Project Pac: A Bite Out of Time

Team ATLAS (Group 10)
Ross Porter and Sam Morgan

Technical Report

Our Task:

We were tasked with creating a simple video game in the style of Pac-Man. The overall gameplay we based our design off was this: the player character navigates a maze filled with ghosts trying to collect all the pellets in the level before running out of time or lives. We created Project Pac: A Bite Out of Time which took this concept to the next level with; a world map and level progression, a unique storyline, new and familiar characters, powerful boosts and abilities, and much more that will be outlined in this document.

How Project Pac meets the client's requirements:

Project Pac presents the user with a welcome screen that allows players to choose from a variety of characters to use in single player, two player, and three player game modes. As well as a help screen which explains certain features of the game. When the game itself is launched a countdown begins, after which the player can navigate the level, consuming time-shards and are chased by the ghosts, each of which has unique, intelligent AI. Should a player and a ghost collide (not under the influence of power-ups or boosts) the player will freeze and lose a life, then both characters will move back to their respective starting positions. The player has three minutes to complete the level or they lose a life and retain their progress. The game can be paused and an exit dialog is displayed if the ESC key is pressed. Appropriate sound effects are played throughout the game. A flowchart of how the game moves between states can be found in the appendix.

Features that improve functionality of the system:

Project Pac has 6 playable characters who change the way the game is played and the strategies required to beat each level. They include: Pac-Man, Ms.Pac-Man, Pac-Kid, Robot, Snac the snake, and Glitch . Pac-Man and Ms.Pac-Man can eat ghosts for a limited time when a power pellet is picked up, Pac-kid gains charges for his wall jump ability, Robot gains charges for his anti-ghost laser, Snac and Glitch do not get powers from power pellets however they have unique passive abilities. Snac is faster than other characters but he grows in size as he eats pellets, Glitch constantly switches colours and can eat ghosts that are the same colour he is.

Project Pac contains 11 unique levels with colour schemes and specific ghosts which take the player on a journey to undo fix the timeline which was damaged by the nefarious Dr.Clocktopus. The levels are in a tree structure which requires previous levels to be completed to unlock. An autosave file is generated when the game is first opened and read on subsequent openings, this autosave keeps track of the players progression through the levels and which characters they have unlocked. Other save files can be loaded from the launch screen.

To assist the player a limited use boost can be chosen once per level to have a powerful effect these boosts include; Dash(Increased movement speed for a short time), Shield(Can absorb hits from ghosts), Pellet Magnet (Pick up pellets in a much wider radius), Invisibility (Ghosts will move to your last known location then move randomly), Time Slow (Ghosts move at reduced speed), Invert Controls(Chance to activate when random boost selected), and Random Teleport.

Top-Level System View:

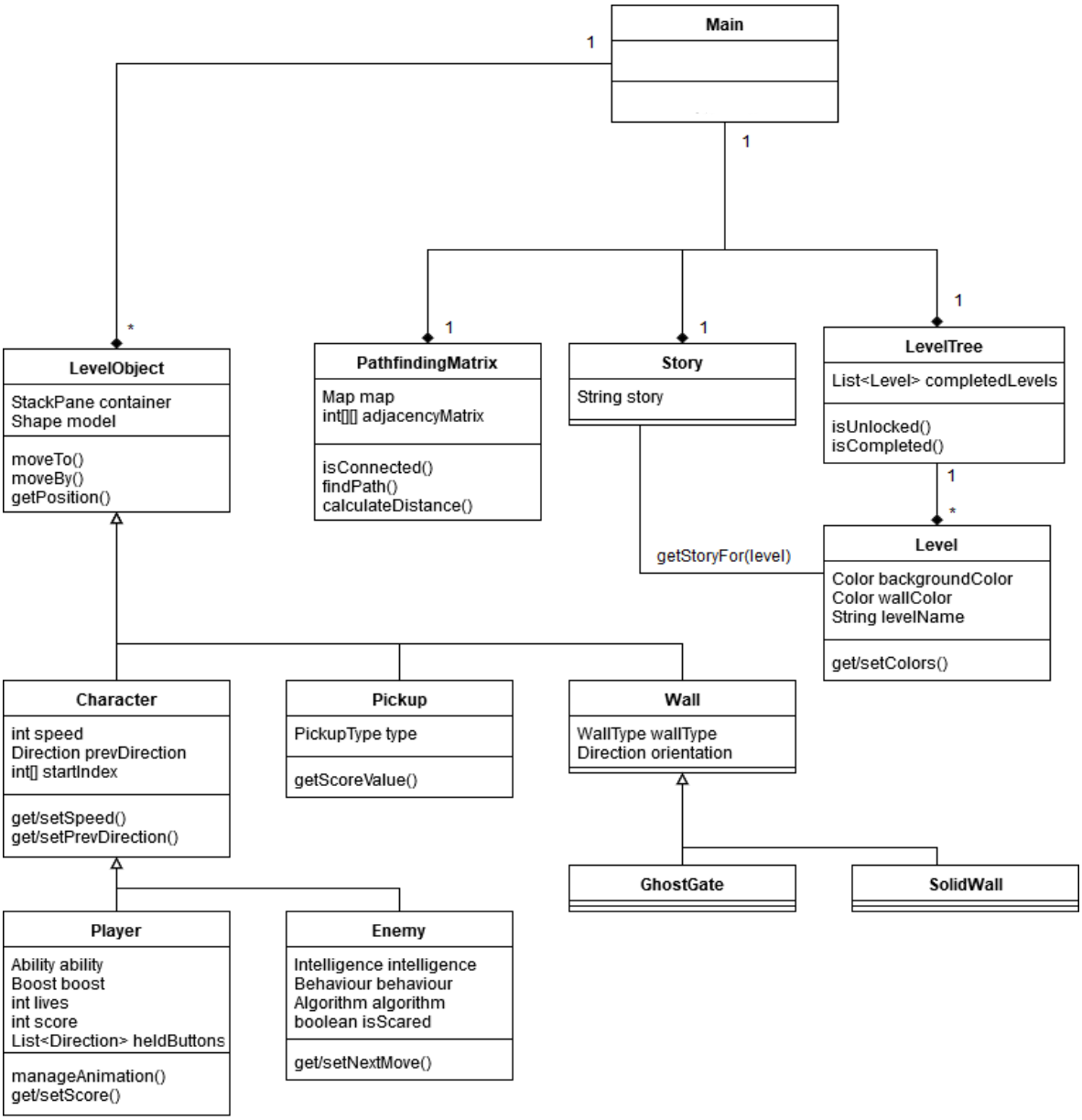

Figure 1 - Final Project Domain Model

Main has an array of LevelObjects (walls, enemies, pickups, etc), a PathfindingMatrix which takes this array as an argument and is used for AI pathing, and a LevelTree which contains all instances of the Level class and keeps track of which levels have been completed and how the levels are interconnected. Story holds all the strings related to Project Pac's narrative. Most of the inputs and outputs and interactions with the user are handled by main. The domain model above is how the final project ended up, this can be compared to our initial domain (which can be found in the appendix).

Suitability of Tools

Java, being an Object Oriented (OO) language, was very helpful for this project, which almost necessitates an OO approach due to how often parts can re-used. If we were to take this project further however, writing in Java rather than a commercial game engine like Unreal or Unity would limit the platforms on which the game would run. For example, game consoles cannot run Java code, but they could run games made in other engines like Unreal since (most have C++ wrappers).

Git was a very useful tool in the development of Project Pac as it allowed us to easily share code between the two of us, work remotely at home, uni and on different computers seamlessly. Git was useful for this project, but I can imagine how having many game assets like high-res textures and 3D models, etc might make git a little clunky. Clearly version control for assets as well as code is good, but if all your assets take up several GB, then constantly having to redownload them onto different machines or whenever changes are made would be time and bandwidth consuming.

Scene Builder was a useful tool to quickly develop user interfaces. In this project Scene Builder 8 and java layouts were used to create how the game is displayed. Scene Builder was good however the FXML integration seemed a little “magic” at times which made it hard to fix when thing went wrong. Just using Java to create on-screen layouts and elements was fast and effective however customising these elements beyond their base values was far slower than Scene Builder without a CSS Style sheet.

Significant Issues

A significant issue encountered was how the tile alignment system interacted with the character models. Since character position was determined by finding the center of the model, changing character models (for animation purposes, power ups, etc.) changed the centre position of the model, this changed the character's actual position on the level, often leading to misalignment errors. I overcame this problem by placing the model in a StackPane with a fixed size, and reading the StackPane’s position, leaving me free to change the model’s size (so long as it’s smaller than a tile) without affecting the character’s position.

Another significant issue was how the characters speed interacted with the tile grid. When a character’s speed is changed (which was a desirable feature) they could potentially become misaligned with the grid. This problem was magnified by the fact that we did not have fine control over speed, as we could only change speed slightly before it had too drastic an effect. We minimised the effect of this by only have three different values for speed that we could handle directly such as implementing checks so that we could modify speed by these fixed values. This approach however does limit the flexibility of our system.

Future improvements:

1. Currently, a model can only be the same size, or smaller than a tile. Further reworking with how the character's position is calculated would allow for adding models larger than one tile. Giving us greater freedom with character designs which could facilitate boss characters.
2. The Main class handles a lot of the game currently. In future, it should be split this up further. For example, it would be much cleaner if there was a Game class that handled all of the gameplay logic. This would leave Main to only deal with the highest-level logic, such as switching between scenes.
3. Animation also needs to be done more rigorously. It was added quite late in the development cycle, animation was largely shoehorned in, hence why all players animate the same way and enemies currently have no support for it. Next time we would likely give animation its own class as each model should animate differently and thus we would be able give each model a unique animation.
4. Score currently has a very minor role in the overall game, in the future a time bonus and a high score leaderboard could make score a more compelling feature to the player and lead to potentially more replayability.
5. Investigate CSS Style sheets and invest more time into overall application aesthetic.

Coupling and Cohesion

Barring the Main class as mentioned previously, most of the classes are responsible for a single task, such as LevelObject, whose job is to handle displaying a model on the screen and moving it around. Subclasses such as Character add support for LevelObjects designed to repeatedly move by offering fields like previousDirection, which returns the direction in which the character last moved.

Enemy and Player add further relevant controls, such as dealing with AI behaviour status, or managing player powerup statuses.

Other examples of high cohesion classes include the Sound class, which is solely responsible for playing sound. The TreeNode and SetArrayList class are both data types which hide complexity and offer simple methods for getting useful information, etc. LevelTree and PathfindingMatrix then use these data types to solve specific problems.

We attempted to minimise coupling where possible by using getters and setters, which let us hide implementation details in classes, and change them later with (hopefully) minimal changes. An example of this was where position checking was changed from checking the character model position into checking a StackPane's position instead. Only things within the LevelObject class needed to be changed in order to make this work.

An example of high coupling was when the Wall class was made abstract and subclasses GhostGate and SolidWall were made. I had been checking for walls using (object instanceof Wall) rather than any getter or setter. This meant that I had to go through all of the code and change all of these to (object instanceof SolidWall). Luckily there were only a couple of cases where this occurred.

Software Development Methodology:

For the most part, we seemed to follow the Waterfall methodology, in which we set our requirements and planned most of the project beforehand, and then began coding in features as we went. We stuck to our initial design plan fairly well however we implemented the save system much later than we expected as it was unnecessary until we had more maps and characters to unlock. We did fall slightly behind initial schedule which was somewhat expected as we were both unfamiliar with Java and JavaFX and could not predict the difficulty of implementing certain features accurately.

As this was a two-man team, we made sure to meet up at least once a week to discuss what we have implemented (and how it works), what we should do next and to check our progress against our schedule.

Also, when possible we pair-programmed features so that we could bounce ideas between ourselves, to check our logic, and make the most of having two developers. This was useful because it made sure that we were never implementing the same or similar features, we were always up-to-date in understanding how the code functions, and we could show each other what we had learned so that learn Java as fast as possible.

Discussion of the game design experience:

As our first game design project and first-time programming in a team we encountered and overcame many programming and design challenges (aside from our inexperience with JavaFX). For example, in our initial design document, we foresaw level development as a time-consuming process which was true but not in the way we thought. The initial design of each level took a relatively small amount of time; however, each level required a substantial amount of playtesting to identify what works and what doesn't. After the slow start, we found ourselves adapting quickly to the Java based environment and were exploring the different tools we had at our disposal.

The feature list we wanted to implement at the start of the design process did not make it to the final product in its entirety as we learned our limitations in ability and time, and we had to choose which features would have the biggest impact to the user experience.

We have a large amount of experience playing games as users and found knowledge and experience this surprisingly helpful in considering how the game should flow, which features will users most enjoy, and how to make certain features more impactful for the user. Overall we found this project very informative on capabilities and limitations of java and multi developer software design with Git.

Appendix:

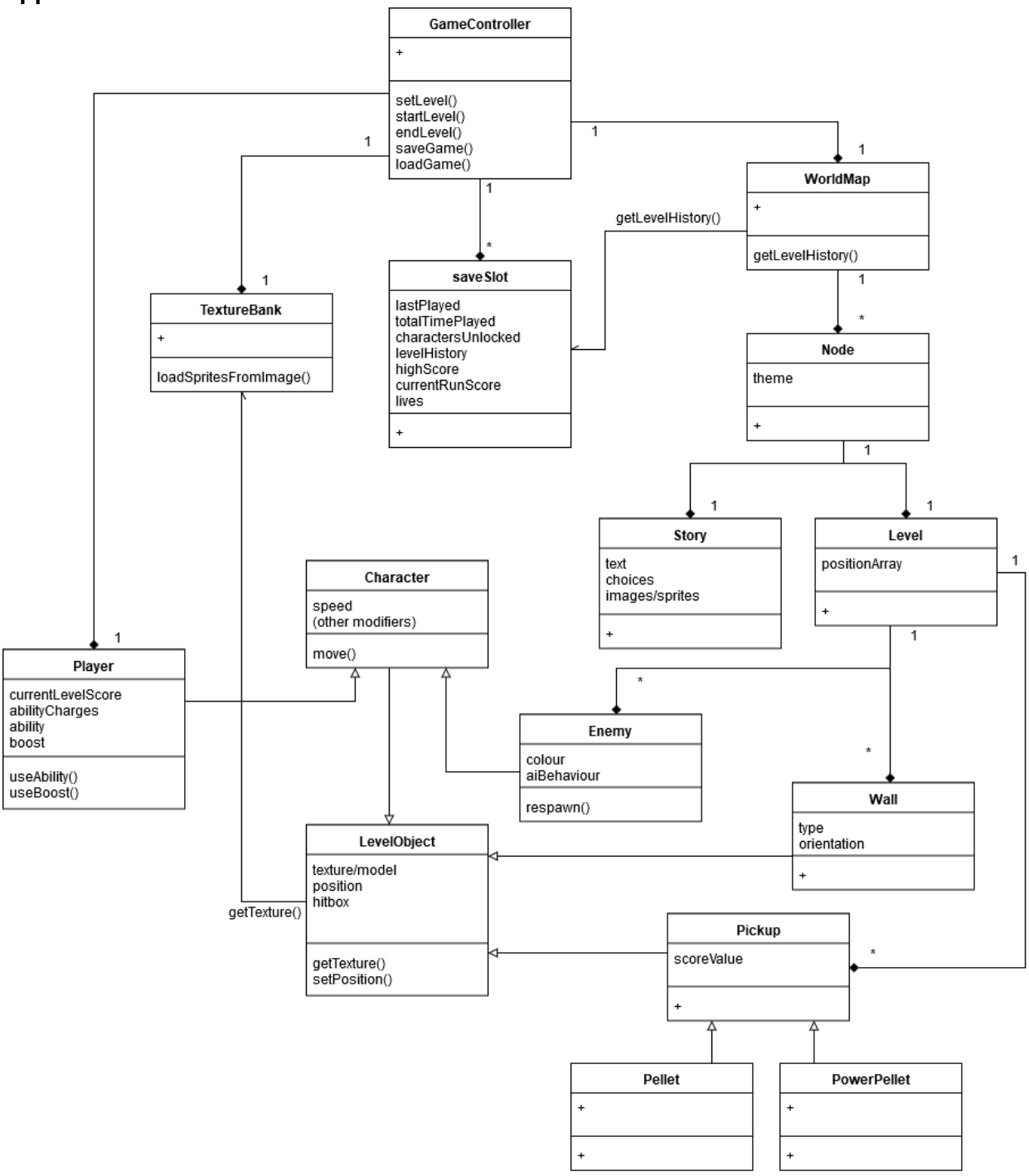

Figure 2 - Initial design domain model

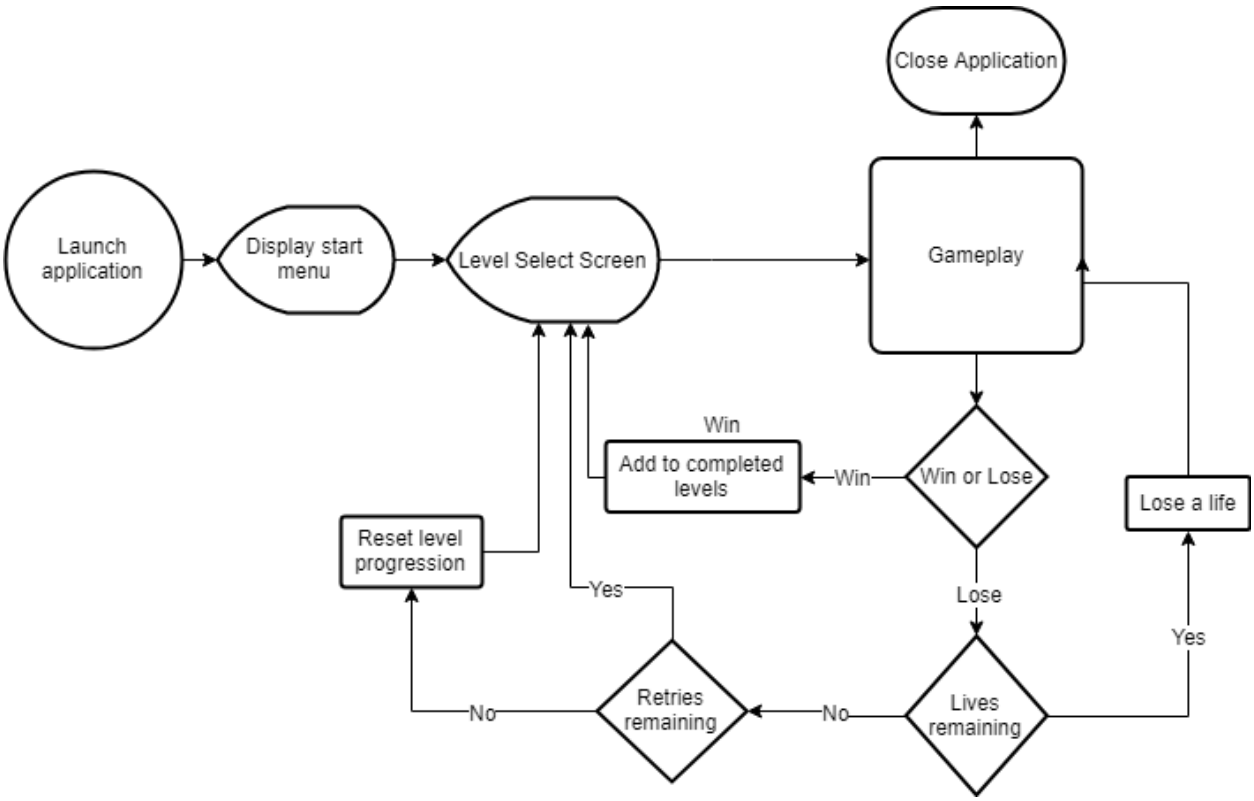

Figure 3 – Basic game state Flowchart