University of BRISTOL

Team Project Report

Team A: Eldon Electronic

TIME MINE

Team Members:

Zac Woodford
Team Manger

Anton Wallstedt
Lead Designer

Nisa Bayraktar
Animation & Model Engineer

Henry Hartnoll
Lead Programmer

Paolo Mura
Code Maintenance

Samuel Balson
Lead R&D

Angus Robertson
Environment & Game-play Engineer

Contents

1 Signed Contributions **3**

2 Top Five Contributions **4**

3 Nine Aspects **5**

4 Abstract **7**

5 Team Process & Project Planning **8**
- 5.1 Design Evolution . . . 8
- 5.2 Collaborative Development . . . 8
  - 5.2.1 Apps and workflow . . . 8
  - 5.2.2 Meetings . . . 8
  - 5.2.3 Paired Programming . . . 9
- 5.3 Agile Sprint . . . 9
- 5.4 Learning Collaborative Development . . . 9
- 5.5 User Feedback . . . 9

6 Individual Contributions **10**
- 6.1 Henry Hartnoll . . . 10
- 6.2 Zac Woodford . . . 11
- 6.3 Nisa Bayraktar . . . 12
- 6.4 Anton Wallstedt . . . 13
- 6.5 Angus Robertson . . . 14
- 6.6 Paolo Mura . . . 15
- 6.7 Samuel Balson . . . 16

7 Software, Tools & Development **17**
- 7.1 Software and Tools . . . 17
- 7.2 Modularity . . . 18
  - 7.2.1 Asset Design . . . 18
  - 7.2.2 Game Objects . . . 18
  - 7.2.3 Code . . . 18
- 7.3 Maintenance . . . 18
  - 7.3.1 Best Practices . . . 18
  - 7.3.2 CI/CD Workflow . . . 18
  - 7.3.3 Documentation . . . 19

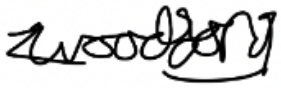
8 Technical Content **20**
- 8.1 Core Functionality . . . 20
  - 8.1.1 Character Controller . . . 20
  - 8.1.2 Game State . . . 20
- 8.2 Time Travel System . . . 20
  - 8.2.1 Overview . . . 20
  - 8.2.2 Capturing State . . . 21
  - 8.2.3 Tracking Players . . . 21
  - 8.2.4 Replay System . . . 21
  - 8.2.5 Player-TTS Interaction . . . 21
  - 8.2.6 TTS Interface . . . 21
  - 8.2.7 Multiplayer Synchronisation . . . 21
- 8.3 Test Framework . . . 22
  - 8.3.1 HUD Debug Panel . . . 22
  - 8.3.2 Unit Tests . . . 22
  - 8.3.3 Time Simulator . . . 22

8.3.4 Logging and Diagnostics . . . 22
8.4 Communicative Mechanics . . . 23
8.5 Level Design . . . 23
8.6 Modelling 3D assets . . . 24
8.7 Aesthetics . . . 24
8.7.1 Dynamic Shaders . . . 25
8.8 Post Processing Effects . . . 25
8.9 2D Assets . . . 25
8.10 HUD . . . 26
8.11 Menu Management . . . 26
8.12 Tutorial . . . 27
8.13 Animations . . . 27
8.13.1 Animating Characters . . . 27
8.13.2 Animating Dissolve Effect . . . 27
8.13.3 Tutorial Animation . . . 28
8.14 Particle Effects . . . 28
8.15 Time Dynamic features . . . 28

Chapter 1

Signed Contributions

Zac Woodford: 1.00	Anton Wallstedt: 1.00	Nisa Bayraktar: 1.00
Henry Hartnoll: 1.00	Paolo Mura: 1.00	Samuel Balson: 1.00
	Angus Robertson: 1.00	

Zac Woodford 

Anton Wallstedt 

Nisa Bayraktar 

Henry Hartnoll 

Paolo Mura 

Samuel Balson 

Angus Robertson 

Chapter 2

Top Five Contributions

The following is a link to our team video: Time Mine Trailer

1. To the best of our knowledge, we are the first game ever with full fluid, granular time travel. We achieve not only this but also deliver the same functionality synced over a distributed network of players. [8.2].
2. Utilised a user centred design approach to develop a system that makes time travel comprehensible. While the time travel itself is a grand achievement, communicating this to a player so its functionality is understandable and the player can begin to utilise time travel for problem solving is, possibly, even more impressive. [8.4].
3. Utilised the universal render pipeline (URP) to enhance the games graphical fidelity along side using advanced techniques within the URP shader graph to create a modular Cel-Shader. [8.7].
4. Created all 3D and 2D assets in-house, with all 3D textures being procedurally generated to save performance. [8.6, 8.9,8.7].
5. Robust test framework for the time travel. [7].

Chapter 3

Nine Aspects

1. Team Process

    - As the game idea became clearer and rate of development sped up, our team communication improved significantly. After the exploratory phase of the software tools ended, team meetings were held at the start of every day to coordinate tasks and broadcast important information between team members. [section: 5.2.2].
    - Used Slack, an industry standard team communication tool, to keep team members connected and messaging organised. [section: 5.2.1].
    - Used Miro, an effective software for brainstorming new ideas, illustrating code concepts, explaining code with UML diagrams and for story boarding. [section: 5.2.1].
    - Utilised paired programming to quickly debug code and overcome obstacles in the development. [section: 5.2.3].

2. Technical Understanding

    - Time travel required an implicit understanding of C# data structures and Photon multiplayer to deliver [section: 8.2]
    - The Toon shader required extensive knowledge of digital rendering technology as well as how to effectively utilise the Universal Render Pipeline (URP) shader graph, even circumventing it's limitations [section: 8.7].
    - Many of our time dynamic system such as the dissolve shader [section: 8.13.2] and animated post processing effects [section: 8.8] utilize parallel post processing systems within unity and as such required knowledge of parallel processing techniques to effectively implement.

3. Flagship Technologies Delivered

    - The filed of communicative mechanics was our effort to make a very complicated mechanic, time travel, understandable, intuitive and engaging for players.[section: 8.4]
    - The time travel system is an incredibly impressive technical system that allows multiple players to experience fully fluid time travel [section: 8.2]

4. Implementation & Software

    - The implementation of C# data structures for time travel outside of the unity environment allowing for simpler user testing. [section:8.2].
    - Circumvention of the URP shader graph limitations via hlsl script to create a stylized cel-shader [section:8.7].

5. Tools, Development & Testing

    - Implementation of a rigorous work flow for developing features [section:7.1].
    - utilization of a test framework for ensuring code stability. [section:8.3].

6. Game Playability

    - The section on communicative mechanics details the teams research into playability [section: 8.4] and how that was achieve with such a complex central mechanic as time travel.

7. Game Look & Feel

- The construction of a series of in house cel-shader modules that allowed the construction of specialised shaders for replicating textures [section: 8.7].
- Application of post processing effects to enhance game aesthetic [section: 8.8].

8. Novelty & Uniqueness

- The first instance (to our knowledge) of a fully fluid time travel system with multiplayer [8.2]
- custom built Toon shaders [section: 8.7]

9. Report & Documentation

- An extensive library of documentation was generated to streamline development. [section:7.3.3]

Chapter 4

Abstract

Time Mine is a game where two teams of players engage in a game of cat and mouse across space and time. One Team, the miners, are trying to collect enough crystal, within five minutes of linear time, to meet their contractual obligation to their employer. These crystals however only appear at certain points in time so miners must utilize the time travel system to find them. The other team, the crystal guardians, try to stop the miners from collecting enough crystals by grabbing them. When caught, a miner looses half their crystals and is flung through time. If the miners have enough crystals to meet their threshold by the end of the game they win, otherwise the guardians win. To find the crystals the Miners have a crystal tracking device. The Guardians however have to utilize the time ghosts to find the miners, like following footprints in time.

The heart of our game is the time travel system [8.2] which allows players to travel backwards and forwards in time (within the time of the game). When time traveling backwards on time players see the world around them reverse fluidly, past versions of players are played in reverse, falling sand rises, destroyed crystals reform, the sun changes position in the sky, players aren't just "warped" to a position in time. likewise, when traveling forward the world around the player accelerates.

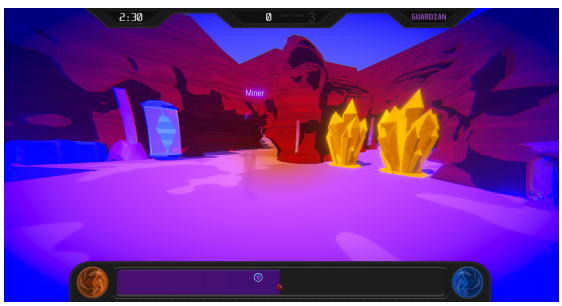


Figure 4.1: Screenshot of player time-travelling forward.

Because of the complexity involved with understanding time travel a large portion of our development was set aside to research how to effectively convey time travel to players in a way that would allow them to utilize it in an effective and engaging way [8.4]. This lead to the development of many ancillary features, the most notable being the time dynamic objects [8.15] such as the collectable crystals, breaking and growing crystals, changing sun position, and falling sand[8.7.1].

Figure 4.2: Screenshot of a crystal breaking.

Much of our development effort was also invested in the game aesthetic. We utilised the universal render pipeline (URP) extensively to build a stylised cel-shaded look which was then enhanced by the application of URP post processing effects. The game also takes place on another planet with a similar geology to the American mid-west mesa giving the game western feel. These factors combined create a game look and feel that is truly unique.

Chapter 5

Team Process & Project Planning

5.1 Design Evolution

The largest obstacle our team faced was that our core game mechanic, time travel, was too novel. Users at every stage of development struggled to understand it. As such we adopted user centered design principles to build features around time travel that would facilitate greater playability. Through constant iterative development and user feedback [5.5] we developed the field of *communicative mechanics* [section: 8.4] as a core technological component of our game. Player and panel feedback was incorporated at every stage of development (concept, MVP, Beta and Final version) to influence the communicative mechanics [section: 8.4]. An example of this would be that our initial game idea was a first person shooter, this was changed to hide and seek as hide and seek through time was more easily understood by users. We then later added an objective for the hiders after receiving feedback that their player experience was un-engaging. However because of this highly fluid development cycle we were unable to accurately plan future development, we initially tried too, by using a gantt chart, but this became quickly outdated as mechanics and features changed. However focusing our development around time travel meant that the rest of our game had a cohesive design, every other mechanic and system served time travel.

5.2 Collaborative Development

5.2.1 Apps and workflow

The team tried as much as possible to work together in person as this greatly enhanced productivity. Being able to look at each others screens, elaborate concepts on a wipe board and have impromptu discussions were invaluable and where we couldn't do this in person, we emulated it with software such as Microsoft Teams, Miro and Slack. It would have been preferable for us to work together in person every day but due to limitations posed by timetabling and the pandemic this sadly wasn't possible. Development features and tasks were recorded in a Kanban board on Asana. This became the focal point for our project as tasks could easily be added or removed in line with changes to the game design. Working from home presented problems as without oversight team members would sometimes over-complicate features; this could have been easily remedied by more collaborative development in person where impromptu discussion could easily resolve issues and simplify systems. Asana was also used to keep track of bugs that were discovered alongside development, allowing other team members to resolve them where they concerned systems that the person who discovered them was ill equipped to fix.

5.2.2 Meetings

In the earliest stages of the project we met in-person as a whole team to brainstorm ideas. Team members would take turns pitching ideas allowing us to fully explore the possibilities time travel presented. Many of us had strong visions on the direction of the game but through these discussions we were able to come to agreements and align our vision. Throughout the project we also held brief, regular meetings almost every day at 10AM via Microsoft Teams. These daily scrum meetings involved each person going round in a circle, stating what they achieved the day before, what they intended to achieve that day and any blockers they need help resolving. Although the whole team wasn't always able to make these meetings, it provided a great way to keep up to date with the macro state of our game while encouraging ourselves to keep progress moving. Occasionally these meetings would be longer, particularly when we needed to align any new ideas or create action plans following rounds of user testing and the ends of agile sprints. When a team member introduced a new feature that affected everyone else (such

as the event system) they gave a seminar on the changes made and how to utilise them.

5.2.3 Paired Programming

Working collaboratively in person facilitated team members programming in pairs or even small teams when features became too difficult for a single person to manage. This also provided a level of oversight to the complexity of systems as team members were required to articulate their feature to other team members.

5.3 Agile Sprint

Given the ever evolving nature of our project, agile sprint work methodology was an obvious choice. Shorter meetings would be held each day to check the progress of already assigned tasks and assign new ones. At the end of a sprint a larger meeting would be held to discuss future development and planning.

5.4 Learning Collaborative Development

During the pandemic collaborative working saw a major upheaval however computer science was uniquely able to weather this due to our already extensively using online tools to coordinate. The larger problem was dragging people back into the office to work collaboratively in person as we discovered team members were far more productive when working in person. If we were to repeat this project we would work collaboratively in person as much as possible, more so than we did this time, as this would have enhanced our productivity greatly.

5.5 User Feedback

User feedback was critical to the development of our project as we need to gauge how well people were able to comprehend time travel [section: 8.4]. Players would be given a link to play the game on their own device, team members would be on hand to answer any questions or resolve technical difficulties. Once they completed playing, testers would fill out an anonymised form on Microsoft teams so that any feedback would be objective. This feedback would then be discussed in team meetings and used to influence game direction.

Chapter 6

Individual Contributions

6.1 Henry Hartnoll

Weighted Contribution - 1

Individual Contributions:

Code and Development Management

- Set up version control.
- Researched multiple framework options for CI/CD. [section:7.1]
- Set up the GitHub actions for CI and testing with gameCI. [section:7.1]
- Researched best hosting platforms and set up google cloud hosting.
- Set up automatic deployment with github actions. [section:7.1]
- Changed the hosting platform to itch.io and set up automatic continuous deployment to work for itch with itch's proprietary command-line tool 'butler'. [section:7.1]

Time Travel

- Early on in the term I helped research the data structures to be implemented on the backend.
- Spent just under a week working on the head management on the new version of time-travel.
- Fixed some animation and visibility management problems. [section:8.2]

Test Framework

- Spent about a week setting up the testing framework on unity to allow edit-mode and play-mode testing.
- Wrote early test files for the RealityManager. [section:8.3]

Crystal-breaking

- Spent 2 weeks on the research and development of the potential second key technology.
- Implemented a system of recording the destruction of a crystal and then creating a new data structure that reads from the animation file containing the location of the shards at each frame. [section:8.15]

Misc

- Spent a few days working on traversal features such as jump pads and speed boosts.
- Implemented the feature that checks which team won the game and sends that to the UI designed by Anton.
- Updated the map to include time-dependent breaking crystals and jump pads.[section:8.15]

6.2 Zac Woodford

Weighted Contribution - 1

Individual Contributions:

Management

- Planning and management: Organizing and chairing meetings. Managing the Kanban board [Sections 5.2.2, 5.2.1].
- Conceptualisation. Engaging with the team and users to plan how the core time travel mechanic and game objectives would work [Section 8.4].
- Bug Fixing and Asset cleanup. Fixing bugs within my own work, removing excess and redundant assets, culling redundant branches.

Aesthetics

- Concept art and mood board: Assembling a mood board of images for inspiring the environment and character design. Creating concept art from mood board.
- modeling: creating models from the concept art and mood board [Section 8.6].
- level design: I performed all 3D level design [section: 8.5].
- shader: I created the Cel-shaded shader and all it's varients [Section 8.7].
- post processing: I applied all post processing effects [Section 8.8].
- character model rigging: I modeled and Rigged both character designs [Section **??**].
- animations: I created the initial Miner animations [Section 8.13].
- special effects: I created the sun position change system, the growing crystal system and the animated post processing effects system [Sections 8.8, 8.15].

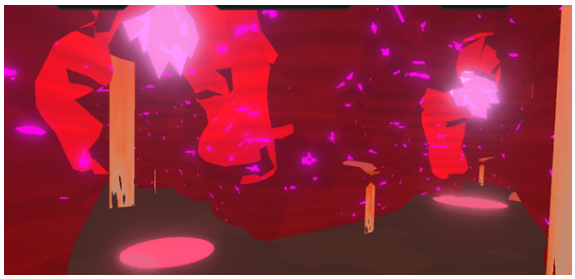


Figure 6.1: Every aspect of this image except for the falling sand was my own work

Figure 6.2: Concept art for the guardian character

6.3 Nisa Bayraktar

Weighted Contribution - 1

Individual Contributions:

Frontend(Player HUD)

- Creating unique tutorial states for both teams in tutorial [section:8.12]
- Helping implementing the tutorial animation and states[section:8.13.3]
- Integrating keyboard icons designed by Anton into the HUD as sprite to make the key controls clear [section:8.9]

3D Asset Design

- Designing canyon rocks and miner character's tracker device using Maya and ZBrush [section:8.6]
- Functions to change the tracker's compass needle position and button colour

Animations

- Character Animations[section:8.13.1]
- Adjusting the Player prefab that we have by adding both miner and guardian characters meshes.
- Dividing and updating some scripts that are attached to the Player Prefab into miner and guardian game objects to control their individual animation.
- Changed rePlayer Prefab (ghost mode) by adding both characters meshes.
- Created guardian animations[section:8.13.1]
- Integrated both guardian and miner characters animations into Unity's Animator graph.[section:8.13]
- Controlled the transitions between the animations in a script[section:8.13.1]
- Synchronised the characters' animations over the network[section:8.13.1]

Shaders

- Made falling sand and character dissolve shaders to improve the visualisation of the time travel using the URP shader graph[section:8.7.1]
- Controlling sand shader's rotation and speed in the code to change the values depending on the time travel direction[section:8.7.1]
- Animating dissolve shader in the code and helped integrate to the time travel code(Paolo's implementation)[section:8.13.2]

Particle Effects

- Dropping crystal effect for the Miners.[section:8.14]
- Fixing bugs in the parts that I was working on.

6.4 Anton Wallstedt

Weighted Contribution - 1

Individual Contributions:

Frontend Development [8.10]

- Menus: Dynamically list rooms available, Dynamically list players inside of a room, With the master client having the option to allocate teams and icons to each player that then get synced across the network and instantiates the timeline on each instance to reflect these selections
- Player HUD: Timeline that fills up as the game time progresses, player icons that are dynamically instantiated on the timeline, to illustrate each player's position in time, Each icon correspond to the selected icon by the master client in the room menu, If it's your own icon, it gets scaled up and has a white outline so it's easily discernible from other player's icons.
- Win Screen: Displays team won, Dynamically instantiates player statistics for each team

2D Asset Design [8.9]

- Designed all 2D assets used in our game from scratch
- Menu assets (buttons and stylised text for our game title)
- Player icons, in multiple colours with illustration of which team it belongs to
- Player HUD: Toolbar at the top, displaying game time, miners grabbed or crystals collected and which team you are in, Timeline at the bottom, Travel buttons, with a swirling portal design in orange and blue, corresponding to the unified colour coordination in our game with respect to time travel. I.e. orange indicates the past, and blue indicates the future, They are also animated to spin to reflect when the abilities are ready to be used (and stop spinning when the cooldown is activated).
- Tutorial assets: Pointing arrows with a futuristic hologram style, Keyboard keys and mouse icons in pressed and unpressed states
- Win Screen Designed our team logo displayed on the loading screen, Designed our game logo and animated it to loop seamlessly

Music, SFX and Ambience

- Coordinated with our composer to get the music we desired
- Implemented sound effects: Ambient background sound, Footsteps and jump SFX, Collectible crystal SFX, Button click SFX
- Implemented a skybox to match our game style

Miscellaneous

- Took meeting minutes on every meeting with Tilo and George
- Helped out with bug fixing
- Helped maintain coding conventions across scripts

6.5 Angus Robertson

Weighted Contribution - 1

Individual Contributions:

Game Mechanics

- Character controller and input handling
- Photon Unity Networking (PUN) [section: 7.1]:
  - Photon Unity Network Library import
  - Synchronising user movement over network
  - synchronising animations over network
  - RPC calls for more complex synchronisation
- Scene transition synchronisation with Master Client privileges
- Initial design/functionality for UI
- Timer support
- Primary integration of Time Travel back-end into multiplayer context
- Implementation of team mechanics with win conditions
- Scoring system with synchronised team scores
- Assets created to flesh out map scene

Assets/Aesthetics

- Assets created to flesh out map scene
- Map design conceptualising stages
- Collectable Crystals [section: 8.15)
  - Asset creation for collectable crystals
  - Overlay alpha cut shader with animation support
  - Collection of crystals for miners with collisions
  - Individual player scoring system
  - Player score updates Team score using RPCs
  - Crystal respawn mechanics with synchronised network timing
  - Crystal collection synchronised over network
  - Player to player interactions to increase/decrease crystal scores

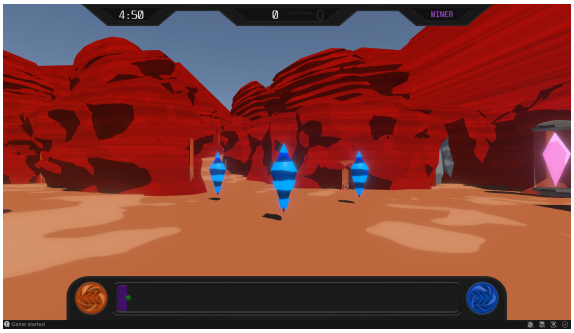


Figure 6.3: Crystals within Game scene

6.6 Paolo Mura

Weighted Contribution - 1

Individual Contributions:

Previsualisation (section 8.15)

- Planning and logistics.
- Directing and filming
- Editing in post production.

Implemented Minor Features

- Particle effects for time travel. (section 8.14)
- Item name tags.
- HUD debug panel interface. (section 8.3)

Time Travel System 2.0 (section 8.2)

- Overhauled the original time travel replay system.
- Simplified the underlying data structures.
- Split functionality across scripts.
- Fixed all the major bugs with version 1.0 in the process.

Test Framework and Debugging (section 8.3)

- Wrote the unit tests for the player states and rewrote those for reality manager to be more comprehensive.
- Created the TimeLord Simulator.
- Created a logging and visualisation system for Player States.

Lead Code Curator (section 7.3)

- Constant refactoring of the code base.
- Introduced a consistent style (following the C# convention).
- Split functionality into (mostly) independent scripts.
- Created an event system to keep track of game state.
- Maintained our documentation
- Created UML diagrams.

Feature Implementation Guides

- Put together the guides which formed the basis of how the following features were implemented:
- Using a state transition model in tutorial.
- Using animations and events in tutorial.
- Collectable crystals (synchronisation across multiplayer and integration with the time travel system).
- Breaking crystal obstacles (integration with the time travel system).
- Tracker device (defined the simple algorithms and suggested progressions).

6.7 Samuel Balson

Weighted Contribution - 1

Individual Contributions:

Time Travel System 1.0

- Data Structure
  - Recording of player states
  - Synchronisation of data creation
  - Custom data structure to easily recall data
- Realities
  - Created a system to separate each player's view of time
  - Each player able to be physically in same space while seeing different times
- Interoperability
  - Exposing data about player's temporal position for timeline
  - Connecting to player controls to allow for user driven time travel

Shaders

- Created more accurate specular reflection shader
- Shader to imitate retro-reflective surfaces

Camera Movements

- System to temporarily remove player from their view to perform baked movements
- Ability to perform a series of predefined camera movements consecutively
- Option to use bezier curves to calculate position of camera during movement for more fluid motion

Tracking

- Finds the closest collectable to the player
- Calculates direction along ground to point towards
- Checks difference between player time and collectable time to tell the player to travel

Miscellaneous

- Fixed various errors in indexing during tutorial
- Fixed various menu errors

Chapter 7

Software, Tools & Development

Throughout the development of Time Mine, the team needed to make sure that we were able to add new functionality to the game with ease and also ensure that those changes would integrate with the main game error free. This required us to put in place a framework in which we could robustly test and review the changes that members of the team made.

In this section we will discuss the 3 core factors that enabled us to efficiently update and maintain our game. The first section will be the 3rd party software, tools and frameworks we used to automatically test and deploy our game. The next will be the modular systems we created that afforded us easy creation and management of new features. And finally the refactoring and code maintenance that we carried out.

7.1 Software and Tools

Development Environment:

During the first few weeks of the project a specialty team spent time researching and setting up the framework required for efficient collaborative work. This first meant setting up an easy to use version control system that the whole team would be able to work on.

We decided that we would choose to use Git as our version control due to the familiarity that we all had with the system already. We then decided to use github as our centralised version control host. Using Github would allow us to use *Github Actions* which enables easy CI/CD to automatically build and test our project whenever anyone makes a pull request.

Setting up Github actions meant creating a YAML file that includes a trigger for whenever anyone creates a pull request, and whenever a new version of the game was released on main.

Pull requests would trigger a testing process that used a command-line tool named GameCI. GameCI would look inside the unity project for the test assembly that includes both edit-mode and play-mode tests and then runs the tests and outputs the results, an example of this is shown below.

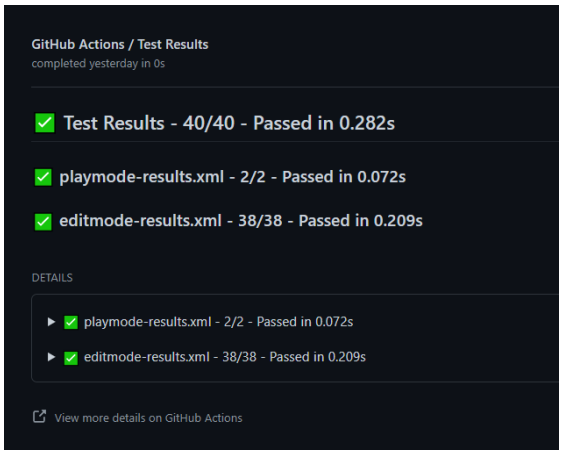


Figure 7.1: Github Actions GameCI Output

If all tests passed and it was able to build then the pull request would be merged. This streamlined our development greatly.

For hosting and deploying our game, we originally decided to host on google cloud because at the start of the project we were keen on potentially using the cloud GPUs for processing particle effects. But after we didn't go ahead with this idea, we decided that a more user friendly hosting platform would be Itch.io as this also met the requirement of password protecting our game.

When deploying our game with Github Actions we needed to select an adequate trigger. We decided that the game should only be built and deployed whenever we manually released a new version of the project on Github. This meant that we would be saving easy to recover snapshots of our game that we could rollback to in case we discovered a major error in a future version of the game.

Once the workflow had been triggered, we would first need to build the game files. We again used GameCI to do this task. Then we used Itch.io's own proprietary command-line tool: Butler. Butler can be called within the YAML file to automatically upload the build

files onto our project page on Itch. This meant we never had to worry about building and uploading the files ourselves and we could carry on working while the game built.

Making the Game:

The game itself was built using Unity, a 3D cross-platform game development engine, which is able to build to WebGL allowing the game to be deployed online in browser, one of the requirements of the project brief.

Unity has many features that made it ideal for our project, including the event system, the fact that it uses C# and so allows for object oriented development, and has several useful features surrounding lighting and post processing that were in line with our intended vision for the game.

For the multiplayer, distributed element of the game, we used Photon Unity Networking (PUN), which handles network connections and, with some implementation, can be allow for multiple clients synchronising their movements across different devices. PUN was ideal, because the library already exists in a unity-compatible format, and can also handle unity animations, which is crucial for a smooth-looking multiplayer game.

PUN's RPC calls also allow for quite heavily configurable aspects of the distributed game. For example, player-to-player interactions such as the guardians grabbing the miners, allowed for score manipulation to be straight forward and deterministic, while also creating maintainable code.

7.2 Modularity

7.2.1 Asset Design

Our assets team worked to create 3D models that could be easily recombined into different formations. In particular, our canyon level is built from just a few components (five walls, two arches, two columns, etc.). This allowed us to rapidly iterate over multiple different level designs and run them through rounds of user testing to determine an optimal final map design.

7.2.2 Game Objects

We also made use of Unity's prefabs to store templates of game objects that we could reuse. Obviously this applies for objects that would need to be instantiated (like collectable crystals). However, we went one step further and also utilised Unity's prefab variant feature, allowing us to create different types of objects from a base object with common shared children and components. This was most useful when working on characters, where miners and guardians, players and NPCs could all be constructed from the same base objects.

7.2.3 Code

We used object oriented techniques to add modularity to our code as well. Through key principles like inheritance and encapsulation, we were able to write our scripts in such as way that the same functionality could be applied to different objects/scenes or extended/overridden when needed. For example, we have an abstract *SceneController* script that tracks the state of the scene, such as score and teams. We then extended the classes *PregameController* and *GameController* from this parent class to add specific behaviour to each scene.

7.3 Maintenance

7.3.1 Best Practices

We aimed to employ best practices throughout our project. These included:

- Directory structure
- Style conventions
- Script separation

All our content, from assets to scripts, was kept in a sensibly-named file structure. It constantly evolved as new files were added so that we could easily find what we needed (both our own and other team members' work).

Within the first few days of development we unanimously agreed to follow the C# style convention (such as naming identifiers) in our scripts.

As our code base grew and game components became increasingly complex, we adopted a policy where we aimed to separate components as far as possible to increase their independence. To achieve this, we used encapsulation and an event system. This massively helped with bug fixing and code readability due to increased simplicity and reduced dependencies.

We enforced all these policies through the following process. First, we introduced and agreed on the policy in a group seminar. Throughout development, we held each other accountable via code reviews. Finally, we employed heavy refactoring to align our code each time a new policy was introduced.

7.3.2 CI/CD Workflow

In the very early stages of development we agreed on the following CI/CD workflow.

1. **Makes changes.** Add functionality or fix bugs on a new branch.

2. **Local game play tests.** Play a walk through of the game, testing that the feature works as intended and has no obvious unintended side effects. This must be done both single player and multiplayer.

3. **Pull request.** Push the changes in regular commits and when ready, make a pull request on GitHub.

4. **Code review.** Ask a colleague to review changes. This involves scanning through the code for style convention adherence and a quick play through in case they can spot any bugs that slipped through.

5. **Automated CI/CD tests.** Run edit mode tests via GitHub Actions.

6. **Merge.** If there are no compile errors, all tests and code review pass, merge into the main branch.

As a team, we made good efforts to stick to this workflow as far as possible. It was a fantastic way to keep the code maintainable and reduce the number of moderate-to-critical level bugs that ever entered the build.

7.3.3 Documentation

Due to the complexity of the core system of our game extensive documentation was required so that team members who weren't developing time travel would be able to use it's infrastructure. Advanced features would be given a dedicated file to describe their implementation and outward facing functions as well as a diagram to showcase their functionality. The whole project was also documented in a series of UML diagrams that show the relationships between components. A script diagram was also created to show the relationships between scripts and the game objects they control.

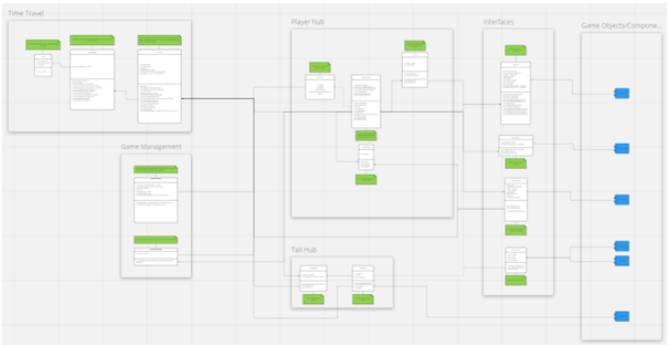


Figure 7.2: Project UML diagram overview

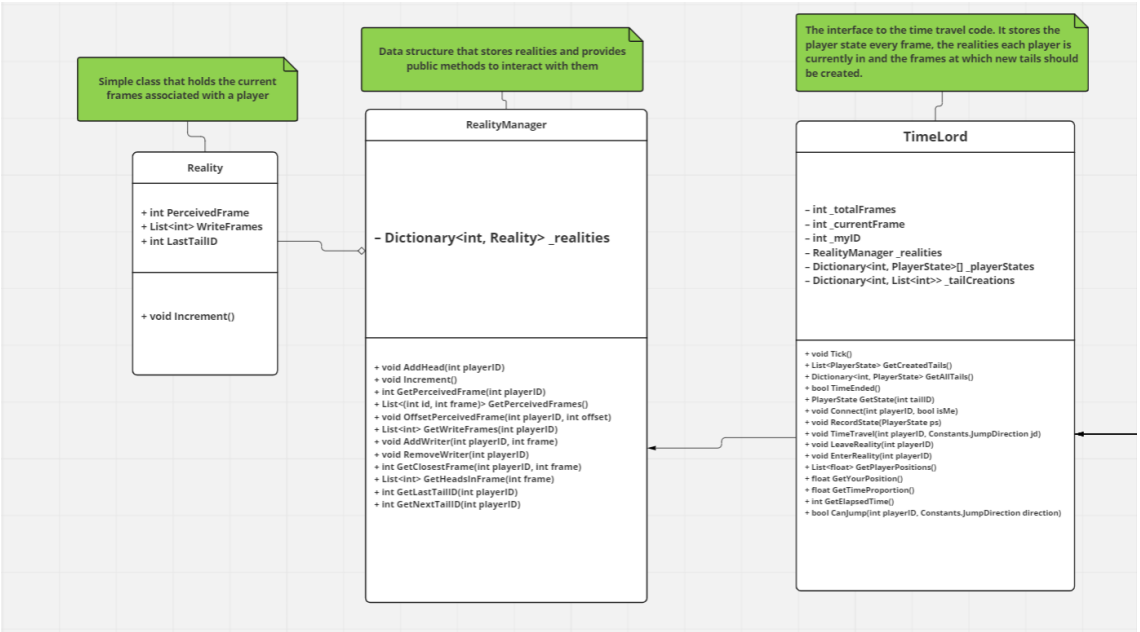


Figure 7.3: Project UML diagram zoomed in.

Chapter 8

Technical Content

8.1 Core Functionality

8.1.1 Character Controller

The Game runs as a First Person, mobility and fast paced problem solving game in that the user's need to either escape their opponents, or catch them to succeed. This required a responsive first person controller as well as a system that was easily configurable to our purposes. Therefore the character controller was built from the ground up, using only unity's collision system. As the game runs in WebGL, the user inputs are obtained from the keyboard and so unity's input system gave the most efficient and responsive implementation.

8.1.2 Game State

The Miner's objective in the game is to collect as many crystals as possible, their team will win if at the end of the game they have more than the stated number of crystals.

The Guardian's objective in the game is to take all of the crystals from the miners, by catching them. They will win if the miner's score total is below the threshold.

The scoring system in the game works as follows:

- If a Miner collects a crystal they gain 1 point
- If a Miner is caught by a guardian they lose 5 points
- Score cannot drop below 0
- Miner's points are pooled as a team to reach their objective
- The Miners must have more than the stated number of points at the end of the game to win

Points earned are synchronised over the network using Photon, and the clients all keep track of how many crystals the Miner team has, but not individual scores.

8.2 Time Travel System

8.2.1 Overview

Although the final realisation of our *Time Travel System (TTS)* took time to settle, the crux of the system has always revolved around the very first idea pitched:

1. Players can freely travel back and forth in the timeline of the game. We define this as "fluid" time travel.
2. When revisiting the past, players experience a replay of events occurring around them.

The grandfather paradox drove us to our final implementation. In essence it states that any change in the past will have a ripple effect on the future. If we tried to simulate the resolution of events, this could be highly computationally expensive and technically challenging beyond reason. If we neglected this process, it may be jarring and unsatisfying for players to experience causes without effects. The solution we landed on was to disallow interaction with the past entirely.

More specifically, our rule states that players may not interact with the environment or past versions of themselves. This freed us from the grandfather paradox but introduced a new problem: what is the point of time travelling if not to change the past?

After months of testing, discussions and a focus group, we landed on the final solution. Collectable items would appear at specific times throughout the game. Once collected, they are removed from the game entirely (i.e. cannot be recollected by returning to the past). Obstacle crystals would grow/break at specific times in the game, rendering certain parts of the map accessible at only set time windows in the game. Together, these provided an incentive to time travel while simultaneously adhering to our rule.

This has all culminated in what is, to the best of our knowledge, the world's first and only multiplayer, fluid time travel game.

8.2.2 Capturing State

One requirement of our TTS was that when travelling to the past, players should see their past selves repeating their original actions. To achieve this, we implemented a replay system similar to that of existing games. It involves capturing the state of all players every frame and storing these states in a data structure that can be read from when replaying events.

The data structure is an array indexed by the frames of the game. Each item of the array is a dictionary that maps IDs to state. Overall this provides constant-time storage and lookup of states.

The state being stored is a simple *PlayerState* object that contains values such as position and rotation.

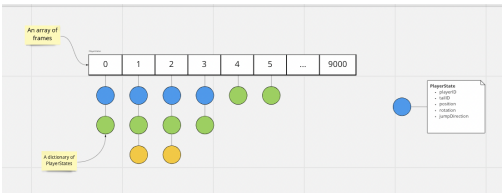


Figure 8.1: Player states data structure

8.2.3 Tracking Players

A second data structure is required for keeping track of each player's position in time. We implemented a *RealityManager* class to encapsulate this. It stores a dictionary that maps each player's ID to their *Reality*, a container for the player's *perceived frame* and *write frames*.

The *perceived frame* is simply the frame that that player is currently experiencing in the game. The *write frames* are the frames they are writing to in the player states data structure. The reason these things are separate is because although you may perceive yourself to be travelling backwards through time, other players will see you disappear in the time you just left or reappear in the time you arrive at. In certain situations there will be overlap, hence the need for keeping track of these frames as well.

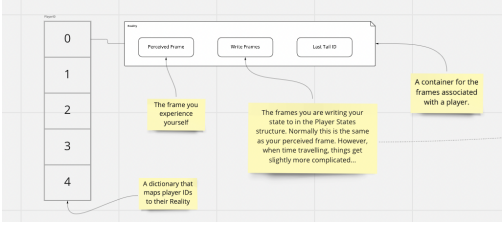


Figure 8.2: Reality manager.

8.2.4 Replay System

To differentiate between a player and their replays, we referred to them as a *head* and *tails* respectively throughout development.

Every update, a script attached to your player accesses the player states data structure at your perceived frame. If state belonging to a new ID is found there, it instantiates a new *Tail* game object.

The *Tail* is a translucent version of the character that originally stored the corresponding state. Every update, it retrieves its state from the data structure and sets its attributes accordingly. The moment it looks itself up but can't find anything, it destroys itself.

8.2.5 Player-TTS Interaction

The player interaction with the TTS is arguably one of the most complex parts of our project. It deals with many interactions split across multiple scripts where timing is critical. Without diving into the specifics of each interaction, a summary of its achievements are as follows:

- Store the player's state every update.
- Validate whether time travel is possible given several criteria (cool downs, game state, position in elapsed time, etc.).
- Trigger time jumping (both intentional and by the "Weeping Angel" attack).
- Trigger effects (dissolve and particles).
- Set player visibility and whether you can interact with them based on who exists in your perceived frame.
- Snap to the closest player's frame on arrival or an unoccupied frame.

8.2.6 TTS Interface

The interface for how most game objects interact with the TTS is very simple. For most practical applications, objects are able to make use of the TTS with just two lines of code (one to access the TTS and another to get your perceived frame for example). Aside from players, the objects that make use of the TTS through this interface include collectable crystals, obstacle crystals and the sun position as mentioned in section 8.15.

8.2.7 Multiplayer Synchronisation

Each client runs their own version of the game locally. Only a small handful of things are synchronised over the network such as animations and player positions. This is done using the *Photon Unity Networking (PUN 2)* package.

Rather than storing the states centrally on a server or master client, each client runs the TTS almost entirely independently. This reduces network delays, keeping the game running smoothly for each client. Although we enforce a

30 FPS frame rate for all players, lag can still occur, leaving different clients' record of perceived frames wildly different.

To mitigate this problem, we introduced an additional level of synchronisation between all clients' TTS. The master client sends out a remote procedure call via PUN 2 every ten frames with a copy of their *RealityManager* state. Each client then overrides their own *RealityManager* with the given values. This does a fantastic job in synchronising perceived frames, so every client experiences the same game with minimal delay.

A second step is then required to balance the player states that are stored. This prevents problems like gaps opening up in the state data structure or showing tails for your current actions.

8.3 Test Framework

8.3.1 HUD Debug Panel

The HUD displays information to players throughout the game. We also added a feature to it in the form of the Debug Panel, which can be toggled on and off by pressing the 'P' key during the game. Toggling it simply displays a panel of debug values on the left side of the HUD.

We used this feature heavily during development to display variable state without generating logging output. Any script in the game can display its variables simply by implementing the *Debuggable* interface and its single *GetDebugValues* method.

8.3.2 Unit Tests

Our test framework includes Unity Edit Mode tests. These are unit tests that can be run automatically during our CI/CD cycle. Since the TTS is crucial to our game, we made heavy use of these tests to thoroughly test every function involved in the backend. This helped identify and patch several small bugs in the logic early on.

8.3.3 Time Simulator

In order to fully test and debug the TTS backend, we made sure to completely sever its ties from Unity, such that it only made use of core C# functionality. This led rise to the *TimeSimulator* class, which is able to simulate a game of time travelling in the backend alone, without ever running a game in Unity. With a simple interface, a time travel simulation could be set up as follows:

```
1 TimeSimulator sim = new TimeSimulator(...);
2 sim.AddJump(...);
3 sim.AddJump(...);
  ...
8 sim.AddJump(...);
9 sim.Run();
```

The *TimeSimulator* constructor takes in parameters for initial setup (such as simulation length, number of players, etc.). You can then use the *AddJump* method to queue when players will make a time jump in the game. Finally, calling *Run* will execute the simulation, dequeuing and performing each jump.

We made use of this powerful tool to perform more advanced unit tests on basic simulations of the TTS by running assertions to check if the data structures were manipulated as intended.

8.3.4 Logging and Diagnostics

One additional tool we used for debugging was the *ProxyTimeLord* class. As the name suggests, it makes use of the proxy design pattern to create a wrapper for the TTS backend. In other words, it provides all the same functionality as the underlying class but if certain flags are set on instantiation, it will also run logging and/or diagnostics.

With the logging flag set, the state of the TTS data structures are captured and logged during and after the game. Using a custom SDL script, we were able to visualise the logging output. This helped us to reason about and correct tail behaviour.

With the diagnostics flag set, summary information is logged about the game. In particular we used this feature when working on the multiplayer synchronisation of the TTS to record and summarise delay between clients.

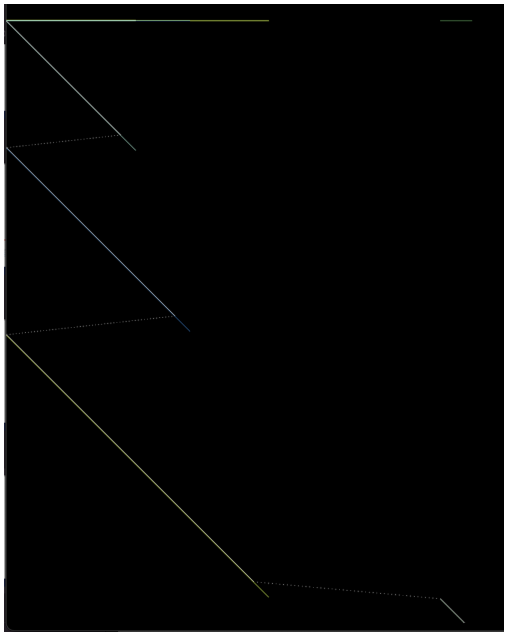


Figure 8.3: Generated logging visualisation

8.4 Communicative Mechanics

Introduction

The implementation of time travel in our game presented two challenges, first the technical implementation and second, plability. Most people are familiar with time travel as a concept but have never been able to actually experience it and as such they lack a sufficient mental framework for comprehending it. This was made obvious when we pitched our initial time travel concepts to focus groups [section: 5.5]. As such we needed to build our game in a way that would facilitate players forming a robust conceptualisation of time travel so that they could navigate through time and even begin to utilise time travel for problem solving. To this end we adopted a development model whereby we would continually iterate and improve systems based on user feedback and comprehension [section: 5.5]. An early product of this methodology were several principles that all further time travel mechanics were built on:

1. Players must always know where in time they are
2. Players must be encouraged to time travel as much as possible
3. Players must always know when they themselves are traveling through time
4. Players must know which direction through time they should travel
5. Players must know when other players are traveling through time

These principles remained unchanged throughout the project. This meant we could change other elements of the game, such as aesthetic and player objective, while maintaining an framework for playability.

What is Communicative mechanics

The concept of communicative mechanics is to communicate time travel to the player via the mechanics of the game. These mechanics were designed in accordance with the previously stated principles:

1. Aspects of the game world must change over time [Principle: 1]
2. Players have an objective that forces them to time travel [Principle: 2]
3. Players should have to time travel to navigate around obstacles [Principle: 2]
4. Elements of the game world should change as the player travels through time [Principle: 3]
5. Players should be explicitly shown their position in time and where they are traveling in time [Principles: 1, 3]
6. Directions in time must have clear iconography, backward is orange, forward is blue [principle: 4]
7. Players should have an overlay effect to indicate that they are traveling through time and show which direction in time they're traveling. [principles: 3 5]

These mechanics changed and evolved throughout development based on user feedback

Features

From these mechanics we developed several features, features being physical implementations of the mechanics. Each of these features are detailed in the following sections. Most of these mechanics require visual communication to the player thus requiring advanced graphical fidelity.. The application of these mechanics is best exemplified in the *time dynamic objects*, these are objects that change with time, either switching between between two states at a certain point in time[blocker crystals, collectable crystals 8.15] or continuously changing with each frame [sun position, 8.7.1].

8.5 Level Design

Like all ancillary features of our game, the process of level design was conducted iteratively based on user feedback and within the guidelines provided by the communicative mechanics [section: 8.4]. Through this process a series of level design principles were adopted: "a time dynamic object shall always be within player vision" [principles: 1,3,4] , "No dead ends" otherwise hiders may become trapped, "Small lines of sight" so that hiders can easily evade pursuers, "lots of junctions" so that hiders can easily evade pursuers, "obviously varied regions" to act as landmarks helping players navigate the map. Furthermore blocking crystals had to be accounted for when considering flow of play and accessibility. This posed a unique challenge and require allot of iteration to perfect. Figure 8.5 illustrates the results.

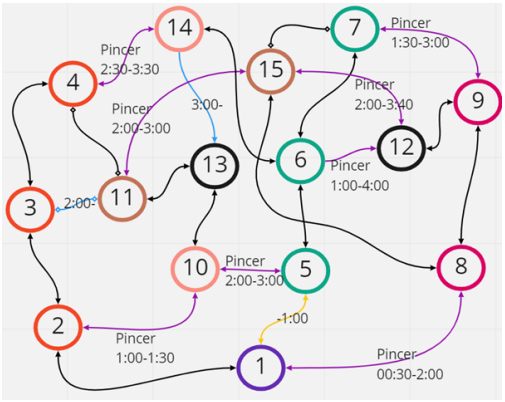


Figure 8.4: Graph of map junctions. Edges show pathways between junctions, numbers on edges show the time period through which the edge is traversable

Lastly, level design offers an opportunity to convey narrative through the environment building on aspects: 7, 8. This was done through the form of 'mining junk' positioned around the map which also serves to make areas of the map unique in accordance with the aforementioned level design principle.

8.6 Modelling 3D assets

This is the section on 3D modelling In order to enhance aspects 7 and 8 we created a library of original 3D assets. Due to the constraint of having to run our game online we had to limit the number and size of these assets, however this lead to us creating universal assets that could be reused. As an example, there are only five different terrain wall pieces. Because textures weren't used, the 3D models in many cases were split into different sections which could be given different materials. The terrain meshes were converted into prefabs before being used in the scenes as this enhanced asset reusablity and modularity. It meant that several different maps could be constructed from the prefab pieces, like Lego. This enabled much easier iteration of level design [section: 8.5] The character models were more complicated and required reference images. Figures below show the concept art for the guardian character as well as the final 3D model. We also avoided modeling faces as these are hard to get right.

Figure 8.5: Concept art for the guardian character

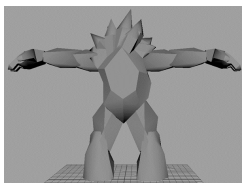


Figure 8.6: 3D model of the guardian character

8.7 Aesthetics

In line with the requirement for advanced graphical fidelity [Section 8.4] we utilised the *Universal Render Pipeline(URP)* as this facilitates the easy enhancement of graphics. *URP* targets any platform and provides a simpler way to create custom effects and graphics using its features such as the *Shader Graph* and *Post-Processing* effects. This meant that the games look and feel [aspect: 7] could very easily be enhanced. We elected to utilise a custom designed cel-shaded aesthetic to increase aspect: 8. A cel-shaded aesthetic tries to emulate a drawn cartoon style by replacing smoothed shadows and specular highlights with solid fill areas, figure 8.7 illustrates this:

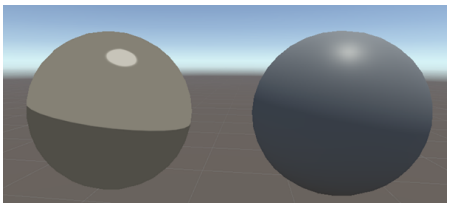


Figure 8.7: Toon shader on the left, standard Lambert shader on the right

The aim of this aesthetic style was to also make the game less intimidating as being chased through time by monster made of stone could be quite scary with a more photo-realistic style. It also received extremely positive feedback from play-testers. The Cel-shader was significantly difficult to implement in the *URP Shader Graph* as it required circumventing the limitations of the *shader graph* via the use of a custom hlsl script. The components used to calculate the cel-shaded lighting were modularised enabling several shaders of similar style to be built quickly and easily. An example of this is the Strata shader used to colour the map terrain pieces wherein the cel-shaded modules where applied ontop of layers of noise to create a toon style rock material without using textures.

8.7.1 Dynamic Shaders

Dynamic shaders apply dynamic effects to their materials. Two such shaders were created for our game, the falling sand and dissolve. The falling sand shader was built for mechanics 1, 4 and uses game time to generate random noise which creates both the texture and holes in the material. However the time can be scaled through script, this allows for the reverse falling effect during time travel. It is also thematically relevant to both the desert mesa theme and the time theme as falling sand has historically been used to measure time.

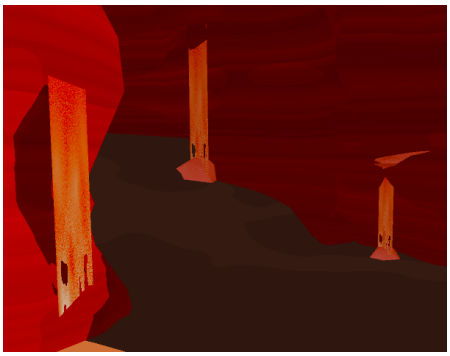


Figure 8.8: Falling Sand

The dissolve shader is built for mechanic 7 as it visualises the character's disappearance and reappearance through time, thus showing other players when a player is time traveling, it also enhances aspect 7.

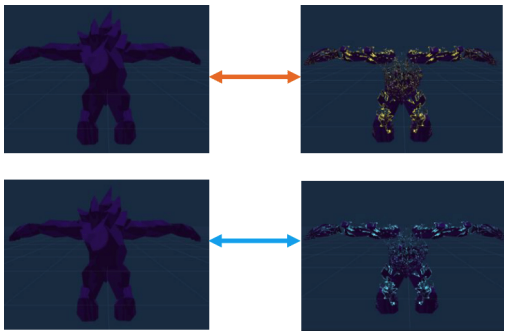


Figure 8.9: Dissolve Effect

To synchronise both shaders with the time travel mechanics, we turned the input nodes of the *Shader Graphs* into a property which allows us to adjust the values of the nodes inside the material. For the falling sand behaviour, we changed the sand's speed and rotation properties in the script depending on the time travel direction. Whereas we applied the edge height and colour properties of the dissolve effect into the animations and linked the animations with the time travel mechanic's code.

8.8 Post Processing Effects

As part of enhancing aspect 7 of our game we utilised post processing effects in several capacities. The first was to generally enhance the aesthetic by altering colour balance, lifting shadows, tone mapping and applying bloom. The Bloom effect was carefully calibrated to make the crystals glow. In the cave section of the map a separate post processing volume was applied to lift the shadows and brightness so that the player can see and navigate more easily.

Dynamic effects

Dynamic post processing effects are applied to the players camera when they time travel, giving them an overlay to indicate that they are time traveling and also the direction they are traveling in, this links to mechanics 5,6,7. Because of the complexity of the time travel system it was simpler to create a separate animation system for the post processing effects using co-routines which are routines that run asynchronously to the system they were deployed from. This necessitated the use of locks so that only one post processing animation is run at a time and a secondary animation can be queued while another one is running. The post processing effects applied to the player while they are time traveling are lens distortion, chromatic aberration and a vignette coloured to the direction they are traveling in line with mechanic 6.

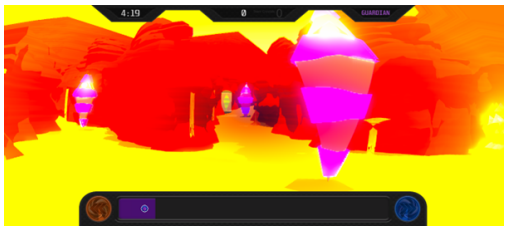


Figure 8.10: Shown are the effects applied when the player travels backwards in time

8.9 2D Assets

When designing the 2D assets, the main consideration was the feel they would provide to our players. With the core of our game being time travel, the 2D assets had to tie in with this fact and feature a futuristic design. The assets can be divided into two parts: (1) menu assets and (2) HUD assets.

Firstly, the menu assets needed to be minimalist and easy on the eye. We do not want to draw attention away from the time travelling mechanic, which is complicated to understand as is. The menu should simply be the interim before entering the game, not much time should be spent trying to understand it.

Secondly, the HUD assets needed to provide as much information as possible, without drawing attention away from the game. This led to a slim toolbar at the top displaying necessary game information, such as time left, how many crystals the miners have collected and

what team a player is in. There is a panel on the right which shows the key controls of the game. At the bottom there is the timeline, which needs to be clutter-free and display information relevant to the time travelling. It features a bar that fills up as the game progresses and player icons that display what team a player is in, and their associated colour. Your own icon is scaled up and outlined in white. A quick glance at the timeline therefore provides the user with their position in time relative to the total elapsed game time, and in relation to other players. Finally, the timeline at the bottom also features two buttons that indicate the abilities of travelling backwards and forwards. Special attention was placed in colouring these, such that the colour of each button illustrates what direction in time they will take a player. That is, orange for travelling backwards and blue for travelling forwards. This colour theme is consistent in our game with regards to anything that concerns travelling in time.

To summarise, there is one main thread running through the design of the 2D assets: to provide as much information as possible with as little clutter as possible – we arrived at minimalism.

8.10 HUD

Aside from the design of these assets, some had to be fused with the back-end of our game to provide useful functionality. This includes the timeline, the player icons on the timeline and the travel buttons.

The time line is an essential aspect of the HUD as it tells players directly where they are in time and what direction they're traveling. It also tells players where everyone else is in time. [mechanic: 5]. The time bar is one of the oldest communicative mechanics [8.4] and was designed from very early user feedback. It was also part of a pre-visualisation we conducted at the start of development as a way to show people watching the video what was happening. The timeline is being updated by the time travel framework: it gets the current elapsed game, the position in time of each player icons, and if a player is able to travel backwards or forwards.

The timeline is updated to reflect the current elapsed game time relative to the total game time, and each of the player icons are updated to match the corresponding player's position in time. The travel buttons are notified when either of the time travelling abilities are ready, and start spinning to indicate that they can be used.

As the icons are selected in the menu scene by the master client, these selections need to be translated to the game scene, so the player icons on each game instance can be instantiated appropriately. This is managed via an RPC call that saves the selected player icons temporarily inside a player preference file, managed by Photon PUN 2. When the game scene is loaded, each icon can be set up with correct icons based on the preference file.

The two circle icons on either side of the time bar show the player when they can time travel and in what direction, to do this the icons swirl around. This swirling effect is designed to draw the users eye. When the time travel ability is on cool-down the icons are shown to slowly fill. At the top of the screen the player is shown how much game time has elapsed, how many crystals the miners have collected (and how many they need to collect) as well as the team the player is on (miner or guardian). This was all information that the players requested to know during user testing.

8.11 Menu Management

The menu scene contains some key functionality that is needed before the game starts. It dynamically displays any rooms available, and when inside a room it displays a list of players. This list is also dynamically updated whenever someone joins or leaves the room.

Each menu is managed from within one scene. They are empty game objects that hold each of their corresponding menu assets. Each menu object has a script attached to it, in which the name of the menu can be set. This allows for a single separate script, a menu manager, to open and close menus both by accessing the game objects but also by referring to their name. This improves readability and workflow, as a reference to a menu can be made via its name.

Moreover, inside of a room, the master client has the option of selecting player icons for each player. They also start the game. If the master client leaves, a new master client is selected. On the game instance of the new master client, each of the list items are updated to display the option of selecting an icon. The new master client also gets the option of starting the game.

Finally, the last menu presented to the player is the win screen. It retrieves statistics from each player, by referring to their Player Controller scripts. Then, Prefabs are instantiated in two containers corresponding to each team. These Prefabs display a player's nickname and the number of crystals collected or miners they grabbed, depending on which team they were in. The prefabs are ordered in descending order of their score.

8.12 Tutorial

During the user tests and panel meetings, most of the feedback that we got was about the complexity of our game. Therefore, we decided to implement a tutorial before our actual game scene to deliver understandable gameplay. In the tutorial, we explained key features in our game that are crucial in the gameplay.

In the implementation process first, we created a prefab which is like a corridor and duplicate it to provide a individual tutorial experience for each player. The corridor prefab contains the game objects that we use in our game such as growing crystals, collectable crystals, and breaking crystals.[mechanic:3],[section:8.15].These game objects are dynamic features of our game which appear at certain times and force the players to time travel[mechanic:2]. Thus, the advanced part of the tutorial implementation was timing each of these dynamic objects and synchronising them to show the player what exactly they look like and how they function.

At the start of the tutorial, we spawn each player in an individual corridor to make a specific tutorial for each player. To explain each feature, we start by showing each of our time dependant game objects simultaneously in the tutorial corridor. We utilised Unity's Animations[section:8.13] to record these movements and played this animation at the start of the tutorial. Secondly, we added states using the Event System and attached them to the animation to update the tutorial text depending on the game feature we are showing.

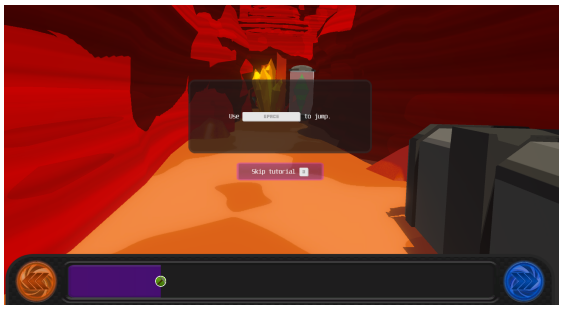


Figure 8.11: Screenshot of tutorial instruction.

Each team has their unique tutorial states and game objects. For instance, for the player Miner, we added a collectable crystal whereas for the player Guardian we put an NPC of the Miner character. Additionally, to make the features more straightforward we put nametags on the game objects that behave as hints to direct the player and inform them what they exactly need to do. Finally, we let the player go through the corridor and pass these individual objects that we introduce to experience the most accurate gameplay and familiarise themselves with the actual game.

8.13 Animations

Unity provides several ways to animate the game objects and visual effects, creating animations by scripting Animation Curves and using Animation Clips. Both methods use key-frames which are the points that store the properties such as positions and colours of the game object and effects. To create the animation, Unity calculates the values between the keys and changes them during the animation. The difference between the 2 animation techniques is the Animation Curves allow to smooth out the animations by making the interpolation between the keys adjustable.

8.13.1 Animating Characters

When animating our characters, we first created Animation Clips for all movements (jump, run, idle etc.) by adding each of the body parts that we want to animate as a key-frame. The smaller meshes of the characters helped us to move their individual body parts [section: 8.6]. Unity has an Animator Controller that behaves as a state machine. Each state in the Animator Controller represents specific Animation Clips, and the sequence of the clips is assisted by the connections between the states called transitions.

Secondly, we implemented all the Animation Clips into the Animator Controller. We added Boolean parameters between the transitions and make them conditional to swap the states with the player's key-press input. We used the same state machine for both characters to keep the same transitions because they have the same basic animations. However, just one animation clip can be assigned to the Animator Controller's states. Therefore, the animations of the Miner character could not be added to the main controller . To overcome this problem, we generated a Unity Animation Override for the Miner character which creates the same state machine of the main Animator Controller and overrides the clips inside the states during run-time.

Finally, we adapted these animations to multi-player mode via PUN2's Photon Animator View script. We attached the script to each character's prefab to synchronise animations over the network and make them visible to all the players instead of writing RPC calls.

8.13.2 Animating Dissolve Effect

When animating the dissolve effect, we created 2 Animation Curves, an in-curve for dissolving in (character's reappearance in time) and an out-curve for dissolving out (character's disappearance in time). We controlled their keys inside the script using an animation co-routine. Both curves have the same keys that represent the

dissolve shader's property values [section:8.7.1], but they set the opposite start and finish times to give them different dissolve directions. The animation co-routine increments the Animation Curves' time iteratively and gets the current time point of the curves, then, sets their key values in that time point. We added this script to each player's Prefab. To synchronise the dissolve effect with the time travel mechanic and adapt it to the multi-player mode, we called the animation co-routine inside the time travel code's RPC calls .

8.13.3 Tutorial Animation

The tutorial mode has a camera movement which goes over all the game objects that we used in the tutorial. [section:8.12]. We used Animations to record the camera movement changing the rotation and the position of the camera transform depending on the game object's directions. To emphasise each object and explain its roles, we created tutorial states. These states are generated in the script using Unity's Event System and have their distinct events. Each event triggers a function that sets the tutorial state and displays its text. Then, we attached the event to the Animation clip by adding the camera movement where the animation indicates the corresponding game object on the keyframe.

8.14 Particle Effects

Particle effects were utilised to strength the games look and feel [aspect: 7] as well as to implement mechanic 7. Two particle effects, time travel splash and circle effects are played in conjunction with the dissolve shader [section: 8.13.2] when a player time travels to indicate to other players that they are time traveling. A third particle effect is played when a miner is caught by a guardian and shows the miner loosing some of their crystals, thus clearly indicating the the guardian that their victim has been caught. Using particle effects allowed us to communicate dynamic visual information to the player without using animations, adding another layer of visual interest.

Figure 8.12: Particle effects on player time-travelling forward.

8.15 Time Dynamic features

The time dynamic features were a direct result of our work with communicative mechanics. They are designed to fulfill the mechanics described in that section [section: 8.4].

Breaking Crystals

In line with the crystal aesthetic and mechanics [1,3] we used crystals to block off certain areas of the map that would then break at a certain point in time allowing access to the area. This is also in line with mechanic [4] as the crystals break and reform as players travel through time.

Achieving this would require a physics simulation that could then appear to be reversed in case a player looking at the crystal decided to travel back in time. In order to achieve this we decided upon a method in which we would carry out the destruction simulation on an instance of the crystal prefab and then record the states of each of the crystal shards at each frame and write this onto a file. We would at the start of each game, have each shard access it's associated states file and parse the file into a list of tuples containing the frames and states for that shard. These data structures could then be used to access the position and rotation of each shard at any given frame of the animation. These would then be played out as the player traveled through time.

Growing Crystals

Growing Crystals were designed to be the opposite to breaking crystals in that they would prevent passage from a certain point in time onward. That point in time as well as their growth period is variable. On each update they look for the players observed frame and if it is greater than their trigger frame but less than their growth period the scale and position of the crystal is modulated by an animation curve. This of course relies on the infrastructure provided by time travel system [section: 8.2].

Sun Position

One of the ways we tell the time in the real world

is by the position of the sun in the sky. Implementing this into our game was a natural and intuitive way to convey the passage of time to the player. This was implemented by scaling the rotation of the directional light object between 90 and 0 degrees, proportional to the players position in game time. This changes the size of the shadows as the game progresses giving a clear indication of the players position in time via the environment [mechanic: 1].

Collectable Crystals

The collectable crystals were introduced at a later stage in the project as an objective for the hiders after receiving feedback that the hider gameplay was not engaging enough [section: 5.2.1]. They are directly related to mechanic [2] as players must travel through time to collect them. The collectable crystals spawn in the game after a few seconds, and each has it's own 'window' of time in which it exists, synchronised over the network. When a crystal is collected, it re-spawns after a short pause, at another randomly chosen time period, accessible by the players. After each collection, the length of the existence 'window' halves, down to a minimum. They too use the infrastructure provided by the time travel system [section: 8.2], along with a dedicated manager to synchronise their existence across devices.