Oh No!!! I Sent My Professor a Rude Email And Need To Sneak In To Get It Back!

200018789 200012955 190031593

May 2024

1 Introduction

This report will detail the creation of a previously-pitched game using Processing.

The game we pitched is called *"Oh No! I Sent My Professor a Rude Email And Need To Sneak In To Get It Back!"*, and is a puzzle-platformer game revolving around a student that needs to sneak into their teacher's office and delete an email from their computer. To do this, they need to progress through various puzzles using an ability to manipulate the rules of the world around them to their advantage.

In this report, we'll discuss our decisions throughout this process, how we implemented these mechanics, and how well we achieved what we set out to create.

2 Design

Genre and Context

From the outset, we wanted to create a game that blended platforming elements with more abstract puzzle mechanics.

We've taken inspiration from some other games as a basis for our mechanics.

- **Super Mario / other platformers** - many of the core mechanics in the game are those of a standard platformer game, inspired by the floaty physics and wall jumping of the *New Super Mario Bros.* games. Our game leans a little harder on puzzle elements than true platforming challenges, though. Additionally, our game focuses more on single-screen challenges rather than larger side-scrolling levels. The ability to grab and throw objects is also inspired by similar mechanics in *Super Mario Bros. 2*.

Image from RetroGameMan[1]

- **Baba Is You** - a puzzle game that revolves around manipulating the rules of the game through words in each level that can form sentences. This inspired us to create a similar system where the rules of the world (the direction of gravity, enemy AI, etc.) are represented as objects in the world that can be manipulated. However, this game is a Sokoban-style block pushing puzzle, rather than a platformer with semi-realistic physics as in our game.

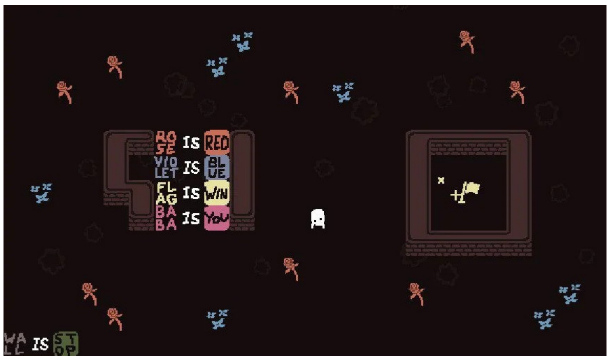


Image from Polygon[2]

- **VVVVVV** - a puzzle-platformer revolving around controlling gravity as a replacement for jumping. Some of the gravity-changing mechanics are similar to VVVVVV, but we still allow the player to jump, since these mechanics are tied to objects in the world that you may still need jumping to reach.

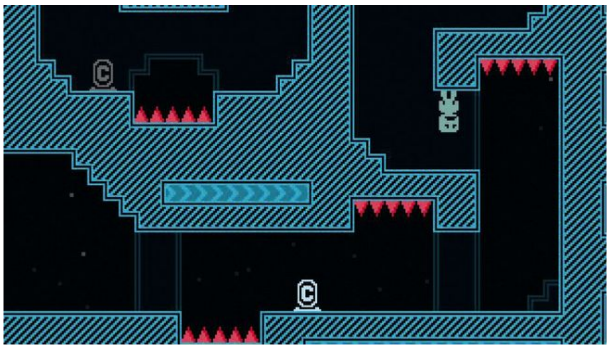


Image from PCGamer[3]

- **Portal 2** - a 3D puzzle-platformer that we've taken the bouncy mechanics from. While the main mechanic of the game (the portal gun) isn't present, *Portal 2* includes a blue goo that you can shoot on surfaces to make them bouncy, similar to the "bouncy" modifier block in our game.

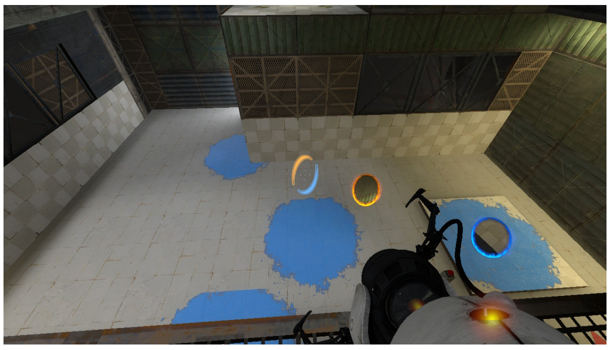


Image from Gameranx[4]

2.1 Aesthetics

We went for a pixel art style for the game, not only because it's easier to create pixel art sprites, but also as a nod to our retro-game inspirations, and other modern games that use a pixel art style.

The player is represented as this hooded character, in order to make them seem stealthy and nondescript. They have a few animations including a running sprite and a sprite for when they're holding an object. The levels are constructed with tile sprites that are intentionally cold feeling to evoke the feeling of being in a university building after dark.

Implementation-wise, making the game look pixel-accurate requires using Processing's `noSmooth()` function to disable anti-aliasing. However, this has the effect of also causing all text on the screen to no longer be smoothed, and we couldn't find a way to disable anti-aliasing for only images and not also text.

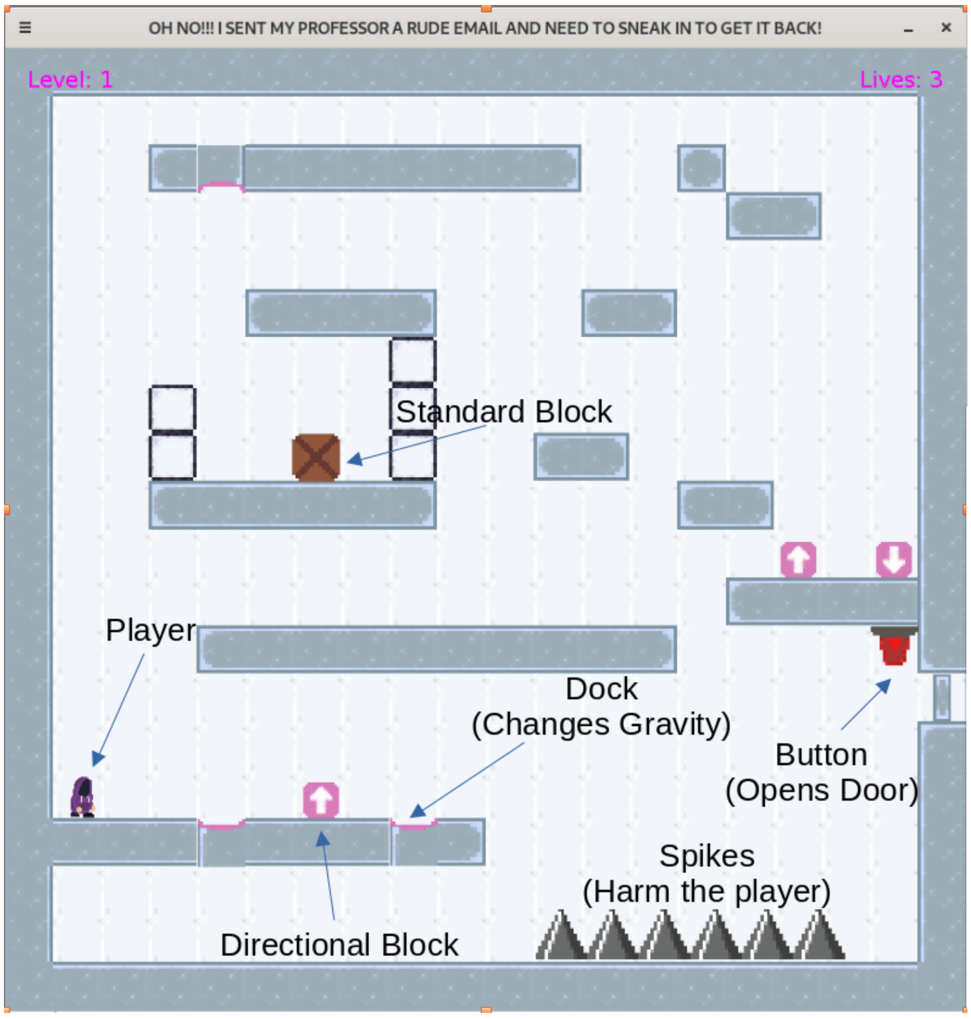
2.2 Rules and Mechanics

The core concept of our game involves being able to modify the rules of the game using certain blocks. The player can pick up these blocks with a button, and throw them with a constant velocity using the same button.

We only ended up implementing a few of these rule changes in the final product; our final list was:

- **Gravity** - being able to change the direction of gravity using blocks with arrows on them. Unlike with VVVVVV, this rotates the whole screen to make it much easier to parse which direction the player should move.
- **Bouncy** - certain objects or surfaces can be made bouncy - in one particular level, the player is made bouncy in order to cross a certain gap that they couldn't cross by simply jumping.
- **Fragile** - certain objects can be made fragile, so that when they collide with another object they are destroyed.

In addition to this, objects can also be used to press buttons that open doors, or can be used as platforms to jump higher. It's up to the player to decide how to use the objects to complete the level.



Annotated Screenshot of Level 1

Level Design

There are 7 levels in total, each represented by a single map that the player can view in its entirety throughout the level. Each level aims to incorporate or introduce a mixture of rule change types or enemies, which, when combined, create a challenge/puzzle that the player must complete in order to progress to the next level. Although no specific approaches were taken when deciding what is incorporated or how the platforms are organised on the map, inspiration was taken from some of the games that influenced our own. For instance, in Baba is You, misdirection techniques are used, which we have also included to increase the difficulty of the game and encourage more level exploration.

3 Implementation

Core Structure

Our project is implemented as specified in Processing, but the game uses the Dominion ECS engine to help manage entities on the screen. As such it can't run as normal in the Processing IDE, and needs to be structured as a standard Java application using Processing as a library, by extending the PApplet class.

The Dominion ECS (Entity Component System) engine allows for efficient managing of objects called **entities**, each of which can have multiple **components** that define their behaviour or other properties. These are managed by **systems**, small functions that run in a multithreaded context in order to provide calculations such as collisions, physics and controls. This allows us to keep behaviour elements small and composable, so we can mix and match different components based on what behaviour an entity should have.

In addition to this, the game reads in its level data from the Tiled[5] application, an open-source tile map editor. This is handled by the TiledReader library[6] since the official Tiled java library is out of date and we had dependency incompatibility problems. We tried to make as much as possible directly linked to the Tiled editor, for example encoding relationships between objects as custom properties in the editor itself.

Physics

Two forces affect the bodies within the game, Gravity and Drag. A Body is any entity that has a Velocity component as any other object is stationary and therefore should not be moving at all. Gravity is a simple acceleration applied downwards to the entity's Velocity, and Drag is the opposite of the velocity vector, scaled down by multiplying by a small float. A multiplier is applied to this that increases the drag when the object is on the floor, or sliding down a wall. This mimics the friction that would be present in reality. These simple physics calculations allow for a realistic physics simulation while being relatively cheap computationally. All changes to the velocity of an object are scaled by the time in between frames, meaning that the frame rate does not determine how fast the changes take place.

The player's velocity is influenced by the key presses made by the user. These will create an acceleration of the player's body up to a maximum speed. A jump is a single vertical impulse applied to the player's body.

The other box objects are also affected by these physical forces, they can be moved only by either being pushed or thrown by the player.

[5]https://www.mapeditor.org/

[6]http://www.alexheyman.org/tiledreader/

Collisions

Our game uses a simple Swept AABB[7] collision system, meaning that we check if the object would go through a given object at any point within a frame by predicting its movement. If so, then we can track when exactly within the frame the collision happened, and thereby continue the movement up until that point. If the object is already inside the object then this system doesn't work, so we also use a standard AABB check as a fallback, pushing the object out if it's already colliding.

If an object is colliding with a static object, for example with a tile in the map, then we project its velocity along the tangent direction and set this as the new velocity. However if two dynamic objects collide with each other, for example if a player is pushing a box, then we spread the velocity evenly between the two objects, thereby assuming that both objects are of the same mass.

Interactable Elements

At each level of the game, the player is able to interact with buttons that can be used to manipulate gravity or open doors. From a programmatic perspective, each button contains event handlers/listeners whose event callbacks are dynamically defined upon the creation of the button. More specifically, the callbacks are specified in such a way that the type of actions triggered when pushing and releasing a button are determined by the object that the button references within the TMX file for that level. For example, if the button references a Door object, the callbacks trigger opening and closing behaviour. Whereas if the button references a Gravity object, the same callbacks will trigger changes in gravity. By making the static representation of the callback independent of the type of action triggered by the pushing and releasing of the button, it allows for greater flexibility and maximises code reuse and extensibility.

A single level may contain one or more doors that allow the player to access closed-off areas of the map or progress to the next level. These doors are controlled by an associated button, which the player can push, either by physically being in contact with it or by placing a box on it. The movement of the door is similar to a portcullis gate, which opens in the direction specified in the TMX file. For example, it may be specified that a vertical door opens lowers downwards or a horizontal door slides open to the left. To make door movement look smooth and enhance the strategic element of the game, the position of the door is only updated periodically, creating a slow door opening/closing effect.

Changeable Rules

Any class can be made to be changeable by allowing it to extend the class Changeable_Interface. These classes will have methods that define the behaviour of how they will be changed. This allows for a uniform interface with many different objects in our game world in the same way, allowing for the switching of the targets of the rule switching.

A block has a value and rule type associated with it. And it can be 'docked' in dock objects. The dock objects have a Changeable object that is used to access an object that extends Changeable_Interface. This extra layer of abstraction is needed as Dominion does not support

[7] Inspired by `https://www.gamedev.net/tutorials/programming/general-and-gameplay-programming/swept-aabb-collision-detection-and-response-r3084/`

filters accounting for polymorphism. A block can only be docked in the dock that has the same rule type, this made the implementation easier as not all weird block combinations will need to be considered. When a block is first docked (collides with the docking object), its position will be set to above the dock object, and its velocity will be removed. The velocity is removed so that it cannot be removed without the player picking it up, meaning that it cannot fall out if the rule it is changing is changing the direction of gravity.

Currently, only some possible rule changes have been implemented, but the way that the rule changing is structured should allow for a wide variety of rules to be added in the future with relative ease. It currently supports boolean values (which can change a boolean value), directional values (which can change the direction of an object), target values (which can change the target of a dock), and operational values (which were used to change the state of objects).

A couple of special classes that extend the Changeable_Interface had to be created to produce the desired effects. An example of one is the All_Ground class, which specifically changed the state of all ground tiles. This is an extra abstraction that is needed for changing the properties of multiple objects of the same class.

One of the rules that the player can activate is the "bounce" rule. The rule makes the ground bouncy, allowing the player to bounce around as if on a trampoline. This effect is achieved by simply applying an impulse to the player when it touches the ground that acts in the opposite direction to the velocity of the player. When a player moves to the ground with a very small velocity, it can create jittery, unrealistic behaviour in the bouncing effect. To prevent this, the impulse is only applied if the player's velocity when it reaches the ground is above a certain threshold. Another rule is the "fragile" rule, which is also added as a flag to every collider, deletes the entity when it collides with any other. If both entities are fragile, they mutually destroy each other.

4 Evaluation & Critical Appraisal

Admittedly, we didn't perform a lot of playtesting on the game. We primarily tested individual mechanics through custom developer levels and gave each level a few plays through in order to check that they were at least beatable. Initially, the player physics were far more fast and sensitive to player input, however after some difficulty beating even the earlier levels we tuned them to be a little more forgiving and slow. We also noticed that there is a slight inconsistency in the difficulty curve throughout the game, making it somewhat challenging for beginners to learn, build their skills and progress through the levels.That said, trying to create levels that balance difficulty, include interesting challenges, and accommodate players who have varying levels of experience with platformer games requires a lot of time and thought. Due to time constraints, we chose to focus more on implementing critical features rather than focusing on level design.

We also changed a lot about the design of the rule-changing mechanics. Originally we wanted a very freeform system through which you could construct sentences out of blocks in the same vein as Baba Is You, however we realized this was confusing to implement, and difficult to design levels for, especially with the constrained mechanics of a platforming game. Instead we decided to implement "docks", slots that blocks can fit into that have predetermined effects. More complex behaviour can be achieved by chaining these docks together, where one dock sets the target of another's effect. This is more restrictive in terms of gameplay but significantly easier to implement and more intuitive.

The final product is a little rough around the edges, but it works well as a showcase of the mechanics at play. The core system is very flexible in design, and can be applied relatively easily to many different mechanical changes.

Unfortunately, we perhaps didn't explore the full range of possible interactions between these mechanics in the level design. Given more time, we think we would have designed more levels and implemented some more modifier blocks - particularly a "direction locking" mechanic was planned in which the player wouldn't be able to move in a certain direction unless a block was replaced, which could have led to some interesting nonlinear level designs.

Additionally, the game could really benefit from some music or sound effects - we decided to focus the time we had on making sure the mechanics themselves worked, but some audio would have made the game feel much more like a complete whole.

5 Conclusion

Overall, we were quite satisfied by the game by the end of the project, at least as a prototype. It shows off the mechanics we set out to show at the beginning, and provides a sizeable challenge in both platforming skill and puzzle solving.

In hindsight, while it helped with structuring the application, the Dominion library we used to manage entities did add some difficulty to the project. The coding style was generally unfamiliar to most of the team, and so it took an adjustment period to get used to it. Additionally, the library is relatively immature, and so there were some strange issues during development that we had to work around - for example, Dominion's `findAllEntities` method doesn't always return all entities in the world, so we had to add a tagging component to entities so we could use `findEntitiesWith` instead.

Additionally, a hierarchical system for entities would have been very useful - in other game engines like Unity or Godot, entities are arranged in a tree, and things like transformations can apply to a whole subtree of the scene graph. Our system doesn't work like this and instead functions more similarly to an engine like Bevy, which doesn't include this tree structure by default.

That said, if we were to develop it further we would likely design more levels, and implement more unique mechanics, for example being able to control objects other than the player in the same vein as Baba Is You, or perhaps more interesting enemy types as in the Super Mario Bros. games.