# Introduction

I have a big plan for my game, bigger than what is achievable during this module. My approach was to plan for big but advance gradually. I started with a simpler graphic to make my task easier and to focus on the functionality. My first task was to make my character move between walls, and then I added a new gaming element or concept on every level. Sometimes, I had to refactor previously written code, as the new element required a different approach, or made the previous implementation faulty.

With this approach, I managed to add all the elements I planned for the first stage of my project. The doors, the switches, guards, the alert system, path-finding for the guards and the keys. It still requires some refactoring and optimisation before further development of new features can start, but it is ready to create more levels that combine the existing objects in creative ways, resulting in harder and harder levels.

My game in its current state has to be seen as a demo that demonstrates what elements and game mechanics one can expect in the final product. New levels have to be created, the graphics have to be changed to something more eye-catching, but the game mechanics for the first part are finalized.

# Demonstration of the project and its development

The most important change I made from the project plan is switching the game engine with Godot. Previously, I was planning to use Orx, which is written in C++ and can process script resource changes during runtime, similar to Godot. It may be more efficient than Godot on the long run, but it proved to have too much of a learning curve. Although its community answers your questions in hours, sometimes in minutes, it does not have so thorough documentation and tutorials as Godot. I found that to be able to use Orx successfully I had to be familiar with game making concepts already, narrowing the learning down only to the working and syntax of the engine. It is more suitable for already experienced game developers. On the other hand, Godot offered ready-made solutions most of the time that only required customisation for my game, or explained game making concepts and designs to beginners in the context of the game engine.

## Design of the code

This changing of the game engine brought some alteration to my original class design. Godot does not have an Entity Component System (ECS). It supports composition over inheritance but its node-system relies on single inheritance too. Composition can be achieved by adding child nodes to other nodes, or by assigning them to groups. The second method works as a tagging system, where a node can be identified with group names. On the other hand, the first method can be used to add additional behaviours to other nodes.

In my project, I only used the tagging system so far. At multiple places in my code, I collect all the nodes from the scene tree that belongs to a certain group to call functions on them. For example, in Global.gd in the \_on\_door\_state\_changed(id, is\_open) function I collect all the doors by calling get\_tree().get\_nodes\_in\_group("doors") to find the one which state have been changed. These group tags replaced most of my interfaces on my class diagrams, like ItemStorer, Carriable or Disconnectible.

The many collision layer Godot offers made my See and Detectable interfaces unnecessary. I just added a designated collision layer (#10) to the nodes that were looked for (like the player) or to the ones that fully obscured view (like walls), then added the corresponding collision mask to the guards who watch out for the player and can detect them.

Godot engine uses several design patterns either in syntax or under the hood. The observer pattern is implemented through signals in the engine and I used it for event-driven programming. One example is when the player interacts with a switch, the interact\_with() function is called in Switch.gd. This emits a signal for all its observers (the game map) that an array of doors needs to change their state. Generally, signals are used to communicate upwards in the node tree while parents subscribe to their children’s signals and control their children by calling functions on them. In the previous example, it is the parent’s responsibility to change its children affected by the switch when receiving the signal. Unfortunately, this behaviour is not included in my class diagram. I did not know I would use them at the time of planning.

Another design pattern implemented by the Godot engine is singleton and it is called AutoLoad. Any node tagged for AutoLoad in the project will act as a singleton. It will load into memory at runtime and is accessible from any nodes in the code. You do not need to instantiate them, just calling their functions or accessing their fields. I used it for my P\_Player.gd and Global.gd scripts. P\_Player is the player node that I move from map to map when the scene changes, preserving its state every time. Global.gd contains functions that have to be reached from multiple nodes and it also handles the scene changes.

Right now, Global.gd is still small, but I can see the signs of it becoming too big and having too much responsibility. I will need to move the disconnectibles and entrances data variables to a resource file and loading them into memory when needed. The path-finding algorithm and map changes are both Global.gd’s responsibilities, which are completely different from each other. These are needed to move into other resource or node files. Global.gd will probably become a façade in front of a bunch of back-end classes.

The third design pattern implemented by Godot is the game loop. Interestingly, Godot has 2 parallel game loops, one for the physics engine and one for general use. It does not offer an update function for nodes instead, every node has a \_process(delta) and a \_physics\_process(delta) functions. The order of calls between nodes is determined by their position in the node tree, although it can be customised if necessary. Both game loop can have different FPS. I used the physics engine to restrict the player’s movements, to move the guards and the player by applying velocity to them, and to detect overlapping areas. Objects can be interacted with only if a character’s and the object’s areas are overlapping. These are detected either by signals, like when the player reaches an entrance that trigger scene changes, or by directly checking them, like in P\_Player.gd in \_physics\_process() where the program checks upon pressing the ‘action’ key if there is an overlapping interactive object’s area.

As I mentioned before, Godot support composition over inheritance. One area that could benefit from composition is my game objects, like Switch, Door, NoiseMaker and Key. They are all objects the player or the guards can interact with, but in the future, I expect new, not interactive objects, or new behaviours that will be applied to only some of the existing objects. I am planning a refactoring where I move this interactivity to a new node and instantiate it in all the interactive objects. The communication will be more complex between the nodes, it will require newly implemented signals for communication, but it will make object behaviours more reusable and variable.

lastly, I used the state design pattern for my guards. Currently, they can be idle, alert or patrolling. In the future, I am planning to refactor Guard.gd and move its state into a new Finite State Machine implementation. It will be implemented in a pack of nodes that can be children of Guard. I hope that this will make my code easier to expand in the future and making the Finite State Machine reusable for new types of characters.

## AI algorithms

My guards can have three states: idle, patrolling and alert. Idle means they are doing nothing, and patrolling means they are following a predesigned path. The transition between the states and their behaviours are implemented in the various methods of Guard.gd. Some of the changes happen internally, while others triggered from the outside, like the alert state in the make\_alert() method.

I implemented one complex algorithm for my guards. It calculates the shortest path on the map for them, including paths that require interaction with objects, e.g. opening doors and using switches. It is located in Global.gd. My navigation map is initialised in init\_navigation(), and this complex path-finding algorithm is implemented in find\_interactive\_path() and \_recursive\_find(). This algorithm builds upon an already existing pathfinding algorithm implemented in Godot, the Astar2D. I used this for finding simple paths on my map that does not require any interactivity.

I expect a later refactoring for this method, as adding new types of interactive objects require the modification of this function. That is not ideal; maybe the visitor design pattern will be useful here.

## Testing

During this assignment, my goal was to implement as many features of my game as possible. That is why the graphics are simple and there is not any sound added to the project yet. My testing strategy reflects this approach. It aimed for finding out if the game mechanics are working as intended or if they feel natural.

The main tester was I, but I also included others when I deemed a new feature ready. I focused on making my game as bug-free as possible, while I asked others to focus on how natural the control and the guards’ behaviour felt. I also asked them to try to break the game if they can.

One change I had to make is to make the player’s collision shape smaller. It was hard to go through doors for one of the testers, especially when they had to hurry. For other suggestions, I did not have enough time to implement them or I deemed them less important and gave them less priority in the list of features. For example, one of my testers reported that opening a door near a guard should alert them. It is a reasonable suggestion but I see it more as a fine-tune rather than an important behaviour at this early stage of my game.

The movement of the guards is also reported to have rapid changes. Direction change should be slower, interacting with objects should take time, and after arriving at a noise’s origin, the guard should look around before heading back to its station or patrol route. I think these are small, but significant improvements I am planning to make in the future.

Some testers found the lack of ability to quit the game also frustrating. A game menu will be developed in the future, but in the meantime, I changed the game to run in a window instead of a full screen.

There are also some bugs still present in the game.

* Guards can be stuck at their position if they are alerted for a second time while they are moving.
* Moving the player should be more responsive. The last pressed direction should be used, or the player should move diagonally too.
* Levels do not reset properly after a guard notices the player, e.g. newly opened doors remain open.

The last bug will be fixed when a saving mechanism is implemented. I am planning to save the map’s and the player’s state after going to a new map, and reload their saved state from the disk every time the player is noticed. This will roll back every change that had happened before the player was caught.

## Ethics

I plan to make my game available for children and adults up from high school age. The current textures the game uses are not the final ones, but I do not intend to use much violence, blood or sexual content. The story is not developed yet, so it may be appropriate for even younger children. If it has themes not suitable for primary school children, I will stick to my original audience.

## Security

The game is exported into a .exe and a .pck file. A file with a .pck extension cannot be unpacked by common software applications, although it is possible. I asked the texture creator if packing it into a .pck file with Godot will fulfil the requirement to not distribute the assets, and they answered that *“if a third party cannot use the asset directly, it is not considered redistribution.”* I considered guarding my assets further, but everything can be cracked and it does not worth the time.

I plan to encrypt my source code with an SHA3 key and to sign it digitally before release. Unfortunately, encryption caused my exe file to crash before the game starts, and I have not yet found the reason for it. For this assignment, I submit an exe file where my source code is not yet protected.

Another question to consider for the final product is whether to use a Digital Rights Management (DRM) tool or not. I am of the opinion, that this is only useful for bigger development studios, as effective DRM technologies are expensive, while cheaper or less effective solutions can cause troubles for paying customers and pirates can crack to code quickly.

# Review and conclusion

If I started my game again I would not plan that much at the beginning. I found that without practical experience in game programming I could not make a detailed plan. I think the agile approach, where I do not plan far ahead in more than broad details, where I refactor my codebase often, and where I learn the how on the run is more suitable for me. If I had started writing my game sooner I could have made a more refined game with more of my planned features.

My other mistake was that I wanted to learn too much at the same time. I planned to learn C++ while writing a game. I did not know that game programming required such a different approach from the projects I had done so far, hence my adamant attachment to Orx, a C++ game engine. Switching the game engine lost me valuable time.

Apart from these mistakes, I learned much from this project. I have never worked with graphics, and learning how graphical elements can interact with each other was a nice experience. I am planning to finish my game with all the planned features, with nicer textures and added audio.