Chrono Corp

A temporal logistics game

TUM Games Lab
Winter Semester 25/26

Anian Kalb
Árpád Horvárth
Lukas Jonsson
Miguel Trasobares

1. Formal Game Proposal

1.1. Game Description

Overview

Chrono Corp is a real-time simulation and strategy game about managing energy networks across layers of time. The player works as a technician for Chrono Corp, a futuristic time-travel agency that powers “time tourism” by stabilizing fragments of time known as Time Ripples.
Each minute of gameplay represents a full cycle: the current timeline duplicates. The new layer becomes the present, while the previous one persists as the past — still active, simulated, and influencing ongoing systems. These “time slices” accumulate over time, forming a web of moments that must remain stable.
The player’s challenge is to maintain energy equilibrium across this growing network. Energy flows both spatially between connected nodes in the same layer and temporally between corresponding points across layers.
As the network expands, the game becomes a balancing act of causality and complexity. Players must think in temporal depth, predicting how current and past configurations will echo through time.

Background / Story Context

Chrono Corp is set in a distant future where humanity has discovered Temporal Current, measured in Causal Kilowatts (CkW), a quantum phenomenon that allows energy transfer not only through space but also through time. This discovery gave rise to Chrono Corp, a company specializing in commercial “time tourism”. The agency enables customers to experience historical moments firsthand by sending them back to Time Ripples, fragments of the past artificially stabilized through Temporal Current.
The player is a Chrono Technician, responsible for maintaining the energy network that keeps these Time Ripples intact. Each ripple is a frozen moment in time that requires constant energy to remain stable. If energy flow is insufficient, a ripple collapses, causing a localized implosion of spacetime… and almost certainly resulting in termination of employment.
Driven by corporate greed, Chrono Corp continuously pushes expansion. More Time Ripples are created to meet growing demand, increasing the complexity of the player’s network. Feeding Temporal Current into these ripples has an unintended side effect: each ripple slowly rewinds in time, representing an earlier moment in its own timeline. As layers accumulate, the ripples stack into a nested structure of past moments that persist and interact.

Relation to the Course Theme

Chrono Corp is built around recursion and time travel, both mechanically and narratively. Recursion appears in the repeated duplication of the world state over time. Every in-game minute, the timeline creates a new layer representing the present while preserving the previous state as an active “past” layer. These past layers continue to run independently, influencing other layers through temporal connections.

The system repeatedly calls itself, creating self-similar structures that feed backwards. Managing these layers requires anticipating how changes in a past state will cascade through subsequent states.

The player’s role as a Chrono Technician is to stabilize Time Ripples. As layers accumulate, the timeline becomes a structure of overlapping moments, each interacting with others throughout the time slices. This structure represents the time travel aspect, where interventions in one layer affect connected layers both backward and forward as well as the management of energy across time.

Each decision is both immediate and ripples through the time layers: rerouting energy in one layer may solve a local problem but create unforeseen consequences elsewhere, requiring continuous adjustment.

Gameplay Overview

In Chrono Corp, the player manages a network of Time Ripples across multiple layers of time. The core gameplay revolves around ensuring a constant flow of Temporal Current to these ripples, keeping them stable while the timeline duplicates and grows.

Core Gameplay Loop:

1. **Connect nodes**: Players link Time Ripples and energy sources using conduits that carry Temporal Current.
2. **Manage energy flow**: Each ripple generates or consumes energy. Players must balance supply and demand across the network to prevent collapses.
3. **Monitor layers**: Every minute, the timeline duplicates. The new layer becomes the present, while past layers remain active and represents the game state from one minute ago. Players must consider how past actions affect the current network and anticipate how present decisions will influence future layers.
4. **Stabilize the network**: Changes in one layer can ripple to neighboring layers, causing issues in other layers. Players intervene to redirect energy and adjust the network topology to maintain overall stability.

Goals and Challenges

- Maintain a positive energy balance in all active Time Ripples and across layers.
- Optimize the network to handle an ever-increasing number of layers as the game progresses.
- Anticipate recursive consequences: changes in one layer may have effects elsewhere.

- Limited amount of conduits so they need to be used wisely and in a manner that maximizes their effectiveness

We draw inspiration from *Mini Metro* in its simple mechanic for creating transport line and simple lose condition.

Complexity and Strategy

As the number of active layers grows, players face increasing temporal and spatial complexity. Each new layer introduces additional nodes and dependencies, requiring planning and foresight. Success depends on recognizing patterns across layers, predicting how interventions will propagate, and adjusting strategies and the network topology. The challenge lies not just in balancing energy but in understanding how actions in one layer can affect all others.

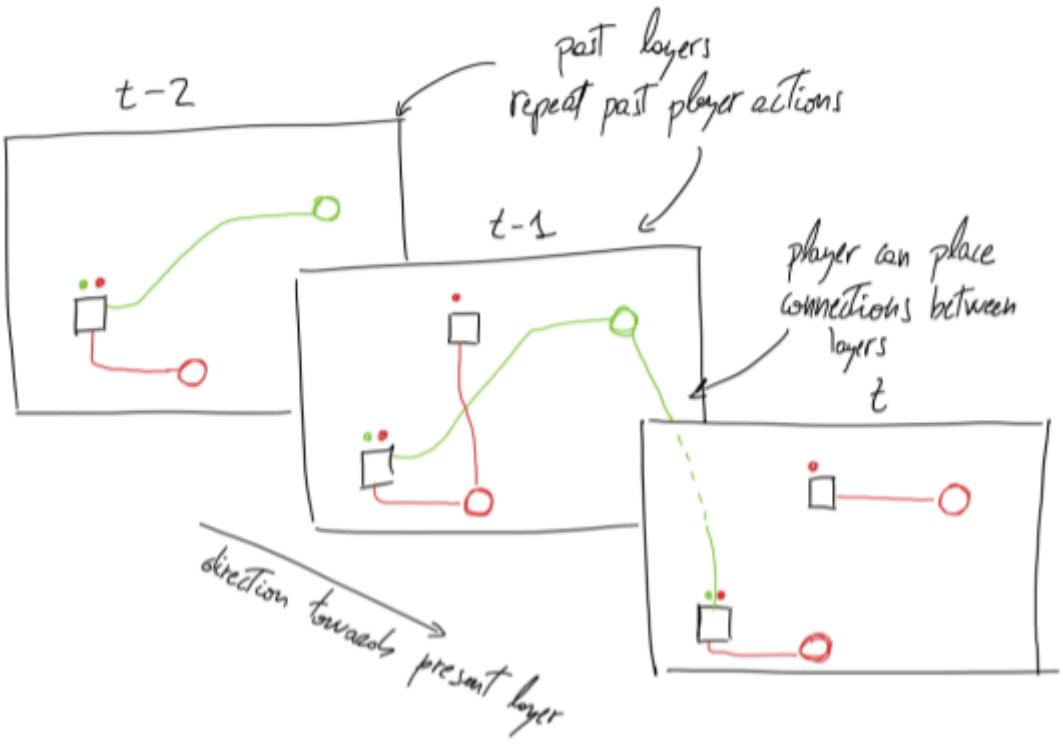
Failure Conditions

If a Time Ripple receives insufficient energy, it collapses, causing a localized implosion of spacetime. This can either cause the immediate termination of employment meaning game over

or deleting the current time slice meaning the player continues from the next layer in time until they run out of layers or after a certain amount of layers imploded.

Player Interaction

Players interact primarily through the network interface, connecting nodes, monitoring energy flow, and adjusting conduits. Decisions are made in real-time and the presence of multiple layers introduces a planning-oriented aspect. Players must evaluate immediate problems while also considering long-term effects, effectively managing a network that spans both space and time.

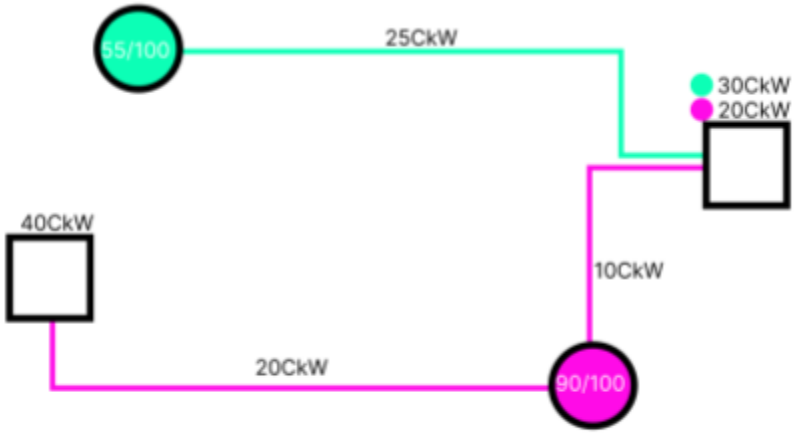


The simulation might require that the player links different layers to keep all the infrastructure going.

Core Mechanics and Systems

Energy Flow System

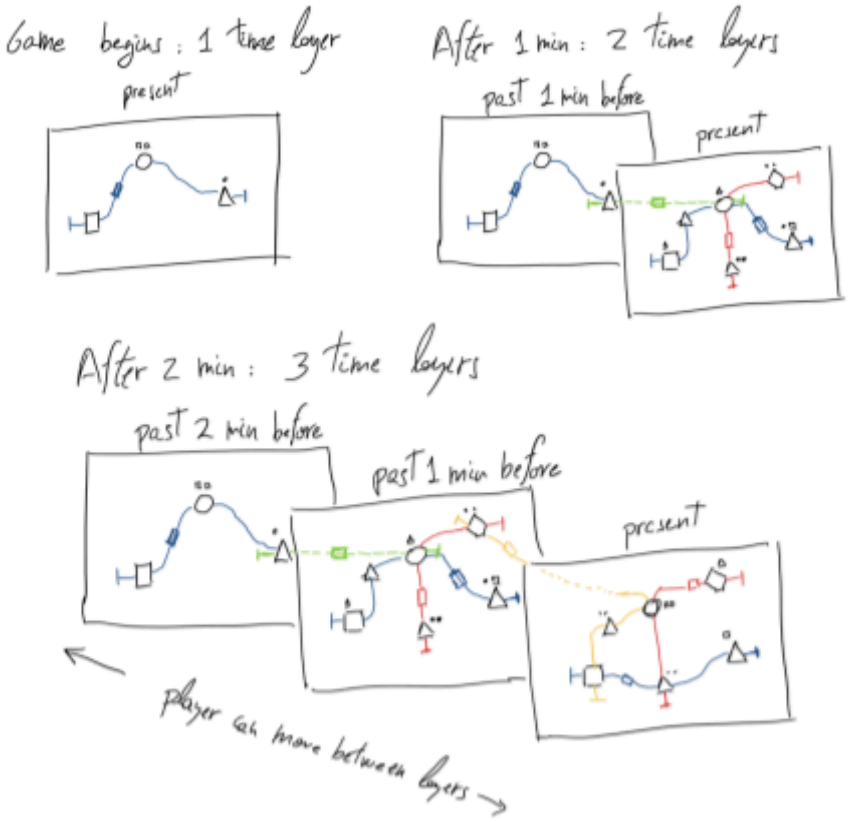
Each node represents a Time Ripple or an energy source. Energy flows along conduits according to demand. Players must balance supply and demand spatially and temporally. Not all energy is the same (represented through different colors) so some nodes only take one type while others might be fine with a mixture or require both.



Time ripples must be sustained by energy sources.

Time Duplication

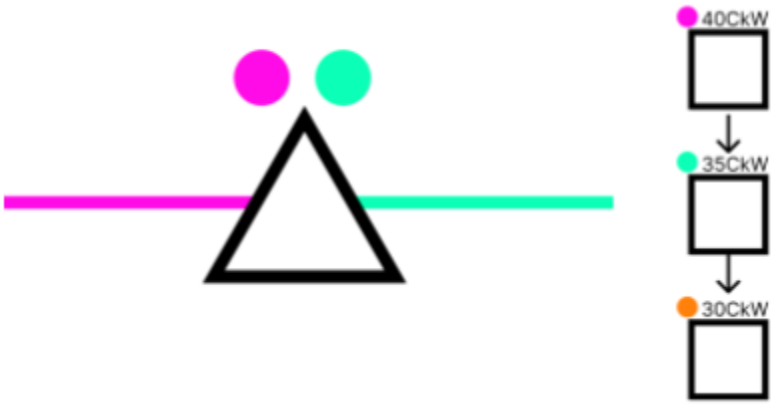
Every minute, the amount of layers increases. The layer representing the present stays and a new layer that represents the gamestate from one minute ago appears. Players must consider how decisions in one layer affect all other layers.



Game run example: time layers appears over time, allow the player to connect their networks and allowing more complex interactions.

Temporal Current Manipulators

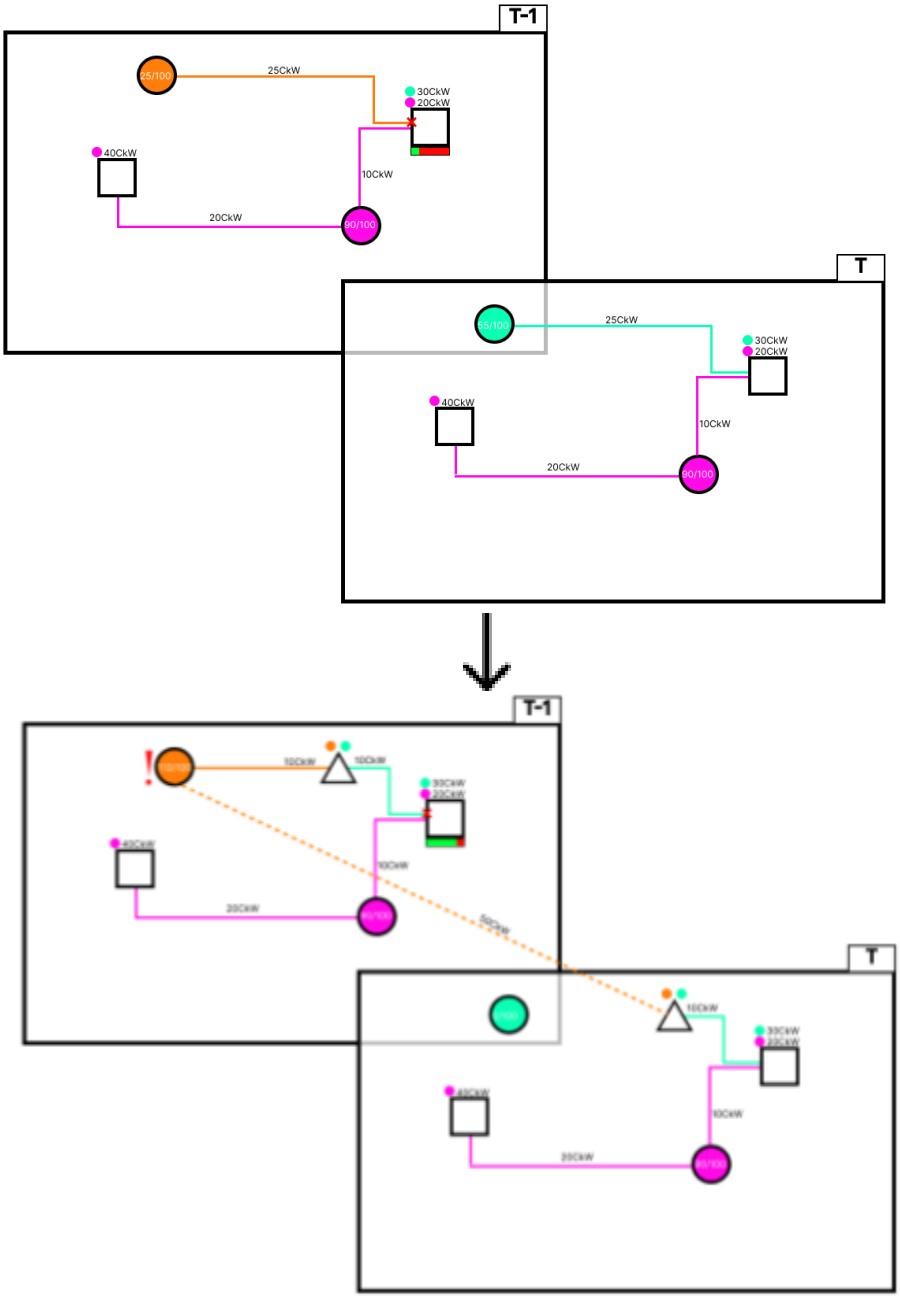
Players might want to use limited items or structures to transform between certain types of energy or influence their degradation, for example maintain the color or let it degrade.



Temporal Current Manipulator (left) and change of energy requirements over time for the nodes (right).

Temporal Dependencies

Different energy sources are only available in certain time slices or are more efficient incentivising the player to connect slices. Energy dependencies can degrade with duplication requiring a different connection meaning a different color. Using items or special nodes in past layers may affect the present. For example, placing a Temporal Manipulator in a past layer affects upcoming layers. This might also have adverse effects, for example preventing nodes from decaying when the player might want them to.



The teal generator from layer **t**, decayed to orange and can no longer sustain the node. The player places a converter in the **t-1** layer but this inadvertently also puts the converter in layer **t**. Now the player has the option to connect either the teal node directly or the orange node crossing the time layers.

1.2. Technical Achievement

Recursive Time Layer Simulation

The main technical achievement will be the design and implementation of the time layers which need to be managed in a way that they work concurrently and can still interact with each other. The requirement with the time layers is that they replay the actions of the present layer, they are their own simulated entities so they can interact with the present layer by creating connections from the future layer to the past and this needs to be implemented in a way that it still achieves feasible performance.
The implementation of this will come down to the following three points:

World State Duplication (Snapshots)

The first aspect is replaying every action made by the player or the game with a delay on a different layer. To achieve this we have to create a copy of every entity in the game. This can be achieved by utilizing the object oriented property of C# to store the data of every object as a dataclass. Everytime we have to clone an object from the present layer and instantiate it in a past layer the data can be serialized to then instantiate a new object which will be managed by a time layer manager. This should increase performance over duplicating the gameobjects.

Parallel Simulation

The main performance bottleneck will be the simulation of multiple time layers because the entire simulation of every layer needs to happen simultaneously. Because the simulation logic is the same for every layer this can be done concurrently using multithreading. C# offers the job system that can be used to schedule every simulation as a parallel job.

Time Layer Connection

The main logical challenge will be to implement the interactions of the past layers while still replaying the actions of the present layer. To achieve this we will work with IDs that are managed by a multi layer manager to retain the references to all entities that work across multiple time layers. Because the multiple time layers work parallel we need to implement a thread safe option to handle these entities to avoid race conditions.

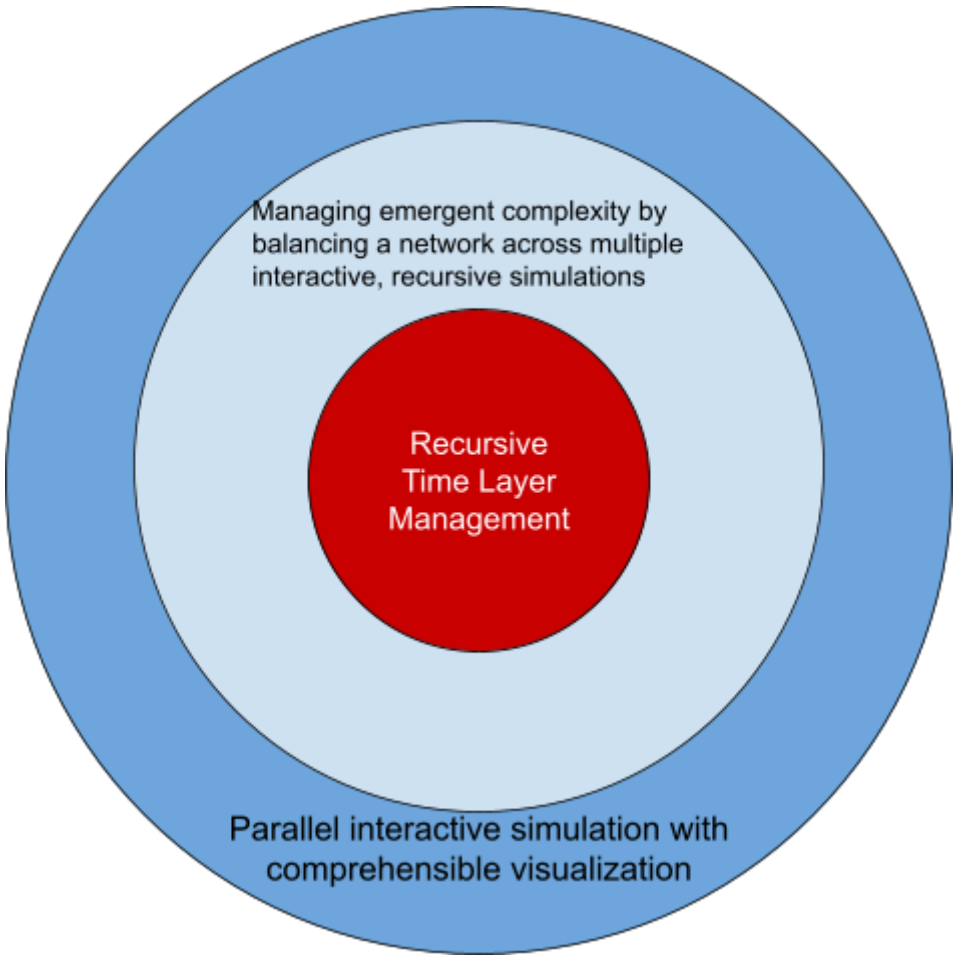
Visual Timelayer Representation

Another technical aspect we have to achieve is the visual representation of the different time layers. They need to be rendered in a way that the player is not overwhelmed with visual information to a point where it is hard to perceive the little details of the game, but the player still needs to receive all the relevant information.
Key aspects will be:

- **Information density:** The information and animations on the screen should not be too cluttered.
- **Time layer differentiation:** The player needs to be able to easily distinguish the different time layers.
- **Temporal connections:** The connections between the layers need to be easily differentiated from connections within one layer.

One way to solve this would be to represent the layers as 2D planes in a 3D space where the layers are placed behind each other with different z coordinates (layer1: z=0, layer2: z=1). This placement of layers can be combined with an orthographic camera that lets the player select a layer to view in the front with the other layers being transparent. Overlapping layers cannot be avoided and lead to visual clutter that needs to be solved

1.3. “Big Idea” Bullseye



1.4. Development Schedule

Our Resources

- Communication: Discord
- Notes and Resources: Google Drive
- GitHub
- Game Engine: Unity (C#)

Layered Development

Functional Minimum:

- One playable time layer
- Node placement system
- Energy flow simulation between nodes
- Basic win/loose condition
- Simple UI showing all relevant information

Low Target:

- Two recursive time layers
- Temporal energy transfers between layers
- Visual feedback
- Minimal UI for switching between layers

Desired Target:

- Fully recursive multi layer simulation
- Polished visuals
- Tutorial and introduction level
- Basic sound design

High Target:

- Polished visuals and sounds
- Dynamic difficulty scaling

Extras:

- Reactive music to simulation

Timeline

Phase	Task	Resource	# Hours	Start Date	End Date
Phase: Game Idea Count: 2			Sum: 10	Max: 10/15/2024	Max: 10/22/2024
	Brainstorming	All	8	10/15/2024	10/22/2024
	Presentation	All	2	10/15/2024	10/22/2024
Phase: Refined Game Proposal Count: 2			Sum: 30	Max: 10/22/2024	Max: 11/5/2024
	Brainstorming	All	20	10/22/2024	11/5/2024
	Documentation / Presentation	All	10	10/22/2024	11/5/2024
Phase: Prototype Count: 4			Sum: 27	Max: 11/5/2024	Max: 11/12/2024
	Create Paper Prototype	Miguel	5	11/5/2024	11/12/2024
	Create UI sketches	Arpad	6	11/5/2024	11/12/2024
	Polishing mechanics	Lukas , Anian	6	11/5/2024	11/12/2024
	Documentation / Presentation	All	10	11/5/2024	11/12/2024
Phase: Interim Demo Count: 8			Sum: 108	Max: 11/12/2024	Max: 12/3/2024
	Create Foundation	Nodes, Connections, Slice Foundation) [All	12	11/12/2024	12/3/2024
	Create Grid Building System	Miguel	24	11/12/2024	12/3/2024
	Create basic UI	Arpad	12	11/12/2024	12/3/2024
	Create Energy Flow Algorithm	Anian	10	11/12/2024	12/3/2024
	Create Graph Traversal Simulation	Lukas	10	11/12/2024	12/3/2024
	Internal Feedback and internal playtesting	All	10	11/12/2024	12/3/2024
	Bug Fixing	All	24	11/12/2024	12/3/2024
	Documentation / Presentation	All	6	11/12/2024	12/3/2024
Phase: Alpha Release Count: 6			Sum: 65	Max: 12/3/2024	Max: 1/14/2025
	Improve spawning algorithms	Anian	4	12/3/2024	1/14/2025
	Basic resource balancing	All	20	12/3/2024	1/14/2025
	Adding -1m time slice	All	15	12/3/2024	1/14/2025

Phase	Task	Resource	# Hours	Start Date	End Date
	Artistic improvements	UI, Sounds, etc...) (Arpad, Lukas	10	12/3/2024	1/14/2025
	Win / Loss conditions	All	10	12/3/2024	1/14/2025
	Documentation / Presentation	All	6	12/3/2024	1/14/2025
Phase: Playtesting Count: 9			Sum: 111	Max: 1/14/2025	Max: 1/28/2025
	Multi time slice support	All	10	1/14/2025	1/28/2025
	Improve mechanics	All	10	1/14/2025	1/28/2025
	Bug fixing	All	15	1/14/2025	1/28/2025
	UI improvements	Arpad	10	1/14/2025	1/28/2025
	External playtests	All	30	1/14/2025	1/28/2025
	Refine visuals	Arpad, Lukas	10	1/14/2025	1/28/2025
	Gather feedback and implement fixes	All	10	1/14/2025	1/28/2025
	Balance difficulty and pacing	All	10	1/14/2025	1/28/2025
	Documentation / Presentation	All	6	1/14/2025	1/28/2025
Phase: Final Release Count: 3			Sum: 34	Max: 1/28/2025	Max: 2/4/2025
	Bug fixing	All	20	1/28/2025	2/4/2025
	Trailer	All	10	1/28/2025	2/4/2025
	Documentation / Presentation	All	4	1/28/2025	2/4/2025

1.5. Assessment

Main Strength and Core Appeal

The main strength of *Chrono Corp* lies in its novel recursive time-layer simulation. This creates a unique experience where the player is not just managing a network in space, but also across multiple layers of time. We aim at giving the player a feeling of an emergent complexity that grows naturally over time given very simple mechanics. Players begin by playing a simple simulation, but as past time-layers are generated, they suddenly find themselves juggling an expanding stack of mutually interacting recursive simulations. Our final goal is to give the player the sense of an orchestra conductor completely immersed in our space-time simulation.

Target Audience

Our game appeals to players who enjoy strategic thinking, systems management, and abstract simulation. Fans of games like *Mini Metro*, *Mini Motorways*, *Factorio*, *Infinifactory*, or *Opus Magnum* will likely appreciate its simulation characteristics paired with the deep and novel recursive gameplay. The minimalistic aesthetics and simple game mechanics are also aimed as a low entry barrier for casual players. Additionally, players intrigued by sci-fi concepts (time travel, causality, …) and novel ideas in videogames might find our game interesting.

Success Criteria

We consider our game a success if it achieves the following:

- **Player Immersion:** The player experiences a tangible sense of temporal causality**,** feeling that their actions impact meaningfully across temporal layers.
- **Easy to Play but Hard to Master:** The game achieves a balance between minimalism and a low barrier of entry for the player and depth, offering complex dynamics from simple mechanics.
- **Clear Visualization:** The recursive simulation remains intuitive and visually comprehensible and pleasant, despite increasing complexity.
- **Aesthetically Pleasing:** Viewers and players have a compelling experience both in visuals and music. Just watching the game being played should entice a player to try our game.
- **Player Engagement:** The player continues playing to complete its current game session without leaving midway due to frustration.

2. Game Prototype

2.1. Prototype Creation

Physical Prototype

We prepare a simple but effective game prototype to check our game ideas and mechanics. We abstract the time layers to a sheet of cardboard, visualizing what the player might see. The objects (Time Ripples, Generators, Current Manipulators) are abstracted using sticky notes with a symbol indicating their type (square, circle and triangle respectively). The connections between objects are made using thread differentiated by color. Everything is held together with pushpins.

We create a basic prototype using common office materials.

Gameplay Simulation Process

For testing our physical prototype one team member assumes the role of player while another replaces the simulation environment, the rest of team members help to catch potential flaws and mismatches between previous ideas and current playtest. Although our game is thought to be played showing the time layers as 3D we decide to playtest using all time layers on the same 2D surface to avoid unnecessary complications. Additionally, we do not follow real-time mechanics and allow the player all the necessary time to think about network adjustments to solve the simulation and continue playing (as a turn-based game in practice) . We play several runs while updating the core game mechanics to obtain a refined version of them.

2.2. Game Mechanics

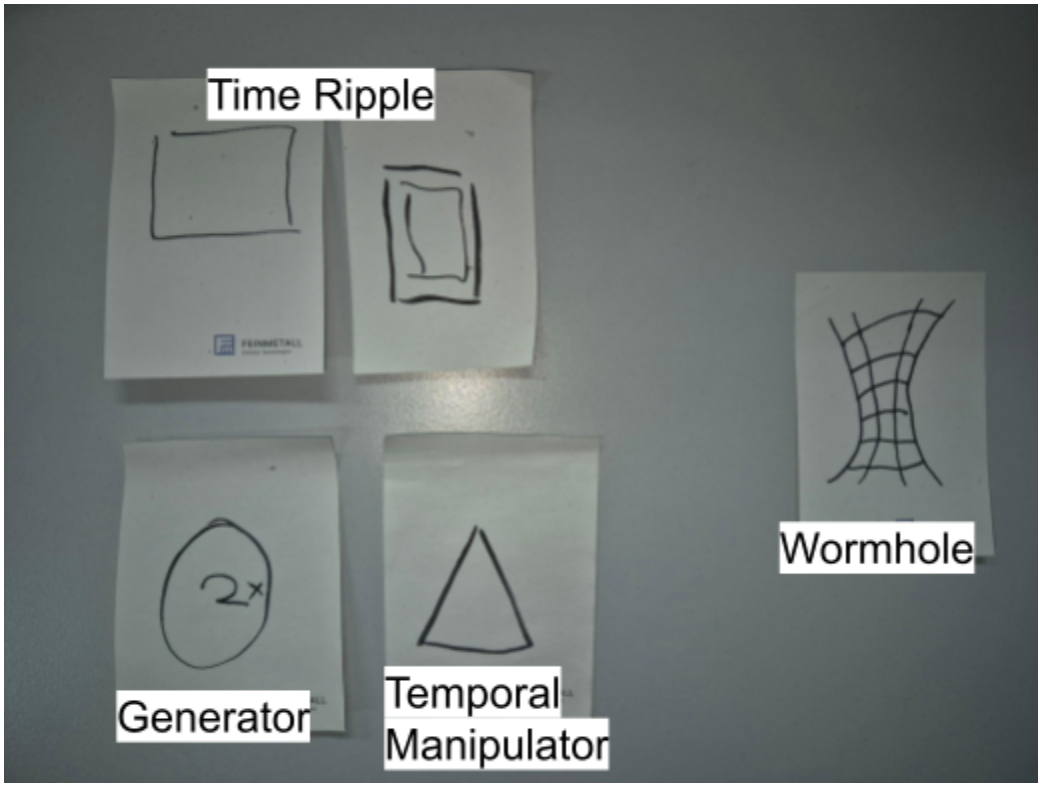
Core Game Mechanics

After playtesting our physical prototype we agree on a core set of game mechanics:

Simulation

- The simulation begins with a single Time Layer (T) that represents the present state of the simulation.
- New Game Objects (Time Ripples, Generators) appear pseudorandomly in Time Layer *T* over time.
- After some time a new Time Layer (T-1, T-2, …) appears. This new layer reflects all the changes regarding the Game Objects, but none from the player actions.
- The player can lose the game if any of the Time Ripples at any Time Layer receives insufficient energy.

Game Objects



- **Generator:** each generator produces some amount of energy packets (2 in our prototype). Each generator has *n* available slots (initially only 1). Each slot can be used by a single connection. The Generator produces energy of the color that matches that of

the first Time Ripple connected to it. An energy packet travels from a Generator to a single Time Ripple of its color through the network. The generator fuels all the Time Ripples that are connected to it in a sequential order, e.g.: if three Time Ripples (A, B, C) are connected then the generator will send one packet to A, then B, C, and finally A again, repeating the cycle.

- **Time Ripples:** each Time Ripple consumes some amount of energy packets (symbolized by its number of squares). Each Time Ripples has a particular color (green, blue, pink, …) that is determined by the simulation. For the simulation to continue every Time Ripple must receive enough energy packets of its color at every moment.
- **Current Manipulator:** A Current Manipulator can modify the color of the pipeline downstream, they receive energy from a particular color and they emit the same amount of energy of a different color. The input and output colors are determined by the simulation.
- **Wormholes:** a Wormhole allows the player to make a single crossing connection between pipelines by spending it.

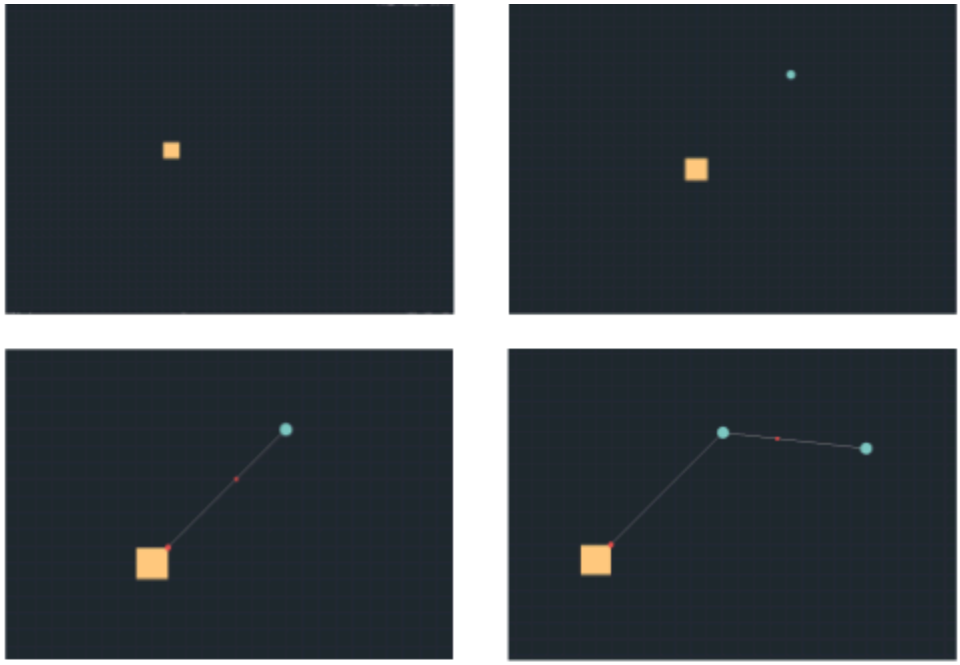
Player Actions

- The player can place directed links between the Game Objects in any Time Layer. Every pipeline begins at a Generator slot and goes through one or more Time Ripples and/or Current Manipulators with no loops. Each pipeline is a single path that moves energy in a specific direction.
- The player can link Game Objects between adjacent Time Layers (e.g. a link from layer T-1 to layer T or T-2, but not to layer T-3).

Additional Ideas

Although we were not able to test fully due to the limitations of the physical prototype we proposed the following additional expansions:

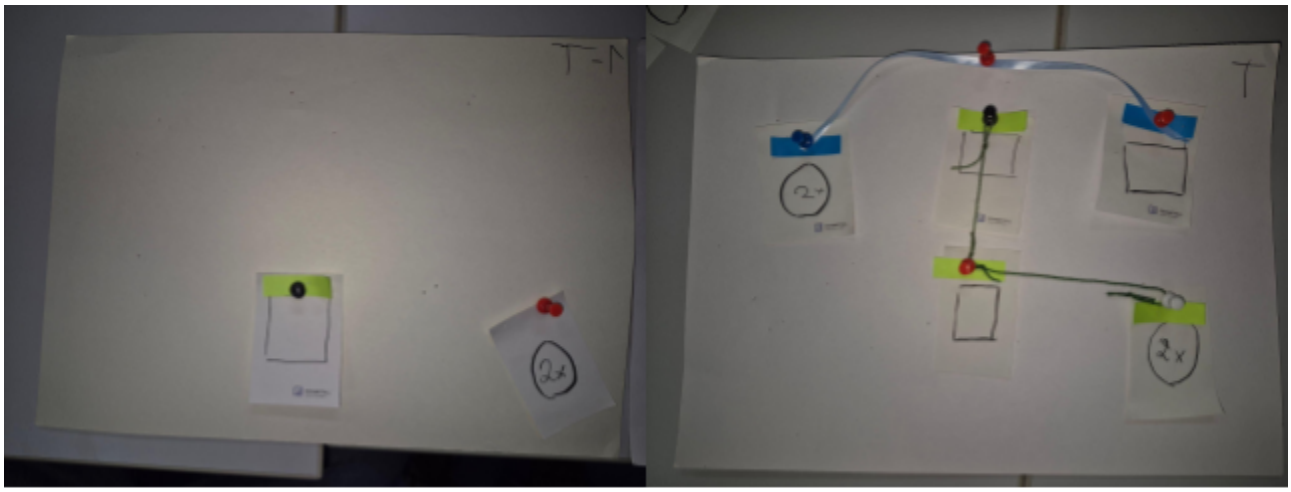
- **Perks:** after some time, the player can choose a perk from a small set of options to expand its gameplay options, such as:
    - Amount and level of Generators: place an extra Generator or upgrade one of them to produce more energy.
    - Additional Current Manipulator to be placed by the player.
    - Move a Generator to a new place within its Time Layer.
    - Additional crossings between pipelines: in order to place a crossing between pipelines on a Time Layer, some resource or card will be required, the player might choose to add some extra crossings to modify his current network.
    - Additional links between Time Layers: in order to place a link between Time Layers, some resource or card will be required, the player might choose to add some extra inter-layer links to modify his current network.



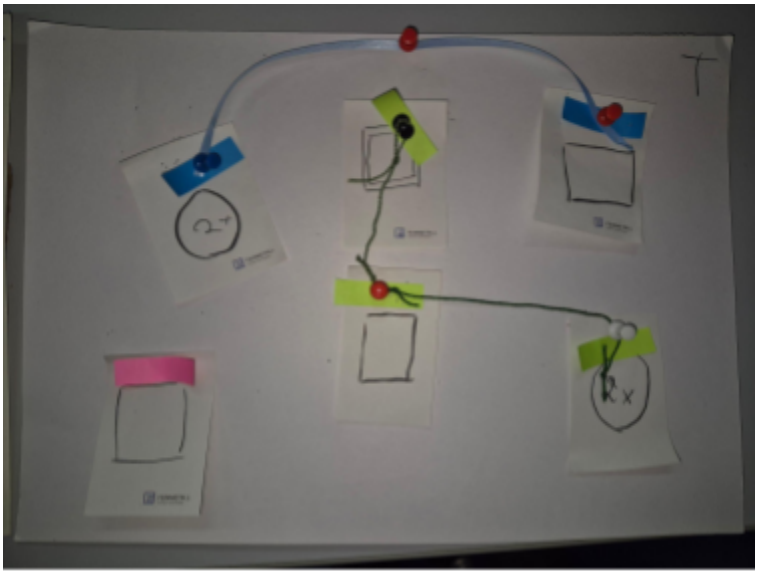
A Gameplay example (left to right and top to bottom) showcasing how the energy is assigned from the Generator to the Time Ripples. First the first Time Ripple connected receives one packet of energy, then the second Time Ripple in the network will receive the second packet.
Note: in this particular example the square symbolizes the Generator and the circles the Time Ripples.

Gameplay Example

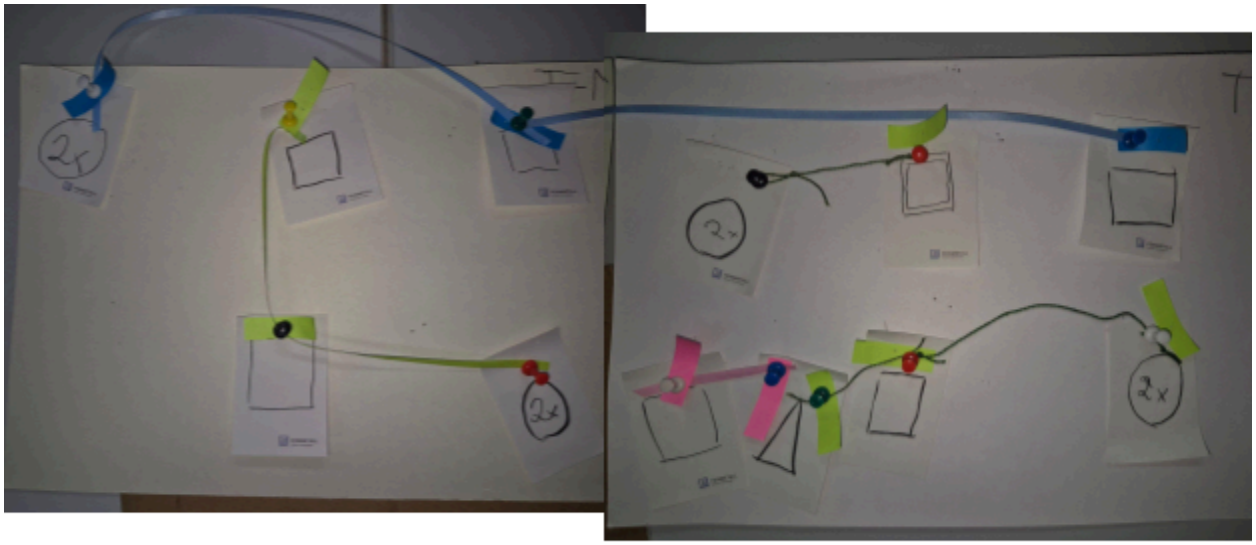
We illustrate how a gameplay test run can look like at three time moments:



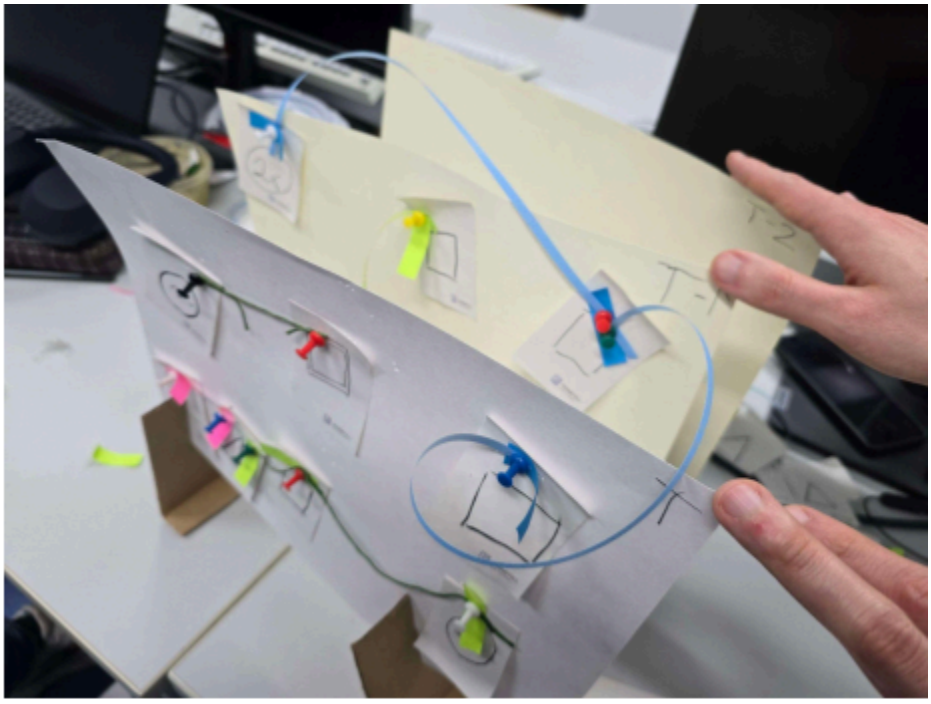
After one minute of ingame time has passed, Layer T-1 shows what layer T looked like one minute ago. The player has already solved the layer T and now can repeat the same connections in T-1.



While both layers continue the simulation, layer T has reached a point where it is no longer solvable within the layer T. There are only two generators producing energy of two colors, but the new node requires a third color (pink). Additionally, one of the green nodes has evolved to now consume two packets, now the two green nodes can no longer be supplied by one single Generator 2x.



There are multiple solutions to this problem. The player decides to link the blue Time Ripple to the blue network from laye
T-1. The double consumption green node is supplied by a single (2x) Generator, while another Generator supplies both
green and a pink node converting the excess green energy into pink energy via the Temporal Manipulator.



The game loop now reaches 2 minutes and layer T-2 gets introduced. Here we showcase our prototype in 3D emulating how the player might visualize the layers in the game. The inability to solve the simulation within a single layer and the necessity to make connections to other layers will inevitably change the gameplay and network in those layers avoiding that the player just repeats what previously worked.

2.3. Insights and Iteration

Insights

During testing, we found out that many of our initial ideas were not fully developed. We adopted the following changes in order to make our game more simple, appealing and fun to play:

- **Simplification:** We simplify the amount of energy that Generators produce and Time Ripples consume. Instead of using arbitrary CausalKw numbers, we choose to use low integer values (1, 2, 3, 4, …), this helps in understanding how the simulation should be solved and diminishes the mental load while playing.
- **Time Layers focus:** we detect early that in order to make our game more fun and original, our game mechanics should move the player to make connections between Time Layers. We aim to achieve this by inducing a mismatch between the set of colors and energy that the Time Ripples required and the Generator at a particular Time Layer so that the player makes use of connections to other layers.
- **Time Layer behaviour:** we believe it to be the most complicated aspect in our game. After thorough testing we decide to simplify the behaviour of past layers in order to repeat only the appearance of Game Objects (not the player actions) as it happened in the initial layer T. The player will be required to make the connections at every layer. This provides the player more flexibility for making inter-layer connections and simplifies the implementation and cognitive aspects of the simulation. In case we find this to be too simple or too cumbersome for the player we might change this behaviour.
- **Balance:** for the simulation to be solvable the overall amount of energy produced should be at least enough to satisfy the demand for energy at every time by all the Time Layers. This encourages interaction between Time Layers while making the game possible to play.
- **Unidirectional flow network:** in order to make the simulation easier to understand and to have a more predictable behaviour, we decide to restrict the network to unidirectional segments with no loops.

Initial Assessment

Although we like the basic gameplay and believe that the current game mechanics make for a challenging and fun game for the player (mostly from its puzzle-like component), the physical prototype shows its shortcomings when giving a real feeling about how intense and engaging the game can be. A digital prototype might be able to more accurately demonstrate the real-time aspect and its interaction with our game mechanics.

3. Interim demo

Progress

We have implemented the foundational components necessary for core gameplay and visualization. This includes establishing the base functionality for one playable layer, allowing players to interact with the board in a single time slice. The systems for manual generator and time ripple placement for testing purposes are fully operational. Furthermore, the energy flow simulation logic between nodes is complete. Finally, the camera controls and layer visualization are functional, utilizing the Helical Staircase display (as detailed previously) to navigate between time slices
For the Interim Demo, we made a crucial decision to delay the creation of a Basic UI in favor of dedicating our resources to the core layer visualization and the underlying simulation logic. This choice was driven by the need to establish a solid, working game backend first. The UI's function is simply to take information from this backend, displaying both global game state (like total energy) and per-node information (like individual charge or connection status) and visualize it. If the simulation logic isn't final, the UI is constantly chasing a moving target.
Our focus, therefore, has been on getting the simulation part working reliably. With a stable backend, we gain immense flexibility. For the Alpha release and playtests specifically, we can easily tweak the core gameplay rules by adjusting simple parameters like the maximum number of connections allowed per node or the spawn rate of energy and new nodes. This allows us to rapidly prototype and tune the most important part of the game, without being bottlenecked by unfinished backend elements.

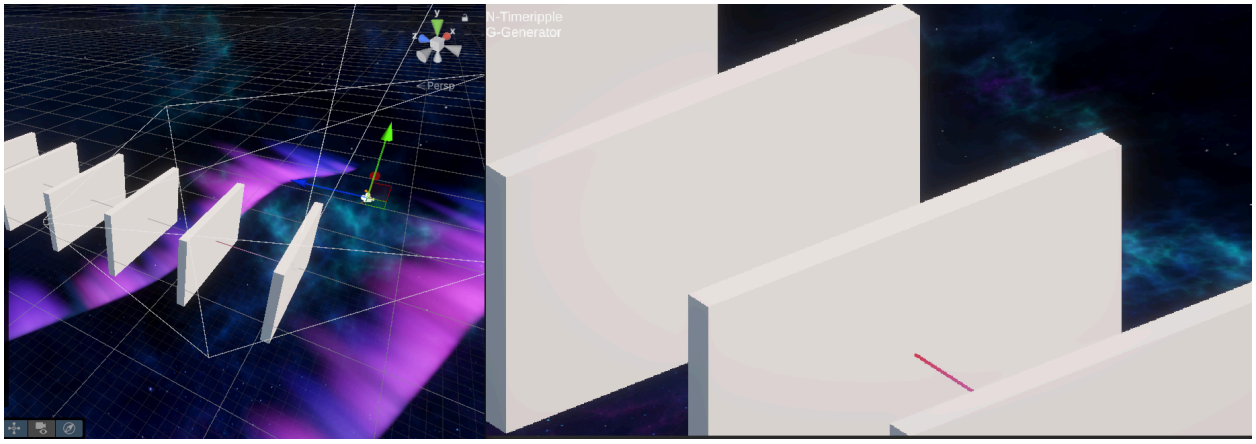
Time layer UI and Corresponding Camera

A significant challenge in our game is displaying the time-layer mechanic in an easy-to-understand way without cluttering the screen with information. Part of this challenge involves showing each slice (or layer) without having them overlap or be obscured by others. To address this, we have implemented three primary parameterized (and one variation, making it 3.5) ways of displaying the gameboard, allowing us to continuously experiment and conduct usability tests.

1. Stacked Layers (The Intuitive Baseline)

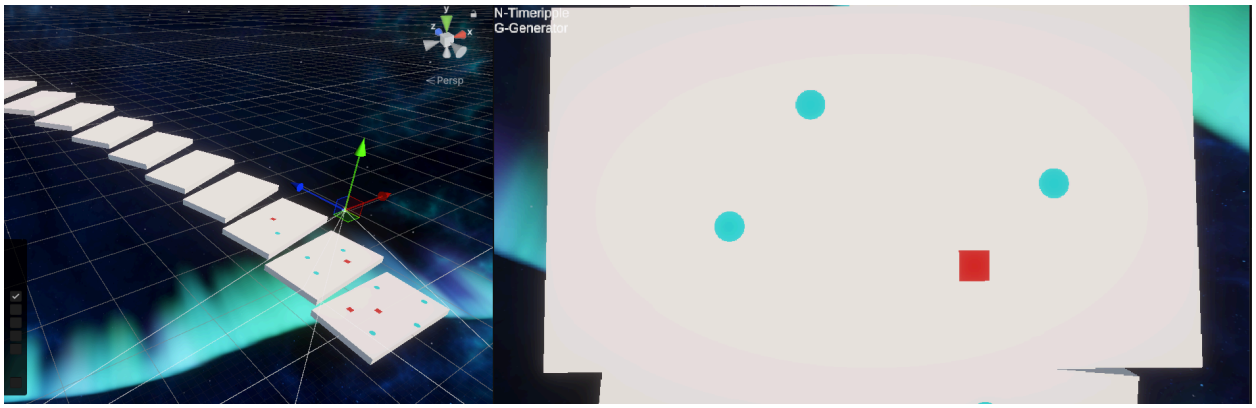
Our initial approach involved layers stacked directly behind each other. This was the most intuitive method and matched the design of our physical prototype. This type of interface can also be found in games like 5D Chess. However, we were dissatisfied with the portrayal due to significant overlap, the small scale of each individual field, and the resulting inefficient screen

usage and empty space surrounding the fields, especially due to the perspective distortion inherent in that setup.



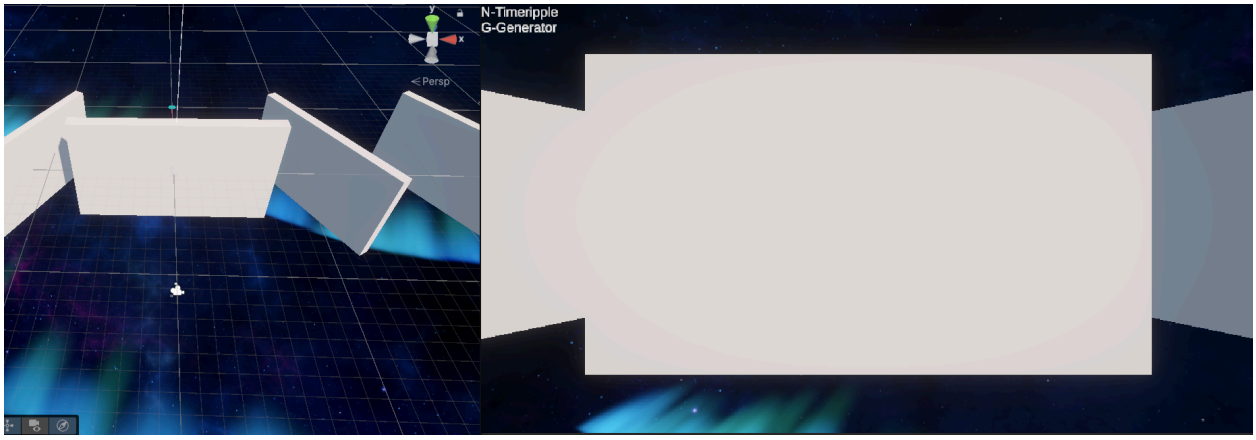
2. Helical Staircase / Fanned Layout

A more sophisticated and successful approach was inspired by card games like Hearthstone, where the deck is "fanned out" or "riffled" to reveal a single card. We structured the board slices like a helical staircase, effectively emulating this fanned/riffled reveal. This allows for a much more efficient use of the available screen space and enables smoother camera motion, with the visible area proportional to the player's focus. We are committing to this approach for now, and our next steps involve creating smoother animations when dragging connections between two frames and refining the visual effect to truly emulate the functional "choosing a card" look. The key parameters controlling this are the radius of the spiral, the elevation separation and the angle of rotation between adjacent slices. A side-benefit is the parameterized variation: by setting the radius of the spiral to be very large, the visible helical motion transforms into a straight vertical scroll akin to browsing a website, giving us a fourth display option.



3. Cover Flow / Carousel

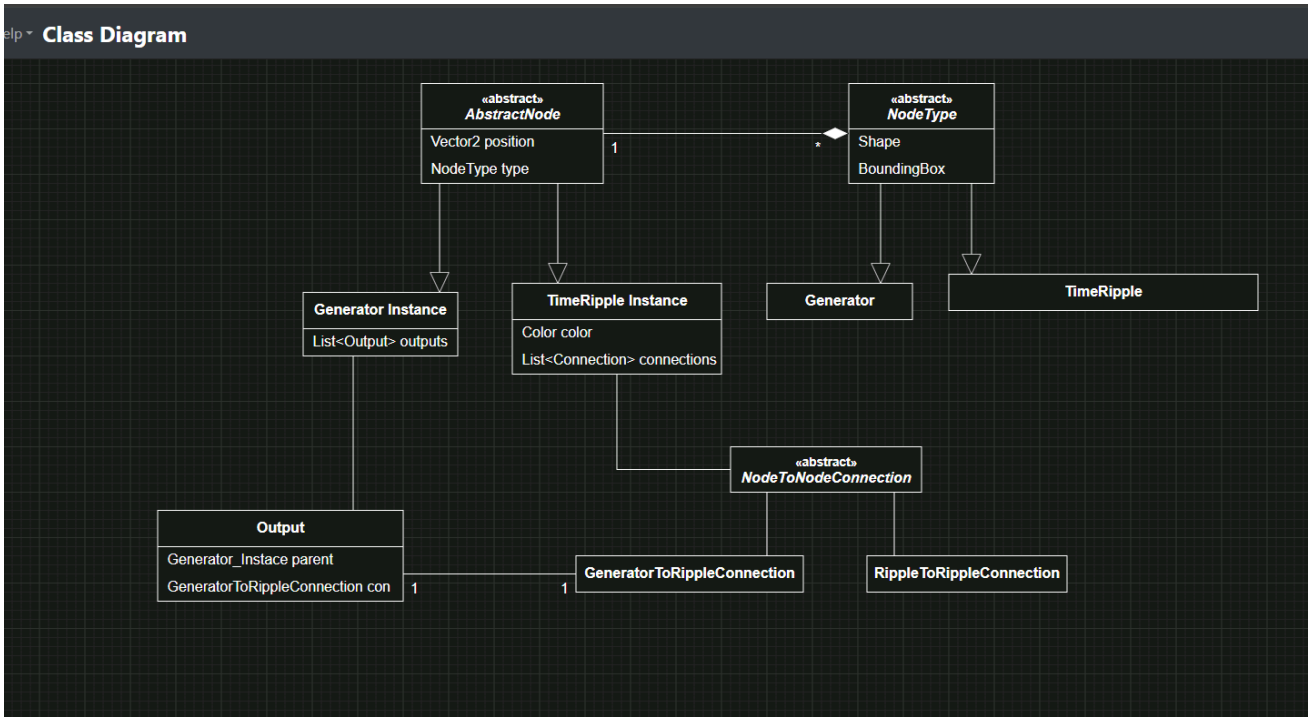
The third way was inspired by Apple's iTunes Cover Flow interface and was a close contender with the helical staircase. It offers a glance at neighboring slices and provides a smooth and satisfying scrolling experience. Ultimately, we decided to pause development on this due to the complexity it introduced: since the entire frame stack needs to move and rotate within the camera's view, this would require dynamically moving and adapting the inter-frame connections to maintain visual integrity, which presents a development hurdle but more importantly could introduce a lot of visual noise with connections shifting around, making the player confused as to what is connected to what.



Frontend-Backend Split

Since our game relies so heavily on graph and traversal logic, we made a deliberate choice to separate the underlying game brain from the visual presentation.
We structured our code to follow the principles of Model-View-Controller (MVC). The core simulation logic tracks the state, runs the algorithms, and enforces game rules and is completely separate in its own set of Model and Controller classes. The View components, which are what the player actually sees (the UI, board and nodes), only pull information from the Model to render the visuals.
We can focus on our specific domain. The systems team optimizes the algorithmic complexity within the Model, while the Frontend team concentrates on visuals, camera mechanics, and smooth user interaction within the View.
The Model and View are decoupled. We can update the core game logic or algorithms within the Model classes without requiring *any* changes to the visual components or the way they are rendered, significantly streamlining iteration and maintenance cycles. For example, changing how time packets are routed in the Model doesn't break the rendering of the time slices in the View. Or changing the display method from the helical Staircase to the cover flow doesn't require changes on the backend and can be done with a click of a single button.
The core game logic (the Model) can be unit-tested as well as pure C# code, without relying on Unity's editor or game loop, leading to a much more reliable codebase.

Implementation



Graph Representation

The nodes on a time slice and their connections can be viewed as a graph with connections. Thus, we chose to define different types of nodes like time energy generators and time ripples. Generators have a fixed amount of output. An output can be connected to any other node except generators.
Time Ripple Instances on the other hand are not limited in the amount of connections they can have. However only one connection between two time ripples is allowed.

Energy Packet Logic

As the game is about building a network that distributes energy to nodes we have implemented a breadth first search algorithm to calculate the paths energy packets can take to travel to a desired node target.
As only generator outputs produce energy they know all routes available and distribute energy packets evenly across the known routes.
With a fixed cooldown between generation ticks each output produces energy packets.
An energy packet defines a constant travel speed and knows its entire travel path on creation.

Tick Logic

The tick heartbeat is provided by Unity's FixedUpdate method and passed to the backend. The Backend distributes the tick signal to the available time slices which tick their nodes and packets accordingly.
The number of the current tick is passed to create deterministic results when replaying the simulation for other past time slices
.

4. Alpha Release

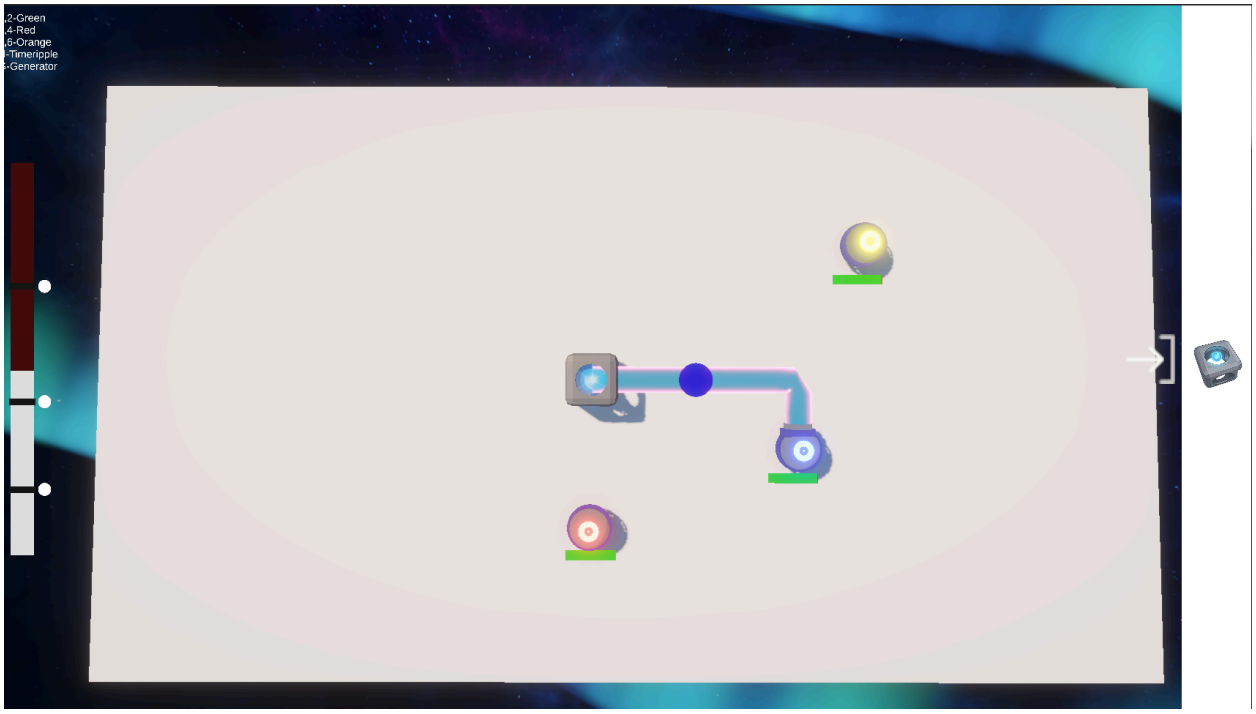
Progress

The main goal for this milestone was to upgrade from a testable simulation to a playable game. To achieve this we implemented multiple layers with automatically spawning time ripples, we added a stability system and an inventory to limit the players resources. Additionally we made some changes to our visual representation of generators and connections.

Player Inventory

The Inventory System in our game functions as a resource pool. It is responsible for tracking the distinct building blocks the player uses to construct their network. The inventory currently tracks the amount of generators and the amount of connections the player can place. For now the player can place 3 generators and an unlimited amount of connections (int.MaxValue).

Before any object is instantiated in the game world, our game controller tests if that specific resource is available and if placed it updates the count in the inventory accordingly. If the player removes a placeable object the available amount of that resource is incremented in the inventory.

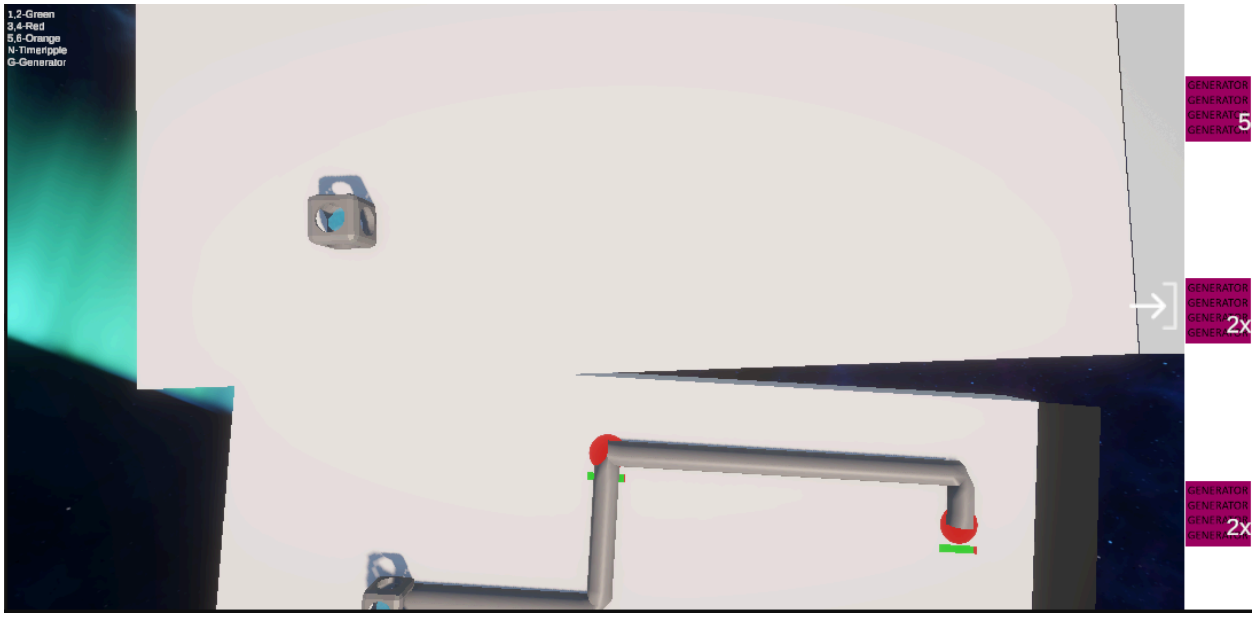


Multilayer Gameplay

Our gameplay is built around a **multi-layer simulation architecture**, where each layer represents an independent *TimeSlice* with its own grid, nodes, and simulation state. Layers run in parallel but follow the same systemic rules, allowing the player to interact with and progress through multiple layers without breaking overall consistency. Each TimeSlice operates on a discrete tick-based simulation loop, ensuring deterministic updates while still enabling emergent behavior across layers.

Randomness is introduced in a controlled, systemic way using seeded pseudo-random generators. This randomness influences node spawning, energy type selection, and spatial placement on the grid through weighted random selection of valid cells. By constraining randomness with clear rules (such as cooldowns, allowed energy types, and available space), the game avoids pure chaos while still guaranteeing variation between runs and layers.

The **multi-layer design** allows us to scale complexity horizontally: systems like energy routing, stability, and node interactions behave consistently within a layer but can diverge across layers due to different random outcomes and player decisions. This creates a gameplay experience where each layer feels familiar in mechanics but unique in state, encouraging strategic planning both within a single layer and across multiple layers over time.



Stability Bar and Debuffs

The game features a **global Stability Bar** that represents the overall integrity of the entire simulation across all layers. Instead of tracking stability per layer, every player action, system interaction, and random event contributes to a single shared stability value. This makes stability

a meta-resource that connects all layers and turns multi-layer gameplay into a strategic balancing act rather than isolated problem-solving.

Stability is influenced by cumulative factors such as energy imbalance across layers, inefficient node configurations, unresolved instabilities within individual layers, and excessive expansion without sufficient support systems. While a single layer can temporarily operate in an unstable state, prolonged or widespread instability will negatively impact the global Stability Bar.

As global stability decreases, the game is designed to introduce **progressive maluses** that affect the entire simulation. The following maluses are **conceptual examples** intended to illustrate the design direction; they are **not yet fully designed, balanced, or implemented in the current game build**.

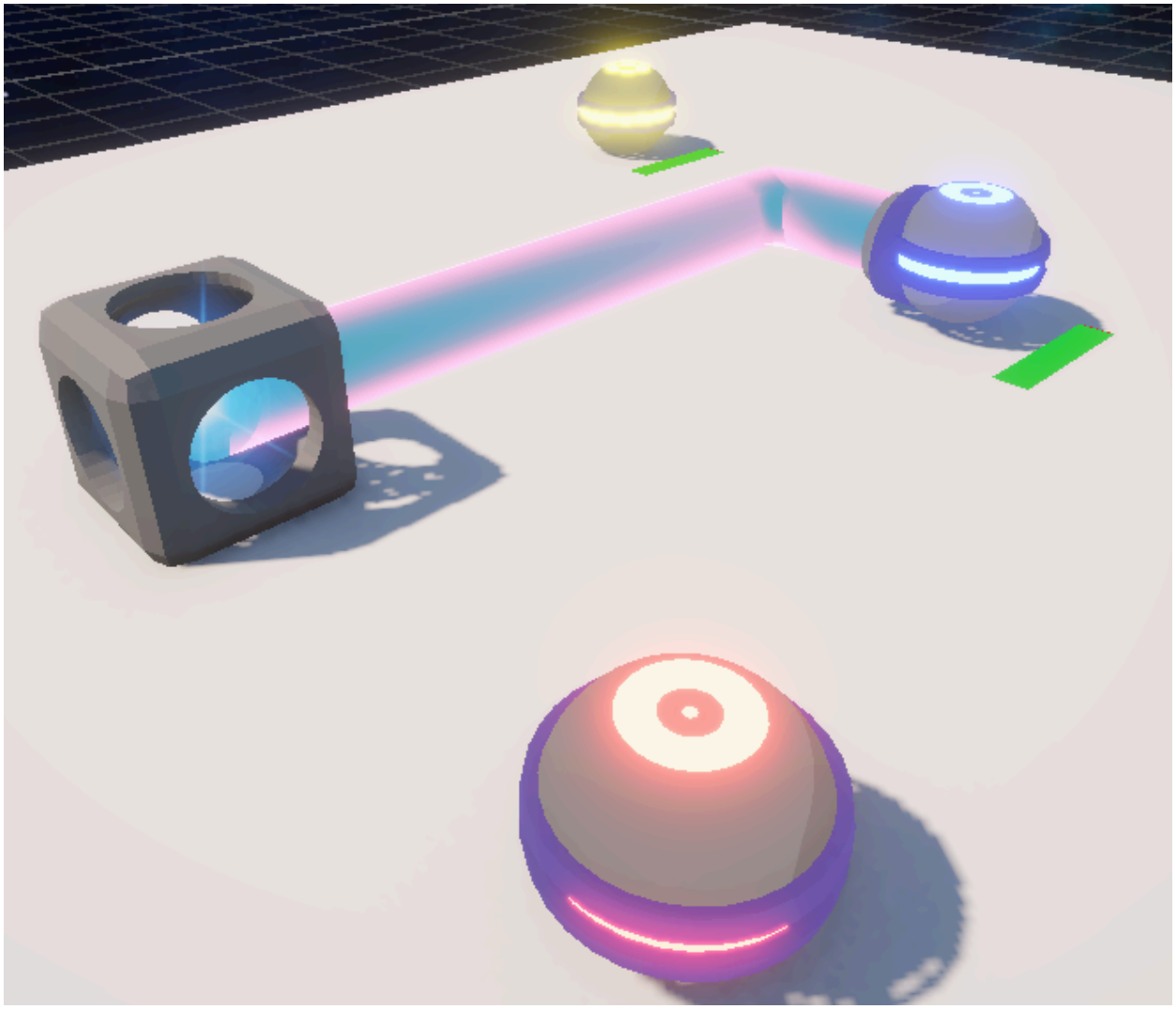
At moderate instability levels, potential maluses may include reduced efficiency across all layers, such as increased energy costs, slower node activation, or longer cooldowns for player actions. At lower thresholds, instability could manifest as systemic disruptions, including random delays in layer simulation ticks, temporary restrictions on layer interactions, or cross-layer interference where events from one layer negatively affect another.

At critical stability levels, the simulation may enter a **critical state**, where severe maluses become possible. Example effects include cascading failures between layers, forced shutdowns of high-energy nodes, or unpredictable restructuring events that partially override player control. These effects are intended to increase tension and decision pressure rather than immediately end a run.

The global Stability Bar is designed as a long-term pressure system that ties all layers together. By making stability a shared resource, the system reinforces strategic planning, controlled growth, and meaningful trade-offs between short-term optimization and long-term systemic health.

Change to 3d models and visuals

In the frontend we upgraded our visual representation of the generators, connections and nodes from 2d to 3d visuals. While leaning on the Mini Metro aesthetic provided a basis to focus on the backend logic, we wanted to differentiate it and put our own twist on it.



We settled on a Sci-Fi aesthetic in a minimalistic low poly esque style with simple visuals and 3D models we could create ourselves.
The Time Ripples according to the story are trapped by Chrono Corp to stabilize them and then fed with energy, so we wanted a simple dyson sphere resemblance. Encased but with energy radiating out.
The conduit was changed to a spline and mesh based line with bends similar to diagram software to adhere to the grid system.
Connector pieces connect the conduit to the node for a smoother transition and cleaner look.
The Generator resembles a Power Crystal with a structure around it with holes to make connections and draining power possible. The goo in the middle is animated with a custom Shader to make it look like it's flowing and living almost. Generators as inventory items also receive their own icon: