

TBL. OF CONTENT









-  Game Concept
-  How-To-Play
-  Implementation
-  Architecture
-  Testing
-  Peer Reviews
-  Shortcomings
-  Future Work



Figure 1: Grid-based world



Figure 2: The wizard

FOR THE WIN

Thomas Desbans & Timo Frank

An Unity ECS-based RTS Game Concept

GAME CONCEPT

“ If you are far from the enemy, make him believe you are near. **Sun Tzu** ”

Real Time Strategy Games (RTS) classify themselves as a mind duel between you and your opponent. It is all about trapping and baiting and requires the players to develop a strategy. Reactivity and creativity skills are important. Most of the RTS games are quite similar: you need to build buildings to spawn units and then wage war until you destroy your opponent's headquarter.

The world of For The Win can be described as RTS game on a grid-based 2D surface (see figure 1) like in games such as Warcraft¹ or Dune 2². Each cell is either walkable or not and can contain only one object at a time. Units can walk from one cell to another by striding surrounding cells in pathfinding and therefore meet other objects. Objects are:

1. **Units** : wizard, knight, elf, worker
2. **Buildings** : headquarters, barrack, farm
3. **Harvestables** : tree, goldmine

In general the same rules as in classic RTS games apply. For example, you start with a given amount of resources. Once you have workers, you can start chopping wood or mining some gold to gather resources. They are transported to the headquarters. Additionally, you can build some structures that enables you to build other types of units. You can train new units to enhance your resource income or to attack the enemy. However what distances For The Win from conventional games are some specific rules regarding the king. He and all other units are described further:

1. **Wizard** : You start the game with this unit. He is your king and shall live forever
 - ✓ Unique, if you loose him, you loose the game
 - ✓ Awesome in combat
 - ✓ Can build a headquarter

¹https://en.wikipedia.org/wiki/Warcraft:_Orcs_%26_Humans

²https://en.wikipedia.org/wiki/Dune_II



Figure 3: The worker



Figure 4: The knight



Figure 5: The elf



Figure 6: The headquarters



Figure 7: A barrack



Figure 8: A farm

2. **Worker** : A weak pathetic peasant, which is useful to keep your kingdom running
 - ✓ Costs 50 gold, 0 wood
 - ✓ Weak in combat
 - ✓ Can chop trees and gather gold from the goldmine
 - ✓ Can build barracks and farms
3. **Knight** : A strong melee unit that can resist to a huge amount of damages due to its protective armour
 - ✓ Costs 100 gold, 0 wood
 - ✓ Good in combat
 - ✓ Attacks enemies with his short-range attack
4. **Elf** : A powerful range unit capable of inflicting serious damages from distance
 - ✓ Costs 100 gold, 50 wood
 - ✓ Good in combat
 - ✓ Attacks enemies with his wide-range attack

You now have many soldiers waiting for your orders. Your goal is to kill your opponent's wizard by any means necessary. You can use your wizard to bait the enemy's army and kill all his farming units, but be careful. Your enemy could also be baiting you to kill your wizard.

Buildings are important to build up your kingdom. After it is clear which unit can build which building, the following describes the function of each structure:

1. **Headquarters** : The cornerstone for each economy and the place where your wizard lives
 - ✓ Costs 1000 gold, 200 wood
 - ✓ Can produce workers
2. **Barrack** : It is the training ground for your combat forces
 - ✓ Costs 400 gold, 150 wood
 - ✓ Can produce knights and elves
3. **Farm** : Produces food to help your kingdom grow
 - ✓ Costs 100 gold, 50 wood
 - ✓ Is needed for exceeding the unit limit



Figure 9: Main Menu



Figure 10: The four units

HOW-TO-PLAY

When the game starts, you are in the main menu (see figure 9). From this menu you can either go for a new game or proceed to a helping page, which tells you how to play the game. As soon as you click on "New Game", the game starts. The previous chapter already introduced all characters and the general purpose of the game. To sum it up you should:

1. **Build some workers** : you start the game with your wizard and a small amount of wood and gold. Build a headquarters with it and create some workers.
2. **Harvest resources** : start chopping wood or mining some gold. Try to protect your peasants from enemy attacks. you should always pay attention to your opponent's moves and try to predict his plans.
3. **Build an army** : Once you have enough resources to build a barrack, you can then create fighting units. Positioning your units is essential in this kind of game. As every unit has the same speed, if your opponent startles you and your units are not well positioned, you will loose the battle. If you combine the range of the elves and the environment (rivers for example), elves can become very strong.
4. **Wage war** Try to defeat the enemy wizard to win the game. As soon as one player's wizard dies, the game is over and each player is sent back in the main menu.

How to move

To move units, you first need to **select** them. Either you can left click and drag a box selection like seen on the left side of figure 24, or you can left click on them while pressing on the left shift key. Once units are selected, you can see a green circle under them, it means that all these units are waiting for your orders. Then just **right click** wherever you want (see the right side of figure 24). If you click on an **enemy**, a sword icon will appear and your units will automatically gather around the enemy. If you click on a **resource**, a tool icon will appear and your units will go to the resource and start harvesting it if they're able to do so. If you click on an **empty cell**, all your unit will move and gather around this cell. Note that units are still selected after every move, this way you can easily carry out several tasks with your units.

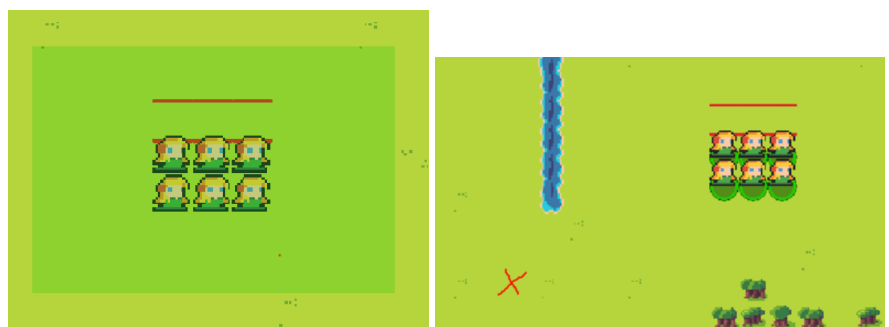


Figure 24: How to move



Figure 11: How to harvest wood



Figure 12: How to harvest mines

How to fight

To fight, you first need to select your units and then right click on the enemies you want to fight. If you have selected more units than you needed to fight, then those who cannot fight will stay in position waiting for other orders. The procedure is visualized in figure 25.

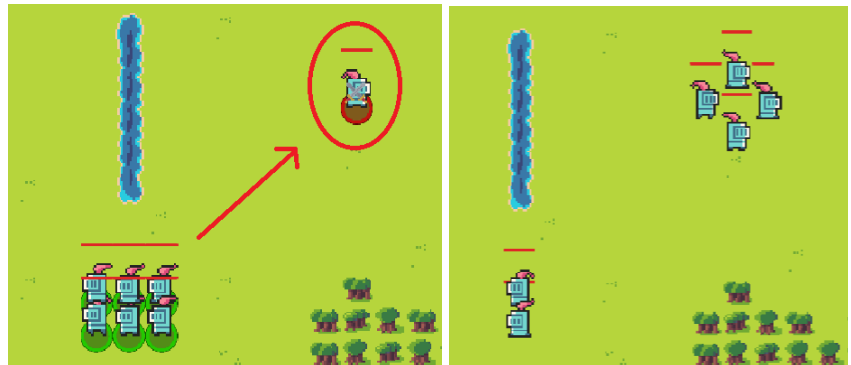


Figure 25: How to fight

How to build

To build, you first need to **select** a unit that is able to build, either the wizard or the peasants. Then a small box, the context menu, appears on the left bottom side of your screen (see left side of figure 26). You now need to **left click** on the building you want to build and then just left click wherever you want on the map. The building will only be built if you have **enough resources** and if the **cell** where you want to build is **free**. Once you have buildings, just left click on the buildings and look at the same box. As the right side of figure 26 visualizes, some units will be available to build. If you have enough resources, just left click on the units and they will spawn all around the building. You can now give orders to these new units.



Figure 26: How to build and spawn units

How to harvest

To harvest resources, just select your workers, and then, click right either on a tree (see figure 11) or on the goldmine (see figure 12). The unit will go to the resource, farm it and then bring the resource back to the headquarter automatically.



Figure 13: Unity logo

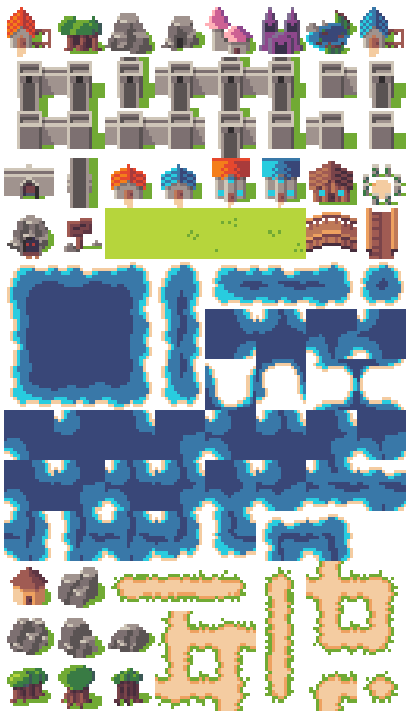


Figure 14: The "Overworld Tileset"-tileset

IMPLEMENTATION

To build our game, we decided to use the game engine **Unity** because we both had some knowledge about Unity and C#. When we started thinking about how we would like to build the game, we realized that Unity recently introduced a preview package that enables a new way of coding:

Entity Component System (ECS) is a new way of writing code that provides high performance by default. Instead of dealing with Game-Objects, we deal with entities in a component-system environment. An entity can have components that enrich its behavior. Components are data stores and always linked to an entity. Systems are actors that can gather and filter entities by their components and perform operations on them or the environment. Unlike the previous way of coding, which is object-oriented programming, this uses a data-oriented program approach. It promises a high performance and clean code structures. So even though we had never used ECS before, we decided to use it for our game. Obviously, we had to learn many things in order to understand the logic of this new way of programming.

Setup

We installed Unity 2019.2.14f1 Personal with the default 2D Frameworks. Additionally we supplemented the project with the following steps to enable our intends:

1. We adjusted a few project settings to make it GiT ready³. This includes setting the version control mode to "visible meta files" and to force text in asset serialization. This ensures that for example changes to the scenes are recognized by the git version control.
2. We installed Unity Preview Frameworks (not recommended for production use) to enable ECS in our Editor:
 - ✓ **Entities (v0.1.1)** : enables entities, components and systems as well as the Unity entity debugger
 - ✓ **Hybrid Renderer (v0.1.1)** : enables to draw entities on the screen
3. We installed additional visual Unity Frameworks:
 - ✓ **2D Tilemap Editor (v1.0.0)** : provides Unity with a built-in editor. Since our game is a 2D grid based RTS game, this extension is extremely helpful
 - ✓ **TextMesh Pro (v2.0.1)** : improves Unity UI texts massively. We use this package for ingame texts like on the menu or the game HUD

Any used assets in this project are gathered from itch.io. For units we obtained "Dungeon Tileset II"⁴ under CC-1.0 license from 0x72

³<https://thoughtbot.com/blog/how-to-git-with-unity>

⁴<https://0x72.itch.io/dungeontileset-ii>

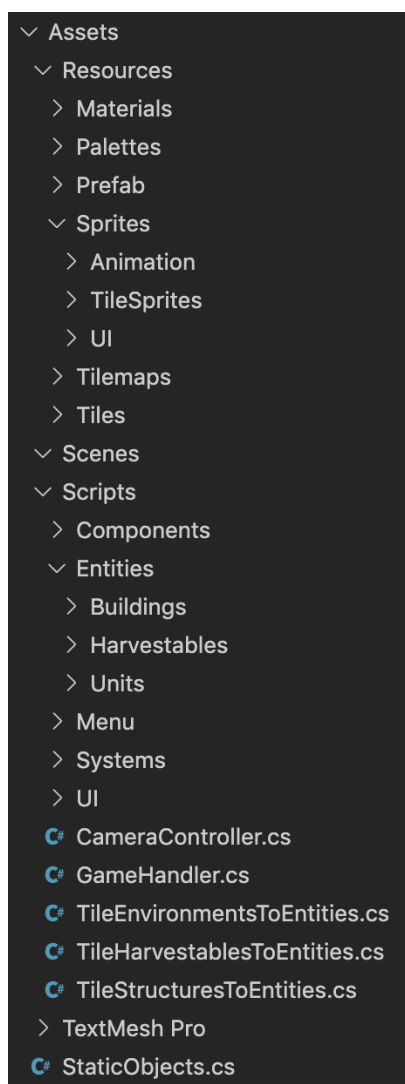


Figure 15: The project folder structure with all subfolder expanded




and our environment tileset relates to "Overworld Tileset"⁵ (see figure 14) under CC-4.0 from WaltonSimons. Our UI has some free elements from "PX UI"⁶ from Karwisch.

Figure 15 on the left side shows you our chosen project folder structure with all subfolders expanded. Folder **Resources** contains all materials and textures that shall be displayed on the screen. This comprises UI/HUD elements, alternative mouse cursors, tilemap palettes, unit sprites and animation images. **Scenes** holds the Unity scenes for MainMenu, HelpMenu and the Game scene. In **Scripts** you can find all C# scripts regarding the execution of Unity and Unity ECS scripts. They are further described in the upcoming Architecture subsection. As the name suggests, the Components folder holds all ECS components, and the Systems folder all ECS Systems. Within the Entities directory you can find all building, harvestable and unit related read-only properties like costs, health or sprite names. The Menu and UI folders contain corresponding UI scripts and any top level scripts are relevant in a global point of view. **TextMesh Pro** contains some framework files that are needed to use the framework.

System Requirement :

As we are using **Unity**, the game can easily run on **Windows**, **macOS** and **Linux** systems. The fact that we use the ECS way of coding provides us a high performance. Besides, it is a 2D game. So, the game can be run on basic computers and does not require a powerful processor and graphic card.

Here are the minimal requirements needed :

- ✓ **CPU** : x64 architecture with SSE2
- ✓ **Graphics** :  DX10 capable
 -  Capable Intel and AMD GPUs
 -  OpenGL 3.2+ / Vulkan capable

ARCHITECTURE

For The Win has four different **entity** types that start off with some preset **components**:

1. **Unit** : a fully rendered, moveable entity, which correspond to a unit object in classical RTS games. It holds the following components:
 - ✓ **Translation, RenderMesh, LocalToWorld** : cares about the positioning and drawing of the entity in the Unity world
 - ✓ **UnitComponent** : holds the unit type
 - ✓ **AnimationComponent** : tells, which animation is currently playing for this entity
 - ✓ **TeamComponent** : decides, if the entity is friendly or an enemy

⁵<https://waltonsimons.itch.io/16x16-overworld-tileset>

⁶<https://karwisch.itch.io/pxui-basic>

- ✓ **HealthComponent** : holds the current amount of life points of the entity
- ✓ **FightComponent** : stores the current target of the entity and additional information such as if the entity is currently fighting

2. **Structure** : a fully rendered, static entity, which correspond to a building object in classical RTS games. It holds the following components:

- ✓ **Translation, RenderMesh, LocalToWorld** : see previous entity
- ✓ **StructureComponent** : holds the building type
- ✓ **BlockableEntityComponent** : tells systems that this entity can not be walked over

3. **Harvestable** : a fully rendered, static entity, that can be addressed for harvesting. It holds the following components:

- ✓ **Translation, RenderMesh, LocalToWorld** : see previous entity
- ✓ **BlockableEntityComponent** : see previous entity
- ✓ **HarvestableComponent** : holds the harvestable type, the amount that can be harvested and how long it takes to harvest this entity

4. **Event** : an invisible short-lived entity, which is used to be fired and deleted shortly after. It has corresponding systems that listens for those events. It holds the following components:

- ✓ **SlotClickedEvent** : holds information about which slot was clicked in the context build menu

Conventional Scripts

Conventional Scripts are Unity's MonoBehaviour scripts, that can co-exist with the ECS system. We use those to access default task like UI management or Unity's camera movement.

User Interface (UI) controls the display of a HUD frame with game information. It cares about any gold and wood resource changes and displays them (see figure 21). Additionally, it handles the context menu on the bottom of the game window, where buildings or units can be built after you've selected an entity (see figure 22).

The **CameraController** element handles any camera movements that can be done with the arrow keys.

GameHandler is a static instance that represents a hodgepodge for all links to other Game Objects. In all other scripts, you can access Unity content and links (for example to the tilemap) from this class.

Tile...ToEntities are scripts that run once in the beginning of the game. They iterate through all tilemap cells and converts any tiles into entities. We discovered, that accessing tiles and parsing their content is really hard from an ECS point of view. Because of this, we made these scripts so we can easily replace the tiles with entities at the exact same place. The gain is enormous as we now must only iterate through relevant entities instead of parsing the whole tilemap over and over again.

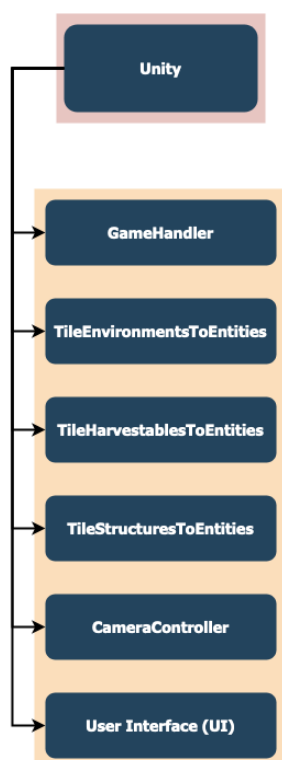


Figure 16: List of all used conventional scripts

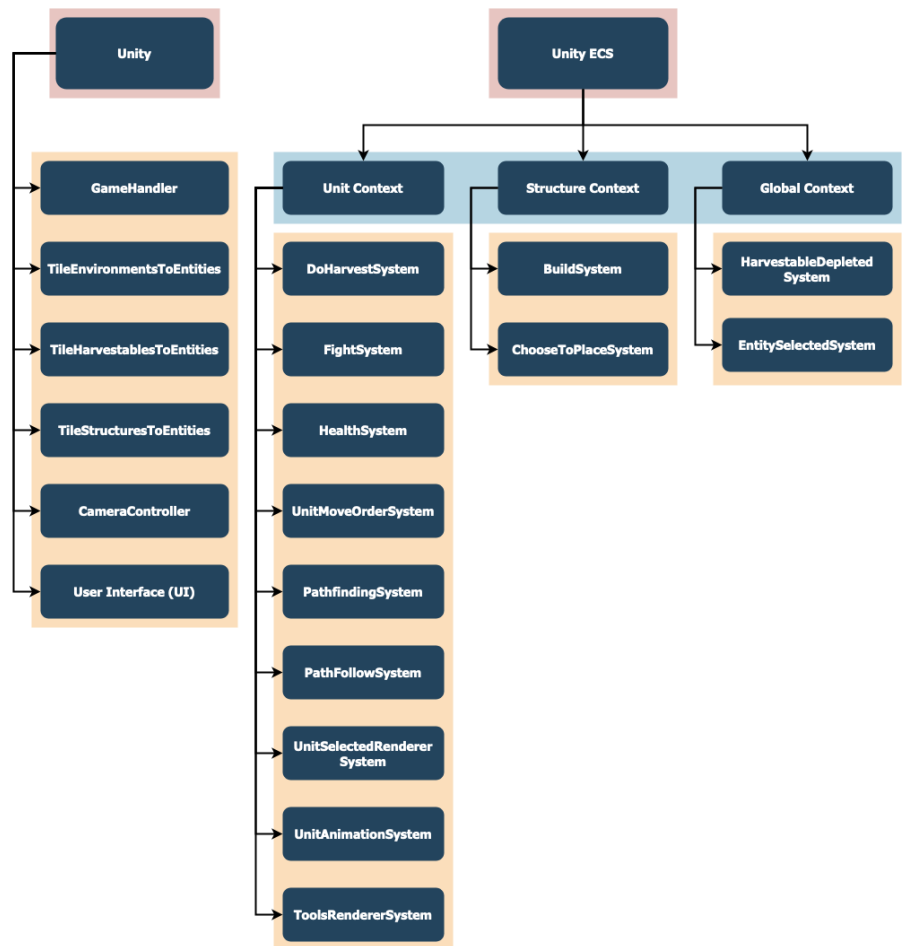


Figure 27: Overview about used conventional Unity scripts and ECS systems

ECS Scripts

Figure 27 shows you the mentioned scripts as well as all available ECS systems that are active in For The Win. As told earlier, these systems read entities and add/remove/modify their assigned components. Even though we separated them in context for better overview in figure 27, they are all equal in the ECS world. They operate as followed:

1. **DoHarvestSystem** : operates the harvesting action on unit entities. This includes: going to the selected harvestable, harvest it for a certain amount of time and returning to the headquarters.
 - ✓ **UnitComponent** : to get the right animation sprites
 - ✓ **DoHarvestComponent** : to store any harvesting information and progress
 - ✓ **AnimationComponent** : to set harvesting animation
2. **FightSystem** : Operates the fighting action between each unit. This includes, auto-detection which allows units to detect if an enemy unit comes nearby, and the real fight mode when two units are dealing themselves damages.
 - ✓ **Translation** : to get the current cell position
 - ✓ **UnitComponent** : to check the type of the unit (elf, knight, wizard, peasant)

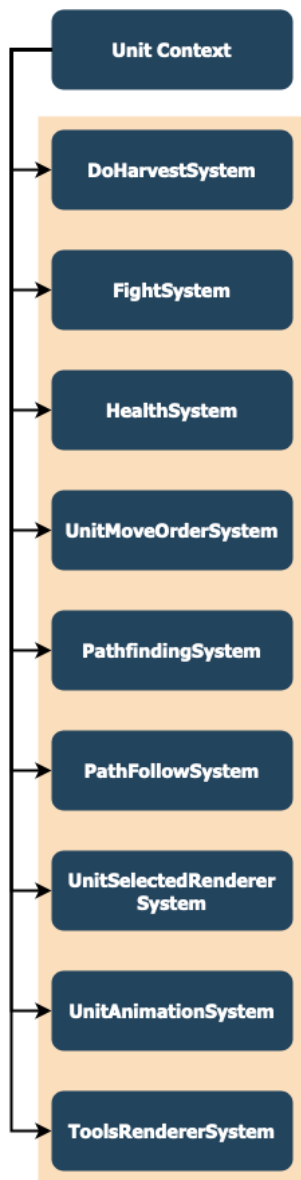


Figure 17: List of all used conventional scripts

- ✓ **FightComponent** : to check if the Boolean variable "is-Fighting" is true or false. And to assign an enemy to each fighting units
 - ✓ **TeamComponent** : to get the team number of the unit
 - ✓ **AnimationComponent** : to set fighting animation
 - ✓ **HealthComponent** : to get the current health points of the unit and to update it, depending of the fight going on
 - ✓ **BlockableEntityComponent** :
3. **HealthSystem** : assign to every unit a health bar and operates modification on the mesh depending on the health points of the unit.
 - ✓ **Translation** : to get the current cell position
 - ✓ **RenderMesh** : to set the right vertices, depending on the health points of the unit
 - ✓ **HealthComponent** : to get the the current health points of the unit
 - ✓ **HealthBarComponent** : to assign a unit for each health bar
 4. **UnitMoveOrderSystem** : Operates every move order of the game. This includes selection of units, move order to go to a different cell, move order to attack an enemy and move order to harvest resources. It also includes that unit stay in formation whenever they move or they fight.
 - ✓ **Translation** : to get the current cell position
 - ✓ **UnitComponent** : to get the type of the unit, depending on it, the formation will be different (knights go closer to enemy than elf)
 - ✓ **StructureComponent** : to check if a cell if free or not
 - ✓ **TeamComponent** : to get the team number of the unit
 - ✓ **HarvestableComponent** : to detect if the unit must start harvesting
 - ✓ **HealthBarComponent** : to check if a unit is a health bar or not, because then it's cell is still available
 - ✓ **BlockableEntityComponent** : to check if a cell is free or not
 - ✓ **EntitySelectedComponent** : to get the selected units, waiting for orders
 5. **PathfindingSystem** : builds up a pathfinding grid and calculates a path from source to target via A* search algorithm⁷.
 - ✓ **Translation** : to get the position of all blocked entities
 - ✓ **PathfindingParamsComponent** : to store the way start and end point
 - ✓ **BlockableEntityComponent** : to check if a entity is blockable
 6. **PathFollowSystem** : moves an entity from one position to the other.

⁷https://en.wikipedia.org/wiki/A*_search_algorithm

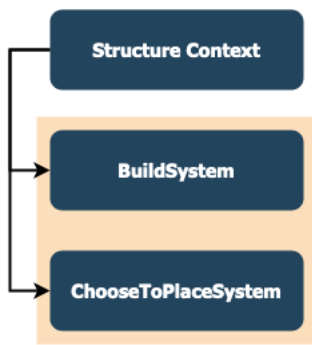


Figure 18: List of all used conventional scripts

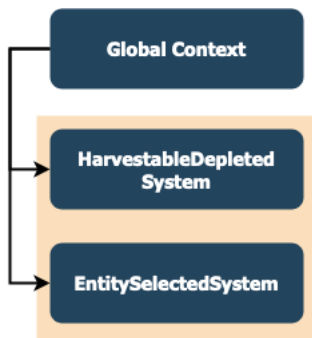


Figure 19: List of all used conventional scripts

- ✓ **Translation** : to get the current position of the entity
 - ✓ **UnitComponent** : to get the unit type
 - ✓ **AnimationComponent** : to set walking animation
7. **UnitSelectedRendererSystem** : displays a circle mesh below a selected entity.
 - ✓ **Translation** : to get the entity position
 - ✓ **TeamComponent** : to know the color that the system has to display
 - ✓ **EntitySelectedComponent** : to get the selected entities
 8. **UnitAnimationSystem** : handles the sprite changes of one entity and therefore animates it.
 - ✓ **RenderMesh** : to display the new sprite
 - ✓ **UnitComponent** : to get the unit type
 - ✓ **AnimationComponent** : to set animation information
 9. **ToolsRendererSystem** : changes the mouse cursor to another shape when attacking or harvesting other entities.
 - ✓ **Translate** : to get the entity position
 - ✓ **UnitComponent** : to get the unit type
 - ✓ **TeamComponent** : to check the team side
 - ✓ **HarvestableComponent** : to check if the entity is harvestable
 - ✓ **EntitySelectedComponent** : to get the selected entity
 10. **BuildSystem** : spawns a new unit or structure entity when a context menu slot was clicked. Finds a free neighbor cell for units. Creates a new entity with a ChooseToPlaceComponent for structures.
 - ✓ **Translate** : to get the entity position
 - ✓ **StructureComponent** : to get the building type
 - ✓ **SlotClickedEvent** : to gather the event
 - ✓ **EntitySelectedComponent** : to get the selected entity
 11. **ChooseToPlaceSystem** : handles and displays the placement of new buildings.
 - ✓ **Translation** : to get the current cell position
 - ✓ **ChooseToPlaceComponent** : to get the building wish and the wanted building type
 12. **HarvestableDepletedSystem** : removes any harvestable entity from the world, that is depleted.
 - ✓ **HarvestableComponent** : to check if the harvestable is depleted
 13. **EntitySelectedSystem** : as soon as an entity is selected, this system tells the UI context menu what to display.
 - ✓ **UnitComponent** : to check if the entity is a unit
 - ✓ **StructureComponent** : to check if the entity is a building

- ✓ **EntitySelectedComponent** : to filter all entities that are currently selected

Interesting parts

- ✓ **Animations** : Thanks to this project, we learnt a lot about animations, mesh and materials. Before, we would always use the Unity animator to display some animations and the Inspector to create default materials and meshes. But through this project, we created animations, mesh and materials only based on code lines. It was very rewarding and this new knowledge will be useful for our upcoming projects.
- ✓ **ECS** : Through this project, we realized that ECS was really worth learning it. Because once you created all your **Entities**, **Components** and your **Systems**, coding becomes very flexible. You can easily write extensions to your entities and make them interact with each other. In our opinion, it will be massively used by developers in the future.

Encountered Problems and Solutions :

- ✓ **Team Work** : As we were two, we needed a way to modify the code and to see the modifications from our partner at any time. We used **GitHub** to update our modifications in real time. But even with GitHub, we faced an issues. If we code at the same time on the same part of the game, it could lead to many merge conflicts. And it takes some time to manually merge the changes. To avoid this issue we chose to never work at the same time on the same part of the project.
- ✓ **ECS** : We chose to work with ECS, which was completely new to us. So we had to spend a lot of time watching some tutorials about this new way of coding in order to understand it.
- ✓ **Pathfinding** : Pathfinding is essential for a RTS game but we started implementing the movement before thinking about it. Because of ECS, entities had no information about the underlying tilemap. No conventional 2D colliders were usable. So we had to screw our movement system and build another one based on an own implemented grid-based pathfinding algorithm (see PathfindingSystem).
- ✓ **Health bar** : Health bars are part of every RTS game but we had some troubles to implement them. We first wanted to create a "prefab Game Object", that would be instantiated each time a unit entity was created and affiliated to the unit's Health Component. But it appeared that **ECS Systems** are compiled before every traditional **MonoBehaviour** script. And as the reference to the prefab was stored inside the MonoBehaviour script, and the prefab was affiliated to the unit's component inside a System script, we were always facing a "NullReference error". To solve this issue, we decided to create health bars as entities instead of Game Object. These entities would have a "HealthBarComponent" in which a reference to the belonging unit entity was stored.
- ✓ **Overlapping units** : We often faced an issue of overlapping units. For example, when units are harvesting resources, they

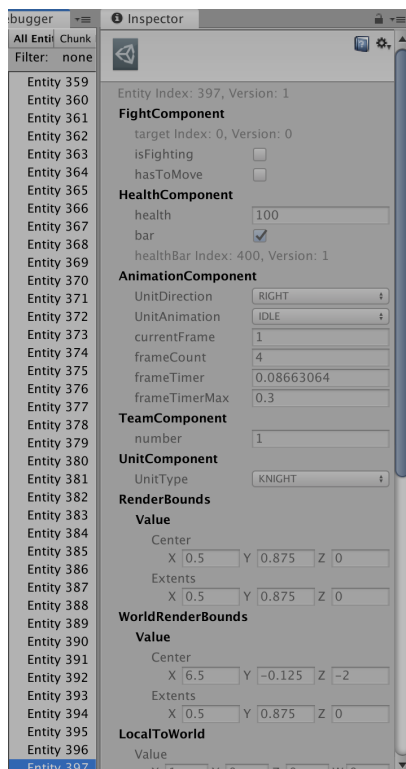


Figure 20: The Unity ECS Debugger with a selected entity and its corresponding components



Figure 21: Improved display of the amount of resources

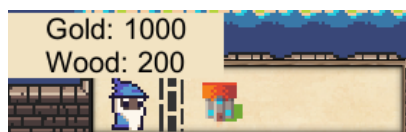


Figure 22: The new hover effect when you build structures

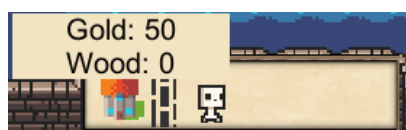


Figure 23: The new hover effect when you build units

can overlap when they come back to the castle to drop the resources. Also with the enemy detection, enemies might overlap when they detect several units coming to them. The reason of these bugs is that we check if a cell is free only once, just before giving the move order to the entity. But in the meantime, other entities might occupy this cell and thus it is not free anymore. To fix this bug, we use a Boolean variable inside the FightComponent called "hasToMove". It was originally created for the enemy detection. Now each unit checks if it's overlapping and if they do, they plot a new way to a free nearby cell.

TESTING

For our project we decided to apply manual testing in combination with the provided ECS Entity Debugger. We placed **Debug.Log** commands in crucial parts of the program to verify the outputs. After each major implementation, like the path finding or the fight system, we manually tried many situations to see if everything was working properly. We used artificially created tricky situations to see if the game was working as it should.

As you can see in figure 20, the **ECS Entity Debugger** provides you with updated runtime information about the entities and their linked component data. When creating new systems, this is a useful tool to see an entity's state. When you pause a Unity execution, all entities are frozen and displayed like normal Game Objects. So you can use Unity for debugging as usual.

PEER REVIEWS

We received some reviews with small improvement suggestions and want to share those:

- ✓ **Resource amount** : According to the peer reviews of **Demo 2**, it was a bit difficult to see the amount of resources possessed. So we changed the UI to make the amount of resources more visible. See the result in figure 21.
- ✓ **Building costs** : During **Demo 2**, it was not possible to see the number of resources required to build a building or a unit. Multiple peer reviews mentioned it. So we decided to implement it. Now the cost of every building and unit is displayed when the mouse hovers an icon on the context menu at the bottom. See the result in figure 22 and 23.
- ✓ **Harvest cycle** : Several reviews in **Demo 2** suggested, that we should add a harvest cycle to the workers. They should continue harvesting after they returned to the HQ. We have added this idea to the Future-Work section of this documentation.

SHORTCOMINGS

- ✓ **Save and load system** : During the last week of the project, we tried to implement a save and load system. But once again, with ECS we are dealing with something completely new and there's no standardized way to do this yet. We would have to write an own serializer and define a customized save format. We realized that we could not reach this goal in time.
- ✓ **Buildings** : From a strategic point of view, it is very important that units can inflict damages to buildings. Units should be able to destroy buildings, and even players should be able to destroy their own buildings. But unfortunately we did not have enough time to implement this system.
- ✓ **Multiplayer** : From the beginning, we knew that there was a high probability that we would not have enough time to implement the multiplayer system. But we agreed that it was more important to make the gameplay work properly even if there is no multiplayer system. The multiplayer system could be added afterward.
- ✓ **Farm**: Currently, it is possible to build farms but their function is not implemented yet. The idea is to limit the number of units you can create according to the number of farms you have. Because farms produce food and your units need food to survive.

FUTURE-WORK

- ✓ **Enemy waves system** : As we do not have a multiplayer system, we thought that it could be interesting to implement a waves system which spawns some enemies and the player has to protect his wizard. Basically it is a mix between a tower defense and a RTS game. We believe that it would be quicker to implement this feature than multiplayer.
- ✓ **Gathering resource system** : Currently, when we farm resources, we need to order the unit to keep harvesting each time it comes back to the castle. A unit should keep farming until it is told to do something different or until there are no more resources left.
- ✓ **Sound effects** : During the whole project, we had no time to care about the sound. But now that units can fight properly, it would be very great to add some sound effects. And maybe a background melody.
- ✓ **Extend the map** : The current map is a fixed standard map that we used to develop the game and to run some tests. So the current map is not very interesting, it does not have some strategic positions. We would like to create a much more interesting map, with more rivers, some bridges and resource points. Maybe even some kind of mountains from where the elves would see their range increase. This way the game would become much more interesting and players could develop many strategies.

- ✓ **Fog of war** : At the moment you can look over the whole map. For future iterations it would be nice to see only around your units what's happening on the map.
- ✓ **Walls** : As any great castle or fortress is surrounded by many walls to protect it, we really want to implement a system that allows player to build walls with gates. We already have the wall tiles so we just need to implement the system. With this new system the game will become even more interesting.
- ✓ **Multiplayer** : Implementing a multiplayer system is the final purpose of the project. Because without it, players can only play against the computer.
- ✓ **Walls** : As any great castle or fortress is surrounded by many walls to protect it, we really want to implement a system that allows player to build walls with gates. We already have the wall tiles so we just need to implement the system. With this new system the game will become even more interesting.