CS4303 - Video Games Practical: The Game

Design

BiSchemes is a puzzle-based platformer game, in which the player moves through a two-tone environment and interacts with it in order to open paths and reach the exit.

The player is a small stickman sprite which can run, jump, wall-jump, interact with levers and portals, and push blocks around. The idea is that this limited set of controls are the minimal amount of control required in order to interact with an environment in interesting ways. Button interactions via levers and such allow a range of behaviours, and direct push/force-based interactions further expand the range of intractability by inviting the player to explore parts of the levels and discover ways of interacting with some new elements.

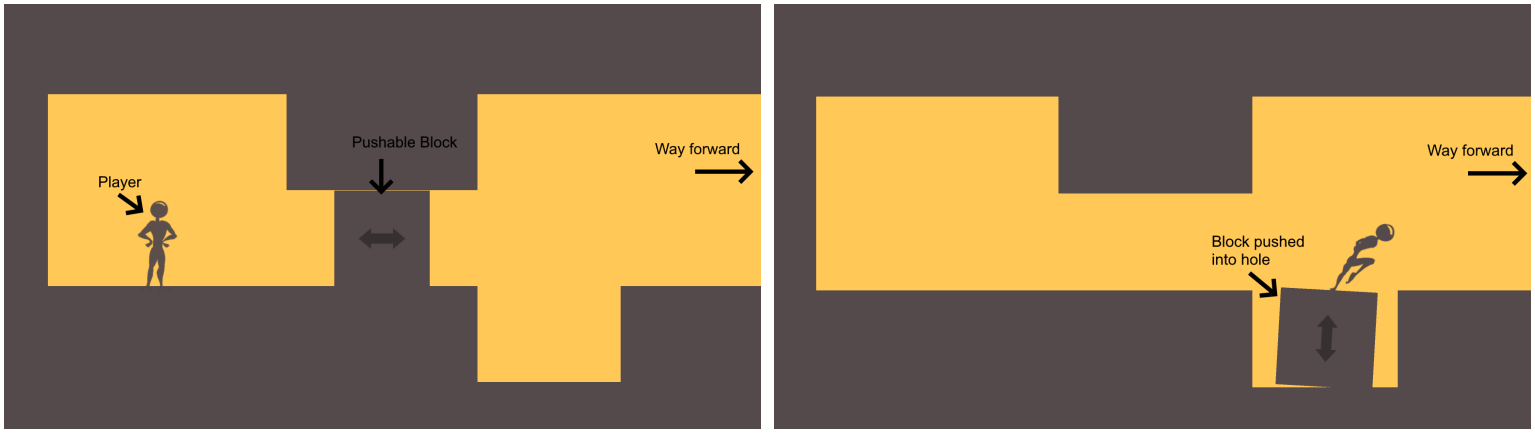
Each level is a two-tone environment with select colours, and is made up of a series of rooms, which can each be accessed by reaching the end of another room. Room size is variable as well, so the gameplay style may switch from static rooms to scroller, depending on the room the player is in. Room-based levels achieve three main objectives. First, it is a throwback to retro platformer games such as Prince of Persia, in which at the time the computational power of a personal computer was not enough to display side-scrolling. Second, level-based design introduces level progression in an intuitive and comprehensive manner - even casual players would know that the deeper down the rooms the player is, the closer to the exit they are. Third, level-based design introduces an extra puzzle mechanic in terms of how rooms are linked between each other. Indeed, one may show an inaccessible part in one room, that may be accessible later. More intricate level design may also allow rooms to loop unless some requirements are met. Thus, a room-based level design introduces a nice aesthetic and makes sense puzzle wise for BiSchemes.

One important discussion point in BiSchemes is the meaning of colours in an environment. As a simple, abstract, puzzle game, colours cannot hold complex meanings like in games such as Gris. However, they may represent one of the most basic components in games: difficulty. An easy level may have colours such as green and blue. A more complex one may have a yellow colour. A harder level would have a pink or red tint. Here, a level's colour is chosen based on two criterias: level difficulty, and aesthetic. A colour should represent difficulty, but it must also look good. Hence why colour is chosen not randomly, but curated on a per-level basis.

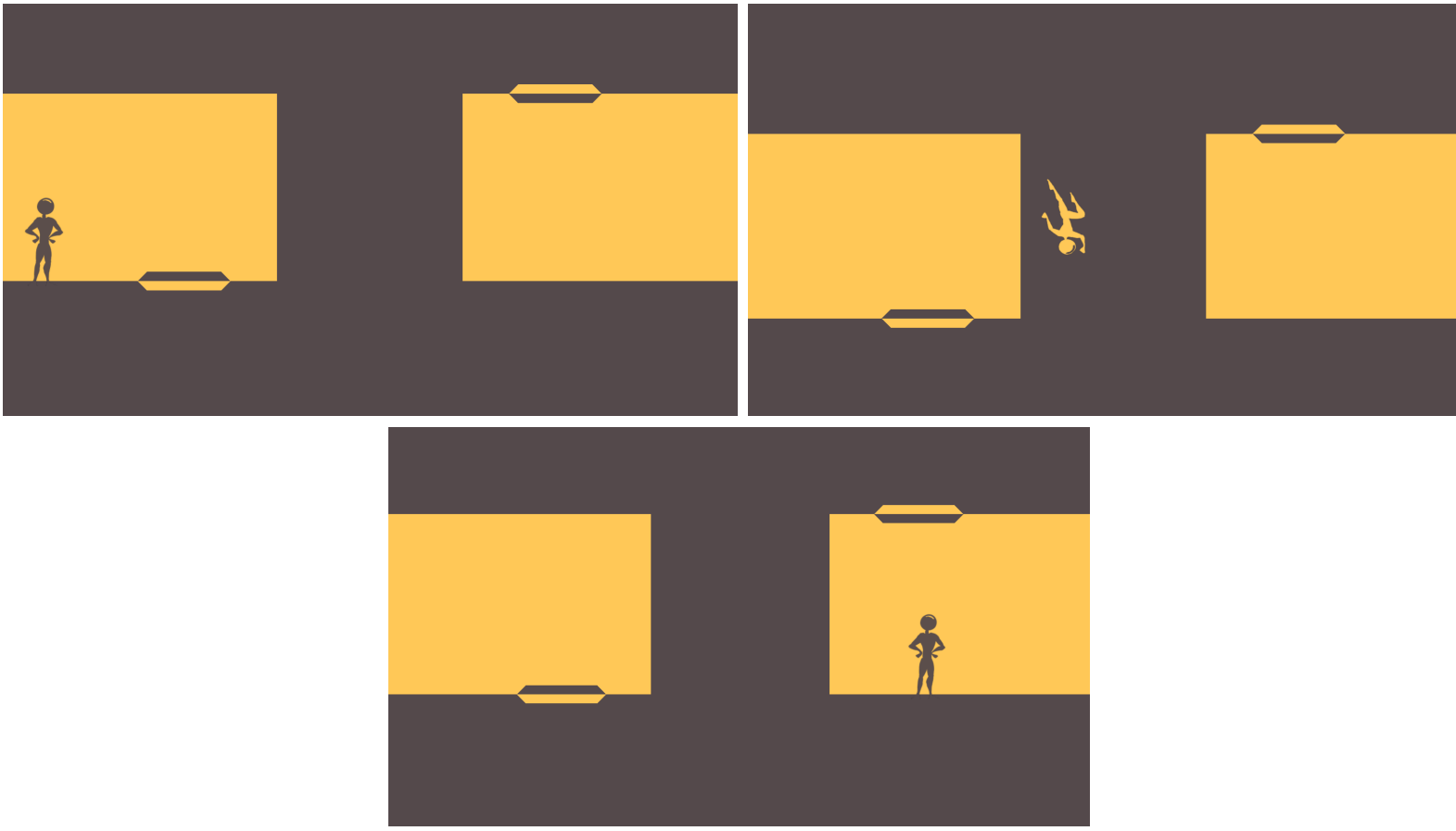
Aside from just being a simple puzzle-based game, BiSchemes is also heavily extendable. Indeed, all levels are specified in a JSON format, making it easy for anybody to create their own levels, add it to the list of maps, and play their own maps after selecting it in the level selection screen. One of BiScheme's main strengths is its extensibility thanks to the JSON level format, and every level in the game is specified that way. Level description is made this way because it makes it easier to make and try new maps - it also allows some level of content creation for consumers. In a puzzle-game, the amount of content is limited, as procedural interesting puzzle generation is not realistically doable. Therefore, it is important to be able to produce content over time in a game, hence the decision to design BiSchemes levels in such a way.

Last but not least, one of the greatest strengths of BiSchemes is the physics system that runs the game behind the scenes. BiSchemes' engine and physics system runs the physics and logic behind all components, and mainly the player and other blocks around

levels. The player and the environment are affected by gravity, may collide, move and rotate. This means interesting physical interactions may happen within the game. In a puzzle-based game, this physics system allows new, interesting, complex, and intuitive ways to interact with the player's environment.



Annotated display of simple BiSchemes, demonstrating a player using properties of the physics engine to push an obstacle out of their way and proceeding with the level.



A simple puzzle where the player must pass through the portal to enter the darker geometry (grey) then return to the original geometry (yellow) through a second portal after going through the obstruction

Implementation

Engine

The main way the whole game runs is via an ensemble of physics and logic systems boiled down to a game engine. The game engine deals with updating behaviour of each component at every frame, rigid body rotational physics, collision detection, and rendering. It is one of the most critical parts of the game.

Physics System

Firstly, the rigid body physics system dictates how objects collide and resolves those collisions. This physics system uses rotational physics. This means that all objects in a scene may contain a rigid body with properties such as mesh, mass, inertia, surface properties… These properties define an object in the world. Then the following things happen:

- Integration - forces are integrated into velocity and rotation of rigid bodies for which a force has been applied to. Forces are applied using a force generator, which has access to the rigid body properties and can assign an appropriate force at each frame.
- Broad-phase collision - This step of a physics engine checks which objects may collide between each other. The best kind of physics engine is one where the least amount of collisions are computed, therefore this step is important to achieve correct performance. In this engine, this is done via an uniform grid, for which all objects take some amount of space in the grid. When an object moves, it takes up other parts of the grid. If a cell in the grid ever has more than 1 object reference in it, then there is an overlap, and the objects within that cell must be checked more thoroughly for collisions.
- Collision detection - The precision collision detection step works in 3 different cases - circle to circle, circle to polygon, and polygon to polygon. The hardest case is polygon to polygon, which requires checking using the Separating Axis Theorem. Furthermore, to get better contact points for later steps, Sutherland clipping is done on the polygon to differentiate edge to vertex and edge to edge collision. This is an important step as differentiating between both allows for more satisfying collisions in the physics system as this tackles some inherent flaws of sequential collision resolution.
- Manifold creation - The manifold is the data structure that stores all the details about a collision - the contact points, the collision normals, the affected primitives, etc. Here, to simplify computations later on, manifolds can be combined if they possess the same primitives or rigid body parents, and take care of reversing the collision normal depending on which entity is considered the first entity of the manifold.
- Collision resolution - The last step of the physics system is the resolution of all collisions by applying impulses to movable objects. This step computes the effects of

the collision and the separation of forces into linear and rotational forces. Furthermore, this also computes the effect of friction to the objects. Finally, those impulses are applied on the rigid bodies which move accordingly.

These steps each require some more complicated elements such as parameter fine-tuning, which may be considered a dark art of what hyperparameters contribute to the best simulation. Although some further improvements can be made, the system seems to work fine for this game.

Scene Handling

To handle the particular difficulties of this game, the engine supports a multi-scene, one camera system, in which 2 scenes, with different physical worlds, are drawn into the same world coordinates, and are affected by the same camera. Although that means that the camera is not an object of either scene, and cannot have forces applied to it, this method is very advantageous for this kind of game as 2 different colours can be separated into 2 scenes, each with their different physical world.

Logic System

To handle in-game logic, the game possesses some event-based logic with EventCallbacks, which may be possessed by any object, and allows them to be attached to some event callers, which will later call the method when the specified event happens. Furthermore, the engine's game object class GObject contains some overridable functions *onHit* and *update*, which are respectively called when a game object's rigid body hits another rigid body, and on each new frame. This event-based logic allows all of the behavioural system of the game, giving it its power.

Rendering

Finally, the last component of the engine is the way it renders 2D objects into the scene. The system works by keeping track of all visual attributes in the scenes, pre-sorting them by size, and drawing all the visual attributes from biggest object to smallest. To avoid sorting the list at every frame, the list is constructed and sorted only when new objects are being added, or old objects removed from the scenes. In the context of BiSchemes, drawing objects using the painter's algorithm, from biggest to smallest, makes sense as it is almost guaranteed that smaller objects will want to appear in front of bigger ones.

Levels

BiSchemes consists of several individual levels, each Level features one or more rooms with each Room being composed of static geometry and complex objects.

RObject and Behaviours

RObjects are an extension of the base GObjects used in the engine and are used to represent the components of game rooms beyond static geometry. RObjects feature unique

(per room) IDs, boolean state, and a collection of special Behaviour objects. Behaviours define additional properties and features for RObjects and different Behaviours may be combined to create RObjects with unique capabilities.

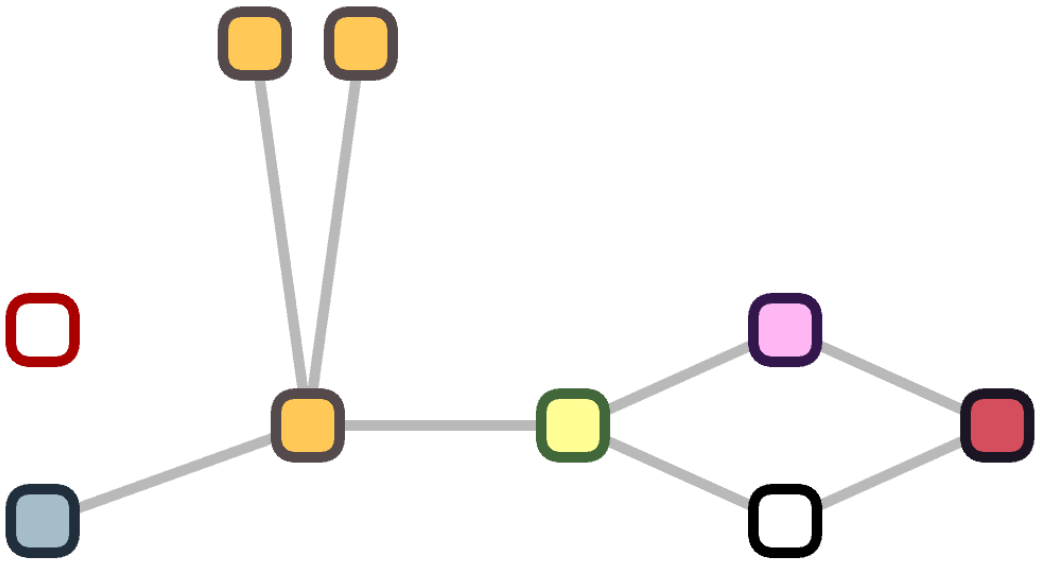
Each Behaviour features a run() method which is called upon reaching certain conditions; there are four types of Behaviour, each with their own conditions: BHit runs when a RObject collides with another; BState runs when a RObject switches its internal state; BUpdate runs every frame through GObject's update() method; and BInteract (an extension of BUpdate) runs when a player enters an Interact command whilst near the RObject. Each of the core 4 types are extended to allow more specific actions upon the calling of run().

Overall there are 12 different Behaviours, 11 of which may be individually customised and combined with other Behaviours to allow for complex RObjects.

Parsing

All the levels, their rooms, and those rooms' individual geometry and objects are parsed from JSON files. The provided JSON guide gives an in depth description of building levels but as an overview, the parser allows for the customisation of room geometry, the placement of objects (either: preset objects like blocks, doors, or levers; or custom objects made with specified physical properties and behaviours) and how the room connects to other rooms.

For all parsed Level, by observing the dependencies of levels (which levels require completion before others may be attempted) a directed acyclic graph is generated and presented in the user interface displayed upon start up of the game.



MapUI generated from JSON Level files present in the levels/ directory.

Context

The game BiSchemes sits in an interesting context. Indeed, puzzle-based games and platformers have since the 1980s fit well together, as many platformer games back then used to have labyrinth-like layouts - like with *Legend of Zelda: The Adventures of Link*, or *Megaman*. This was not always seen as a good mechanic, as those kinds of games were infamous for being very difficult, not only for their layouts, often requiring the user to carefully read the player notice, or to take notes. They were also difficult due to enemy density. Countless are the examples of platformer games which failed due to their incredible difficulty, like *Mutant Ninja Turtles* on the NES.

Nowadays, the platformer genre has evolved into another kind of skill-based system, with one such example being 2018's *Celeste*, which couples interesting gameplay mechanics like dashing into a parkour 2D retro-style platformer game. *Celeste* is one of the main inspirations for this game in the room-based level layout of *BiSchemes*.

However, the puzzle-based mechanic of *BiSchemes* stems from slightly different roots than that of classical platformer games. A great inspiration for *BiSchemes* is *Portal*, due to its teleportation mechanic. Compared to *Portal*, *BiSchemes* doesn't allow any placement of portals, but does switch the players' gravity based on where the portal was. The idea of different environment, different interactions also stems from the video game *Water Boy and Lava Girl*, a flash game in which two players must navigate a static level, solving puzzles in which one player may only interact with some components, and the other player with some other components. However, *BiSchemes* is not a multiplayer game, so the multiple environment system is instead made by having the player switch environments using the portals.

Furthermore, one of the greatest inspirations from *BiSchemes* does not come in terms of the mechanics, but in terms of how the game teaches those mechanics. Here, two opposing philosophies interact with each other. On one hand, games such as *Celeste* are perfect information games - upon entering a room, the game shows the player the whole map, such that the player may be able to deduce a path before attempting it. BiSchemes implements this by showing the player the whole room, letting the player deduce what path it should take. On the other hand, games such as *Legend of Zelda: Breath of the Wild* teach the player in a different way - by naturally leading the player to a waypoint, letting the player try out things, until they discover a new mechanic. The important thing about this is that the player does not feel like they were taught, but instead that they discovered a mechanic of the game. BiSchemes works in a similar way as well, by presenting the player with new obstacles, and naturally leading them to discover new mechanics such as moving blocks. Thus, the teaching style of BiSchemes is a mix between two interesting ways to lead the player to learn the mechanics of the game.

Finally, the idea of switching colours or worlds in a game is not new. *Legend of Zelda: A Link Between Worlds* introduces similar mechanics by being able to enter walls and access areas that were previously inaccessible. *Colour Switch* is a famous mobile game in which the player switches the colour of a ball to phase through obstacles. *Ikagura* is a bullet hell game in which the player may switch their ship's colour to phase between some projectiles and survive. In an interesting way, the game *Sonic Frontier* also introduces the very same mechanic in its final boss fight, showing that such a gameplay style has harnessed some interest. However, those games are all different from the mechanics of

BiSchemes, and as far as we have searched, no well-known game implements a two-tone mechanic such as what BiScheme has.

Evaluation

To test and evaluate parts of the game, various techniques were used. On one hand, the physics system and the engine require foolproof collisions, as not doing so leads to a less-than-desirable experience. Therefore, unit testing was implemented for the physics engine to try out the collision system.

In regards to collision resolution, the art of parameter fine tuning is a difficult one. This one is made by playing the game and seeing what parameters were best - a trial and error.

On the other hand, JSON parsing was made in a similar fashion, by running a program in the ‘levels’ subdirectory and checking if the given levels could compile correctly. It is important to make sure that levels are parsed correctly, hence why a significant amount of care was put behind it.

During game development, a question arose, based on the question of the player’s learning mechanics - a debate on whether to opt in for a more perfect information approach, or a more “trial and error” method. As game development went on, a compromise between both was made - all interactable elements have a symbol on them, and the player should learn how to interact with those.

Finally, due to time constraints, level evaluation had to be limited - therefore testing and evaluation of the levels in the game was made by trying out levels to check if they had any bugs, and letting them through if we saw them fit. However, evaluation of level design was mostly made beforehands by hand drawn design. This is the case of levels 1 and 0, which were designed by hand and evaluated then, before being inserted into the game.

Critical Appraisal

The design of BiSchemes has two major strengths. The game’s physics engine is incredibly advanced and goes far beyond the designs and techniques demonstrated in the CS4303 module: it allows for complicated physics computations involving flexible and configurable rigidbodies and complex (and convex) meshes of vertices; it allows for the modular assembly of primitive structures; and it uses grid partitioning to reduce the number of computations to a manageable and efficient amount. The second strength of BiSchemes is the modular and customisable level design: an advanced and flexible, JSON-based parser is used to read in designed levels; the level parser allows all elements of a level to be specified, including, geometry, room connections and individual objects; and customisable objects can be easily created and allow for complex in game interactions and behaviours.

The main weakness of BiSchemes is how well its two major strengths were combined. Due to time constraints too small a time period was used to connect the physics engine, level design and core game runner meaning there was insufficient time to properly correct unexpected and incorrect interactions between the games elements. This means many features of both the physics engine and level designs go unused due to core features’

validity being prioritised; this means primitive assemblies, circular primitives, and certain level objects (including many behaviours) were unable to be implemented causing them to be wasted.
There are also several object errors present in the game that could not be fixed in the remaining time:; interaction (for portals and levers) indicators do not show when the player is nearby; and features such as spikes and or one-way portals cannot be created due to unfixed errors.
Overall, to prove the full capabilities of the game it would have been preferable to add additional levels to fully demonstrate the capabilities of the game but due to time constraints we were only able to develop two.
Overall BiSchemes has amazing foundations but due to time constraints these foundations are not fully utilised and many bugs could not be removed leading to a sub-par and arguably unplayable experience.