Prototype Game Design Documentation

Lab Portfolio

Lab 1: Asteroids

In this exercise our task was to create a simple replica of the infamous arcade game. To achieve this, we had to use Paper 2D Sprites. Through this lab we were introduced to some basic concepts of Unreal Engine (UE), such as handling input, handling collisions, and positioning in a 3D world.

Dangerous Asteroids

To complete this task the Asteroid Blueprint (BP) was extended. The asteroid handles the hit event: in case the other Actor in the hit was the ship, then it destroys both Actors. This mechanic is rather crude, as it leaves no room for error, and makes the game difficult. It could have been improved with a health bar mechanic.

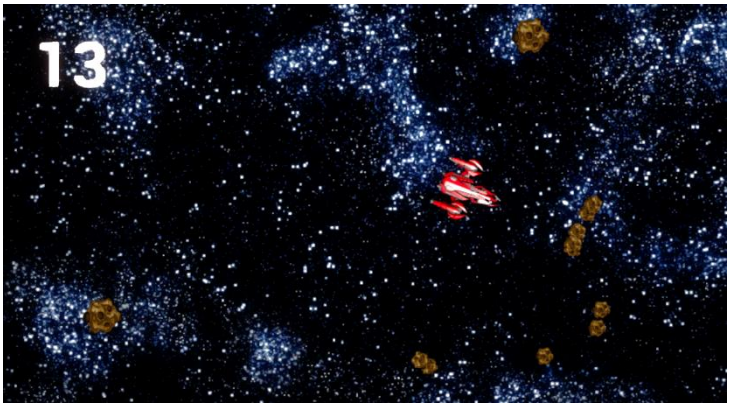

Figure 1 Asteroids game

Breakable Asteroids

Again, the Asteroid BP was extended. A new variable was introduced that defines the size of the asteroid, big or small. At first only one smaller asteroid was spawned in the place of the original, but it was changed later to spawn five, instead. A problem that was encountered here is the positioning of the new Asteroids. If they are spawned in the same location, they will get stuck together, making a single overpowered one that takes multiple shots to stop, giving no visual cues as to how many shots are required. To solve this problem the small asteroids spawn randomly within a small distance of the original. This still makes it possible for them to get stuck together, which seemed like an issue at first, but having played with it, it seemed like a more fun mechanic. This behaviour introduced asteroids of different shapes, that also behaved differently as they required multiple shots, but contrary to earlier this was paired with visual feedback as the parts that were hit disappeared.

Testing with Others

The chosen challenge for this task was a simple high-score mechanic that is at the core of a lot of arcade games. A score variable was introduced in the game state that is incremented by the asteroids before they are destroyed. The score was displayed in the level through a Text Render Actor. Some other lab participants tried the game; however, they did not find it fun, mainly complaining about the manoeuvrability of the ship. Some tweaks were made to the constants that govern the ship's movement to improve on this.

Lab 2: Platformer

In this lab we learned how to achieve separation of concerns over visuals and controls. To do this, we used engine features such as the Player Controller and action mapping. We also learned how to animate objects using Timelines and how to create levels using a Tile Map.

Safe Jumps

The created level (Figure 2) starts with many safe jumps between the white platforms. This introduces the controls and the behaviour of the pawn to the player. To increase the difficulty, moving hazards are introduced early on.

Difficult Horizontal Jumps

The next challenge in the level is a difficult horizontal jump with two moving obstacles. The alignment of the platforms was optimised so that it can just barely be made with good coordination. Though, there is no penalty for failing the jump so this is hardly a challenge.

Leap of faith

After this jump, the player is faced with a leap of faith. An arrow was drawn within the Tile Map on another non-colliding layer, so that this is made obvious to the player. While the player is falling there are some static obstacles. The challenge here is to use in-air controls of the character to avoid these; it may require multiple tries to learn the position of the obstacles.

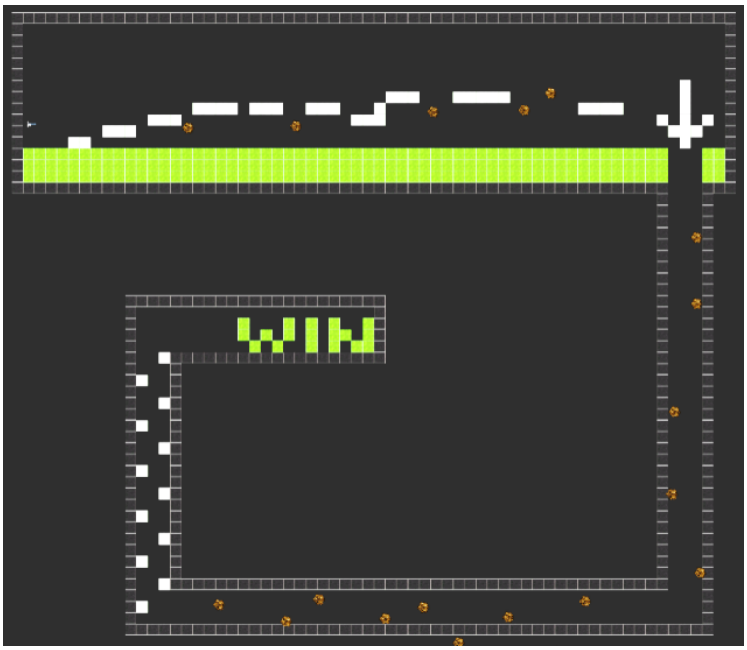

Figure 2 Simple platformer level (player starts top-left)

Pattern Recognition

The next challenge is a moving hazard maze. Each obstacle in this corridor moves at its own pace, and direction, that is defined through BP parameters individually. The player has to learn their movement and avoid them accordingly. Since the obstacles all move in a linear fashion as shown in the tutorial, this is not really an interesting challenge, even if they move at different speeds. It could be improved by including different moving behaviours, or movement that somehow responds to the player's actions.

Climbing

The final task for the player is a ladder. Here, they have to jump from side to side until they make it to the top. Perhaps the gap that was left here is not wide enough, as it is easy to correct a mistake, because the character has considerable in-air control that is required to make the leap of faith. When the player completes the challenge, they are faced with a background "WIN" text signifies the finish of the level.

Lab 3: First Person Shooting Gallery

In the game created through this exercise the player can move around and shoot targets while avoiding obstacles that cause damage, while having the goal to clear all targets before the time runs out. This lab taught us how to create a HUD and how to handle health and damage.

Level Design

The top-down view of the level that was created in this lab can be seen in Figure 3. The circle wireframes are the targets, the black rectangles are holes that restart the level if the player falls in, the square wireframes are damage-causing obstacles. The damaging obstacles have been encapsulated into a BP as opposed to having a Damage Causing Volume over a Particle System, that was demonstrated in the guide. The level can be completed even without making the last jump which gives players choice.

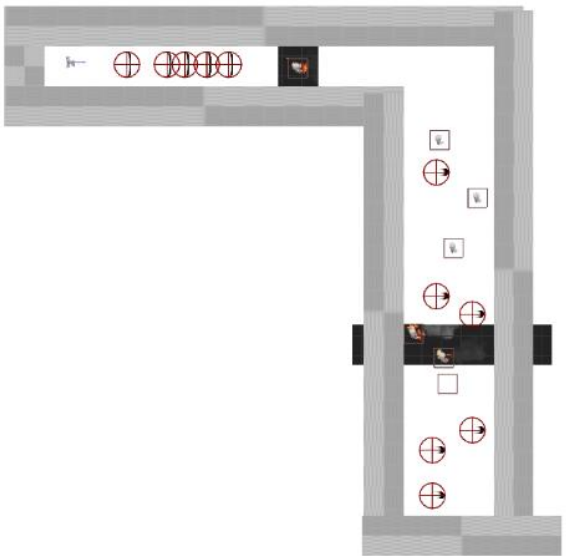

Figure 3 First Person Shooter level (player starts top-left)

Time Pressure

To add time pressure to the game a timer was added below the health display. This timer starts at 10 seconds and every time a target is hit the timer is incremented by a second. The actual variable was stored within the Game Mode, while it is updated via communication with the Targets. Either the level design or the time reward could have been adjusted more carefully to make this mechanic more interesting as most often the level finishes with more than 10 seconds on the timer as targets are placed too close together.

Lab 4: Lateral Thinking

In this lab we created a Portal-like gravity gun that can pick up objects in the world. Through this we learned how to do line traces, how to communicate between Actors, and how to use physics constraints, to create puzzles.

To solve this lab the following level was created (Figure 4). The circles are buttons, squares are pickup objects. The rooms are separated by doors, opened by the pressure sensitive buttons. Note the connections between doors and buttons on the figure. To reach the goal in the bottom-right corner there is a platform that can be reached only through the grey see saw and an explosive barrel has to be used to destroy the black wall.

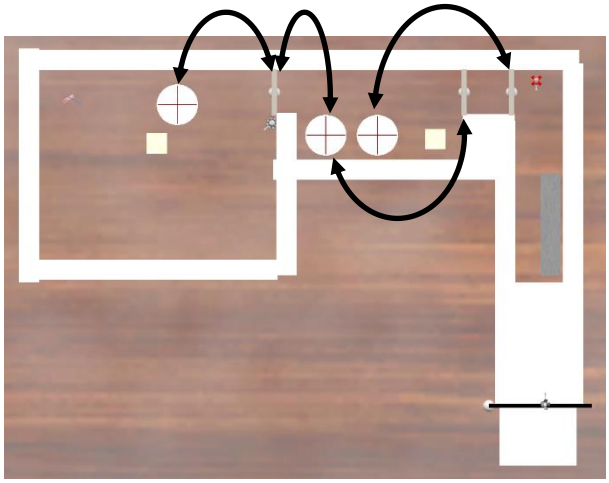

Figure 4 Lateral Thinking level (player stars top-left)

To complete the level the player has to open the first door with the first object. Then in the second room the player has to exploit the fact that two buttons can be operated with only one object. This opens up all doors, using only one pickup. Then the first object needs to be retrieved again to operate the see saw. Once the ramp is raised, a barrel needs to be picked up from below the other end of the see saw. Finally, the barrel needs to be placed close to the black wall and shot from a distance to break it and finish the level. If the barrel explodes too close to the player, then the level restarts.

To achieve this behaviour the Buttons and Doors were extended to send and receive on/off signals so that they can be controlled in a more granular fashion. The logic for wiring the Doors and Buttons is hard coded in the level blueprint. From a technical perspective, the connection between these components could be defined as a BP Interface, so that the connection between them could be changed easier. A new Barrel BP was created, that applies a lot of damage in a small radius, if it is hit by a bullet. The Player Character was extended to fire bullets the same way as in the previous exercise, as well as to restart the level if it takes any damage.

Lab 5: Third Person Tag

In this lab we have built a game where an AI chases the player's avatar in an open environment. This task has showed the basics of Perception and Behaviour Trees.

Harmful AI Patrols

The AI Character was extended to restart the game on the hit event, as a penalty. The Waypoint BP was extended to include an editor-only Cone mesh. This way the path of the patrol can be seen while editing the level, but it is invisible while playing.

Seeking AI

To complete this exercise the map in Figure 5 was created. The connections between Waypoints are denoted by arrows. There are three AI Characters and three patrol paths. The player first has to go through a corridor that is patrolled end to end. This serves as an introduction challenge. After passing this corridor the unpatrolled corridor can be used to safely observe the behaviour of the next two patrols in the open space, using the peeking behaviour of the third person view. With the right timing the player can run over to the blind spot in the corner and from there to the goal area.

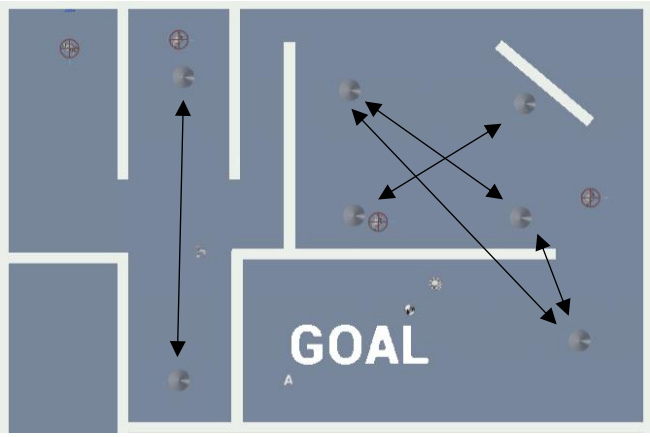

Figure 5 Seeking AI level (player starts top-left)

Using additional Behaviour Tree Tasks, the AI Character was modified to use a slower pace when patrolling, but a higher running speed when chasing the player. This was done by modifying the Max Speed variable within the Character Movement Component. If the AI touches the player's avatar the level restarts. After this, the AI was further extended to remember and move to the last known location of the player if the line of sight is broken. This extension increased the difficulty of the challenge considerably, as this made it practically impossible to hide after the player has been seen.

Shooting AI

Finally, the Behaviour Tree was extended with an additional Task that turns towards the player and shoots, if the player's avatar is in range. To ease the difficulty of the challenge a health counter was added, that is decremented when a bullet hits the player; the player gets 3 health initially. When this counter reaches 0 the level restarts.

Lab 6: Inventory

In this lab exercise we had the opportunity to make a game where items can be picked up, stored in an inventory and later used to solve puzzles. Through this we learned how to create Widgets and how to create a simple inventory mechanic with BP Interfaces.

Obstacle Variations

To complete this exercise two simple locks were created, that open doors. To hint the relationship between the key and lock they use matching shapes and materials. The first lock is square shaped and takes a quad pyramid shaped key, both having a golden colour. The second lock is circle shaped and takes a capsule shaped key, both having steel colour. The locks open the same doors that were used in lab 4.

Multi-Key Obstacle

To drive the logic of the connection between Doors and Locks, a two-way BP Interface was created. The Door implements the Unlockable interface that defines the Unlock function. The Locks implement the Locking interface which defines an Announce method. The connection is described via an array of Actors implementing Locking, on the Door BP. When play begins, the Door Announces itself to each Lock. This way, when the Locks are activated they will call Unlock on the right Door(s). When all Locks associated with a Door are activated the Door opens.

This allows for efficient handling of a many-to-many relationship between Locks and Doors. The level design could have been improved by using this feature more extensively with additional challenges.

Single Use Keys

When a Lock is activated it checks whether the right key is being used, i.e. whether the correct inventory item is active (configured in the level). If the key passes the check, then it calls the Remove Item function on the Widget where the Inventory is stored. This removes the item from the Inventory, ensuring that the Inventory Index stays in bounds. If a Lock does not remove the item from the Inventory then it effectively becomes a multi-use item.

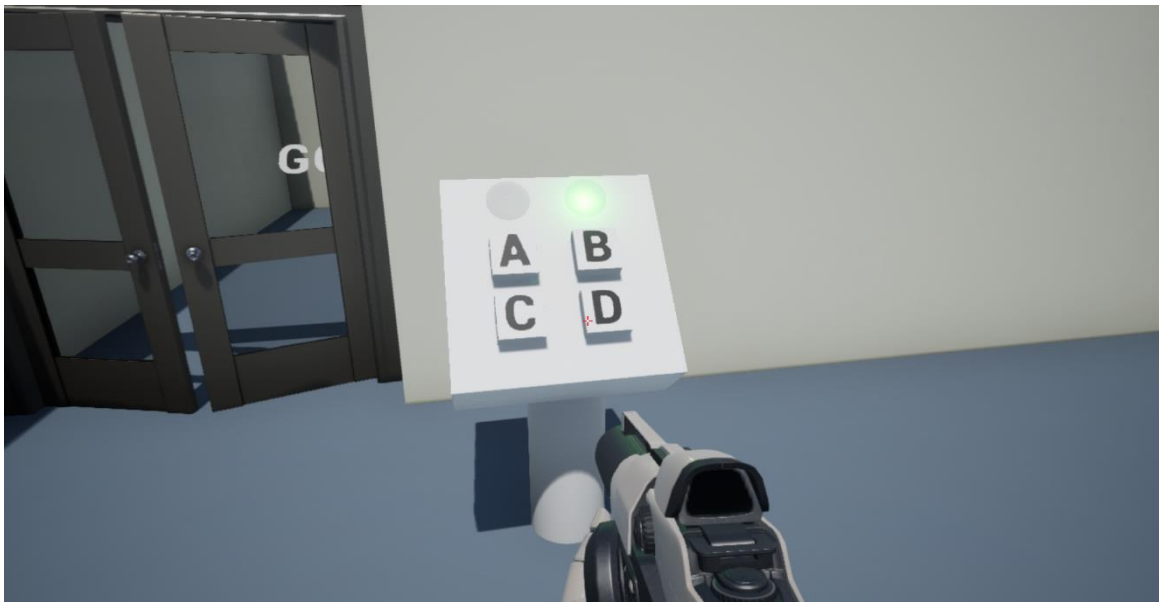

Figure 6 Code lock, opening a Door

Extended Interfaces

For this exercise a dynamic, key-less lock was created, pictured in Figure 6. This Lock can be configured (with a String) to open with a specific code.

When this Lock is used by the Player Character, it uses line tracing to determine whether a button was pressed. It then accumulates an input buffer of button presses as a String. If this buffer matches with the code parameter then it calls Unlock on the associated Door(s). The panel is built of simple Meshes and Text Render Components. A red and green coloured Point Light is used to provide feedback to the player, when using the buttons. A short red flash is shown when a button press was registered. A longer red flash is shown when the input buffer is reset (after 3 seconds of inactivity). The green light is permanently turned on if the right code is entered (see Figure 6).

Using these features the following level was created, pictured in Figure 7. The player is spawned facing a Door, which is see-through, showing the player the "GOAL" text, to signify that the objective is to open the Door. To accomplish this the player needs to pick up the key in the corner of this room. The key needs to be carried to the next room where it can be used to activate a similar looking Lock. Before leaving the Door that is just opening to the left the player needs to pick up the steel object next to the Lock. This object activates the Lock to the last room, where the code is revealed: "BAD". When this code is entered on the panel in the first room the last Door opens and the objective is complete.

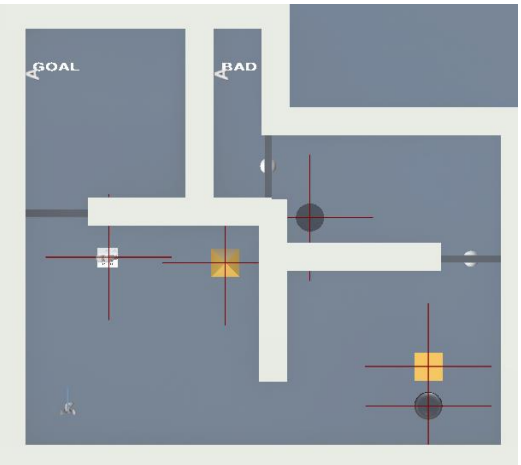

Figure 7 Inventory level (player starts bottom-left)

Lab 7: Driving

Through the exercises of this lab we learned how to create BP Components, and how to use the multiplayer features of UE.

Additional Components

To complete this task the Can Pickup Component was extended to call an Increment Score function on the Game Instance. An additional Actor was created, Big Coin, that has the same behaviour as the Coin, however it increments the player's score by 5. The spinning behaviour of the Coins were encapsulated into another Component, called Rotates, that takes a parameter, defining the rotation speed.

A third BP Component was implemented called Adds Impulse. This Component adds an impulse in the forward vector direction of the player's actor. This can be used either to create a Boost by adding a positive impulse or an Obstacle by adding a negative one, defined by a parameter to the Component. The Boost Actor was used more extensively (see Figure 8) as the car can pick up great speeds in a short distance when many Boosts are stacked after one another, which is quite entertaining in a driving game.

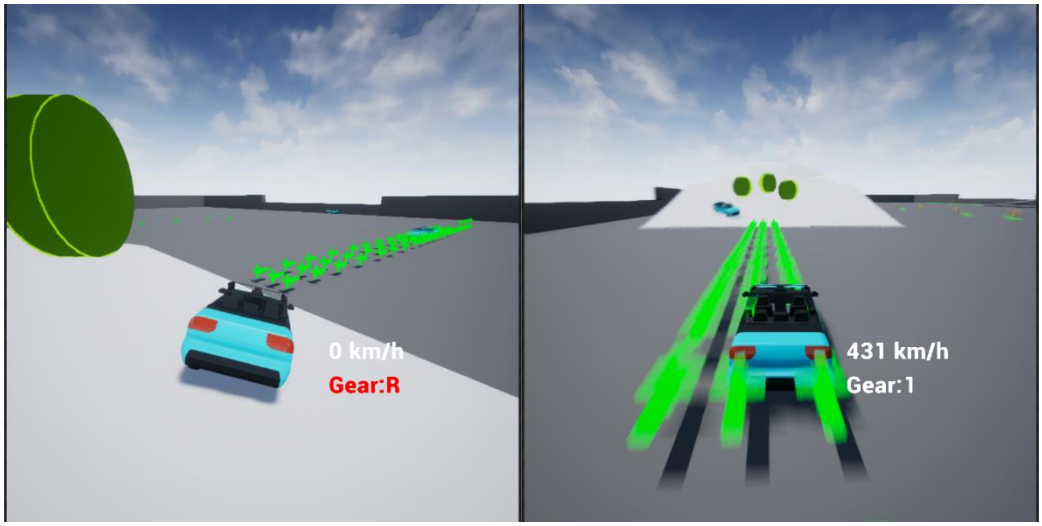

Figure 8 Ready for lift-off?

Multiplayer

To complete this exercise an extra level was created as a duplicate of the first level. The arrangement of pickups was changed. The main menu was extended with another button to start this second level. The levels start in server mode, and the second player can join via a third, "Slave" button.

The levels were extended with a pickup that needs to be collected to win the level and progress to the next one. This item is also responsible for the countdown. It keeps players' controllers disabled until there are two players, then it starts a countdown timer (simple Print String), enabling the controllers in the end. This item has a parameter that defines which level to progress to. In this game this was defined as Level1 to Level2 to Main Menu.

Game Design & Specification

Core Game Play

The rules are based on Santorini [1], with a different visual theme. The game is played between two players, taking turns. To set up the game each player places two builders (Figure 9, [2]) on the 5x5 square grid board.

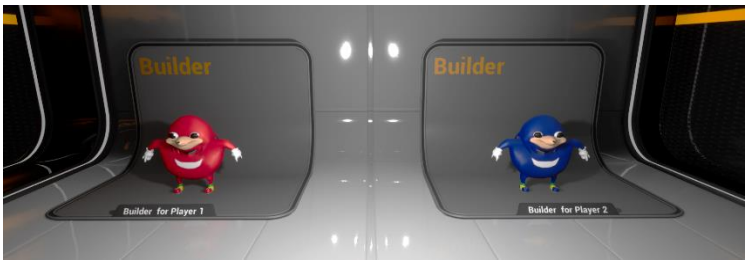

Figure 9 Builders

After the setup phase, during each turn, a player must move a builder to an adjacent position, and then use that piece to build a block (Figure 10). A builder can move to any adjacent positions (in 8 directions), that is at most one level taller, and is not occupied by another builder. Players can build to any adjacent position that is unoccupied.

Figure 10 Blocks

The goal of the game is to move any builder to a third level block (Figure 11). The trick is that both players can build a top-level block (Figure 12, [3]) on top of the third level, that blocks both players from moving or building there anymore (Figure 13). If a player cannot make legal moves anymore, they lose.

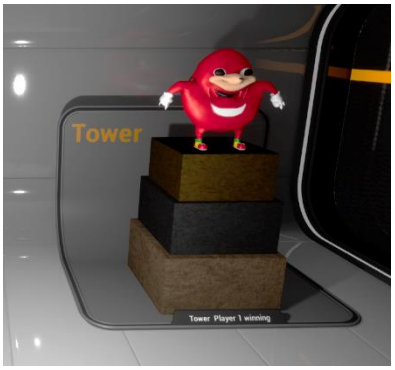

Figure 11 A winning move

Game Flow

The game encourages free play as there are no achievements, leader boards or other competitive features in place. This seemed appropriate as many board games are structured like this.

There is a difficulty curve however, in terms of AI difficulty. This varies from easy through medium to hard. In the finished product, an easy AI should be easily beaten by a child. The medium AI should provide some difficulty for average players. The hard AI should be able to exploit the player's mistakes consistently.

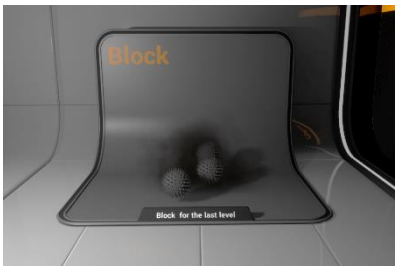

Figure 12 Top-level block

In terms of rules, there is a difficulty curve within games as well. Making the right moves in the late-game is considerably harder (for a human) as there are many possible sequences of moves, that could lead to a victory, for both players.

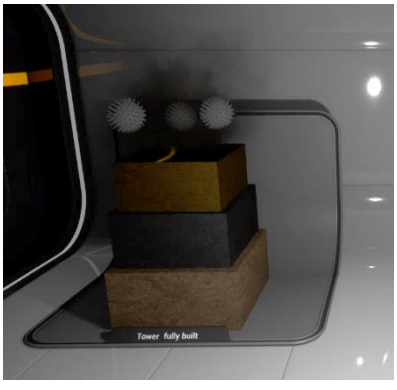

Figure 13 A fully built tower

Characters

The player's characters are controlled by mouse click events. All possible moves of the player are displayed on the board via indicators. Since both builders can move to the same position, an immediate step is required to select which builder is to move. This is shown using the indicators pictured in Figure 14.

Figure 14 Selection indicators (selected state on the right)

An inverse quad pyramid, floats above the builders, matching their colour. This was designed taking inspiration from the Sims diamond, so players should feel familiar. The indicator's internal state can be represented as a state chart in Figure 15. In normal state, the indicator spins and hovers up-down to show that it can be interacted with. In active mode, the indicator's vertical motion is faster. In the selected state, it lights up to indicate the selection, keeping the increased vertical speed. When another builder is selected then the indicator transitions back into normal state.

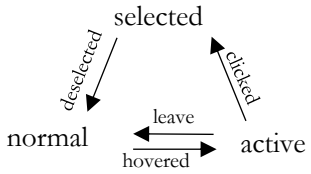

Figure 15 Selection indicator state chart

Once a builder is selected, the displayed moves are filtered to the ones this builder can make. Moves are displayed using pulsating lights on the board (Figure 16). When this indicator is hovered, it provides a preview of the move, by moving an appropriate Actor to the target location temporarily. E.g. when a build move is hovered, the block required for that level is displayed under the pointer. A technical note: if the block actors have an internal state, then consistency between the preview and placed static block should be ensured. In the prototype this is handled via a Character Pool, that is shared between the different Move Indicators and the Board. It encapsulates the logic of displaying a Character permanently or temporarily on the Board.

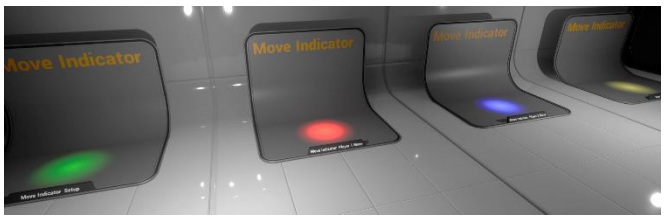

Figure 16 Move indicator

To allow the player to visualise available moves before committing to them, an action buffer is used that lets players undo their actions within a turn. This is facilitated by an "UNDO" button on the right side of the screen. When no more moves are available to the player another "END TURN" button is activated, that commits the moves in the action buffer, and gives control to the next player.

As the game progresses, some positions become hidden from certain angles due to the height of the towers. The game uses a natural drag and move interaction to rotate the player's view around the board using the right mouse button.

AI

Since this game requires some planning of moves to play properly, UE's built in Behaviour Tree based AI is not adequate to create a competent AI. To account for this, a Monte Carlo Tree Search algorithm is used, that was implemented based on [4], [5]. The algorithm is changed in some ways to adopt it for a real-time setting and to fit it into UE. For an introduction to the algorithm please refer to [6].

Since UE does not allow for recursive structs, the game tree is implemented as a BP, even though the game state is stored using structs. Thousands of Game Tree nodes are created with every search, therefore spawning them should be fast. Creating Actors takes more time as they have to be placed in the level, therefore Object is used as a base class to create the Game Tree. However, Objects do not inherit from BP Function Libraries, where the game rules were implemented. Therefore, every Game Tree node has to store a reference to a "rule delegate", a singleton Actor that has access to the required methods from the rules library.

MCTS uses so called tree policies to navigate the search and default policies that execute a "playout" from a given game state. Using ideas from [5], these are implemented as separate BPs that are used from the search core through BP Interfaces. This allows for AI Controllers that have different capabilities, making it easy to create different AI levels.

BPs do not allow for multi-threading; thus, the search must not block the thread. To create non-blocking graphs, 0s Delay nodes can be used. This puts the continuation from Delay on the end of the event queue, giving other events the opportunity to update. If the Delay is used too often, then it can result in a slow search, however. This trade-off should be tuned to give 30 fps on the recommended configuration in the released product.

The AI Controller uses two parameters that affect its behaviour: rollout depth and target confidence. The rollout depth limits the maximum number of moves the default policy can make. Semantically, this translates to "if the game does not end in *depth limit* turns, then the winner is unknown". This results in more accurate simulations, as they do not explore unlikely long games. To further improve simulations, a greedy move selection strategy is used, i.e. if upward moves are available then one is chosen randomly. This limits the length of simulations further, but it might make the rest of the algorithm greedier; further exploration of the effects of this is required. The other difficulty parameter is target confidence. This essentially means that, at least *target confidence* nodes must be explored before choosing a move. The more moves are explored the better the selection is, making this the key indicator of AI strength.

Since both players make somewhat dependable moves, therefore throwing away the search tree after a move is selected would be wasteful. The event delegation system of the game works in a way that allows for Controllers to receive the opponent's moves. This allows the AI Controller to "advance the tree" using the opponent's moves. A permanent game tree can be used to look up the subtree corresponding to the move. If this subtree was explored by an earlier search then it becomes the new root, otherwise a new tree is generated with the next search. This ensures that no searches are wasted; and also means that the AI can be faster in the late-game, as it can carry over computation from previous turns, i.e. less moves need to be explored to reach *target confidence*.

Unfortunately, this was not enough to create a competent AI, that fits the specifications described earlier. All difficulty levels can make reasonable moves, however they often make mistakes in the late-game, such as failing to block an opponent when it is possible to prevent a win. Furthermore, they take too long to conduct the tree search, making the prototype rather boring, when the player has to wait about a minute for the hard AI to make a (bad) move. To improve on this, the finished product should use better heuristics for simulation or evaluate other search algorithms that can tackle this problem space. Perhaps it might be enough to implement the tree search in C++ to increase its performance, by running it at full speed on a separate thread.

Levels

Levels in this game are used to separate states or "screens". The game starts with the Main Menu; the player can either choose to start the Tutorial or a regular Game. There is an additional Demo level not available to the player, that was used to create the screenshots, and to quickly preview visual changes.

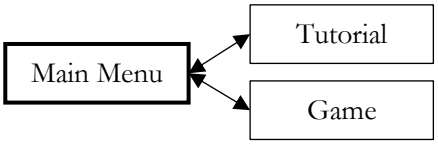

Figure 17 Level transitions

Before starting the game, the Main Menu provides a configuration screen. This lets the player to 1) choose who is controlling which colour 2) tune the AI difficulty 3) enter a name that is used throughout the game (e.g. "Your turn Bob!" ticker message). Both player vs player and AI vs AI games are allowable.

The Tutorial level is a scripted version of the full game. The starting board state is designed to show all rules and features within a few moves, where the player has no choice, after that, the player is free to explore the game. A simple dialogue system is used to teach the controls and rules. A limited narrative is used to frame the game rules and characters ("get to the top of Uganda").

Tutorial Walkthrough

- ☐ Show initial board, prompt player to read "story"
- ☐ Show the goal of the game
- ☐ Prompt the player to select a builder
  - Click the builder with an indicator
- ☐ Prompt the player to move closer to the goal
  - Click the only space with an indicator
- ☐ Prompt the player to build
  - Click the only space with an indicator
- ☐ Prompt the player to end their turn
  - Click the button that just showed up
- ☐ Tell player that this was a turn that is repeated through the game
- ☐ Prompt player to wait for the opponent's turn
- ☐ Make the scripted moves with some delay to simulate an AI and show the progress bar
- ☐ Introduce limitations: cannot move to occupied space
- ☐ Cannot move more than 1 level up
- ☐ Prompt the player to move closer to the goal (one step away from win)
  - Click builder
  - Click the only space with an indicator
- ☐ Introduce limitations: cannot build on an occupied space
- ☐ Prompt player to build
  - Click the only space with an indicator
- ☐ Prompt the player to end their turn
  - Click the button that just activated
- ☐ Simulate AI turn, making sure that the builder is now invisible if the board was not rotated, and that the player is blocked from making a winning move
- ☐ Tell the player that this space is now blocked
- ☐ Tell the player that the camera can be rotated using the right mouse button
- ☐ Tell the player about the final rule: if one cannot make legal moves then they lose
- ☐ The tutorial is complete
  - Return to Main Menu or keep playing using normal rules on the current board (vs easy AI)

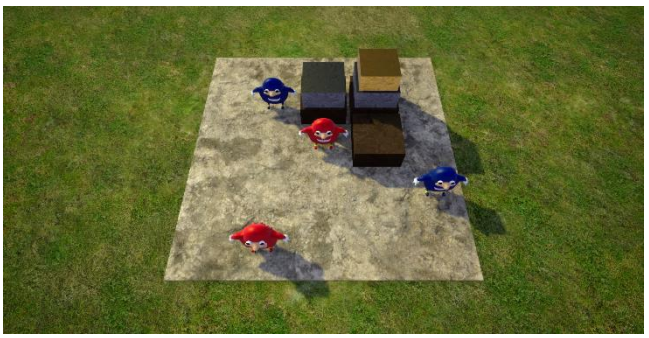

Figure 18 Initial tutorial board state

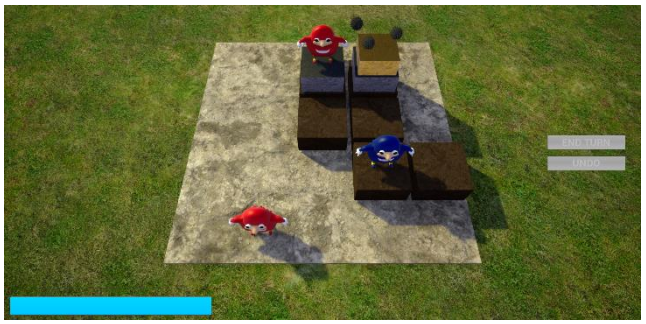

Figure 19 Final tutorial board state

Figure 20 Tutorial complete

REFERENCES

[1] Dr. Gordon Hamilton, 'Santorini', *Roxley Game Laboratory*. [Online]. Available: https://roxley.com/product/santorini/. [Accessed: 01-May-2018].

[2] tidiestflyer, 'Knuckles', *DeviantArt*. [Online]. Available: https://tidiestflyer.deviantart.com/art/The-Knuckles-meme-as-a-3d-model-704695335. [Accessed: 06-May-2018].

[3] Loïc Dautry, 'Virus'. [Online]. Available: https://sketchfab.com/models/ef45c469c17b4c60884dc0cddf868bc5/. [Accessed: 06-May-2018].

[4] C. B. Browne *et al.*, 'A survey of monte carlo tree search methods', *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[5] J. Kulick, *mcts: An implementation of Monte Carlo Tree Search in python*. 2018.

[6] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. V. D. Herik, J. W. H. M. Uiterwijk, and B. Bouzy, 'Progressive strategies for monte-carlo tree search', *New Math. and Nat. Computation*, vol. 04, no. 03, pp. 343–357, Nov. 2008.