

DRMBM (1994) remake

Christopher Ashford

September 2022

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	The Engine	16
2.2.1	Why?	16
2.2.2	Engine utility methods	17
2.2.3	Object structure	19

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.

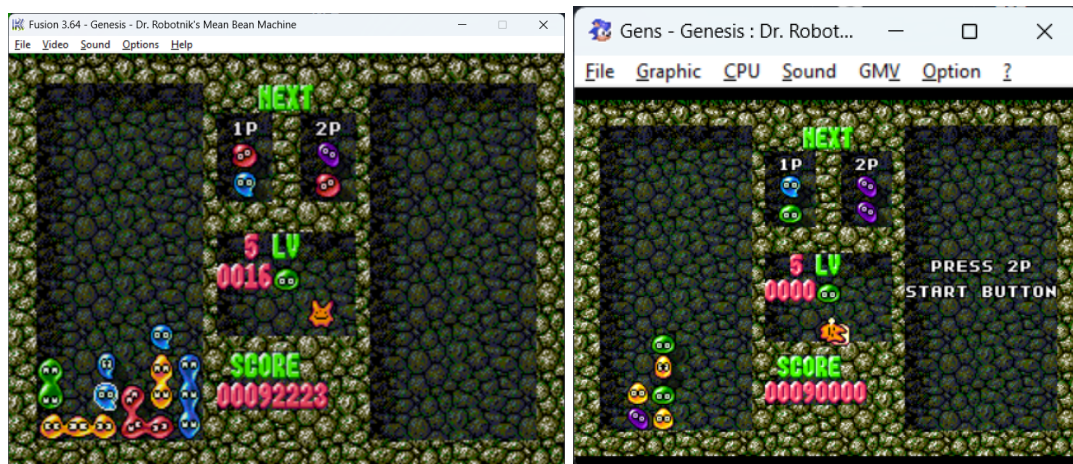
1.2.1 Emulation

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



(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

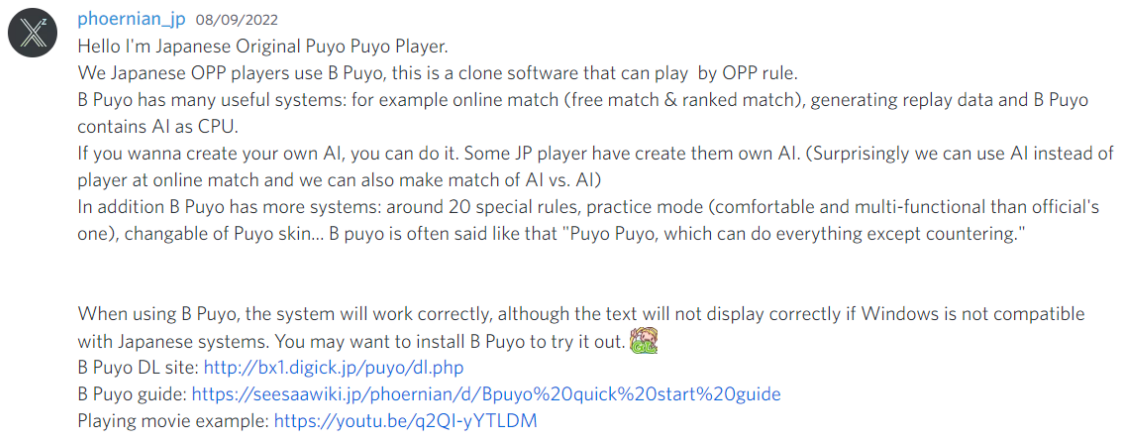


Figure 1.3: Information about B Puyo from a well known Japanese player.

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

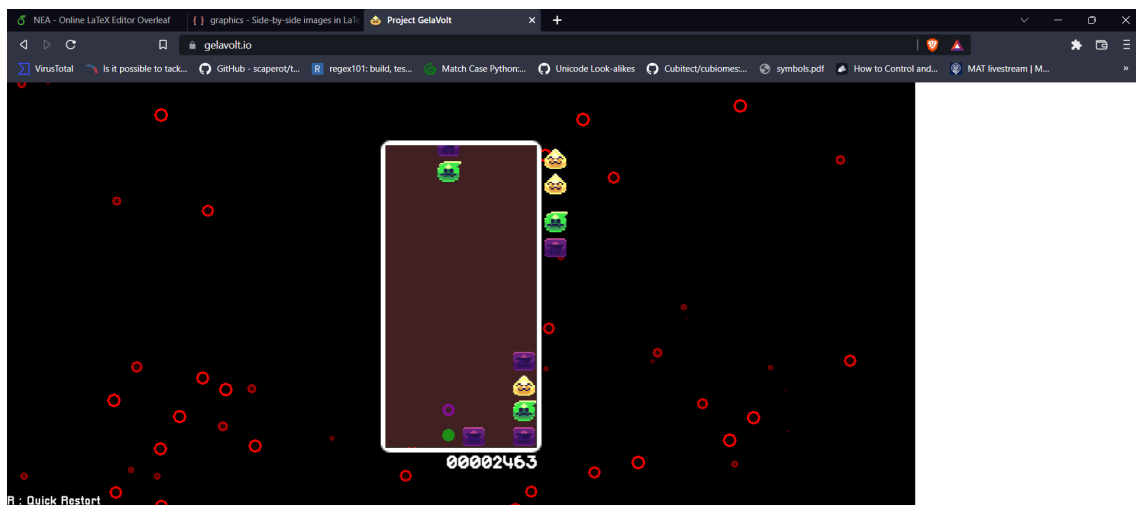


Figure 1.4: A screenshot of GelaVolt running in a chromium-based web browser.

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 contributors 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/NEA.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

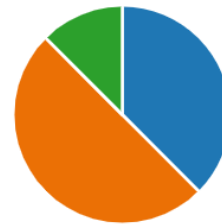


Figure 1.6: Results to the first survey question.

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 permitted.

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 permitted.

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](#)

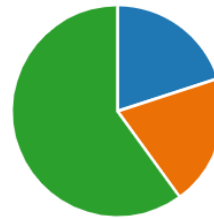
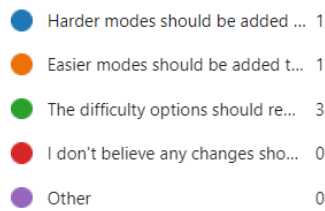


Figure 1.9: Results to the fourth survey question.

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](#)

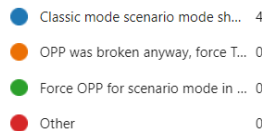


Figure 1.10: Results to the fifth survey question.

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 permitted.

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](#)

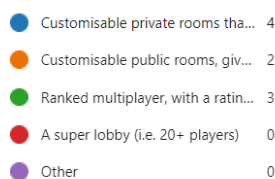


Figure 1.12: Results to the seventh survey question.

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 permitted.

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

[More Details](#)

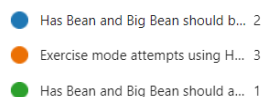


Figure 1.13: Results to the eighth survey question.

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](#)

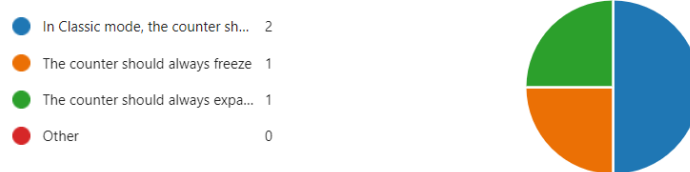


Figure 1.14: Results to the ninth survey question.

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)

B-puyo							
<div> <div>menu</div> <ul style="list-style-type: none"> 遊び方 ダウンロード 動作環境 よくある質問 IRC(チャット) Link <div>rate</div> <ul style="list-style-type: none"> レート戦って？ ユーザー登録 登録情報修正 ランキング <div>skin</div> <ul style="list-style-type: none"> スキンって？ 使い方 作り方 作り方(サウンド編) キャラスキン 仕様 <div>server</div> </div>							
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 (yathasi)	119391	54552	64751	45.7%	2022-10-07
10	2022	RICK (rick'seye)	124454	65923	58399	53.0%	2022-10-03
11	2003	ARMAVUL (armavul)	10271	4423	5841	43.1%	2022-08-28
12	1977	NUTMEG (Nutome167)	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 (aerозawa)	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.

[More Details](#)

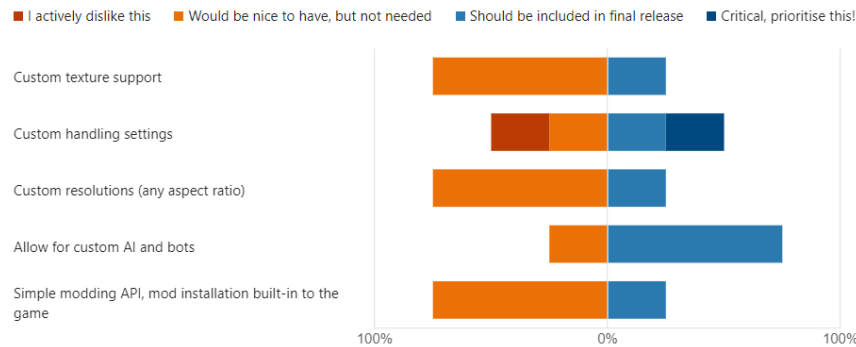


Figure 1.18: Results to the twelfth survey question.

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 initialise 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 separate 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 The Engine

2.2.1 Why?

Stacker games like Puyo Puyo at a competitive levels have a lot going on very fast at a competitive level. It is important that there is very little "input lag" (the amount of time between pressing a button and an action happening on-screen) and processing of the game's backend will need to be fast and responsive. When considering the fast-paced, accurate gameplay of a game like this, the minimum acceptable speed for the games backend to be running in terms of updating things like board state would be a rate of 600 updates per second. Furthermore, it is important that this number be a constant rate, for ease of storing replays and to ensure that when comparing the games of two competitive players that neither has an unfair advantage.

However, this presents a problem: the majority of computers are unable to process graphical output at a rate of 600 frames per second, it is simply too fast for most computers to handle. Somehow, the program needs to be capable of handling the updating of the game's state, and rendering the games graphics, at independent rates. One potential solution to this could be something similar to what a game like Minecraft does, running a server in an entirely separate process to where the game's rendering takes place, and then passing information over a local network. I would prefer to avoid this solution as network packets do not allow you to easily transfer Python objects directly, which will limit what the software is capable of and make development harder in the future. Instead, I intend to implement a multithreaded solution, which is what the engine exists to manage.

Before we continue it is important to understand how threading works in Python. Python has a feature known as the Global Interpreter Lock, or GIL for short. The GIL ensures that Python bytecode can only execute on one thread at once, ensuring that data cannot be corrupted by having multiple threads accessing it at once.

When trying to run a program across multiple CPU cores, a developer in python has three options provided with Python's standard library: the threading, multiprocessing and asyncio modules.

The threading module allows the user to run tasks across multiple CPU cores that share the same space in memory, therefore meaning it is affected by the GIL. This means that for computationally intensive tasks, threading is not faster as only thread can run bytecode at a time. However, when considering programs that require waiting, such as using the sleep function, waiting for input from the user or waiting for a response from a network request, other threads can process while this waiting is taking place. Threading is also very useful for structuring a program.

The multiprocessing module is capable of speeding up programs on multi-core machines by running multiple processes. However, this has the disadvantage of these processes not sharing memory, making it difficult to share data between them.

The asyncio module implements asynchronous development in python. Many tasks are completed out of order on a thread. It is faster for things that do not need to be ran in order and require

waiting, such as completing networking tasks.

Multiprocessing would make sharing data complicated and asyncio wouldn't be appropriate due to its out-of-order execution so my proposed solution is to use the threading module. This may seem confusing at first due to there being no major speedup, however the threading module solves the problem given above: updating and rendering can be run on separate threads.

The Engine class is a class that is designed to handle these threads. If separate threads are handling the game's backend and frontend, then the backend can run at a constant 600 updates per second using sleep statement and rendering can run at a different rate on a different thread.

There are further tasks which the engine must handle, which are described below.

2.2.2 Engine utility methods

Input and pygame events

Pygame events are unintuitive; you are required to iterate through a generator object every frame or the program will freeze, and even then the events that are returned aren't particularly user friendly. Due to their link to frames the rendering thread will have to iterate through events however they will be accessible using a public attribute of the engine. Additionally for ease of use there shall be several methods of the engine that simply return True or False for queries such as "has this key been pressed" or "is the mouse within this radius of this point/within this box". This should make entity code easier to read.

Networking

As mentioned before, asyncio is very useful for making network requests. For this reason, the module aiohttp will be used to make network requests on the main thread. Executing requests on the main thread like this prevents the program from freezing by not blocking other threads. However, executing the requests on a separate thread presents a problem of how to return the information, since the function that tries to make the request cannot simply return the information, as this would block the rest of that thread's objects from being updated.

My proposed solution is as follows: When an object needs to make a request, it shall call a method of the Engine class. The Engine class will have a counter (starting at 0 and increasing by 1 every time this subroutine is called). Calling the method will increase this counter, and append the information required to make the request (such as the type of request, the url, the data to be passed, etc.), which will have been passed in as keyword parameters, to a dictionary (that is a variable of the engine class), using the value of the counter as the key and the information required to make the request as the value. The value of the counter should then be returned to be saved by the object that made the request.

Every 0.01 seconds, the main thread will execute all the requests in the dictionary as aiohttp asynchronous requests (note that these are non-blocking), with the results to be saved in a second dictionary, with the key being the same counter value and the value being the results from the returned request (note that these will have to be parsed because the original request object must be parsed asynchronously and thus cannot be used by object that made the request, which is synchronous).

Then, when the object that wanted to make the request is later called again, it should call a second method of the Engine class, this time with the originally returned counter value as a parameter, acting as an ID for the request that was made. If the main thread has not yet completed execution of the request, then the method should return a None value, and if the request has been completed then the value stored in the dictionary should be stored in a local variable, deleted from the dictionary to save on memory consumption and finally returned to the object. Using this ID system, requests can be made by objects in a non-blocking manner, and using aiohttp will allow a large number of requests to be made in a short amount of time for tasks such as the retrieval of assets from an external server.

Sound

Pygame has two ways to play sound: the simpler mixer.music class and the advanced mixer class. mixer.music is not appropriate for this project, because it only allows you to play one sound file at a time, thus mixer is required. However, the mixer class comes with some limitations, and it's these limitations that the engine class exists to overcome.

pygame.mixer uses channels, where any channel can play one sound at a time. Pygame defaults to using 8 channels, and the number of channels can be increased or decreased at any time. I

want any object that wishes to play a sound to be able to do so with a single function call, thus the engine will have a method that accepts sound bytes as input. The engine will then create a new channel if necessary and play the sound, returning an integer that corresponds to the channel. This then allows the object to use other engine methods to control the channel, such as pausing or stopping the sound. Other useful engine sound methods include the ability to stop all playing sound.

pygame.mixer accepts .ogg files as input, to allow multiple sounds to be played as once. However, for the convenience of modders and people attempting to alter the game's assets, I believe that any audio file should be acceptable as an asset. For this reason, the program shall be distributed with an FFmpeg binary. FFmpeg is a powerful, open-source media conversion tool. In this case, it will be used by the engine to convert other audio files to .ogg. Upon initialisation, the engine will load the textures in using their original filenames and extensions. Then, a temporary folder will be created, and for any file with an extension contained within a hard-coded list of audio files such as .wav or .mp3, FFmpeg will be launched as a subprocess to convert the files to .ogg in the temporary folder, the new files will be loaded and converted to pygame.mixer.Sound objects, still referred to by their original filenames, and the temporary folder will be deleted.

Graphics

There are some complex, commonly used functions that don't make sense to re-write every time you want to use them, and thus it makes sense to simply have a method for them. An example that comes to mind would be particles: the engine could have a method where a texture, position and particle count and whether the particles should have gravity would be passed in, and an object would be created to fulfill that task, instead of being down to the programmer using the engine to implement. An exhaustive list won't be provided here as I simply don't know what will be useful, and what I'll have time for at this point.

Logging

This engine will be written in Python 3. Python 3's print function is slow, slow enough that I feel the need to directly flush to stdout for a program that is completing this many operations on a single CPU core. The engine will have two private attributes, the console log and the full log. Log functionality will require five built-in python modules: sys, os, threading, atexit and time.

The engine will have a method that can be called to append something to the log. A single parameter will be used to pass in what should be appended to the log as a string. The method will then convert this string into a standard format: the threading module will be used to find the ID of the thread that the function was called from and the time function will be used to give the log entry a timestamp, representing the number of seconds since the program started. Then a coloured version of the log (using ANSI escape codes) will be appended to the console log and a plaintext version to the full log. An example log entry could look like this:

Welcome to Kris's Engine!

Log initialised at 1668108523.8011909

[Main thread at 0.1000968]

Warning! Failed to convert asset "purple13.png" to Pygame surface object. Corrupted file? Purging from textures.

[Update thread at 39.7935968] Error! ZeroDivisionError updating Entity of type AI, skipping.

[Render thread at 51.615771] Error! BaseException render Entity of type Bean, skipping.

[Update thread at 161.0327687] Warning! Last update took 2.3ms longer than allocated, the game is lagging!

Upon engine initialisation, first the code "os.engine("")" must be ran. This will enable ANSI escape code colours on Windows, and do nothing on other platforms. A function shall exist that simply dumps the full log to a log text file, and this function should be queued up using atexit. This module will ensure that the function always executes when the program terminates. Then, a thread will be started: this thread shall every 0.1 seconds use sys to flush the contents of the console log to stdout (the console) if it's not empty, and then empty it.

Global error handling

We can use `sys.excepthook` to create a function that globally handles errors by reporting the error to both the user and a remote web server without the program crashing.

2.2.3 Object structure

The engine is a complex object oriented model that work on the basis of all composed classed inherited from two classes: scenes and entities. Since I'll probably end up naming the engine Kris's Engine, when importing `kris_engine.main` we now have `kris_engine.main.Engine`, `kris_engine.main.Scene` and `kris_engine.main.Entity`.

Entity

An Entity represents a building block, an element of what should be going on. In it's simplest capacity, an entity is something you want to draw to the screen, however an entity doesn't have to be visible and can be used only for processing in order to keep the code's structure cleaner.

For an Entity object to be valid, it must have at least two methods: `update` and `render`. If an entity does not have this, the engine will error, thus all valid entities must inherit from `kris_engine.main.Entity`. `kris_engine.main.Entity` will contain the two methods `update` and `render`, which both take no arguments other than themselves (`self`), as all information they need can be gotten from the engine (it of course makes sense for the entity object to contain the engine object within itself as an attribute). `Render` will draw to the screen a message detailing that this entity hasn't been correctly overwritten, a similar message will be appended to the log upon initialisation, and the `update` function will simply pass and not do anything. Inheriting from this class ensures that an object has to have the `update` and `render` methods, giving it less chance of erroring the engine. It is easy to verify whether an object has inherited from this class using the `"__mro__"` attribute. The engine simply has to check whether the default entity class is contained within the `__mro__` tuple and if it isn't, an object that is supposed to be an entity should be discarded and an error message logged.

Scenes override the entities that are loaded, but what if an entity should persist across scenes? Say, a message feature that should always be accessible in game or a custom music player add-on. For this reason all entities should have a `"persist"` attribute. This value will be a boolean `True` or `False`, and will default to `False`. `False` means the entity object should be destroyed when loading in to a new scene and `True` means the entity should not be destroyed. Persistent objects are expected to be rendered first relative to all other objects, and objects from a new scene shall be appended after persistent entities in the list of entities.

It may be useful to be able to uniquely identify an entity, especially for logging purposes. Thus, upon initialisation, an entity shall be assigned a public `"id"` attribute by the engine. This shall be unique and shall be obtained using a counter from the engine class that is increased every time an entity is initialised. Additionally, so that entities may interact with other entities, a public attribute of the engine, `"entities"`, will house a dictionary where the key is the public id of the entity and the

Any textures that are accessed by an entity should use a method of the engine class, `get_texture`, passing in a string referring to the name of the texture file (texture file names must be unique), which will then return the pygame surface object for the texture. There will be a `deload_texture` method for if an entity knows that a texture is no longer required, such as during a cutscene.

Any other attributes are the responsibility of the developer using the engine to assign and manage.

Scene

Separate scenes should be split into separate python files, which should make it easy to find, read, edit and manage scenes.

A valid scene should meet the following:

- A scene should be contained within a python script file
- The first line of the file should be `"from kris_engine import Scene"`
- The file should contain a single class
- The last thing this class inherited (as shown by the `__mro__` attribute) should be the `kris_engine.main.Scene` class.

- Additionally, the engine should check for the string `__mro__` within the scene's code, to prevent potential modification.

All of these verification steps are justifiable.

- Using a python file to contain a single scene makes it easy to add new scenes without modifying existing files.
- Any engine project will contain multiple python files. Using the first line to import the scene class allows the engine to quickly identify which files should be considered scene classes and which should be ignored. If a file that isn't a scene wants to import the scene class, this can easily be bypassed by importing other modules first or adding a comment.
- By having a scene not contain the entities that it will create within its own file and instead having to import them, entities that are used in multiple scene files will not be written out multiple times.
- The `__mro__` string prevents the engine from running arbitrary code from a file that is not a valid scene class. Additionally we can check if the attribute is referenced within the scene itself, a basic way to check if the value may have been modified. This is a simple way to prevent the user from running a potentially invalid or malicious scene file.

It is the job of a scene file to define which entities should be present in a scene and provide useful information about things such as the framerate and update rates and the order entities should be rendered.

A developer can set two optional attributes, `framerate` and `updaterate`. These are optional because if not given the engine will use hardcoded defaults.

Depending on the scene a loading screen may be useful. So, a scene will have two attributes, `"loading_screen"` and `"loading_screen_entity"`. By default `loading_screen` will be `False` but may be set to `True` by a scene. By default `loading_screen_entity` will refer to a built-in entity file that exists as a default loading screen. If set to any other value that entity will be loaded. There will be an attribute of the engine that the entity can access to monitor the progress of loading in order to allow a loading bar to exist.

While this is happening of course, we want to be loading something. The `get_texture` method mentioned earlier will either create a pygame surface object and cache it or return it if it's already been cached, so another convenient method of the engine class, `load_textures`, will allow a scene during its initialisation to pass in a list of strings and load all of the required textures.

It is recommended that a scene

NOT FINISHED NOT FINISHED NOT FINISHED NOT FINISHED