1. Compilation Instructions

Don’t Drown was made using the Processing library for Java, as well as the Minim library [1]. Required .jar files, as well as the licenses for Minim and the font used in the game, are included in the lib/ folder of the submission.

Commands to be run from the DontDrown/ folder:

```
    $ javac -cp lib/minim/*:lib/core.jar:src/ src/*.java
    $ java -cp lib/minim/*:lib/core.jar:src/ DontDrown
```

The class diagrams for my codebase (excluding my defined Enums) are included as a separate PDF. They were generated from the .class files, and as such may vary slightly from the .java files.

2. Design & Implementation

- Genre: vertical scrolling platformer.
- Player character (PC): a circle (Figure 1).
- Opponents: drowning.
- Mechanics:
    - Standard platforming elements of jumping and landing.
    - Force resolution for character movement.
    - A wave that pursues the player up the level.
    - The stress of being close to the wave affects horizontal steering.
    - Per-level debuffs that primarily affect the stress mechanic.
- Goals:
    - Reach the top platform of each level ahead of the wave.
    - Deviate from the optimal path to collect tokens along the way.

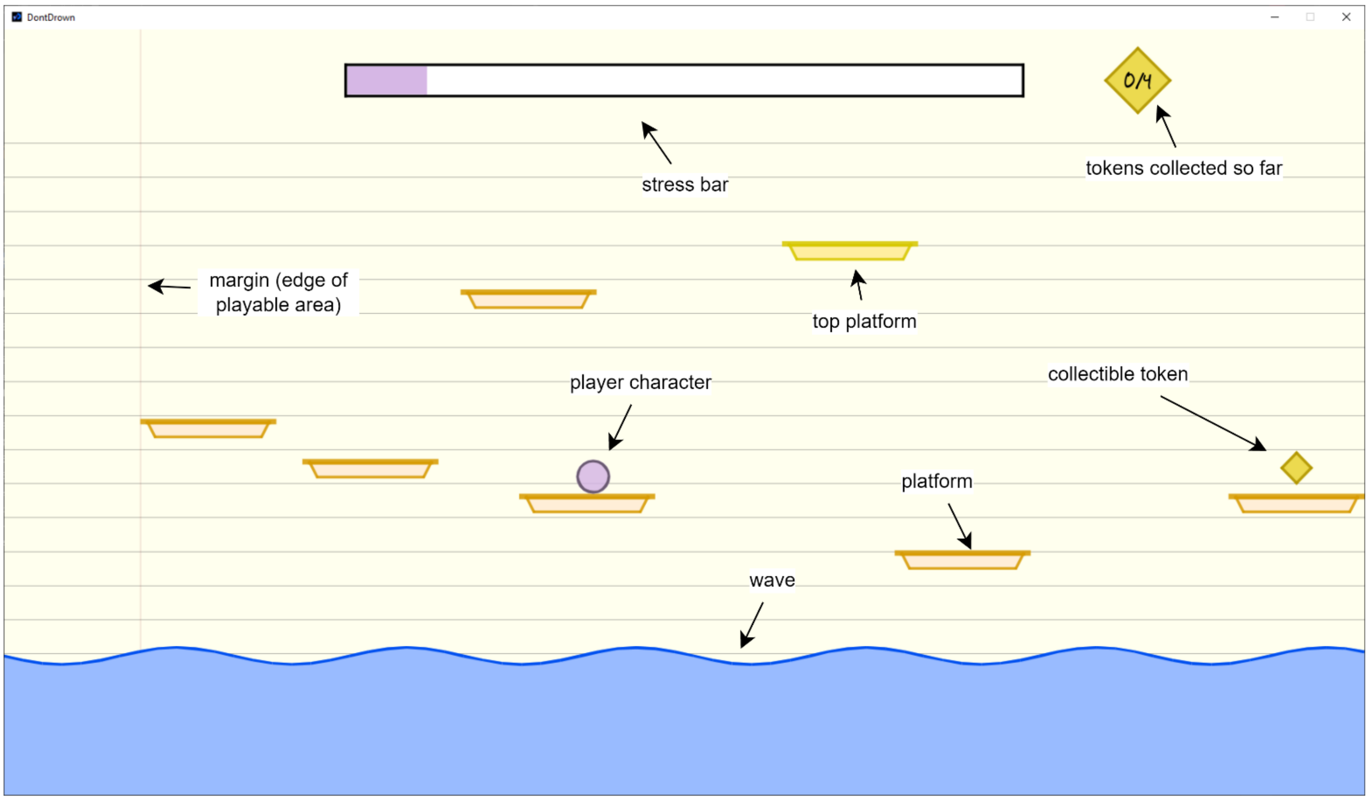


Figure 1

2.1. Platforming

2.1.1. Horizontal Thrust and Drag

I chose to implement force resolution for the PC, but my way of doing so is odd in a number of ways. Firstly, I modelled horizontal friction/drag as a force, and its magnitude is not related to the PC's speed, instead being decided by the current stress level. The maximum and minimum values of the drag force are calculated at runtime so that the PC will take 5 and 20 frames to come to a complete rest from full speed at minimum and maximum stress respectively (see Equations 1, 2 and 3). The sense of slippiness I wanted from high stress is why I chose to model drag as a force independent of velocity.

$$(1)\ v = u + at, v = 0 \rightarrow a = \frac{v}{t}$$

Where v is the final velocity, u is the initial velocity, a is the acceleration, and t is the time spent accelerating

$$(2)\ F = ma$$

Where F is the accelerating force, and m is the mass of the PC

$$(3)\ (1) \text{ into } (2) \rightarrow\ F = m \times \frac{v}{t}$$

The magnitude of drag increases linearly between the stress effect threshold and the maximum stress value. The magnitude of the horizontal thrust force is calculated similarly, such that the PC takes between 25 and 10 frames to reach maximum horizontal speed from rest. I originally had the drag force apply whenever the PC was in motion, and accounted for the drag forces in calculating the magnitudes of thrust to ensure that the thrust was always sufficient to accelerate the PC, but the difference between drag values was greater than that between the thrust values, which meant that the intended minimum thrust exceeded the maximum thrust. Instead, the drag force only applies when the player is either steering against the PC's horizontal velocity or when the player is not steering at all. The steering state (left, right or neither) and horizontal movement state (at rest, accelerating or at max speed) are tracked as two Enums and used to essentially form a finite state machine of horizontal movement.

The maximum horizontal speed is determined at runtime, and if it is reached then the move state is set to max speed and remains there until the player either stops steering or starts steering in the other direction. When at max speed, neither thrust nor drag is applied to the PC, to save on redundant calculations.

Both drag and thrust are applied when the character is mid-air, but at significantly reduced magnitudes, which forces the player to take runups for big jumps, and to identify oversteering early in a jump to correct it in time.

2.1.2. Vertical Thrust, Gravity and Drag

Jump state is also tracked with an Enum, and different values of gravity are applied depending upon state. When the PC is on a platform, at the very peak of a jump, or in coyote time (discussed later) then the gravity applied is 0. When the PC is on the upwards arc of a jump, the gravity is slightly less than the gravity applied when the PC is falling, which allows the PC to travel more than half the horizontal distance of a jump before starting to fall, which allows for long jumps that seem impossible when one assumes a normal parabolic arc (which I find quite satisfying to complete when playing).

The impulse force of jumping is statically defined, with a multiplier being used during force resolution to scale it based on window size. At runtime, the number of frames of the upwards and downwards arcs are calculated, so that the peak jump height and approximate maximum horizontal distance traversable in the time it takes to fall back to the same height (i.e. the jump range) can be determined for the sake of level generation. A multiplication-based drag is applied to the vertical velocity after force resolution if the PC is falling so that there is a terminal velocity (although in practice this is unlikely to be reached before colliding with the wave).

The impact of the lowered gravity when moving upwards is complemented by 3 frames of hang time at the peak of a jump, wherein the PC moves neither upwards nor downwards. It is not so long as to feel oddly floaty, but again helps the player land difficult jumps without making them feel as though their hand is being held. Coyote time and early jump frames achieve the same goal. ‘Coyote time’ is a term I discovered when watching a YouTube video about platformer design [2], and is a reference to Wile E. Coyote briefly hanging in the air before realising he had gone over the edge of a cliff in the Road Runner cartoon; in Don’t Drown, the PC will not start falling and will still be able to jump for a couple of frames after leaving the edge of a platform. Early jump frames allow the player to press jump up to 5 frames before actually landing on a platform, and they will jump as soon as they land, giving players an approximately 0.08s margin of error to “perfectly time” a jump.

2.1.3. Level Generation

Levels are generated randomly at runtime. I originally had intentions of consistent levels between different runs of the program, but creating serializable versions of the involved Processing classes would have taken too much time. The compromise I reached is that levels are consistent per execution of the program, which in honesty I do not find to really be a shortcoming: it allows a player the satisfaction of completing the game (provided they do not overthink that feeling), but it will still be a new challenge next time they play it. I did have an option to regenerate the set of levels from within the game, but I felt that made the falsehood behind “completing” the game too transparent, so I replaced that option with an arcade mode that generates an endless collection of random levels.

Level generation essentially takes two parameters: debuff (discussed later), and difficulty (easy, medium, hard, or very hard). Difficulty affects:

- the overall height of the level,
- whether or not the first platform is the full playable width,
- the speed at which the wave rises,
- and the verticality of the level (i.e. the average density of platforms per unit of height).

Within the code it can also affect the maximum rate at which tokens are spawned, but in practice I found that this was not necessary, so each difficulty actually has the same spawn rate for tokens (in terms of the minimum number of platforms there are between tokens).

The width of the first platform and the speed of the wave are only changed for easy levels, and are consistent between medium, hard and very hard levels. The height of a level increases the difficulty simply because the player must make it further without making mistakes. The impact of verticality is somewhat more nuanced: in a level with less verticality, the PC must jump between more platforms per unit of height gained, and because the wave moves constantly upwards, this means that the player has less time to waste between jumps.

In level generation, jumps between platforms are split into three categories: a reflection jump, a vertical jump, and a horizontal jump (see also Figure 2, wherein the centre point of the top edge of the next platform can be anywhere within a marked area, and jump range is not to scale with jump height, and Figure 3):

- A vertical jump places a platform in the top quartile of a jump height above the current platform, and in the lower quartile of a jump range to the left or right of the current platform. Vertical jumps cannot occur twice in a row, and their frequency is determined by the verticality of the level’s difficulty.
- A reflection jump places a platform in the top quartile of a jump height above the current platform, and between a half and a whole jump range to the left or right of the current platform. Reflection jumps occur when the current platform is so far to the left or right that another platform would not horizontally fit between the current platform and the edge of the playable area.
- A horizontal jump places a platform between a quarter and a half of a jump height above the current platform, and between a half and a whole jump range to the left or right of the current platform. Horizontal jumps are the most common, and happen by default.

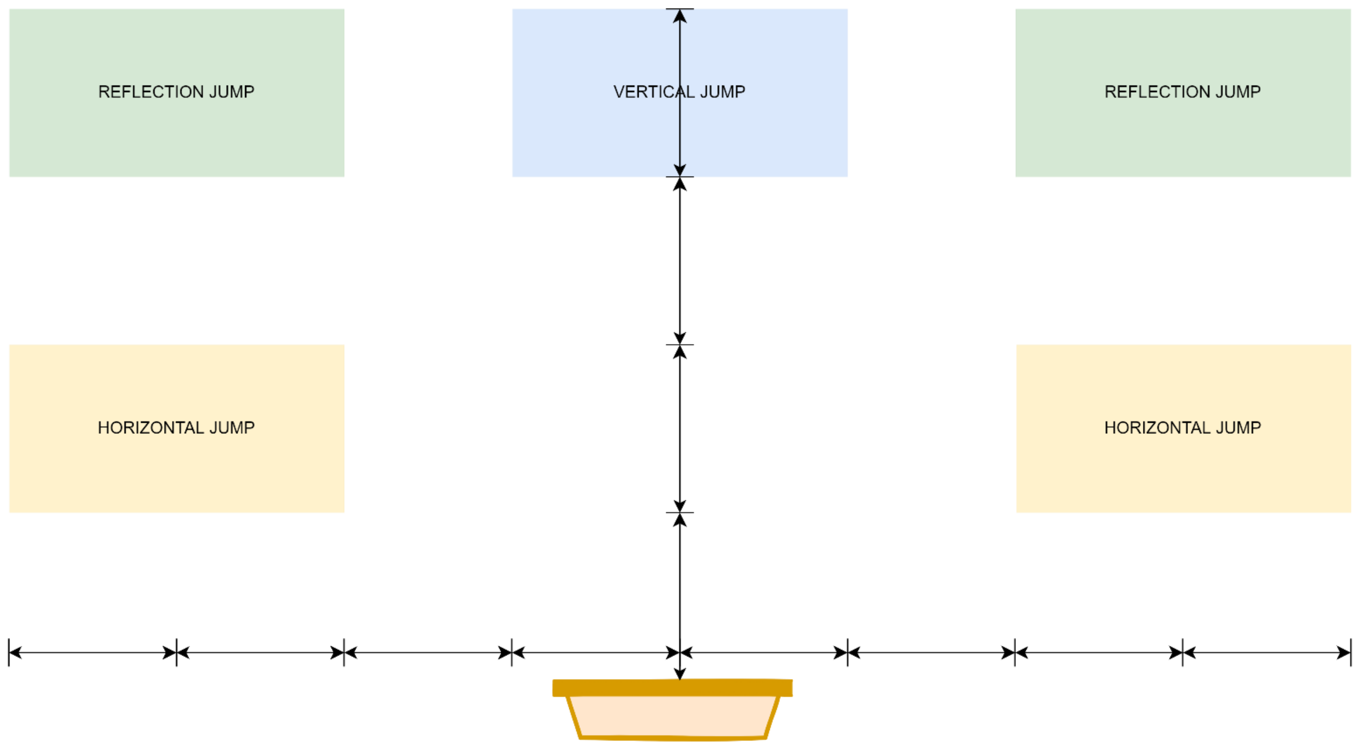


Figure 2

Platforms are generated as a series of jumps until a platform is placed within 0.75 jump heights of a top limit dependent upon level height. In this way, a path that is guaranteed to be navigable is generated, such that opportunities to skip over platforms are rare but not non-existent. In playtesting, my housemates derived particular satisfaction from managing to skip platforms. There is a low chance of the path changing horizontal direction before a reflection jump becomes necessary.

Concurrent with the horizontal jump immediately after a vertical or reflection jump, a red herring platform is placed (provided enough non-red herring platforms have been placed since the last one). A red herring platform is at the same height as the platform derived from the horizontal jump, but is on the opposite side to the previous platform (see Figure 3), such that landing on both will usually force double-backing without a gain in height, which slows the PC down in their race to the top of the level. Tokens are placed on red herring platforms, so that there is an incentive for the player to take a non-optimal path to the top: this is in large part what adds interest to the race away from the wave. If the player does not attempt to collect tokens then it is quite easy to complete a level well ahead of the wave, but it gets significantly harder if they want to get every token.

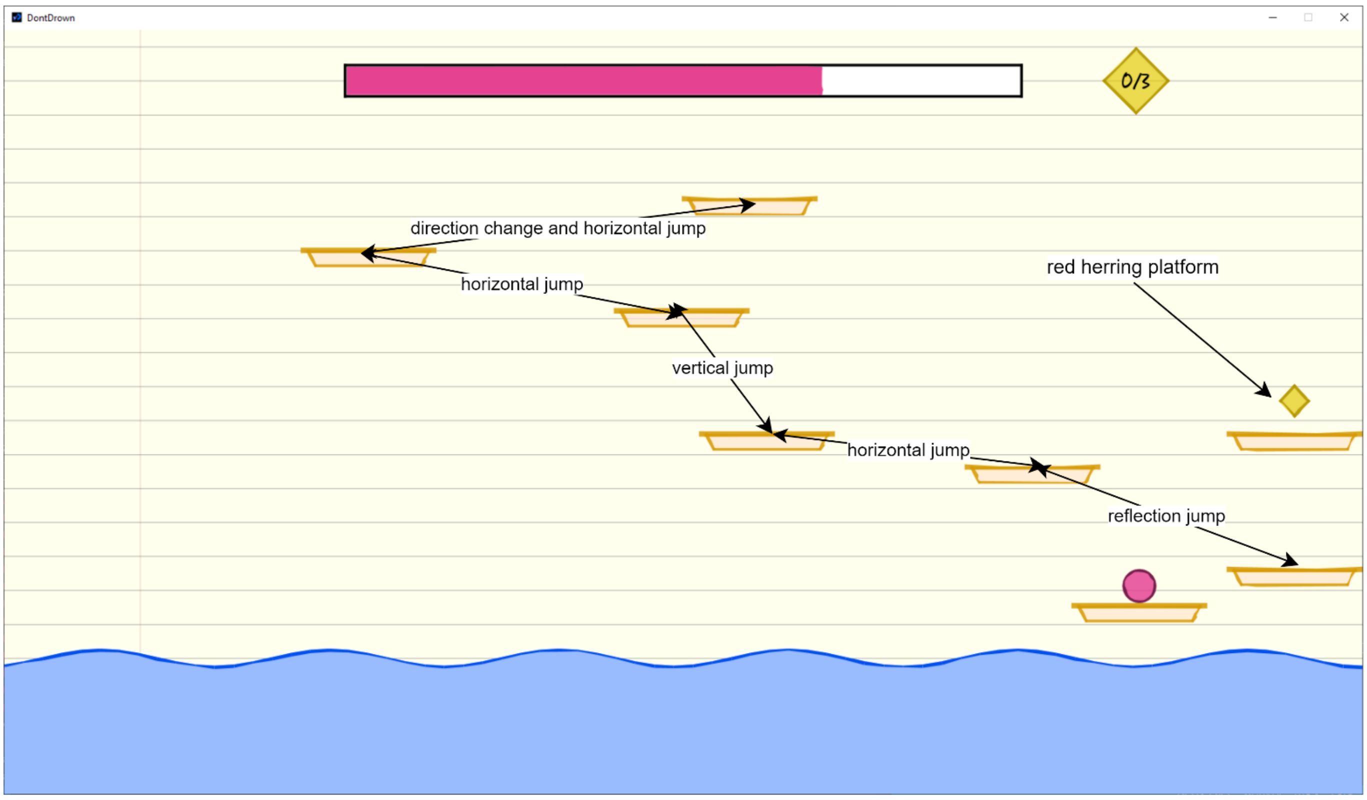


Figure 3

2.1.4. Panning

To handle levels being taller than the application window, panning was implemented. The top and bottom of the screen are padded, and if the PC is within two jump heights of the top padding or one jump height of the bottom padding then panning occurs. Padding occurs at either the vertical speed of the PC, or at a minimum panning speed, which helps smooth the panning out across the course of a jump. Panning is reversed when a level is completed, so that platforms and tokens are correctly placed when the level is replayed.

2.1.5. Collision Detection

There are two types of collision detection: collision with platforms, and collision with tokens. Both types of collision vertically sort the relevant colliders and cut off search after the first object that is higher than the PC for the sake of efficiency. Both types of collision detection consider the path between the PC's previous position (i.e. at the last round of collision detection) and the PC's current position, so that the PC cannot pass through objects when moving at high speeds.

Platform collision detection only occurs when the PC is falling: the PC passes through the underside of platforms when travelling upwards, and if the PC is on a platform (and therefore at vertical rest) then it maintains a pointer to that platform, and a different type of collision detection is performed to check if the player has moved off the edge of that one platform. If a platform is between the PC's previous and current positions, then the horizontal position of the PC at time of vertical overlap is calculated and compared to the horizontal position and width of the platform. To account for purely horizontal movement, the horizontal component of the PC's previous and current positions are also compared to the platform.

I should note that the vertical component of the PC's previous position does not account for the radius of the PC, but the vertical component of the PC's current position does. Ideally, the previous position would account for some amount of the PC's radius, as it can at times seem like the PC glitches upwards through a platform if they are just below it when they start to descend (e.g. at 3:00 in the video), but when I try to account for it the PC will pass through platforms if they are falling too quickly for reasons I cannot explain, and I consider the upwards glitching to be the lesser of two evils. One of my housemates said they felt like a speed-runner exploiting bugs when they managed to “glitch” up through a platform, so at least there is some fun to the discrepancy.

Collision with tokens is modelled as circle-to-circle collision detection, with the added complexity of determining the point along the path between the PC’s previous and current position that is closest to (the centre of) a token. I did try a primitive form of ray-casting that modelled both the PC and token as diamonds and checked for overlap between the token and the dotted lines of Figure 4, but that was complex to calculate, surprisingly error-prone, and theoretically not much more accurate than approximating the tokens as circles (where ray-casting underestimated the area of the PC, circle-to-circle overestimates the area of tokens (see Figure 4)), so I opted to use the circle-to-circle method instead.

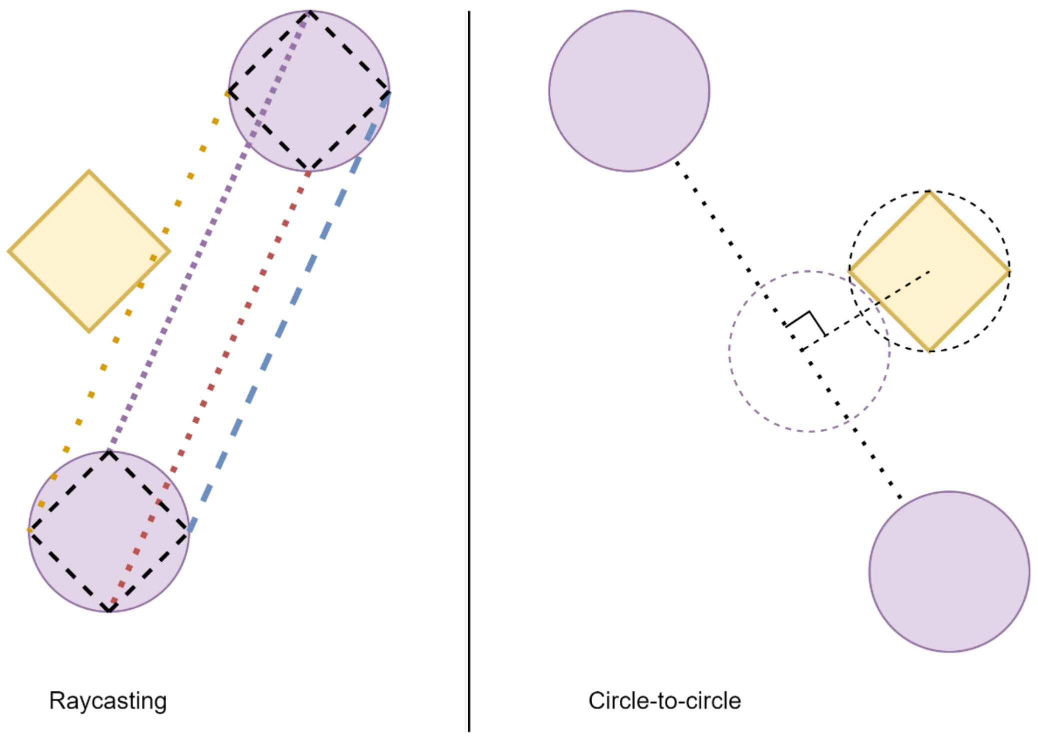


Figure 4

2.2. Stress

What sets Don’t Drown apart from other vertical platformers is the stress mechanic, which is to my knowledge a novel feature (although I do not claim to have played an exhaustive collection of platformers). The degree to which the PC is stressed affects the difficulty of the platforming by changing the magnitude of the horizontal thrust and friction forces, discussed in more detail later. Aside from the mechanical impact, higher stress levels also affect the rendering style of the game (compare Figure 5 with Figure 1) and make the background music play faster, to ensure that the player is as stressed as the PC. The stress has no effects beneath a threshold of 20%, but this is not communicated to the player; again, this is to make life somewhat easier without the player realising, and to improve the accuracy of the metaphor.

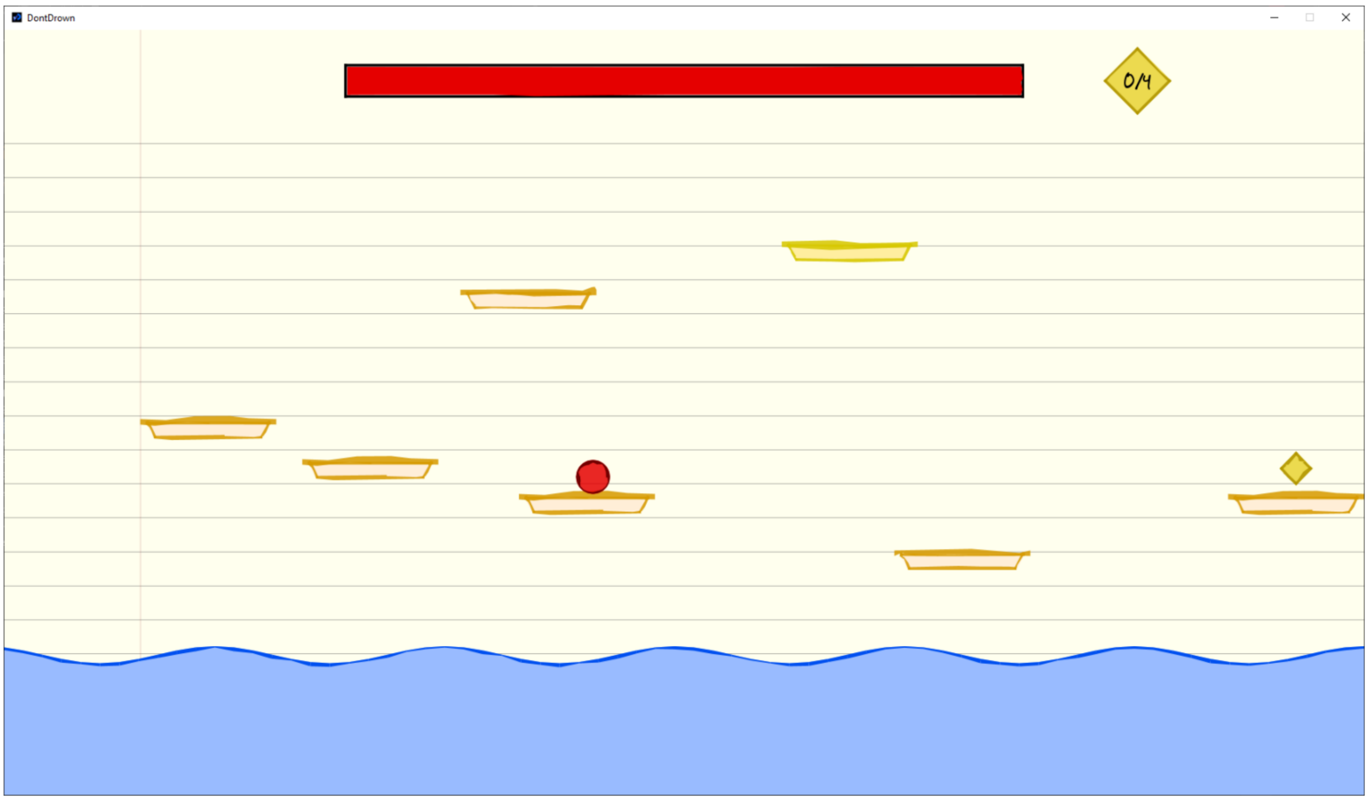


Figure 5

2.2.1. The Wave

The wave rises at a constant speed during a level, and is slower for easy levels (as mentioned previously). Unlike the PC, platforms, and tokens, the wave is opaque, to communicate to the player that they cannot dip beneath it. It also extends past the margin of the page, again setting it apart from other game elements. Mechanically, the wave is quite simple: for the sake of collision detection with the player, the wave is modelled as a straight line, which means that the player can dip very slightly into peaks of the wave. This is not a bug: I find it satisfying to make a jump just in time and break the top of a wave peak, and my play-testers agreed.

When the PC is within a certain distance of the wave, the stress increases in inverse proportion of the distance between the PC and the wave. If the PC is further away from the wave, then stress decreases at a rate proportional to the distance between the wave and the PC (with an upper cap on destress rate to prevent instantly de-stressing with two vertical jumps).

2.2.2. Steering

As previously mentioned, when the PC is highly stressed, they accelerate more quickly and decelerate slower. This makes it easy to oversteer, but also reduces the amount of runup needed for long jumps. I chose this to be analogous to throwing oneself into work without much forward planning when stressed in real life.

2.2.3. Rendering Style

In the pitch, I designed the game to look hand drawn on lined paper, to frame it as absent doodles when on is trying to get work done. To achieve this in Processing, I implemented a PApplet subclass, Sketcher, that can draw lines and some primitive shapes in a "hand drawn" style. It does this by treating edge lines as polygons; essentially very thin rectangles to start with. The top and bottom edge of the rectangle are broken into one or more sections depending upon a shakiness parameter, and each breakpoint is deviated from the original straight line by a random amount based on a variability rate parameter and the intended thickness (i.e. weight) of the line, as shown in Figure 6. For curved shapes, because they are already split into a number of short straight lines, each straight-line section is just made to thicken or widen evenly along its length, with one edge remaining even.

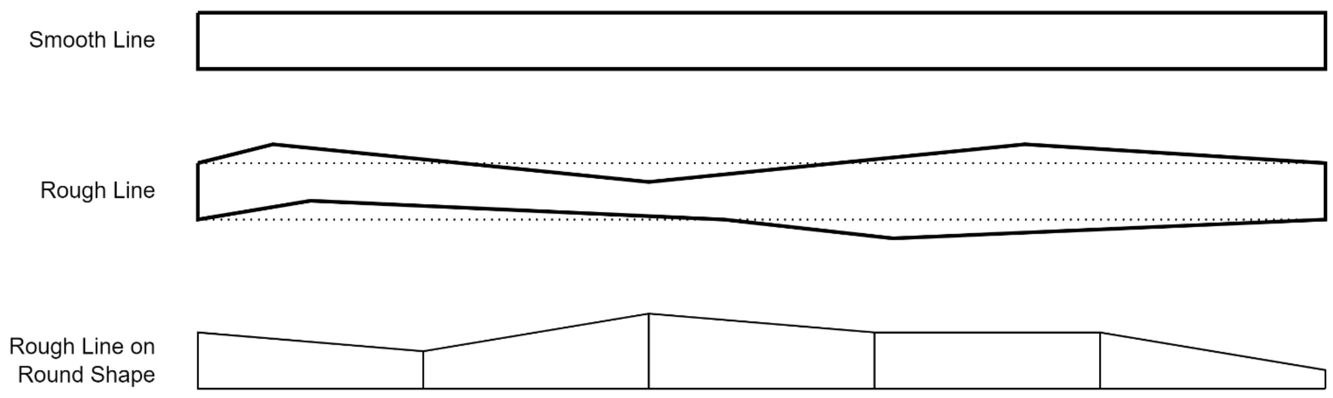


Figure 6

The PShape generated by the hand drawing method is a composite object of the edge polygons and a smooth-sided fill shape. Implementing the sketchy colouring in shown in the pitch seemed too complex for too little benefit to warrant implementation.

Every subclass of AbstractDrawable generates a set of PShapes at runtime (referred to as tokens within the codebase, which admittedly might be confusing) with different degrees of shakiness and variability that relate to stress values, so that when stress is high the drawing quality appears to be worse. There are multiple PShapes per stress index for each drawable class, and items will iterate through them at a rate dependent upon stress: at low stress, the drawing quality is high and objects rarely redraw themselves; at high stress, objects are redrawn frequently and poorly. The process of generating the PShapes is too slow to occur multiple times per frame, which is why the sets of PShapes are generated during start-up ("redrawing" really refers to iterating over the set of generated PShapes). To prevent all objects redrawing at the same time, every object has a random frame offset at which to redraw itself. To prevent objects lagging behind when stress rises suddenly, they also redraw themselves if the stress changes by a large amount between scheduled redraws.

The PC and stress bar's fill also vary their colour by stress value, and share a colour palette. The stress bar's fill is unique in that it redraws itself much more frequently than other objects, to ensure that it fills and empties smoothly.

The platforms have two static sets of PShapes: one for standard platforms and one for the highest platforms of levels, which are a different colour. The wide platforms on easy levels have their own array of PShapes.

One drawback of the hand drawing I have implemented is that it does not handle curved lines particularly well: along the edge that the line segments vary their weight they do not meet correctly, either leaving a gap or extending past one another (see Figure 7). I could not find a satisfactory way to correct the overlap; it can be somewhat unsightly, but I do not find it to be a significant issue.

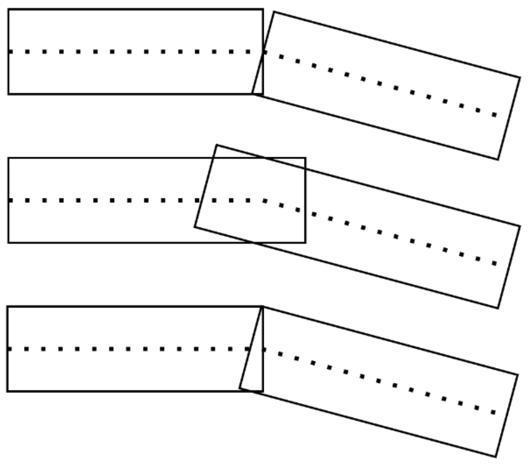


Figure 7

2.2.4. Music

The tempo of the music will increase with stress, as mentioned. I used the Minim library for Processing [1] to implement the music, which is a simple synthetic rendition of the tune of the alphabet song/Twinkle Twinkle Little Star. I chose that tune primarily because the original author is unknown so copyright would not be an issue, although I do think its place in school (albeit primary school) makes it somewhat fitting for the theme of Don't Drown. Given how short the melody is it can become annoyingly repetitive, so it can be paused at any time via the Settings menu, but I recommend playing at least some levels with it.

Really the music acts as a proof of concept: an original score devised by someone with more musical talent than me could be easily integrated into the game with no change to the code that affects the tempo. The tempo of the Minim AudioOutput is never changed, rather a multiplier for note duration is derived from the stress before queuing a note, and the timestamp for that note's end is recorded and used to trigger the playing of the next note. The music speed is the only element of the game not tied to framerate, because otherwise the music at low framerates would be somewhat unbearable.

2.2.5. Debuffs

Acting as a layer of complexity on top of the platforming and base stress mechanic, the game levels are grouped by the applied debuffs, of which there are seven (including a lack of debuff):

- Feeling Typical is the in-game label for levels with no debuff applied.
- Overworked is the only debuff to affect level generation: every platform except the first and last has a token on it, and red herrings are still generated. This means that in order to get every token on the level a player cannot skip a single platform and in practice my housemates considered this to be one of if not the hardest debuff(s).
- Panic Prone makes the stress level increase once every few seconds regardless of wave distance. This was also considered to be difficult to deal with because the steering can change quite significantly mid-jump.
- Stress Motivated makes the steering sluggish when stress is too low, to the point that longer jumps are impossible to achieve, which forces the player to stay close to the wave. A player can get used to steering at high stress, but staying so close to the wave still reduces their margin of error in terms of jump timings.

- Can’t Unwind prevents the stress level from ever decreasing. This may be the least impactful debuff to experienced players, because if you just max out your stress early then the steering is consistent for the rest of the level.
- Tunnel Vision is my personal favourite debuff in terms of novelty, as it restricts the player’s view of the screen to a vertical slit that extends one and a half jump heights above the PC and one jump height below the PC, making it harder to plan routes and to land recoveries from visiting red herring platforms (as the lower platform will go out of sight at the peak of the jump). Additionally, the wave does not stress the player unless it is within the reduced field of vision, but increases it very quickly when it is visible, and stress only starts to decrease when the wave has been kept out of sight for a number of frames proportional to the value from which the stress is decreasing.
- Lacking Self-awareness removes the effect of stress on everything except steering, and hides the stress bar, so the player must try to keep track of how stressed the PC is based only on where the wave is and how aggressive the steering is.

The effect of the debuffs aside from Overworked are implemented in the StressAndTokenState class, which handles stress-based calculations.

2.2.6. Extensions and Repercussions

As explained in the game guide, the player can use the spacebar to briefly halt the wave once per level (referred to as an extension in the codebase). However, it is a double-edged sword, because when the wave resumes its movement it travels faster for as long as it was paused, exactly fast enough to make up for the pause (referred to in the codebase as the repercussion), so it does not grant extra time to deal with the later platforms. I find that it can be necessary to use the extension for some levels, as seen in the video demonstration.

2.3. Menus

When not mid-level, the cursor is enabled to allow the player to navigate menus. I used the sf-grunge-sans.bold.ttf font [3] because I thought its handwritten aesthetic matched the rest of the game. Most of the menu pages are statically defined, but the Level Selector is generated during start-up and updated during runtime to reflect high scores for each level. The hitboxes for clickable elements extend to the end of the line, and on hover the clickable elements are highlighted green for ease of use. The Settings menu could be improved by updating to reflect the chosen framerate and whether or not the music has been paused, but due to how I have implemented the menus it is surprisingly complex to do so, and I did not consider it a priority.

A debugging overlay can be activated at any time with Shift + D, and when debugging is active some cheats can be used (see the codebase for more), but in my experience the overlay tends to have noticeable impact on performance, so it is not recommended for gameplay.

From the Settings menu it is possible to cap the framerate at 60, 45, 30 and 15 (60 is default). I added this feature because both myself and my play-testers have had a lot of time to get to grips with the controls, and I am somewhat concerned that a marker might become frustrated if the game we find to be engagingly challenging is instead frustratingly hard. The speed of all game elements is tied to the framerate, so capping it lower will make the game slower and easier to play. By tying speed to framerate I also avoid the problem of lag spikes teleporting the PC across the screen, although such spikes are rare when neither running the game via my IDE’s debugger nor screen-capturing.

In the Level Selector menu, the maximum number of tokens collected for each level is shown, as well as the seconds remaining for that run of the level. The seconds remaining are calculated based off the time it would have taken the wave to reach the top platform, accounting for framerate. Changing the framerate partway through a level (including via lag spike) can cause incorrect values to be calculated and displayed. Because repercussions make up for extensions, they are not accounted for in time remaining calculations, which can cause also incorrect values if the repercussion has not finished when the level is completed (as seen at 1:35 of the video).

3. Context

Don’t Drown is a vertical scrolling platformer, of which there are many others. As stated previously, the novel feature of Don’t Drown is the stress mechanic, around which much of the game is built. One can see similarities with Doodle Jump, in that they are both vertical scrollers wherein the objective is to go upwards, and going back down will kill you, but the specifics of each game make them distinct. Like Don’t Drown, Rainbow Islands featured a wave that would kill the player on contact, but in that game the wave was not the main focus, and it only appeared after quite some time, such that a player could complete a level without ever seeing the wave or before it even appeared. Don’t Drown lacks Mario-esque mechanics like enemies and environmental hazards, and aside from the stress mechanic is a very simple platformer.

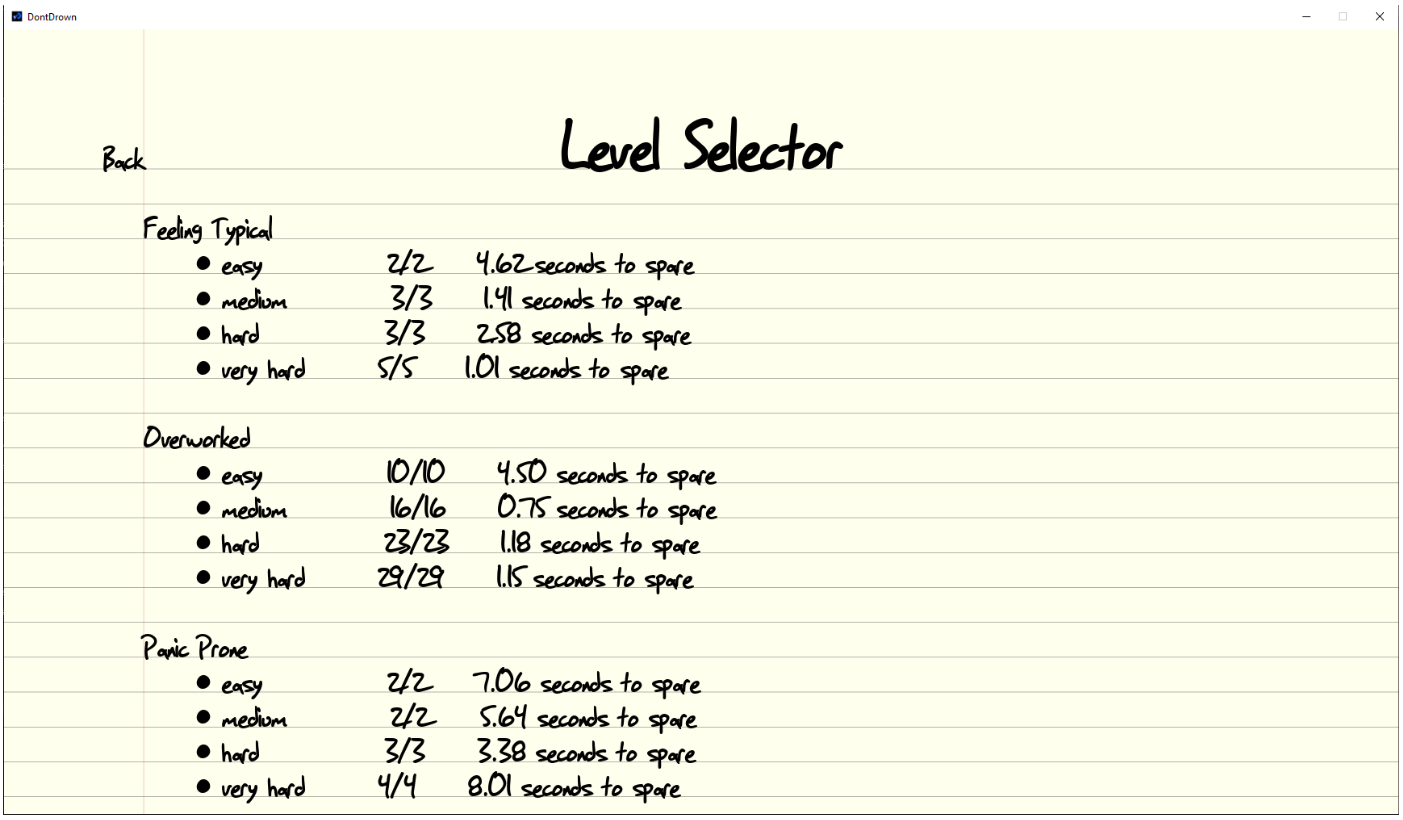
As is openly alluded to in the codebase, game guide and the game itself, Don’t Drown is meant as a light-hearted metaphor for living life, especially life at university: the tokens are tasks to complete (whether coursework or just remembering to get your groceries), and the wave is the inexorable march of time. Trying to stay on top of everything gets stressful, and sometimes in order to make it through you have to allow yourself some extra time, or put aside a task you simply do not have time for. As I developed the game, I always kept how the mechanics tied into that metaphor in mind, and overall I am very happy with how well the game works on that level. Is it a gross simplification of what makes life stressful and how stress affects us? Yes, of course it is: it is meant to be fun, after all.

4. Evaluation

I primarily tested the game by playing it a lot, and getting my housemates to play it. This sometimes showed up bugs to be fixed, but mostly was done to help tune the difficulty of the game. My play-testers agree that the final game is enjoyably difficult and plays well (moves smoothly etc.). Their real-life biases motivated them to strive to collect every single token, and made them reluctant to pause the wave during levels, which I think is incredibly fitting. The greatest compliment paid to the game is that one housemate completed 100% of tokens in a single sitting (see Figure 8, Figure 9 and Figure 10), which took quite a few attempts on harder levels.

On a more technical level, trying to keep operations performant has led me to take some liberties with the object-oriented paradigm. Two key examples are how many class properties are public, and that the class handling stress calculations also houses the current score for a level as it is played, which is a somewhat arbitrary way to separate concerns. However, my code is still readable and works correctly, so I think the compromise between code hygiene and performance is reasonable.

When I pitched the game, I included a number of ideas for powerups (e.g. capping the impact of stress or temporarily increasing jump height) and special platforms (e.g. that dropped away after being landed on), as is common among platformers. In practice, I spent a long time in a feedback loop of modifying level generation, the physics, the stress mechanics and the debuffs to create an enjoyable and challenging experience, so I did not get around to the other features. Some of the debuffs are the inverse of planned powerups, so that planning was not entirely for nothing.


DontDrown

Back

Level Selector

Feeling Typical

● easy	2/2	4.62 seconds to spare
● medium	3/3	1.41 seconds to spare
● hard	3/3	2.58 seconds to spare
● very hard	5/5	1.01 seconds to spare

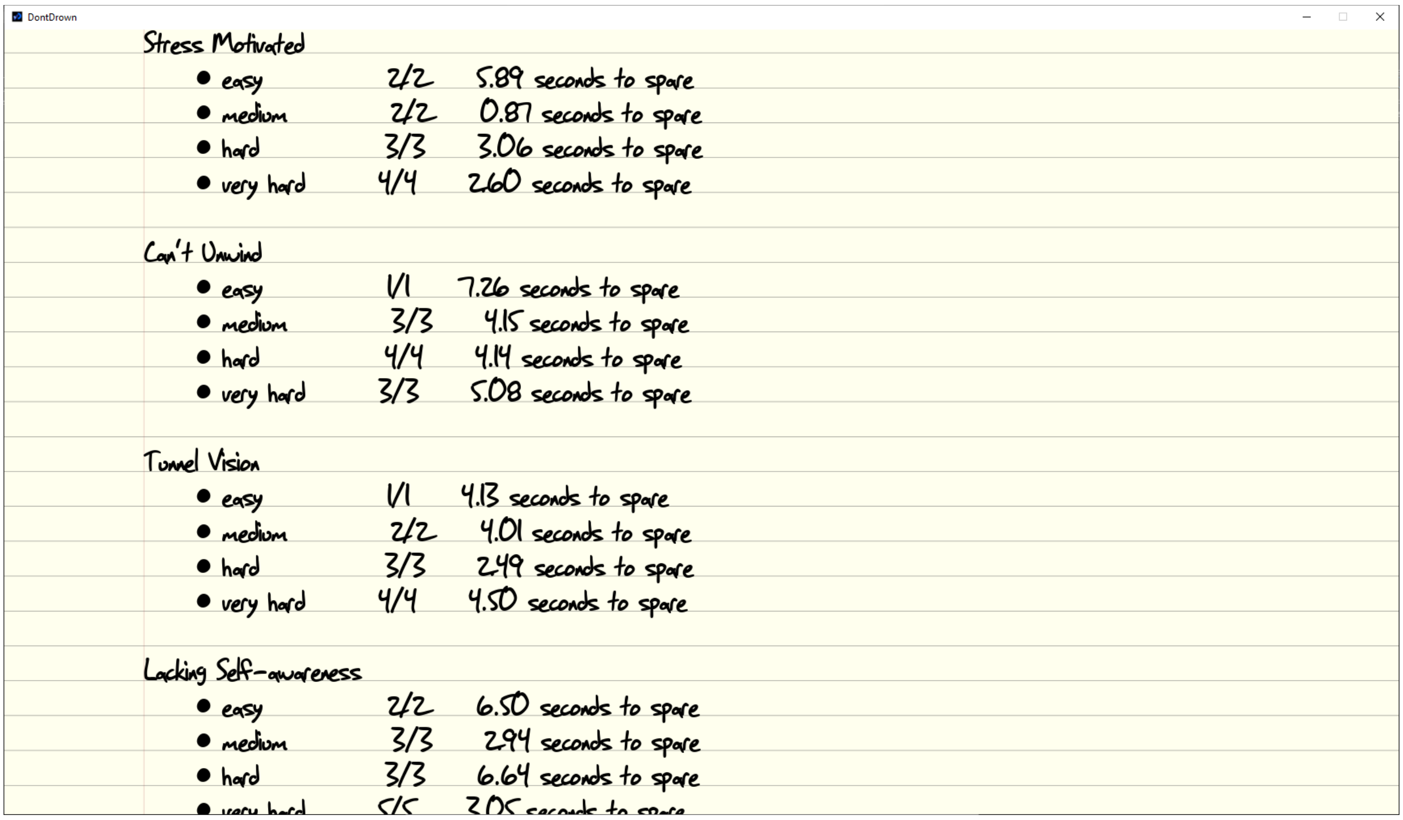
Overworked

● easy	10/10	4.50 seconds to spare
● medium	16/16	0.75 seconds to spare
● hard	23/23	1.18 seconds to spare
● very hard	29/29	1.15 seconds to spare

Panic Prone

● easy	2/2	7.06 seconds to spare
● medium	2/2	5.64 seconds to spare
● hard	3/3	3.38 seconds to spare
● very hard	4/4	8.01 seconds to spare

Figure 8


DontDrown

Stress Motivated

● easy	2/2	5.89 seconds to spare
● medium	2/2	0.87 seconds to spare
● hard	3/3	3.06 seconds to spare
● very hard	4/4	2.60 seconds to spare

Can't Unwind

● easy	1/1	7.26 seconds to spare
● medium	3/3	4.15 seconds to spare
● hard	4/4	4.14 seconds to spare
● very hard	3/3	5.08 seconds to spare

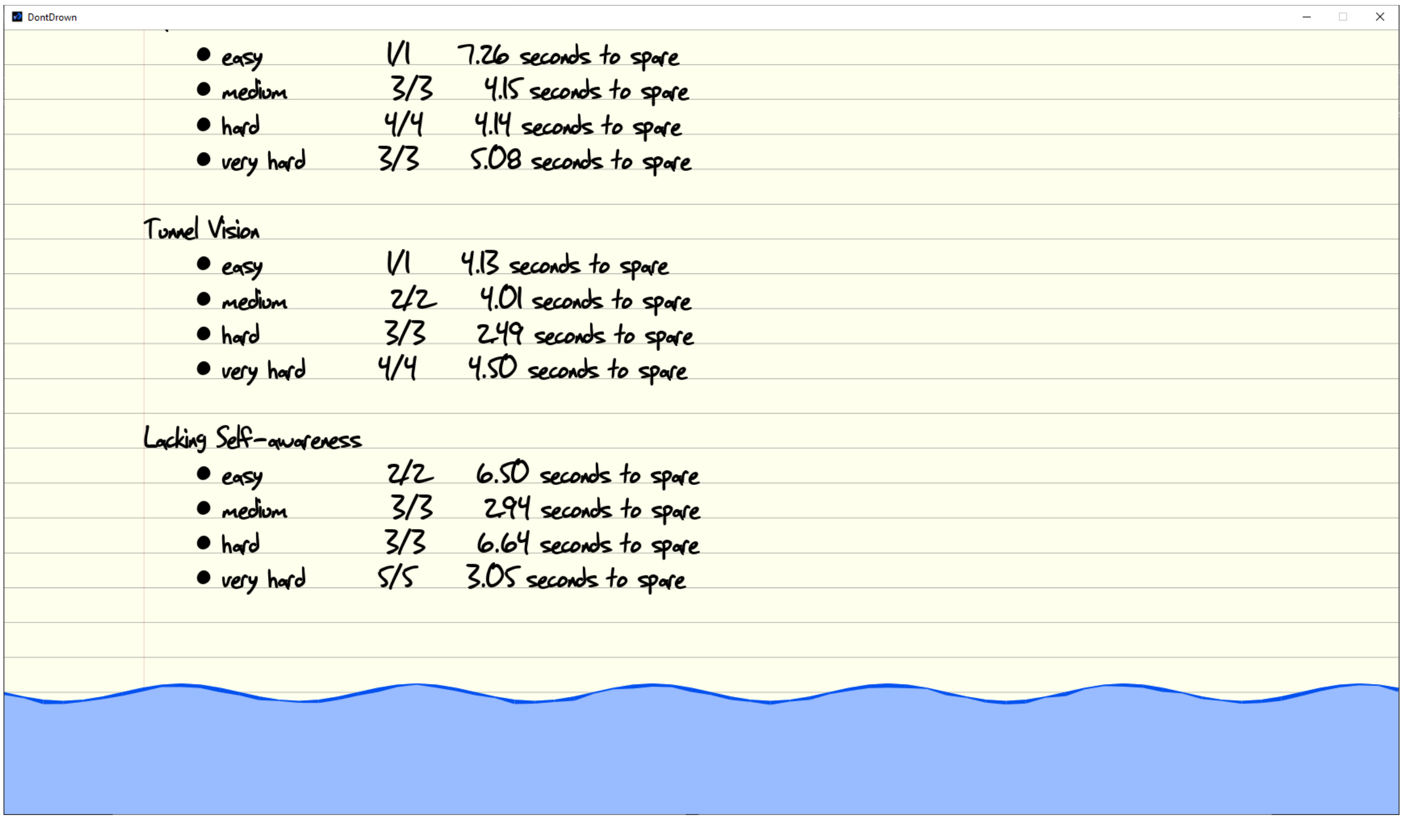
Tunnel Vision

● easy	1/1	4.13 seconds to spare
● medium	2/2	4.01 seconds to spare
● hard	3/3	2.49 seconds to spare
● very hard	4/4	4.50 seconds to spare

Lacking Self-awareness

● easy	2/2	6.50 seconds to spare
● medium	3/3	2.94 seconds to spare
● hard	3/3	6.64 seconds to spare
● very hard	5/5	3.05 seconds to spare

Figure 9



- easy 1/1 7.26 seconds to spare
- medium 3/3 4.15 seconds to spare
- hard 4/4 4.14 seconds to spare
- very hard 3/3 5.08 seconds to spare

Tunnel Vision
- easy 1/1 4.13 seconds to spare
- medium 2/2 4.01 seconds to spare
- hard 3/3 2.49 seconds to spare
- very hard 4/4 4.50 seconds to spare

Lacking Self-awareness
- easy 2/2 6.50 seconds to spare
- medium 3/3 2.94 seconds to spare
- hard 3/3 6.64 seconds to spare
- very hard 5/5 3.05 seconds to spare

Figure 10

References

[1] D. Di Fede, "https://code.compartmental.net/minim/," 3 August 2019. [Online]. Available: https://code.compartmental.net/minim/. [Accessed 10 May 2022].

[2] Game Maker's Toolkit, "Why Does Celeste Feel So Good to Play? - YouTube," 31 July 2019. [Online]. Available: https://www.youtube.com/watch?v=yorTG9at90g. [Accessed 10 May 2022].

[3] ShyFoundry Fons, "SF Grunge Sans Font Family - 1001 Fonts," [Online]. Available: https://www.1001fonts.com/sf-grunge-sans-font.html. [Accessed 10 May 2022].

[4] "Java Program to Solve Quadratic Equation – Javatpoint," [Online]. Available: https://www.javatpoint.com/java-program-to-solve-quadratic-equation. [Accessed 18 January 2022].