DRMBM (1994) remake

Christopher Ashford

March 2023

Contents

1 Analysis 2
1.1 Introduction and Background . . . 2
1.2 Alternative Solutions . . . 2
1.2.1 Emulation . . . 2
1.2.2 B Puyo . . . 3
1.2.3 Puyo Puyo Tetris 2 . . . 5
1.3 End User Input . . . 6
1.4 Input, Data Processing, Output . . . 12
1.4.1 Goals . . . 13
1.4.2 Extension objectives . . . 15

2 Documented Design 16
2.1 Language and rendering module . . . 16
2.2 Engine classes . . . 16
2.2.1 Entity . . . 16
2.2.2 Scene . . . 17
2.2.3 Engine . . . 18
2.2.4 Class diagram . . . 20
2.3 Other engine scripts . . . 20
2.3.1 files.KAP . . . 20
2.3.2 Other files.py functions . . . 22
2.3.3 pbar.py . . . 23
2.4 Classes for gameplay . . . 23
2.4.1 Grid . . . 23
2.4.2 BeanQueue . . . 25
2.4.3 Bean . . . 26
2.4.4 GravityBean . . . 26
2.4.5 FallingBean . . . 26

3 Technical Solution 28
3.1 Features . . . 28
3.2 kris_engine/__init__.py . . . 28
3.3 kris_engine/files.py . . . 39
3.4 kris_engine/pbar.py . . . 46
3.5 exercise.py . . . 48
3.6 gameplay.py . . . 49
3.7 main.py . . . 66

4 Testing 67

5 Evaluation 68

Chapter 1

Analysis

1.1 Introduction and Background

Dr Robotnik's Mean Bean Machine is a 1994 Westernised port of Puyo Puyo for the Sega Genesis/Mega Drive. It is a game that I have enjoyed throughout my childhood on many different forms – cheap emulation consoles, the Sega Mega Drive Collection for Xbox 360, using Fusion emulator on PC among other forms. However, all of these present glaring issues that directly affect the enjoyment of the player – emulation consoles usually are slow with clunky controllers and are not good for much else and thus are not practical to use permanently; the Sega Mega Drive collection on Xbox 360 suffers with a noticeable input lag problem, with inputs sometimes taking hundreds of milliseconds to be processed, directly affecting how fast you can play; PC emulation either results in a small or blurry image and makes it difficult to play with others or share your scores and achievements.

The goal of this project is to solve these problems by creating a superior, native PC remake of the game. Everything in the original game shall work exactly as in the original, including reconstructing the algorithms used for the playstyles of the various AI opponents. I also intend to include many quality of life improvements to solve the problems listed about: multiple customisable input method and handling will be supported, many algorithmic optimisations shall be made to improve performance, graphics shall be upscaled in a way that remains a crisp pixel look instead of introducing blur, an SQL web server will allow score and time leaderboards to exist and a replay file system shall be introduced to allow players to easily share gameplay. This project exists to create a superior version of DRMBM for a new generation to enjoy, as well as offering a way for modern Puyo Puyo players to enjoy the OPP rule set on modern devices.

If the goals above are reached, further extension goals include the introduction of my own custom AI opponents with algorithms designed for optimal, "perfect" gameplay and the use of web sockets to facilitate real-time online matches between two remote players.

1.2 Alternative Solutions

In this section I shall present my research on other Puyo Puyo games, compare the advantages and disadvantages of different versions from the perspective of the end user and take inspiration for my own project.

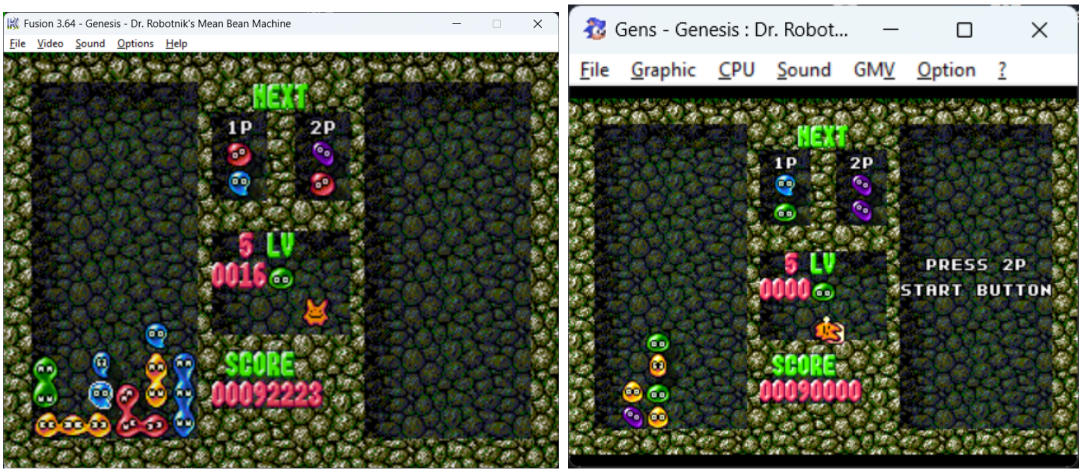
1.2.1 Emulation

Link: cannot be provided due to specialised hardware being required to dump the ROM. Yet another disadvantage.

Many different emulators exist for the Sega Mega Drive, such as Fusion or Gens shown above, or the official Sega emulator that can be found on Steam. These are programs that accept a binary ROM dump of the original cartridge and attempt to emulate the code.

Advantages:

- Convenient for mass production and distribution. Sega can create one Mega Drive emulator and release an entire of library of games that use the same program
- True to the original experience. Since you are playing a copy of the original game, you can be sure you are getting an authentic experience


Fusion 3.64 - Genesis - Dr. Robotnik's Mean Bean Machine
File Video Sound Options Help
NEXT
1P
2P
5 LV
0016
SCORE
00092223
Gens - Genesis : Dr. Robot...
File Graphic CPU Sound GMV Option ?
NEXT
1P
2P
5 LV
0000
PRESS 2P
START BUTTON
SCORE
00090000

(a) Fusion emulator

(b) Gens emulator

Figure 1.1: Some examples of emulators running the game

- While clunky, save states allow you to save high scores and progress through the story, as well as letting you manipulate sequences of beans

Disadvantages:

- Resolution is locked at the console's original and upscaling is blurry and unappealing
- Very static and not customisable. It is incredibly difficult to edit a ROM if you wanted to play with, for example, different handling or textures
- Saving progress is difficult
- Emulators are difficult to run and can easily lag on lighter hardware, running the game at higher levels can struggle on older processors
- It is impossible to play with friends remotely (or if it is possible, then it's too difficult for the average user to achieve)

1.2.2 B Puyo

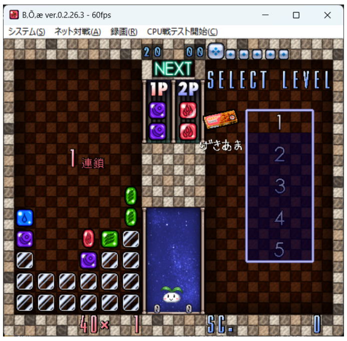
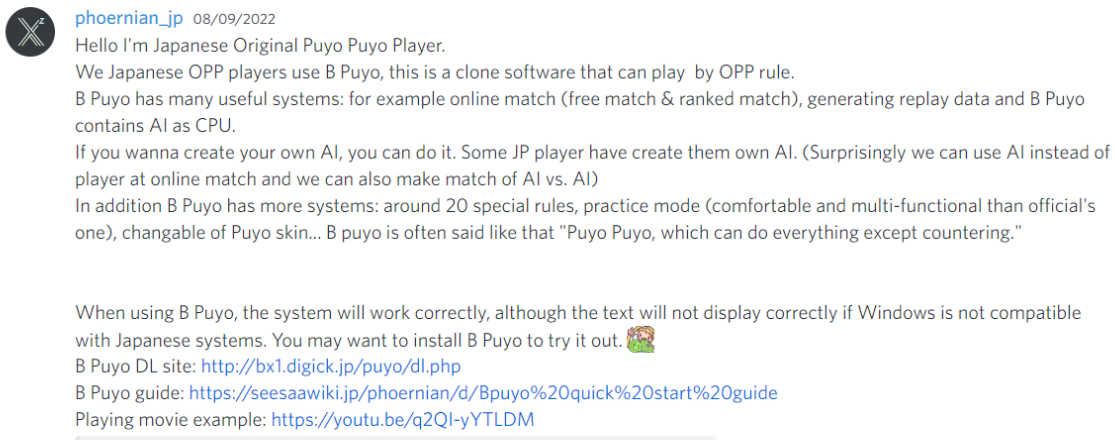


Figure 1.2: A screenshot of B Puyo. Some text is broken running on an English computer.

Link: http://bx1.digick.jp/puyo/dl.php
B puyo is a popular online Puyo-clone recommended to me by the Japanese community.

Advantages:

- Custom textures, custom AI, custom rules, custom anything really
- Easy to use online multiplayer
- Great performance as a native PC program



phoernian_jp 08/09/2022

Hello I'm Japanese Original Puyo Puyo Player.
We Japanese OPP players use B Puyo, this is a clone software that can play by OPP rule.
B Puyo has many useful systems: for example online match (free match & ranked match), generating replay data and B Puyo contains AI as CPU.
If you wanna create your own AI, you can do it. Some JP player have create them own AI. (Surprisingly we can use AI instead of player at online match and we can also make match of AI vs. AI)
In addition B Puyo has more systems: around 20 special rules, practice mode (comfortable and multi-functional than official's one), changable of Puyo skin... B puyo is often said like that "Puyo Puyo, which can do everything except countering."

When using B Puyo, the system will work correctly, although the text will not display correctly if Windows is not compatible with Japanese systems. You may want to install B Puyo to try it out.
B Puyo DL site: http://bx1.digick.jp/puyo/dl.php
B Puyo guide: https://seesaawiki.jp/phoernian/d/Bpuyo%20quick%20start%20guide
Playing movie example: https://youtu.be/q2QI-yYTLDM

Figure 1.3: Information about B Puyo from a well known Japanese player.

Disadvantages:

- Will only run on Windows, excluding Mac and Linux users
- The entire thing is in Japanese, with no translation options. Furthermore, servers are in Japan, creating ping issues for non-Japanese players. This is great for the Japanese community, but unfortunately disadvantages me as a Western player
- The resolution is locked to being a small window, making it uncomfortable to use on high-resolution displays

Project GelaVolt

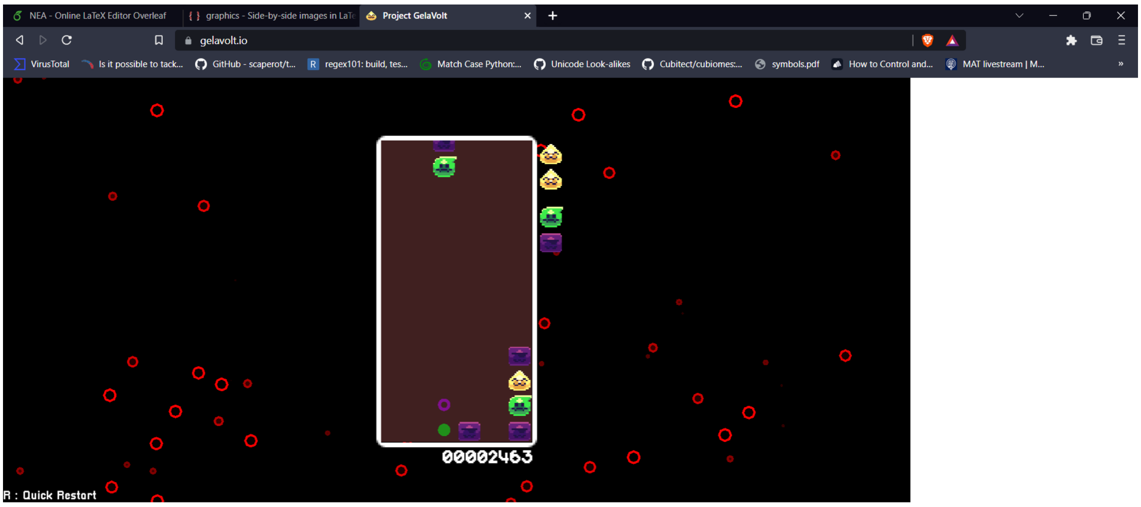


Figure 1.4: A screenshot of GelaVolt running in a chromium-based web browser.

Link: https://gelavolt.io/
To quote the game's creator, "Project GelaVolt is a modern, techno-themed pixel art fangame of SEGA's Puyo Puyo series, one of Japan's most successful puzzle fighter franchises. Currently, GelaVolt is focused on the competitive aspects of the game and it's intended purpose is to help introduce people and help people get better at Puyo Puyo. However, if all goes well, GelaVolt will become a free alternative that plans to solve some of the communities problems: lack of players, lack of crossplay and lack of general quality netcode." It is a Puyo-clone written in Haxe that runs in browsers.

Advantages:

- Appealing design
- Is lightweight and capable of running well in browsers
- Supports many different control schemes out of the box (controller, keyboard, etc.)
- Only version I've played that has hard drop

Disadvantages:

- Multiplayer is in the works but is currently not supported at the time of writing
- Things such as textures are not customisable
- Is unstable and crashes regularly

1.2.3 Puyo Puyo Tetris 2

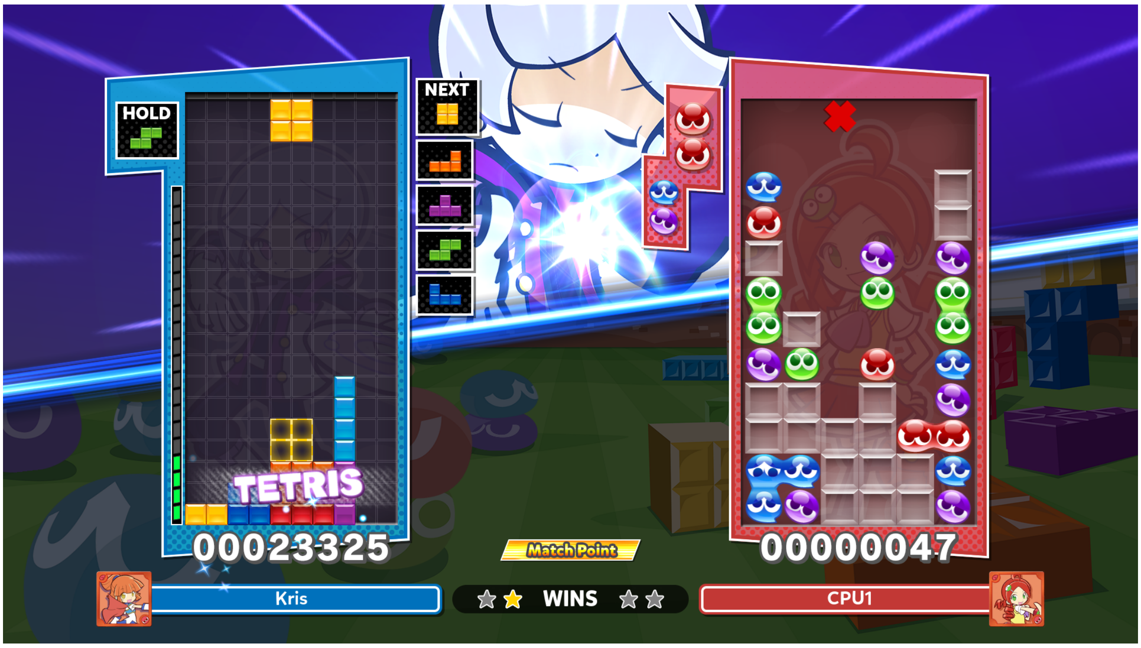


Figure 1.5: A screenshot of a versus battle, I'm playing Tetris and the CPU is playing Puyo Puyo.

Link: https://store.steampowered.com/app/1259790/Puyo_Puyo_Tetris_2/
Puyo Puyo Tetris 2 is the latest Puyo Puyo game released by Sega and combines Puyo Puyo gameplay with Tetris, allowing players of both games to seamlessly play against one another. It has a full story and online mode.

Advantages:

- Cutesy art style is appealing to many, but can be swapped out with unlockable designs
- Being an official release, it is very stable with a consistent online multiplayer
- CPU opponents
- Fully voice-acted story with unique and creative characters
- Active modding community

Disadvantages:

- Ranked multiplayer is fundamentally flawed as leaving matches is not punished

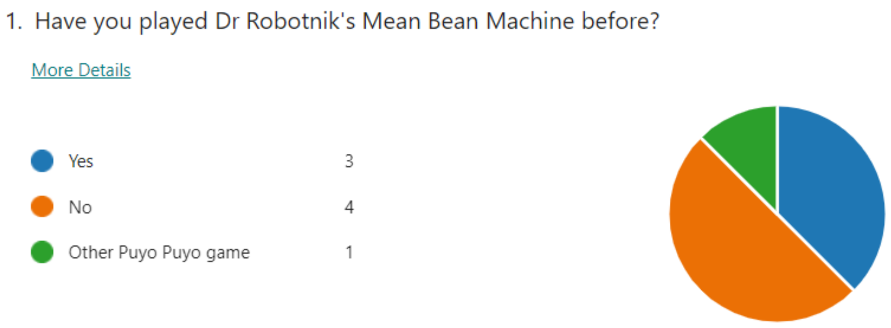
- CPU opponents fail to provide a challenge
- The game is very expensive, whereas all other options listed above are free
- Tsu ruleset, unable to be changed

1.3 End User Input

Being a popular game, many people enjoy the Puyo Puyo franchise, but the best people to survey for this project were the people who were most familiar with DRMBM specifically - speedrunners. A lot of the research in this document was greatly helped by the members of the ”Puyo Speedrun” Discord server, and the contributers to the DRMBM-specific channel they have there. In order to efficiently collect statistical end user input, a form was created using Microsoft Forms, a PDF version of which can be found here: https://github.com/Kris-0605/nea/blob/master/documentation/Survey.pdf
Question 1: Have you played Dr Robotnik’s Mean Bean Machine before?

- Yes
- No
- Other Puyo Puyo game

Only one answer was permitted.



1. Have you played Dr Robotnik's Mean Bean Machine before?

More Details

●	Yes	3
●	No	4
●	Other Puyo Puyo game	1

Figure 1.6: Results to the first survey question.

The only notable thing about this question is that anyone who answered ”No” was taken to the end of the form and was unable to answer any other questions. Thus, only 4 people continued to fill out the rest of the form.

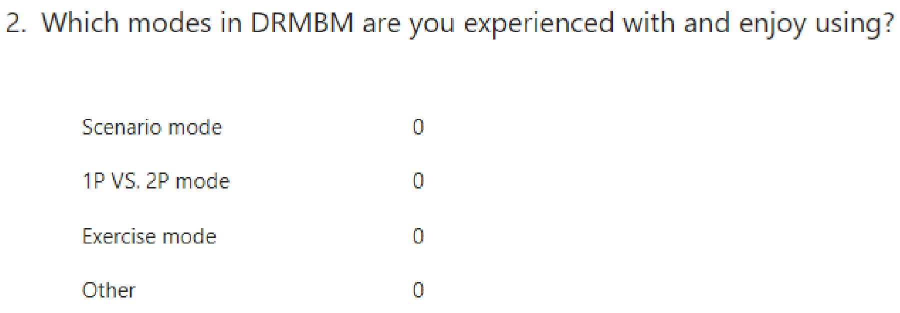
Question 2: Which modes in DRMBM are you experienced with and enjoy using?

- Scenario mode
- 1P VS. 2P mode
- Exercise mode
- Other

Any number of answers were pemitted.
No-one answered this question, thus nothing meaningful is gained from it.

Structure ”This project is intended to both remake the original game in it’s purest form, apply enhancements to it, thus the game will be split into two modes, that will from now on be referred to as ”Classic mode” and ”Enhanced mode”. Classic mode is intended to be an exact recreation of the original game, and Enhanced mode should contain any additions and improvements.”

A message explaining some of the games structure that is important to understand when considering survey questions, which will be discussed further in the documented design section.



2. Which modes in DRMBM are you experienced with and enjoy using?

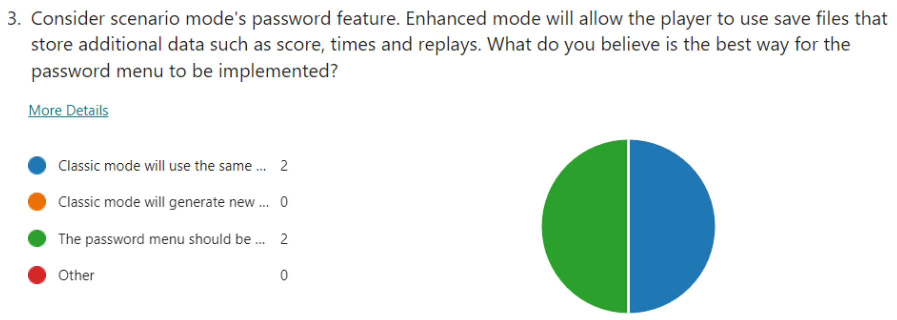
Scenario mode	0
1P VS. 2P mode	0
Exercise mode	0
Other	0

Figure 1.7: Results to the second survey question.

Question 3: Consider scenario mode's password feature. Enhanced mode will allow the player to use save files that store additional data such as score, times and replays. What do you believe is the best way for the password menu to be implemented?

- Classic mode will use the same passwords from the original game in their original form, taking you to a level but not restoring data such as score
- Classic mode will generate new unique password that stores a hidden save file, so that the user is still required to use a password, but this password restores data such as score when used
- The password menu should be entirely replaced by save files in both modes
- Other

Only one answer was permitted.



3. Consider scenario mode's password feature. Enhanced mode will allow the player to use save files that store additional data such as score, times and replays. What do you believe is the best way for the password menu to be implemented?

More Details

● Classic mode will use the same ...	2
● Classic mode will generate new ...	0
● The password menu should be ...	2
● Other	0

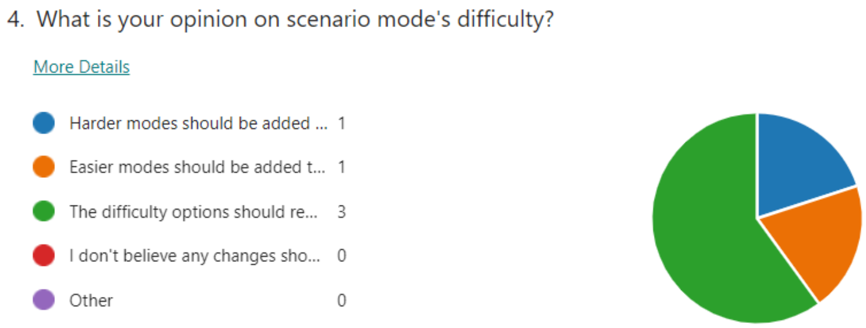
Figure 1.8: Results to the third survey question.

The results to this question were an exact 50/50 split, thus I shall stick to my original plan of having Classic mode use passwords in their original form without any additional data, and using save files for enhanced mode.

Question 4: What is your opinion on scenario mode's difficulty?

- Harder modes should be added to challenge more difficult players
- Easier modes should be added to help new players
- The difficulty options should remain the same in scenario mode, more customisable opponents should be available in a separate "training mode" in enhanced mode
- I don't believe any changes should be made
- Other

Any number of answers were pemitted.
The majority vote represented the solution that I believe would fit best and already planned on implementing: in both classic and enhanced mode, difficulty shall remain the same as in the original. However, in enhanced mode, you can play against customisable opponents, such as the same



4. What is your opinion on scenario mode's difficulty?

More Details

●	Harder modes should be added ...	1
●	Easier modes should be added t...	1
●	The difficulty options should re...	3
●	I don't believe any changes sho...	0
●	Other	0

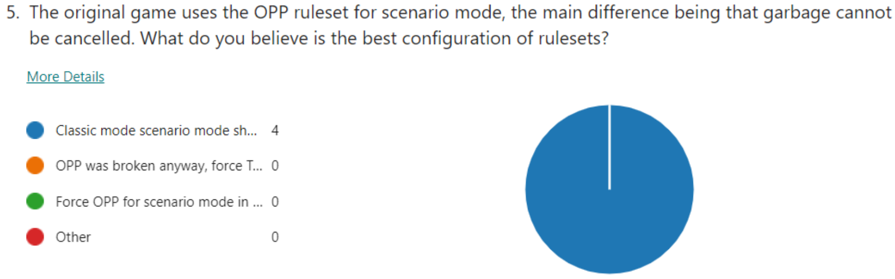
Figure 1.9: Results to the fourth survey question.

algorithms from scenario mode with different speeds, as well as new AI altogether.

Question 5: The original game uses the OPP ruleset for scenario mode, the main difference being that garbage cannot be cancelled. What do you believe is the best configuration of rulesets?

- Classic mode scenario mode should use the OPP ruleset to recreate the original game and Enhanced mode should allow the user to choose before starting a save file
- Force OPP for scenario mode in both modes and allow players to choose Tsu when creating custom games
- Other

Only one answer was permitted.



5. The original game uses the OPP ruleset for scenario mode, the main difference being that garbage cannot be cancelled. What do you believe is the best configuration of rulesets?

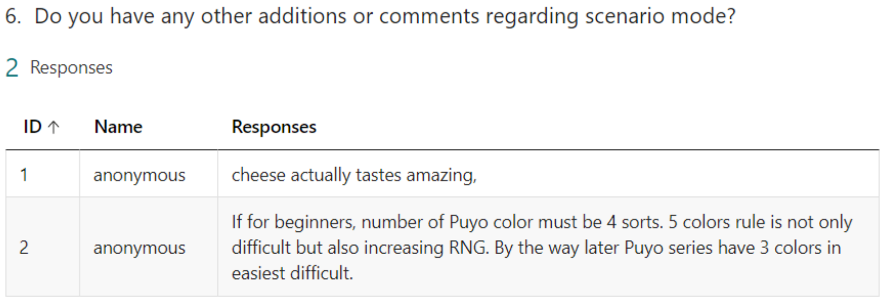
More Details

●	Classic mode scenario mode sh...	4
●	OPP was broken anyway, force T...	0
●	Force OPP for scenario mode in ...	0
●	Other	0

Figure 1.10: Results to the fifth survey question.

The only unanimous result in the entire survey, as well as the solution I was planning on implementing. I will talk more about rulesets in the documented design section.

Question 6: Do you have any other additions or comments regarding scenario mode? This question permitted a text answer.



6. Do you have any other additions or comments regarding scenario mode?

2 Responses

ID ↑	Name	Responses
1	anonymous	cheese actually tastes amazing,
2	anonymous	If for beginners, number of Puyo color must be 4 sorts. 5 colors rule is not only difficult but also increasing RNG. By the way later Puyo series have 3 colors in easiest difficult.

Figure 1.11: Results to the sixth survey question.

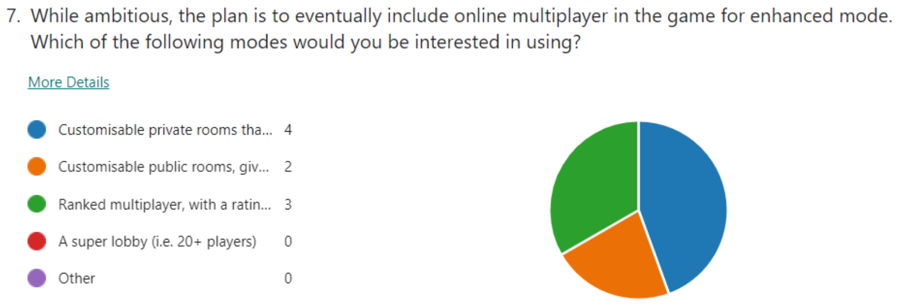
Response ID 2 makes a very valid point. In newer versions of puyo puyo, difficulty settings change the number of colours that appear in play between 3, 4 and 5, whereas being an older game

DRMBM uses 5 puyo colours in all difficulty modes. I will be sure to include the suggestion in enhanced mode.

Question 7: While ambitious, the plan is to eventually include online multiplayer in the game for enhanced mode. Which of the following modes would you be interested in using?

- Customisable private rooms that you can invite other players to
- Customisable public rooms, given in a listing that anyone can join
- Ranked multiplayer, with a rating system
- A super lobby (i.e. 20+ players)
- Other

Any number of answers were pemitted.



7. While ambitious, the plan is to eventually include online multiplayer in the game for enhanced mode. Which of the following modes would you be interested in using?

More Details

●	Customisable private rooms tha...	4
●	Customisable public rooms, giv...	2
●	Ranked multiplayer, with a ratin...	3
●	A super lobby (i.e. 20+ players)	0
●	Other	0

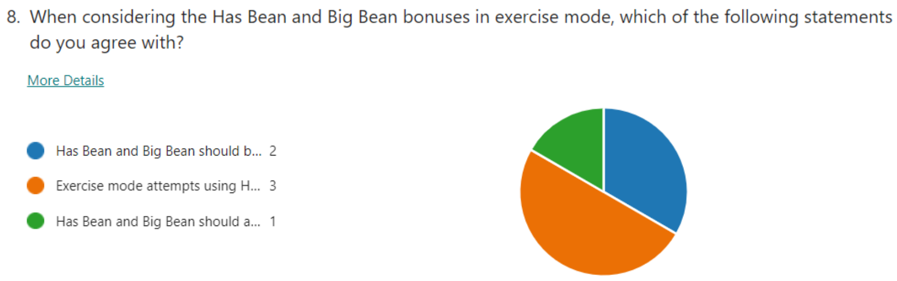
Figure 1.12: Results to the seventh survey question.

All of the above are planned to be implemented, but the distribution of votes gives me a timeline with which to work on each feature.

Question 8: When considering the Has Bean and Big Bean bonuses in exercise mode, which of the following statements do you agree with?

- Has Bean and Big Bean should be toggleable when playing exercise mode in enhanced mode
- Exercise mode attempts using Has Bean and Big Bean should use a separate leaderboard
- Has Bean and Big Bean should always be forced in exercise mode since they are part of the game mode, and should be toggleable when playing custom games
- Other

Any number of answers were pemitted.



8. When considering the Has Bean and Big Bean bonuses in exercise mode, which of the following statements do you agree with?

More Details

●	Has Bean and Big Bean should b...	2
●	Exercise mode attempts using H...	3
●	Has Bean and Big Bean should a...	1

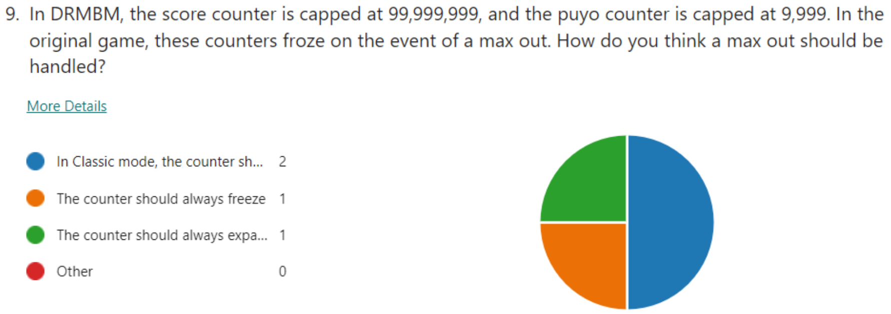
Figure 1.13: Results to the eighth survey question.

The majority of people would like leaderboards to be split between runs that use Has Bean/Big Bean and runs that do not. This is surprising to me, but not particularly difficult to implement thus shall be included. This overrides the one person's comment about always forcing them.

Question 9: In DRMBM, the score counter is capped at 99,999,999, and the puyo counter is capped at 9,999. In the original game, these counters froze on the event of a max out. How do you think a max out should be handled?

- In Classic mode, the counter should freeze, in Enhanced mode the counter should physically expand to accommodate more digits
- The counter should always freeze
- The counter should always expand
- Other

Only one answer was permitted.



9. In DRMBM, the score counter is capped at 99,999,999, and the puyo counter is capped at 9,999. In the original game, these counters froze on the event of a max out. How do you think a max out should be handled?

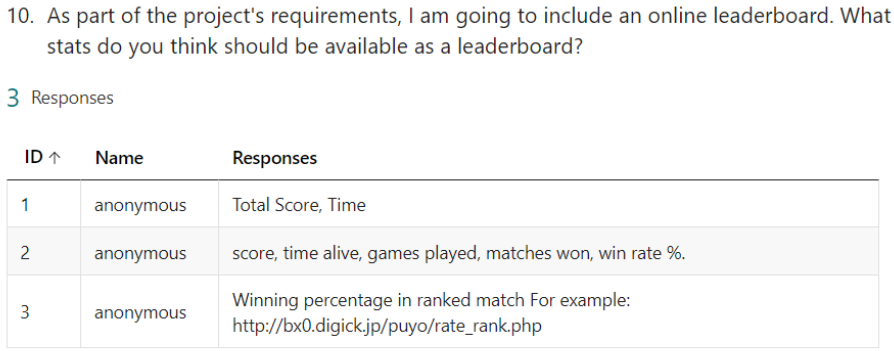
More Details

● In Classic mode, the counter sh...	2
● The counter should always freeze	1
● The counter should always expa...	1
● Other	0

Figure 1.14: Results to the ninth survey question.

A majority of people would like to implement the solution that I personally had in mind: freezing the counters in classic mode and allowing them to physically expand in enhanced mode, so this is what I shall implement. I understood that allowing the counters to freeze in classic mode would be important to keep because a popular speedrun of the game is trying to max out the score counter in the least possible time, and removing this bug would take away from one of the ways people enjoy the game.

Question 10: As part of the project's requirements, I am going to include an online leaderboard. What stats do you think should be available as a leaderboard? This question permitted a text answer.



10. As part of the project's requirements, I am going to include an online leaderboard. What stats do you think should be available as a leaderboard?

3 Responses

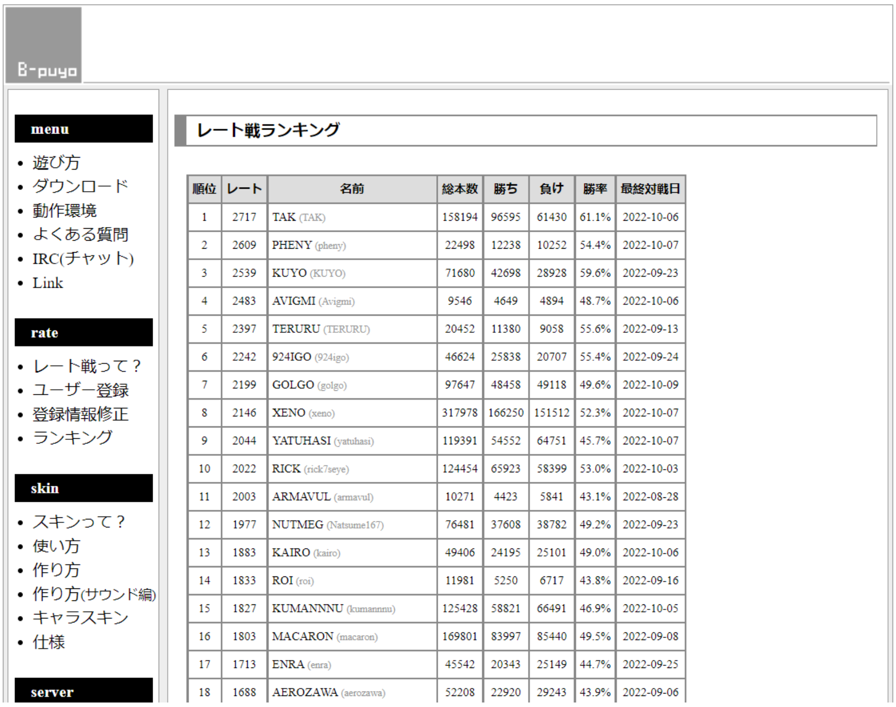
ID ↑	Name	Responses
1	anonymous	Total Score, Time
2	anonymous	score, time alive, games played, matches won, win rate %.
3	anonymous	Winning percentage in ranked match For example: http://bx0.digick.jp/puyo/rate_rank.php

Figure 1.15: Results to the tenth survey question.

These are all fairly generic examples, but I do appreciate the link provided to a Japanese ranked BPuyo leaderboard to use as an example.
Question 11: Enhanced mode will allow the game to support a 16:9 aspect ratio. What do you believe should be used to fill the space? This question permitted a text answer.
The solution to this problem remains to be determined, so I will probably fill it with empty space for now and see if I figure out something convenient later.

Question 12: Below are other features that I plan to implement into Enhanced mode. Rate their importance. The options contained within rows were:

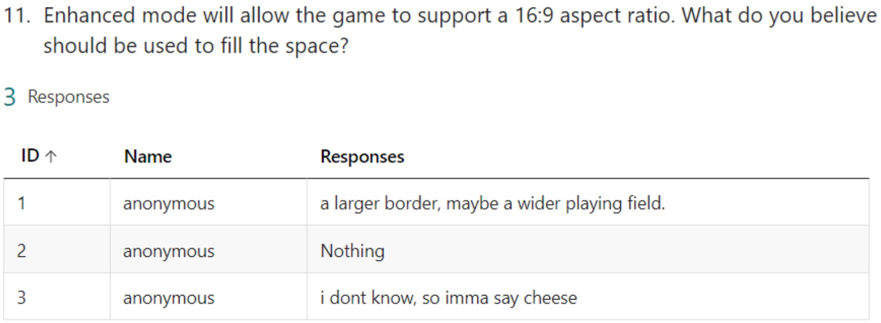
- Custom texture support
- Custom handling settings
- Custom resolutions (any aspect ratio)



レート戦ランキング

順位	レート	名前	総本数	勝ち	負け	勝率	最終対戦日
1	2717	TAK (TAK)	158194	96595	61430	61.1%	2022-10-06
2	2609	PHENY (pheny)	22498	12238	10252	54.4%	2022-10-07
3	2539	KUYO (KUYO)	71680	42698	28928	59.6%	2022-09-23
4	2483	AVIGMI (Avigmi)	9546	4649	4894	48.7%	2022-10-06
5	2397	TERURU (TERURU)	20452	11380	9058	55.6%	2022-09-13
6	2242	924IGO (924igo)	46624	25838	20707	55.4%	2022-09-24
7	2199	GOLGO (golgo)	97647	48458	49118	49.6%	2022-10-09
8	2146	XENO (xeno)	317978	166250	151512	52.3%	2022-10-07
9	2044	YATUHASI (yatuhasi)	119391	54552	64751	45.7%	2022-10-07
10	2022	RICK (rick7seye)	124454	65923	58399	53.0%	2022-10-03
11	2003	ARMAVUL (armavul)	10271	4423	5841	43.1%	2022-08-28
12	1977	NUTMEG (Natsume167)	76481	37608	38782	49.2%	2022-09-23
13	1883	KAIRO (kairo)	49406	24195	25101	49.0%	2022-10-06
14	1833	ROI (roi)	11981	5250	6717	43.8%	2022-09-16
15	1827	KUMANNNU (kumannnu)	125428	58821	66491	46.9%	2022-10-05
16	1803	MACARON (macaron)	169801	83997	85440	49.5%	2022-09-08
17	1713	ENRA (enra)	45542	20343	25149	44.7%	2022-09-25
18	1688	AEROZAWA (aerozawa)	52208	22920	29243	43.9%	2022-09-06

Figure 1.16: The ranked leaderboard from the Bpuyo website.



11. Enhanced mode will allow the game to support a 16:9 aspect ratio. What do you believe should be used to fill the space?

3 Responses

ID ↑	Name	Responses
1	anonymous	a larger border, maybe a wider playing field.
2	anonymous	Nothing
3	anonymous	i dont know, so imma say cheese

Figure 1.17: Results to the eleventh survey question.

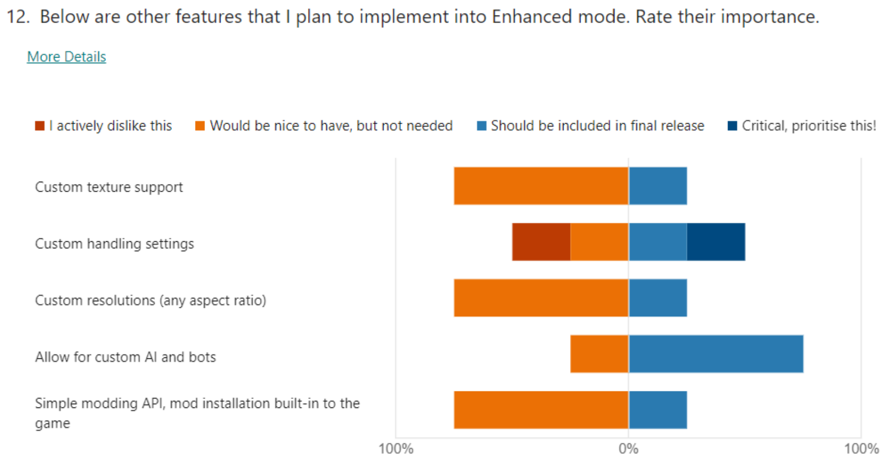
- Allow for custom AI and bots
- Simple modding API, mod installation built-in to the game

The options contained within columns were:

- I actively dislike this
- Would be nice to have, but not needed
- Should be included in final release
- Critical, prioritise this!

All of the items listed are planned to be included, it is simply a matter of prioritising what the end user considers important. For custom textures, 3 people said it would be nice to have and 1 said it should be in the final release. The inclusion of custom texture support itself is fairly trivial to implement due to the nature of having to import textures using the engine anyway, the time-consuming part would be writing documentation that explains how people can create their own texture packs that would be compatible with the game. I now know that this should not be prioritised.

Custom handling settings was the most devisive option, with each of the 4 applicants choosing separate options. I don't understand the rational behind actively disliking custom handling settings as the default will be the same as they are in the actual game, however it may be worth considering



12. Below are other features that I plan to implement into Enhanced mode. Rate their importance.

Figure 1.18: Results to the twelth survey question.

forcing certain handling settings in ranked matches or games that will be displayed on leaderboards; perhaps different leaderboards with enforced handling and custom handling? This will have to be considered.

Custom resolutions received the same reception as custom textures - it would be nice to have but isn't overly important. Different aspect ratios are actually especially challenging and non-trivial to implement. My original design for the engine involved scene data containing a background image variable, however this static image doesn't account for different resolutions and aspect ratios. Thus in order to account for future support for multiple aspect ratios, the engine must be coded to accept the background as a function that draws the background. Then when coding scenes, the specific scene can decide the solution that is most appropriate for drawing the background, whether that be a solid colour, stretching an image to fit a resolution, having multiple images to support multiple aspect ratios or some kind of tiling solution.

Custom AI and bots received a positive reception. I shall have to create an object that allows for the implementation of AI to create the CPUs in scenario mode, thus it shall be trivial to allow modders to run their own function within this class (with some kind of primitive virus protection by not allowing external modules to be accessed).

The described "modding API" will simply be an expanded version of what is described above - allowing users to modify their game by importing new objects written by other users that are compatible with the engine, at the user's own risk.

The survey was supposed to include a poll about replays, but unfortunately I forgot to include it. I can only assume it would be a desired feature.

1.4 Input, Data Processing, Output

The program is started with main.py. This script shall verify the integrity of local game assets using SHA hashing and querying a simple API on a web server, retrieving assets as necessary. Then, the script shall import Kris's Engine, an engine that shall be written and packaged by me with the game.

Kris's Engine is built upon the idea of two fundamental class templates Scene and Entity, which shall be further described within the Documented Design section of this report. The engine shall first initalise itself, loading textures and initialising modules such as aiohttp and pygame. The engine shall then import the scene that is predefined by main.py, which will probably be title.py to load the title screen. From then, the engine shall use two threads: one for rendering and one for updating entities. Each entity must have the methods render and update, where the engine on seperate threads will call update 600 times per second and render a variable amount of time, that is able to be changed by the user, that will default to 60. The update method for an entity is responsible for data processing and the render method is responsible for any visual output that may be needed.

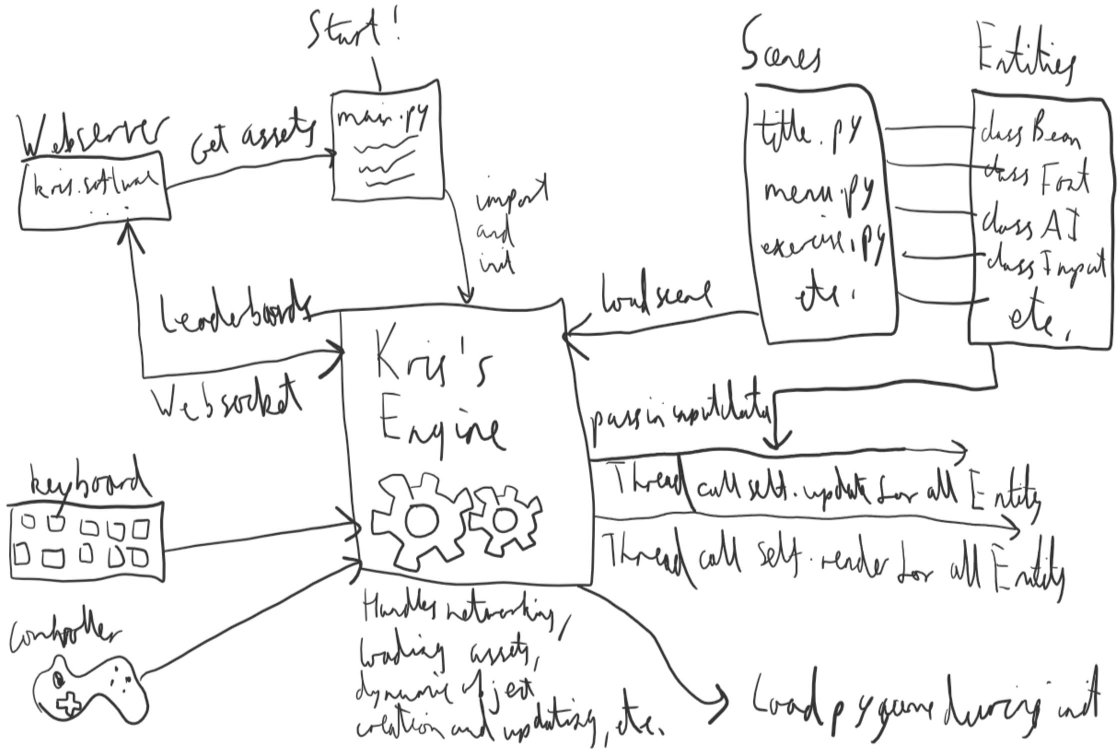


Figure 1.19: An Input, Data Processing, Output diagram.

1.4.1 Goals

Main objectives

- The project shall contain an engine module, which consists of an Engine class. This Engine class should:
  - Define the Scene and Entity classes, to be used as templates. These objects will be described further in depth in this document's Documented Design section.
    - The Scene class should allow an object that inherits from it to:
      - define what assets (images, sound files, etc) need to be present for a given scene.
      - define some functions required for rendering a scene such as a function for rendering the background of a scene that works for different resolutions
      - define what entities should be loaded in with a scene, their order and initialisation parameters
    - Any object that inherits from the Entity class should:
      - Define an update method which handles the data processing for a given entity.
      - Define a render method which handles the output for a given entity, be that pygame rendering functions, console print statements, logging to a file, etc.
  - Initialise pygame and take ownership of all pygame objects, such as the screen object being located at Engine.screen
  - Initialise two threads:
    - A thread shall be responsible for handling the update method of every entity that is owned by the engine. The aim is to run this 600 times per second.
    - A thread shall be responsible for handling the render method of every entity that is owned by the engine. The aim is to run this at least 60 times per second, however due to the constant rate of the update function, this could be ran at any speed without affecting any game logic. For this reason, counter-intuitively it is critical that things that must be done at a constant rate, such as animations, rely on the update function instead of the render function, and the render function should be used only for drawing.
  - Contain a method for loading scenes that entities can invoke

  - Contain methods that allow entites to easily make requests without causing the program to freeze
  - Contain methods for handling backend tasks such as changing resolution, setting the window to fullscreen, etc.
  - Contain methods for easily playing sounds on different channels. The engine is responsible for ensuring that a sound can always be played and that the number of pygame mixer channels is never exceeded.
  - The engine could contain generic methods that prevent repeating complex code, for example a method that creates a spray of particles. This is something that would be time consuming to implement into every entity that requires it and would be incredibly resource intensive if an entity was used for each particle, thus it makes sense to have it as a function that can be used by any program that uses the engine, with a replaceable texture.

- The game shall be split into two modes, Classic mode and Enhanced mode.
  - Classic mode shall attempt to be a faithful recreation of the original game. This includes:
    - A recreation of all 13 stages in "scenario mode", the game's equivalent of a story mode. Various algorithms are used for your computer opponents and an attempt shall be made to recreate these algorithms as faithfully as possible, though lack of documentations means that some compromises will have to be made.
    - "excercise mode" should function as in the original, with three speed difficulties, counters for score and difficulty, and the spawning of the Has Bean and Big Bean power-ups. Two simulatenous, separate games should be supported, allowing to players to play locally on the same device independantly.
    - "1P VS. 2P mode" should have the original 5 difficulty modes and two players should be able to use separate input devices to play two linked games in a competitive match locally.
    - An "options" menu. This will only contain the the settings found in the original settings menu, other settings shall be found in the menu found when the game starts up for selecting between Classic and Enhanced mode.
  - Classic mode shall be developed first, and Enhanced mode will be built upon Classic mode. The aim is to include the following changes:
    - A customisable resolution. Certain scenes such as the animated segments before levels in scenario mode will need to be locked to certain aspect ratios such as 4:3 or 16:9 in order to look correct, whereas other scenes can function at any aspect ratio. Due to pygame not allowing windows to be resizeable, there will have to be a setting to allow the user to change the resolution, and locked aspect ratios shall be achieved by black bars, which will be rendered by the background method of a given scene object.
    - The ability to store and play replays of games from any mode.
    - Save files for scenario mode.
    - The ability to customise handling, such as the amount of time before a bean starts to repeat movement when the left or right key is pressed down (this is a value known as "DAS")
    - An online leaderboard in excercise mode. This will function using a simple script running on an external, central web server. The leaderboard should use an SQL database to store user information, locations of replay files and information such as scores and times. The leaderboard factor will be implemented using a merge sort algorithm to sort scores into the correct order. The user should be able to easily play the replays of users on the leaderboard. As requested from end user input, there shall be separate leaderboards for attempts that make use of the now toggleable Has Bean and Big Bean powerups, as well as separate leaderboards for users who choose to use custom handling.
    - General bug fixes should be implemented such as, as requested during end user input, the score counter in excercise mode should expand to allow for scores greater than 100 million.

        - ∗ Implementation of the Tsu ruleset. The Tsu ruleset includes rules such as the cancellation of pending garbage beans and bonuses for things like perfect clears. When creating a save file the user should be asked which ruleset they want to use.
        - ∗ Allow the user accessibility to the games underlying classes so they can easily modify the game with things such as custom textures and importing their own AI opponents.

1.4.2 Extension objectives

If the project goes well, the aim is to include:

- – Real-time online multiplayer using web sockets.
- – Implementation of my own custom AI.

Chapter 2

Documented Design

2.1 Language and rendering module

The project shall be written in Python 3 because it is the language I have the most experience with. The third-party Pygame module shall be used for rendering graphics to the screen because of it's well-written documentation and ease-of-use.

2.2 Engine classes

When creating a game, it is important to create a generalised, versatile and reusable structure. Additionally, performance and consistent timing of backend tasks are important to this project. Thus, before approaching the game I first created a module that would be helpful in the game's development by containing code and classes that would be consistently reused. This module, which I will refer to as the engine module, contains three main classes: Engine, Entity and Scene. Entities are things individual things that need to be drawn on screen or processed, a Scene object defines reusable functions and information about how entities should be defined and an Engine object has a relationship of aggregation with Scenes and Entities, is capable of loading and destroyed them, and handles backend and threading tasks.

2.2.1 Entity

An Entity is a base class that is designed to be inherited from to quickly implement new types of objects on screen or that need to be processed. It has the following attributes and methods:

persist

The Entity class defines persist to be False by default, but it could be changed to True by someone creating an entity. If persist is set to False, then when the engine loads a new scene, the entity will be destroyed. If persist is true, it will remain throughout different scenes until explicitely destroyed.

__init__

This method has the parameters engine, scene and id. It expects the inputs to these parameters to be the engine object, the scene object that is creating the entity, and a unique integer ID. This function should be overwritten so that a developer may initialise their own entity, and the developer is expected to maintain these three positional arguments in this order, in addition to adding their own arguments and keyword arguments. A developer can easily execute this __init__ function in addition to their own by running ”super().__init__(engine, scene, id)”

init

This method should largely be ignored and is used initialise the attributes of the default Entity in the case that this class was to be initialised directly instead of being inherited from. Loading Entity directly creates a spinning square.

update

A developer is expected to override this function. It is executed at a set rate by the update thread, explained more in the Engine class section. It should be used for backend tasks, and tasks that need to happen at a constant rate such as animations.

render

A developer is expected to override this function. It is executed every time a frame is rendered and therefore should not be expected to run at a constant rate. This function should not complete any backend tasks and should instead draw on screen with pygame functions a representation of what is happening in the backend.

2.2.2 Scene

A Scene is a base class that is designed to be inherited from to easily implement the loading of many entities. An example of this would be transitioning from a menu to gameplay by loading the gameplay scene, destroying the entities from the menu scene It has the following attributes and methods:

load_with_pbar

This attribute should be an iterable such as a list or tuple. If filled with filenames, then before the loading of this scene, a built-in progress bar scene will be used to load the files listed into the engine's asset cache.

update_rate

This attribute should be a positive non-zero integer, and represents the number of updates per second the engine will attempt to execute for the duration of the scene. Since tasks are designed to expect a constant rate from this number, halving this number would create the effect of running a scene's backend at half speed, and drastically reducing it would make the scene appear to be running in slow motion.

render_rate

This attribute should be a positive integer, and represents the number of frames per second the engine will attempt to render for the duration of the scene. Reducing this number would make the scene's backend still run at the same speed, but the output would appear choppy. Setting this number to 0 will cause the engine to try and render as many frames as possible without waiting.

background

background is both an attribute and a method. It is expected that background be a function object, but by default this is implemented as a lambda function assigned to a variable. background will be called every frame before entities are rendered and by default will fill the screen with black, clearing it.

music

Another combination between an attribute and a method, music is a function that should return a pygame Sound object. This sound will then loop until another scene is loaded.

kap

This attribute should be an iterable, such as a list or tuple, of strings. When loading the scene, the engine will ensure that all the KAP files listed are loaded to ensure that textures are accessible. KAP files will be explained later in this section.

__init__

This method is expected to be overrided. Its only positional argument is engine, expecting the engine object that is loading the scene, which is then assigned to self.engine. A developer can either implement this by running "super().__init__(engine)", or just running "self.engine = engine".

2.2.3 Engine

The Engine contains many methods and serves many purposes. The main purpose of the engine is to manage a multi-threaded structure. Updates and rendering are handled independantly on different threads at different rates. This is done to allow them to run at different rates, as well as to allow for an uncapped frame rate and improved performance in such a performance-critical genre such as stacker games.

__init__

This method has many keyword arguments.

- scene represents the scene that the engine should load. If none is given, the default Scene class is initialised.
- width and height represent the pixel size of the window
- engine_path and log_path are used to specify the location of files relative to the programs current point of execution
- texture_quality controls the resolution at which image textures are loaded, using this feature of KAP files
- log_max_size specifies a maximum size for the log file before it begins to be trimmed from the top. This defaults to 10MB.

All of these are stored in private variables that can be access but not written to, with the exception of texture_quality, which has a setter than flushes the engine's asset cache when the texture_quality is changed in order to load the textures of a new quality.
This function initialises id, a getter that increments the id attribute for creating unique entity IDs.
The logging thread is then initialised.
The now depreciated config file is then loaded.
Some timers are set and attributes are defined, then pygame is initialised.
Default assets, to be used if an asset it missing, are loaded.
More attributes are defined and the pygame window is created.
The update and render threads will be started, and the specified scene will be loaded.
Then, the engine will collect pygame events, using event_gotten to synchronise with frames on the render thread, until the program exits when pygame will be quit and the main thread will terminate.

init_log

Printing in python is slow. The logging thread periodically flushes log messages to the console, flushing multiple messages at a constant interval to save on processing time. These messages are in a constant format detailing the time since execution of the program began, and colour coding messages with ANSI escape codes, the colour being dependant on the thread the message was made with. Additionally a separate log is kept without ANSI escape codes to be stored in a text file.
This method creates the necessary variables for this system to function, and starts the thread that dumps the log to the console. It has no parameters.

loop_log

The method that is passed to the thread for execution until the program is terminated. This method outputs the log to the console and then waits for a period of time specified in init_log. Sleeping in a thread only pauses that thread, so this allows other execution to continue. It has no parameters.

exit_log

This method uses Python's "atexit" module to write the log to a text file when the program is terminated. It has no parameters.

append_log

This method takes a message (a string) as a parameter and formats a log message by getting information about the current thread and current time.

load_config

This method used to be used to load the information that is now specified by keyword variables. It is no longer used, but can be used by a developer if they wish to load additional settings. It looks for a config.json file and if it exists, assigns the contents of the file to a config dictionary attribute. It has no parameters.

save_config

This method saves the content of the config parameter to a JSON file. It has no parameters.

get_events

This method is used to fetch pygame events and put them into a list, the events attribute. It also checks for a pygame.QUIT event, which happens when the close button is pressed on the window. This sets the running attribute to False. All threads should use ”while self.running” for any infinite loops, meaning that they will finish executing that iteration and then terminate when self.running is set to False. This function has no parameters.

get_asset

This method is accepts a filename as a required parameter. It will either load the given file into a cache (at the assets attribute, or the font_assets attribute), or load it from the cache if it is already there. It will return a pygame object version of the specified file. So, if audio is set to True, then a pygame.mixer.Sound object is returned. If font is set to an integer then a version of that font rendered at the given font size is returned as a pygame.font.Font object. Otherwise, a pygame surface object is returned. This function can also scale image textures using the __scale method and store the scaled copies, to prevent scaling being done frequently. Additionally, if a given filename is not found in an loaded KAP files, this function will return an alternative of the same filetype from ”default.kap” in the engine folder.

__scale

Accepts two positional arguments, scale and path. scale can be one of three types: a string ”raw”, a tuple containing a width and height for the image to be scaled to, or a float. If a float is provided, then the image will maintain it’s original asepct ratio, and be scaled such that the height is equal to the float specified multiplied by the window height of the engine.

load_entity

This method accepts an entity and a scene as positional arguments, as well as allowing for additonal arguments and keyword arguments to be passed into the entity upon it’s initialisation. scene should be an initialised scene object while entity should be a class that hasn’t been initialised yet. First this method will verify that the class given has inherited from the Entity class, then it will generate a unique ID, initialise the entity and store it, creating a relationship of aggregation between the Grid and the entity. It will then return the entity object.

destroy_entity

Takes a single positional argument, being an initialised entity object. First the method verifies that the object passed in inherited from the Entity class, then it removes it from the entity list and dictionary of entity IDs. This is the only way to destroy an entity with persist set to True. This method will fail if the passed in entity is no longer owned by the engine.

load_scene

This takes a positional argument, being a not-yet-initialised class that inherited from Scene. pbar is a keyword argument that needs to be set to False by a progress bar when trying to load a scene that has been loaded with a progress bar to prevent an infinite loop of calling the progress bar, otherwise it should be left alone. All other arguments and keyword arguments are passed into the scene upon initialisation. First, this method sets the ready attribute to False, which pauses the update and render threads. Then, it verifies that the class being passed in inherited from the Scene class. Then, it will load all the KAP files given by the kap attribute of the scene. Then, it will destroy all non-persistant entities from the last scene. Then, if the pbar keyword is set to True and the scene has files in the load_with_pbar attribute, then the progress bar class is loaded for loading the files. Otherwise, other attributes from the scene are processed, the scene object is initialised, timers are reset and the update and render threads are unpaused.

__size

A method used for calculating the amount to increment the progress bar by for a given file, by using its compressed size. It takes a single argument, being the file name.

update_loop

This is a method that takes no arguments and is executed by the update thread. First, it assigns itself a colour, and then loops while the running attribute is True. Then, it does nothing if ready is set to False, to pause execution while a scene is being loaded. If ready is True, then the update_counter attribute is incremented. For every entity in the ordered entity list, the update method is called. Then, the events attribute, a list of pygame events, is trimmed to remove the events that have already been process. Finally, the method calculates how long it needs to wait for to maintain a constant update rate, and outputs a lag message every two seconds if it detects that it is behind.

render_loop

This is a method that takes no arguments and is executed by the render thread. First, it assigns itself a colour, and then loops while the running attribute is True. Then, it does nothing if ready is set to False, to pause execution while a scene is being loaded. If ready is True, then the render_counter attribute is incremented. The background method for the currently loaded scene is called to clear the scene. Then, for every entity in the ordered entity list, the render method is called. Then, the thread waits for the main thread to collect pygame events (this must be done once per frame but also must be done by the main thread). Once this is done, the pygame display object is updated, drawing the frame. Finally, if the render rate is not uncapped, the method calculates how long it needs to wait for to maintain a constant frame rate, and outputs a lag message every two seconds if it detects that it is behind.

2.2.4 Class diagram

2.3 Other engine scripts

You will have seen referenced above many times the concept of KAP files. This is because of another module: kris_engine.files. This implements a custom file type KAP which packages and compresses textures.

2.3.1 files.KAP

A wrapper for KAP files. KAP stands for Kris's Asset Package and a KAP file is designed to store many textures in a single file. It will also store different resolution copies of images for different texture qualities and compress data with a variant of run length encoding.
A KAP file is structured in two parts: a string portion and a byte portion. Some rules about the string portion:

- All integers are unsigned and most significant bit first
- All strings are UTF-8

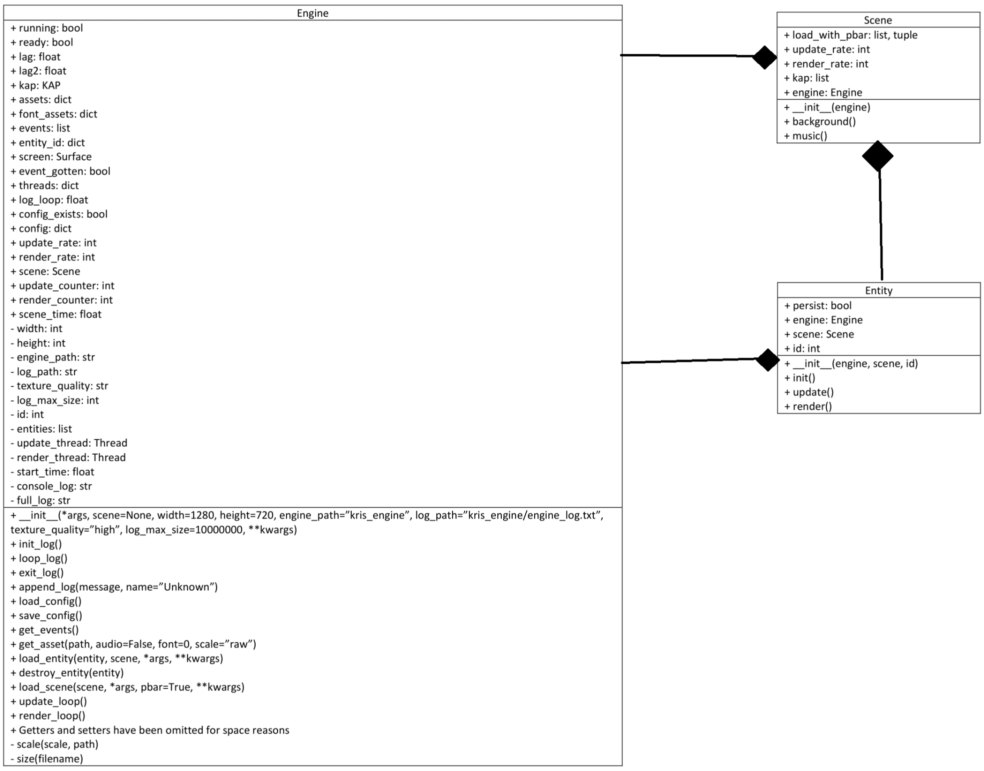


Figure 2.1: A class diagram of the main three engine classes

- A 0 byte is eight bits with a value 0

The file begins with a 16-bit integer magic byte, 1993. This helps verify that we're loading the right file type. Then, an 8-bit integer representing the number of texture quality options available. Then, for as many qualities as the integer specified:

- string data detailing the name of the setting
- a 0 byte
- a non-zero 8-bit integer that the texture quality should be assigned to

Then, there's a 32-bit integer detailing how many textures there are. Each entry of the rest of the string portion is structured as follows:

- string data detailing the name of the file
- 0 byte
- 8-bit integer representing how many texture qualities the file has
- if this byte is 0:
  - 64-bit integer, indicating the pointer at which the file data starts in bytes
  - 64-bit integer, indicating the size of the file in bytes
- if this byte is greater than 0, repeat the following for the value of the byte:
  - 8-bit integer corresponding to a texture quality
  - 64-bit integer, indicating the pointer at which the file data starts in bytes
  - 64-bit integer, indicating the size of the file in bytes

Then the corresponding files will be dumped as RLE bytes into the files at their corresponding pointers.
RLE bytes are bytes that are run length encoded. RLE bytes start with a byte that is comprised of two 4-bit integers. The first represents counter_length, the second pattern_length. Pattern length represents the size of chunks of data that we are processing (allowing us to compress something like 101010101010 efficiently with pattern length 2), and counter length represents the numnber of bytes allocated to storing the multiplier for a given piece of data. These are required information to encode and decode RLE bytes. Then we have to consider the excess. Take an example where we have a file with an odd number of bytes and pattern_length 2. For the algorithm to look for patterns of length 2, we need to trim the first byte off. So, the second byte is an 8-bit integer representing the size of the excess. If this is 0, move straight on to the data, otherwise append the following number of bytes to the front of the output. Then, the rest of the bytes are in the pattern of counter_length bytes, pattern_length bytes. To decode, repeat the pattern_length bytes the value of the counter_length bytes times.
The KAP class is used as a wrapper for loading KAP files. It has the following methods:

__init__

This method has a positional parameter path, representing the path to the KAP file that is to be loaded. It also has an optional engine paramater, like many of the functions in the KAP class, which is used for outputting to the engine log if the engine object is passed in.
This function creates a file map, which is a dictionary that tells us which KAP file an asset comes from (the KAP class can load multiple KAP files at once). The assets dictionary is also created, containing information about the location and size of each file in a KAP file. An open dictionary is created, storing the file objects for the open KAP files, and a function is registered that closes them when the program terminates. Then the named KAP file is loaded with load_kap.

__del__

Used to close all open files objects when the KAP instance is deleted or the program terminates. Takes no arguments.

load_kap

Takes one argument, being the name of the KAP file to be loaded. Uses advanced file operation to load the KAP file, the structure of which is described above.

string_data

A method used by load_kap for finding where strings of unknown length terminate.

load

A method used for loading a file from a KAP file. It has a filename parameter and an optional quality parameter.

2.3.2 Other files.py functions

rle_encode

Implements the varianets of run length encoding specified above. See the comments in the technical solution for more details.

rle_decode

Decodes RLE bytes.

rle_brute

Used to test multiple pattern length and counter lengths, returns the compressed file that is the smallest (and therefore most effective).

build

Uses advanced file operations, as well as Pillow for image scaling for different image qualities, to build a KAP file.

2.3.3 pbar.py

This contains a scene and entity that creates a progress bar loading screen. See comments in technical solution for more details.

2.4 Classes for gameplay

For the implementation of the classic version of exercise mode, the code was split into two script files: exercise.py, containing the ExerciseClassic scene, and gameplay.py, containing everything else.

2.4.1 Grid

Grid is an entity that manages most of the gameplay. It has a relationship of aggregation with Bean, (which is not an entity).

__init__

In addition to the required entity parameters, grid has one positional argument, input_handler, which is another entity that acts like a controller. It also has many keyword arguments: rows and columns representing the size and shape of the grid, values allowing the grid to start in a non-empty state, the position of the grid and the bean queue. This method creates most of the engine attributes, sets the grid's current state to GRAVITY and loads the BeanQueue entity and Score entity, which is loaded by the grid because it needs to directly access methods and attributes of these objects. Another thing worth nothing here is that values, containing Bean objects or None (representing an empty cell), is a 1D list, not a 2D list. This is because of the fact that beans can be placed "above" the grid (above the inputed number of rows, which simply decides the square that kills you and where falling beans spawn). Increasing the size of the list is easier with a 1D list than with a 2D list.

__str__

Outputs the contents of the grid as a neat string. Used for debugging.

update

At any time, the grid will be in a given state. This method acts as a lookup table, executing a different update function depending on the grid's current state.

update_verify

This method is called when updating in the VERIFY state. First we check the contents of the verify attribute, a set containing any beans that need to be removed (due to being in a colour group) this update. If there are beans that need to be removed, we first calculate the score for removing those beans, update the score counter and replace the beans with None in the grid. Our state then changes to VERIFY_ANIMATION.
Alternatively, if there are no beans that need to be removed, we check the third cell of the top row (where beans spawn), and if it's filled then we change our state to DIE. (Death is not actually implemented, but will simply destroy the grid entity and display a message in the console). If we didn't die, then we get the next pair of falling beans from the bean queue, and calculate our score, which I will discuss now.

chain_power_lookup, colour_bonus_lookup and group_bonus_lookup

These are static methods that act as lookup tables for different bonuses, but it gives me an opportunity to talk about how scoring is calculated.

You will see during gameplay two numbers multipled together. The first number is equal to 10 multiplied by the maximum number number of beans popped at one time during the chain. During any point in the chain, count the number of beans that are currently being popped, and if that number is greater than the beans_popped attribute, that becomes the new value of the beans_popped attribute.
The second number is equal to (chain power bonus + colour bonus + group bonus). Chain power bonus is the value gotten from chain_power_lookup, inputting the chain power when the chain concludes. Colour bonus and group bonus are different, incrementing the bonus at each stage of the chain. At a given point during the chain, the group bonus is incremented for any group that contains more than 4 puyos. At a given point during the chain, the colour bonus is incremented if there are multiple different colours of puyo being popped at the same time. Additionally, in our score calculation, (chain power bonus + colour bonus + group bonus) must be at least 1, even if the output is 0.

animate_verify

This method is called for updates during the VERIFY_ANIMATION state. It increments a counter that causes actions to occur at different time periods, such as changing textures or playing a sound effect.

animate_gravity

This method is called for updates during the GRAVITY_ANIMATION state. The first part of this happens while there are gravity beans in the gravity attribute list. Each of these has a generator object that can be iterated through to animate the bean. When StopIteration is raised, the animation is finished, and the bean is placed in the correct position and removed from the gravity list. Additionally we also check for colour groups where this bean has fallen and correct it's texture.
When all the beans have been removed from the gravity list, then we start counting to 50 before changing to the VERIFY state. This causes a short pause beforce beginning verification again.

update_gravity_quick

This is an unused method that uses a quicker way of calculating where beans fall by simply removing empty cells in columns. This would be useful for replay files, but cannot be used for gameplay as since we don't know which beans have fallen we cannot animate them falling.

update_gravity

This method is called for updates during the GRAVITY state.
For every column in the grid, we get a list containing just the contents of that column. If that column is full, we ignore it, otherwise we use the split_list static method. This method acts exactly like the split method for a string, but for lists. It returns a 2D array, split by a separator, acting on a 1D array. This split list is very useful because the index of the list that each bean is in represents the number of rows that bean will fall. We use this information to replace these beans with gravity beans that are set to fall this distance, and replace the column with empty cells before changing state to GRAVITY_ANIMATION.

place_bean

Useful method for placing beans. Pads the grid if the desired position doesn't yet exist, places the bean in the grid, then checks for colour groups and updates the newly placed bean's texture.

render

At any time, the grid will be in a given state. This method first draws the beans in the grid. Then, this method acts as a lookup table, executing a different render function depending on the grid's current state. Then, we draw background3.png over the top, creating the effect of beans being behind this background.

render_grid

Converts 1D list into 2D positions and draws the beans.

render_bean

Accepts a bean, a row and a column as arguments. Uses pygame to draw the bean to the screen.

render_gravity

Method executed for render calls in the GRAVITY_ANIMATION state. Runs the render_bean function for every gravity bean.

render_verify

Method executed for render calls in the VERIFY_ANIMATION state. Runs render_bean for beans being verified, but additionally uses the counter to make them flash and be animated.

render_falling

Method executed for render calls in the FALL state. Draws the falling bean pair by first calling render_bean on the primary bean and then using a lookup table to find the relative position of the secondary bean dependant on rotation state.

eval_all_textures

Method used when passing in a pre-filled grid on initialisation. Evaluates the texture for every bean in the grid.

eval_texture

For a given bean, checks the beans adjacent to it and updates the bean's attributes to match. Updating these attributes also updates the bean's texture.

eval_up, eval_down, eval_left, eval_right

Very similar methods used for checking the particular side of a bean, to see if a bean of the same colour, or any bean at all is present there.

eval_surrounding

Evaluates the texture for a bean, then evalutates the texture of every bean adjacent to that beans. If you have just placed a bean, run this function to update grid textures efficiently.

count_all

A version of count that checks for groups in the entire grid. Used when loading a non-empty grid.

count

Method for checking for groups in one part of a grid once you're placed a bean. First, check if the bean is already in the list of beans to be removed. If it isn't, then create a set containing that bean. Sets must contain unique items and will remove duplicates. Then, we use a function to get all the beans adjacent to that bean that are the same colour, and add them to a list. Then, for every bean in that list, we get the beans that are adjacent to those beans and the same colour. We keep doing this until we run out of unique beans in our list to check. Then, if the group is larger than 4 beans, we add it to the necessary attributes.

get_surroundings

A method for getting beans adjacent to a bean that are the same colour as that bean and also not already in the group.

2.4.2 BeanQueue

A BeanQueue is an entity that is loaded by and belongs to a Grid.

__init__

The only additional parameter a bean queue takes on initialisation is position, which decides where it is rendered on the screen. It has a next attribute, which is a tuple of two beans of random colours that are generated using randint from python's random module.

render

A copy of the render_bean method from the grid, for drawing the next upcoming pair of falling beans.

update

Since I ran out of time to animate it, this entity does not have an update method.

get_next

Returns the bean pair that was being displayed in the queue and randomly generates a new bean pair.

2.4.3 Bean

A Bean is not an entity. It is created and used by Grid.

__init__

A bean takes a colour as an argument on initialisation. This can either be the name of a colour, or the associated ID of that colour. It additionally has keyword arguments that allow you to get its texture.

__str__, __repr__

Debugging method that prints the bean's colour.

get_texture

The boolean True or False of whether a bean of the same colour is adjacent to this bean on a given side can be combined into a 4-bit integer, which can be assigned to the correct texture to show in those states. Other textures require the state attribute to be set, which overrides this 4-bit integer on what texture to select.

Getters and setters

Changing a variable that would effect the texture of a bean causes it to re-evaluate it's own texture.

2.4.4 GravityBean

GravityBean is not an entity, it inherits from Bean. It is used to animate a bean falling into a new position.

__init__

This method has man positional arguments. bean represents the Bean object that the GravityBean is based on. destination represents the index in the grid that the bean will be placed at when it's animation is finished. row_distance represents the number of rows that the bean will fall downwards. row and column represent the starting position of the bean. Once these attributes have been assigned, a generator object is used to animate the falling bean. This is stored in the position attribute and should be advanced once per update.

2.4.5 FallingBean

FallingBean is not an entity and it does not inherit from anything. It contains Beans and it represents the pair of beans that the player has control over.

__init__

Takes the grid object as an input parameter, initialises attributes.

update

Has a counter that is incremented for timing. The texture of the primary bean is changed relative to the counter. Then we check if a rotation key is held down then we rotate. The rotation function makes it so that if a wall is in the way the bean is pushed away from the wall. Then we check if a move left or right action should occur and verify whether it is possible or if there is something in the way. If there is something in the way then moving left or right fails. We move the bean downwards relative to the counter and fall rate, fall rate is decreased if the down arrow is held. If we hit the bottom of the grid of there is a bean below when we are trying to fall, place the beans there. Otherwise, move the beans downwards. If the down arrow is held then we award points.

Chapter 3

Technical Solution

3.1 Features

List operations/complex algorithms/recursive algorithms: merge sort, gameplay.py, line 909 Complex algorithm: variation of run length encoding, kris_engine/files.py, line 152 Complex algorithm: efficient sorting of beans into colour groups, gameplay.py, line 432 Mathematical formula: score calculation, gameplay.py, line 154 Complex file operations: kris_engine/files.py, line 281 Complex OOP: Can be seen throughout the entire project. See the documented design. Multiple classes inherit from Entity and Scene, many classes are dynamically created and composed of other classes, see Grid creating Bean objects, polymorphism is used whenever the update and render methods of an Entity are overriden... The list goes on.

3.2 kris_engine/__init__.py

```python
# pygame is a third-party module for rendering graphics and playing sound
import pygame
# Here, json is used to create a config file where settings can be stored
import json
# os is used for file system operations such as checking if a file exists
import os
# My own module for outputting coloured text in the console using ANSI escape codes
# Much of the code was written by another student so will not be included in the Technical Solution
from kris_engine.colour import Colour
# Used for global except hook and dumping to stdout
import sys
# Used for managing threading
import threading
# Used for running function on program termination
import atexit
# Used for getting information about the current time and how long the program has been running for as well as sleeping
import time
# Used to assign random text colour to an unidentified thread
import random
# A module of my own creation used to load textures from a custom file type
# See files.py for documentation around this
import kris_engine.files

# Defining the main engine class
# When an Engine object is initialised, it will loop forever until something causes self.running to become False
class Engine:
    # These keyword parameters were formerly stored in a config.json file but this is more convenient
    # Scene refers to the scene loaded upon initialisation
    # width and height represent the size in pixels of the window created
    # engine_path represents where the folder that the engine is stored in is relative to where the program is being run. So here, the program from the directory where the engine is stored, and it must be specified if it is not.
    # log_path is the same as engine path but specifically refers to where the log file is stored
    # texture_quality refers to the resolution of the texture that is loaded, as loading maximum quality texture can often require gigabytes of RAM. Only applies to images currently, uses the texture quality feature of KAP files from the files module.
    # log_max_size is the maximum size that the log file can be in bytes, defaulting to 10MB
    # *args and **kwargs are the arguments and keyword arguments for initialising the scene passed into the scene keyword
    def __init__(self, *args,
        scene=None,
        width = 1280,
        height = 720,
        engine_path = "kris_engine",
        log_path = "kris_engine/engine_log.txt",
        texture_quality = "high",
        log_max_size = 10000000,
        **kwargs):

        # If a scene is passed in, load that scene. Else, load the default scene defined by the Scene class.
        scene = scene or Scene

        # Imports the program bar module. This is a built-in Scene and Entity used for loading screens.
        # It must be imported separate from other modules, upon initilisation of an Engine object, because the module imports classes from this module, and importing it while this module has not yet finished importing creates an unresolvable dependancy loop.
        import kris_engine.pbar

        # DO NOT USE WHILE TRUE
        # Only self.running, so threads terminate
        # Threads should be written in such a way that settings self.running to False stops whatever the thread is doing as soon as possible, and this should be taken into account during iteration, for example when loading something with the progress bar.
        # self.ready is used for starting and stopping the updating and rendering process, so that it can be paused during the loading of a scene
        self.running = True
        self.ready = False

        # Assign the keyword arguments to private attibutes so they cannot be altered by other objects during execution
        self.__width = width
        self.__height = height
        self.__engine_path = engine_path
        self.__log_path = log_path
        self.__texture_quality = texture_quality
        self.__log_max_size = log_max_size

        # Counter for generating unique integers, see the id property
        self.__id = 0

```

```python
        self.init_log() # Initialises logging thread
        time.sleep(0.01) # Just to make sure the thread has started

        self.append_log("Loading config file...")
        self.load_config() # Config file has been largely replaced by keyword arguments but remains in case a developer wants to use it for their own settings

        # Attibutes used to store the last time a message saying that the update or render threads were behind was sent so that the console doesn't get spammed
        self.lag, self.lag2 = time.perf_counter(), time.perf_counter()

        # self.event_gotten is an attibute used to synchronise the fetching of pygame events on the main thread and the rendering of frames on the render thread.
        # This is because events must be gotten once per frame.
        self.event_gotten = True

        self.append_log("Initialising pygame...")
        pygame.init()

        self.append_log("Loading default assets...")
        # I made missing.png but it's based on an industry standard
        # I made missing.ogg in Audacity it's 1 second of 0.8dB 1000Hz sine wave
        # temp_shop.ogg from https://undertale.fandom.com/wiki/Tem_Shop_(Soundtrack)
        # ComicMono.ttf from https://dtinth.github.io/comic-mono-font/

        # Loads the KAP file containing default assets. These are used if an asset is not found.
        # KAP files contain assets. To learn more about them, look at files.py
        self.kap = kris_engine.files.KAP(self.engine_path+"/default.kap", engine=self)
        # Fonts are stored in the font assets dictionary due to the need to store different font sizes, and all other assets (audio, textures) are stored in the assets dictionary
        self.assets, self.font_assets = {}, {}

        # Used for storing pygame events
        self.events = []

        # Ordered list of all loaded entities, order decides rendering and updating order
        self.__entities = []
        # The key is a unique integer ID and the value is an entity object
        self.entity_id = {}

        # Create window
        self.append_log(f"Creating window, (width, height) = ({self.width}, {self.height})")
        self.screen = pygame.display.set_mode((self.width,self.height))
        # Set window title
        pygame.display.set_caption("Kris's Engine")

        self.append_log("Spawning threads...")
        self.__update_thread = threading.Thread(target=self.update_loop)
        self.__update_thread.start()
        self.__render_thread = threading.Thread(target=self.render_loop)
        self.__render_thread.start()
        self.append_log("Loading scene...")
        self.load_scene(scene, *args, **kwargs)

        # Pygame is weird and getting events must be done on the main thread. The event_gotten attribute is set to False every frame and True every time events are fetched. Pygame will crash if events are not gotten every frame so this attribute prevents a frame from proceeding before events are collected, however this should be hardly needed as there is a sleep statement to sync it with the framerate. The subtraction of 1ms is to account for the time it take to run the get events function.
        while self.running:
            self.get_events()
            self.event_gotten = True
            if self.render_rate != 0:
                time.sleep((1/self.render_rate)-0.001)
        pygame.quit()

    def init_log(self):
        self.__start_time = time.perf_counter() # Used for calculating how long the program has been running
        # Use the built in thread ID system to assign colours to threads
        self.threads = {threading.get_ident(): {"colour": Colour(0xff00ff), "name": "Main"}}
        # The coloured thing we flush to the console
        self.__console_log = Colour(0xff00ff)+"Welcome to Kris's Engine!\n"
        # The thing we dump to a text file
        self.__full_log = "Welcome to Kris's Engine!\n"
        self.log_loop = 0.5 # Number of seconds between dumping to console
        atexit.register(self.exit_log) # Log will be saved on termination
        threading.Thread(target=self.loop_log).start() # Create logging thread

    def loop_log(self):
        # Assign a colour to the logging thread
        self.threads[threading.get_ident()] = {"colour": Colour(0x10ebe1), "name": "Logging"}
        self.append_log(f"Logging thread initialised at {time.time()}")
        while self.running:
            sys.stdout.write(self.__console_log) # Flush to console, faster than print!
            self.__console_log = "" # Clear console log
            time.sleep(self.log_loop) # Wait. Note this only pauses this logging thread, not other threads

    def exit_log(self): # Dump log to text file on program termination
        try: # Try and read from the file and trim the front from it to comply with the maximum log size
            with open(self.log_path, "r") as f:
                log = f.read()
            log += self.__full_log + "\n"
            with open(self.log_path, "w") as f:
                f.write(log[-self.__log_max_size:])
        except FileNotFoundError: # If the log doesn't exist just write what we have
            with open(self.__log_path, "w") as f:
                f.write(self.__full_log)

    def append_log(self, message, name="Unknown"):
        # Get information about the thread that the logging request is being made from
        # Pick a colour if we don't know about it
        # Name is only used the first time a thread is started, to give it a name, otherwise it is ignored
        t = threading.get_ident()
        if t not in self.threads:
            self.threads[t] = {"colour": Colour(random.randint(0, (2**24)-1)), "name": name}
        thread = self.threads[t]
        # Only call time function once so time is consistent across logs
        name_and_time = f"{thread['name']} thread at {time.perf_counter()-self.__start_time}"
        self.__console_log += f"{thread['colour']}[{name_and_time}] {Colour(0xffffff)}{message}\n"
        self.__full_log += f"[{name_and_time}] {message}\n"

    # reads from config file, converts json to dictionary, sets config attribute to data and returns data
    def load_config(self):
        self.config_exists = os.path.exists("config.json")
        if self.config_exists:
            with open("config.json", "r") as f:
                self.config = json.loads(f.read())
            self.append_log("config.json loaded")
        else:
            self.append_log("No config.json found")

        # The config file has been replaced with keyword arguments
        # This function still exists if a developer wants to use it

    # overwrites file with the config attribute converted to json format
    def save_config(self):
        with open("config.json", "w") as f:
            f.write(json.dumps(self.config))
        self.append_log("The config file was overwritten.")

    def get_events(self):
        # Simple alias for getting pygame events
        for x in pygame.event.get():
            if x.type == pygame.QUIT:
                self.running = False
            self.events.append(x)

    # All assets should always be gotten through get_asset so they can be cached
    # This prevents a rookie developer from making the mistake of loading a file every frame
    # It also caches scaled copies of images to prevent the lag causes by scaling an image every frame
    # And if an asset is not found then it replaces that asset with a default that will be noticeable and indicitive to a user of a problem.
    # path represents the name of the texture, note that this does not refer to a real file but the name of a file in a loaded KAP file
    # If audio is set to true, then an audio file is loaded. If a font size is specifies, a font of that size is loaded. Otherwise, an image is loaded
    # More details on the contents of scale are given at the __scale function
    def get_asset(self, path, audio=False, font=0, scale="raw"):
        # If we are loading an audio file
        if audio:
            try: # If it's in the cache, return it, otherwise
                return self.assets[path]
            except:
                try: # Load it as a pygame Sound object and assign it to the cache then return it
```

```
                    self.assets[path] = pygame.mixer.Sound(self.kap.load(path,
                    ↪   engine=self))
                    return self.assets[path]
                except: # If it failed to load, output an error message and call
                ↪   this function again but getting missing.ogg as a replacement
                    # Note that if default.kap is missing this will recursively
                    ↪   error as the engine is not installed properly
                    self.append_log(f"Error! Asset {path} was not found! Is your
                    ↪   required KAP file loaded?")
                    return self.get_asset("missing.ogg", audio=True)

        # If we are loading a font of a given size font
        if font:
            try: # See if that font size is in the cache and return it
                return self.font_assets[path][font]
            except:
                try:
                    try: # See if that font is loaded into the cache at all
                        self.font_assets[path]
                    except:
                        self.font_assets[path] = {} # If it isn't loaded into the
                        ↪   cache, we store a dictionary at that path name
                        # This dictionary has font sizes as keys
                    # We then load in the font at the desired size, store it in
                    ↪   cache and return it
                    self.font_assets[path][font] =
                    ↪   pygame.font.Font(self.kap.load(path, engine=self), font)
                    return self.font_assets[path][font]
                except: # If it failed to load, output an error message and call
                ↪   this function again but getting ComicMono.ttf at the same
                ↪   font size as a replacement
                    # Note that if default.kap is missing this will recursively
                    ↪   error as the engine is not installed properly
                    self.append_log(f"Error! Asset {path} was not found! Is your
                    ↪   required KAP file loaded?")
                    return self.get_asset("ComicMono.ttf", font=font)

        # If it's not a font or an audio file then it's an image (that's
        ↪   everything that is supported)
        try: # See if it's in cache or not
            self.assets[path]
        except: # We use a dictionary again but this time with different scaling
        ↪   factors as keys
            self.assets[path] = {}
        try: # If we already have this scale in cache then return it
            return self.assets[path][scale]
        except: # Otherwise, we call the scaling function. Yes, even if the scale
        ↪   is raw, we let the function handle it.
            self.assets[path][scale] = self.__scale(scale, path)
            return self.assets[path][scale]

    def __scale(self, scale, path):
        # scale can be three things: the string "raw", a tuple of width and
        ↪   height floats, or a float
        # The float represents maintaining the original aspect ratio of the
        ↪   texture but scaling it so that the height equals the provided float
        ↪   multiplied by the height of the engine's window
        # If we got this far and the scale is raw thaen we actually need to load
        ↪   it, and all scales are transformations of this raw image
        if scale == "raw":
            try: # Try and load the image as a surface object
```

```
                return pygame.image.load(self.kap.load(path, quality=self.texture_quality, engine=self))
            except: # If we can't find it, load the default missing.png
                self.append_log(f"Error! Asset {path} was not found! Is your required KAP file loaded?")
                return self.get_asset("missing.png")
        self.append_log(f"Scaling asset {path} with settings {scale}")
        obj = self.get_asset(path)
        if isinstance(scale, tuple):
            # Just in case someone decided to not follow my instructions, ensures tuple of size 2
            if len(scale) == 1:
                scale = scale[0]
            else:
                scale = scale[:2]
        if isinstance(scale, float) or isinstance(scale, int):
            x = self.height*scale
            scale = (int((x/obj.get_height())*obj.get_width()), int(x))
        # Now in both cases we have a tuple of length two with target pixel values
        return pygame.transform.scale(obj, scale)

    def load_entity(self, entity, scene, *args, **kwargs):
        # While one could theoretically initialise an entity directly, it wouldn't do anything without the engine having ownership of it
        # So the parameter name entity is slightly misleading, as you actually pass in the uninitialised class as a type object
        # Here we assign the engine an ID and initialise it with it's required and optional parameters
        # The initialised entity is returned
        if Entity not in entity.__mro__:
            # This rudimentary check ensures that the type being passed in has inherited from the Entity class
            # At some point hopefully I'll be able to ensure than an entity has been set up properly with the right initalisation parameters
            raise TypeError("Cannot initialise non-entity")
        id = self.id
        e = entity(self, scene, id, *args, **kwargs)
        self.__entities.append(e)
        self.entity_id[id] = e
        return e

    def destroy_entity(self, entity):
        # Check if the object passed in is an entity (has inherited from the Entity class)
        if Entity not in type(entity).__mro__:
            raise TypeError("Cannot destroy non-entity")
        # Remove the object from the entity list and the entity ID dictionary.
        # Python will return an error if the object does not exist.
        self.__entities.remove(entity)
        del self.entity_id[entity.id]

    def load_scene(self, scene, *args, pbar=True, **kwargs):
        # Similar to load_entity, scene should not be an initialised scene but instead the class of the scene to be initialised
        # pbar being set to False will automatically not use the pbar even if the scene provided requests it, and should be True in most cases
        # *args and **kwargs will be passed into the scene that is being initialised
        # self.ready being set to False pauses the execution of updates and rendering while a scene is being loaded
        self.ready = False
        # Verifies that the class being passed in inherits from the Scene class
        if Scene not in scene.__mro__:
            raise TypeError("Invalid scene was blocked!")
        for x in scene.kap: # A scene should specify the KAP files it needs to be loaded
            # These files will already be loaded if they are not loaded already, so if multiple scenes need the same KAP file there is no reason to not list those KAP files in every scene
            if x not in self.kap.open:
                self.kap.load_kap(x, engine=self)
        # Destroy all non-persistant entities
        for x in self.__entities:
            if not x.persist:
                self.destroy_entity(x)
        # If pbar is enabled by the keyword parameter, and the scene has assets to load by pbar, then we call this function again to load the pbar class
        # This rudimentary implementation loads textures in their raw form, perhaps later I will implement the ability to scale with a progress bar
        # More information about the progress bar scene can be found in pbar.py
        if pbar and scene.load_with_pbar:
            self.load_scene(kris_engine.pbar.Pbar,
            # Function to get audio and image files
            # Pbar loading does not support fonts as fonts are always scaled to a given size, there is no raw form
            lambda x: self.get_asset(x, **({"audio": True} if x[-3:] == "ogg" or x[-3:] == "wav" else {})),
            scene.load_with_pbar,
            # Calls a function to get the size of files
            # This allows for the pbar to be incremented a different amount depending on the size of the file being loaded
            sum(self.__size(x) for x in scene.load_with_pbar),
            self.__size, # Function for measuring the amount to increment for each file
            # Information about the scene to be loaded once the progress bar is complete
            scene, *args, **kwargs)
        else:
            self.append_log(f"Scene being loaded: {scene}")
            # Grab information from the scene and initialise a scene object.
            self.update_rate, self.render_rate = scene.update_rate, scene.render_rate
            self.scene = scene(self, *args, **kwargs)
            self.update_counter, self.render_counter = 0, 0
            # Loop the music infinitely
            if self.scene.music:
                self.scene.music().play(loops=-1)
            # Reset timers
            self.scene_time = time.perf_counter()
            self.lag, self.lag2 = time.perf_counter(), time.perf_counter()
            # Allow updates and rendering to process
            self.ready = True

    def __size(self, x):
        # Gets information about a file's size from KAP file header. Returns raw size or size of given texture quality if available
        # It may be worth noting that the size being used for the pbar is the compressed size of the file, not it's uncompressed size
```

```python
        try: return self.kap.assets[x][1] if type(self.kap.assets[x]) == tuple else self.kap.assets[x][self.texture_quality][1]
        except: return 0

    def update_loop(self):
        # The function that is executed by the update threads
        # Assign the thread a text colour
        self.threads[threading.get_ident()] = {"colour": Colour(0x6cd663), "name": "Update"}
        self.append_log("Update thread started!")
        while self.running: # The thread will continue until every entity has been updated. Simply the most convenient way to implement it
            if self.ready: # If updates are allowed to proceed, otherwise do nothing
                self.update_counter += 1
                event = len(self.events)
                # Go through list of entities in order
                for e in self.__entities:
                    e.update()
                # All events are appended to self.events by the main thread
                # But it's the update thread that actually needs them so this means the events will be processed in the update then removed
                self.events = self.events[event:]
                t = time.perf_counter()
                # "How long do I have to wait to limit the rate of updates"
                # Except a millisecond less actually because it's better to do an update early than late
                s = (self.update_counter/self.update_rate)-(t-self.scene_time)-0.001
                if s > 0:
                    time.sleep(s) # wait
                # If it's negative then oh no we're behind
                elif self.lag2+2 < t:  # Explained earlier, no console spam
                    x = int(self.update_rate*-s)
                    if x > self.update_rate/20: # If we're less than 0.05 seconds behind why even bother
                        self.append_log(f"Warning! Update was {round(-s*1000, 1)}ms behind! {x+1} updates behind!")
                        self.lag2 = time.perf_counter()

    def render_loop(self):
        # The function that is executed by the update threads
        # Assign the thread a text colour
        self.threads[threading.get_ident()] = {"colour": Colour(0xf2e40d), "name": "Render"}
        self.append_log("Render thread started!")
        while self.running: # The thread will continue until every entity has been rendered and the frame updated. Simply the most convenient way to implement it
            if self.ready: # If rendering is allowed to proceed, otherwise do nothing
                self.render_counter += 1
                # Draw a scene's background, pygame applications typically wipe the entire screen every frame
                self.scene.background()
                # Go through list of entities in order
                for x in self.__entities:
                    x.render()
                while not self.event_gotten: # Wait for the main thread to get events
```

```python
                    time.sleep(0.001)
                pygame.display.update() # Draw the next frame on screen
                self.event_gotten = False # Tell the main thread to get the next updates
                t = time.perf_counter()
                if self.render_rate != 0: # The render thread allows frame rate to be uncapped, in which case we don't need waiting or lag reporting
                    # "How long do I have to wait to limit the frame rate"
                    # Except a millisecond less actually because it's better to do a frame early than late
                    s = (self.render_counter/self.render_rate)-(t-self.scene_time)-0.001
                    if s > 0:
                        time.sleep(s)# wait
                    # If it's negative then oh no we're behind
                    elif self.lag+2 < t: # Explained earlier, no console spam
                        x = int(self.render_rate*-s)
                        if x > 10: # If we're less than 10 frames behind why even bother
                            self.append_log(f"Warning! Frame was {round(-s*1000, 1)}ms behind! {x+1} frames behind target framerate!")
                            self.lag = time.perf_counter()

    @property # Getter that increments id every time you access the attibute to always produce a unique ID
    def id(self):
        self.__id += 1
        return self.__id

    @property # All these simple getters exist to make a private attribute accessible but read only
    def width(self):
        return self.__width

    @property
    def height(self):
        return self.__height

    @property
    def engine_path(self):
        return self.__engine_path

    @property
    def log_path(self):
        return self.__log_path

    @property
    def texture_quality(self):
        return self.__texture_quality

    @texture_quality.setter # Setter so that if the texture quality is changed, all the old textures of incorrect quality are flushed from the cache
    def texture_quality(self, a):
        self.append_log(f"Changing texture quality to {a}... this may take a moment")
        for x in list(self.assets.keys()):
            if type(self.assets[x]) == dict:
                del self.assets[x]
        self.__texture_quality = a
```

```

class Entity: # Every entity should inherit from this class
    persist = False # The entity will be destroyed upon loading a new scene
    # If True, then the entity will remain across scenes until explicitly
    ↪  destoroyed

    def __init__(self, engine: Engine, scene, id): # Every entity should have an
    ↪  engine, ID and scene attribute
        self.engine = engine
        self.id = id
        self.scene = scene
        # This code needs to be in a separate init function so that it's not
        ↪  called for every entity
        # The correct implementation is to:
        # - Inherit from the Entity class
        # - Have the first 3 positional arguments of the __init__ function be
        ↪  engine, scene and id, followed by your own parameters
        # - run super().__init__(engine, scene, id) as the first line to properly
        ↪  give your entity the attributes above
        if type(self) == Entity:
            self.init()

    def init(self): # Used for defining a default entity, for use in the default
    ↪  Scene class
        self.engine.append_log("The default entity class has been initialised.
        ↪  Please check the log for errors.")
        self.rotation = 0
        self.pre_calc_width = self.engine.width*0.99
        self.pre_calc_height = self.engine.height*0.01

    def update(self): # spin
        if type(self) == Entity:
            self.rotation += 0.025
            self.rotation %= 360
        # It is recommended that this function be overriden and replaced with
        ↪  pass if not used, or your own code
        # This is because checking if the current entity is the Entity class is
        ↪  more computationally expensive than skipping that check
        # It is additionally worth noting that that this is the function called
        ↪  every update by the update thread

    def render(self):
        if type(self) == Entity:
            w, h = self.engine.width/2, self.engine.height/2
            t = self.engine.get_asset("missing.png", scale=0.2)
            self.engine.screen.blit(pygame.transform.rotate(t, self.rotation),
            ↪  ((w)-(t.get_width()/2), (h)-(t.get_height()/2)))
        # It is recommended that this function be overriden and replaced with
        ↪  pass if not used, or your own code
        # This is because checking if the current entity is the Entity class is
        ↪  more computationally expensive than skipping that check
        # It is additionally worth noting that that this is the function called
        ↪  every frame by the render thread

class Scene: # Every scene should inherit from this class
    load_with_pbar = [] # The raw versions of the image and audio files in this
    ↪  list will be loaded with a progress bar displayed before the scene is
    ↪  initialised
    update_rate = 2400 # Updates per second, must be positive integer
    render_rate = 0 # Frames per second, must be positive integer, or 0 for
    ↪  uncapped
```

```python
    # The background function is called every frame before any entities are rendered and is usually used to clear the screen
    background = lambda self: self.engine.screen.fill(0) # Fills the screen with black
    # The music function should return a pygame sound object that will be looped for the duration of the scene
    # Tem Shop from the Undertale soundtrack belongs to Toby Fox and Materia Collective
    music = lambda self: self.engine.get_asset("tem_shop.ogg", audio=True)
    # This should be a list containing all KAP files that are required to be loaded for the scene to work
    kap = ["default.kap"]

    def __init__(self, engine: Engine):
        # Override this function! Use it to load your textures and entities
        # Have engine as your first input parameter and run super().__init__(engine), or just assign self.engine = engine yourself
        self.engine = engine
        if type(self) == Scene:
            self.engine.get_asset("missing.png", scale=0.2)
            e = engine.load_entity(Entity, self)
```

3.3 kris_engine/files.py

```python
from tqdm import tqdm
# My favourite progress bar
import os
# Used for file operations
from PIL import Image
# Used for image scaling when building
from io import BytesIO
# Used to return object in memory as buffer that acts like a file object
import atexit
# Close the KAP file on program termination
from math import log2
# Used for calculating number of options to check for RLE brute

class KAP:
# Presenting the KAP file: Kris's Asset Package!
# Now you can put all the textures and assets for your Kris's Engine application in one file
# Along with essential information about things like texture quality
# Compressed with run length encoding!

# A KAB file is structured in two parts: a string portion and a byte portion
# Some rules about the string portion:
# - All integers are unsigned and most significant bit first
# - All strings are UTF-8
# - A 0 byte is eight bits with a value 0

# The file begins with a 16-bit integer magic byte, 1993
# This helps verify that we're loading the right file type
# Then, an 8-bit integer representing the number of texture quality options available
# Then, for as many qualities as the integer specified:
    # string data detailing the name of the setting
    # a 0 byte
    # a non-zero 8-bit integer that the texture quality should be assigned to

# Then, there's a 32-bit integer detailing how many textures there are
# Each entry of the rest of the string portion is structured as follows:
# string data detailing the name of the file
# 0 byte
# 8-bit integer representing how many texture qualities the file has
# if this byte is 0:
    # 64-bit integer, indicating the pointer at which the file data starts in bytes
    # 64-bit integer, indicating the size of the file in bytes
# if this byte is greater than 0, repeat the following for the value of the byte:
    # 8-bit integer corresponding to a texture quality
    # 64-bit integer, indicating the pointer at which the file data starts in bytes
    # 64-bit integer, indicating the size of the file in bytes
# Then the corresponding files will be dumped as RLE bytes into the files at their corresponding pointers

# RLE bytes are bytes that are run length encoded.
# RLE bytes start with a byte that is comprised of two 4-bit integers
# The first represents counter_length, the second pattern_length
# These are required information to encode and decode RLE bytes
# Then, the excess. Take an example where we have a file with an odd number of bytes and pattern_length 2.
# For the algorithm to look for patterns of length 2, we need to trim the first byte off.
# So, the second byte is an 8-bit integer representing the size of the excess.
# If 0, move straight on to the data, otherwise append the following number of bytes to the front of the output.
# Then, the rest of the bytes are in the pattern of counter_length bytes, pattern_length bytes
# To decode, repeat the pattern_length bytes the value of the counter_length bytes times

    def __init__(self, path, engine=None):
        # You'll see quite a few "if engine" throughout this module
        # It can be used with or without the main Engine class being initialised
        # If the engine is initialised, the module will post log messages
        if engine:
            engine.append_log("Files module, initialising KAP class")

        self.file_map, self.assets = {}, {}
        # Filemap tells us which KAP file an asset comes from
        # Assets contains information about where in the file the asset is and how big it is
        self.open = []
        # A list of all the KAP files that are open, stripped to their base filename
        self.load_kap(path, engine=engine)
        atexit.register(self.__del__)
        # Will close file on program termination or if the KAP object is deleted with the del keyword

    def __del__(self):
        for x in set(self.file_map.values()):
            x.close()
        # Make sure KAP files are closed

    def load_kap(self, path, engine=None):
        if engine:
            engine.append_log(f"Files module, loading KAP file {path}")

        f = open(path, "rb")
        # File stays open for reduced data access time
        qualities={}
        if not f.read(2) == b'\x07\xc9':
            raise TypeError("Magic byte not found!")
        # Checks for magic byte at start of file to confirm file is of KAP type
        for x in range(f.read(1)[0]): # 8-bit integer referring to number of textures
            name = self.string_data(path, f) # Get texture name
            qualities[f.read(1)] = name # Store with 8-bit texture ID
        for x in range(int.from_bytes(f.read(4), "big")): # For number of textures in 32-bit integer
            name = self.string_data(path, f) # Get file name
            if (num_of_qualities := f.read(1)[0]) == 0: # Get number of qualities in 8-bit integer
                # If the number of qualities is 0
                self.assets[name] = (int.from_bytes(f.read(8), "big"), int.from_bytes(f.read(8), "big"))
                # Store 64-bit integer position, 64-bit integer size
            else: # If the number of qualities is more than 0
                self.assets[name] = {} # Then where our position and size tuple would be we store a dictionary
                for y in range(num_of_qualities):
                    key = qualities[f.read(1)] # Using the 8-bit texture ID to get the string name of the quality
                    self.assets[name][key] = (int.from_bytes(f.read(8), "big"), int.from_bytes(f.read(8), "big"))
                    # Store # Store 64-bit integer position, 64-bit integer size with string name of quality as key
            self.file_map[name] = f
            # Store in file map which KAP file this texture belongs to
        self.open.append(os.path.basename(path))
        # Add the filename to the list of open files if successful

    def string_data(self, path, file):
        # String data and other data are separated by 0 bytes
        # This function will return the decoded string data of unknown length
        pointer = file.tell()
        for x in range(pointer, os.path.getsize(path)):
            # For every byte between our current position in the file and the length of the file
            # It would make sense to use a while loop that continues infinitely
            # However if we never get a definitive 0 byte to signify the end of the script the function will error
            # This way, if we reach the end of the file, we'll return None, signifying no string was found
            if file.read(1) == bytes(1): # bytes(1) in python generates our 0 byte
                # Reading one byte from the file also moves the pointer by one
                # So, this loop reads through the file until it hits a 0 byte
                file.seek(pointer)
                # Then we return to our starting point
                y = file.read(x-pointer)
                # Read everything between our starting point and the 0 byte
                file.seek(1, 1)
                # Move past the 0 byte
                return y.decode("utf-8")
                # And decode the data so it's returned as a python string object!

    def load(self, filename, quality=None, engine=None):
        pointer = self.assets[filename]
        # This shortens finding the filename in the assets dictionary to "pointer" because it's less to type
        f = self.file_map[filename]
        # This shortens finding the file object in the file map dictionary to "f" because it's less to type

        if isinstance(pointer, dict): # If the object in the assets dictionary is a dictionary then the file has qualities
            if not quality: # If the function call didn't specify a quality
                quality = list(pointer.keys())[0] # Then we pick the first one in the dictionary, which using the build function will be the highest quality
            if engine:
                engine.append_log(f"Files module, loading {filename}, quality {quality}")
            position, size = pointer[quality] # Unpack our position and size tuple for the quality that we want
        else: # If it's not a dictionary the file doesn't have qualities and assets will only have a tuple in it
            if engine:
                engine.append_log(f"Files module, loading {filename}")
            position, size = pointer # Unpack position and size tuple

        f.seek(position) # Move to the correct position in the file
        x = f.read(size) # Read the number of bytes that make up the file
        return BytesIO(rle_decode(x)) # Decode the RLE bytes and return them as a file-like object

def rle_encode(bytedata, counter_length, pattern_length):
    if counter_length == 0 or pattern_length == 0:
        return bytes(1) + bytedata
    # Sometimes, the best encoding is no encoding at all
    # If either paramter is 0 the algorithm can't work so we just return the raw bytes with the encoding byte at the front
    if x := (len(bytedata) % pattern_length): # If excess, trim off front and store
        excess = bytedata[:x]
        data = bytedata[x:]
    else:
        excess = b''
        data = bytedata
    dat, counter = data[0:pattern_length], 0
    # Get a the number of bytes of length pattern_length from the front of the data
    # dat will be used to store whatever the last byte was for comparing
    # counter will be used to store how many times that byte has been seen
    out = [
        int((counter_length << 4) + pattern_length).to_bytes(1, "big"),
        int(len(excess)).to_bytes(1, "big"),
        excess
        ] + [b'']*(len(bytedata)*2)
    out_position = 3
    # Our output is a list to take advantage of a fun python quirk
    # So at the front of the list is the thing we need at the front of the file
    # In order to store the two 4-bit integers within a single byte in python
    # we need to take counter length, and shift it up 4 bits so it has 4 bits of empty space.
    # Then, we can add our pattern_length
    # The next byte stores how long the excess is in bytes
    # And then the excess data itself is added
    # The rest of the list is made up off empty bytes
    # This takes advantage of a quirk of python. When appending to a bytes
    ↪ object,
    # Python has to copy the entire object to an entirely new space in memory
    ↪ whenever it gets bigger.
    # This causes significant slowdown as the file increases in size, making the
    ↪ code unable to run.
    # The same applies to appending to lists, but to a lesser extent.
    # To counter this, we create a list that is bigger than we will need full of
    ↪ empty bytes objects
    # Then instead of expanding the list we use out_position to overwrite
    ↪ existing values, which is faster
    # because python does not have to find a new space in memory to store the
    ↪ list.
    # The list is filled with twice as many empty bytes objects as bytes in the
    ↪ bytedata
    # because the maximum length we could need to store would be a counter of 1
    ↪ on every single byte
    # Then if the list is big we can iterate through it to write it to a file,
    # or use bytes.join if the system we're on has enough memory.
    max_counter_capacity = 2**(counter_length*8)-1
    # Calculates the maximum value of an unsigned integer with size x bytes
    for a in tqdm(iterable=range(0, len(data), pattern_length), desc=f"RLE
    ↪ compression, counter_length {counter_length}, pattern_length
    ↪ {pattern_length}") if len(bytedata) > 10_000_000 else range(0, len(data),
    ↪ pattern_length):
    # Starts at 0, counts up through the length of the data with a step of
    ↪ pattern_length.
    # Creates a progress bar if the file is bigger than 10MB.
        x = data[a:a+pattern_length] # Get a chunk of data of size pattern
        ↪ length
        if x == dat: # If it's the same as the last byte
            if counter == max_counter_capacity: # And we've hit the maximum
            ↪ amount of repeats that the counter can store
                # Put the counter and data into our output
                out[out_position] = counter.to_bytes(counter_length, "big") + dat
                # Reset the counter and move forward to the next slot in the
                ↪ output
                counter = 1
                out_position += 1
            else:
                # If the counter isn't at it's maximum size increment the
                ↪ counter
                counter += 1
        else: # If it's different then the previous bit of data has stopped
        ↪ repeating
            # So we put that data and the number of times it repeated into the
            ↪ output
            out[out_position] = counter.to_bytes(counter_length, "big") + dat
            # Reset the counter and set the variable for the previous bit of data
            ↪ to our current bit of data
            counter = 1
            dat = x
            out_position += 1
    # Return our list of output data to be combined into a bytes object later (it
    ↪ may not be worth it)
    out[out_position] = counter.to_bytes(counter_length, "big") + dat
    return out

def rle_decode(bytedata):
```

```python
    # Two 4-bit integers contained within the first byte, so we used AND and bit shifting to get the two separate numbers
    counter_length = (bytedata[0] & 0b11110000) >> 4
    pattern_length = bytedata[0] & 0b00001111
    if counter_length == 0 or pattern_length == 0:
        return bytedata[1:] # No compression, so no decompression either, just trim the 0 byte from the front!
    # Here we have a byte representing the number of bytes of excess, so we trim that excess from the front
    if bytedata[1] == 0:
        excess = b''
        data = bytedata[2:]
    else:
        excess = bytedata[2:bytedata[1]+2]
        data = bytedata[bytedata[1]+2:]
    iterator = range(0, len(bytedata), counter_length+pattern_length)
    out = [excess] + [b'']*len(iterator)
    out_position = 1
    for x in iterator:
        dat = data[x:x+counter_length+pattern_length] # Get a chunk of data
        # Repeat the data for the number of times the pattern length specifies
        out[out_position] = (int.from_bytes(dat[:counter_length], "big") * dat[counter_length:])
        out_position += 1
    # Convert our list of bytes into one bytes object and return it
    return b''.join(out)

def rle_brute(bytedata): # Function for testing many compression settings and seeing which is the most effective
    z = rle_encode(bytedata, 0, 0) # No compression at all is our baseline
    out = ([z], len(z)) # Used to store the smallest output data and its size
    best_pattern_length = 0 # Once we've tested every pattern length with counter length 1, we only need to test every counter length with that pattern length
    a = int(log2(len(bytedata))/8) # Calculates how many pattern lengths need to be tested based on the size of the file being compressed
    if len(bytedata) > 1_000_000: # Use a progress bar if the file being compressed is larger than 1MB
        pbar = tqdm(desc="Compressing large file, please wait...", total=15+a-1)
    for x in range(1, 16):
        # Test all pattern lengths
        y = rle_encode(bytedata, 1, x)
        # Get size of output file by summing lengths, because function returns a list of bytes that will need to be combined
        length = sum(len(z) for z in y)
        if length < out[1]: # Store it if it's smaller than the smallest currently recorded
            out = (y, length)
            best_pattern_length = x
        try: pbar.update()
        except: pass
    if best_pattern_length:
        # Test all counter_lengths up to the one that would encapsulate the entire size of the file
        for x in range(1, a+1 if a < 15 else 16): # (Unless you are somehow compressing a super big file and then it maxes out at 15)
            y = rle_encode(bytedata, x, best_pattern_length)
            length = sum(len(z) for z in y)
            if length < out[1]:
                out = (y, length)
            try: pbar.update()
            except: pass
    else:
        try: pbar.update(a-1)
        except: pass
    # If the compressed data is less than 50MB then we combine it now and return
    ↪  it as a single bytes object
    # Otherwise, you would need several gigabytes in order to use .join, so you
    ↪  must iterate through the list of bytes in order to write it to a file
    return b''.join(out[0]) if out[1] < 50_000_000 else out[0]

# Function used for the creation of KAP files
# qualities is a dictionary: {quality name: float scale} where the float is the
↪  multiplier with which images should be scaled
# with_quality is a list of filenames
# without_quality is a list of filenames
# out_path is a string specifying the output file path
def build(qualities, with_quality, without_quality, out_path, compress=True):
    print("Building KAP file...")
    zero = bytes(1) # eight 0 bits
    if len(qualities) > 255:
        raise OverflowError("Too many qualities! Why do you need that many?")
    elif len(qualities) == 0:
        quality_ids = {"empty":int(1).to_bytes(1, "big")}
    else:
        # unique integer are generated by incrementing
        quality_ids = {y:x.to_bytes(1, "big") for x,y in
        ↪  enumerate(qualities.keys(), start=1)}
    pointers = {}
    with open(out_path, "wb") as f:
        f.write(b'\x07\xc9') # Magic byte 1993
        f.write(len(qualities).to_bytes(1, "big"))
        for x in qualities: # For each quality write it's name and ID
            f.write(x.encode("utf-8"))
            f.write(zero)
            f.write(quality_ids[x])
        # 32-bit integer, how many textures in total (counting quality variants
        ↪  as the same texture)
        f.write((len(without_quality)+len(with_quality)).to_bytes(4, "big"))
        for x in without_quality: # Until we've compressed the data we don't know
        ↪  the location and sizes so we leave a blank placeholder
            f.write(x.encode("utf-8"))
            f.write(zero*2)
            pointers[x] = f.tell()
            f.write(zero*16)
        for x in with_quality:
            f.write(x.encode("utf-8"))
            f.write(zero)
            f.write(len(qualities).to_bytes(1, "big"))
            pointers[x] = {}
            for y in qualities:
                f.write(quality_ids[y])
                pointers[x][y] = f.tell()
                f.write(zero*16)
        for x in tqdm(iterable=without_quality, desc="Dumping raw data for assets
        ↪  without quality"):
            with open(x, "rb") as y:
                raw = y.read()
                raw = rle_brute(raw) if compress else rle_encode(raw, 0, 0)
                # Write the compressed data to the file, then go back to the
                ↪  file's location in the header and write it's location and
                ↪  size
                pointer = f.tell()
                f.seek(pointers[x])
                f.write(pointer.to_bytes(8, "big"))
                f.write(len(raw).to_bytes(8, "big"))
                f.seek(0, 2)
                # If it's a list of bytes that is too big to combine we iterate through that list, otherwise we write the bytes object
                if type(raw) == list:
                    for z in raw:
                        f.write(z)
                else:
                    f.write(raw)
        for x in tqdm(iterable=with_quality, desc="Dumping raw data for assets with quality"):
            raw = Image.open(x)
            for y in qualities:
                # The same as before but now we are resizing the images to different texture qualities and storing information about each quality
                out_raw = BytesIO()
                out = raw.resize((int(round(raw.width*qualities[y], 0)) or 1, int(round(raw.height*qualities[y], 0)) or 1)) if qualities[y] != 1 else raw
                out.save(out_raw, "png", optimize=True)
                out_raw = rle_brute(out_raw.getvalue()) if compress else rle_encode(out_raw.getvalue(), 0, 0)
                pointer = f.tell()
                f.seek(pointers[x][y])
                f.write(pointer.to_bytes(8, "big"))
                f.write(len(out_raw).to_bytes(8, "big"))
                f.seek(0, 2)
                if type(out_raw) == list:
                    for z in out_raw:
                        f.write(z)
                else:
                    f.write(out_raw)
    print("Done!")
```

3.4 kris_engine/pbar.py

```
from kris_engine import Scene, Entity
from kris_engine.colour import Colour
import threading
import pygame

class ProgressBar(Entity):
    # func is the thing that's going to be called on each item during loading, this is get_asset by default
    # iterable is the list of things that func is being called on
    # total represents the size of the progress bar, not on screen but relative to prog_incr
    # prog_incr is a function that each item of iterable can be passed into, that returns the amount that the progress bar should increase by upon processing that item
    # exit_scene is the scene that will be loaded once loading is complete
    # *args and **kwargs are the initialisation parameters for the scene
    def __init__(self, engine, scene, id, func, iterable, total, prog_incr, exit_scene, *args, **kwargs):
        super().__init__(engine, scene, id)
        self.total = total
```

```python
        self.prog_incr = prog_incr
        self.progress = 0
        self.engine.pbar_out = None
        # Start the progress bar thread
        self.thread = threading.Thread(target=self.monitor_progress, args=(func, iterable, self.engine, exit_scene, *args), kwargs=kwargs)
        self.thread.start()

    def update(self):
        pass

    def render(self):
        # Draws the progress bar, it's just a rectangle that is cropped by the area parameter based on how much progress we've made relative to total
        w, h = self.engine.width, self.engine.height
        thickness = 0.006*h

        t = self.engine.get_asset("pbar.png", scale=(0.8*w, 0.1*h))
        self.engine.screen.blit(t, (0.1*w, 0.7*h), area=(0, 0, t.get_width()*(self.progress/self.total), t.get_height()))

        pygame.draw.rect(self.engine.screen, 0xffffff, (w*0.1, h*0.7, w*0.8, h*0.1), round(thickness))

    def monitor_progress(self, func, iterable, engine, exit_scene, *args, **kwargs):
        # Give the progress bar thread a text colour
        self.engine.threads[threading.get_ident()] = {"colour": Colour(0x4C19FF), "name": "Progress Bar"}
        engine.pbar_out = []
        # For each item in iterable, call function on it, update the progress variable and check if the engine is still running
        # This check is done so that if the user quits during a progress bar, then loading quits also
        for x in iterable:
            try: engine.pbar_out.append(func(x))
            except: pass
            try: self.progress += self.prog_incr(x)
            except: pass
            if not self.engine.running:
                return
        # Load the exit scene once loading is complete, but this time without using the progress bar again
        engine.load_scene(exit_scene, *args, pbar=False, **kwargs)

class Pbar(Scene):
    update_rate = 1
    # A low render rate make the progress bar appear choppy but reduces load times
    render_rate = 5
    music = None

    def __init__(self, engine, func, iterable, total, prog_incr, exit_scene, *args, **kwargs):
        # Loads the image used on the progress bar and initialises the progress bar entity
        self.engine = engine
        self.engine.get_asset("pbar.png")
        e = self.engine.load_entity(ProgressBar, self, func, iterable, total, prog_incr, exit_scene, *args, **kwargs)
```

3.5 exercise.py

```
from kris_engine import Scene
from gameplay import Grid, Bean, InputHandler
from random import randint
# Load classes
# randint used for testing by filling grid with random beans

class ExerciseClassic(Scene):
    # attributes that are explained in __init__.py
    kap = ("base.kap",)
    load_with_pbar = ("exercise.ogg", "pop1.ogg", "pop2.ogg", "pop3.ogg",
    ↪  "pop4.ogg", "pop5.ogg", "pop6.ogg", '1_1.png', '1_10.png', '1_11.png',
    ↪  '1_12.png', '1_13.png', '1_14.png', '1_15.png', '1_16.png', '1_17.png',
    ↪  '1_18.png', '1_19.png', '1_2.png', '1_20.png', '1_21.png', '1_22.png',
    ↪  '1_23.png', '1_24.png', '1_25.png', '1_26.png', '1_27.png', '1_3.png',
    ↪  '1_4.png', '1_5.png', '1_6.png', '1_7.png', '1_8.png', '1_9.png',
    ↪  '2_1.png', '2_10.png', '2_11.png', '2_12.png', '2_13.png', '2_14.png',
    ↪  '2_15.png', '2_16.png', '2_17.png', '2_18.png', '2_19.png', '2_2.png',
    ↪  '2_20.png', '2_21.png', '2_22.png', '2_23.png', '2_24.png', '2_25.png',
    ↪  '2_26.png', '2_27.png', '2_3.png', '2_4.png', '2_5.png', '2_6.png',
    ↪  '2_7.png', '2_8.png', '2_9.png', '3_1.png', '3_10.png', '3_11.png',
    ↪  '3_12.png', '3_13.png', '3_14.png', '3_15.png', '3_16.png', '3_17.png',
    ↪  '3_18.png', '3_19.png', '3_2.png', '3_20.png', '3_21.png', '3_22.png',
    ↪  '3_23.png', '3_24.png', '3_25.png', '3_26.png', '3_27.png', '3_3.png',
    ↪  '3_4.png', '3_5.png', '3_6.png', '3_7.png', '3_8.png', '3_9.png',
    ↪  '4_1.png', '4_10.png', '4_11.png', '4_12.png', '4_13.png', '4_14.png',
    ↪  '4_15.png', '4_16.png', '4_17.png', '4_18.png', '4_19.png', '4_2.png',
    ↪  '4_20.png', '4_21.png', '4_22.png', '4_23.png', '4_24.png', '4_25.png',
    ↪  '4_26.png', '4_27.png', '4_3.png', '4_4.png', '4_5.png', '4_6.png',
    ↪  '4_7.png', '4_8.png', '4_9.png', '5_1.png', '5_10.png', '5_11.png',
    ↪  '5_12.png', '5_13.png', '5_14.png', '5_15.png', '5_16.png', '5_17.png',
    ↪  '5_18.png', '5_19.png', '5_2.png', '5_20.png', '5_21.png', '5_22.png',
    ↪  '5_23.png', '5_24.png', '5_25.png', '5_26.png', '5_27.png', '5_3.png',
    ↪  '5_4.png', '5_5.png', '5_6.png', '5_7.png', '5_8.png', '5_9.png',
    ↪  'backdrop3.png', "backdrop8.png")
    music = lambda self: self.engine.get_asset("exercise.ogg", audio=True)
    background = lambda self:
    ↪  self.engine.screen.blit(self.engine.get_asset("backdrop8.png",
    ↪  scale=(self.engine.width, self.engine.height)), (0, 0))
    update_rate = 300

    def __init__(self, engine):
        # Load the input handler and grid entities. The comments you see below
        ↪  were used for testing.
        self.engine = engine
        input_handler = self.engine.load_entity(InputHandler, self)
        e2 = self.engine.load_entity(Grid, self, input_handler)#,
        ↪  values=[Bean(randint(1,5)) for x in range(60)])
        #e = self.engine.load_entity(Grid, self, values=[
        #    Bean(1), Bean(5), Bean(5), Bean(5), Bean(1), Bean(2),
        #    Bean(1), Bean(2), Bean(3), Bean(4), Bean(1), Bean(2),
        #    Bean(1), Bean(2), Bean(3), Bean(4), Bean(1), Bean(2),
        #    Bean(1), Bean(3), Bean(4), Bean(5), Bean(2), None,
        #    Bean(2), Bean(3), Bean(4), Bean(1), None, None,
        #    Bean(2), None, None, None, None, None,
        #    None, None, None, None, None, None,
        #    None, None, None, None, None, None,
        #    None, None, None, None, None, None,
        #    None, None, None, None, None, None,
        #    None, None, None, None, None, None,
        #    None, None, None, None, None, None,
        #])
```

3.6 gameplay.py

```
from kris_engine import Entity
from random import randint
from copy import copy
import pygame
# Pygame used for key library and some constants

# Constants that effect program behaviour

COLOUR_IDS = {
    "RED": 1,
    "GREEN": 2,
    "YELLOW": 3,
    "PURPLE": 4,
    "CYAN": 5
    }

TEXTURE_STATE_IDS = {
    "WHITE": 16,
    "NORMAL": 17,
    "SQUISH1": 18,
    "SQUISH2": 19,
    "SHOCKED": 20
}

GAMEPLAY_STATES = (
    "FALL",
    "GRAVITY",
    "GRAVITY_ANIMATION",
    "VERIFY",
    "VERIFY_ANIMATION",
    "DIE"
)

ACTION_IDS = {
    "NOTHING": 0,
    "START": 1,
    "MOVE_LEFT": 2,
    "MOVE_RIGHT": 3,
    "DOWN": 6,
}

HANDLING_SETTINGS = {
    "DAS": 45,
    "ARR": 15
}

# Grid is an entity
class Grid(Entity):
    # columns is more important as rows is not restricting, the grid can contain
    ↪ more than that many rows of beans but it defines the death point for the
    ↪ grid
    # values allows a non-empty grid to be loaded
    def __init__(self, engine, scene, id, input_handler, rows=12, columns=6,
    ↪ values=[], position=(16/320, 16/224), bean_queue_position=(0.4, 40/224)):
        super().__init__(engine, scene, id)
        self.input_handler = input_handler
        # score entity aggregated by grid entity
        self.score = self.engine.load_entity(Score, scene)
        # 1D list filled with Bean objects or None representing an empty cell
        self.values = values
        # used to store GravityBeans
        self.gravity = []
        self.state = "GRAVITY"
        # Used for score calculation
        # Falling beans can set off chains, chain power increments each time we
        ↪ reach the VERIFY state without spawning a new falling bean
        # Spawning a new FallingBean will reset the counter
        self.chain_power = 0
        # Scoring bonus for when multiple groups are popped at once that have
        ↪ different colours
        self.colour_bonus = 0
        # Scoring bonus for when more than 4 beans are popped at once
        self.group_bonus = 0
        # Scoring calculation variable representing the maximum number of beans
        ↪ popped at once during any point in the chain reaction
        self.beans_popped = 0
        self.rows, self.columns = rows, columns
        # Commonly used in drawing calculations, so precalculated to prevent
        ↪ unnecessary repeated division
        self.cache = (16/320, 16/224)
        self.position = position
        # Correct textures if a non-empty grid has been loaded
        self.eval_all_textures()
        # List representing groups that are being popped during a chain reaction
        self.groups = []
        # Set of all beans that need to be popped
        self.verify = self.count_all()

        # BeanQueue entity aggregated by Grid object
        self.queue = self.engine.load_entity(BeanQueue, self.scene,
        ↪ bean_queue_position)
        # method of BeanQueue used to get the next falling bean
        self.falling = FallingBean(*self.queue.get_next(), self)

    def __str__(self): # Used to convert the current grid into a neat string for
    ↪ debugging
        return "\n".join(" ".join(str(self.values[y].colour if y <
        ↪ len(self.values) and self.values[y] else 0) for y in
        ↪ range((x-1)*self.columns, x*self.columns)) for x in range(self.rows,
        ↪ 0, -1))

    def update(self):
        # Lookup table for what to do depending on what state the program is in
        match self.state:
            case "GRAVITY":
                self.update_gravity()
            case "GRAVITY_ANIMATION":
                self.animate_gravity()
            case "VERIFY":
                self.update_verify()
            case "VERIFY_ANIMATION":
                self.animate_verify()
            case "FALL":
                self.falling.update()
            case "DIE":
                self.engine.append_log("Death not yet implemented, terminating, close the program")
                self.engine.destroy_entity(self)
            case _:
                pass

    def update_verify(self):
        # If we are in the verify state, check if there are any beans to be popped (calculated by gravity beans when they terminate)
        if self.verify:
            self.chain_power += 1
            # We use a set to remove duplicates to get the number of unique colours that are currently being popped, for the colour bonus
            colours = set()
            if (z:= sum(len(x) for x in self.groups)) > self.beans_popped:
                # beans_popped is the maximum number of beans that are popped at the same time during the chain reaction
                # So we overwrite it if we have bigger than the current value
                self.beans_popped = z
            # Bonus for the number of groups being popped at a time
            for x in self.groups:
                self.group_bonus += self.group_bonus_lookup(len(x))
                colours.add(x.pop())
            self.colour_bonus += self.colour_bonus_lookup(len(colours))
            # Change the score display to show the current ongoing score calculation
            # Score is not added until the chain is complete
            self.score.text = (f"{10*self.beans_popped}x{z}" if (z := self.chain_power_lookup(self.chain_power)+self.group_bonus+self.colour_bonus) else str(10*self.beans_popped)).rjust(9)
            self.state = "VERIFY_ANIMATION"
            self.counter = 0
            self.counter_goal = 300
            temp = copy(self.verify)
            # Reset the variables containing the beans we're about to pop
            self.verify = set()
            self.groups = []
            for x in temp:
                # Remove our colour groups from the grid
                self.values[x].row = x // self.columns
                self.values[x].column = x % self.columns
                self.verify.add(self.values[x])
                self.values[x] = None
        else: # The chain has ended
            try:
                # Check for death
                if self.values[(self.rows-1)*self.columns + 2]:
                    self.state = "DIE"
                    self.score.dump_score()
                    self.score.output_top_scores()
            except: pass # If the function errored then the grid isn't big enough to contain the death cell
            # In which case it definitely doesn't have anything in it
            if self.state != "DIE":
                # Get our next falling bean
                self.falling = FallingBean(*self.queue.get_next(), self)
                self.state = "FALL"
                # Update the score and the score text to match
                self.score.score += 10*self.beans_popped*(z if (z := self.chain_power_lookup(self.chain_power)+self.group_bonus+self.colour_bonus) else 1)
                self.score.text = str(self.score.score).zfill(9)
                # Reset our calculating variables
                self.chain_power = 0
                self.colour_bonus = 0
                self.group_bonus = 0
                self.beans_popped = 0

    @staticmethod
    def chain_power_lookup(CP):
        if CP == 1:
            return 0
        if CP > 8:
            return 999
        return 2**(CP+1)

    @staticmethod
    def colour_bonus_lookup(CB):
        if CB == 1:
            return 0
        return 2**(CB-2)*3

    @staticmethod
    def group_bonus_lookup(GB):
        if GB < 5:
            return 0
        if GB > 10:
            return 10
        return GB-3

    def animate_verify(self): # Simple animation implementation
        self.counter += 1
        if self.counter == self.counter_goal:
            self.verify = set()
            self.state = "GRAVITY"
        elif self.counter == 180:
            for x in self.verify:
                x.state = TEXTURE_STATE_IDS["SHOCKED"]
        elif self.counter == 230: # The pitch of the sound gets higher as longer chains are produced
            self.engine.get_asset(f"pop{self.chain_power if self.chain_power < 7 else 6}.ogg", audio=True).play()

    def animate_gravity(self):
        for x in self.gravity:
            try:
                # Gravity beans are animated using generator objects that can cleanly set things like their current position and texture
                next(x.position)
            except StopIteration:
                # When the animation finishes we make a new bean in it's place, update it's texture and see if it fell to make a group
                self.gravity.remove(x)
                col = x.colour
                bean = Bean(col)
                self.values[x.destination] = bean
                self.eval_surrounding(x.destination)
                self.count(x.destination)
        if not self.gravity: # Once all gravity beans have finished their animations
            self.counter += 1 # A short pause before the next verification
            if self.counter == self.counter_goal:
                self.state = "VERIFY"

    def update_gravity_quick(self): # A faster implementation of gravity, but it doesn't let use know which beans have fallen in the resulting grid
        # Would be useful for replays
        for n in range(self.columns):
            column = [self.values[x] for x in range(n, len(self.values), self.columns) if self.values[x] != None]
            for x in range(n, len(self.values), self.columns):
                if column:
                    self.values[x] = column[0] if column else None
                    column.pop(0)
                else:
                    self.values[x] = None

    def update_gravity(self):
        # For every column
        for n in range(self.columns):
            # Extracts a column from the 1D list
            column = [self.values[x] for x in range(n, len(self.values), self.columns)]
            # If there are no empty cells then there's no reason to check for falling beans
            if None in column:
                # Returns a 2D list which, flattened into a 1D list would tell us the final resulting output of the column on the grid
                # However, the position of the list in the outer list tells us how many rows a bean will fall
                split_column = self.split_list(column, None)
                # For all the distances of 1 or more
                for a in range(1, len(split_column)):
                    # For all the beans falling that distance
                    for b in split_column[a]:
                        # Get the current row from the original column data
                        row =  column.index(b)
                        # Create a gravity bean, inputing the bean's destination with what we already calculated
                        self.gravity.append(GravityBean(b, n+(row-a)*self.columns, a, row, n))
                # For all the beans in the column above the ones that didn't fall at all
                for x in range(n + len(split_column[0])*self.columns, len(self.values), self.columns):
                    # Remove them from the grid (they'll be replaced when the gravity beans are done falling)
                    self.values[x] = None
        # This represents the small wait after all the beans are finished falling
        self.counter = 0
        self.counter_goal = 50
        self.state = "GRAVITY_ANIMATION"

    @staticmethod
    # Works like the split method on a string, but on a 1D list, returning a 2D list
    def split_list(lst, sep):
        out = []
        x = 0
        for n in range(len(lst)):
            if lst[n] == sep:
                out.append(lst[x:n])
            x = n + 1
        out.append(lst[x:])
        return out

    def place_bean(self, bean, position):
        # Pads the list with empty cells if it's not long enough already
        self.values += [None]*(position-len(self.values))
        self.values[position] = bean
        # Checks for new colour groups and updates the bean's texture
        self.count(position)
        self.eval_surrounding(position)

    def render(self):
        # Draw the grid with all the beans in it
        self.render_grid()
        # Do any special drawing a state may require
        match self.state:
            case "GRAVITY_ANIMATION":
                self.render_gravity()
            case "VERIFY_ANIMATION":
                self.render_verify()
            case "FALL":
                self.render_falling()
            case _:
                pass
        # Then draw backdrop3.png last
        # This means that beans that are outside of the grid, such as falling beans that just spawned in, appear behind the background
        # Not seeing this represents death, since the bottom of the grid has opened up
        self.engine.screen.blit(self.engine.get_asset("backdrop3.png", scale=(self.engine.width, self.engine.height)), (0, 0))

    def render_grid(self):
        # Draws our 1D array in a 2D grid on screen by moving to the next row each time we hit the number of columns
        column = 0
        row = 0
        for x in self.values:
            if type(x) == Bean:
                self.render_bean(x, row, column)
            column += 1
            if column == self.columns:
                column = 0
                row += 1

    def render_bean(self, x, row, column):
        # Used all over the code to draw a bean on screen
        self.engine.screen.blit(
            self.engine.get_asset(x.texture, scale=self.cache[1]),
            ((self.cache[0]*column+self.position[0])*self.engine.width,
            (1-(self.cache[1]*row+self.position[1]*2))*self.engine.height)
        )

    def render_gravity(self):
        # Gravity beans are already being animated by the update thread so we just need to draw them in their current state
        for x in self.gravity:
            self.render_bean(x, x.row, x.column)

    def render_verify(self):
        # Beans that are being popped flash on and off the screen every other frame, which instead of having an empty texture is done by just not drawing it
        if self.counter > 230:
            pass
        if 25 < self.counter < 180 and self.counter%10 < 5:
            for x in self.verify:
                self.render_bean(x, x.row, x.column)

    def render_falling(self):
        # Draws the primary falling bean, renders the position of the secondary bean, which is relative to the primary bean and dependant on rotation state
        self.render_bean(self.falling.primary, self.falling.row, self.falling.column)
        match self.falling.rotation_state:
            case 0:
                self.render_bean(self.falling.secondary, self.falling.row+1, self.falling.column)
            case 1:
                self.render_bean(self.falling.secondary, self.falling.row, self.falling.column+1)
            case 2:
                self.render_bean(self.falling.secondary, self.falling.row-1, self.falling.column)
            case 3:
                self.render_bean(self.falling.secondary, self.falling.row, self.falling.column-1)
            case _:
                raise ValueError("Invalid rotation state")

    def eval_all_textures(self):
        # Used for getting the right texture for every bean when loading in a grid instead of updating in real time
        for pos in range(len(self.values)):
            self.eval_texture(pos)

    def eval_texture(self, bean_pos):
        # Updates the texture for a given bean, in every direction
        try:
            bean = self.values[bean_pos]
            col = bean.colour
        except:
            return None

        self.eval_up(bean_pos, bean, col),
        self.eval_down(bean_pos, bean, col),
        self.eval_left(bean_pos, bean, col),
        self.eval_right(bean_pos, bean, col)

    # The below are functions that check whether a bean in a given direction exists and is of the same colour and sets that attribute of the bean

    def eval_up(self, bean_pos, bean, col):
        try:
            if self.values[bean_pos+self.columns].colour == col:
                x = True
            else:
                x = False
        except:
            x = 0

        bean.up = x

    def eval_down(self, bean_pos, bean, col):
        try:
            if bean_pos-self.columns < 0:
                x = 0
            elif self.values[bean_pos-self.columns].colour == col:
                x = True
            else:
                x = False
        except:
            x = 0

        bean.down = x

    def eval_left(self, bean_pos, bean, col):
        try:
            if not bean_pos % self.columns:
                x = 0
            elif bean_pos-1 < 0:
                x = 0
            elif self.values[bean_pos-1].colour == col:
                x = True
            else:
                x = False
        except:
            x = 0

        bean.left = x

    def eval_right(self, bean_pos, bean, col):
        try:
            if not (bean_pos + 1) % self.columns:
                x = 0
            elif self.values[bean_pos+1].colour == col:
                x = True
            else:
                x = False
        except:
            x = 0

        bean.right = x

    def eval_surrounding(self, bean_pos):
        # Updates the texture for a bean as well as all the beans surrounding it
        for x in (bean_pos, bean_pos+self.columns, bean_pos-self.columns, bean_pos-1, bean_pos+1):
            self.eval_texture(x)

    def count_all(self): # A function to scan the entire grid for colour groups. Similar to count, see explanation there
        to_be_counted = [x for x in range(len(self.values)) if type(self.values[x]) == Bean]
        to_be_destroyed = set()
        while to_be_counted:
            group = {to_be_counted[0]}
            surroundings = self.get_surroundings(to_be_counted[0], group)
            for x in surroundings:
                group.add(x)
                surroundings += self.get_surroundings(x, group)
            for x in group:
                to_be_counted.remove(x)
            if len(group) >= 4:
                to_be_destroyed = to_be_destroyed.union(group)
                self.groups.append(group)
        return to_be_destroyed

    def count(self, bean): # Checks if a bean is in a colour group
        # First, we check if the bean has already been detected as being in a colour group. If it hasn't,
        if bean not in self.verify:
            # Create a set containing the bean
            group = {bean}
            # Use get surroundings to get all the adjacent beans of the same colour
            surroundings = self.get_surroundings(bean, group)
            # We iterate through the list
            for x in surroundings:
                # And add all of these adjacent same colour beans to our set
                # We will get duplicates but those will be removed by the set
                group.add(x)
                # And then we find the same colour beans that are adjacent to those beans
                surroundings += self.get_surroundings(x, group)
            # Iteration will continue until the end of the list is reached, which signifies that every bean in the set doesn't have an adjacent bean that isn't already in the set
            # So if the group is bigger than 4, then set it to be popped
            if len(group) >= 4:
                self.verify = self.verify.union(group)
                self.groups.append(group)

    def get_surroundings(self, x, group):
        tests = []
        if x+self.columns < len(self.values): # If the row above the bean is contained within the grid
            tests.append(x+self.columns) # Check the bean above
        if x-self.columns >= 0: # If the bean isn't on the bottom row
            tests.append(x-self.columns) # Check the bean below
        if x % self.columns: # If the bean isn't on the left edge
            tests.append(x-1) # Check the bean on the left
        if (x + 1) % self.columns: # If the bean isn't on the right edge
            tests.append(x+1) # Check the bean on the right
        # Return a list of beans that share the same colour
        return [y for y in tests if y >= 0 and y < len(self.values) and self.values[y] and self.values[y].colour == self.values[x].colour and y not in group]

# BeanQueue is an Entity
class BeanQueue(Entity):
    def __init__(self, engine, scene, id, position):
        super().__init__(engine, scene, id)
        # Beans are randomly generated using randint
        self.next = (Bean(randint(1,5)), Bean(randint(1, 5)))
        self.cache = 16/224
        self.position = position

    def render(self): # An exact copy of the render_bean function, but must be duplicated because passing in the grid overcomplicates things
        self.engine.screen.blit(
            self.engine.get_asset(self.next[0].texture, scale=self.cache),
            (self.position[0]*self.engine.width,
            self.position[1]*self.engine.height)
        )
        self.engine.screen.blit(
            self.engine.get_asset(self.next[1].texture, scale=self.cache),
            (self.position[0]*self.engine.width,
            (self.cache+self.position[1])*self.engine.height)
        )

    def update(self):
        # Was planned to be animated but dropped due to time constraints, the potential is still there
        pass

    def get_next(self): # Returns a tuple of two beans and queues the next pair
        x = self.next
        self.next = (Bean(randint(1,5)), Bean(randint(1, 5)))
        return x

# Bean is NOT an Entity, it is just aggregated by Grid
class Bean:
    # The values of up, down, left and right may be slightly confusing
    # But they are restricted to 0, False and True for performance
    # 0 means there is no bean there
    # False means there is a bean there, but it's not the same colour
    # True means there is a bean there, and it's the same colour
    def __init__(self, colour, up=0, down=0, left=0, right=0, state=0):
        self.colour = colour if type(colour) == int else COLOUR_IDS[colour]
        self.__up = up
        self.__down = down
        self.__left = left
        self.__right = right
        self.__state = state
        self.get_texture()

    def __str__(self): # For debugging
        return f"Bean({self.colour})"
    __repr__ = __str__

    def get_texture(self):
        # The combination of up, down, left and right bits can be used to generate a unique 4-bit integer representing the correct texture
        # self.state is used to assign other textures that don't apply to this, and it takes priority over this texture
        if self.state:
            texture = self.state
        else:
            texture = (self.up << 3) + (self.down << 2) + (self.left << 1) + self.right
        if texture == 0:
            texture = TEXTURE_STATE_IDS["NORMAL"]
        self.texture = f"{self.colour}_{texture}.png"
        return self.texture

    # The below setters are used to automatically update the texture of a bean when one of the parameters affecting it's texture changes

    @property
    def state(self):
        if not self.__state and not (self.up or self.down or self.left or self.right):
            return TEXTURE_STATE_IDS["NORMAL"]
        return self.__state

    @property
    def up(self):
        return self.__up

    @up.setter
    def up(self, a):
        self.__up = a
        self.get_texture()

    @property
    def down(self):
        return self.__down

    @down.setter
    def down(self, a):
        self.__down = a
        self.get_texture()

    @property
    def left(self):
        return self.__left

    @left.setter
    def left(self, a):
        self.__left = a
        self.get_texture()

    @property
    def right(self):
        return self.__right

    @right.setter
    def right(self, a):
        self.__right = a
        self.get_texture()

    @property
    def state(self):
        return self.__state

    @state.setter
    def state(self, a):
        self.__state = a
        self.get_texture()

# GravityBean inherits from Bean, but it is not an Entity
class GravityBean(Bean):
    def __init__(self, bean, destination, row_distance, row, column):
        super().__init__(bean.colour)

        self.row = row
        self.column = column
        self.destination = destination
        # Row distance is the number of rows that the bean will fall
        self.row_distance = row_distance


        # As shown above generators are really helpful for animations
```

```python
        # They allow me to use for loops where each iteration of the for loop represents an update
        # It allows for this really clean animation code that allows me to run any code I want, in this case changing positions and textures
        def position():
            #for n in range(40):
            #    yield
            self.state = TEXTURE_STATE_IDS["NORMAL"]
            for n in range(row_distance*10):
                if n % 5 == 0:
                    self.row -= 0.5
                yield
            for i in range(2):
                self.state = TEXTURE_STATE_IDS["NORMAL"]
                for n in range(5):
                    yield
                self.state = TEXTURE_STATE_IDS["SQUISH2"]
                for n in range(15):
                    yield
                self.state = TEXTURE_STATE_IDS["NORMAL"]
                for n in range(5):
                    yield
                self.state = TEXTURE_STATE_IDS["SQUISH1"]
                for n in range(10):
                    yield
            self.state = TEXTURE_STATE_IDS["NORMAL"]
            for n in range(5):
                yield
            self.state = TEXTURE_STATE_IDS["SQUISH2"]
            for n in range(10):
                yield
            self.state = TEXTURE_STATE_IDS["NORMAL"]
            for n in range(5):
                yield
            self.state = TEXTURE_STATE_IDS["SQUISH1"]
            for n in range(5):
                yield
            self.state = TEXTURE_STATE_IDS["NORMAL"]
            for n in range(5):
                yield
            self.state = TEXTURE_STATE_IDS["SQUISH2"]
            for n in range(30):
                yield

        self.position = position()

# FallingBean is not an entity
# It has an update method, but this is simply called by the grid
# The grid is passed in upon initialisation because grid values are accessed a lot
class FallingBean:
    def __init__(self, top, bottom, grid: Grid):
        # The secondary bean rotates around the primary bean
        self.primary = Bean(bottom.colour)
        self.secondary = Bean(top.colour)
        self.grid = grid

        self.row = grid.rows
        self.column = 2
        self.counter = 0
        # It would be really easy to implement levels I just ran out of time
        self.fall_rate = 120

        self.rotation_state = 0
        # Let P be the primary bean and S be the secondary bean
        # 0 -   S
        #       P
        # 1 -   PS
        # 2 -   P
        #       S
        # 3 -   SP

    def update(self):
        self.counter += 1

        # Flashes the primary bean with a white outline
        if self.counter % 200 == 90:
            self.primary.state = TEXTURE_STATE_IDS["WHITE"]
        elif self.counter % 200 == 0:
            self.primary.state = TEXTURE_STATE_IDS["NORMAL"]

        try:
            # Rotation
            if self.grid.input_handler.rotation:
                self.rotate()

            # Movement right or left
            if self.grid.input_handler.state == ACTION_IDS["MOVE_RIGHT"] or self.grid.input_handler.state == ACTION_IDS["MOVE_LEFT"]:
                if self.verify_placement():
                    self.column += 1 if self.grid.input_handler.state == ACTION_IDS["MOVE_RIGHT"] else -1

            # Falling
            # Holding the down key will temporarily greatly increase the fall rate used
            if self.counter % (self.fall_rate//20 if self.grid.input_handler.state == ACTION_IDS["DOWN"] else self.fall_rate) == 0:
                # If half way between two grid cells, then fall another half a grid cell you know there's nothing beneath you
                if self.row % 1 == 0.5:
                    self.row -= 0.5
                # If we hit the bottom of the grid then place there
                elif self.row == 0 or (self.rotation_state == 2 and self.row == 1):
                    self.place_beans()
                # If there is a bean below either the primary or secondary bean then place here
                elif self.grid.values[self.primary_position()-self.grid.columns] or self.grid.values[self.secondary_position()-self.grid.columns]:
                    self.place_beans()
                # Otherwise, fall down half a grid cell
                # If the down arrow is being pressed, this adds a point for every grid cell
                else:
                    self.row -= 0.5
                    if self.grid.input_handler.state == ACTION_IDS["DOWN"]:
                        self.grid.score.score += 1
                        self.grid.score.text = str(self.grid.score.score).zfill(9)
        # If we have an index error, we pad grid out with Nones
        except IndexError:
            self.grid.values += [None]*(int(self.column+self.grid.columns*(self.row+1))-len(self.grid.values))
            if self.counter % self.fall_rate == 0:
                self.row -= 0.5

    def primary_position(self):
        # Gets position from row and column. Row is truncated.
        return self.column+self.grid.columns*int(self.row)

    def secondary_position(self):
        # Secondary position is relative to primary position and rotation state
        match self.rotation_state:
            case 0:
                return self.column+self.grid.columns*int(self.row+1)
            case 1:
                return self.column+1+self.grid.columns*int(self.row)
            case 2:
                return self.column+self.grid.columns*int(self.row-1)
            case 3:
                return self.column-1+self.grid.columns*int(self.row)
            case _:
                raise ValueError("Invalid rotation state")

    def rotate(self):
        position = self.primary_position()
        # Adding 1 is clockwise, subtracting 1 is anti-clockwise, that's how rotation state is defined
        self.rotation_state += self.grid.input_handler.rotation
        self.rotation_state %= 4
        # If you try to rotate a pair of beans and that rotation would cause the bean to be inside a wall, these if statements push you away from the wall
        if self.rotation_state == 1 and (self.column == self.grid.columns-1 or self.grid.values[position+1]):
            self.column -= 1
        if self.rotation_state == 3 and (self.column == 0 or self.grid.values[position-1]):
            self.column += 1
        if (self.rotation_state == 0 or self.rotation_state == 2) and (self.row < 1 or self.grid.values[position-self.grid.columns]):
            self.row += 1

    def relative_to_falling(self, primary_or_secondary): # True for primary, False for secondary
        # Calculates if there is a bean, or the edge of the grid, to the left or to the right of the secondary or primary bean
        offset = 1 if self.grid.input_handler.state == ACTION_IDS["MOVE_RIGHT"] else -1
        position = self.primary_position() if primary_or_secondary else self.secondary_position()
        if offset == 1 and position % self.grid.columns == self.grid.columns - 1:
            return True
        if offset == -1 and position % self.grid.columns == 0:
            return True
        # or is for extra check when half way between two grids cells
        return bool(self.grid.values[position+offset]) or (False if not self.row % 1 else bool(self.grid.values[position+offset+self.grid.columns]))

    def verify_placement(self):
```

```python
        # If there's nothing next to the primary and secondary bean we're safe to move in a direction
        return not (self.relative_to_falling(True) or self.relative_to_falling(False))

    def place_beans(self):
        # While it seems strange to me, the original game this is based on awards 1 point if the down arrow is held when a bean is placed
        if self.grid.input_handler.state == ACTION_IDS["DOWN"]:
            self.grid.score.score += 1
            self.grid.score.text = str(self.grid.score.score).zfill(9)
        # Reset the bean textures and place them
        self.primary.state, self.secondary.state = 0, 0
        self.grid.place_bean(self.primary, int(self.column+self.grid.columns*self.row))
        self.grid.place_bean(self.secondary, self.secondary_position())
        # End the falling state and make the newly placed beans fall
        self.grid.state = "GRAVITY"

# InputHandler is an Entity
class InputHandler(Entity):
    def __init__(self, engine, scene, id):
        super().__init__(engine, scene, id)
        self.state = ACTION_IDS["NOTHING"]
        self.direction = 0
        self.rotation = 0

    def render(self):
        # While nothing is currently visible this could easily be used for something that displays what keys are currently being pressed
        pass

    def get_inputs(self):
        self.left = False
        self.right = False
        self.down = False
        self.A = False
        self.B = False
        self.start = False

        # These keys, register only when they are initially pressed
        for x in self.engine.events:
            if x.type == pygame.KEYDOWN:
                match x.key:
                    case pygame.K_RETURN:
                        self.state = True
                    case pygame.K_c:
                        self.A = True
                    case pygame.K_x:
                        self.B = True

        # These keys register when they are pressed, i.e. they register when held down
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            self.left = True
        if keys[pygame.K_RIGHT]:
            self.right = True
        if keys[pygame.K_DOWN]:
            self.down = True

    def update(self):
        self.get_inputs()

        # You get output from two different attributes, rotation and state, because those two things can happen simultaneously, with rotation taking priority

        # If both rotation button are pressed then they cancel out
        if self.A and self.B:
            self.rotation = 0
        elif self.A:
            self.rotation = 1
        elif self.B:
            self.rotation = -1
        else:
            self.rotation = 0

        # DAS causes the piece to repeatedly move in a direction when it is held down, so we need to record how long a direction has been held for
        # Pressing both directions is impossible on a hardward controller and cancels your DAS completely
        if self.left and self.right:
            self.direction = 0
        elif self.left:
            if self.direction > 0:
                self.direction = -1
            else:
                self.direction -= 1
        elif self.right:
            if self.direction < 0:
                self.direction = 1
            else:
                self.direction += 1
        else:
            self.direction = 0

        # Start overrides everything but I actually never got round to implementing pause
        if self.start:
            self.state = ACTION_IDS["START"]
        else:
            # In this game, moving left or right cancels movement downwards. So, not moving left or right means moving downwards or doing nothing.
            if not self.direction:
                if self.down:
                    self.state = ACTION_IDS["DOWN"]
                else:
                    self.state = ACTION_IDS["NOTHING"]
            # If direction is 1 or -1 then we just pressed the arrow down so we move left or move right
            elif self.direction == 1:
                self.state = ACTION_IDS["MOVE_RIGHT"]
            elif self.direction == -1:
                self.state = ACTION_IDS["MOVE_LEFT"]
            # If we hit the DAS amount we move again
            elif self.direction == HANDLING_SETTINGS["DAS"]:
                self.state = ACTION_IDS["MOVE_RIGHT"]
            elif self.direction == -HANDLING_SETTINGS["DAS"]:
                self.state = ACTION_IDS["MOVE_LEFT"]
            elif self.direction > HANDLING_SETTINGS["DAS"]:
                # Then you move every <ARR time>, provided the button is still held down, as letting go resets the counter to 0
                if (self.direction-HANDLING_SETTINGS["DAS"]) % HANDLING_SETTINGS["ARR"] == 0:
                    self.state = ACTION_IDS["MOVE_RIGHT"]
                else:
                    self.state = ACTION_IDS["NOTHING"]
            elif self.direction < -HANDLING_SETTINGS["DAS"]:
                if (self.direction+HANDLING_SETTINGS["DAS"]) % HANDLING_SETTINGS["ARR"] == 0:
                    self.state = ACTION_IDS["MOVE_LEFT"]
                else:
                    self.state = ACTION_IDS["NOTHING"]
            # If you're not on any of these special amounts, do nothing
            else:
                self.state = ACTION_IDS["NOTHING"]

# Score is an Entity
class Score(Entity):
    def __init__(self, engine, scene, id):
        super().__init__(engine, scene, id)
        # I didn't have time to get the actual font from the game so here is a placeholder
        self.font = self.engine.get_asset("ComicMono.ttf", font=40)
        self.score = 0
        self.text = "000000000" # Due to the ability to display calculations text has to be controlled separately

    def update(self):
        pass

    def render(self):
        text = self.font.render(self.text, True, 0xffffffff)
        self.engine.screen.blit(text, (0.4*self.engine.width, 0.5*self.engine.height))

    def output_top_scores(self):
        # Read scores from text files, split by new lines, sort them and print out the top 5
        with open("scores.txt", "r") as f:
            scores = [int(x) for x in f.read().split("\n") if x]
        sorted_scores = self.merge_sort(scores)
        MAX_SCORES_OUTPUT = 5
        print("Top scores:")
        iter_length = len(sorted_scores) if len(sorted_scores) < MAX_SCORES_OUTPUT else MAX_SCORES_OUTPUT
        for x in range(1, iter_length+1):
            print(f"{x}. {sorted_scores[-x]}")

    def dump_score(self):
        # Try writing to the already existing file
        try:
            with open("scores.txt", "a") as f:
                f.write("\n")
                f.write(str(self.score))
        # If that file doesn't exist make a new one
        except:
            with open("scores.txt", "w") as f:
                f.write(str(self.score))

    def merge_sort(self, lst):
        length = len(lst)
        if length == 1:
            return lst
        length //= 2
        left = self.merge_sort(lst[:length])
        right = self.merge_sort(lst[length:])
        out = []
        while left and right:
            out.append((left if left[0]<right[0] else right).pop(0))
        return out + left + right
```

3.7 main.py

```
from kris_engine import Engine
from exercise import ExerciseClassic
e = Engine(scene=ExerciseClassic, width=960, height=672)
```

Chapter 4

Testing

https://youtu.be/AEm3DuqjUY8
In the above video, you can verify a number of tests, including:

- Confirming that rotation works
- The falling pair is pushed away from any walls that they rotate near
- The pair does not pass through any walls or other beans
- The pair places in the correct position and falls with gravity
- All the correct textures appear
- Scoring is correct, with footage from an emulator to prove that it produces the same values
- Beans fall correctly and match in groups when falling
- Dying is implemented correctly, no matter whether dying to an odd or even number of beans
- The score board displayed in the console is in the correct order

Chapter 5

Evaluation

Ultimately, I greatly overestimated the scope of this project and my ability to complete it. I have successfully managed to make a working Puyo Puyo prototype, however I did not manage to create a suitable replacement to an emulated version of the game, nor even something that could be considered its equal. If I had the opportunity to complete the project again I would choose a project that was more focused on backend tasks and algorithmic complexity as these are things that I really enjoy programming, and ultimately I should stick to what I'm good at.