Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Invisibee: Multiplayer Game with Secret Strategies on a Shared Screen

Lab Report

Linus Kortesalmi
kolinus@student.ethz.ch

Nico Kurmann
kurmannn@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Michael König, Manuel Eichelberger
Prof. Dr. Roger Wattenhofer

August 8, 2017

Acknowledgements

We thank Manuel Eichelberger and Michael König for their continuous feedback while designing the game, as well as our partners and families for their unwavering support and help testing the game. Also we thank TIK for supplying controllers.

Abstract

This work describes *Invisibee*, a multiplayer *Real-Time Strategy (RTS)* game. Unlike other RTS games, it is played on a shared screen by making use of hidden inputs via gamepads. This allows the players to create units, construct buildings, and move armies in secret; the locations and details of which are only revealed when units enter an enemy's sight range.

Contents

Glossary

CP: Control Point - A position on the map that can be claimed by units and upgraded for superior defence and resource production.

FoW: Fog of War - Visual layer ensuring that only the part of the map which units can see is revealed to players.

FPS: Frames Per Second - The number of frames that are calculated and rendered per second. 60 FPS are typically seen as a minimum benchmark for modern games.

HP: Hit Points - Represents a unit's or building's health. When this reaches zero, it dies.

NPC: Non-player character - A character not controlled by any player(s). Instead it is controlled by the computer via predetermined or responsive behaviour.

RTS: Real-time strategy - A game genre typically centred around collecting resources and building armies to deny opponents their resources.

UI: User Interface - The domain where a player interacts with the game. The UI usually contain information relevant to the player, such as HP.

Chapter 1

Introduction

Many multiplayer games like Jass[1] or Stratego[2] rely on hidden information to introduce some form of asymmetry of information. This makes the games more interesting and challenging, since many possible scenarios have to be taken into account as well as their respective likelihoods.

Consequently, to not divulge too much information as to one's own strategy, it is often ideal to randomly select one of several strategies. Furthermore, some games allow players to perform actions that will only be revealed at a later point in time[3], making these strategic choices have more impact and requiring players to anticipate each other's moves.

Normal shared-screen games have the inherent property of displaying the same information to all players. The goal of this work was to explore the possibilities of hidden information in shared-screen games. The product is *Invisibee*, a *real-time strategy (RTS)* game that selectively hides information about the game state.

1.1 Related Work

Inspiration for this project comes from a video game called *Hidden in Plain Sight*[4], which consists of a series of mini-games, all revolving around the theme of hidden identities. Each player secretly controls a character on the screen while a large number of computer-controlled characters (NPCs) act as decoys. Since players do not inherently know which character is theirs, they have to find out by observing which one responds to their controller input. Most games can be won in one of two fashions: either by being the first to achieve an objective, or by eliminating all other human players. This results in players trying to perform the task while not drawing any attention, or risk exposing their identity, which usually leads the character's rapid demise.

[1] Jass, card game: https://en.wikipedia.org/wiki/Jass

[2] Stratego, strategy board game for two: http://www.stratego.com/en/

[3] A Game of Thrones: The Board Game: https://www.fantasyflightgames.com/en/products/a-game-of-thrones-the-board-game-second-edition/

[4] http://store.steampowered.com/app/303590/Hidden_in_Plain_Sight/

The concept of using decoys to "hide in plain sight" seems to have been extensively explored by the game, which raises the question of what other ways there are to incorporate hidden information in a shared screen experience.

We decided to design and implement a shared-screen game that makes use of hidden information in a different way, namely hidden input.

Chapter 2

Game Mechanics

2.1 Core Gameplay

Invisibee is played on a flat playing field featuring impassable walls and unclaimed *Control Points (CP)*, see Figure 2.1. Every player starts off with a base and the goal of the game is to destroy the opponent's base.

The bases generate a small amount of resources, thematically called honey. Players can spend honey to create units at their base. These can be moved around and will attack on sight any enemy units and buildings in range. Apart from fighting, these units are also capable of claiming unclaimed CPs.

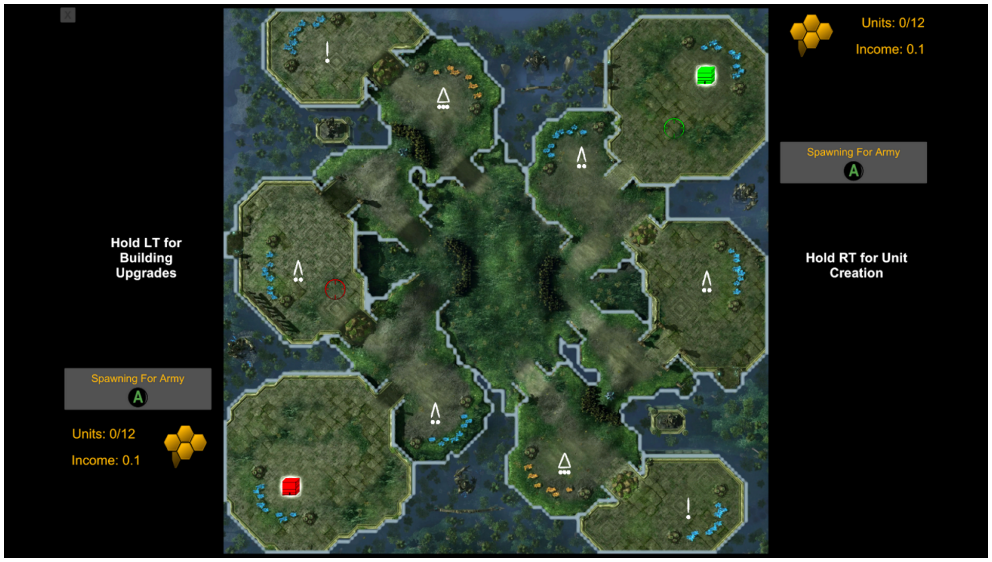

Figure 2.1: Screenshot of the map with the game's UI. The white highlighted edges are impassable walls and the white objects with 1-3 circles under them are CPs. The four golden hexagons represent the current amount of honey. The red and green buildings are bases.

A further use for honey is to upgrade a claimed CP. Before a claimed CP can

be captured by an enemy, all of its upgrades must first be destroyed. They also provide additional benefits depending on the type of upgrade:

Hive: Produces additional honey.

Freeze: Damages one enemy[1].

Heal: Heals several allies, produces a tiny amount of additional honey, and grants armour.

All units and buildings have a certain number of *hit points (HP)*. When one takes damage, its HP are reduced. When its HP reach zero, it is destroyed. Only the upgrade built last takes damage, and is thus the first one being destroyed. Units can be healed by standing close to a Heal upgrade, while upgrades can be repaired by issuing a repair command.

2.2 Unit Counterplay

The game has three different unit types:

Ladybug: Fast, short-range unit that tackle enemies to inflict damage.

Bee: Mid-range unit that shoots slow projectiles which follow enemies. The damage is dealt when the projectile reaches its target.

Badger: Expensive, long-range, high damage unit. Badgers shoot projectiles similar to Bees.

Different unit types allow for interesting counterplay: While Ladybugs have the highest damage output compared to their cost, they are efficiently dealt with by the longer-ranged Bees. Likewise, Bees trying to defeat a Badger with its significantly larger pool of HP will be at a disadvantage since the number of Bees will dwindle over the course of the fight, while the Badger's damage output will not decrease as it takes damage. Lastly, in an even fight between Ladybugs and a Badger, the Ladybugs will have the upper hand due to their large damage output, while the Badger's attacks will overkill Ladybugs in a single blow, thus wasting damage output.

2.3 Armies and Movement

Both due to the constraints of controller input and the lack of visual feedback when playing with *Fog of War (FoW)* (see Section 2.4), we chose to issue commands to groups of units. When a unit is created, it is assigned to one of four

[1] Up to 3 in "extreme" settings preset

available armies (A, B, X, Y). The army that new units are assigned to can be chosen via pressing a button combination.

Each player controls a visible cursor that can be moved around freely. By pressing one of the army buttons (A, B, X, Y), all units of the corresponding army are ordered to pathfind to the location the cursor had when the button was pressed. Pathfinding means that they will navigate through the world and avoid any obstacles or walls until the target location is reached. If they encounter an enemy along the way, they will engage automatically. In order to allow disengaging from a fight, units can be forced to ignore enemies by keeping their army button (A, B, X, Y) pressed down. Even though the opponent can see the player's cursor, he or she does not know which or when an army button was pressed.

A third way to move units is to keep the army's button pressed down and to simultaneously move the cursor's joystick in a certain direction. This will force all affected units to stop pathfinding and instead try to move in the joystick's direction. This allows very direct control and allows units to be moved in parallel to each other. To ensure that reinforcements from the base find their way to the front, units that have not yet joined up with the bulk of the army will ignore this command.

2.4 Hidden Information

In order to hide the player's moves from the opponent, all units and upgrades are hidden. This is represented by a "selectively transparent", dark layer covering the map and hiding all units and buildings beneath it. This is called the *Fog of War (FoW)*. When a unit or CP spots an enemy within its sight range, a small area around the enemy is revealed, temporarily lifting the FoW. The lifted FoW is added back after a few seconds of play time, unless the unit gets revealed repeatedly. The design decision of only temporarily lifting the FoW allows the player to react to being spotted by the opponent. Players can try and disengage from the combat and yet again hide their units in the FoW, or they can use this opportunity to flank with another army of units already hidden in the FoW. The FoW's main function is thus to hide the unit positions. However, it cannot effectively hide information concerning what types of units are produced (army composition) and what upgrades are built. To hide these, units and upgrades to be built are selected via hidden controller inputs. Figure 2.2 shows the two visual indicators for producing upgrades and units, respectively.

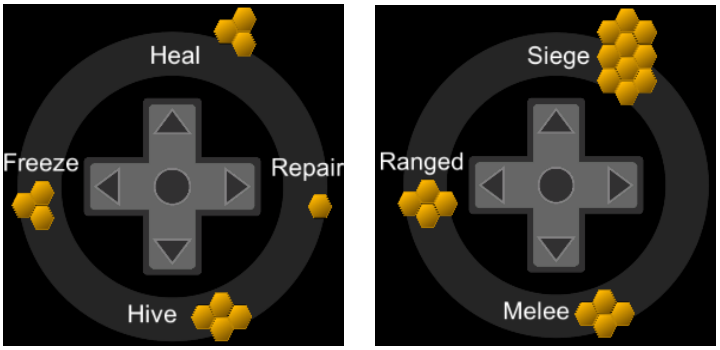

Figure 2.2: Visual indicators for selecting what upgrade or unit to build. The opponent is unable to determine which direction (Up, Left, Down, Right) was pressed.

Chapter 3

Implementation

In this chapter, some of the most interesting parts with regards to implementation are presented and motivated.

3.1 Unity3D

We chose to implement the game using the Unity3D[1] game engine, as it provides useful functions such as:

- Simple physics simulation.
- Gamepad support.
- Prebuilt UI components.
- Automatic garbage collection.
- Well-documented API.

Furthermore, it can be programmed using a high-level language (C#) and offers cross-platform support for Mac OSX, Linux (Beta), and Windows.

Unity3D allows a simple creation of scenes, which can then be gradually populated with game objects. Scripts and components can then be attached to the game objects to define their behaviour. This proved to be useful for rapid prototyping.

A scene can be seen as an isolated container for objects in the game world. Changing scenes removes all instantiated game objects, populating the scene with only the predefined objects for that particular scene. Objects can be configured to carry over between scenes, which is useful for objects containing general player state for the game, such as lives in a level-based game. The end result consists of three scenes: one for a 2-player game, a title scene (main menu), and a configuration scene.

[1] https://unity3d.com/

3.2 Firing

Figure 3.1 shows the finite state machine governing the firing of units. A unit enters the state machine in the *Idle / Waiting For Target (Idle)* state, which it remains in until a *legal* target has been acquired. A legal target is any unit belonging to the other player that is in firing range and is currently alive.

Before a unit can fire however, it has to stand still for a certain amount of time, which takes place in the *Take Aim (Aim)* state. If the unit is moved by a player during this state, it will return to the Idle state and the Aim timer is thus stopped. If the timer completes a final legal target check is performed. If the target is no longer legal the unit will check if there are any other legal targets. If no legal targets are found the unit will go back to the Idle state, skipping the *Cooldown (CD)* state. If a legal target is found, or if the original target is still legal, the unit will enter the *Fire* state and fire at the target. After firing it enters the CD state; it will only return to the Idle state once the CD timer has completed.

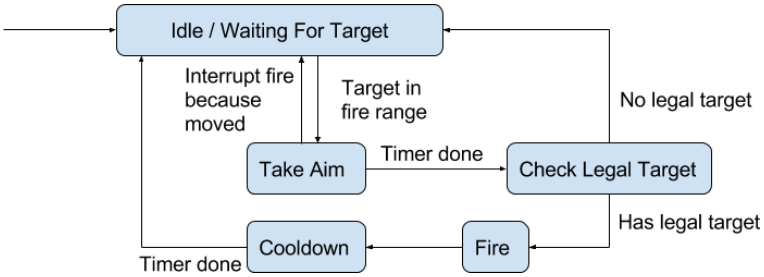

Figure 3.1: The finite state machine governing firing

The design of the firing state machine enables a tactic called “Stutter-Stepping”. When a ranged unit engages an enemy, it can take a shot, then move freely until its cooldown expires. With good timing, this allows units to be moved around without decreasing their damage output. This tactic can be used to get into an advantageous position, or to retreat from an enemy with a shorter range, forcing them to move closer.

3.3 Armour

Before damage is applied to a unit or building, it is reduced depending on the unit or building’s *armour* score. For every point of armour, it can survive additional damage equal to 10 percent of its base health, see Equation 3.1. The equation

was inspired by the armour system of the video game "Warframe"[2].

$$\text{Damage} = \frac{\text{Strength}}{(1 + \text{Armour}/10)} \tag{3.1}$$

This has several interesting effects. For one, since the Heal upgrade grants a CP armour, it makes more sense to build the Heal first, so that it is destroyed last when the tower takes damage. This allows other upgrades to benefit from the Heal's armour, making them sturdier.

Secondly, the badger is the only unit with armour. Since it has 10 armour as default, this effectively doubles its health pool. As a result, healing applied to a badger is twice as effective, as it undoes double the healed amount's worth of damage.

3.4 Pathfinding

Agent pathfinding is arguably a critical feature of any RTS game. Different variations of the popular A* search algorithm were applied and tested during the lifespan of our project. A* is an extended version of the famous algorithm by Dijkstra, using heuristics to speed up the performance.

A representation of the map is needed in order to use the A* algorithm. Our representation is covered in Section 3.4.1. Three variations of A* that we tried are highlighted in greater detail in the Sections 3.4.2, 3.4.3, and 3.4.4.

3.4.1 Grid

A simple and straightforward matrix representation that we call our *Grid* was chosen for our game map. In our case we have a map with a resolution of 2067×2067 pixels which is divided into 159 rows and columns, each cell being 13×13 pixels. Two cells are considered *adjacent* if they can be reached from one to another by increasing or decreasing the row- and/or column-index by 1, see Figure 3.2.

$C_{i-1,j-1}$	$C_{i-1,j}$	$C_{i-1,j+1}$
$C_{i,j-1}$	$C_{i,j}$	$C_{i,j+1}$
$C_{i+1,j-1}$	$C_{i+1,j}$	$C_{i+1,j+1}$

Figure 3.2: Cells that are adjacent to cell C with row- and column-index i and j, respectively. (Denoted $C_{i,j}$.)

[2] https://www.warframe.com/landing

Static Obstacles

Support for static obstacles like walls was needed in the Grid representation. This is done by simply setting a Boolean value *walkable* to be false if the cell is occupied by a wall or static obstacle. Adding information about static obstacles to the representation requires us to introduce the concept of *reachable adjacent* cells. Reachable means that a unit can move to the cell from the cell it currently resides in. A cell is reachable adjacent if it fulfils the adjacent definition (Figure 3.2), and the added constraint on reachability (Constraints 3.2 and 3.3).

Reachability constraint:

A walkable **non-diagonal** adjacent cell is always considered reachable. (3.2)

A walkable **diagonal** adjacent cell is reachable if and only if at least one of the two shared (non-diagonal) adjacent cells are walkable. (3.3)

See Figure 3.3 for an example of cells satisfying and violating the reachability constraint.

C_A	Wall	C_B
Wall	C	C_C
C_D	C_E	Wall

Figure 3.3: Cell C_A is adjacent but unreachable from cell C, due to the two shared adjacent cells being walls. Cells C_B and C_D are both reachable adjacent cells to C, satisfying Constraint 3.3. Cells C_C and C_E are also reachable adjacent, satisfying Constraint 3.2.

3.4.2 Simple A*

Our initial attempt at pathfinding used A* with Squared Euclidean Distance as our choice of heuristic function. This does not satisfy the triangle inequality (see Equation 3.4) but it does result in a performance gain by avoiding calculating the square root of the distance. This could lead to a suboptimal path being computed, which introduces a trade-off between optimality and speed.

$$(1+1)^2 \not\leq 1^2 + 1^2 \tag{3.4}$$

Running A* individually on multiple agents resulted in poor performance; the game was below 20 *Frames Per Second (FPS)* as soon as the unit count grew above 20 units. This forced us to think about alternative approaches that could lead to improvements. The considered approaches that we did not implement are mentioned in Section 4.4. The two that we implemented are explained in Sections 3.4.3 and 3.4.4.

3.4.3 Coroutine A*

The first approach we tried was utilising Unity's coroutines[3] to distribute the computational load over multiple frames. It was a straightforward fix to the existing implementation, with performance gains that were acceptable but not perfect; the FPS increased from roughly 20 frames to 50 frames with only 20 units in the game.

The motive behind this approach is the noticeable performance hit it took to calculate a full path in one single frame. If we could spread the computation, it would require less time per frame, making the game run smoother overall. If too many frames are allocated for the calculation, there would be a visible delay in the responsiveness of the pathfinding agent.

In our implementation we limited the number of cells to explore at each frame. When the limit was reached it would pause execution and let the frame end. During this frame the unit would stand still if no previous path had been calculated, otherwise it would continue following it. In our final implementation where we achieved 50 FPS, there was a noticeable delay where the units either stood completely still or continued following an old path going in the wrong direction from the new target.

3.4.4 Region A*

We moved away from the coroutine approach and decided to try using a high-level representation of the map and combine it with the Grid representation mentioned in Section 3.4.1. The high-level approach was inspired by the works of Alex J. Champandard [1].

Early-Stopping

Before we begin outlining the high-level approach it is important to discuss an approach called *Early-stopping*. The reasoning behind this approach is that only a part of the path is needed, since it is being updated frequently. The agent only utilises maybe the first ten cells of the path. Early-stopping would find a (potentially) great initial path, while keeping the remainder of the path unknown and uncalculated. There are no guarantees that the path found would be optimal or even near-optimal; the path could easily lead the unit into a dead end that it would never escape.

[3] https://docs.unity3d.com/Manual/Coroutines.html

High-Level Representation

A high-level representation of the map could be created automatically by grouping cells together into clusters. The criteria could be to only group reachable adjacent cells until the cluster was of a chosen size. A path is then be a series of clusters and there is be a trade-off between cluster size and speed. Many small clusters result in a large search space, while few large clusters result in a loss of precision; the goal is reached as soon as the unit enters the cluster and if more precision is needed a new search in the lower-level representation has to be executed.

We chose to exert more control over the process and instead created the regions manually by reading data from a map file.

The map file contains the Grid representation with information about whether each cell is walkable or not. We added so it also contains what region a cell belongs to. By doing so, we could control the regions while automatically creating an internal representation by parsing the map file. With regions, the pathfinding problem changes into finding a suitable sequence of regions to reach the goal's region from the start's.

Gateways

Gateways are cells that connect two regions in the higher-level representation. A region usually has more than one gateway. These gateways were marked manually in the map file.

For each pair of gateways of a region, the connecting path is precomputed and stored in a cache when the game is loaded.

We wanted to use these precomputed paths to quickly find the sequence of optimal regions to traverse. Our first step was to find the best gateway in the agent's region (start region). The best start gateway is the gateway with the lowest cost to reach the gateway of the goal's region (end region) with the smallest Squared Euclidean Distance to the goal cell. The cost is thus the sum of the actual costs of the precomputed paths between regions in order to reach the best gateway of the end region.

We used the simple A* algorithm and limited it to only explore cells in the start region. We call this approach *local A**.

In our first draft we also found the path from the best gateway to the goal by doing local A* in the goal's region. Later, we discarded this step when taking inspiration from the Early-stopping approach, as it was not needed to have the full path. We only need to know the path leading up to the best gateway, as the goal might change before the agent even reaches the gateway to the old goal's region.

3.4.5 Additional Problems

There is more to pathfinding than choosing the algorithm to use. This section will cover three additional problems solved during the lifespan of the project: path traversal, obstacle hugging, and dynamic obstacles.

Path Traversal

Retrieving a path of cells to follow from start to goal is only one part of the solution for agent navigation. Movement in Unity is typically done with their physics engine which computes the effects of forces applied to agents' bodies.

Despite the paths consisting of a number of discrete cells, it is important for agents move without any apparent stops. In the first draft of the path traversal solution, agents had to be in close proximity to the centre of a cell before they would start moving to the next cell on the path. We denote the process of clearing a cell and moving on to the next one as a *step*. The *step cell* is the cell currently being moved towards on the path.

We identified two particular scenarios that could lead to problems when reaching a step cell:

Colliding with other agents: When colliding with other agents it could happen that they were pushed past a cell without triggering it as a step. If this happened, the pushed agent would have to turn back and reach the step cell before they could continue on their path. This was mainly a problem if the following cells on the path were closer to the pushed agent than the step cell on the path.

Overshooting the step cell: Overshooting happens naturally as there is a trade-off between the distance of the proximity check and the tightness of following a path. If the distance is too large, the agent would too quickly step through the path, leading to problems when navigating around corners and obstacles. If the distance is too small, the agent would have to really hit the centre of the step cell before advancing on the path, which can be complicated by physics or low frame rates.

We decided to focus on solving the overshooting problem as it also partly solves the colliding problem. We did this by introducing a method to *prune* a step:

1. The prune method calculates the distance to the step cell and saves it in memory.
2. In the next iteration it compares the new distance to the step cell with the saved one calculated in the previous iteration.

3. If the new distance is bigger, we assume that the agent overshot and we trigger a step. Otherwise we save the new distance and go to the next iteration.

This only solved the colliding problem in the case where the agent was pushed past the step cell, making it being closer to the following cells on the path. In the case where it was pushed away from the step cell and thus the rest of the path, it would trigger a premature step. We tried solving the premature step by also checking the distance to the next cell as well as the current cell. If both distances are larger a step would not be triggered. This worked in the case where the unit is pushed away from the path. It causes problems however if the unit is pushed too far ahead on the path, so that the distance to the current and the next are both bigger, as it would not trigger a step; then the unit has to move back. This could potentially be fixed by also checking the distance to the third, the fourth, the fifth step cell and so on, and prune if any of those cells are closer. However, this approach does not address the underlying problem of the basic solution, as it does not handle the worst case scenario where the next step cells are unreachable from the agent's current position as a result of pruning. This regularly happened when moving around corners, with either approach.

We removed the premature step solution, as it added complexity to the code while not solving the real problem: the worst-case scenario. Instead, the worst-case scenario was solved without trying to implement an even more complex prune method but by simply running the pathfinding algorithm repeatedly at a fixed time interval. This primitive solution to the problem introduces a trade-off in performance and effectiveness. If the fixed time interval is too short it decreases the performance of the game, but makes stuck units become "unstuck" quicker. If the fixed time interval is too long the performance is better, but units can be stuck for a longer period of time.

Wall Hugging

When pathfinding around obstacles, the pathfinding algorithm chooses a path directly touching the obstacle, as this minimises the cost function of the search algorithm.

This caused especially larger agents to get visibly slowed down by colliding with corner obstacles. We called this behaviour *wall hugging*. An example of the wall hugging problem, and its solution, is explained and visualised in Figure 3.4. This problem is further exacerbated when moving as a group. It also causes problems with the prune method described before in Section 3.4.5, Paragraph *Path Traversal*, by frequently enabling the worst-case scenario.

We discouraged the wall hugging behaviour by changing from a uniformly weighted Grid, where cells could only be marked walkable or not, into a Grid where cells closer to static obstacles had an increased cost to them. The cost increased the

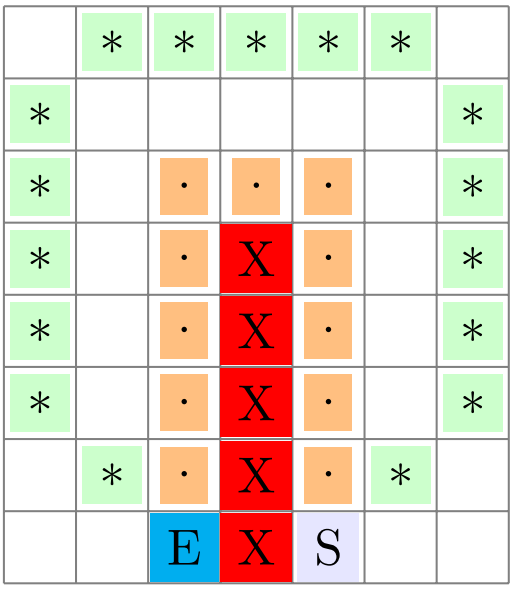

Figure 3.4: Wall hugging example: The start is the purple cell containing S; the goal is the blue cell containing E. The orange dot-path is wall hugging the red X-wall. The green star-path is using the non-uniformly weighted Grid and is thus longer than the dot-path. The cost for the dot-path is smaller than the cost for the star-path; however, the time it takes to traverse the star-path is shorter than the time to traverse the dot-path, as there are no collisions to slow down the unit.

closer the cells were, punishing agents from hugging the walls tightly. This means that it is still possible for an agent to pass through a narrow gap in the wall, if the cost of going around the wall is bigger.

Dynamic Obstacles

When agents engage in combat, it often results in front lines being formed. This behaviour leads to agents being stuck behind the front line, unable to partake. The desired behaviour would be for agents to expand the front line or create a firing arc around the enemy army.

To achieve this, agents have to plan their route when engaging enemies and take other agents into consideration. Our solution was to introduce the concept of *dynamic* obstacles, also called temporary obstacles. A unit is typically larger than a grid cell; so at every frame, every unit marks the centre grid cell it occupies as blocked (containing a dynamic obstacle) by adding it to a *centre blocked cell* list. It then also adds all of its adjacent cells to a *adjacent blocked cell* list. Both lists get cleared at the end of every frame.

The cells in this list have then a modified cost when using them for constructing

paths: centre blocked cells have a higher cost than adjacently blocked cells. This approach is similar to the wall hugging solution, as it discourages units to use centre and adjacently blocked cells for navigating while still allowing for traversal through these cells if the cost of going around them is too great.

3.5 Game Configuration

We tried to make the game as balanced and as fun as possible. Configuring a game's parameters in such a way that it is most fun to play, however is a lengthy process that even seasoned game developers struggle with. As people learn to play a game, their preferred strategies change and at a certain point it may be the case that certain strategies are too strong, leading to a less fun experience. That is why we created a simple way of adjusting the game's parameters.

All relevant parameters of units and upgrades can be changed via an in-game configuration scene (see Figure 3.5) accessible from the main menu. There are also fields allowing FoW to be turned off or on and alternative upgrade behaviour. Turning off FoW proved to be a great way to learn the game. It gives the players a good overview of what is happening when pressing what button, and it allows players to focus on learning the controls instead of trying to keep track of where their armies are positioned.

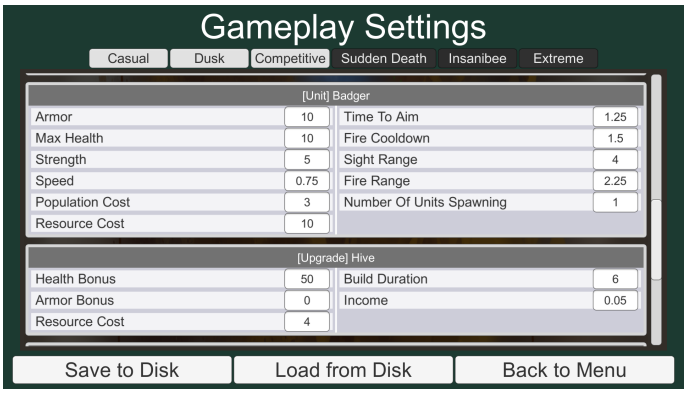

Figure 3.5: Configuration scene automatically populated from `GameRules.cs`

Additionally, several configuration presets are included. The three main presets offer varying mounts of hidden information, as new players may otherwise be confused by the lack of feedback the game provides. Three additional presets were included offering vastly different playing experiences.

In the interest of being able to perform adjustments easily, all relevant settings are stored in a single object called `Config.cs`. Whenever a script needs to look up a value, it does so there. In order to present the typical user with only the most relevant options, we created a new class called `GameRules.cs` containing

74 of the most important parameters from `Config.cs`.

Upon loading the game scene, the current instance of the game rules are applied to the configuration object. This allows users to modify the game rules beforehand or load one of the six preset rule configurations; some containing only a few changes to the 74 parameters present, while others change a lot more. Additionally, the modified game rules can be stored to disk or retrieved with a single press of a button.

3.6 Army Formations

We played around with different army formations during the lifespan of the game development. The current implementation uses no particular army formation, it simply lets the units group and form formations based on the result from Unity's physics engine. With the addition of dynamic obstacles mentioned in Section 3.4.5, Paragraph *Dynamic Obstacles*, units would no longer group up in a ball formation when fighting opponents. Instead, they would form firing arcs around the opponent, or even surround the opponent fully.

Before the addition of dynamic obstacles, we had two different formations the army could use: ball formation and line formation. The ball formation is the same approach we currently have, where we let Unity's physics engine take care of the grouping of units by applying a small force towards the centre of the formation. The line formation applies forces only in movement direction, causing units to gather up in a line perpendicular to the movement direction.

The concept of choosing between several different formations is a relic of the time when each player only had two armies, each controlled directly by a thumb stick. With the change to four armies, which can be ordered to pathfind to any place on the map, the concepts of group movement and a movement direction have become somewhat obsolete and less clearly defined. This lead to formations eventually being discarded. In a future version, one could return to the idea of army formations and see if it could be reworked and reintroduced in the current movement system.

Chapter 4

Discussion

As discussed in the motivation chapter, we set out to design a game that makes use of hidden input. We aimed to give the game strategic depth and tried to make it fair while remaining interesting.

4.1 Game Dynamics Supported by Hidden Input

The reason behind exploring the concept of hidden input is to enable game dynamics not previously possible on a shared screen. Game dynamics are the behaviours resulting from a set of game rules. In the case of Invisibee, we have the mechanic of a rock-paper-scissors game[1] between the three unit types.

In a setting without hidden information, this mechanic adds some depth to fights. Adding hidden input to hide the armies' unit compositions takes this dynamic to another level: players have to anticipate their opponent's strategy, both with regards to army composition and positioning.

Boom vs. Rush vs. Turtle[2] is another dynamic that is as old as the RTS genre. Players have the option to increase their resource income by building Hives. This is beneficial in the long run, however investing too many resources in the economy makes the player vulnerable to attacks until the investment has paid off. This opens up another rock-paper-scissors game of strategies (attack – defend – expand). This dynamic requires hidden information to exist. Our implementation does however give away certain key information, such as the number of CPs claimed and the number of upgrades built. Luckily, the strategies *expand* and *defend* can look very similar, because both rely on upgrading CPs, allowing this mechanic to still exist, albeit to a lesser extent. Despite limiting this game dynamic, we think that revealing this information leads to a better game as the visual feedback it provides is essential. It might be worth considering to only reveal this information when the player presses a certain button, but that would add additional complexity to an already elaborate control scheme. Fur-

[1] A mixed Nash Equilibrium with three strategies.

[2] Strategies centred around early expanding, early attacking, or defending early aggression: http://aoeo.heavengames.com/strategy/guides/rush-vs-turtle-vs-boom/

thermore, we think that some amount of visual feedback is required to keep the game appealing.

Another decision players are faced with is the choice between static defences and a mobile army to keep their base(s) safe. While static defences deal more damage, they are more predictable and are far more difficult to use offensively.

Lastly, there is the trade-off between the number of armies a player uses. While there is no direct downside to using all four army slots available, managing all of them is challenging and stressful. In our experience it is often more advantageous to limit oneself to using only two armies, as using more can become a significant burden, especially for novice players.

4.2 Cognitive Load

There are several features of the game that require a great deal of attention by players, as they need to keep track of:

- The number and the types of units per army. The types are hidden until units are engaged in combat, while the number can be displayed using a button combination.
- Positioning and moving armies via a mental image of the situation, as the armies are normally hidden.
- Spending honey on units or upgrades.
- Producing units for the correct army.
- Managing units in combat. One can utilise the inner mechanics of different unit types to mitigate damage taken and maximise damage dealt.
- Performing or defending against multipronged attacks.

Compared to other games, this is a lot to keep track of, especially given how little visual feedback the game can provide without giving away hidden information. This is especially pertinent in the case of novice players who in addition are only just getting accustomed to the controller scheme. Playing with FoW enabled takes the game to another level, which is why it can be disabled to allow players to learn all the other game mechanics first.

4.3 Depth and Richness of Interaction

We tried to make the control of units as intuitive and as expressive as possible. Units can be ordered to go to a certain location, ignore enemies, or even just move in a certain direction without pathfinding.

A player can have up to four armies that they can issue commands to separately. Some players may prefer not to use all available armies, as keeping track of them is a task in its own right.

4.3.1 Game Strategies

The game offers a richness of strategies, each with their own merits. This leads to an interesting metagame in which players try to out-think each other in the strategies they apply.

Two examples of strategies that can be used are:

Standard Opener: In a Standard opener, the natural expansion – the control point right outside the main base – is captured first. This CP can support up to one upgrade. After capturing the natural, the player can branch the strategy and decide whether to invest in a Freeze upgrade (Turtle branch – defensive), a Hive (Boom branch – expand), or save up resources for another unit (Rush branch – attack).

Swedish Opener: Instead of capturing the natural expansion outside the main base, this strategy first captures the CP above/below the main base (second expansion). After capturing this CP, the player is free to branch the strategy in any direction, depending on the information available. The advantage this brings, is that it leaves the natural expansion free to be upgraded with a defensive tower at a later point in time.

Hill Gambit: If the opponent does not immediately claim their second expansion, it is possible to claim it for oneself. By immediately constructing a freeze tower, it is possible to establish a foothold near the opponent's base that is neigh impervious to early-game pressure. Later on it can be upgraded with a heal tower and be used as a basis for many harassing attacks.

Of course, the execution of strategies also relies on mastery over the controller commands, thus dexterity plays a significant role as well.

4.3.2 Playability

Without FoW, the game is complex:

The players are able to monitor the opponent's actions. This leads to players trying to have a slightly larger army than the opponent. This is either achieved by expanding and defending, or by attacking undefended expansions.

With FoW, the game is complex and hard to navigate:

Delayed information about the opponent's strategy leads to the choice of strategy having a larger impact on the game, as it may be too late to build adequate defences against an opponent's attack, or the fact that they had undefended Hives producing large amounts of income may be discovered too late. This makes strategies harder to counter and information becomes a valuable commodity.

Balance

As far as we were able to test it, game balance was decent. No particular strategy seemed overpowered. However, experience shows that good balancing can only be achieved through extensive play. Balance and the pacing of the game can be changed via the configuration scene.

4.4 Approaches to Improve Pathfinding

The following approaches were also considered when trying to improve the performance of the pathfinding algorithm. They could possibly be combined with the existing approaches to create an even more powerful pathfinding behaviour.

4.4.1 Group Pathfinding

When a group of agents close to each other pathfind to the same goal, a lot of duplicate computation is performed. The paths computed for agents in close proximity typically only differ in the first few cells, after which they are identical, making it wasteful to calculate the whole path for every agent.

A solution we considered implementing was to calculate the full path from the most centred agent in the group, while only calculating the path from every other agent to any cell on the full path by using *Breadth-first search (BFS)*.

4.4.2 Common-Goal Pathfinding

When several agents pathfind to the same goal from spread out starting cells, it could be beneficial to save repeated computations by changing from an A* approach to a BFS approach.

Instead of running a pathfinding search algorithm from each start to the same goal, one could perform a BFS from the goal to find all start cells. Whenever a starting cell is found, the path from the goal to the starting cell is returned to the respective agent.

It is a straightforward and simple approach to implement, but combining it with other approaches would increase the complexity: how would one deal with Early-stopping when it is the beginning of the path that is relevant and not the end?

What would the critical "level of spread out starting cells" be in order for this approach to be more viable compared to the *Group Pathfinding* approach?

However, an expansive BFS could prove to be more expensive than performing several A* searches. It may thus be worth considering replacing the BFS with a modified version of A* that keeps its state between path computations. We are not sure if such a thing is possible or efficient, as we have not explored it further.

Another approach, which incorporates Early-stopping, is to compute a BFS from every gateway in every region. Key information from the search can then be added to every Grid cell in that gateway's region. The key information could for example be the best reachable adjacent neighbour to move to, in order to reach a particular gateway. This information can be precomputed and easily combined with gateways from the Region A* approach to find the best gateway to reach the goal's region (see Section 3.4.4). The problem of pathfinding is then reduced to following the precalculated sequence of best neighbours in order to reach the best gateway. This approach would however have problems with dynamic obstacles, as it is purely based on precomputed information.

4.4.3 Coroutine A* Improvements

There are obvious improvements that could be made to this approach. One could be to combine the approach with the Early-stopping technique mentioned in Section 4.4 and return the best path currently found at each frame, in order to make the agent seem more responsive. However, one could argue that the Early-stopping technique on its own would solve the performance hit of having heavy calculations in one single frame, rendering the coroutine approach redundant.

In hindsight it would probably have been better to apply the Early-stopping approach instead of the coroutine approach, even though the coroutine approach had a straightforward implementation. The results were not as good as we anticipated and needed, see Section 3.4.3 for numbers. Spreading the computation resulted in problems later down the line when we wanted to quickly change the pathfinding goal. It simply took too many frames to find the full path to the goal: There was a visible delay in the responsiveness of units.

4.4.4 Path Traversal Improvements

The solution to the worst-case scenario during path traversal, mentioned in Section 3.4.5, Paragraph *Path Traversal*, is rerunning the pathfinding algorithm continuously at a fixed time interval. This naturally introduces a trade-off in performance and effectiveness of the solution. The shorter the fixed time interval is, the more performance it drains, while also making units become "unstuck" quicker.

One approach that could improve the trade-off is to remove the fixed time interval

and instead dynamically decide whether a long or a short time interval should be used. A short time interval is only needed in the worst-case scenario: when a unit is stuck. Methods could be added to the unit to make it decide whether or not it is currently stuck at this particular frame. It could for example compare the last few grid cells to the current grid cell. If the cells are the same it either means that there is no path to follow, or that the unit is stuck. This method could then trigger *unstuck mode*, where the unit shortens its time interval so that it quickly can become unstuck. When the problem is solved the unit returns to using a longer time interval.

4.4.5 Dynamic Obstacles Improvements

When a unit tags its centre cell as temporarily blocked, it also adds its immediately adjacent cells to that list as well. The level of adjacent cells is currently a fixed number. This causes problems for units that are larger, or smaller, than the fixed number. If a unit only occupies one grid cell in size, it would block too much space for other units; on the other hand, if a unit takes up more a lot more than the 8 adjacent cells, it would results in collision and the hindering of a front line forming.

The solution would be to dynamically increase or decrease the number of cells being added. Units could calculate exactly which Grid cells it is currently overlapping, and add all of them as blocked. This would require the introduction of new methods for finding Grid cells for different parts of the unit, or some method to map a radius in the size of the unit to a list of Grid cells.

4.4.6 Pathfinding Into Fire Range

When a unit is within sight range of an enemy, it will try to pathfind towards the enemy until it is in firing range. The issue with this is that units will pathfind towards the enemy, and not towards the closest point within firing range. This causes units to not pathfind towards a wall, even if the enemy is just on the other side of it. Instead, they try to take the long way around. This could be remedied by passing the fire distance into the A* pathfinding method, or by units trying to close the gap by simply moving in the enemy's direction for a couple of seconds, and only pathfind to the enemy, should that fail.

A second problem is that while pathfinding into fire range of an enemy, a unit can possibly pathfind out of sight range. Unless another destination is set or another enemy appears, it immediately stops moving, as the path is cleared when the target disappears. One solution to this is to make units able to identify the scenario and not clear the path. This results in the unit moving all the way to the last known position of the opponent.

Chapter 5

Future Work

Apart from the different approaches to improve pathfinding described in Section 4.4, there are a few more conceptional ideas that could be explored further.

5.1 Expanding Invisibee

The game can be expanded in different ways: Allowing more than two players to play at the same time can introduce team play mechanics to the game. Creating more maps could require players to adapt their strategies to the terrain. We also considered introducing a fourth unit type that can be spawned anywhere on the map to allow players to buy mobility at the cost of resources, and opening up new strategies.

5.1.1 Line of Sight and Vertical Positioning

Another possible expansion of game mechanics would be to introduce *Line of Sight (LoS)* and vertical positioning, via the introduction of high-grounds and low-grounds to the game.

Classically, units standing on higher ground enjoy a visibility advantage over units on the low ground.

LoS makes units standing on higher ground hidden to units currently positioned on lower ground.

This concept can also be applied to combat: low-ground units cannot fire at high-ground units, unless the high-ground units are revealed. Adding LoS and the notion of high- and low-ground could result in the need for units or abilities to reveal units hiding on higher ground. Another problem arises in the form of how to visualise the difference in vertical positioning when the game is in a perfect top-down view.

5.2 Application of Hidden Input in Other Genres

While we chose to explore the concept of hidden input in an RTS game, it could very well be applied in a different setting. An example would be a round-based strategy game, similar to "A Game of Thrones: The Board Game"[1] in which players concurrently during a round prepare actions by placing face-down markers. This forces players to commit to a certain strategy ahead of time without knowing all the opponents' actions. In a PC version, these could be placed via hidden input.

Figure 5.1: Secret command tokens from "A Game of Thrones: The Board Game" that players could place face-down on the game board.

5.3 Output-Only Approach to Hidden Information

The decoy approach used by "Hidden in Plain Sight" works because the player recognises which on-screen character responds to their input. Thus, it relies on knowledge of the controller input and screen output, and their respective timing. The exchange of hidden information here is bidirectional.

In our approach for Invisibee however, the transfer of hidden information is largely unidirectional. At its core, the transfer of hidden information in our approach relies exclusively on controller input: a player issues a command to build something or move units, and the game obeys without the opponent learning about these actions.

This begs the question whether secret information can also be exchanged by only using screen output. Since all information on screen is public, this would require some form of encryption scheme simple enough to be performed in one's head.

Furthermore, each player has to know a secret key shared with the game. These keys would probably be communicated at the start of the game. One way of doing this would be by making creative use of one of the two aforementioned hidden information exchange methods: A player selecting their own code via hidden input springs to mind; or a player figuring out which one of several codes moves when they press a button, with the addition of unused decoy codes that move randomly.Alternatively, the initial secret code could be communicated by using a side-channel such as a private screen or a physical booklet to decipher

[1] https://www.fantasyflightgames.com/en/products/a-game-of-thrones-the-board-game-second-edition/

the meaning of a code displayed publicly by the game[2]. This would only be done once, at the start of the game; the side-channel would then be discarded and unused for the remainder of the game. It is only used as a flavour to communicate the secret code in the beginning of the game.

The simple encryption scheme visualised in Figure 5.2 allows the game to communicate one bit of information to one or several players. Note that it is possible to send different messages to different players using the same ciphertext.

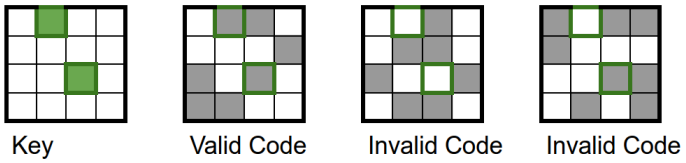

Figure 5.2: Encoding secret screen content

A possible application would be to mark potentially dangerous fields on a playing field in such a way that only certain players can understand. For instance, if the key matches the code, the field is guaranteed to be safe. If not, it is uncertain. Due to the simplicity of the encryption scheme, it is likely that players may figure out each others' keys over time allowing them to gain additional information.

[2]Inspiration drawn from the boardgame "Betrayal at House on the Hill": http://avalonhill.wizards.com/games/betrayal-at-house-on-the-hill

Bibliography

[1] A. J. Champandard, "Near-optimal hierarchical pathfinding (hpa*)," October 2007, (Date last accessed 18-May-2017). [Online]. Available: http://aigamedev.com/open/review/near-optimal-hierarchical-pathfinding/"

Appendix A

How to play

Each player starts off with a base that produces a minimal amount of honey. This honey can be used to produce units or to upgrade control points. The goal is to destroy the opponent's base.

Producing Units

To produce a unit, hold down the right trigger and select the unit to be produced using the directional pad (Up, Left, Down, Right). Units are assigned to one of four armies (A, B, X, Y). By default army A is chosen, but you can change this by pressing the right trigger (RT) + (A, B, X, Y).

Moving and Fighting

To move an army, position your cursor to the desired location using the left thumbstick (LS), then press the corresponding army button (A, B, X, Y). The units will then pathfind towards the selected location. Units will automatically attack enemies they encounter. If the army's button is held down then the units are forced to ignore enemies and move towards the cursor. Instead of ordering units to pathfind to a certain point, they can also be ordered to move in a certain direction. This is done by holding down the army button and moving the left thumbstick (LS) in the desired direction. You can at any time display the number of units in an army by pressing the right bumper (RB) + (A, B, X, Y) for the respecive army's unit count.

Control Points

Control Points are specific locations on the map. Units close to a CP will automatically claim it over time. A claimed CP can be upgraded: Hives produce honey, Freezes attack, and Heals heal allies. A CP can support 1 to 3 upgrades, this is represented by the number of circles below it. An upgrade can be applied more than once. When taking damage, the last upgrade is destroyed first.