to try out a lot of different features in order to get to the final conclusions and keep the good ones and throw out the bad ones.

The shared screen concept in combination with the 2K HD resolution was representing a tough constraint and I am glad, that we figured out a solution.



We also experimented with different angles of projection. One idea was to develop this game as a table surface game. We dismissed this idea, because it did not provide enough benefits to balance out the complicated setup.

5. Understanding Architecture

There is a differentiation between: *what is performing the action* and *what is controlling the performer*, through *which controlling device*.

The performer, could be the representative game entity, the avatar for example.

The controller, could be a human being controlling this avatar through any input device, but also an non human system, like for example an AI could be the controller.

This comes especially handy, whenever the controller wants to change its performer, like for example in racing games, where the player (her/his brain) chooses between different performing car models before the race, or like in first person shooters, when the performer dies and a new performer has to be assigned to the controller in order to jump right back into the action.

Even more so, it enables easy setup to let AI control the same type of performer, as a human being would. This comes handy if, for example empty slots need to be filled up with bots.



All these mentioned cases do not apply for our game.

Each player has exactly one performer, and this one performer cannot die nor will the player be able to change it. The performer of the player in our game is her/his cursor, which he can be used to select units or objects on the map and to change their state.

The cursor will never be possessed by any another *human* or *AI controller*. The units themselves, which are also *performer*, are starting off with an *AI controller*, which will never be changed and they will never be possessed by any *human controller*.

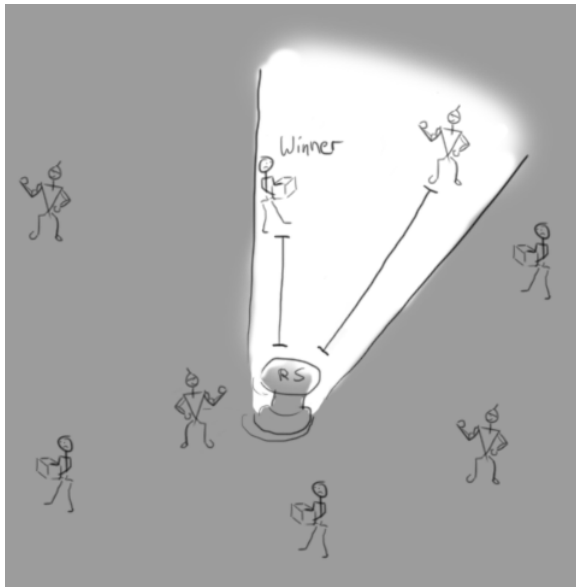
Although it was not necessary for this project, it wanted to keep the general structure of *performer*, *controller* and *controlling device* right, for best practice.

Those terms can get quite confusing and are by no means standardized! Like for example in Unreal Engine 4 the *controlling device* is being called *controller*.

Those terms are purely laid out for myself. I found this to be the easiest way to describe the architecture and structural dependencies.

5.1.1 Dual Cursor System

Since the cursor is the player's real *performer*, it was important to make it feel very natural and easy to use. It took me several attempts to get the cursors controls to this final state. Each player gets two different cursors. One is represented as a circle on the floor and one is represented as a triangle above the floor.



Selecting

In order to give specific commands to specific units, the player needs to be able to select those specific units.

This can be done with the circle cursor, by using the left analog stick of the game pad. To select a new unit, the player needs to pull the stick into the direction of the desired unit and the circle cursor will automatically snap onto the next unit.

Pulling the analog stick into a direction will result in a corresponding vector. A function calculates the vectors from the current unit to all other units and compares it with the vector of the analog stick. If the deviation is within an acceptable range, it will save the concerning unit into an array. Having an array of all units in acceptable direction, the next step is simply to pick the one out of the array, which is closest to the currently selected unit. After that, the array needs to be reset and a cool down of less than a second is initialized to keep the cursor from skipping units.

At last, the cursor's location will be interpolated towards the newly selected unit, which makes its snapping visible.

Assigning Tasks

Having the desired unit selected, the player is able to assign specific tasks to the unit with the triangle cursor.

In order to snap the triangle cursor on relevant tasks on the screen, the player needs to push the right analog stick into the direction of the desired object. Just as the circle cursor, it will automatically snap onto the next object.

In order to command the unit, which is selected by the circle cursor, to go to perform a task on the object, which is selected with the triangle cursor, the player simply needs to pull the right shoulder trigger.

Pulling the left shoulder trigger will result the unit to cancel its task and to hold the current position.

Upgrading, Selling ...

An unit, which is selected with the circle cursor, can also be upgraded or sold.

A simple press on the top, left or right face button of the game pad will subtract the total resource amount of the upgrade cost and upgrade the unit by one in either speed, power or intelligence.

Toggle Modes

To get back to the main base mode and create additional units, the player can either move the circle cursor to the main base by pulling the left analog stick into the direction of the main base or simply use the special left gamepad button to toggle between those two modes.

5.1.2 AI performer

Before setting up the Artificial Intelligence, I need to set up its representative. Meaning in this case, the performing body which can execute tasks in the environment.

I have been setting up two different blueprints, representing the units. One for the worker class and one for the fighter class.

The main function of these blueprints are to provide the AI controller a visible body which has a location in 3D space and shows their activity through visuals like for example through animations and icons.

These bodies are not only visible to the players, who are controlling the cursors, but are also senseable to other AI controllers, which are also controlling other bodies in the game.

Because the units can interact with each other, their bodies are also carrying information about, their functionalities. Just as with the “interactables”, I have set up a Blueprint Interface for characters. This Interface adds unified events like for example, “receive damage” and “get robbed” to those bodies.



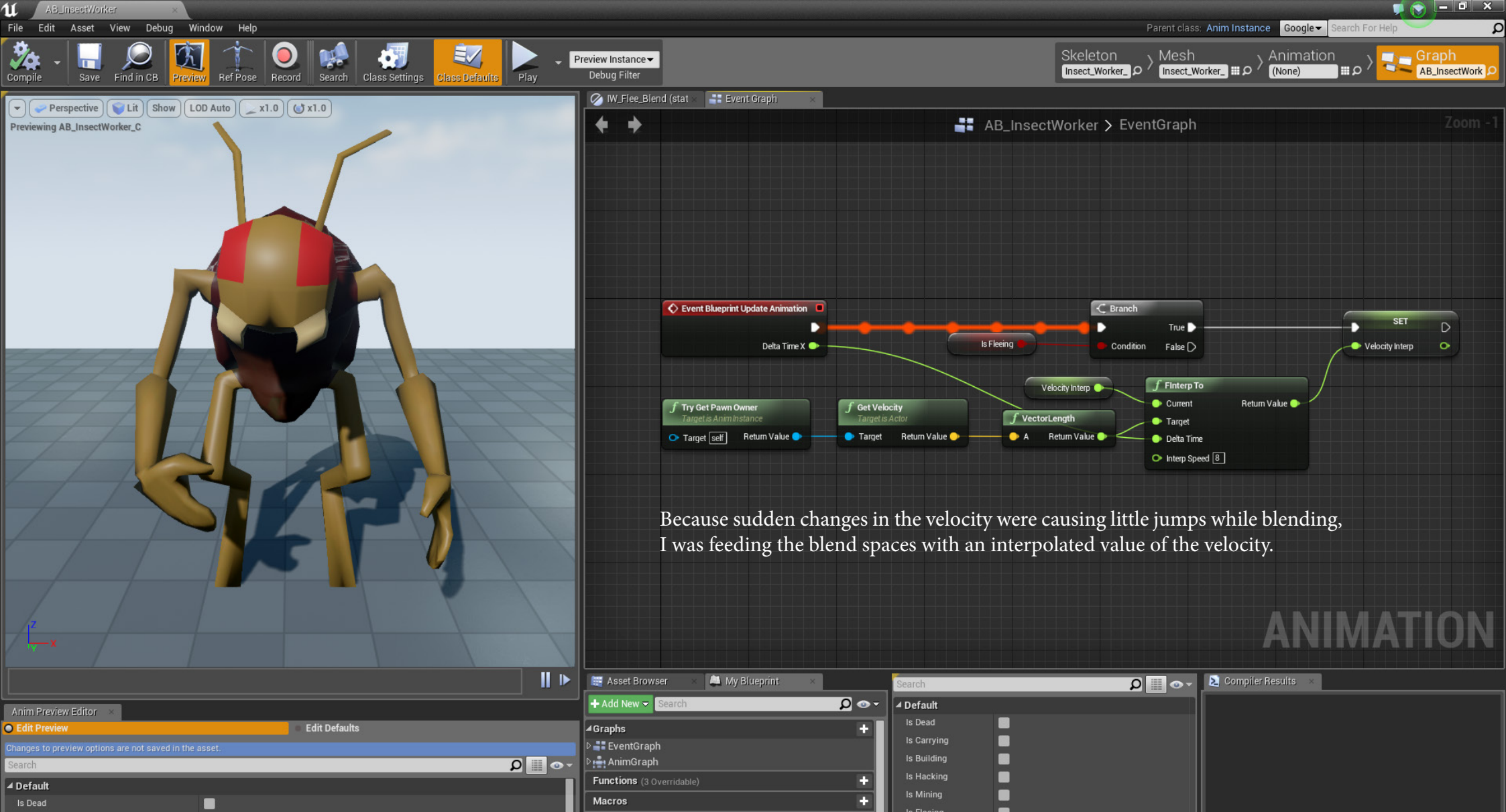
5.1.3 Animation

Another Task of mine was to implement the animation assets into our game. That involved setting up the Animation Blueprints and Animation Blendtrees.

Animation is important to give the player visual feedback about the state of the game. It is necessary to communicate and visualize the current activity of the units.

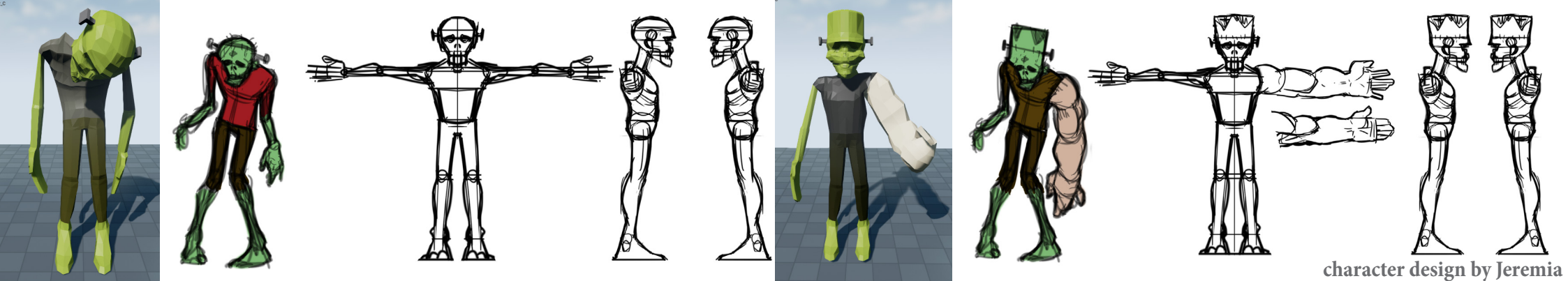
To guarantee a smooth transition from idling to running, I was setting up one dimensional blend spaces with loopable animation assets of idling, walking and running.

Depending on the velocity of the unit it would perform a scaleless blend from idle to walk and from walk to run.



Because sudden changes in the velocity were causing little jumps while blending, I was feeding the blend spaces with an interpolated value of the velocity.





AB_RobotWorker

File Edit Asset View Debug Window Help

Compile Save Find in CB Preview Ref Pose Record Search Class Settings Class Defaults Play Preview Instance Debug Filter

Skeleton Robot_Worker_ Mesh Robot_Worker_ Animation (None) Graph AB_RobotWork

Perspective Lit Show LOD Auto x1.0 x1.0

Previewing AB_RobotWorker_C

AB_RobotWorker > AnimGraph > New State Machine

Zoom 1:1

This is an example of the worker units State Machine.

A State Machine regulates the timing of the transition from one animation state to another.

It observes whether predefined conditions for a transition are matched and performs the transition from one state to another if necessary.

Robot_Worker_Mine

Robot_Worker_Death

RW_Walk_Blend

RW_Heavy_Blend

RW_Flee_Blend

ANIMATION

Asset Browser My Blueprint

+ Add New Search

Graphs

EventGraph

AnimGraph

Functions (3 Overridable)

Anim Preview Editor

Edit Preview Edit Defaults

Changes to preview options are not saved in the asset.

Search

Default

25

File

Edit

Asset

View

Debug

Window

Help

Compile

Save

Find in CB

Preview

Ref Pose

Record

Search

Class Settings

Class Defaults

Play

Preview Instance

Debug Filter

Skeleton

Rig_Skeleton

Mesh

Rig

Animation

(None)

Graph

AB_InsectFight

AB_InsectFighter_C

Perspective

Lit


Show

LOD Auto

x1.0

x1.0

Previewing AB_InsectFighter_C



Event Graph

Anim Graph

AB_InsectFighter > AnimGraph

Zoom -1

State Machine

State Machine

LocomotionCache

Pose

Use cached pose 'LocomotionCache'

Use cached pose 'LocomotionCache'

Slot 'UpperBody'

Group 'DefaultGroup'

Source

Layered blend per bone

Base Pose

Blend Poses 0

Blend Weights 0

1.000000

Add pin

Final Animation Pose

Result

... and stiched onto an active pose, beginning from a dedicated bone.

This way, the fighters final pose can receive the locomotion pose of the state machine while performing the attack animation for the upper body.

Asset Browser

My Blueprint

Search

+ Add New

Graphs

EventGraph

AnimGraph

Functions (3 Overridable)

Macros

Variables

IsDead

IsFleeing

VelocityInterp

Default

Is Dead

Is Fleeing

Velocity Interp

0.0

Root Motion

Root Motion Mode

Root Motion from Montages Only

Compiler Results

Anim Preview Editor

Edit Preview

Edit Defaults

Changes to preview options are not saved in the asset.

Search

Default

Is Dead

Is Fleeing

Velocity Interp

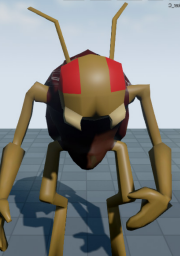
0.0

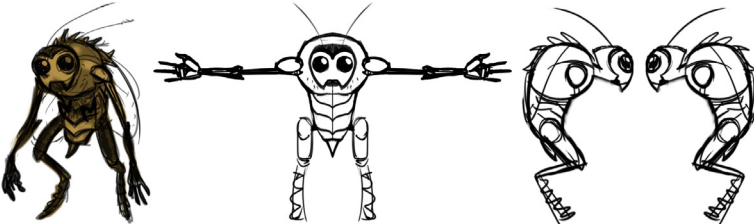
Root Motion


IsDead

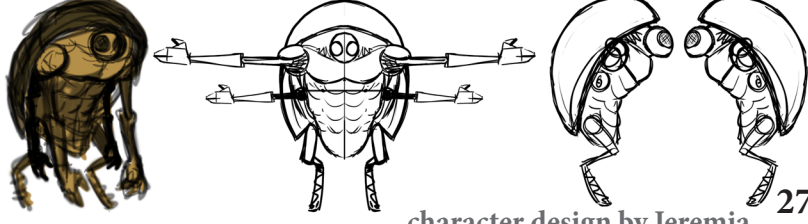
IsFleeing

VelocityInterp









character design by Jeremia

27

5.1.4 Artificial Intelligence

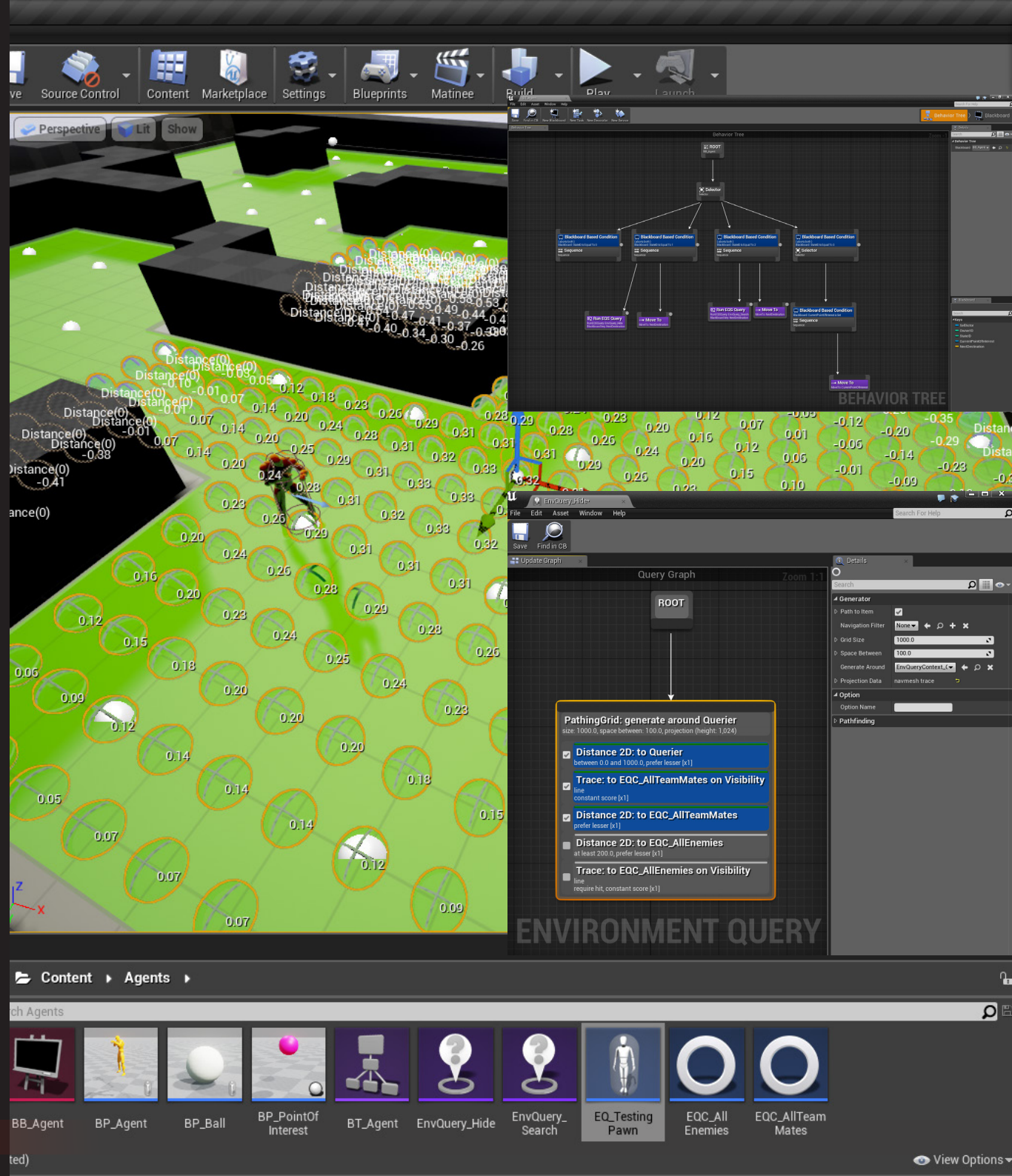
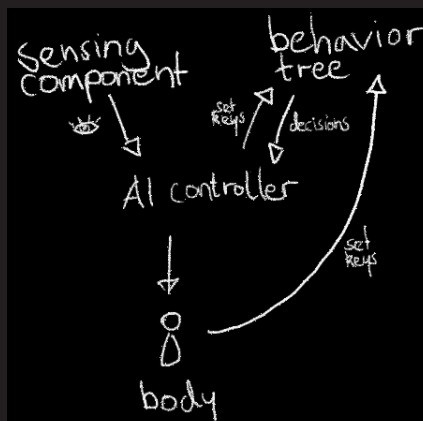
After having setup the representative body for the units, I now needed to feed it with a behavior.

For that I needed to setup an AI Controller and a Behavior Tree with all its Services, Decorators and Tasks.

According to my previous example the representative body would be the “performer”, while the behavior tree would be the “controller” and the AI controller would be the “controlling device”

This is where it gets a little bit confusing:

The sensing component enables an AI to sense his environment through vision or hearing. Unlike with human players, the AI does not look at the screen, but rather senses the surroundings out of the bodies perspective. Nonetheless the sensing component is part of the AI controller.



The AI Controller does not only possess the body and runs the behavior tree on it, but also contains the sensing component. With this sensing component, the AI is able to sense specific objects in its surroundings. In this case, the sensing component can sense all the intractable game objects.

Every sensed object is being categorized and analyzed in order to figure out, whether it is relevant and should influence the behavior.

In this example a worker is sensing a fighter:

- > is this fighter one of ours?
- > is this fighter already occupied with something?
- > am I already threatened by another fighter and if so, is this other fighter closer to me than the sensed one?

If any of these checks are answered with yes, then the sensed fighter will be ignored. If all of them are answered with no, then the sensed fighter is a enemy fighter, which has no other occupation right now and is closer than any other threatening fighter.

This relevant fighter information will then be sent to the behavior tree and saved into a variable. Those variables, which are being set by the sensing component in combination with variables which are set by the player, when assigning tasks, are forming different combinations, which can result in different outcomes for the behavior.

In this example we have a fighter with different conditions setup for entering a certain behavior:

fighter sensed relevant enemy worker
--> go to enemy worker

fighter sensed relevant enemy fighter
--> go to enemy fighter

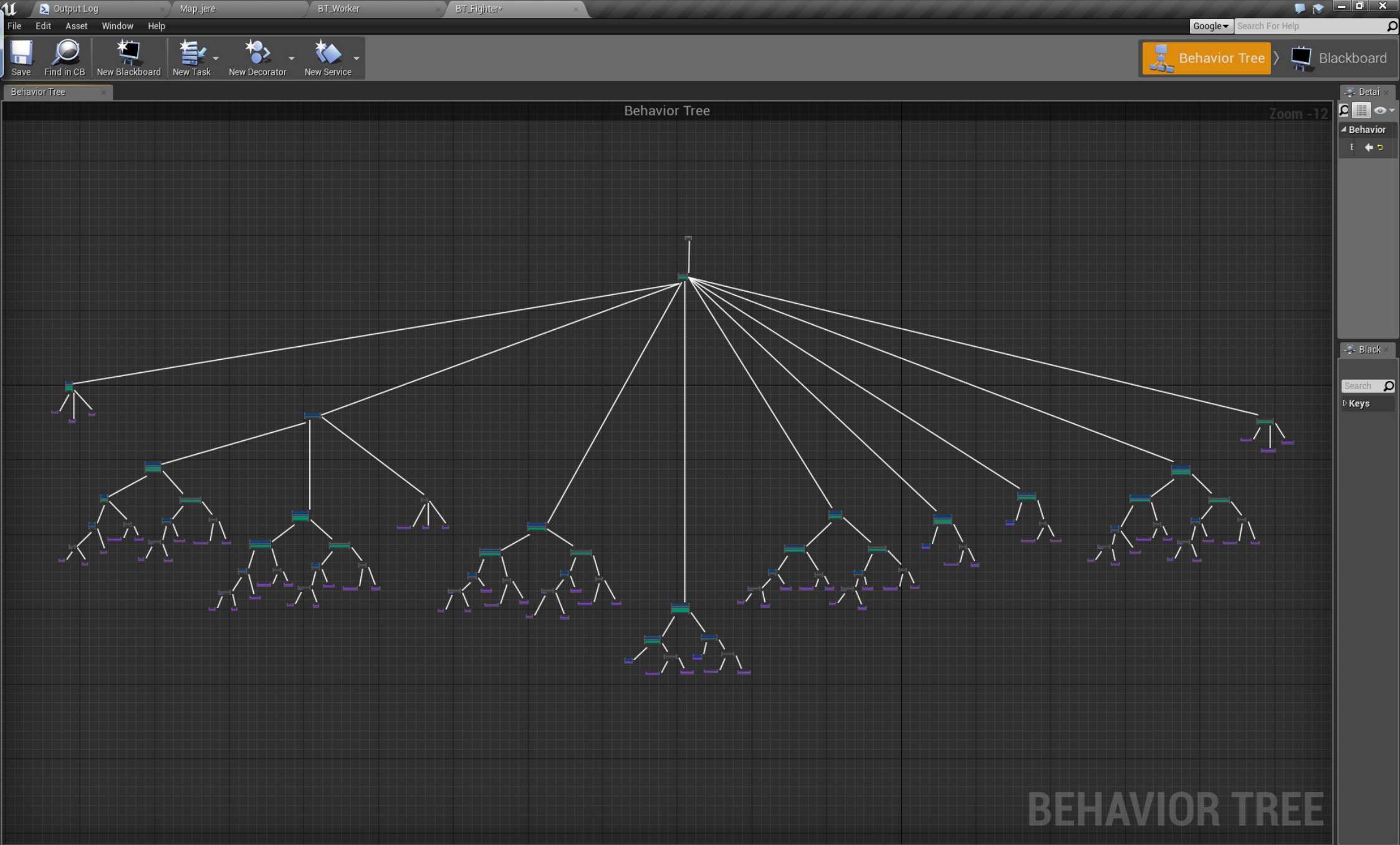
fighter is low on health
--> hide

Having these conditions and behaviors ready, we can now setup the priorities for each branch. In this example the lowest condition would have the highest priority, whereas the second lowest has the second highest.

Having these priorities setup will let the behavior tree decide which behavior is more important.

In our example: If the fighter senses an enemy worker and an enemy fighter at the same time, he would approach the enemy fighter, due to its higher priority. In any case, if he is low on health he would flee, since it has highest priority.

There are way more AI functionalities in UE4 and the setup of the AI gets way more complex, but describing the whole behavior setup I did for this game would go far beyond the scope of this paper, really.



the final Behavior Tree of the fighter unit

5.2 Giving Power to the Player

I am very satisfied with the result of the cursors, but creating them took way longer than expected. I have heavily underestimated the required time to get this running as it is now.

The real difficulty of this project came with setting up the units and their behavior. It took me quite a while to get to know how to use the Behavior Tree and the Environment Query System of the Unreal Engine 4.

During development, I had to realize, that the scope I have been setting for myself in terms of the AI system was too large. This project is the first time, me being confronted with creating a complex AI system.

I did not need to start from scratch, because the Unreal Engine 4 comes with a lot of essential AI components (Perception, Behavior Trees, Path-finding System), but also advanced AI components (Environment Queries).

Having this in mind, I simply underestimated, how complex AI systems can get and how difficult they can become to manage, even with all the components ready out of the box.

The tools for creating such an advanced AI were quite unfamiliar to me and I had to spend most of my time figuring out how all these different components work.

A lot of things I have done were worst practice. But, even this helped me a lot to understand more about what I am actually doing and why it is not practical or even totally wrong.

Most of the time, I have been working on this project, went into optimizing AI and then discarding the whole work again. But right now, I am at the point, where I think, that I really understand the concept of the AI components.

I realized, that the initial idea of creating an AI, which would be mostly independent from the player and makes smart decisions, was too ambitious as the first project involving advanced AI.

Especially, because we included a stats system, which gave each unit own unique strengths and weaknesses, the major problem was the dependencies of decision making. In order to make smart decisions, the AI not only had to always evaluate its own capabilities, but also the strengths and weaknesses of the others and constantly communicate with them.

As an example:

a fast unit is good for collecting resources, while a smart unit is more efficient unlocking doors. In this scenario, the fast unit might want to reach resources which are behind a locked door. Depending on how many other units are available he would need to either call for the most efficient other unit or unlock it himself. In order to calculate the most efficient decision, it would not be just enough to compare the stats of all units. There are several variables which have to be taken in account, like for example: What is each unit currently doing, how far away from the door is each unit, how long would it take each unit to unlock the door. And again, if another unit would drop its task to open the door, should this task be taken over by some other unit?

From this point, the whole chain of dependencies would loop around again.

Because it became too complicated to quickly, we had to implement a feature, which gives the player the power to make decisions for her/his units.

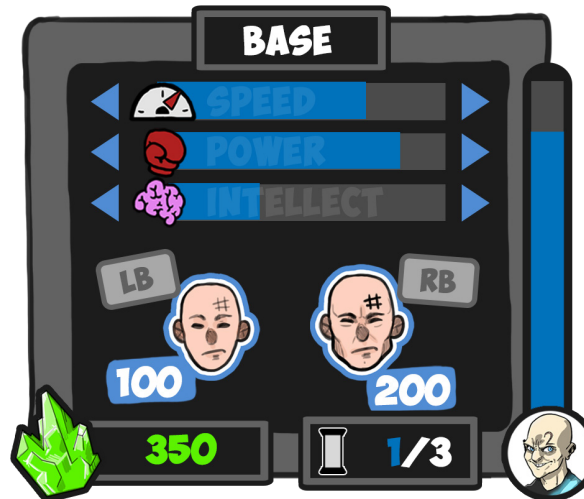
6. Graphical User Interface

The Graphical User Interface provides crucial information about the game state. Because the screen is shared by 4 players, we had to figure out how to implement the graphical user interface without using up too much of the screen. We decided to place one info panel for each player next to her/his base and to use icons and health bars only when necessary.

While Jeremia was creating all the graphical assets, my task was to implement their functionality.

For that I was using the UE4 Widget Blueprint. While setting up the bindings for the UI elements, I was confronted with two major different setups. Either to let the widget look into the necessary locations of the code and react accordingly, or let different bits of codes update the widget whenever needed.

I figured, that the Widget would check the variables in different part of the codes for every frame, even when no changes appeared.



The two options as an example for displaying the amount of game currency available:

- 1.) Widget goes to the referenced main base and reads out the variable for the game currency. (This happens every frame)
- 2.) Widget has a variable for game currency. This variable is updated to mirror the game currencies variable of the main base whenever necessary. (This update is called, when the player gains or spends game currency)

Because I was afraid, that the first option would have an impact on the performance, I decided to let the code overwrite the variables in the Widget only when needed.

Everytime an action is performed by the player, which would change the amount of currency she/he has, a function to update the widget is necessary to reflect the change.

6.1 Icons

I would not do it this way next time anymore, because of several reasons.

It is very error-prone: every time a feature, which is anyhow connected to the GUI, is added or removed, you have to think of, where and when an GUI update is needed to be called. This is very tedious and very easy to forget.

Nonetheless, I stucked to the 2nd method for this project. It went well and the game is calling out all necessary updates reliably. But I am sure, that with an even more complex economical system it can get quite confusing to figure out, which action will influence which parts of what GUI. Not to mention in a multiplayer network game in which the clients can influence each others GUIs.

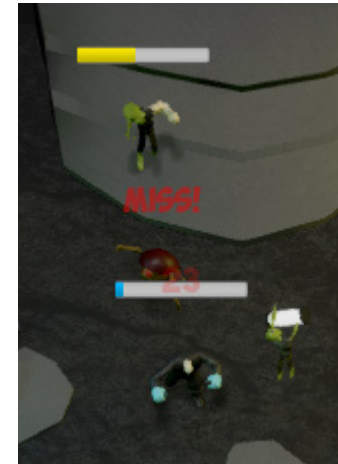
In order to display the health and interaction progress of each unit, we decided to use icons, which are shown next to the concerning unit.

This is realized by adding a widget component to the units blueprints.

This widget component contains items for hit points, health bar and a progress circle.

These items are then made visible when needed and are fed with variables, like for example the hit points, the current health or the current progress of the current interaction.

The health bar for example is only visible if the unit does not have full health. This way we emphasize damaged units and avoid spamming the screen with full health bars.



6.3 Reaction Screens



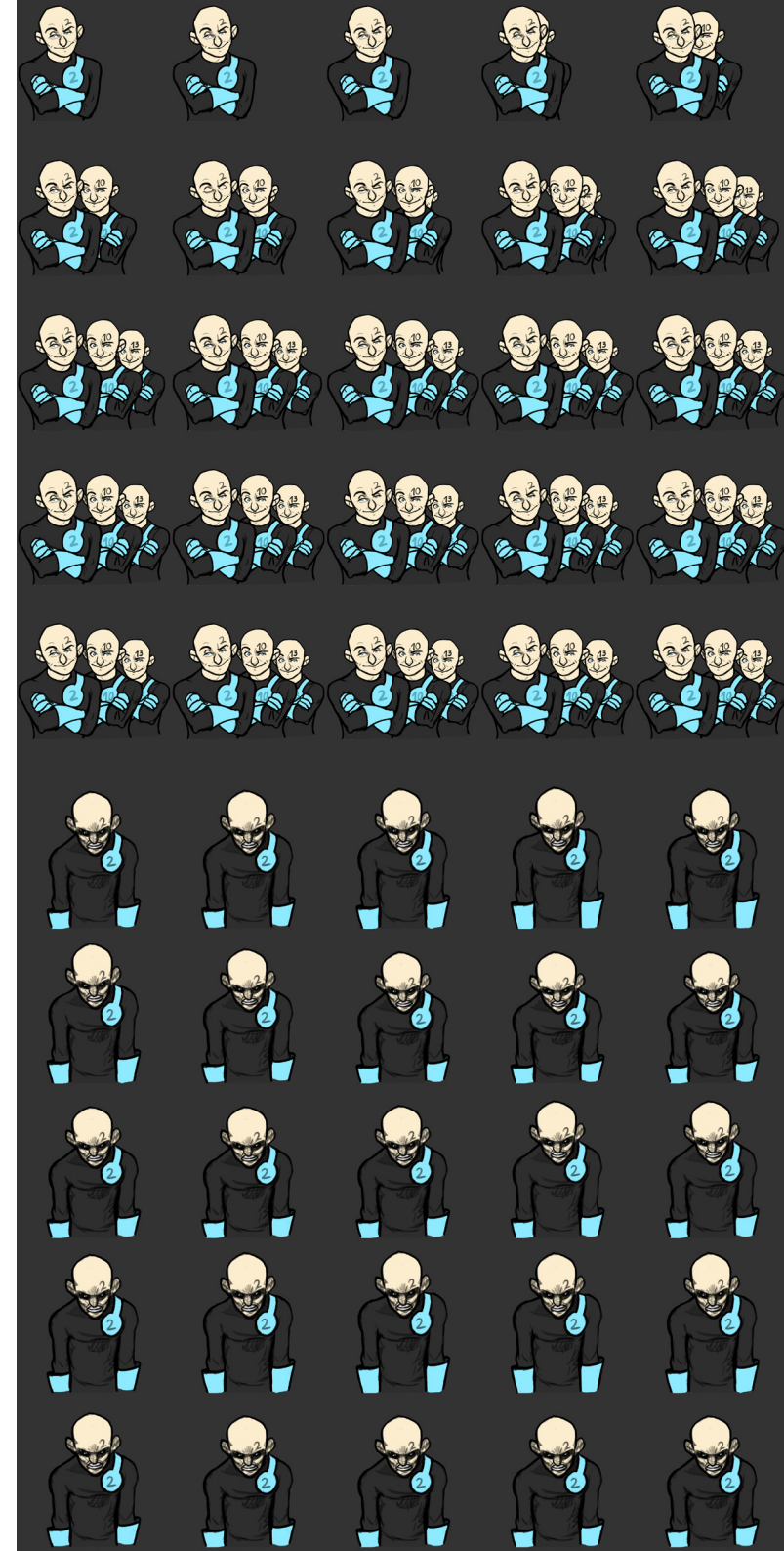
Unlike other real time strategy games, in which each player is handling a rather large amount of units, the players in dominion are handling a relatively small amount of units in comparison.

This also means, that every loss of an unit has a higher weight.

We wanted to emphasize that by giving the players strong feedback about the happenings on the screen.

Because we think games are all about feedback, we created screens, which are appearing from time to time, showing the reaction of the scientists.

My task was the implementation of those. I realized it with materials, in which I set the frame of the animation through an parameter by runtime.





Characters by Jeremia, Animations by Lina





from early prototype



to finished game

6. Recap

Compared to previous projects, this turned out to be the most complicated one so far. I have tremendously underestimated the workload of complex AI systems. It took me a lot of time to get used to the AI components of Unreal Engine 4 and I needed numerous alliteration steps.

Nonetheless, I feel like I have learned a lot about AI programming, which is most important.

I have the feeling, that I have learned a lot about game AI and that I have a much better understanding about it now. I think that this knowledge will come very handy for future projects.

The concept of the game has changed a lot during development. Features had to be replaced or removed. One of the major changes was the introduction of the assignable tasks. I think, that only through these several alliteration processes, it was possible to get into that final stage.

I have to learn to set the scope right next time, in order to get my work/life balance right. The crunch time was getting very tense and I want to avoid that for future projects.

In overall, I am very satisfied with the outcome of this project. The game is fun and helped me a lot to understand the basics of game AI.

I am very interested in creating a server/client network game and a mobile game for future projects.

Jeremia is happily working in our studio.



my workplace :)

