**Project Documentation for Component 03/04**

of OCR Computer Science – H446

**Contents**

[Analysis 3](#_Toc102941237)

[Outline of the Project 3](#_Toc102941238)

[Stakeholders 3](#_Toc102941239)

[Identification of the Target Platform 3](#_Toc102941240)

[Explanation of the User’s Needs 3](#_Toc102941241)

[Computational Methods and Approach 4](#_Toc102941242)

[Identification and Explanation of Computational Methods 4](#_Toc102941243)

[Research and justification of proposed features 5](#_Toc102941244)

[X-Moto 5](#_Toc102941245)

[Angry Birds 5](#_Toc102941246)

[Littlebigplanet 6](#_Toc102941247)

[Features of the proposed solution 8](#_Toc102941248)

[Identification of limitations and scope 8](#_Toc102941249)

[Meeting with the stakeholders 8](#_Toc102941250)

[Requirements 10](#_Toc102941251)

[Success Criteria 10](#_Toc102941252)

[Design 11](#_Toc102941253)

[Structure of the solution 11](#_Toc102941254)

[Decomposition of the Problem 12](#_Toc102941255)

[Explanation of each of the modules, procedures and functions required 12](#_Toc102941256)

[A Justification of the Approach Taken 16](#_Toc102941257)

[Key Variables and Structures 16](#_Toc102941258)

[Explanation and justification of the data structures 16](#_Toc102941259)

[Algorithms 18](#_Toc102941260)

[Physics Simulation 18](#_Toc102941261)

[Usability Features 22](#_Toc102941262)

[Explanation and Justification of the User Interface 22](#_Toc102941263)

[Description and Justification of the Usability Features 23](#_Toc102941264)

[Post-Development Test Data 23](#_Toc102941265)

[Description and Justification of the Validation Required 25](#_Toc102941266)

[Test Data for Text Input 25](#_Toc102941267)

[Development 27](#_Toc102941268)

[Physics Simulation 27](#_Toc102941269)

[Version 1 27](#_Toc102941270)

[Version 2 35](#_Toc102941271)

[Version 3 41](#_Toc102941272)

[Graphics 44](#_Toc102941273)

[Version 1 44](#_Toc102941274)

[Version 2 50](#_Toc102941275)

[Version 3 55](#_Toc102941276)

[Level Design 63](#_Toc102941277)

[Version 1 63](#_Toc102941278)

[Version 2 66](#_Toc102941279)

[Version 3 71](#_Toc102941280)

[Menu 75](#_Toc102941281)

[Version 1 75](#_Toc102941282)

[Version 2 80](#_Toc102941283)

[Version 3 85](#_Toc102941284)

[Evaluation 97](#_Toc102941285)

[Testing to Inform Evaluation 97](#_Toc102941286)

[Stakeholder Testing 98](#_Toc102941287)

[Meeting the Success Criteria 99](#_Toc102941288)

[Physics Simulation 99](#_Toc102941289)

[Menu 102](#_Toc102941290)

[Level Design 104](#_Toc102941291)

[Graphics 105](#_Toc102941292)

[Usability Features 106](#_Toc102941293)

[Limitations 110](#_Toc102941294)

[Pushing PhysObjects 110](#_Toc102941295)

[Moments 111](#_Toc102941296)

[Performance 111](#_Toc102941297)

[Maintenance 112](#_Toc102941298)

[Adding New Features 112](#_Toc102941299)

[Adding New Levels 113](#_Toc102941300)

# Analysis

## Outline of the Project

I intend to produce a platforming game featuring mechanics that model concepts found in physics. The software will simulate the crucial aspects of physics and mechanics while providing a satisfying gaming experience.  I have in mind a KS3-4 user base, as this age group engages with science at school and needs to experience scientific concepts in a variety of contexts. Various physical phenomena will be included within the core features of the game - for example, all objects will carry a mass and be affected by force accordingly, therefore acquiring resultant accelerations, velocities, and momenta.

Although the gameplay will not make overt these features of physics, they will, nevertheless, be present.  In order to bring them out so that the player notices them, there will be various game mechanics attuned in a way that they take advantage of them, such as being able to push and throw objects to solve puzzles and having the player’s movement directly tied into these mechanics. While the game itself is not purposefully developed with the intent of educating the end-user, its mechanics feature opportunities to educate the player, as it makes use of accurate data on forces, mass, velocity, and momentum. To make this overt, a “learning”/debug mode will be implemented into the game, allowing the user to observe the underlying physical values in real-time

## Stakeholders

In order to ensure the accurate representation of the mechanics involved, my first stakeholder is Mr Abdul Bahir; as a physics teacher and expert, I believe his knowledgeable input will prove invaluable when evaluating the objective quality of the game’s mechanics. Despite his lack of computer expertise, he could provide generalised advice which could then be coded as features of the game.

In addition, my fellow student Alex Buhai (in Y10) will be involved during the prototyping and testing phase.  As an avid game player, and as someone who is knowledgeable about game mechanics and development, I believe his input will also be valuable concerning the entertainment and playability of the game. Moreover, he falls within the intended target audience of KS3-KS4 students, so his feedback will provide valuable insight into the features desired by that group.

## Identification of the Target Platform

The target platform is a PC, ideally running windows 10 with Python 3.7+ and the Pygame and Numpy modules installed.

## Explanation of the User’s Needs

The users will be playing the game in hopes of a fun, smooth, enjoyable experience with realistic (or, at least, realistic by appearance) game physics and mechanics which take advantage of those features. As a result, the effectiveness and consistency of game mechanics will be of top priority to ensure an entertaining experience for the end-user, while features such as a debug-like analysis mode will be added later in development to imbue educational value.

## Computational Methods and Approach

### Identification and Explanation of Computational Methods

A physics-based game is a perfect fit for the application of certain computational methods. For example, the concept can be clearly decomposed into several core components by implementing the ideas of force, velocity, acceleration etc. These in-turn serve as foundation for every other feature in the game and complement future improvement.

Abstraction will be widely used throughout the project due to sheer potential for complexity in the game. Several aspects of physics will have to be omitted in favour of time and after considering their diminishing improvement to the gameplay. For example, variations in air density and composition, specific friction values between different materials and exact surface areas for drag calculations etc.

Heuristic methods will be used in some aspects of development to reduce the complexity and time demands of the project. For example, ray-casting (shooting lines in a direction and gathering data about it e.g., objects hit, distance etc.) would have to be coded specifically for the game using a loop which traces a straight line in a direction using y=mx+c. This may not be the most efficient method as many pixels will be checked “unnecessarily”, but it would not affect the performance drastically.

Data mining could be done post-development using the scoring system. This could be analysed for various patterns in gameplay for the target audience, and perhaps to find a correlation between approaches used and the success rate of players.

On the contrary, divide and conquer is not an applicable computational method as none of the datasets used in the game will be sorted with the exception of the leaderboards.

## Research and justification of proposed features

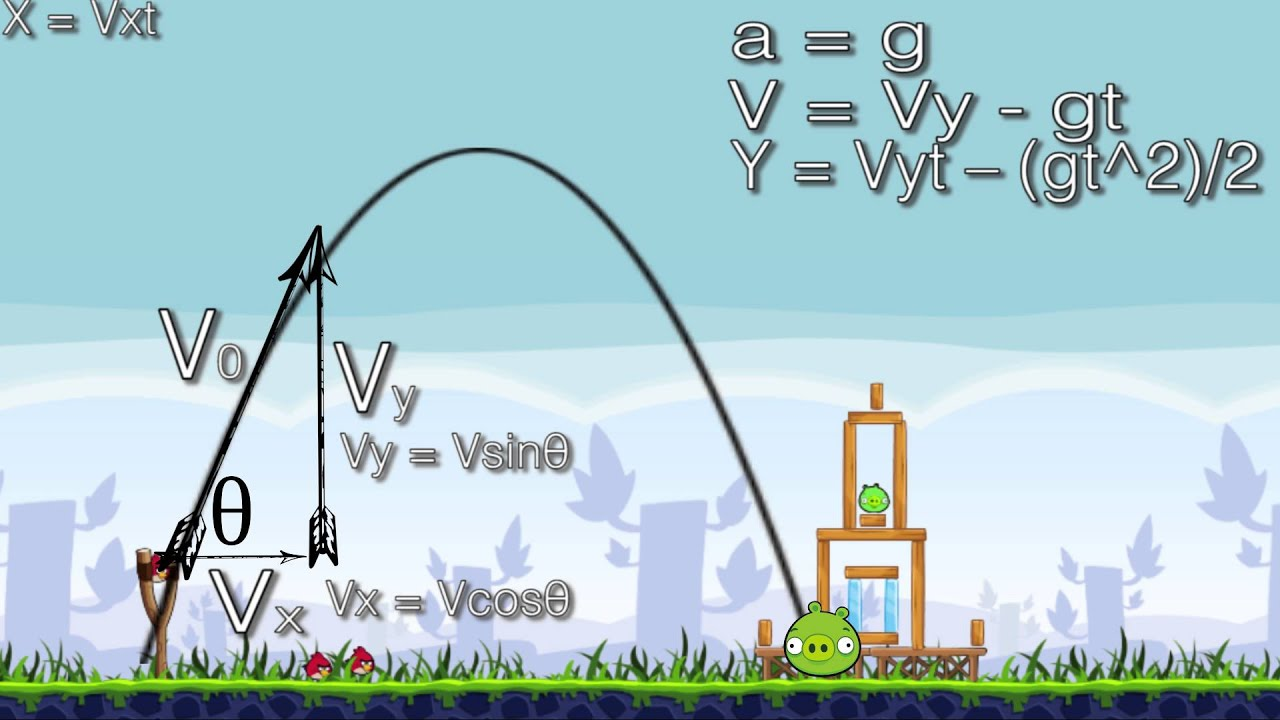
In order to gather ideas and knowledge in relation to the development of the game I will conduct research into various similar projects which possess high levels of mechanical detail.

### X-Moto

The game “X-Moto” is an open-source physics-based driving platformer which exemplifies aspects of physics such as momentum, forces, and moments etc. which are essential to the success of the player. For example, the player is in charge of controlling the driving force and direction of the bike, braking, and most importantly the rotation/weight bias of the bike itself. This is used to navigate various platforms and obstacles in the game – all movement has its effects fully simulated through the in-built physics engine. Rotating the bike causes the player to gain angular momentum, which quickly becomes a detriment in many scenarios, as any touching of the player’s body against other objects leads to failure.

In order to transfer these features to my game in a reasonable manner, certain unnecessary details will be omitted from the final physics engine. For example, ragdoll physics will not be included in the game due to the engine and time limitations of the project. Instead, most objects could be modelled as per their sprites and not include several separate joints, allowing for easier incorporation of forces and moments acting on the objects.

### Angry Birds

The popular and addictive game Angry Birds lends much of its success to a satisfying yet challenging gameplay loop. Players launch birds at chosen trajectories and power with the goal of destroying as much of the level as possible while also attempting to hit all the pigs on the level. The player cannot see the target from the launch point, and so must undergo a trial-and-error-like process in order to find the optimal conditions for launching the projectile. This increases the replayability of a level and creates fun physics-based challenges with multiple ways to succeed.

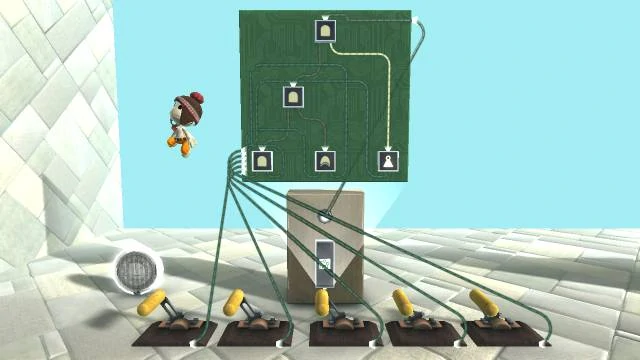
In my project, I would like to implement projectile motion indirectly via the existing engine physics and use it for puzzle solving. I would also like to implement the idea of colliding with objects and moving them through momentum, similarly to how angry birds collide realistically with the wooden blocks in the game, and pigs will also move depending on the blocks around them.

### Littlebigplanet

Littlebigplanet is a puzzle platformer game which shares many features with my intended design. First and foremost, its perspective mimics that of a 2D side scroller (such as the one I will be making) but is actually a 3D environment where the player’s camera is rotationally locked to a single axis. This makes for interesting gameplay, as the game can take advantage of the fidelity of 3D game engine physics to construct its various puzzles and events.



*In this image, the player has constructed a pendulum-like structure, consisting of a mass hung by a “winch” object, which possesses attributes such as “Length”, “Timing” and “Strength”. These properties factor into the movement of the object, and my game will share such properties for some objects, only they will be limited to the internal environment.*

In addition, Littlebigplanet inexplicitly exposes the player to computational thinking and logic through the use of logic gates and circuits which can manipulate and control other objects by equating the values of these systems. Inputs are represented as power/energy sent into the system, e.g., via a digital input such as a button, or analogue input such as a battery.

*Notice the “NOT” gate in the bottom right of the circuit board outputting power to the “AND” gate at the top of the circuit despite not having any input value.*

Similarly in my game, I would like to implement puzzles which take advantage of components of physics without directly questioning the player. Implementing the concept with subtlety allows me to maintain a non-educational theme for the overall game, while still indirectly teaching the player basic physics concepts and forcing them to use this knowledge to their advantage. Players may have to create an imbalance of weight on an unstable object in order to alter the environment and solve a platforming puzzle. For instance, this could be a plank on the edge of a surface, used to bridge the gap between two platforms or launch an object across it.

## Features of the proposed solution

### Identification of limitations and scope

Due to the limited time and resources being allocated to the game, not all features can be implemented to their full extent, and many features found in widely used physics engines will have to be omitted. In order to concentrate effort on the most substantial and desired aspects, interviews will be conducted with the selected stakeholders.

### Meeting with the stakeholders

Mr Abdul Bahir:

Q. What core features do you expect to see in the physics engine?

I expect to see every object in the game respond and move at least somewhat realistically according to the environment. Objects should experience air resistance, friction, and movement should be calculated by resolving all acting forces. In addition, objects should properly collide, and momentum should be traded correctly between them. Moments should also be implemented so objects can rotate.

Q. How could the game be designed to be of educational value to physics students?

The most practical way I can come up with is to implement live free-body diagrams for objects in the game. Give the player the option to view forces acting on different objects and their physics values in real-time: e.g., velocity, resultant force, acceleration. In addition, final statistics could be shown when completing a level to tell the student how efficient their solution was, and how their actions influenced the way it played out.

Q. As a coder yourself, how would you go about implementing these features?

My first thought would probably be to decompose and split the elements of the game into separate classes and objects. For instance, a physics-based object would be of a “Physics Object” class while something like a static map object may be its own class - unaffected by the physics engine.

Q. What do you believe may be too difficult or unnecessary to implement into the game?

Obviously, I don’t expect the game to feature particle-level physics simulation - this would be extremely unnecessary and difficult. Also, simulating field-based physics - be them gravitational, electric, magnetic etcetera, would also be very hard to implement to complete accuracy. Instead, if those mechanics are necessary in the game, they could be abstracted into their most fundamental form, simply affecting other objects to a degree in which they could trick the player into thinking they are being fully simulated. Fluid dynamics would also be a feature that would probably exceed the scope of the game and may be entirely unnecessary, but fluids could still be present as environmental hazards - perhaps oil could be implemented as a modifier to the movement of the player, slowing them down due to high viscosity/drag.

Alex Buhai:

Q. What do you expect to see in the game, considering its physics-based nature?

Well, I expect to see forces and stuff, and things should move naturally like they would in real life. Maybe to prove things are moving accurately you should show the forces and things acting on all the objects.

Q. How do you think the game could be made more appealing or fun to play?

Maybe you could implement a lot of levels, and make sure there’s variation in between all of them. I won’t want to play/replay a game if I’m just doing the same thing over and over again every time. Also, make sure you utilise your physics engine to its fullest extent, it’s the main appeal of your game over others and the game should reflect the time you spent on it.

Q. How could the game be more accessible to all audiences?

The game should have intuitive controls and mechanics. They shouldn’t be overwhelmed with mechanics from the beginning, instead they should be eased into the game’s playstyle and introduced to new features as they proceed.

Q. How would you enhance the replayability of the game?

I think the best feature to encourage replayability would be a leader board or some sort of scoring system. Players could aim to optimize their times through their understanding of the game’s mechanics and compete with each-other in beating their scores on challenging levels.

After considering the various proposed features from the stakeholders, it is clear that several features will be implemented and omitted. Particularly, Mr Dias’ proposed implementation of moments will be too difficult in Python and easily avoidable by redesigning the game around this omission. However, most of the other features mentioned by both him and Alex Buhai will be present in the final solution.

## Requirements

### Success Criteria

|  |  |  |  |
| --- | --- | --- | --- |
| **Strand** | **Requirement** | | **Evidence** |
| Physics Simulation | I | Motion must be simulated as a result of the application of force. Forces should be evaluated and produce resultant motion in the form of acceleration and velocity. | Video of object moving over a given time, along with the values for the motion. |
| II | Environmental effects will be simulated on the player. Specific examples include friction and air resistance but may include many forms of drag or drive that are applied to the player as a result of the properties of their environment. | Screenshot of an object experiencing drag forces. |
| III | Collisions with both the environment and other objects will be simulated. Momentum will be transferred between objects according to various factors and objects’ motion will be adjusted accordingly. | Video of objects colliding with each-other and walls. |
| Menu | IV | Basic menu system must be implemented to allow the player to play, view leaderboards, level select etc. | Screenshot of the menu |
| V | A login system is needed to save the score of the player to a desired username and track progress. | Screenshot of the login screen |
| VI | A leaderboard is required to track top scores and allow players to compare their scores to others. | Screenshot of the leaderboard |
| Level Design | VII | Levels must be stored using external data structures (likely CSV files) and loaded in through a suitable algorithm | Screenshot of a stored level structure. |
| VIII | Levels must contain set objectives to be met in order to determine completion. These objectives would involve both player and object interaction. | Video of a level being completed. |
| IX | Environmental hazards will be implemented into the game and can be placed on any level | Video of a level being completed with environmental hazards. |
| Graphics | X | A Camera function should be in the game to allow traversal around a level and prevent the game being limited only to the size of the screen. | Video of a level being played. |
| XI | Animations will play to show the ship accelerating, dust kicking up from collisions, other environmental effects etc. | Video of a level being played. |
| XII | A HUD will consist of a fuel meter and, if enabled, an overlay visually displaying the forces and physics values acting on different objects in the simulation. | Video of a level being played. |

# Design

## Structure of the solution

An Overview of the Structure of the Solution (Systems Diagram)

## Decomposition of the Problem

### Explanation of each of the modules, procedures and functions required

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Classes and Methods | | | | | |
| **Type** | **Name** | **Parameters** | | **Description** | **Return** |
| **Type** | **Name** |
| **Class** | Material | **Float** | Static  Kinetic | Defines “Material” objects. Stores static and kinetic frictional coefficients which can be assigned to all the entities in the world. | n/a |
| **Class** | Vec2 | **Float** | X  y | Represents a 2D vector with x and y components. | n/a |
| **Method**  **[Vec2]**  **[Override]** | add | **Vec2**  **Float** | N  n | Adds a vector by its components correspondingly or a constant to both components. | **Vec2**  **Object** |
| **Method**  **[Vec2]**  **[Override]** | sub | **Vec2**  **Float** | N  n | Same as add but subtracts the parameter instead. | **Vec2**  **Object** |
| **Method**  **[Vec2]**  **[Override]** | eq | **Vec2** | n | Checks if two vectors are equal according to their components. | **Bool** |
| **Method**  **[Vec2]**  **[Override]** | mul | **Vec2**  **Float** | n  n | Same as add but multiplies by the parameter. | **Vec2**  **Object** |
| **Method**  **[Vec2]**  **[Override]** | truediv | **Float** | n | Performs float division on each component correspondingly by the parameter. | **Vec2**  **Object** |
| **Method**  **[Vec2]**  **[Override]** | floordiv | **Float** | n | Same as truediv but removes any fractional component. | **Vec2**  **Object** |
| **Method**  **[Vec2]**  **[Override]** | pow | **Float** | n | Exponentiates each component to the given parameter. | **Vec2**  **Object** |
| **Method**  **[Vec2]**  **[Override]** | str | **n/a** | **n/a** | Represents a vector as a string in the form: “(x, y)” | **String** |
| **Method**  **[Vec2]**  **[Override]** | getSqrMag | **n/a** | **n/a** | Calculates the squared magnitude of a vector with Pythagoras. | **Float** |
| **Method**  **[Vec2]**  **[Override]** | getMag | **n/a** | **n/a** | Calculates the true magnitude of a vector – involves square roots. | **Float** |
| **Method**  **[Vec2]**  **[Override]** | getNormalized | **n/a** | **n/a** | Returns the normalized version of the vector (magnitude of 1) | **Vec2**  **Object** |
| **Method**  **[Vec2]**  **[Override]** | Inverse | **n/a** | **n/a** | Returns a vector with the same magnitude and opposite direction. | **Vec2**  **Object** |
| **Class**  **{Vec2}** | Force | **Float**  **String**  **PhysObj** | x  y  name  source | Represents a 2D force with x and y components, a name, and a source. | **Force**  **Object** |
| **Class** | ForceManager | **PhysObj** | parent | Manages the forces acting on an object; holds a list of all the forces belonging to its parent. | **n/a** |
| **Method**  **[ForceMan]** | GetForce | **PhysObj**  **String** | source  name | Returns the force in question from the list of forces contained in the Force Manager. | **Force**  **Object** |
| **Method**  **[ForceMan]** | GetAnyForce | **String** | name | Returns a list of all the forces with the given name, regardless of their source. | **List** |
| **Method**  **[ForceMan]** | AddForce | **PhysObj**  **Vec2**  **String** | source  \*args  name | Adds a force to the Force Manager’s list.   * If the Force’s values round to 0, removes the Force if it already exists. If the Force’ * If the Force doesn’t already exist, adds the Force. * If the Force does exist, replaces it. | **n/a** |
| **Method**  **[ForceMan]** | RemoveForce | **PhysObj**  **String**  **/**  **Force** | source  name  /  \*args | Removes a force from the Force Manager identified either by its source and name or the exact object reference. | **n/a** |
| **Method**  **[ForceMan]** | GetResultantNOF | **n/a** | n/a | Returns the resultant force without the friction and air resistance. | **Vec2**  **Object** |
| **Method**  **[ForceMan]** | Update | **List**  **Float** | colliders  dt | Updates the behaviour of forces and adds/removes environmental forces.   * Removes redundant and un-applicable forces. * Handles the weight force. * Handles the Normal force and collisions. * Handles Air Resistance * Handles Friction * Updates resultant force | **n/a** |
| **Class** | WorldCollider | **Rect**  **String** | rect  material | Represents a rectangle in the world with the frictional coefficients of the material specified. | **WorldCollider**  **Object** |
| **Class** | PhysObject | **Vec2**  **Image**  **Float** | pos  image  mass  Cd  COR | Represents a physics-simulated object; stores its position, mass, image, angle, drag coefficient, coefficient of restitution, velocity, acceleration, and a force manager. | **Physics**  **Object** |
| **Method**  **[PhysObj]** | Draw | **Surface** | surface | Blits the image of the physobject to the screen and rotates it accordingly. | **n/a** |
| **Method**  **[PhysObj]** | Rotate | **Float**  **List** | scale  colliders | As long as the object is able to rotate, it rotates by 1 degree every call. The rotation is converted into radians and applied to the image. | **n/a** |
| **Method**  **[PhysObj]** | Update | **List**  **Float** | colliders  dt | Handles physical movement according to forces.   * Rounds velocity to 0 if it’s close. * Adjusts | **n/a** |
| **Class**  **{PhysObj}** | Player | **Vec2**  **Image**  **Float** | pos  image  mass | Represents the player’s physics object. | **Player**  **Object** |
| **Class** | Collision | **PhysObj** | object  collider | Represents and handles a state of collision between two overlapping/touching objects. | **Collision**  **Object** |
| **Method**  **[Collision]**  **(Static)** | pushing | **PhysObj** | obj1  obj2 | Traces the direction of velocity and resultant force of obj1 to return whether it is “pushing” obj2. | **n/a** |
| **Method**  **[Collision]** | Resolve | **n/a** | **n/a** | Resolves a collision by transferring momentum accordingly to each object. | **n/a** |
| **Class** | CollisionHandler | **n/a** | **n/a** | Manages all the collisions occurring in the world. | **CollisionHandler**  **Object** |
| **Method**  **[CollisionHandler]** | ColScan | **List** | world | Scans all objects in the world and creates collision objects representing those which are in the state. | **n/a** |
| **Method**  **[CollisionHandler]** | Update | **List** | world | Conducts a collision scan, then resolves and manages all the generated collisions. | **n/a** |
| **Function** | trace | **Vec2**  **PhysObj**  **Bool** | source  dir  target  detailed | Detects whether a target object is hit along a traced line. | **Bool** |
| **Function** | lINTerp | **Float** | lb  ub  frac | Returns the linearly interpolated value of the input fraction from the lower-bound to the upper-bound. | **Int** |
| **Function** | getCameraTrack | **Vec2**  **Int** | Pos  lPos  lwidth  lheight | Returns the new level position required to track the player’s movement with the camera. | **List** |
| **Class** | Particle | **Vec2**  **Float**  **Bool** | Pos  Velocity  Timer  weightless | Represents a particle object | **Particle**  **Object** |
| **Method**  **[Particle]** | Update | **Float** | dt | Updates the position and velocities of the particles, as well as tracking their elapsed time/lifespan. | **n/a** |
| **Class**  **{Particle}** | EngineParticle | **Vec2**  **Float** | pos  velocity  timer | Represents an engine particle. Engine particles are spawned at the “bottom” of the ship using trigonometry. | **EngineParticle**  **Object** |
| **Method**  **[EngineParticle]** | Update | **Float** | dt | Updates as a regular particle, and linearly interpolates the colour value of the engine according to how long it has existed. | **n/a** |
| **Method**  **[EngineParticle]** | Draw | **Surface** | surface | Draws a circle on the screen according to the particle’s position, colour, and transparency. | **n/a** |
| **Class** | ParticleHandler |  |  | Manages all the particles in the scene, including updating and deleting them accordingly. | **ParticleHandler**  **Object** |
| **Method**  **[ParticleHandler]** | Update | **Float** | dt | Runs the update and draw functions on each existing particle, as well as deleting those which have expired. | **n/a** |
| **Function** | level\_load | **String** | level | Loads a level from the level files. Level data is stored in CSVs representing lists of different objects and their initial values. The function reads them and assembles local data structures accordingly for world, game, player, and objectives, as well as the background image of the level. | **List** |

## A Justification of the Approach Taken

As is necessary with physics simulation, the sheer density of information needed to be tracked for each game element quickly becomes extremely difficult to organise without an Object-Oriented approach. By adopting a heavy use of OOP techniques, the tracking and adjustment of hundreds of sets of data becomes far more manageable and streamlined.

Physical Game objects such as PhysObjects and WorldColliders are represented in OOP with many attributes tracking their image, position, velocity etc. The objects themselves have methods which allow them to passively update their values according to the physics rules, which lays the foundations for other features. By isolating and polishing fundamental behaviours of a physics-compliant object such as the changes in physics values according to the forces acting on it and its position in the world etc., future features can be implemented around them without having to worry about errors in the most basic components. As well as this, different objects can inherit those behaviours without having to need recoding.

Other classes include those which manage and handle other game objects. For example, ForceManagers and CollisionHandlers. These objects manage other objects by ensuring they update properly each frame, and whether they should currently exist etc. This ensures encapsulation and improves the organisation of the code, making it much easier to develop in the future; developments can be conducted on a specific area of the program, without having to worry about other parts being affected or broken because of them.

## Key Variables and Structures

### Explanation and justification of the data structures

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| **Constant** | DEBUG | Determines whether debug components will be rendered and printed while the game runs. |
| **Constant** | BLUE, BLACK, WHITE, RED… | Tuples representing the RGB values of different colours. |
| **Constant** | OPTIMUS, OPTIMUS\_BOLD, FUTURE\_LIGHT | Pygame font objects used for text rendering. |
| **Constant** | GRAVITY, AIR\_DENSITY, PLAYER\_DRAG\_COEFFICIENT… | Various physics constants used for calculations in the physics engine. |
| **Constant** | METRE | A constant value representing the number of pixels equal to a “metre” in the game world. The value is produced using the height of the player as a baseline of 1.7m. |
| **Constant** | RAD | Stores the conversion ratio from degrees to radians. (pi/180) |
| **Attribute**  **{MATERIAL}** | Static, kinetic | Stores static and kinetic friction coefficients for a material |
| **Attribute**  **{PhysObject}** | Pos, velocity, acceleration, weightless, rForce, momentum, mass, Cd, COR, forces | Stores different physics data required in calculations for a physobject. The attribute “forces” is a reference to the PhysObjects “ForceManager” child. |
| **Attribute**  **{PhysObject}** | Angle, AngleDir | The angle in degrees and the angle represented in normalized vector form, where 0o is (0, 0) and 90o is (1,0) etc. |
| **Attribute**  **{PhysObject}** | Image\_clean, halfwidth, halfheight, rect | Various attributes which determine the image and collider properties of the PhysObject. Halfwidth and halfheight are used for locating the bottom of the ship when spawning particles. |
| **Attribute**  **{Collision}** | Object, collider, resolved | Stores the two objects involved in the collision, and whether the collision has been resolved in order to prevent recalculation of the resultant motion. |
| **Attribute**  **{CollisionHandler}** | collisions | A list of all the collisions in the game world. |
| **Attribute**  **{Particle}** | Pos, velocity, acceleration weightless, rect | Stores motion and physics data for each particle and their rects. |
| **Attribute**  **{Particle}** | Timer, elapsed | A timer is passed representing the time a particle should exist for. Elapsed tracks how much time has passed since the particle was created. |
| **Attribute**  **{ParticleHandler}** | particles | A list of all the particle objects in the game world. |
| **Attribute**  **{Force}** | Source, name | Source is a reference to the object of origin of the force; name is used in conjunction with source to differentiate between different forces. |
| **Attribute**  **{ForceManager}** | parent | A reference to the parent object the force manager belongs to. |
| **Attribute**  **{ForceManager}** | Forces, rForce | A reference to the parent object it belongs to, and a value for the sum of all forces (resultant force). |
| **Attribute**  **{WorldCollider}** | Rect, material | A rect object to represent the collision hitbox of the worldcollider, along with an associated material object to represent its frictional properties. |
| **File** | Objectives.csv, objects.csv, world.csv | Store properties for Objective, PhysObject and WorldCollider objects respectively for a level. These are stored in csv format where rows are objects and columns are properties in the order they are passed into the constructors in the code e.g objects.csv:  50, 50, 100  100,100,assets/sprites/ball.png,150,0.5,TRUE,0.35  The first line represents a player object, all lines afterwards are regular PhysObject(posx, posy, img, mass, Cd, circular, COR). |

## Algorithms

### Physics Simulation

#### Force Evaluation and Natural Force Application:

*def* Update(*self*, colliders, dt):  
 parent = *self*.parent  
 v = parent.velocity  
  
 touching = touchingany(parent, colliders)  
 touchingEnts = [x[0] *for* x *in* touching]  
  
 rForce = *self*.GetResultantNOF() *for* i, force *in* enumerate(*self*.forces):  
  *if* force.source *not in* touchingEnts *and* force.source != *self*.parent:  
 *self*.forces.pop(i)  
 *if* force.name == "FrictionX":  
 *if* v.x == 0 *and* rForce.x == 0:  
 *self*.forces.pop(i)  
 *if* force.name == "FrictionY":  
 *if* v.y == 0 *and* rForce.y == 0:  
 *self*.forces.pop(i)  
 *if not* parent.weightless *and not self*.GetForce(parent, "Weight") *and* GRAVITYON:  
 *self*.AddForce(parent, "Weight", 0, parent.mass \* GRAVITY)  
 *if not* GRAVITYON *or* parent.weightless:  
 *self*.RemoveForce(parent, "Weight")  
 *for* result *in* touching:  
 ent, side = result  
 *if* isinstance(ent, WorldCollider):  
 *if* side == "left" *or* side == "right" *and* identity(rForce.x) == dir[side].x:  
 *self*.AddForce(ent, "ReactionX", -rForce.x, 0)  
 *elif* side == "top" *or* side == "bottom" *and* identity(rForce.y) == dir[side].y:  
 *self*.AddForce(ent, "ReactionY", 0, -rForce.y)  
 *if* v.GetSqrMag() > 0: *# If moving* A = 0  
 *if* v.x > v.y: *# Simplify cross-sectional area calculation - If horizontal > vertical v, take height as A* A = parent.rect.height / METRE  
 *else*:  
 A = parent.rect.width / METRE  
 dragmag = v.GetSqrMag() \* 0.5 \* AIR\_DENSITY \* parent.Cd \* A  
 drag = v.GetNormalized().Inverse() \* dragmag *# Turn drag magnitude into an opposite-facing vector to velocity* parent.AddForce(parent, "Air Resistance", drag)  
  
 *## ADD FRICTION ##  
 for* result *in* touching:  
 ent, side = result  
 *if* isinstance(ent, WorldCollider):  
 *if self*.GetForce(ent, "ReactionY") *or self*.GetForce(ent, "ReactionX"):  
 N = abs(*self*.GetForce(ent, "ReactionY").y) *if* side == "top" *or* side == "bottom" *else* abs(*self*.GetForce(ent, "ReactionX").x)  
 threshold = ent.GetMuStatic() \* N  
 kinetic = ent.GetMuKinetic()  
  
 *if* side == "top" *or* side == "bottom":  
 *# X friction  
 if* v.x != 0 *or* rForce.x != 0: *# Ensure the object is moving or trying to move* scale = -(identity(rForce.GetNormalized().x)) *if* v.x == 0 *else* -(identity(v.x)) *# Determine the direction for friction to act, first from the force, otherwise from velocity.  
 if* v.x == 0 *and* rForce.x != 0: *# If the object is not moving but is trying to move  
 if* abs(rForce.x) < threshold:  
 *self*.AddForce(ent, "FrictionX", Vec2(-rForce.x, 0)) *# Apply static friction  
 else*:  
 *self*.AddForce(ent, "FrictionX", Vec2(scale \* N \* kinetic, 0)) *# Apply kinetic friction  
 elif* v.x != 0: *# If the object is moving  
 self*.AddForce(ent, "FrictionX", Vec2(scale \* N \* kinetic, 0))  
  
 *elif* side == "left" *or* side == "right":  
 *# Y friction  
 if* v.y != 0 *or* rForce.y != 0: *# Ensure the object is moving or trying to move* scale = -(identity(rForce.GetNormalized().y)) *if* v.y == 0 *else* -(identity(v.y)) *# Determine the direction for friction to act, first from the force, otherwise from velocity.  
 if* v.y == 0 *and* rForce.y != 0: *# If the object is not moving but is trying to move  
 if* rForce.y < threshold:  
 *self*.AddForce(ent, "FrictionY", Vec2(0, -rForce.y)) *# Apply static friction  
 else*:  
 *self*.AddForce(ent, "FrictionY", Vec2(0, scale \* N \* kinetic)) *# Apply kinetic friction  
 elif* v.y != 0: *# If the object is moving  
 self*.AddForce(ent, "FrictionY", Vec2(0, scale \* N \* kinetic))  
  
 *## RECALCULATING RESULTANT FORCE ##  
 self*.rForce.Set(0, 0)  
 *for* force *in self*.forces:  
 *self*.rForce += force *# Sum all the forces acting on the body* parent.acceleration = *self*.rForce / parent.mass *# a = F/m* resultantv = parent.velocity + parent.acceleration \* dt  
  
 *#### PREVENT FRICTION CAUSING OPPOSING MOTION ####* frictionX = *self*.GetAnyForce("FrictionX")  
 frictionY = *self*.GetAnyForce("FrictionY")  
  
 *if* resultantv.x != 0 *and* frictionX:  
 friction = sum(frictionX)  
 *if* (identity(resultantv.x)) == (identity(friction.x)): *# If the resultant velocity is in the same direction as the friction  
 for* force *in* frictionX:  
 *self*.RemoveForce(force)  
 *self*.rForce -= friction *# Remove and ignore the frictional force - recalulate resultant force and acceleration before moving on  
  
 if* resultantv.y != 0 *and* frictionY:  
 friction = sum(frictionY)  
 *if* (identity(resultantv.y)) == (identity(friction.y)): *# If the resultant velocity is in the same direction as the friction  
 for* force *in* frictionY:  
 *self*.RemoveForce(force)  
 *self*.rForce -= friction *# Remove and ignore the frictional force - recalulate resultant force and acceleration before moving on* parent.rForce = *self*.rForce  
 parent.acceleration = parent.rForce / parent.mass

#### Collision Simulation:

*def* Resolve(*self*):  
 obj1 = *self*.object  
 obj2 = *self*.collider  
 pTotal = obj1.momentum + obj2.momentum  
 finalObj1V = obj2.velocity - obj1.velocity *# + finalObj2V* pTotal = pTotal - (finalObj1V \* obj1.mass)  
 finalObj2V = pTotal / (obj1.mass + obj2.mass)  
 finalObj1V = finalObj1V + finalObj2V  
 obj1.SetVelocityVec2(finalObj1V)  
 obj2.SetVelocityVec2(finalObj2V)  
 *self*.resolved = *True*

#### Frame-Independent Movement Simulation:

*def* Resolve(*self*):  
 *self*.forces.Update(colliders, dt)  
 *self*.velocity += *self*.acceleration \* dt  
  
 *if* round(*self*.velocity.x, 1) == 0 *and* identity(*self*.velocity.x) != identity(  
 *self*.rForce.x): *# If velocity is basically 0, and velocity is opposing the direction of resultant force, set this velocity to exactly 0.  
 self*.velocity.x = 0  
 *if* round(*self*.velocity.y, 1) == 0 *and* identity(*self*.velocity.y) != identity(*self*.rForce.y):  
 *self*.velocity.y = 0  
  
 *self*.momentum = *self*.velocity \* *self*.mass  
  
 collision\_types = {"top": *False*, "bottom": *False*, "left": *False*, "right": *False*}  
  
 *## HANDLE X MOVEMENT ##* prevposx = *self*.pos.x  
 *self*.pos.x += *self*.velocity.x \* dt \* METRE  
 *self*.rect.centerx = round(*self*.pos.x)  
 hit\_list = coltest(*self*.rect, colliders)  
  
 *for* entity *in* hit\_list:  
 *if* isinstance(entity, WorldCollider):  
 *if self*.velocity.x > 0:  
 *self*.rect.right = entity.GetRect().left  
 *self*.pos = Vec2(*self*.rect.center)  
 collision\_types["right"] = *True  
  
 elif self*.velocity.x < 0:  
 *self*.rect.left = entity.GetRect().right  
 *self*.pos = Vec2(*self*.rect.center)  
 collision\_types["left"] = *True  
  
 if self*.COR > 0:  
 bounce = abs(*self*.velocity.x) \* *self*.COR  
 *if* bounce > 1:  
 *self*.velocity.x \*= -1 \* *self*.COR  
 *else*:  
 *self*.velocity.x = 0  
 *else*:  
 *self*.velocity.x = 0  
  
 *## HANDLE Y MOVEMENT ##* prevposy = *self*.pos.y  
 *self*.pos.y += *self*.velocity.y \* dt \* METRE  
 *self*.rect.centery = round(*self*.pos.y)  
 hit\_list = coltest(*self*.rect, colliders)  
  
 *for* entity *in* hit\_list:  
 *if* isinstance(entity, WorldCollider):  
 *if self*.velocity.y > 0:  
 *self*.rect.bottom = entity.GetRect().top  
 *self*.pos = Vec2(*self*.rect.center)  
 collision\_types["bottom"] = *True  
  
 elif self*.velocity.y < 0:  
 *self*.rect.top = entity.GetRect().bottom  
 *self*.pos = Vec2(*self*.rect.center)  
 collision\_types["top"] = *True  
  
 if self*.COR > 0:  
 bounce = abs(*self*.velocity.y) \* *self*.COR  
 *if* bounce > 1:  
 *self*.velocity.y \*= -1 \* *self*.COR  
 *else*:  
 *self*.velocity.y = 0  
 *else*:  
 *self*.velocity.y = 0

#### Main Game Loop:

*def* RunFrame(*self*, dt):  
 oldLPos = *self*.lPos  
 *self*.lPos = getCameraTrack(player.GetPos(), *self*.lPos, background\_image.get\_size()[0],  
 background\_image.get\_size()[1])  
  
 *# move game objects accordingly with the level* diff = list(numpy.subtract(*self*.lPos, oldLPos)) *# convert the numpy array to a regular list* player.SetPos(*self*.player.GetPos() + Vec2(diff)) *# PhysObjects can be moved via vector addition  
 for* object *in* objects:  
 object.SetPos(object.GetPos() + Vec2(diff))  
 *for* objective *in* objectives:  
 objective.SetPos(objective.GetPos() + Vec2(diff))  
 *for* particle *in* particleHandler.particles:  
 particle.SetPos(particle.GetPos() + Vec2(diff))  
 *for* wc *in* world: *# Worldcolliders are tracked by rects only, so use numpy list subtraction* wc.Move(diff)  
  
 screen.blit(background\_image, tuple(*self*.lPos))  
  
 font = Pygame.font.Font(*None*, 30)  
 render\_fps = font.render(str(int(clock.get\_fps())), *True*, WHITE)  
 screen.blit(render\_fps, (0, 0))  
 render\_mousepos = font.render(str(Pygame.mouse.get\_pos()), *True*, WHITE)  
 screen.blit(render\_mousepos, (500, 0))  
  
 colliders = world + objects *# Everything the player can collide with  
  
 ## UPDATING ALL GAME OBJECTS ##* player.Update(colliders, dt)  
 player.Draw(screen)  
  
 *for* objective *in* objectives:  
 objective.Update()  
 objective.Draw(screen)  
  
 newcolliders = [x *for* x *in* world]  
 newcolliders.append(player)  
 *for* object *in* objects:  
 object.Update(newcolliders + [x *for* x *in* objects *if* x != object], dt)  
 object.Draw(screen)  
  
 newcol = [x *for* x *in* objects]  
 newcol.append(player)  
 colHandler.Update(newcol)  
  
 particleHandler.Update(dt)

#### Player Controls:

*if* keys[Pygame.K\_RIGHT]:  
 player.Rotate(1, colliders)  
*if* keys[Pygame.K\_LEFT]:  
 player.Rotate(-1, colliders)  
*if* keys[Pygame.K\_SPACE]:  
 base = Vec2(0, PLAYERFORCE)  
 rads = player.angle \* RAD  
 base.x, base.y = (base.x \* math.cos(rads)) - (base.y \* math.sin(rads)), \  
 -((base.x \* math.sin(rads)) + (base.y \* math.cos(rads)))  
 player.AddForce(player, "Drive", base)  
 recoil = base.Inverse().GetNormalized()  
 *for* i *in range*(0, 10):  
 x, y = *tuple*(player.engine)  
 uv = recoil \* 5 + random.randint(-6, 6)  
 particleHandler.Add(EngineParticle(player.engine, uv, random.uniform(0.5, 1.5)))  
*elif* keys[Pygame.K\_LSHIFT]:  
 base = Vec2(0, PLAYERFORCE)  
 rads = player.angle \* RAD  
 base.x, base.y = -((base.x \* math.cos(rads)) - (base.y \* math.sin(rads))), \  
 (base.x \* math.sin(rads)) + (base.y \* math.cos(rads))  
 player.AddForce(player, "Drive", base)

## Usability Features

### Explanation and Justification of the User Interface

The Menu will consist of a simple design with the game title at the top, followed by buttons for each possible component the user may want to access. A main “Play Game” button will allow the user to begin the game from scratch as it was intended to be played. A level selection will also be available if a player wants to load a specific level and play the game from there. In addition, a Leaderboards button will be available for the player to view others’ high scores and submit their own. This will be coupled with a login system which allows recurring players to play and overwrite their old scores or submit new ones to said leaderboard.

Text

Description automatically generated

It may be the case that the menu will render a background consisting of some physics simulated objects bouncing/moving in random ways in a predesigned background world, in order to improve the appearance of what is otherwise a relatively plain design. This would replace the plain grey background of the current concept design.

Within the game itself, the HUD will be mostly empty except for the inclusion of a fuel meter for the player. The fuel meter will colour itself in relation to the amount of fuel available in the aircraft, changing from white to red over the course of its depletion; this would be achieved via linear interpolation. The game will consist of relatively plain background colours yet include bright particle effects for various points of interest. For example, the engine of the player’s aircraft will produce orange “propellant” which gradually becomes white smoke through a linearly interpolated colour change. Objectives will emit small particles of their colours at all times, and collisions may also produce “dust”.

### Description and Justification of the Usability Features

Player input will be conducted entirely via the keyboard. The arrow keys will be used to direct the player’s aircraft, allowing them to rotate it in place. The spacebar and shift keys will be used as static throttle controls. Controlling the player will rely entirely on physics-simulated movement, which induces a natural difficulty in the game as all movements are conducted at the mercy of forces such as gravity, air resistance etc.

Despite the colour scheme following a mostly grey-scale design, some components of the game possess colour in order to emphasise their importance (e.g., engine fumes, objective markers, special physics objects), so less-abled players should still be considered. In order to support these players, the advanced particle system will be generously applied across many parts of the game to provide extra indication of their purpose. For example, objective markers will emit particles according to their type; collisions with certain objects may produce special particle effects.

## Post-Development Test Data

|  |  |  |  |
| --- | --- | --- | --- |
| **Strand** | **Module** | **Test Objectives** | **✓** |
| Physics Simulation | Force Evaluation | Resultant Force should be composed of all forces acting on the object |  |
| Physics values (velocity, acceleration, etc.) should be represented accurately and independent of framerate |  |
| Pixel movement amount and direction should closely represent its real-world equivalent as it relates to the METRE constant. |  |
| All objects must follow F=ma, such that their acceleration is proportional to their resultant force in any direction. |  |
| Collision Simulation | Objects should be stopped or rebound from collisions with WorldColliders |  |
| Objects should experience a transfer of momentum accurate to their physical values when colliding |  |
| Objects should be able to push others by maintaining contact with them and exerting a force in their direction. |  |
| Natural Force Application | Friction should only act when in contact with a surface, according to the material it’s made out of |  |
| Air Resistance should act proportionally to oppose the direction of motion |  |
| Friction should act for both non-moving (but trying to) objects and moving objects. |  |
| Menu | Basic Menu | Buttons should not experience more than a single mouse button press. |  |
| Level Select should display all the levels available to the player and initiate the game correctly. |  |
| Login System | Text input for the user’s username should abide by a 15-character limit, and must include and start with an alphabetic character |  |
| The scores attained during the session should be attributed to the same username. |  |
| Leaderboard | Should display the top 10 scores attained in each level, with their associated usernames. |  |
| Level Design | Environmental Hazards | Moving within dense liquids should result in a representatively increased amount of drag |  |
| Physics Objects should experience a force when intersecting the path of a “wind” hazard |  |
| Score should be deducted when colliding with sides of the level |  |
| Colliding with a forbidden section of the level should immediately result in failure/reset |  |
| Playability | All levels should be complete-able |  |
| Levels should be free of bugs which could be caused by the player and prevent completion. |  |
| Difficulty should progress with level number |  |
| Graphics | Animations | Rotating the player should maintain collision consistency (should not phase through other objects) |  |
| Input from the player should result in a graphical response; e.g thrust should produce engine fumes in the ship |  |
| Rotation should cause minimal image distortion |  |
| HUD | Fuel meter should deplete accurately in accordance with the remaining fuel in the player’s ship |  |
| Physics information should display accurately and be readable if the player chooses to enable it |  |

### Description and Justification of the Validation Required

A clear issue in any platformer game is the unintended ability for the player to leave the area/map. In order to counteract this, all maps are initialised with WorldCollider objects surrounding their borders, preventing the player and any physics objects from ever leaving the confines of the level.

*Snippet of level initiation code:*

*def \_\_init\_\_*(*self*, stateobj, background, world, objects, player, objectives):  
 *self*.state = stateobj  
 *self*.background\_image = Pygame.image.load(background).convert\_alpha()  
 *self*.level\_size = *self*.background\_image.get\_size()  
 *self*.lwidth, *self*.lheight = *self*.level\_size  
  
 *self*.world = world  
 *self*.world.append(WorldCollider(Pygame.Rect(-1, 0, 1, *self*.lheight)))  
 *self*.world.append(WorldCollider(Pygame.Rect(0, -1, *self*.lwidth, 1)))  
 *self*.world.append(WorldCollider(Pygame.Rect(*self*.lwidth, 0, 1, *self*.lheight)))  
 *self*.lPos = [0, 0]  
  
 *self*.objects = objects  
 *self*.player = player  
 *self*.objectives = objectives  
  
 *self*.colHandler = CollisionHandler(*self*.level\_size)  
 *self*.particleHandler = ParticleHandler()  
 *self*.timer = Timer((0,0))

Control validation will also be conducted when the player is touching another object or WorldCollider. This is done to prevent a glitch where the player begins to move through objects due to its hitbox phasing through other hitboxes in the world:

*def* Rotate(*self*, scale, colliders):  
 *if len*(touchingany(*self*, colliders)) == 0:  
 old\_rect = copy.deepcopy(*self*.rect)  
 scale \*= -1 *# We want to interpret + rotation as clockwise  
 self*.angle += PLAYER\_ROTATION\_SPEED \* scale  
 rotated\_image = Pygame.transform.rotate(*self*.image\_clean, *self*.angle)  
 *self*.image = rotated\_image  
 *self*.rect = *self*.image.get\_rect(center=old\_rect.center)  
 *self*.angleDir = Vec2(math.cos((90 + *self*.angle) \* RAD), -math.sin((90 - *self*.angle) \* RAD)).GetNormalized()

Other validation will come in the form of input text validation when entering score details. For example, names should not contain more than 15 characters to prevent memory overflow and start with a letter to keep the scoreboard readable.

## Test Data for Text Input

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Type** | **Test Data** | **✓** |
| Username Entry | Valid | *“bob445”* |  |
| *“bobbillson”* |  |
| *“bobby404-\_”* |  |
| Boundary | *“a”* |  |
| *“bbbbbbbbbbbbbbb”* |  |
| *“c\_c------------”* |  |
| *“d1”* |  |
| Invalid | *“---bob”* |  |
| *“1”* |  |
| *“1bob1”* |  |
| *“1234b”* |  |
| *“abbbbbbbbbbbbbbc”* |  |
| Null |  |  |

# Development

## Physics Simulation

### Version 1

#### Force & Motion Evaluation

The core of the game revolves around the evaluation and translation of forces acting on objects into movement over time. This is done using various physics formulas relating to rigid body mechanics and programming the motion of PhysObjects around these. This is done by the ForceManager object, attached to all PhysObjects.

The simulation is computed through a hierarchy of objects which handle differing parts of motion and arithmetic. In version 1, it is as follows:

The Force object is a subclass of the Vec2 object – which enables them to store x and y components and compute 2d vector arithmetic with each other (and, by nature, any other object which is a Vec2 or a derivation of it). This becomes very useful when writing the force evaluation algorithm, as standard mathematical operators can be used to make these calculations much more maintainable and accessible.

The ForceManager itself performs most of the computation pertaining to the forces acting on the object. Most importantly, it calculates the resultant acceleration possessed by the object due to the resultant force acting upon it (the vector sum of all forces). For example, the resultant force of an object with (1000, 1000)N and (-1000,-1000)N forces acting upon it would be (0,0)N.

##### Acceleration

The most fundamental equation used is Newton’s Second Law of motion:

Where:

* F = Resultant Force on the object
* m = Mass of the object
* a = Acceleration of the object

This equation forms the basis for any movement in the game; force is directly proportional to acceleration, therefore, rearranged:

This is represented in the code for the ForceManager’s “Update” method (called every frame) as:

*## RECALCULATING RESULTANT FORCE ##  
self*.rForce.Set(0, 0)  
*for* force *in self*.forces:  
 *self*.rForce += force *# Sum all the forces acting on the body*parent.acceleration = *self*.rForce / parent.mass  
resultantv = parent.velocity + parent.acceleration \* dt  
  
*#### PREVENT FRICTION CAUSING OPPOSING MOTION ####*frictionX = *self*.GetAnyForce("FrictionX")  
frictionY = *self*.GetAnyForce("FrictionY")  
  
*if* resultantv.x != 0 *and* frictionX:  
 friction = sum(frictionX)  
 *if* (identity(resultantv.x)) == (identity(friction.x)): *# If the resultant velocity is in the same direction as the friction  
 for* force *in* frictionX:  
 *self*.RemoveForce(force)  
 *self*.rForce -= friction *# Remove and ignore the frictional force - recalulate resultant force and acceleration before moving on  
  
if* resultantv.y != 0 *and* frictionY:  
 friction = sum(frictionY)  
 *if* (identity(resultantv.y)) == (identity(friction.y)): *# If the resultant velocity is in the same direction as the friction  
 for* force *in* frictionY:  
 *self*.RemoveForce(force)  
 *self*.rForce -= friction *# Remove and ignore the frictional force - recalulate resultant force and acceleration before moving on*parent.rForce = *self*.rForce  
parent.acceleration = parent.rForce / parent.mass

In order to maintain framerate-independence, velocity is increased by the acceleration multiplied by “dt” [time since last frame] to ensure only the specific fraction of a second’s worth of velocity is added.

The method “GetAnyForce” returns all the forces on the object with the specified name:

*def* GetAnyForce(*self*, name):  
 t = [x *for* x *in self*.forces *if* x.name == name]  
 *if* len(t) > 0:  
 *return* t  
 *else*:  
 *return None*

The identity function takes turns values into 1 or -1 to allow us to compare their signs/directions:

*def* identity(n):  
 *if* n != 0:  
 *return* n / abs(n)  
 *else*:  
 *return* n

By utilising these two algorithms, we check and ensure frictional forces never act to support motion or when motion is not occuring, as by nature they exist only to counteract motion.

##### Velocity

Once acceleration has been calculated, the program moves back to the parent PhysObject to complete the rest of the process. Velocity is determined by:

Where:

* v = Final (new) velocity
* u = Initial (current) velocity
* t = Time (possessing the same acceleration)

Represented in the PhysObject’s “Update” method as:

*self*.forces.Update(colliders, dt)  
*self*.velocity += *self*.acceleration \* dt  
  
*# If velocity is basically 0, and velocity is opposing the direction of resultant force, set this velocity to exactly 0.   
if* round(*self*.velocity.x, 1) == 0 *and* identity(*self*.velocity.x) != identity(*self*.rForce.x):  
 *self*.velocity.x = 0  
*if* round(*self*.velocity.y, 1) == 0 *and* identity(*self*.velocity.y) != identity(*self*.rForce.y):  
 *self*.velocity.y = 0

Following this, velocity must then be converted into actual translation on the screen itself.

Where s = displacement

In version 1, this is accomplished by computing the change in x and y coordinates of the sprite seperately to allow for independent collision detection:

prevposx = *self*.pos.x  
*self*.pos.x += *self*.velocity.x \* dt \* METRE  
*self*.rect.x = round(*self*.pos.x)

prevposy = *self*.pos.y  
*self*.pos.y += *self*.velocity.y \* dt \* METRE  
*self*.rect.y = round(*self*.pos.y)

Both of which are framerate-independent computations, as shown by the use of “dt”. The METRE value is a constant determined by the height of the player and is based on the fact that the player is intended to be 1.7m tall in the in-game world (average height of a person).

METRE = player\_image.get\_height() \* (1 / 1.7)

#### Natural Force Application

In addition to forces applied by other objects, an object’s ForceManager must also generate and manage the natural forces that should apply to it according to the environment it is in. These include:

* Weight (Gravity)
* Normal Reaction Force
* Friction
* Air Resistance

It is also important to note that the “AddForce” method of ForceManager performs its own cleanup operations when called, to ensure no duplicate or empty forces are added:

*def* AddForce(*self*, source, name, \*args):  
 vec = Vec2(0, 0)  
 *if* len(args) == 2:  
 vec.Set(args[0], args[1])  
 *else*:  
 vec = args[0]  
 *if* round(vec, 1) != Vec2(0, 0):  
 *if self*.GetForce(source, name) *is None*:  
 *self*.forces.append(Force(source, name, vec))  
 *else*:  
 *self*.RemoveForce(source, name)  
 *self*.forces.append(Force(source, name, vec))  
 *else*:  
 *self*.RemoveForce(source, name)

##### Weight

By far the simplest, the magnitude of the weight force can be determined by:

A substitution of Newton’s Second Law, it uses the gravitational field strength (an acceleration experienced by all objects within) to determine the size of the weight force. In the game, the idea of a changing gravitational force with height is unnecessary and insignificant, and the force always acts directly downards.

*if not* parent.weightless *and not self*.GetForce(parent, "Weight"):  
 *self*.AddForce(parent, "Weight", 0, parent.mass \* GRAVITY)

GRAVITY is a global constant storing the gravitational field strength in the game world.

##### Pre-computation

Before computing any contact forces (friction, normal reaction), the objects which the parent object is in contact and which side it is occuring on should first be determined:

touching = touchingany(parent, colliders)  
touchingEnts = [x[0] *for* x *in* touching]

Another issue with calculating drag forces arises when including the drag forces themselves as part of the resultant force used in the calculations. As a result, we must use an alternate version of the resultant force which excludes these:

rForce = *self*.GetResultantNOF()

*def* GetResultantNOF(*self*):  
 rForce = Vec2(0, 0)  
 *for* force *in self*.forces:  
 *if* "Friction" *not in* force.name *and* force.name != "Air Resistance":  
 rForce += force  
 *return* rForce

##### Normal Reaction Force

The Normal Reaction Force acts as a combined result of Newton’s First and Third Laws of Motion. It acts to oppose the resultant force from one object onto another, as long as those objects are in contact. This is accomplished as such:

*for* result *in* touching:  
 ent, side = result  
*if* side == "left" *or* side == "right" *and* identity(rForce.x) == dir[side].x:  
 *self*.AddForce(ent, "ReactionX", -rForce.x, 0)  
*elif* side == "top" *or* side == "bottom" *and* identity(rForce.y) == dir[side].y:  
 *self*.AddForce(ent, "ReactionY", 0, -rForce.y)

“dir” is a dictionary which translates a side string into a normalized Vec2 representative of that side’s cardinal direction, with ±1 or 0 for x and y

##### Air Resistance

The formula for the magnitude of air resistance is as follows and always acts directly opposite to the direction of velocity:

Where:

* ρ = Density of the fluid
* CD = Drag coefficient of the shape
* A = Cross sectional area of the shape

Since the player will almost definitely not notice slight inaccuracies in the calculation of air resistance (nor will it truly affect gameplay that much), most of the values used are rough estimates:

* The Density of the fluid is determined by a global constant AIR\_DENSITY and modelled on the Earth (1.2kg/m3).
* The drag coefficient is specified for each object. In the game, I used an [online engineering table](https://www.engineeringtoolbox.com/drag-coefficient-d_627.html) listing drag coefficients for common shapes and objects. E.g., the player has the coefficient for a person standing (1.0 – 1.3 ~ 1.15)
* The cross-sectional area is a simplified calculation where all shapes on the screen are assumed to be cuboids with width and height according to their Rects, and a depth of 1m. This allows us to simply multiply the width or the height by 1m and refer to that as the rough cross-sectional area of the shape when calculating drag force in that direction.

*if* v.GetSqrMag() > 0: *# If moving* A = 0  
 *if* v.x > v.y: *# Simplify cross-sectional area calculation - If horizontal > vertical v, take height as A* A = parent.rect.height / METRE  
 *else*:  
 A = parent.rect.width / METRE  
 dragmag = v.GetSqrMag() \* 0.5 \* AIR\_DENSITY \* parent.Cd \* A  
 drag = v.GetNormalized().Inverse() \* dragmag *# Turn drag magnitude into an opposite-facing vector to velocity* parent.AddForce(parent, "Air Resistance", drag)

The methods “GetNormalized”, “GetSqrMag” and “Inverse” are all members of the Vec2 class, and work as described.

##### Friction

Friction comes in two forms – static and kinetic – whose nature depends on the value of Fmax (maximum friction between the surfaces) given by:

Where:

* µ = The coefficient of friction between the two surfaces
* R = The reaction force experienced by the object on the surface

Static friction acts upon an object while it is stationary to completely counteract a driving force across a surface (that is, equal its magnitude in the opposite direction) until that force exceeds the value of Fmax. From this point, the magnitude and direction of [kinetic] friction is constant and equal to Fmax until the resultant force from the object drops below its value, the value of R changes, or the object stops moving in that direction.

This intuitively makes sense as the harder an object is pressed into a surface, the higher its magnitude of R will be and the higher the friction it will experience. Likewise, the less an object is pressed into a surface (and of course, if it is not touching it at all, wherein R would be 0), the less it would experience friction.

In practice, many checks are needed to determine the direction, nature, and magnitude of a frictional force to applied by a surface, as shown in the x component of the friction algorithm:

*for* result *in* touching:  
 ent, side = result  
 *if* isinstance(ent, WorldCollider):  
 *if self*.GetForce(ent, "ReactionY") *or self*.GetForce(ent, "ReactionX"):  
 N = abs(*self*.GetForce(ent, "ReactionY").y) *if* side == "top" *or* side == "bottom" *else* abs(*self*.GetForce(ent, "ReactionX").x)  
 threshold = ent.GetMuStatic() \* N  
 kinetic = ent.GetMuKinetic()  
  
 *if* side == "top" *or* side == "bottom":  
 *# X friction  
 if* v.x != 0 *or* rForce.x != 0: *# Ensure the object is moving or trying to move* scale = -(identity(rForce.GetNormalized().x)) *if* v.x == 0 *else* -(identity(v.x)) *# Determine the direction for friction to act, first from the force, otherwise from velocity.  
 if* v.x == 0 *and* rForce.x != 0: *# If the object is not moving but is trying to move  
 if* abs(rForce.x) < threshold:  
 *self*.AddForce(ent, "FrictionX", Vec2(-rForce.x, 0)) *# Apply static friction  
 else*:  
 *self*.AddForce(ent, "FrictionX", Vec2(scale \* N \* kinetic, 0)) *# Apply kinetic friction  
 elif* v.x != 0: *# If the object is moving  
 self*.AddForce(ent, "FrictionX", Vec2(scale \* N \* kinetic, 0))

Note the fact that friction only applies between collisions involving a PhysObject and a WorldCollider, and in no other circumstance. This simplifies many aspects of the game and creates less room for issues which would otherwise provide insignificant benefit to the gameplay experience.

#### Collision Simulation

In Version 1, collision simulation is highly simplified and simply acts to prevent objects phasing through each other. Between PhysObjects and worldcolliders, an illusory “bounce” is simulated for objects with a coefficient of restitution greater than 0 (set when constructing a PhysObject).

##### Pre-computation

All the objects we are colliding with should first be retrieved:

hit\_list = coltest(*self*.rect, colliders)

*def* coltest(rect, colliders):  
 hit\_list = []  
 *for* collider *in* colliders:  
 *if* rect.colliderect(collider.GetRect()):  
 hit\_list.append(collider)  
 *return* hit\_list

We then compute collisions seperately for x and y just like what was done for movement on the screen.

##### PhysObject → Any

1. Check the side of collision estimated by values of velocity
2. Reset our position to the edge we assumed we collided with

*for* entity *in* hit\_list:  
 *if self*.velocity.y > 0:  
 *self*.rect.bottom = entity.GetRect().top  
 *self*.pos = Vec2(*self*.rect.topleft)  
 collision\_types["bottom"] = *True  
  
 elif self*.velocity.y < 0:  
 *self*.rect.top = entity.GetRect().bottom  
 *self*.pos = Vec2(*self*.rect.topleft)  
 collision\_types["top"] = *True*

Collision computation for y component only

##### PhysObject → PhysObject

There is an attempt to resolve physobject collisions in version 1, however the algorithm is almost completely incorrect and does not work. It attempts to apply forces to both the objects when they collide and transfer velocity naively.

*if not* isinstance(ent, WorldCollider):  
 *if* side == "left" *or* side == "right":  
 *if* identity(rForce.x) == dir[side].x:  
 ent.AddForce(parent, "CollisionX", Vec2(rForce.x, 0))  
 *else*:  
 initialparentv = parent.velocity.x  
 initialentv = ent.velocity.x  
 parent.velocity.x = -(parent.mass/ent.mass) \* initialentv  
 ent.velocity.x = -(ent.mass/parent.mass) \* initialparentv  
 *if* side == "top" *or* side == "bottom":  
 *if* identity(rForce.y) == dir[side].y:  
 ent.AddForce(parent, "CollisionY", Vec2(0, rForce.y))  
 *else*:  
 initialparentv = parent.velocity.y  
 initialentv = ent.velocity.y  
 parent.velocity.y = -(parent.mass/ent.mass) \* initialentv  
 ent.velocity.y = -(ent.mass/parent.mass) \* initialparentv

##### PhysObject → WorldCollider

* If the COR is greater than 0, rebound the object at the appropriate velocity using the (naive) formula:
* If the COR is 0, simply set our velocity in that direction to 0.

The values given for coefficients of restitution of PhysObjects are approximated and simply act to provide the illusion of a fully simulated bounce. A realistic algorithm would be extremely complex and would, again, add insignificant benefit to the gameplay experience.

*if* isinstance(entity, WorldCollider):  
 *if self*.COR > 0:  
 bounce = abs(*self*.velocity.y) \* *self*.COR  
 *if* bounce > 1:  
 *self*.velocity.y \*= -1 \* *self*.COR  
 *else*:  
 *self*.velocity.y = 0  
 *else*:  
 *self*.velocity.y = 0

WorldCollider collision for the y component

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Force & Motion Evaluation | All objects in the world should experience the same downwards acceleration as a result of their applied weight force | All objects possess acceleration of Vec2(0, GRAVITY) | With GRAVITY set to 15 m/s2,  All objects experienced acceleration (0,15)m/s2 | **✓** |
| Objects should attain resultant velocity according to their acceleration and the time between frames | An object should gain a velocity according to its acceleration and the time between frames | An object at rest with an acceleration (0,15)m/s2 ended up with a velocity of (0, 0.3828786) m/s after a frametime of 0.02552524 seconds. | **✓** |
| Natural Force Application | Objects must always a weight force acting on them | All objects possess an accurate weight force according to their mass and are affected by it accordingly | Player object with a mass of 75 kg, GRAVITY of 15 N/kg  Player object had a force of (0, 1125)N and a resultant acceleration of (0,15)m/s2 | **✓** |
| Objects should experience a Reaction force when applying a force against a worldcollider | The player should experience a reaction force from the floor equal and opposite in magnitude to its weight | Forces acting on the player on the floor:  Weight – (0, 1125)N  ReactionY – (0,  -1125)N | **✓** |
| Drag forces should accurately on an object according to their formulas | For the player in free fall after reaching 15m/s they should experience air resistance of  (0.61)  = 95 | At 15m/s:  Air Resistance is (0,  -93.38171260582484)  N  Close enough to the expected value (which is inherently inaccurate due to the surface area of 0.61 being rounded) | **✓** |
| Friction should act on an object when passing over a surface | For a player with weight 1125N on a surface of frictional coefficient 0.65:  = 731.25N and should oppose the direction of motion along the surface. | Player of weight (0, 1125) with a kinetic coefficient on “Asphalt” material of 0.65 and velocity (9.6, 0.0) m/s:  FrictionX = (-731.25, 0)N  Exactly the expected value  However, a bug causes the game to crash when an object falls off the edge of a surface. | ✘ |
| Collision Simulation | Objects should rebound from each other according to their momentum | A 75kg player moving towards a 60kg ball at rest should push it in the direction of movement and lose some of its own velocity. | The objects hardly move and vibrate rapidly in place. The algorithm completely fails to simulate collisions between physics objects. | ✘ |
| Objects should be able to push each other. | A 75kg player moving into a 60kg ball (and maintaining its force in the direction of the ball) should be able to push it in that direction. | Same as collisions, objects hardly move except for small jitters. There is no proper method of “pushing” available in this version. | ✘ |
| Objects should bounce off WorldColliders | Object with a positive x and y velocity and a coefficient of restitution of 0.5 should rebound with half its vertical velocity when colliding with a worldcollider directly underneath it. | Ball with velocity (0,5)m/s ends up with a final velocity of (2.493, -2.5)m/s after colliding with a worldcollider below it. | **✓** |

### Version 2

#### Force & Motion Evaluation

The algorithms for determining acceleration and velocity for an object are unchanged from Version 1 to Version 2.

#### Natural Force Application

##### Friction

The basic method of friction application is unchanged from Version 1 to Version 2 – however, a bug causing the game to crash when sliding off an edge was fixed and was caused by a problematic method of retrieving the normal reaction from the surface:

N = *abs*(*self*.GetForce(ent, "ReactionY").y) *if* side == "top" *or* side == "bottom" *else abs*(*self*.GetForce(ent, "ReactionX").x)

The problem here is that it assumes a reaction force always exists, and when this is not the case (such as at the very moment and object slides off the edge of an object) the program tries to fetch the x component of what is actually a *NoneType* object. This results in a crash, and was fixed as so:

N = 0  
*if* side == "top" *or* side == "bottom":  
 force = *self*.GetForce(ent, "ReactionY")  
 *if* force *is not None*:  
 N = *abs*(force.y)  
*else*:  
 force = *self*.GetForce(ent, "ReactionX")  
 *if* force *is not None*:  
 N = *abs*(force.x)

#### Collision Simulation

##### PhysObject → PhysObject

PhysObject collision in Version 2 is far improved from the original system and provides the foundations for the collision system found in the final version. A hierarchy of classes is established which relate to the handling of collisions in the world. These classes manage collisions of objects externally and handle/provide methods for dealing with them.

###### Ray-Casting

In Version 2, there exists a way to allow objects to push each other. This uses a new function which provides a ray-casting function in the game. Lines can be drawn from a point in a given direction (given by a normalized Vec2) and information is returned about whether a specified target was hit.

*def* trace(source, dir, target, detailed=*False*):  
 *"""* ***:param*** *source: A source point or object* ***:param*** *dir: Normalized vector representing direction of the ray* ***:type*** *dir: Vec2* ***:param*** *target: What to test for ray collision on* ***:param*** *detailed: Detailed info about the hit* ***:return****: Whether a hit was registered  
 """*

The function parameters and the documentation included in the program to describe each parameter’s purpose and nature.

It uses the line equation:

The function scans along an axis by incrementing along it and plotting the equivalent points on the alternate axis. These points are checked one by one for a detection of the target. It is important to understand that the gradient of a line is the change in y for a specific change in x:

A gradient without an x component is undefined as it involves a division by zero, so the function must handle it seperately.

The function determines whether the direction has an x-component or not first, and how much of it composes the total gradient. This value has a maximum of 1 since the vector is normalized, where the line is completely horizontal. This is also used as the increment amount when tracing the line.

width, height = WINDOW\_SIZE  
xDir = *int*(identity(dir).x)  
end\_pos = (0, 0)

For a standard gradient, the bounds are determined as the x coordinate of the source, and the level width is added in the direction of the line to cover the entire level (it will always compute more than is necessary, but does not cause a performance hit)

*if* xDir != 0:  
 gradient = dir.y / dir.x  
 bounds = [*int*(source.x), LEVEL\_SIZE[0] *if* xDir == 1 *else* -LEVEL\_SIZE[0]]

LEVEL\_SIZE[0] is the x-component of the level\_size (width)

*else*:  
 yDir = *int*(identity(dir).y)  
 bounds = [*int*(source.y), LEVEL\_SIZE[1] *if* yDir == 1 *else* -LEVEL\_SIZE[1]]

The y-axis equivalent code

The bounds are then scanned in increments equal to the normalized gradient, and every scanned point is tested for a collision with the target rect.

*for* x *in range*(bounds[0], bounds[1], xDir):  
 y = x \* gradient + source.y  
 *if* DEBUG *and* x == bounds[1] - xDir:  
 end\_pos = (x, *int*(y))  
 Pygame.draw.aaline(screen, RED, (source.x, source.y), end\_pos)  
 *if* target *is not None*:  
 *if* target.collidepoint(x, y):  
 *return True*

X-Axis

*for* y *in range*(bounds[0], bounds[1], yDir):  
 *if* DEBUG *and* y == bounds[1] - yDir:  
 end\_pos = (source.x, y)  
 Pygame.draw.aaline(screen, RED, (source.x, source.y), end\_pos)  
 *if* target *is not None*:  
 *if* target.collidepoint(source.x, y):  
 *return True*

Y-Axis

###### The Collision Object

Collision objects are instantiated by PhysObjects (and added to the CollisionHandler object) when they collide with another PhysObject:

*if isinstance*(entity, WorldCollider):  
 *if self*.COR > 0:  
 bounce = *abs*(*self*.velocity.y) \* *self*.COR  
 *if* bounce > 1:  
 *self*.velocity.y \*= -1 \* *self*.COR  
 *else*:  
 *self*.velocity.y = 0  
 *else*:  
 *self*.velocity.y = 0  
*else*:  
 colHandler.AddCol(*self*, entity)

They contain two PhysObjects and have 3 main methods for dealing with a collision. They possess a Boolean value “resolved” which is set to True once their first momentum transfer is computed. After this, if they are still overlapping and resolved, the algorithm checks if one is pushing the other using ray-casts.

Once the collision is created, it is resolved as soon as possible using the following algorithm:

*def* Resolve(*self*):  
 obj1 = *self*.object  
 obj2 = *self*.collider  
 pTotal = obj1.momentum + obj2.momentum  
 finalObj1V = obj2.velocity - obj1.velocity *# + finalObj2V* pTotal = pTotal - (finalObj1V \* obj1.mass)  
 finalObj2V = pTotal / (obj1.mass + obj2.mass)  
 finalObj1V = finalObj1V + finalObj2V  
 obj1.SetVelocityVec2(finalObj1V)  
 obj2.SetVelocityVec2(finalObj2V)  
 *self*.resolved = *True*

The algorithm is based upon the law of conservation of momentum. Since all collisions are treated as perfectly elastic\*, the total velocity in the system is also conserved, and the transfer of momentum between the objects can be calculated without having to worry about highly complex factors such as compression etc.

\*: No energy is lost during the collision – instead, energy is lost due to drag forces opposing the motion of the objects in general.

The pushing method is used to check whether an object is pushing on another in an overlap:

@staticmethod  
*def* pushing(obj1, obj2): *if* obj1.GetResultantForce() != Vec2(0, 0) *and* obj1.GetVelocity() != Vec2(0, 0):  
 source = Vec2(obj1.GetCentre())  
 *return* trace(source, obj1.GetResultantForce().GetNormalized(), obj2) *and* \  
 trace(source, obj1.GetVelocity().GetNormalized(), obj2)  
 *else*:  
 *return False*

Returns whether the direction of the resultant force is into the second object and it is moving into the second object.

The ResolveOverlap function uses the pushing method to check how to apply pushing and reaction forces between the two objects. It also cleans up forces which should no longer apply.

*def* ResolveOverlap(*self*):  
 obj1, obj2 = *self*.object, *self*.collider  
  
 *if* Collision.pushing(obj1, obj2):  
 obj2.AddForce(obj1, "Push", obj1.GetResultantNOF())  
 obj1.AddForce(obj2, "Reaction", obj1.GetResultantNOF())  
 *else*:  
 obj2.RemoveForce(obj1, "Push")  
 obj1.RemoveForce(obj2, "Reaction")  
  
 *if* Collision.pushing(obj2, obj1):  
 obj1.AddForce(obj2, "Push", obj2.GetResultantNOF())  
 obj2.AddForce(obj1, "Reaction", obj2.GetResultantNOF())  
 *else*:  
 obj1.RemoveForce(obj2, "Push")  
 obj2.RemoveForce(obj1, "Reaction")

Note that frictional forces are ignored when retrieving the resultant force.

###### The Collision Handler

The CollisionHandler is a basic manager class (similar to the force manager) which stores and handles all the collision in the world. It provides a filtered method for adding collisions and resolves collisions accordingly in its update method. It alos cleans up collisions which no longer apply.

*def* Update(*self*, world): *for* i, collision *in enumerate*(*self*.collisions):  
 *if not* collision.resolved:  
 collision.Resolve()  
 *else*: *if not* collision.CheckOverlap():  
 collision.PreRemoval()  
 *self*.collisions.pop(i)

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Natural Force Application | Friction should act on an object when passing over a surface, and properly handle sliding off an edge. | For a player with weight 1500N on a surface of frictional coefficient 0.65:  = 975N and should oppose the direction of motion along the surface. The player should be able to slide off the surface and its friction be handled accordingly (removed). | Player of weight (0, 1500) with a kinetic coefficient on “Asphalt” material of 0.65 and velocity (9.6, 0.0) m/s:  FrictionX = (-975, 0)N  Exactly the expected value. When the player slides off, their frictional force is set to 0 (and subsequently removed). | **✓** |
| Collision Simulation | Objects should rebound from each other according to their momentum | A 75kg player moving towards a 150kg ball (which is moving in the opposite direction) should experience a momentum transfer, where the player ends up with a higher velocity than the ball. | For a 75kg player with v = (25.7, 0)m/s and 150kg ball with v = (-11.3, 0)m/s,  Final velocities:  Player: (-18.67, 0)m/s  Ball: (18.31, 0)m/s | **✓** |
| Objects should be able to push each other. | A 75kg player moving into a 60kg ball (and maintaining its force in the direction of the ball) should be able to push it in that direction. | The player can push the ball, but excessive vibration causes bugs in the collision detection, leading to objects teleporting around each other to different sides. The behaviour is not consistent or successful enough to consider the test passed. | ✘ |
| Objects should bounce off WorldColliders | Object with a positive x and y velocity and a coefficient of restitution of 0.5 should rebound with half its vertical velocity when colliding with a worldcollider directly underneath it. | Ball with velocity (0,5)m/s ends up with a final velocity of (2.493, -2.5)m/s after colliding with a worldcollider below it. | **✓** |

### Version 3

#### Force & Motion Evaluation

##### Velocity

Translation on the screen is handled differently in Version 3. Due to the camera and particle functionalities present in this version - in order to decrease the complexity and size of the program - the movement code was turned into a static method in the CollisionHandler class. This allows it to be used on any object in the game which contains a rect.

@staticmethod  
*def* SafeMove(object, colliders, delta):  
 returnVals = {"x": *False*,  
 "y": *False*}  
  
 *## HANDLE X MOVEMENT ##* oldRect = copy.deepcopy(object.GetRect())  
 object.pos.x += delta.x  
 *if isinstance*(object, WorldCollider):  
 object.rect.left = object.pos.x  
 *else*:  
 object.rect.centerx = object.pos.x  
 colliders = [x *for* x *in* colliders *if* object != x]  
 hit\_list = coltest(object.rect, colliders)  
  
  
 *for* entity *in* hit\_list:  
 *if* delta.x > 0 *and* oldRect.right <= entity.rect.left:  
 object.rect.right = entity.GetRect().left  
 object.pos = Vec2(object.rect.center)  
  
 *elif* delta.x < 0 *and* oldRect.left >= entity.rect.right:  
 object.rect.left = entity.GetRect().right  
 object.pos = Vec2(object.rect.center)  
  
 returnVals["x"] = *True*

The X-component of the SafeMove method

The returnVals dictionary is used to represent whether the object collided on the x-axis and/or the y-axis when it moved. This information can be used in various ways, but the most notable is to make PhysObjects rebound from WorldColliders.

delta is the parameter storing the change in y and change in x desired by the function caller. In the case of PhysObject movement, this would be the distance travelled in the time dt (calculated externally):

tempcolliders = [x *for* x *in* colliders *if isinstance*(x, WorldCollider)]  
delta = *self*.velocity \* dt \* METRE  
  
bounceAxis = CollisionHandler.SafeMove(*self*, tempcolliders, delta)  
  
*if* bounceAxis["x"]:  
 *if self*.COR > 0:  
 bounce = *abs*(*self*.velocity.x) \* *self*.COR  
 *if* bounce > 1:  
 *self*.velocity.x \*= -1 \* *self*.COR  
 *else*:  
 *self*.velocity.x = 0  
 *else*:  
 *self*.velocity.x = 0

Snippet of PhysObject movement code; demonstrates each step in the new translation process.

#### Collision Simulation

##### PhysObject → PhysObject

Collision Simulation in Version 3 makes collision detection an external process; instead of letting objects create their own collisions, the collision handler scans the scene for any collisions. This provides the advantage of making collisions occur after all movement has happened, rather than the ‘first-come first-served’ system previously used. As a result, collisions are much more consistent and realistic, and objects never teleport around each other’s sides.

The CollisionHandler now possesses a new method which is called every frame named ColScan:

*def* ColScan(*self*, world):  
 *for* object *in* world:  
 otherObjects = [x *for* x *in* world *if* x != object] *# All the other objects in the world* collisionIndex = object.GetRect().collidelistall(otherObjects) *# Get the indexes for every object we are colliding with* colliders = [otherObjects[x] *for* x *in* collisionIndex] *# Create a list of objects by their indexes  
 for* collider *in* colliders:  
 collision = Collision(object, collider)  
 *if* collision *not in self*.collisions:  
 *self*.collisions.append(collision)

collidelistall is an in-built Pygame method which returns the indexes of colliding rects in the provided collider list

Collisions between PhysObjects and WorldColliders are still handled by objects themselves. Collisions are resolved via the same algorithm in the previous version.

Unfortunately, this new system removes the possibility for one object to push another – they will instead phase through each other if they remain in an overlap. While this is unrealistic, it prevents the endless issues caused by the former, and such an occurrence is relatively unlikely to occur while playing the game, and the levels do not require the player to push any objects – only hit them.

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Collision Simulation | Objects should rebound from each other according to their momentum | A 75kg player moving towards a 150kg ball (which is moving in the opposite direction) should experience a momentum transfer, where the player ends up with a higher velocity than the ball. | For a 75kg player with v = (25.7, 0)m/s and 150kg ball with v = (-11.3, 0)m/s,  Final velocities:  Player: (-18.67, 0)m/s  Ball: (18.31, 0)m/s | **✓** |
| Objects should be able to push each other. | A 75kg player moving into a 60kg ball (and maintaining its force in the direction of the ball) should be able to push it in that direction. | Trying to “push” an object now instead induces multiple collisions on the target object until the two objects overlap. Instead of having a method to deal with this, the objects simply phase through each other. | ✘ |
| Objects should bounce off WorldColliders | Object with a positive x and y velocity and a coefficient of restitution of 0.5 should rebound with half its vertical velocity when colliding with a worldcollider directly underneath it. | Ball with velocity (0,5)m/s ends up with a final velocity of (2.493, -2.5)m/s after colliding with a worldcollider below it. | **✓** |

## Graphics

### Version 1

#### Animations/Particles

##### Background

Shape, rectangle

Description automatically generatedThe background of the level serves as both the backdrop as well as the level itself. WorldColliders are added manually to the appropriate places after a background (a level) has been drawn. This is made much easier through photoshop.

The first test background; notice the ramp surfaces on the right which were used to test diagonal hitboxes. These ended up not being possible to create in Pygame. This background is the only one available in Version 1.

The background also determines the size of the level, as all images are not resized in any way when being used in the game. In this case, the level size is 1920x720 pixels.

##### Drawing Sprites

Shape, rectangle

Description automatically generatedThe sprites used in the game are small images of various sizes, stored in the “assets” folder in the root directory of the game. The sprites are created in exactly the dimensions used in the game, and their hitboxes are in turn derived from these.

Example of sprites being drawn for objects in the game. On the floor are the player and a ball PhysObject with their hitboxes highlighted in yellow.

##### Rotation

The rotation of sprites appears simple to do in Pygame, but several problems arise as a result. In Version 1 the centre of rotation is at the top-left of the sprite, meaning sprites will experience translation across the level when rotating (an unwanted effect). Rotation also spreads out hitboxes as Pygame cannot handle rotated rect objects.

In order to allow rotations to translate well into use for things such as force application at a desired angle, we must produce a way of representing in a vector format.

All objects are instantiated in an upright position, which is designated as 0 degrees. The main angle attribute is the angle vector – a normalized vector which allows multiplication by a constant magnitude to return a vector with the same magnitude in the desired direction. The rotation method is as follows:

*def* Rotate(*self*, scale, colliders):  
 *if* len(touchingany(*self*, colliders)) == 0:  
 scale \*= -1 *# We want to interpret + rotation as clockwise  
 self*.angle += PLAYER\_ROTATION\_SPEED \* scale  
 rotated\_image = Pygame.transform.rotate(*self*.image\_clean, *self*.angle)  
 *self*.image = rotated\_image  
 *if not self*.circular:  
 *self*.rect = *self*.image.get\_rect(topleft=*self*.rect.topleft)  
 *self*.angleDir.x, *self*.angleDir.y = -math.sin(*self*.angle \* RAD), math.cos(*self*.angle \* RAD)

PLAYER\_ROTATION\_SPEED is a constant degree value (usually 1o).

The angle vector is calculated using the python math module and the trigonometric functions included in it. The formula presented creates the proper angle vector for any rotation; the angle of rotation must be converted to radians since the module is programmed for that format.

Since the upright position is designated as Vec2(0, 1), and the fact that a negative angle should actually produce positive x-values, the x component is determined by -sin and the y determined by cos.

###### Mathematical Explanation

Chart, diagram

Description automatically generatedA Cartesian unit circle which visualises the trigonometric functions. Red = Radius (1), Blue = Opposite (sin), Yellow = Adjacent (cos)

The trigonometric functions return the vector (0,1) at 90o (pi/2 rad). In order to make them return this vector for our angle of “0 o” when facing upwards, the trig functions must be translated 90o ahead:

*self*.angleDir.x, *self*.angleDir.y = -math.cos((90 - *self*.angle) \* RAD), math.sin((90 - *self*.angle) \* RAD)

However, we can then use the identity that sin and cos reflected on the y axis and shifted 90o left simply equal the other –

*self*.angleDir.x, *self*.angleDir.y = -math.sin(*self*.angle \* RAD), math.cos(*self*.angle \* RAD)

Chart, line chart

Description automatically generatedWhich is why the x component is given by sin and y component given by cos, despite this being mathematically unusual. Sin becomes -sin because clockwise rotations must provide a positive x value, despite giving a negative angle:

A picture containing diagram

Description automatically generatedsin(x)



Chart, line chart

Description automatically generated

*fig. 1 -sin(x)*

For a negative acute angle, we want positive x; likewise, a positive acute angle must give us a negative number. The untransformed sin function (1st graph) will not produce this - we must negate (\*-1) the sin function to produce these results.

The result of the entire process will be an accurate vector of magnitude 1 which will produce the direction of the angle. Example:

In fig.1, the player posseses an angle of -39o to the vertical. His unit angle vector is then calculated using the given algorithm, producing a result of:



Chart

Description automatically generatedThis can then be confirmed on a unit circle. The unadjusted angle would be 90-39, so 51o:

The horizontal and vertical values found confirm the accuracy of the algorithm.

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Animations/Particles | Objects should always be drawn accurate to their positions and rotational value | A player at the position (25, 50) with an 87-degree clockwise rotation should be drawn at that position either at their centre or topleft with the appropriate rotation applied | The player sprite is drawn from the topleft at position (25, 50) rotated an angle -87o | **✓** |
| Worldcolliders should match the background at all times | Worldcollider should be present across the entire “floor” component of the background used in version 1 | The worldcollider is present and works as intended. | **✓** |
| Particles should recoil from the rear of the player ship when they accelerate. | Particles fly out of the back of the player which resemble engine fumes and should travel with a velocity opposite to the direction of the thrust. | Does not exist in version 1 | ✘ |
| Key objects should emit particles to symbolise their importance | Particles are released from special objects in a random fashion and should possess the same colour as their parent object. | Does not exist in version 1 | ✘ |
| Rotation should produce accurate vector results and lead to no translation across the screen | The player should be able to rotate -45o, keep the same position, and acquire an angle vector of (0.71, 0.71) | The player acquires an angle vector of (0.71, 0.71).  However, when the player rotates, its rect hitbox transforms around it. The hitbox’s position does not remain constant as it can extend further than usual by simply rotating. | ✘ |
| HUD | The player’s fuel and level timer should be displayed on screen. | A fuel bar and timer present somewhere on the screen, accurate to the desired readings. | Does not exist in version 1 | ✘ |
| An optional “details mode” available to the player to show detailed information about forces acting on objects | Text displayed somewhere on the screen to show the physical values possessed by the player and/or other objects in the scene | Does not exist in version 1 | ✘ |
| Working Camera | A camera functionality to track the player around the level | The player should be in view at all times and their position accurate in the level, so we can see what’s around them. | Does not exist in version 1 | ✘ |
| Camera movement should not produce unwanted movements in world objects; all objects should maintain positional consistency | The player should be able to move around the level without any objects moving out of their expected positions. | Does not exist in version 1 | ✘ |

### Version 2

#### Animations/Particles

##### Particles

The particle system is first introduced in Version 2. It is similar to other hierarchical class structures previously seen in this project, such that it consists of base classes (representing particles themselves) and a single global manager class.

Particles are very basic objects with limited lifespan. They simply track their position and velocity and move accordingly, drawing a desired shape/colour combination at the location on the screen. Their update method performs a basic movement procedure:

*def* Update(*self*, dt):  
 *self*.elapsed += dt  
 *self*.acceleration = Vec2(0, GRAVITY *if* GRAVITYON *and not self*.weightless *else* 0)  
 *self*.velocity += *self*.acceleration \* dt  
 *self*.pos += *self*.velocity \* dt  
 *self*.rect.center = *tuple*(Vec2(*self*.rect.center) + (*self*.velocity \* dt))

Note that particles can have “weight”. They do not actually possess a mass but can optionally be affected by gravity such that they experience the acceleration caused by it.

The particle handler calls every particle’s Update method, removes particles that should no longer exist and draws all the existing particles to the screen:

*def* Update(*self*, dt):  
 *for* i, particle *in enumerate*(*self*.particles):  
 particle.Update(dt)  
 *if* particle.elapsed >= particle.timer:  
 *self*.particles.pop(i)  
 *for* particle *in self*.particles:  
 particle.Draw()

###### Engine Particles

The engine particles generated by the player are a special particle class called “EngineParticle”. They inherit from the base particle class but are larger and possess a unique colour gradient functionality.

*def* Update(*self*, dt):  
 *super*().Update(dt)  
 *if self*.elapsed < *self*.timer:  
 frac = *self*.elapsed / *self*.timer  
 *self*.colour[1] = lINTerp(174, 255, frac)  
 *self*.colour[2] = lINTerp(0, 255, frac)  
 *self*.colour[3] = lINTerp(0, 255, frac)  
 *self*.radius = lINTerp(2, 5, frac)

The colour of engine particles starts at a dark yellow/orange and linearly interpolates into a white colour over its lifespan. This can be seen here:

A picture containing shape

Description automatically generatedLinear interpolation is a technique used to produce values within a given range produced by “entering” that range by a desired fractional amount; e.g, linearly interpolating by 0.5 between 4 and 8 will return 6. The lINTerp function is named as such because it returns the integer value of the interpolated result, which is required for colour values. The code is seen here:

*def* lINTerp(lb, ub, fraction):  
 *return int*(((ub - lb) \* fraction) + lb)

lb and ub stand for “lower-bound” and “upper-bound” respectively

A problem that must also be dealt with is the fact that engine particles must always originate from the “engine” of the player’s ship, and so this position must be tracked throughout all rotation and translational motion. Luckily, similar mathematics can be used for this as was used for rotation in version 1:

*self*.engine = *self*.GetPos() + Vec2(*self*.halfheight \* math.sin(*self*.angle \* RAD), *self*.halfheight \* math.cos(*self*.angle \* RAD))

self.halfheight represents half the pixel height of the player image

The standard sin and cos functions can be used since they apply themselves to the “halfheight” value to provide an offset towards the engine position. The algorithm can be seen at work using the debugging overlays:

A screenshot of a computer

Description automatically generated with low confidence Logo

Description automatically generated with medium confidence A picture containing text, device

Description automatically generated

The red dot at the bottom of the ship marks the “self.engine” position.

##### Rotation

Rotation in version 2 is centred, making all rotations much more consistent and easier to work with – for example when calculating the engine position. It also prevents so much unwanted translational motion when rotating. This was done by adding the following lines to the rotation method:

*def* Rotate(*self*, scale, colliders):  
 *if len*(touchingany(*self*, colliders)) == 0:  
 old\_rect = copy.deepcopy(*self*.rect)  
 scale \*= -1 *# We want to interpret + rotation as clockwise  
 self*.angle += PLAYER\_ROTATION\_SPEED \* scale  
 rotated\_image = Pygame.transform.rotate(*self*.image\_clean, *self*.angle)  
 *self*.image = rotated\_image  
 *self*.rect = *self*.image.get\_rect(center=old\_rect.center)  
 *self*.angleDir = Vec2(math.cos((90 + *self*.angle) \* RAD), -math.sin((90 - *self*.angle) \* RAD)).GetNormalized()

#### Camera Tracking

Since Pygame does not provide any in-game camera functionality, a rudimentary one must be constructed for the game. Instead of allowing the gameworld to exist statically (maintain a constant position on the coordinate grid) we must move the entire level around the player according to where the camera needs to be.

The camera functionality should track and display the player and the environment around them at all times. This includes maintaining the positional accuracy of all the objects that aren’t the player in the world.

The camera aims to centre on the player at all times, but when they move outside the boundaries of the screen the change in the level’s position must be calculated in order to move all the objects accordingly. This is done via the function GetCameraTrack, which returns the change in position required to be applied to all objects:

*def* getCameraTrack(pos, lpos, lwidth , lheight):x, y = pos.x, pos.y  
 swidth, sheight = WINDOW\_SIZE *# Screen width and height* halfw, halfh = swidth / 2, sheight / 2  
 maxWidthOffset = lwidth - swidth *# The maximum width and height the level can move before the image ends* maxHeightOffset = lheight - sheight  
 newlpos = *list*(lpos) *# New level position will be calculated and stored in a list  
  
 if* x + halfw > swidth *and* lpos[0] > -maxWidthOffset:  
 difference = x + halfw - swidth  
 newlpos[0] = lpos[0] - difference  
 *elif* x - halfw < 0 *and* lpos[0] < 0:  
 difference = *abs*(x - halfw)  
 newlpos[0] = lpos[0] + difference  
  
 *if* y - halfh < 0 *and* lpos[1] > -maxHeightOffset:  
 difference = y - halfh  
 newlpos[1] = lpos[1] + difference  
 *elif* y + halfh > sheight *and* lpos[1] < 0:  
 difference = y + halfh - sheight  
 newlpos[1] = lpos[1] + difference  
 *return* [*round*(a) *for* a *in* newlpos]

The parameters pos, lpos, lwidth, lheight refer to the player’s position, the level’s position, the level width, and level height respectively.

The algorithm checks at the start of every frame whether the player’s position + half the screen width or height exceeds the borders of the screen, and whether there is still more level to be seen (hence the maxWidthOffset and maxHeightOffset). After this, the difference needed to recentre the player is calculated and placed into an array before being returned to the function caller. This delta value can then be used to move any objects by the required amount around the player:

lPos = getCameraTrack(player.GetPos(), lPos, background\_image.get\_size()[0], background\_image.get\_size()[1])  
  
  
*# move game objects accordingly with the level*diff = *list*(numpy.subtract(lPos, oldLPos)) *# convert the numpy array to a regular list*player.SetPos(player.GetPos() + Vec2(diff)) *# PhysObjects can be moved via vector addition  
for* object *in* objects:  
 object.SetPos(object.GetPos() + Vec2(diff))  
*for* particle *in* particleHandler.particles:  
 particle.SetPos(particle.GetPos() + Vec2(diff))  
*for* wc *in* world: *# Worldcolliders are tracked by rects only, so use numpy list subtraction* wc.Move(diff)

Every object in the world that has a hitbox needs to be moved in accordance with the level background and the camera.

A picture containing chart

Description automatically generated

Parts of the level are viewable by the player that weren’t before, e.g the ramps can now be seen. It should also be noted that the player is at the centre of the screen when not at the edges of the level. The debug overlay is enabled to show that the objects are moved with the level.

A notable feature of this function is that when the player is at the edge of the level, the centering mechanic does not apply (the player’s position is “unlocked” on the screen). This can be seen here:

Text

Description automatically generated

Position of the camera when the player reaches the furthest right edge of the level.

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Animations/Particles | Rotation should produce accurate vector results and lead to no translation across the screen | The player should be able to rotate -45o, keep the same position, and acquire an angle vector of (0.71, 0.71) | The player acquires an angle vector of (0.71, 0.71), and experiences no translation. | **✓** |
| Particles should recoil from the rear of the player ship when they accelerate. | Particles fly out of the back of the player which resemble engine fumes and should travel with a velocity opposite to the direction of the thrust. | Engine particles are spawned and behave as expected. | **✓** |
| Key objects should emit particles to symbolise their importance | Particles are released from special objects in a random fashion and should possess the same colour as their parent object. | Does not exist in version 2 | ✘ |
| HUD | The player’s fuel and level timer should be displayed on screen. | A fuel bar and timer present somewhere on the screen, accurate to the desired readings. | Does not exist in version 2 | ✘ |
| An optional “details mode” available to the player to show detailed information about forces acting on objects | Text displayed somewhere on the screen to show the physical values possessed by the player and/or other objects in the scene | Does not exist in version 2 | ✘ |
| Working Camera | A camera functionality to track the player around the level | The player should be in view at all times and their position accurate in the level, so we can see what’s around them. | The camera properly tracks the player around the level provided in version 1 and 2. | **✓** |
| Camera movement should not produce unwanted movements in world objects; all objects should maintain positional consistency | The player should be able to move around the level without any objects moving out of their expected positions. | The camera moves objects accurately in the level provided in version 1 and 2, but several bugs were discovered when changing the level to one with more height than the screen width. WorldColliders would not move correctly, objects would collide haphazardly with worldcolliders while being moved, and the camera did not initiate in the correct place when the 0,0 point on the level was not where the player actually started. | ✘ |

### Version 3

#### Animations/Particles

##### Particles

The ParticleHandler’s functionality is slightly expanded upon in version 3, where the “Emit” method is created. This allows various objects in the gameworld to emit their passive particle effects, a behaviour I had long intended to add to emphasise the importance of chosen objects, such as those related to objective completion. It can also be used in other ways, e.g., to show a stream of air particles.

*def* Emit(*self*, obj, colour, life, velocity, weightless=*False*, colSim=*False*, parent=*None*):  
 pos, rect = *tuple*(obj.GetPos()), obj.GetRect()  
 pos = rect.center *if* pos != rect.center *else* pos  
 x = random.randint(*int*(pos[0] - (rect.width / 2)), *int*(pos[0] + (rect.width / 2)))  
 y = random.randint(*int*(pos[1] - (rect.height / 2)), *int*(pos[1] + (rect.height / 2)))  
  
 *self*.Add(Particle(Vec2(x, y), velocity, life, weightless, colour, 2, colSim, parent))

The code spawns particles in random spots within the object’s rectangle, with the desired velocity provided in the parameters.

A picture containing outdoor object, star, night sky

Description automatically generated

White dots representing air particles being pushed to the right from the AirStream source (left)

A picture containing text, electronics

Description automatically generatedParticles being emitted from a goal object (green), as well as a special level completion object (purple)

##### Rotation

It was discovered that the rotation mechanic changed speed depending on the player’s framerate (the amount of rotation per second was not framerate-independent). This was easily fixed by adjusting the code to account for the user’s frametimes:

*def* Rotate(*self*, scale, colliders, dt):  
 *if len*(touchingany(*self*, colliders)) == 0:  
 old\_rect = copy.deepcopy(*self*.rect)  
 scale \*= -1 *# We want to interpret + rotation as clockwise  
 self*.angle += PLAYER\_ROTATION\_SPEED \* scale \* dt  
 ...

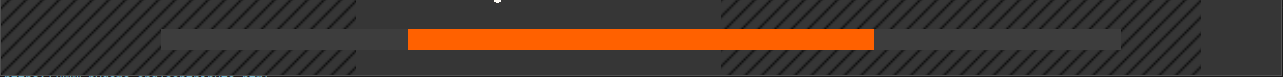
#### HUD

##### Fuel

The HUD is finally implemented and populated in version 3. Firstly, the fuel system is added, and the player’s fuel is displayed at the bottom of the screen:



The fuel bar when its nearly full



The fuel bar at around 50% fuel



The fuel bar nearly empty

The effect demonstrated here clearly resembles that which was used to colour the engine particles of the ship, and indeed uses the same linear interpolation technique. As the fuel gets emptier, the fuel bar is coloured “redder” from its starting orange/yellow appearance.

fuelBackgroundRect = Pygame.Rect(0, 0, *int*(0.75 \* swidth), *int*(0.03 \* sheight))  
fuelRect = Pygame.Rect(0, 0, *int*(0.75 \* swidth \* *self*.player.fuel / *self*.player.tank), *int*(0.03 \* sheight))  
  
fuelBackgroundRect.center, fuelRect.center = (swidth // 2, *int*(0.95 \* sheight)), (swidth // 2, *int*(0.95 \* sheight))  
Pygame.draw.rect(screen, NEARLYBLACK, fuelBackgroundRect)  
Pygame.draw.rect(screen, (255, lINTerp(0, 200, *self*.player.fuel / *self*.player.tank), 0), fuelRect)

The highlighted area is where the green component of the colour is linearly interpolated to counteract the red component valued at a full 255, depending on how much fuel the player has. As the player’s fuel emptied, the linearly interpolated green value would tend to 0.

##### Details Mode

The details mode is implemented as hovering boxes listing important physics values for every PhysObject currently in the level. This can be enabled by pressing the appropriate key, usually TAB.

Text

Description automatically generated with medium confidence

The functionality heavily utilises the Vec2 string-casting override:

*def \_\_str\_\_*(*self*):  
 *return* f"{*str*(*self*.x)}, {*str*(*self*.y)}"

The Vec2 string casting method

*def* DrawDetails(*self*, surface):  
 details = [  
 f"Mass: {*self*.mass} kg",  
 f"Velocity: ({*str*(*round*(*self*.velocity, 1))}) m/s",  
 f"Acceleration: ({*str*(*round*(*self*.acceleration, 1))}) m/s²",  
 f"Resultant Force: ({*str*(*round*(*self*.rForce, 1))}) N",  
 "Forces:"  
 ]  
 *for* force *in self*.forces.forces:  
 details.append(f" {force.name}: ({*str*(*round*(force, 1))}) N")  
 *if isinstance*(*self*, Player):  
 details = [f"Engine Drive: {*self*.thrust} N"] + details

The first half of the method is dedicated to adding strings to a list of each physics value we intend to display on the HUD. The final line only applies to the player, wherein the first value in their details box will be the magnitude of the player’s engine thrust – specific to the level they are currently in.

fontSize = tinyFont.size("a")  
rectHeight = (*len*(details) \* fontSize[1]) + 10  
rectWidth = 200  
detailsRect = Pygame.Rect(0, 0, rectWidth, rectHeight)

We must then find out the height of the font in use in order to calculate the required height of our details box. The width is fixed to stop the box from constantly changing shape – highly distracting during gameplay.

detailsRect.left = *self*.rect.right  
detailsRect.centery = *self*.rect.centery  
Pygame.draw.rect(surface, NEARLYBLACK, detailsRect, 0, 7)  
Pygame.draw.rect(surface, BLACK, detailsRect, 3, 7)

The box is then positioned correctly and drawn next to the object.

Following this, the physics details themselves are blit onto the screen in order from the list, positioned according to their index within the list, plus a small offset in both axes to make it fit entirely within the box.

*for* i, detail *in enumerate*(details):  
 textRender(tinyFont, (detailsRect.topleft[0] + 8, detailsRect.topleft[1] + 4 + (fontSize[1] \* i)), detail,  
 WHITE, *False*)

##### Timer

The timer class has also been added to the game. It is a straightforward component which utilises the time module, and simply tracks the time spent by the player on the level from when they begin. It displays this time in minutes and seconds at the top left of the screen.

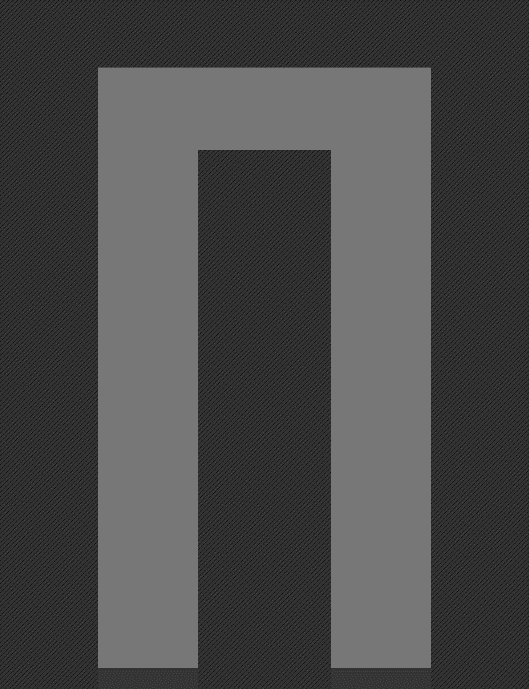
*class* Timer:  
 *def init*(*self*, pos, active=*True*):  
 *self*.pos = pos  
 *self*.epoch = time.time()  
 *self*.formattedTime = ""  
 *self*.elapsed = 0  
 *self*.active = active  
 *def* Update(*self*):  
 *if self*.active:  
 *self*.elapsed = time.time() - *self*.epoch  
 minutes = *int*(*self*.elapsed // 60)  
 secondsRemaining = *int*(*self*.elapsed - minutes \* 60)  
 *self*.formattedTime = f"{'0' *if* minutes < 10 *else* ''}{*str*(minutes)}:{'0' *if* secondsRemaining < 10 *else* ''}{*str*(secondsRemaining)}"

Most of the logic in the Update method is dedicated to correctly formatting the time for display.

#### Camera Tracking

##### Tracking Logic

As shown in Version 2’s testing, a multitude of bugs arose as a result of the camera tracking code, all being revealed when a new level was added in version 3:

The new level background

The first bug was a blatant error in the logic of the GetCameraTrack function:

*if* y - halfh < 0 *and* lpos[1] > -maxHeightOffset:  
 difference = y - halfh  
 newlpos[1] = lpos[1] + difference  
*elif* y + halfh > sheight *and* lpos[1] < 0:  
 difference = y + halfh - sheight  
 newlpos[1] = lpos[1] + difference

The highlighted lines contained various logic errors which were overlooked due to the old test level not having a height greater than the screen height (as a result, the camera would never have to move vertically, and the logic errors never revealed themselves in the game).

These were quickly fixed when comparing the code to the equivalent code for the x values:

*if* y - halfh < 0 *and* lpos[1] < 0:  
 difference = *abs*(y - halfh)  
 newlpos[1] = lpos[1] + difference  
*elif* y + halfh > sheight *and* lpos[1] > -maxHeightOffset:  
 difference = y + halfh - sheight  
 newlpos[1] = lpos[1] - difference

##### Initiating the Camera

The next problem arose due to the level not having the player’s start position within screen-sized view of the (0,0) position on the level background.



(In orange) The perceived position and view of the “camera”.

(In red) Where the player spawns in the level.

On this background the problem was not present, as the player happened to spawn in a part of the level which the screen already covered, as well as being off-centred while the screen was at the edge of the level. All these conditions were unrealistically perfect for the camera, so hid its underlying lack of functionality.

The new level uncovered this because of the layout it now created: A picture containing brick, building material

Description automatically generated

The new level is far bigger in size, hence the smaller boxes.

This ended up being a problem, as it now made the level unplayable since the camera no longer knew how to behave nor was it positioned in the correct area at all. The level background would load out of the boundaries of the screen and trying to move the player around would cause it to break even more, creating stuttering and random movement of the level.

In order to account for this, the level’s instantiation had to include a procedure for relocating itself around the player, no matter where they may have started on the level. A new branching statement was added to the GetCameraTrack function to implement this:

*if not* Rect(0, 0, swidth, sheight).contains(playerRect):  
 ideal = Vec2(*max*(x - halfw, 0), *max*(y - halfh, 0))  
 levelRect = Rect(0, 0, lwidth, lheight)  
 idealBottomRight = ideal + Vec2(swidth, sheight)  
 diff = idealBottomRight - Vec2(levelRect.bottomright)  
 diff.x, diff.y = *max*(diff.x, 0), *max*(diff.y, 0)  
 ideal = ideal - diff  
 *return* [*round*(a) *for* a *in* ideal.Inverse()]

The algorithm worked like this:

1. Find the ideal position for the topleft of the screen (since the level’s position was at its topleft) for the current position of the player. This would be where we *want* the topleft of the rectangle to be, but if it ends up too far to the left it will end up being outside the left-most or top-most borders of the level. As such, we choose the maximum between that position and (0,0).
2. Next check where the bottomright of the rectangle would be if we chose this position.
3. If this bottomright corner exists outside of the level, perform some arithmetic to find out how far we have to move the screen to get it back within the necessary confines.
4. Add this difference to the previously “ideal” position.
5. Invert the “ideal” position.

Since the camera starts at (0,0), any position vectors are actually just translation vectors with origin (0,0). This is a mathematical convenience that makes our algorithm simply allow us to return the position and actually interpret it as a “delta” as is usually returned by this function. It should also be remembered that we are moving the level, not the “camera”, so the vector must be inverted, since we are dragging the level upwards/left rather than a “camera” downwards.

##### Maintaining collision consistency

At first, the new solution appeared to work perfectly for adjusting the initial position of the level. However, it was quickly discovered using the debug overlay that certain WorldColliders and physobjects were interacting when the camera moved around, and were being permanently displaced, teleporting to random sides of objects due to their velocities tricking the collision system and moving up and down or side to side as the camera did. These were fixed through some adjustments to the movement algorithm and making use of the “SafeMove” function described earlier (page 40):

diff = Vec2(*list*(numpy.subtract(*self*.lPos, oldLPos))) *# convert the numpy array to a regular list and then to a Vec2  
  
if not* Rect(0, 0, swidth, sheight).contains(player.GetRect()):  
 player.SetPos(player.GetPos() + diff)  
 *for* object *in* objects:  
 object.SetPos(object.GetPos() + diff)  
 *for* particle *in* particleHandler.particles:  
 particle.SetPos(particle.GetPos() + diff)  
 *for* wc *in* world:  
 wc.Move(diff)  
*else*:  
 CollisionHandler.SafeMove(player, world, diff)  
 *for* wc *in* world:  
 wc.Move(diff)  
  
 *for* object *in* objects:  
 CollisionHandler.SafeMove(object, world, diff)  
  
 *for* particle *in* particleHandler.particles:  
 *#CollisionHandler.SafeMove(particle, world, diff)* particle.SetPos(particle.GetPos() + diff)

The initial camera movement (in the if branch) is conducted by unfiltered movements since otherwise the level can move arbritrarily and this leads to unpredictable results. Furthermore, collision detection doesn’t matter before the game has started.

Admittedly, the second part of the algorithm was devised through simple trial and error, and this order of operations was found to be the most optimal. It was also found that worldcolliders should not use the “SafeMove” function, as they are likely to be displaced when colliding with lots of moving PhysObjects.

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| HUD | The player’s fuel and level timer should be displayed on screen. | A fuel bar and timer present somewhere on the screen, accurate to the desired readings. | The fuel bar is present at the bottom centre of the screen and functions accurately. | **✓** |
| An optional “details mode” available to the player to show detailed information about forces acting on objects | Text displayed somewhere on the screen to show the physical values possessed by the player and/or other objects in the scene | When pressing TAB, a box appears next to all physics objects displaying their physics data accurate to 1 decimal place. In addition, the player’s detail box displays the magnitude of their engine thrust. | **✓** |
| Key objects should emit particles to symbolise their importance | Particles are released from special objects in a random fashion and should possess the same colour as their parent object. | Special objects (goal objects and their object counterparts) behave as expected and emit coloured particles at all times. | **✓** |
| Working Camera | A camera functionality to track the player around the level | The player should be in view at all times and their position accurate in the level, so we can see what’s around them. | The camera properly tracks the player around the level in all types of levels. | **✓** |
| Camera movement should not produce unwanted movements in world objects; all objects should maintain positional consistency | The player should be able to move around the level without any objects moving out of their expected positions. | The camera moves objects accurately in all types of levels, without any adverse effects. All collisions are maintained. | **✓** |

## Level Design

### Version 1

#### Level Loading

Levels were only separated and implemented at a late stage in development, and as such there is very little change in the loading algorithm itself between each version. Levels are loaded through the function level\_load which reads the group of files associated with each level, assorting them into a single file structure (a dictionary) for use when initiating the Game object.

In version 1, only 3 files are associated with each level. These are the background itself as an image (in png format), along with two csv files: “world.csv” and “objects.csv”. These contain spawn data for WorldColliders and PhysObjects respectively.

The CSV files are formatted so that their data values are ordered the same as their associated parameters in the code. The objects.csv treats the first line of its content as the values for spawning the player:

50,50,100   
100,100,assets/sprites/ball.png,150,0.5,TRUE,0.35

Contents of the objects.csv file

*class* Player(PhysObject):  
 *def init*(*self*, pos, image, mass):

The player init parameters (first line of the csv)

*class* PhysObject:  
 *def init*(*self*, pos, image, mass, Cd=0.5, circular=*False*, COR=0):

PhysObject init parameters (second line of the csv)

As such, all the algorithm needs to do is read the CSV as-is and convert the parameters to their appropriate data types and organise the data into the runtime data structure.

*def* level\_load(level):  
 info = {  
 "background": os.path.join("levels", level, "background.png"),  
 "world": [],  
 "objects": [],  
 "player": *None* }  
 *with* open(os.path.join("levels", level, "world.csv"), "r") *as* file:  
 reader = csv.reader(file)  
 *for* row *in* reader:  
 objInfo = list(map(int, row)) *# Convert all coordinate values for the rect into integers* info["world"] = info["world"] + [WorldCollider(Pygame.Rect(objInfo[0], objInfo[1], objInfo[2], objInfo[3]))]  
 *with* open(os.path.join("levels", level, "objects.csv"), "r") *as* file:  
 reader = csv.reader(file)  
 *for* i, row *in* enumerate(reader):  
 pos = tuple(map(int, row[0:2]))  
 *if* i == 0:  
 info["player"] = Player(pos, player\_image, int(row[2]))  
 *else*:  
 physInfo = list(map(float, row[3:5])) + [bool(row[5])] + [float(row[6])] *# Convert all physInfo to floats and bool types* info["objects"] = info["objects"] + [PhysObject(pos, Pygame.image.load(row[2]).convert\_alpha(), physInfo[0], physInfo[1], physInfo[2], physInfo[3])]  
 *return* info

The level\_load algorithm

The map function is used extensively to convert consecutive items with the same data type into that data type (all data read from a csv is initially a string).

As shown, the function collects all the data into four keys in the dictionary, and returns this data to the function caller to be unpacked and used as the initiation parameters for a new Game object:

gameData = level\_load("test")  
world, objects, player = gameData["world"], gameData["objects"], gameData["player"]  
*self*.state.newstate(Game(*self*.state, world, objects, player))

How the Menu object uses the data it receives from level\_load

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Level Loading | Different levels should be able to be loaded along with their game object contents. | A level should be able to load with its background, a player object, at least one PhysObject and one worldcollider | A level is loaded with a single player object, worldcollider and physobject, aligned with the desired position on the background of the level. | **✓** |
| Objectives | Objectives should detect touching by their activating object. | A physobject landing on a Physics objective should cause it to return True/activate itself. | Does not exist in Version 1 | ✘ |
| Objectives should light up/make it obvious when its activated | An activated objective should change colour | Does not exist in Version 1 | ✘ |
| When all objectives are met, the level should end, and the player should be given a score | Meeting all objectives but taking double the optimal time should both end the level and reward the player with half the score for completing it in the optimal time. | Does not exist in Version 1 | ✘ |
| Environmental Hazards | AirStreams should apply a force to objects only while they are within their zone. | A player object and physobject should both be pushed with the same force in the same direction as the airstream. | Does not exist in Version 1 | ✘ |
| Death boxes should immediately end and fail the level if the player touches them | A player driving into a death box should cause the level to end and receive 0 score | Does not exist in Version 1 | ✘ |
| Colliding with walls too fast should cause a penalty to the player’s final score | A player driving into a vertical wall with horizontal velocity >10m/s should reduce the player’s final score by a fixed amount. | Does not exist in Version 1 | ✘ |

### Version 2

#### Objectives

##### Objective Blocks

The objectives system was added in version 2 to implement goals for the completion of levels. These exist in the form of Physics Objectives and Player Objectives; both require something to touch the objective block, a physics object, or the player respectively. Since they behave so similarly, a parent Objective class (itself a WorldCollider subclass) is created which both types of objectives inherit their behaviours from:

Every frame, an objective class checks for collisions/touching with each object provided to it in a list of colliders. If at least one detection occurs, the objective is considered activated, and it should change colour.

*def* Update(*self*):  
 *for* trigger *in self*.triggers:  
 *if* trigger.GetRect().colliderect(*self*.rect) *or* touching(trigger, *self*):  
 *self*.colour = GREEN  
 *return True  
 self*.colour = *self*.original  
 *return False*

The update method for objectives

The physics and player objective subclasses simply change the colour of the objective block to allow the player to differentiate between them. The PlayerObjective class also passes the player to the super.init() method within a list, as the parent class requires it in such a format.

Graphical user interface, application

Description automatically generated*class* PhysObjective(Objective):  
 *def \_\_init\_\_*(*self*, pos, width, height, obj):  
 *super*().*\_\_init\_\_*(pos, width, height, obj)  
 *self*.original, *self*.colour = MAGENTA, MAGENTA

A PhysObjective

*class* PlayerObjective(Objective):  
 *def \_\_init\_\_*(*self*, pos, width, height, player):  
 *super*().*\_\_init\_\_*(pos, width, height, [player])  
 *self*.original, *self*.colour = YELLOW, YELLOW

Notice the [player] parameter – triggers is expected as a list, so despite having only the player as a trigger we must maintain this type consistency to prevent logic/syntax errors.

A picture containing text

Description automatically generatedA PlayerObjective

##### Key Objects

Paired with objective blocks are key objects, which can activate any PhysObjective in the world. Naturally, these are subclasses of PhysObjects which possess special features like bright colours and particle emission.

*class* KeyObject(PhysObject):  
 *def \_\_init\_\_*(*self*, pos, image, mass, colour, Cd=0.5, COR=0):  
 *super*().*\_\_init\_\_*(pos, image, mass, Cd, COR)  
 *self*.colour = colour  
 *self*.lastEmission = time.time()

A picture containing text, silhouette

Description automatically generatedAs can be seen here, there is not much to them except for new attributes relating to their graphical features.

#### Level Loading

The level loading remained largely the same, but accomadated the new Objective features and separated the player’s spawn data from other physobjects data. As such, two new files were added to all levels: player.csv and objectives.csv.

Objectives.csv contains a special column used to determine whether the data in a row relates to the spawning of an Objective block or a Key Object:

OBJECT,600,100,assets/sprites/pinkcube.png,75,181,49,201,0.5,0.1  
PHYS,550,200,100,10

Player.csv simply contains one row for the player spawn data:

50,50,100,500,3000,TRUE

New algorithms had to be added for both new files, the most substantial one being for the objective.csv file:

*## LOADING LEVEL OBJECTIVES ##  
with open*(os.path.join("levels", level, "objectives.csv"), "r") *as* file:  
 reader = csv.reader(file)  
 *for* row *in* reader:  
 *if* row[0] == "PLAYER":  
 objInfo = *list*(*map*(*int*, row[1:]))  
 info["objectives"] = info["objectives"] + [PlayerObjective(Vec2(objInfo[0], objInfo[1]), objInfo[2], objInfo[3], info["player"])]  
 *elif* row[0] == "PHYS":  
 objInfo = *list*(*map*(*int*, row[1:]))  
 info["objectives"] = info["objectives"] + [PhysObjective(Vec2(objInfo[0], objInfo[1]), objInfo[2], objInfo[3], [x *for* x *in* info["objects"] *if isinstance*(x, KeyObject)])]  
 *elif* row[0] == "OBJECT":  
 pos = *tuple*(*map*(*int*, row[1:3]))  
 conv = *list*(*map*(*float*, row[4:]))  
 colour = *tuple*(conv[1:4])  
 info["objects"] = info["objects"] + [KeyObject(pos, Pygame.image.load(row[3]).convert\_alpha(), conv[0], colour, conv[-1], conv[-2])]

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Level Loading | Different levels should be able to be loaded along with their game object contents. | A level should be able to load with its background, a player object, at least one PhysObject, a worldcollider, a key object and a PhysObjective | A level is loaded with a single player object, worldcollider, physobject, a PhysObjective and a Key Object hovering above it - all aligned with the desired position on the background of the level. | **✓** |
| Objectives | Objectives should detect touching by their activating object. | A physobject landing on a Physics objective should cause it to return True/activate itself. | After adding a line of code to print what the objective block returns each frame, a key object touching it causes the console to flood with “True”, proving its activation.  If the Key Object’s spawn data is written after any Objective spawn data, those Objectives will not detect the Key Object. This is because the key object in question did not exist while collecting the triggers for the Objectives, as such it will not be included in its detection filter. | ✘ |
| Objectives should light up/make it obvious when its activated | An activated objective should change colour | When the bug is avoided, the Objective changes colour when detecting the Key Object in the previous test. | **✓** |
| When all objectives are met, the level should end, and the player should be given a score | Meeting all objectives but taking double the optimal time should both end the level and reward the player with half the score for completing it in the optimal time. | Does not exist in Version 2 | ✘ |
| Environmental Hazards | AirStreams should apply a force to objects only while they are within their zone. | A player object and physobject should both be pushed with the same force in the same direction as the airstream. | Does not exist in Version 2 | ✘ |
| Death boxes should immediately end and fail the level if the player touches them | A player driving into a death box should cause the level to end and receive 0 score | Does not exist in Version 2 | ✘ |
| Colliding with walls too fast should cause a penalty to the player’s final score | A player driving into a vertical wall with horizontal velocity >10m/s should reduce the player’s final score by a fixed amount. | Does not exist in Version 2 | ✘ |

### Version 3

#### Environmental Hazards

##### Airstreams

One of the environmental hazards introduced in Version 3 are Airstream objects. These are map hazards which “push air” in a desired direction continuously. In game terms, they constantly apply a fixed force to any Physics-obeying objects within their zone of action (any PhysObject subclass e.g the Player).

They are instantiated with their source block dimensions as well as the dimensions of the stream itself which they control, and the force it should apply.

*class* AirStream(WorldCollider):  
 *def \_\_init\_\_*(*self*, pos, width, height, streamWidth, streamHeight, force):  
 *super*().*\_\_init\_\_*(Pygame.Rect(pos.x, pos.y, width, height), material="Steel")  
 *self*.colour = STEEL  
 *self*.force = force  
 *self*.lastEmission = time.time()  
 *self*.oldPos = copy.deepcopy(*self*.pos)  
  
 *if* streamWidth != 0:  
 *if* streamWidth > 0:  
 *self*.streamRect = Pygame.Rect(\**self*.rect.topright, streamWidth,

height)  
 *else*:  
 *self*.streamRect = Pygame.Rect(pos.x + streamWidth, pos.y, streamWidth, height)  
 *else*:  
 *if* streamHeight > 0:  
 *self*.streamRect = Pygame.Rect(\**self*.rect.bottomleft, width, streamHeight)  
 *else*:  
 *self*.streamRect = Pygame.Rect(pos.x, pos.y + streamHeight, width, streamHeight)

A picture containing outdoor object, star, night sky

Description automatically generatedThe highlighted part of the code snippet shows some logic within the init method. This is used to create the stream’s rectangle – either streamWidth or streamHeight should be equal to 0 while the other has a value. Using this information, we can determine which side the stream zone should start from, and the other value will simply take on the maximum it can for that side. An example:

The AirStream Source (left, in orange) has been instantiated with a streamWidth of 1175 and streamHeight 0. As a result, the algorithm has determined that the stream (right, in black) should be placed on the right-hand side of the source rectangle, since the width is greater than 0. It also automatically makes the height of the stream’s rectangle equal to the height of the source block itself. If the width was negative, it would be placed on the left-hand side. This also applies to vertical airstreams, however positive heights will go downwards.

Every frame, the AirStream object checks all PhysObjects for whether they are in its stream zone and apply a “Wind” force accordingly. Likewise, it will attempt to remove a “Wind” force (of its own making) from any objects which should not possess it. Keep in mind that the RemoveForce method is null-safe, such that it will not throw an exception if the force in question cannot be found. This is useful for avoiding unnecessary checks on every object:

*def* Update(*self*, objects):  
 *for* obj *in* objects:  
 *if self*.streamRect.contains(obj.GetRect()) *or self*.streamRect.colliderect(obj.GetRect()):  
 obj.AddForce(*self*, "Wind", *self*.force)  
 *else*:  
 obj.RemoveForce(*self*, "Wind")

The second half of the update method concerns the position of the streamRect, specifically ensuring that the stream always maintains the same relative position to the source block, no matter where it moves:

*if self*.pos - *self*.oldPos != Vec2(0, 0): *# check if the source was moved and move the airstream accordingly  
 self*.streamRect.topleft = *tuple*(Vec2(*self*.streamRect.topleft) + (*self*.pos *self*.oldPos))   
 *self*.oldPos = copy.deepcopy(*self*.pos)

##### Death Boxes

Death boxes are red obstacles placed around the level to increase the difficulty of traversal and add complexity to levels which would otherwise be far too easy. These are simple worldcollider subclasses which cause a level to fail if the player collides with them. As such, their code is very basic:

*class* Obstacle(WorldCollider):  
 *def init*(*self*, pos, width, height, player):  
 *super*().*\_\_init\_\_*(Pygame.Rect(pos.x, pos.y, width, height), material="Asphalt")  
 *self*.colour = RED  
 *self*.player = player  
 *def* Update(*self*, player):  
 *if self*.player.GetRect().colliderect(*self*.rect) *or* touching(*self*.player, *self*):  
 *return True  
 return False*

A picture containing shape

Description automatically generated

Example of a death box

The return value from their Update method is captured in the main game loop to cause a game-over:

*for* obstacle *in* obstacles:  
 *if* obstacle.Update(player):  
 *self*.state.newstate(Menu(*self*.state))  
 obstacle.Draw(screen)

##### Ship Damage

“Ship Damage” is added as a passive hazard in the game. The player records a damaging collision whenever it collides with a worldcollider at a greater velocity than the threshold value determined in constants.py.

*if* bounceAxis["x"]:  
 *if isinstance*(*self*, Player):  
 *if abs*(*self*.velocity.x) >= DAMAGETHRESHOLD:  
 *self*.collisions += 1

*if* bounceAxis["y"]:  
 *if isinstance*(*self*, Player):  
 *if abs*(*self*.velocity.y) >= DAMAGETHRESHOLD:  
 *self*.collisions += 1

These collisions are penalised in the player’s final score by a fixed amount – HITPENALTY in constants.py.

#### Objectives

Scoring and a Scoring Screen are now present in version 3. All levels are given an optimal time to be completed in – completing the level under this time will provide extra points over a BASESCORE (usually 10000) up to double the score. Taking longer than this time will reduce the points below the BASESCORE, both proportionally to how much longer or shorter the player takes:

*class* ScoringScreen:  
 *def \_\_init\_\_*(*self*, stateobj, objectives, optimal, time, collisions):  
 *self*.state = stateobj  
 *self*.totalobj = *len*(objectives)  
 *self*.objmet = *len*([x *for* x *in* objectives *if* x.complete])  
 *self*.optimal = optimal  
 *self*.time = time  
 *self*.collisions = collisions  
  
 timeDiff = optimal - time *# This will be negative if the player took longer than the optimal* timeMult = *min*(*abs*(timeDiff) / optimal, 1) *# Only penalise/bonus for up to double the time and down to 0 seconds* timeMult \*= -1 *if* timeDiff < 0 *else* 1 *# Bonus or penalty  
 self*.score = *int*(SCOREBASE + (SCOREBASE\*timeMult) - (HITPENALTY \* collisions))  
 *self*.score = 1 *if self*.score == 0 *else self*.score

#### Level Loading

Finally in Version 3, the level loading algorithm is again expanded (to its fullest extent) to accommodate all the possible features of a level. Changes in this version include processing spawn data for the new airstream and death box objects, stored in hazards.csv and obstacles.csv respectively. As with most other level objects, they are processed and stored according to their parameters:

*## LOADING OBSTACLES ##  
with open*(os.path.join("levels", level, "obstacles.csv"), "r") *as* file:  
 reader = csv.reader(file)  
 *for* row *in* reader:  
 objInfo = *list*(*map*(*int*, row))pos = Vec2(objInfo[0], objInfo[1])  
 info["obstacles"] = info["obstacles"] + [Obstacle(pos, objInfo[2], objInfo[3], info["player"])]  
  
*## LOADING HAZARDS (AIRSTREAMS) ##  
with open*(os.path.join("levels", level, "hazards.csv"), "r") *as* file:  
 reader = csv.reader(file)  
 *for* row *in* reader:  
 objInfo = *list*(*map*(*int*, row))pos = Vec2(objInfo[0], objInfo[1])  
 force = Vec2(objInfo[-2], objInfo[-1])  
 info["hazards"] = info["hazards"] + [AirStream(pos, objInfo[2], objInfo[3], objInfo[4], objInfo[5], force)]

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Level Loading | Different levels should be able to be loaded along with their game object contents. | A level should be able to load with its background, a player object, at least one PhysObject, a worldcollider, a key object and a PhysObjective | A level is loaded with a single player object, worldcollider, physobject, a PhysObjective and a Key Object hovering above it - all aligned with the desired position on the background of the level. | **✓** |
| Objectives | Objectives should detect touching by their activating object. | A physobject landing on a Physics objective should cause it to return True/activate itself. | After adding a line of code to print what the objective block returns each frame, a key object touching it causes the console to flood with “True”, proving its activation.  Triggers are passed every frame rather than being stored as an attribute of the objective, preventing the bug from Version 2’s test. | **✓** |
| Objectives should light up/make it obvious when its activated | An activated objective should change colour | When the bug is avoided, the Objective changes colour when detecting the Key Object in the previous test. | **✓** |
| When all objectives are met, the level should end, and the player should be given a score | Meeting all objectives but taking double the optimal time should both end the level and reward the player with half the score for completing it in the optimal time. | When all objectives are met, a player taking 37 seconds to complete a level with an optimal time of 30 seconds, after experiencing 3 damaging collisions, received a score of 10000 – (10000\*(7/30)) – (500 \* 3) = 6166 {floored int} | **✓** |
| Environmental Hazards | AirStreams should apply a force to objects only while they are within their zone. | A player object and physobject should both be pushed with the same force in the same direction as the airstream. | A player and a PhysObject within an AirStream applying a force of (1200, 0) N acquire a “Wind” Force of (1200, 0) N. | **✓** |
| Death boxes should immediately end and fail the level if the player touches them | A player driving into a death box should cause the level to end and receive 0 score | When the player touches a death box, the level ends. | **✓** |
| Colliding with walls too fast should cause a penalty to the player’s final score | A player driving into a vertical wall with horizontal velocity >10m/s should reduce the player’s final score by a fixed amount. | A player with a velocity of (12.55, 8.5394)m/s colliding with a vertical wall while DAMAGETHRESHOLD = 10 increments the player’s # of collisions by 1. | **✓** |

## Menu

### Version 1

#### Basic Menu

##### The States System

The menu feature was added midway through development, and with it came the “State” system, used to allow the game to switch between different sections of the program. For example, the Menu’s play button induces a state switch from a Menu state class to a Game state class.

The state class itself is a class which, through polymorphism, can keep track of what section of the game should be active and induce its behaviours:

*class* State:  
 *def init*(*self*, newstate):  
 *self*.state = newstate  
 *def* newstate(*self*, newstate):  
 *self*.state = newstate  
 *def* RunFrame(*self*, dt):  
 *self*.state.RunFrame(dt)

Different parts of the game (such as the Menu class or Game class) can be stored in the self.state attribute of a State object and are not subclasses of State itself. All of these parts have their own RunFrame methods, which allows the State class to induce their behaviour. As such, the outermost game-loop is now reduced to just a few lines, since it simply has to call RunFrame for the State object its using, and that will run the correct method for whatever section of the game we are in:

state = State(*None*)  
menu = Menu(state)  
state.newstate(menu)  
  
prev\_time = time.time()  
*while True*:  
 clock.tick()  
 now = time.time()  
 dt = now - prev\_time  
 prev\_time = now  
  
 state.RunFrame(dt)  
  
 Pygame.display.update()  
 clock.tick(FPS)

All these different ‘states’ of the game also have the State object itself passed to them, so they can call it themselves and tell it to change to a new state.

For example, the Menu class is passed the active state object as part of its init parameters:

*class* Menu:  
 *def init*(*self*, stateobj):  
 *self*.state = stateobj

It will later call the State object’s .newstate function when the player presses the “Play Game” button to change the State object’s self.state attribute into a Game object, rather than the Menu object its currently running:

*if* click:  
 *if self*.buttonList[0].collide(mousePos):

...

*self*.state.newstate(Game(*self*.state, world, objects, player))

After this, the Menu object has now effectively deleted itself as the only place it existed was as an attribute of the State object. Instead, that attribute now holds a Game object, and that will be what has its RunFrame method called in the outer game loop every frame instead.

##### Main Menu

The Main Menu was the first non-game section to be developed, and consisted of a title and three buttons for playing the game, viewing the leaderboards, and quitting the game:

Graphical user interface

Description automatically generatedThe title text is simply a Pygame font object blitted onto a rectangle of the same size at the desired position in the RunFrame method:

largeBoldMenu = Pygame.font.Font(QUALY, 100)

mediumMenu = Pygame.font.Font(EXO, 28)

Creation of the Font objects for Menu text; largeBoldMenu is used for the title while mediumMenu is used by the buttons.

*def* RunFrame(*self*, dt):  
 screen.blit(*self*.background\_image, (0, 0))  
 title = largeBoldMenu.render("PhysX", *True*, ORANGE)  
 titleRect = title.get\_rect()  
 titleRect.center = ((swidth / 2), 100)  
 screen.blit(title, titleRect)

The buttons on the main menu are instantiated through the use of the DrawButton method and stored in a list called buttonList in the form of MenuButton class:

*class* Menu:  
 *def init*(*self*, stateobj):  
 *self*.state = stateobj  
 *self*.buttonList = []  
 *self*.DrawButton("Play Game")  
 *self*.DrawButton("Leaderboards")  
 *self*.DrawButton("Quit")  
 *self*.background\_image = Pygame.image.load("assets/background/background.png").convert()

*class* MenuButton:  
 *def init*(*self*, text, pos):  
 *self*.text = text  
 *self*.buttonText = mediumMenu.render(text, *True*, BLACK)  
 *self*.buttonRect = Pygame.Rect(pos[0], pos[1], swidth / 2, 50)  
 *self*.buttonTextRect = *self*.buttonText.get\_rect()  
 *self*.buttonTextRect.center = *self*.buttonRect.center  
 *def* Draw(*self*):  
 Pygame.draw.rect(screen, GREY, *self*.buttonRect, 0, 7)  
 Pygame.draw.rect(screen, BLACK, *self*.buttonRect, 4, 7)  
 screen.blit(*self*.buttonText, *self*.buttonTextRect)  
 *def* collide(*self*, mousePos):  
 *return self*.buttonRect.collidepoint(mousePos)

The MenuButton class

The MenuButton class simply stores the required graphical Pygame components to render them on the Menu screens; it also stores the rect of the button itself for use in click detection.

The DrawButton assigns positions to each button according to how many there are, in order to avoid clipping issues and maintain a clean look. It also improves maintainability, as future additions to the menu simply require another DrawButton call, and the method will manage positioning for us:

*def* DrawButton(*self*, text):  
 newpos = ((swidth / 4), (2 / 5 \* sheight) + 60 \* *len*(*self*.buttonList))  
 *self*.buttonList.append(MenuButton(text, newpos))

As shown, the y component of the button’s position is offset by +60 for every button that already exists.

The Menu object then has to call each button’s Draw method in its RunFrame method:

*for* button *in self*.buttonList:  
 button.Draw()

Every frame, the Menu object checks for a left mousebutton click using the get\_pressed() method to eliminate repeated clicks:

click, \_, \_ = Pygame.mouse.get\_pressed()  
mousePos = Pygame.mouse.get\_pos()

Notice that despite the get\_pressed method also returning bools for a mouse2 or mouse3 click, we dump these returns as we don’t need them.

We then write sections of code for dealing with which button the user clicked. This is done by checking each button’s collide method – if any return true they were where the mouse was positioned. As such, the “Play Game” button happens to be in position 0 of the buttonList, and its payload is as follows:

*if* click:  
 *if self*.buttonList[0].collide(mousePos):  
 objects = [PhysObject((100, 100), ball\_image, 150, SPHERE\_DRAG\_COEFFICIENT, *True*, 0.35)]  
 *for* i *in range*(0, 5):  
 objects.append(PhysObject((random.randint(0, swidth - ball\_image.get\_width()), random.randint(0, sheight - ball\_image.get\_height())), ball\_image, 150, SPHERE\_DRAG\_COEFFICIENT, *True*, 0.35))  
 player = Player((50, 100), player\_image, 100)  
  
 world = [WorldCollider(Pygame.Rect(0, 474, 1279, 720 - 474))]  
 *self*.state.newstate(Game(*self*.state, world, objects, player))

*elif self*.buttonList[2].collide(mousePos):  
 Pygame.quit()  
 sys.exit()

The highlighted portion of code is simply for initiating the objects which were included in the test level present in this version of the game. These objects are then passed to the new Game object which the State object will store.

The leaderboard functionality is not actually present in this version, so there is no code for its button-press.

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Basic Menu | The program should be able to switch between states seamlessly | Pressing the “Play Game” button should immediately switch the running frames to frames of the game itself, and no components of the menu should be left over. | Pressing Play Game immediately begins the only existing level, removing all aspects of the Menu from the runtime environment. | **✓** |
| Buttons should register a single mouse press and execute their payload a single time | Any button on the menu should function as expected. | Pressing the Play Game instantiates a new Game object once, and there is no flashing on the screen signifying repeated executions of the initialisation code. | **✓** |
| Leaderboard | A final score screen should appear after completing a level | After successfully completing the first level, a score screen should display with their total score and a breakdown of it. | Doesn’t exist in version 1 | ✘ |
| The leaderboard should update scores appropriately for each level | When a user enters their username to save a score, the leaderboard should update the score for that username only if it exists and the new score is higher, otherwise should simply add it to the list of other scores. A score of 0 should not be saveable. | Doesn’t exist in version 1 | ✘ |
| The leaderboard should be sorted from highest score to lowest | When viewing the leaderboard, it should be in descending order. | Doesn’t exist in version 1 | ✘ |
| Login System | Usernames should conform to a naming convention | A username with more than 15 characters or that doesn’t begin with a letter should be rejected. | Doesn’t exist in version 1 | ✘ |

### Version 2

#### Leaderboard

##### Scoring Screen

The Scoring Screen is a new state added in Version 2, representing the screen that shows at the completion of a level. It presents the player with their final score, breaks down all the modifiers to said score and allows the user to save that score under their chosen username.

Graphical user interface, text, application

Description automatically generated

Example of a scoring screen (optimal level time 15 seconds)

When a level is complete, the required parameters to calculate the player’s score are passed to the ScoringScreen’s init method:

completed = *True  
for* objective *in* objectives:  
 objective.Update([player] *if isinstance*(objective, PlayerObjective)  
 *else* [x *for* x *in* objects *if isinstance*(x, KeyObject)])  
 *if not* objective.complete:  
 completed = *False* objective.Draw(screen)  
*if* completed:  
 *self*.state.newstate(ScoringScreen(*self*.state, objectives, *self*.timer.GetTime(), player.collisions, *self*.player.fuel / *self*.player.tank, *self*.levelnum))

The algorithm for determining whether all objectives are met – part of the Game class’ RunFrame method.

*class* ScoringScreen:  
 *def init*(*self*, stateobj, objectives, timer, collisions, fuelperc, levelnum):  
 *self*.state = stateobj  
 *self*.totalobj = *len*(objectives)  
 *self*.objmet = *len*([x *for* x *in* objectives *if* x.complete])  
 *self*.optimal = OPTIMALS[*str*(levelnum)]  
 *self*.time = *int*(timer)  
 *self*.collisions = collisions  
 *self*.fuelperc = fuelperc  
 *self*.levelnum = *int*(levelnum)

The first half of the ScoringScreen init method

###### Score Calculation

The second half of the init method is dedicated to combining these variables into a final score.

The final score is calculated as follows:

1. A base score is the score given for the optimal time, no damaging collisions, with all objectives met. This is usually 10000.
2. The user’s time is then compared to the optimal time for that level, stored in the OPTIMALS dictionary in constants.py for all the levels in the game. The proportion of time under or over the optimal time that the player took is the proportion of the BASESCORE that will be added or deducted respectively. E.g., finishing in 11 seconds on a level with a 10 second optimal time will lead to 1000 points being deducted from the final score.
3. The player’s final fuel percentage is compared to 50%. For a fuel tank of 50% or more, 1000 bonus points are awarded, reduced by an amount proportional to how much less final fuel the player had.
4. Every damaging collision then deducts a fixed number of points from the totalscore, determined by HITPENALTY in constants.py.

*if self*.objmet == *self*.totalobj:

timeDiff = *self*.optimal - timertimeMult = *min*(*abs*(timeDiff) / *self*.optimal, 1)timeMult \*= -1 *if* timeDiff < 0 *else   
 self*.fuelbonus = *int*(*min*(*self*.fuelperc / 0.5, 1) \* 1000)  
 *self*.score = *int*(SCOREBASE + (SCOREBASE\*timeMult) + *self*.fuelbonus - (HITPENALTY \* collisions))  
 *self*.score = 1 *if self*.score <= 0 *else self*.score  
 *self*.score = 0

A player who succeeds can only reduce their score to a minimum of 1, since a score of 0 is considered a complete failure and cannot be saved on the leaderboard.

###### Buttons

The buttons on the ScoringScreen are of the same MenuButton class as used on the Main Menu. They are instantiated in the init method:

*self*.buttonList = [MenuButton("Replay", (swidth/5, sheight \* (3/5)), swidth/4),  
 MenuButton("Next Level", (swidth/5, sheight \* (3/5) + 60), swidth/4),  
 MenuButton("Main Menu", (swidth \* (3/5), sheight \* (3/5)), swidth/4)]

Their payloads are managed in the RunFrame, all of which are self-explanatory:

*if* click:  
 *if self*.buttonList[0].collide(mousePos):  
 *self*.state.newstate(gameInit(*self*.levelnum, *self*.state))  
 *elif self*.buttonList[1].collide(mousePos) *and len*(OPTIMALS) >= *self*.levelnum+1:  
 *self*.state.newstate(gameInit(*self*.levelnum + 1, *self*.state))  
 *elif self*.buttonList[2].collide(mousePos):  
 *self*.state.newstate(Menu(*self*.state))

Note that the length of OPTIMALS is used to check if there is a next level. If there isn’t the button simply won’t work.

gameInit is a new function simply designed to shorten the code for loading a new level and reduce repetition:

*def* gameInit(levelnum, stateobj):  
 gameData = level\_load(levelnum)  
 background, world, objects, objectives, obstacles, hazards, player = gameData["background"], gameData["world"], gameData["objects"], gameData["objectives"], gameData["obstacles"], gameData["hazards"], gameData["player"]  
 *return* Game(stateobj, background, world, objects, player, objectives, obstacles, hazards, levelnum)

###### Score Details

Along with the total score, the ScoringScreen displays a breakdown of each aspect of the player’s gameplay which affected their score. Beneficial actions are highlighted in green, penalising ones in red, and unchanging ones in white.

This is achieved through the use of a method and a series of algorithms which essentially do the same thing: blit the required information onto the screen after determining the correct colour for it.

*self*.detailnum = 0  
colour = WHITE  
*if self*.objmet < *self*.totalobj:  
 colour = RED  
textRender(mediumText, ((swidth / 2), 220 + (*self*.detailnum \* 60)), f"Objectives Met: {*str*(*self*.objmet)}/{*str*(*self*.totalobj)}", colour)  
*self*.detailnum += 1  
  
colour = WHITE  
*if self*.time < *self*.optimal:  
 colour = GREEN  
*elif self*.time > *self*.optimal:  
 colour = RED  
*self*.detailRender("Time", Timer.formatTime(*self*.time), colour)  
  
  
colour = WHITE  
*if self*.collisions > 0:  
 colour = RED  
textRender(mediumText, ((swidth / 2), 220 + (*self*.detailnum \* 60)),  
 f"Damaging Collisions: {*str*(*self*.collisions)} ({*str*(-1 \* *self*.collisions \* HITPENALTY)})", colour)  
*self*.detailnum += 1

Snippet of the ScoringScreen RunFrame method

These all take advantage of the new textRender function – a much more reusable function for placing text on the screen:

*def* textRender(font, pos, text, colour):  
 rendered = font.render(text, *True*, colour)  
 rect = rendered.get\_rect()  
 rect.center = pos  
 screen.blit(rendered, rect)

The detailnum attribute tracks how many attributes are displayed on the ScoringScreen and makes adding and blitting them easier. It uses the same logic as the buttons from the MainMenu, in that each new score detail is offset by a fixed amount based on how many details there already are.

The detailRender is a method that does the same as the other two sections of code, it only exists for future reusability, in case new factors are considered when calculating the score.

*def* detailRender(*self*, detail, value, colour):  
 detailText = mediumText.render(f"{detail}: {*str*(value)}", *True*, colour)  
 detailRect = detailText.get\_rect()  
 detailRect.center = ((swidth / 2), 220 + (*self*.detailnum \* 60))  
 screen.blit(detailText, detailRect)  
 *self*.detailnum += 1

Examples of score screens:

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Both levels completed on a level with optimal time 15 seconds

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Leaderboard | A final score screen should appear after completing a level | After successfully completing the first level, a score screen should display with their total score and a breakdown of it. | For a player completing a level in 12 seconds with an optimal time of 15 seconds after experiencing 1 damaging collision:  The objectives were listed in white (1/1)  The time taken was listed in green (00:12)  The number of collisions was in red (1), and the points deducted next to it (-500).  The final score was displayed at the top: 11130 | **✓** |
| The leaderboard should update scores appropriately for each level | When a user enters their username to save a score, the leaderboard should update the score for that username only if it exists and the new score is higher, otherwise should simply add it to the list of other scores. A score of 0 should not be saveable. | Doesn’t exist in version 2 | ✘ |
| The leaderboard should be sorted from highest score to lowest | When viewing the leaderboard, it should be in descending order. | Doesn’t exist in version 2 | ✘ |
| Login System | Usernames should conform to a naming convention | A username with more than 15 characters or that doesn’t begin with a letter should be rejected. | Doesn’t exist in version 2 | ✘ |

### Version 3

#### Basic Menu

##### Main Menu

The Main Menu is updated further in Version 3. All menu objects have their colour changed to darker black to contrast with the orange accents, and a new button has been added to accommodate a level-selecting feature:

A screenshot of a computer

Description automatically generated with medium confidence

This was easily done due to the built-in reusability of the MenuButton class:

*class* Menu:  
 *def \_\_init\_\_*(*self*, stateobj):  
 *self*.state = stateobj  
 *self*.buttonList = []  
 *self*.DrawButton("Play Game")  
 *self*.DrawButton("Level Select")  
 *self*.DrawButton("Leaderboards")  
 *self*.DrawButton("Quit")

As shown, all the addition required was a new line for DrawButton(“Level Select”)

It also required an addition to the mouse-press checking in the RunFrame method:

*if self*.buttonList[1].collide(mousePos):  
 *self*.state.newstate(LevelSelect(*self*.state, *True*))

###### Menu Minigame

In addition, the main menu received a feature which allows the player to insert physics-simulated balls around the screen and launch them at random velocities. The buttons and text on the menu act as WorldColliders, and the objects collide realistically with each other and the menu items:

Graphical user interface

Description automatically generated

The Menu’s init and Update methods are adjusted accordingly to accommodate the new Physics elements:

*# Physics info for the hidden right click  
self*.world.append(WorldCollider(titleRect, "Steel")) *# WorldColliders for the text  
self*.world.append(WorldCollider(rightRect, "Steel"))  
  
*self*.world.append(WorldCollider(pygame.Rect(-1, 0, 1, sheight))) *# WorldColliders for edges of the screen  
self*.world.append(WorldCollider(pygame.Rect(0, -1, swidth, 1)))  
*self*.world.append(WorldCollider(pygame.Rect(swidth, 0, 1, sheight)))  
*self*.world.append(WorldCollider(pygame.Rect(0, sheight, swidth, 1)))

*self*.world = [WorldCollider(x.GetRect(), "Steel") *for* x *in self*.buttonList] *# WorldColliders for the buttons*

*self*.objects = []  
*self*.colhandler = CollisionHandler((swidth, sheight))

Snippet of the Menu.init method

*for* i, obj *in enumerate*(*self*.objects):  
 colliders = [x *for* x *in self*.objects *if* x != obj] + *self*.world  
 obj.Update({"gravity": 2, "airdensity": 0}, colliders, dt)  
 obj.Draw(screen)  
 *if not* Rect(0, 0, swidth, sheight).contains(obj.GetRect()):  
 *self*.objects.pop(i)  
  
*self*.colhandler.Update(*self*.objects)

Snippet of the Menu.Update method

Gravity is set very low here (in the above example, to 2m/s2), and there is no air present. As well as this, all WorldColliders use the ‘Steel’ material, which possesses a low frictional coefficient. This lack of drag forces is simply to maximise the movement of the objects on the screen, and just make it generally more fun to experience this more sandbox-like implementation of the physics engine.

##### Level Select

The Level Select sub-menu was developed to present the user with a page listing all the available levels and choose accordingly, either for playing a level or for viewing the leaderboards for it:

A picture containing shape

Description automatically generated

The LevelSelect class first examines the ‘levels’ subfolder in the game’s root directory for the filenames within, since all levels have folder names of their number.

*class* LevelSelect:  
 *def init*(*self*, stateobj, game):  
 *self*.state = stateobj  
 *self*.game = game  
 *self*.levels = os.listdir("levels")  
 *self*.levels.sort(key=*lambda* x: *int*(x))

First half of the LevelSelect init method; notice that the folder names are sorted ascending according to their numeric value.

It was chosen before-hand that the LevelSelect GUI would take on a grid-like format, wherein there are 3 rows of 5 levels each. These level buttons would be equidistant to each other and to the borders of the screen – forming a grid-like structure.

The first step of this process would be to split the level list into sublists of a maximum 5 levels in each, done as follows:

*# Split the list of levels into sections of 5*fullRows = *len*(*self*.levels) // 5  
*for* i *in range*(fullRows):  
 insert = i \* 5  
 *self*.splitLevels.append(*self*.levels[insert:insert+5])  
*if len*(*self*.levels) % 5 != 0:  
 *self*.splitLevels.append(*self*.levels[-(*len*(*self*.levels) % 5)::1])

1. Find out how many full rows we can get using floor division.
2. Every full row will begin at a multiple of 5 starting from 0, and end 4 indexes ahead. As such, we add 5 instead of 4 since list slicing does not include the final index.
3. Now that all full rows are considered, check if there are any leftovers – the modulus operator will provide the exact number of leftovers.
4. These leftovers can be gathered into the final row/sublist using reverse list slicing, then appended to the overall splitLevels list.

###### Coordinate Calculations

Positioning for each level button is calculated via an algorithm derived from an analysis of what the final appearance of the menu should be.

Fundamentally, the width and height of the screen must be split into sections of equal measure in order to produce the desired effect. These will be somewhat like grid ‘squares’, which determine the size of each gap as well as the buttons themselves. Consider the following diagram:

A picture containing shape

Description automatically generated

Yellow boxes represent ‘grid squares’ down the y axis, while blue boxes represent grid squares along the x axis.

From the diagram, it is clear (by counting the boxes) that to produce the desired grid structure of 5x3, the screen’s width must be divided by 11 (11 boxes) and the screen’s height must be divided by 7:

sliceWidth = swidth / 11  
sliceHeight = sheight / 7

After this, we must determine the pattern for placing each level button.

It is clear that, if n is equal to the number of ‘jumps’ of a width slice, each button is placed at 1 jump, 3 jumps, 5 jumps, etc.

Similarly, on the y axis, each button is placed at 1 jump, 3 jumps, 5 jumps (of a height slice).

Mathematically, this can be represented by the sequence S as:

Where n is the position in the sequence (starting from 0):

Nested loops can then be used to find the value n (i in the code) and position each button accordingly:

*self*.buttonList = []  
*for* i, row *in enumerate*(*self*.splitLevels):  
 *#sequence is 2n+1* y = (2 \* i + 1) \* sliceHeight  
 *for* i, level *in enumerate*(row):  
 x = (2 \* i + 1) \* sliceWidth  
 *self*.buttonList.append(MenuButton(level, (x, y), sliceWidth, sliceHeight))

A back button is also added to this menu, to allow the player to return to the main menu.

#### Leaderboards

##### Saving Scores

Scores are saved in CSVs stored in a subfolder to the root, named ‘scores’. Within each CSV (named [levelnumber].csv) are rows of username-score pairings. The SaveScore object/menu allows a player to save their scores to the database, all while enforcing a naming convention and maintaining a sorted structure.

To accommodate this feature, a ‘Save Score’ button was added to the scoring screen:

Graphical user interface, text

Description automatically generated

After pressing the button, the user is taken to another screen:

Graphical user interface, application

Description automatically generated

The user can enter up to 40 characters within the field, which will be processed and checked with the filterName method. The screen is constantly printing a string named self.error underneath the field, which will contain a value if the filterName method returns False and changes its value:

*def* filterName(*self*):  
 *if len*(*self*.text) >= MAXUSERNAMECHARS:  
 *self*.error = f"This username is too long. It must be less than {MAXUSERNAMECHARS} characters."  
 *return False  
 if len*(*self*.text) == 0:  
 *self*.error = "Please enter a name."  
 *return False  
 if not self*.text[0].isalpha():  
 *self*.error = "The first character of the name must be alphabetic."  
 *return False  
 self*.error = ""  
 *return True*

MAXUSERNAMECHARS is a constant in constants.py

Examples of error messages displayed:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

###### File Handling

When saving scores, to avoid using a full sorting algorithm (which can become memory/time intensive with large datasets), and repeatedly sorting the data, new scores are simply inserted in the right position in the first place – a much more practical solution. The recordScore method accomplishes this task.

The first part of the algorithm opens the appropriate score file in append mode and immediately closes it. This is done to ensure it exists before the algorithm continues; Python’s open function will create the file if it doesn’t exist when in append mode. If it does exist, nothing will happen, and it will carry on.

*def* recordScore(*self*):  
 board = []  
 presenceCheck = *open*(os.path.join("scores", f"{*self*.level}.csv"), "a")  
 presenceCheck.close()

board will be the list variable holding the leaderboard we retrieve from the csv.

Before we can insert our score, we have to retrieve all the current scores in the file to determine exactly where to position it. This is easily done via a csv reader; we loop through and append each row to board:

*with open*(os.path.join("scores", f"{*self*.level}.csv"), "r", newline='') *as* file:  
 reader = csv.reader(file)  
 *for* row *in* reader:  
 board.append(row)

If the board is empty, we just append our new score:

newScore = [*self*.text, *self*.score]  
*if len*(board) == 0:  
 board.append(newScore)

If not, we should first check if the username in question already exists in the leaderboard. If so, and our new score is higher, it should be changed in the leaderboard:

*else*:  
 names = [x[0] *for* x *in* board]  
 i = *None  
 if self*.text *in* names:  
 i = names.index(*self*.text)  
 *if* i *is not None and int*(board[i][1]) < *self*.score:  
 board = board[:i] + board[i+1:]

Notice that we don’t actually input the new score at this point. Instead, we will just remove the old score and let the usual algorithm place the new score. This segment of code simply makes sure we don’t have a duplicate.

We should then deal with the case where the new score is higher than the top score on the leaderboard:

*for* i, row *in enumerate*(board):  
 row[1] = *int*(row[1])  
 *if* i == 0 *and* newScore[1] >= row[1]:board = [newScore] + board  
 *break*

The score is added via list addition to avoid moving all the list items one index ahead (a time consuming and unnessecary process). The newScore is inserted into a temporary 2d list and added to the board list. Since board is also a 2d list, this maintains the dimensions of the overall list appropriately.

If that wasn’t the case, we just need to go through the list and check if there exists a position such that our score’s numerical value is in between two other scores.

*if* i != *len*(board) - 1:  
 *if* row[1] >= newScore[1] *and int*(board[i+1][1]) <= newScore[1]:  
 temp = board[i+1:] *# copy the board items ahead of where we want to insert* board = board[:i+1] *# remove those items from the board* board.append(newScore)  
 board = board + temp  
 *break*

The list slicing is described within the inline comments.

If we get to the end of the list, we obviously just append as usual since the score must be lower than all the others:

*else*:  
 board.append(newScore)  
 *break*

After all this list comprehension, the final list can then be rewritten into the csv file:

*with open*(os.path.join("scores", f"{*self*.level}.csv"), "w", newline='') *as* file:  
 writer = csv.writer(file)  
 writer.writerows(board)

##### The Leaderboard Screen

The leaderboard was finally added in version 3 with full functionality:

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

When the leaderboard button is pressed on the main menu, the level select menu appears to allow the player to choose which level they want to view the leaderboards for. This is passed into its init parameters:

*class* Leaderboard:  
 *def init*(*self*, stateobj, levelnum):  
 *self*.state = stateobj  
 *self*.scores = []  
 *self*.level = levelnum  
 *with open*(os.path.join("scores", f"{levelnum}.csv"), "r", newline='') *as* file:  
 reader = csv.reader(file)  
 *for* row *in* reader:  
 *self*.scores.append(row)  
 *self*.scroll = 0  
 *self*.backButton = MenuButton("<", (20, 20), swidth / 8, 50)

Notice that the scores are loaded from the csv within the init method.

The self.scroll attribute stores the page the leaderboard is currently displaying, starting from 0.

In order to produce the screens above, we should first calculate where each entry in the leaderboard will be placed. All entries will share the same x coordinate, but will be at equidistant y coordinates, and we want to fit 10 scores on each page. As a result:

x = swidth / 2  
  
heightget = hudFont.size("c")[1]  
sliceHeight = sheight / 10

heightget will retrieve the pixel height of a character in the chosen font

Similarly to finding the number of rows for LevelSelect, we will calculate the number of pages we need to fit all the entries on the leaderboard – easily accomplished through floor division.

maxPages = *len*(*self*.scores) // 10

After this, we need to draw every 10 scores to the screen according to where they would be indexed in the list; every increment of self.scroll equals another page of leaderboard scores, and in turn another 10 jumps into the leaderboard list. We will also need a separate tracker to maintain the 1-10 counter for graphics processing. For example, while an entry might be at the 15th position in the leaderboard, it is still only the 5th position on its page. This is represented by the ‘itemNum’ variable before the loop:

itemNum = 0  
*for* i *in range*(0 + (10 \* *self*.scroll), *min*(*len*(*self*.scores), 0 + (10 \* *self*.scroll) + 10)):  
 stringStart = f"{i+1}. {*self*.scores[i][0]}"  
 strin = stringStart + ((28 - *len*(stringStart)) \* " ") + *str*(*self*.scores[i][1])  
  
 colour = YELLOW *if* i == 0 *else* WHITE  
  
 textRender(hudFont, (x, (sliceHeight \* itemNum) + (heightget / 2)), strin,  
 colour)  
 itemNum += 1

Notice the addition of spaces between each user-score pairing – this is done to produce a lined-up appearance for every entry in the leaderboard. The first half of the string is created first, and spaces are added depending on how many letters there are in the first half starting from 28 (random number), to ensure every score begins at the same x pixel coordinate.

At first, a standard font was used which did not produce the desired effect. This was why the font was changed to the hudFont (Unispace) – which happened to be a font with constant spacing. This was required to avoid complicated font size calculations and allow this much simpler algorithm to remain effective.

The player can scroll up and down the leaderboard using the arrow keys:

*for* event *in* Pygame.event.get():  
 *if* event.type == Pygame.MOUSEBUTTONDOWN:  
 mousePos = Pygame.mouse.get\_pos()  
 *if self*.backButton.collide(mousePos):  
 *self*.state.newstate(LevelSelect(*self*.state, *False*))  
 *if* event.type == Pygame.KEYDOWN:  
 *if* event.key == Pygame.K\_DOWN:  
 *self*.scroll += 1 *if self*.scroll != maxPages *else* 0  
 *elif* event.key == Pygame.K\_UP:  
 *self*.scroll += -1 *if self*.scroll != 0 *else* 0

Scrolling too far is handled directly within the input processing

#### Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Test** | **Expected Outcome** | **Actual Outcome** | **✓** |
| Leaderboard | The leaderboard should update scores appropriately for each level | When a user enters their username to save a score, the leaderboard should update the score for that username only if it exists and the new score is higher, otherwise should simply add it to the list of other scores. A score of 0 should not be saveable. | With the following leaderboard:  NewPlayer – 13087  James – 12933  Robert – 12754  Robert achieves a score of 12954. The leaderboard now looks like this:  NewPlayer – 13087  Robert – 12954  James – 12933  A player called Jim is then inserted with a score of 13052:  NewPlayer – 13087  Jim - 13052  Robert – 12954  James – 12933 | **✓** |
| The leaderboard should be sorted from highest score to lowest | When viewing the leaderboard, it should be in descending order. | All scores are added in their correct position beforehand; the leaderboard is never unsorted. | **✓** |
| Login System | Usernames should conform to a naming convention | A username with more than 15 characters or that doesn’t begin with a letter should be rejected. | Error messages are produced when breaking ruling conventions, and presence checks are carried out. | **✓** |

##### Input Testing

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Type** | **Test Data** | **✓** |
| Username Entry | Valid | *“bob445”* | **✓** |
| *“bobbillson”* | **✓** |
| *“bobby404-\_”* | **✓** |
| Boundary | *“a”* | **✓** |
| *“bbbbbbbbbbbbbbb”* | **✓** |
| *“c\_c------------”* | **✓** |
| *“d1”* | **✓** |
| Invalid | *“---bob”* | **✓** |
| *“1”* | **✓** |
| *“1bob1”* | **✓** |
| *“1234b”* | **✓** |
| *“abbbbbbbbbbbbbbc”* | **✓** |
| Null |  | **✓** |

All rejected entries are not passed further into input handling, and appropriate error messages are produced to the user, as shown in the Saving Scores section.

# Evaluation

## Testing to Inform Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **Strand** | **Module** | **Test Objectives** | **✓** |
| Physics Simulation | Force Evaluation | Resultant Force should be composed of all forces acting on the object | **✓** |
| Physics values (velocity, acceleration, etc.) should be represented accurately and independent of framerate | **✓** |
| Pixel movement amount and direction should closely represent its real-world equivalent as it relates to the METRE constant. | **✓** |
| All objects must follow F=ma, such that their acceleration is proportional to their resultant force in any direction. | **✓** |
| Collision Simulation | Objects should be stopped or rebound from collisions with WorldColliders | **✓** |
| Objects should experience a transfer of momentum accurate to their physical values when colliding | **✓** |
| Objects should be able to push others by maintaining contact with them and exerting a force in their direction. | ✘ |
| Natural Force Application | Friction should only act when in contact with a surface, according to the material it’s made out of | **✓** |
| Air Resistance should act proportionally to oppose the direction of motion | **✓** |
| Friction should act for both non-moving (but trying to) objects and moving objects. | **✓** |
| Menu | Basic Menu | Buttons should not experience more than a single mouse button press. | **✓** |
| Level Select should display all the levels available to the player and initiate the game correctly. | **✓** |
| Login System | Text input for the user’s username should abide by a 15-character limit, and must include and start with an alphabetic character | **✓** |
| The scores attained during the session should be attributed to the same username. | **✓** |
| Leaderboard | Should display the top 10 scores attained in each level, with their associated usernames. | **✓** |
| Level Design | Environmental Hazards | Moving within dense liquids should result in a representatively increased amount of drag | **✓** |
| Physics Objects should experience a force when intersecting the path of a “wind” hazard | **✓** |
| Score should be deducted when colliding with sides of the level | **✓** |
| Colliding with a forbidden section of the level should immediately result in failure/reset | **✓** |
| Playability | All levels should be complete-able | **✓** |
| Levels should be free of bugs which could be caused by the player and prevent completion. | **✓** |
| Difficulty should progress with level number | **✓** |
| Graphics | Animations | Rotating the player should maintain collision consistency (should not phase through other objects) | **✓** |
| Input from the player should result in a graphical response; e.g thrust should produce engine fumes in the ship | **✓** |
| Rotation should cause minimal image distortion | **✓** |
| HUD | Fuel meter should deplete accurately in accordance with the remaining fuel in the player’s ship | **✓** |
| Physics information should display accurately and be readable if the player chooses to enable it | **✓** |

## Stakeholder Testing

|  |  |  |
| --- | --- | --- |
| **Requirement** | | **Stakeholder Feedback** |
| I | Motion must be simulated as a result of the application of force. Forces should be evaluated and produce resultant motion in the form of acceleration and velocity. | Both stakeholders were completely satisfied with the force and motion system in the game. In particular, they described how the control of the rocket was much more difficult due to the use of forces rather than direct control but did not find this to be a negative feature. |
| II | Environmental effects will be simulated on the player. Specific examples include friction and air resistance but may include many forms of drag or drive that are applied to the player as a result of the properties of their environment. | Both stakeholders were satisfied with the implementation of drag forces in the level environments but criticised the visibility of air resistance arose. Abdul Bahir believed the presence of air resistance is not visible enough and can only really be noticed if the player opens the details-mode. |
| III | Collisions with both the environment and other objects will be simulated. Momentum will be transferred between objects according to various factors and objects’ motion will be adjusted accordingly. | Both stakeholders were mostly satisfied with object collisions and did not notice throughout the fact that pushing objects were not simulated.  However, both stakeholders noticed that collision resolution was somewhat disconnected from the true shapes of the sprites themselves. This is, in fact, an inherent limitation of Pygame’s rect collision system. |
| IV | Basic menu system must be implemented to allow the player to play, view leaderboards, level select etc. | Both stakeholders were satisfied with all the menu screens. |
| V | A login system is needed to save the score of the player to a desired username and track progress. | Both stakeholders tested the login system several times to observe the correct changes were being populated into the leaderboard and found no errors. |
| VI | A leaderboard is required to track top scores and allow players to compare their scores to others. | Both stakeholders were impressed with the leaderboards but noted that the method of scrolling was slightly unusual; Mihai suggested a scroll bar rather than the use of arrow keys. |
| VII | Levels must be stored using external data structures (likely CSV files) and loaded in through a suitable algorithm | The stakeholders did not check the method of data/level storage, nor the algorithm involved as they were not knowledgeable in the concept. However, both the stakeholders experienced no issues loading the levels. |
| VIII | Levels must contain set objectives to be met in order to determine completion. These objectives would involve both player and object interaction. | Both stakeholders found the levels intuitive (particularly the first two, which contained instructions) and understood the concept of objectives blocks and key-objects.  Despite this, level 3 was still a difficult task for both stakeholders, as it took them some time to realise the penalty for damaging collisions was a necessary loss to complete the level. |
| IX | Environmental hazards will be implemented into the game and can be placed on any level | Both stakeholders were impressed with the environmental hazards already in the game, but Abdul Bahir suggested more hazards such as thick liquids or moving blocks. |
| X | A Camera function should be in the game to allow traversal around a level and prevent the game being limited only to the size of the screen. | The stakeholders did not take particular notice of the camera function but experienced no problems in its functionality. |
| XI | Animations will play to show the ship accelerating, dust kicking up from collisions, other environmental effects etc. | Both stakeholders were impressed with the engine fumes being released as the ship thrusts, but they noted that the particle effects from other objects in the world were sometimes excessive. A particular example was in level 3, where objective and hazard particles overlap each other and clutter the screen-space. |
| XII | A HUD will consist of a fuel meter and, if enabled, an overlay visually displaying the forces and physics values acting on different objects in the simulation. | Both stakeholders made full use of the HUD features like the fuel and timers. When opening the ‘details-mode’ they were both pleased with the simplicity of the window yet impressed by the amount of detail presented for each object. Abdul Bahir described it as a ‘novelty’ but was happy that it was optionally there for people who are interested in seeing it. |

## Meeting the Success Criteria

### Physics Simulation

|  |  |  |  |
| --- | --- | --- | --- |
| **Strand** | **Requirement** | | **Evidence** |
| Physics Simulation | I | Motion must be simulated as a result of the application of force. Forces should be evaluated and produce resultant motion in the form of acceleration and velocity. | Video of object moving over a given time, along with the values for the motion. |
| II | Environmental effects will be simulated on the player. Specific examples include friction and air resistance but may include many forms of drag or drive that are applied to the player as a result of the properties of their environment. | Screenshot of an object experiencing drag forces. |
| III | Collisions with both the environment and other objects will be simulated. Momentum will be transferred between objects according to various factors and objects’ motion will be adjusted accordingly. | Video of objects colliding with each-other and walls. |

The entirety of the Physics Simulation Strand was met in the game, as the physics simulation was present from the early versions of the game. Forces and environmental effects were always present, while collisions were added later in development when the correct algorithms were derived. The only feature not present is the ability for objects to push each other in a continuous collision – the reason for this is described in the limitations section.

All equations for motion are described in the Development section, and work perfectly, with Vec2 objects being the key to representing physics values for objects on a 2D plane.

#### Proof of Accuracy

Text

Description automatically generatedEnvironmental effects are easily displayed with the ‘details mode’ enabled:

Screenshot of an object experiencing drag forces

The player object has a mass of 100kg, and the gravitational field strength (GRAVITY constant) is set to 15 m/s2 (N/kg). Since , the force in question is Weight, so N. The direction of the gravitational field is downwards, which is +y, and so the Weight force of (0, 1500) N is correct.

The Drive vector has been split into x and y components according to the angle of the ship. The Engine drive is 2000N, so the magnitude of the Drive vector should equal this. Note that values displayed in the details mode are rounded to 1 decimal place:

N

Therefore, the drive force is split into its x and y components correctly.

The ReactionY force is caused by the ship touching and applying a force against the surface below it. It should be equal and opposite to the total force in the same plane against the surface (+y direction). This would be: N, so the force of (0, -816.5) N is correct.

Friction is also caused by this contact between surfaces. In this case, there is only friction present along the X axis (as the ship is pushing along a horizontal surface) and is given by . In this case, the surface has the “Asphalt” material, which has a kinetic coefficient of friction of 0.65 – so . As calculated above, the reaction force from the surface (in the -y direction) is R, so .

and so (-530.7, 0) N for FrictionX is correct.

Air resistance will oppose the direction of motion (determined by velocity, in this case, +x), and is dependent on the equation . To determine A, we take the height of the sprite at this point (since it is moving entirely horizontally, so air will be hitting it at the front end) and multiply this by 1, since all the objects have an imaginary 3D depth of 1. The height of the sprite at this point is 1.1163m, so m2

The air density is set to 1.2041, so .

The drag coefficient of the player object is set to 1.15, so

N

This should be multiplied by -1 to oppose the +x motion, so the air resistance of (-426.8, 0) N is correct.

Resultant force is the vector sum of all forces and comes out to 922N in +x. Acceleration is determined by , so m/s2; the acceleration of (9.2, 0) m/s2 is therefore also correct.

### Menu

|  |  |  |  |
| --- | --- | --- | --- |
| **Strand** | **Requirement** | | **Evidence** |
| Menu | IV | Basic menu system must be implemented to allow the player to play, view leaderboards, level select etc. | Screenshot of the menu |
| V | A login system is needed to save the score of the player to a desired username and track progress. | Screenshot of the login screen |
| VI | A leaderboard is required to track top scores and allow players to compare their scores to others. | Screenshot of the leaderboard |

Table

Description automatically generatedA full menu system is present, and all features work perfectly as described.

Screenshot of the Main Menu

The login screen is present in the game as a username entry field the player can use for attributing their achieved score after a level to a desired username. The field input is filtered by a method designed to enforce the rules in place for username formats, and functions correctly. When usernames are saved, they are immediately placed into the correct location in the leaderboard to avoid having to constantly re-sort them later on.

Graphical user interface, application

Description automatically generated

Text

Description automatically generated

### Level Design

|  |  |  |  |
| --- | --- | --- | --- |
| **Strand** | **Requirement** | | **Evidence** |
| Level Design | VII | Levels must be stored using external data structures (likely CSV files) and loaded in through a suitable algorithm | Screenshot of a stored level structure. |
| VIII | Levels must contain set objectives to be met in order to determine completion. These objectives would involve both player and object interaction. | Video of a level being completed. |
| IX | Environmental hazards will be implemented into the game and can be placed on any level | Video of a level being completed with environmental hazards. |

Graphical user interface, text

Description automatically generated with medium confidenceAs seen on the left, levels are stored in folders under the {root}/levels subdirectory and are named after their level number. Within these folders are multiple CSV files containing data about different types of components of the game.

* constants.csv: Stores physics constants for the level (e.g., air density, gravitational field strength)
* hazards.csv: Stores location and force data for AirStream objects.
* objectives.csv: Stores location and type data for objective blocks as well as physics values and image/colour data for KeyObjects.
* objects.csv: Stores all physics values, image data and position data for regular PhysObjects in the level.
* obstacles.csv: Stores location and dimension data for Death Boxes.
* player.csv: Stores physics values, image data, position data, fuel amount and thrust for the Player object.
* world.csv: Stores location and dimension data for WorldColliders.

All level data is loaded in through the level\_load function, as described in the Development section.

A screenshot of a computer

Description automatically generated with medium confidenceExample of the world.csv and player.csv file formats.

Objectives exist in the form of Player and Physics Objectives, and all objectives in a level must be completed in order to pass. AirStream and Death Box objects exist as the environmental hazards in different levels, and passive obstacles such as fuel consumption and damaging collisions also affect the player’s final score.

### Graphics

|  |  |  |  |
| --- | --- | --- | --- |
| **Strand** | **Requirement** | | **Evidence** |
| Graphics | X | A Camera function should be in the game to allow traversal around a level and prevent the game being limited only to the size of the screen. | Video of a level being played. |
| XI | Animations will play to show the ship accelerating, dust kicking up from collisions, other environmental effects etc. | Video of a level being played. |
| XII | A HUD will consist of a fuel meter and, if enabled, an overlay visually displaying the forces and physics values acting on different objects in the simulation. | Video of a level being played. |

The HUD, camera and particle animations are all present and function perfectly in the game. All these features are displayed in the included video tests.

## Usability Features

Within the limited user interface present in the game, the design principle of simplicity is maintained throughout, wherein design is expected to communicate information clearly and simply.

In addition, text on special screens, such as the scoring screen, are large and coloured according to their status (e.g., beneficial score components are highlighted in green, penalising ones in red, unaffecting ones in white) to improve visibility:

Graphical user interface, text

Description automatically generated

Text

Description automatically generated with medium confidenceThe first levels in the game introduce the player to certain controls they may not have guessed, and the less obvious mechanics. For example, level 1 of the game includes text as part of the background to tell the player how to activate their engine:

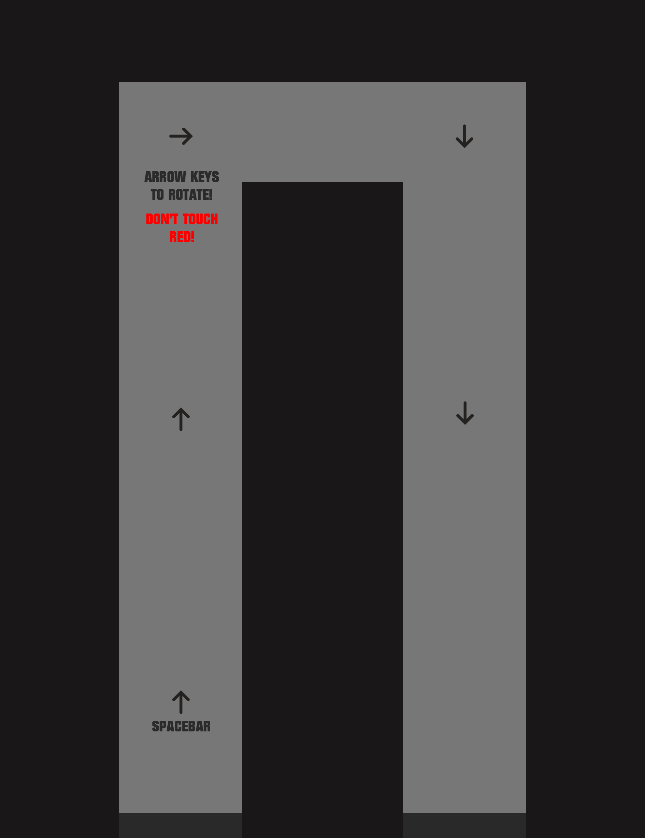
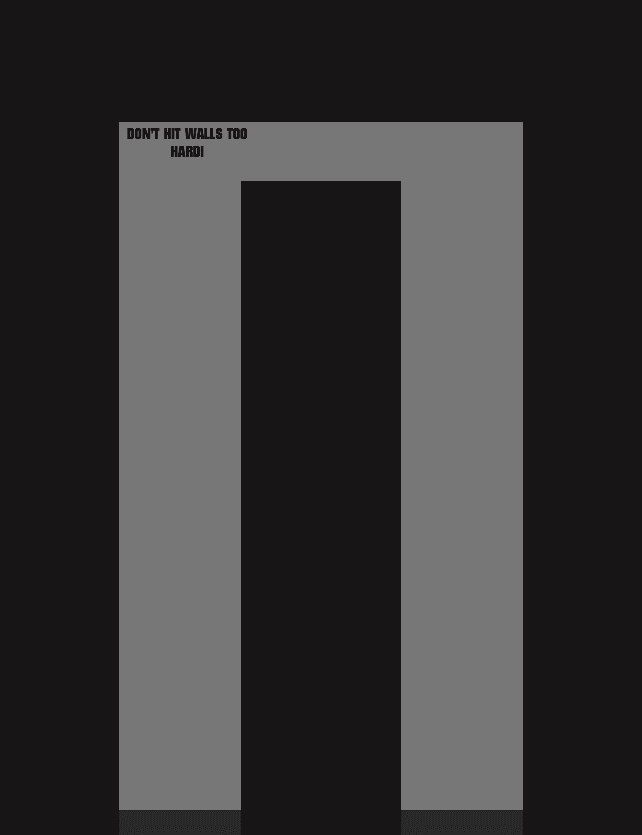
Note also the arrows present in level 1 to direct the player in the right direction

Text

Description automatically generatedText is also present in level 1 to tell the player not to touch red boxes, and how to rotate their ship (since they will need to once they reach the corner).

The colour red is a universal visual cue specifically chosen to suggest to the player that the object is dangerous even if they, for example, cannot read English.

In level 2, the level design of level 1 is copied but adjusted so the height of the vertical columns is much lower:

 Level 1 Level 2

There is also text saying, “Don’t hit walls too hard”, with the thought being that the player would accidentally hit the roof, wrongly assuming the level was just as tall as before. The player would then realise the consequences of such actions when receiving their final score and receiving a penalty for “Damaging Collisions”.

A consistent structure of design is also maintained throughout every screen in the game and every level, and considerable effort is made to emphasise the important of certain objects over others and differentiate them properly.

Chart

Description automatically generatedDespite not telling the player explicitly how objective blocks work and the different types present, nor how objects work, the player is expected to learn through trial and error in the first two levels. This is facilitated by simple designs for the first two levels wherein only one and two objectives are present respectively, and possess different colours for their different types:

Shape

Description automatically generated with medium confidenceLevel 1’s PLAYER Objective

Level 2’s PHYSICS Objectives

Rectangle

Description automatically generated with medium confidenceIt’s important to also note the use of the particle system to highlight these objects and emphasise them even more. An entirely clueless player could still deduce that completion of the level must involve these special objects, and while it may take them some time to discover it, they will quickly discover how they work; level 2 especially is set up to accelerate this process:

Note the brightly coloured and particle-emitting nature of the objects due to their ‘KeyObject’ status

The objects are placed on the edge of the platform in order to allow the player to progress even if they have no idea what to do – it is likely the player will accidentally touch them anyway, and send them falling onto the objective blocks:

Shape

Description automatically generated with low confidence

After seeing this, a player would quickly realise what the coloured objects do. At this point in the level, they would not have enough fuel to continue, but could easily reset and try again with newfound knowledge and achieve a good optimal time/score for the level.

A screenshot of a computer

Description automatically generated with medium confidenceAirStream objects also display their intentions quite clearly, emitting white particles (resembling air) at high velocities in the direction of their push. These are first present in level 3, which is generally a good demonstration of all the usability features:

In the picture:

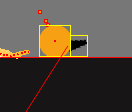
* An AirStream object exists at the centre of the level, emitting white particles in the direction of the PLAYER objective at the top of the level.
* The PLAYER objective is initially bright yellow to differentiate it from the pink PHYSICS objective inside the chamber to the left.
* Within the chamber exists another airstream object sending white particles downwards, suggesting it pushes objects downwards into the PHYSICS objective. If the player travels here either by accident or on purpose, they will soon realise they cannot leave, as their engine thrust will not overcome the force of the AirStream.
* A coloured object appears on the right-hand side of the screen. The player has already played level 2 by this point and should be able to realise the object must be somehow moved on top of the pink objective box.

One aspect of the design that could have been improved involved the damaging collisions feature. Feedback to the user here could have been improved by inducing some sort of visual effect or shake to the screen when colliding with a wall too hard, as at the moment no feedback is presented to the user on damaging collisions until they receive their final score for the level.

## Limitations

### Pushing PhysObjects

The most notable and obvious limitation in the physics engine is the inability for a continuous “pushing” force between two objects. For example, if an object tries to stick to another and push it in a direction, their collision will simply resolve once, and the objects will be able to phase through each other.

As seen on the left, the player continues applying a driving force to the right, and eventually passes through the orange ball if the hitboxes stop separating (the final collision is resolved, and the objects didn’t separate).

The bug saw several attempts to resolve it, but all produced logic errors. The closest it came was in the early prototype named “Object-Handled\_Collisions” (Version 2 of Physics Simulation) – an experimental branch in development where collision objects are checked and instantiated by objects themselves rather than a global manager. This almost produced the desired effect, where no objects phased through each other, and objects produced the illusion of pushing each other.

*for* entity *in* hit\_list:  
 *if self*.velocity.x > 0:  
 *self*.rect.right = entity.GetRect().left  
 *self*.pos = Vec2(*self*.rect.center)  
 collision\_types["right"] = *True  
  
 elif self*.velocity.x < 0:  
 *self*.rect.left = entity.GetRect().right  
 *self*.pos = Vec2(*self*.rect.center)  
 collision\_types["left"] = *True  
  
 if isinstance*(entity, WorldCollider):  
 *if self*.COR > 0:  
 bounce = *abs*(*self*.velocity.x) \* *self*.COR  
 *if* bounce > 1:  
 *self*.velocity.x \*= -1 \* *self*.COR  
 *else*:  
 *self*.velocity.x = 0  
 *else*:  
 *self*.velocity.x = 0  
 *else*:  
 colHandler.AddCol(*self*, entity)

Snippet of the collision code in this version of the game

This is illusory though, as in reality the objects simply resolve collisions repeatedly for as long as they are connected, and just rebound with smaller and smaller velocities, giving the illusion of a “pushing” effect. There are no forces being applied, and as a result, a 150kg and a 5kg ball behave the same way after being in this “pushing” state with another PhysObject for long enough – they just end up with lower and lower velocity until reaching 0m/s. In addition, several bugs are present, objects teleport to the wrong sides of others because of how collisions are resolved, using troublesome trace methods and inconsistent velocity checks.

As a result, the final game and its levels are built around this limitation, and it is very difficult to replicate because of how unlikely it is for two objects to maintain such a state for long enough to expose this bug. It does not affect the player’s ability to complete any levels.

### Moments

Moments are not included in the game. They are mathematically complex to implement, and while the concept of a moment is simple to understand in A Level Physics, applying the rules of moments to objects and determining the resulting movement is outside the scope of this project.

More importantly, rects in Pygame are impossible to rotate and adapting these objects to a force application system in which forces can be applied to and from different parts of objects would be extremely difficult and function very poorly. The simulation of rotational forces is unnecessary as it can be built around, but its implementation *would* allow for interesting new features and open the door to a plethora of new puzzle and level types.

### Performance

Due to the game’s complexity, lower-end systems will struggle to run the game at a smooth framerate. There are limited solutions available to counter this, as Pygame itself is inherently unoptimised.

One possible solution could have been not drawing objects which are outside the boundaries of the screen as they will not be seen by the player anyway. The effectiveness of this is questionable though, as most of the resource demand comes from the sheer number of objects on the screen and calculations required for each object. This is especially true with particle effects (physics-enabled particles being the worst culprits) where a level could have hundreds or thousands of particles with their own motion values being tracked all at once. These would cause CPU/memory bottlenecks, so selective drawing may not improve the performance on a system with a

Such issues would not exist in a more powerful game engine, and CPU/memory bottlenecks are difficult to deal with in a language such as python without closer memory management features. Small improvements may have been produced by avoiding square root functions in certain parts of the game, e.g the Vec2 object has a method to retrieve its squared magnitude rather than the true magnitude.

## Maintenance

### Adding New Features

Text

Description automatically generatedA picture containing text

Description automatically generatedThe game was programmed to enable easy maintenance through its object-oriented code. Different sections are split into different classes and link to each other logically – an IDE can visualise the structure of the main two files: physics.py and main.py.

The main advantage of this is that separate components of the game can be changed without making laborious changes across the entire codebase relating to that specific functionality. For example, polymorphism and inheritance are used extensively in the code, most base objects include “Update” and “Draw” methods which are shared by both their subclasses and most other objects. As a result, the main game loop does not have to differentiate between every single object type and can call these general methods instead. Additionally, any changes to the object’s behaviour can simply be added to these methods and will be reflected immediately in the game without having to modify the outer game loop.

As well as this, the fundamentals for any new features involving physics are present within the engine. For example, when adding the AirStream class, the specifics of applying and evaluating forces was abstracted away as the code could just reference the ForceManager object and utilise its built-in methods, such as .AddForce().

### Adding New Levels

The number of levels currently in the game are few, and more levels would be needed to make the game ‘release quality’. Making new levels follows a straightforward process and can be achieved with a photo editing tool. Rect coordinates can be derived from an image manipulation application while the level is being created.

A screenshot of a computer

Description automatically generated with medium confidence

Photoshop displays the selected rectangle’s topleft corner’s coordinates

Levels are automatically detected by the level select object and are easily loaded as long as all the level data is valid and in the correct format.