# Compilation Instructions

Commands to be run from the Don’t Drown folder:

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

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

# Design & Implementation

## Platforming

### Horizontal Thrust and Drag

I chose to implement force resolution for the player character (PC), but my way of doing so is odd in a number of ways. Firstly, I did model drag as a force, but 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.

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 from rest the PC takes between 25 and 10 frames to reach maximum horizontal speed from minimum and maximum stress respectively. 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 in order to correct it in time.

### 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 rare 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 [1], 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.

### Level Generation

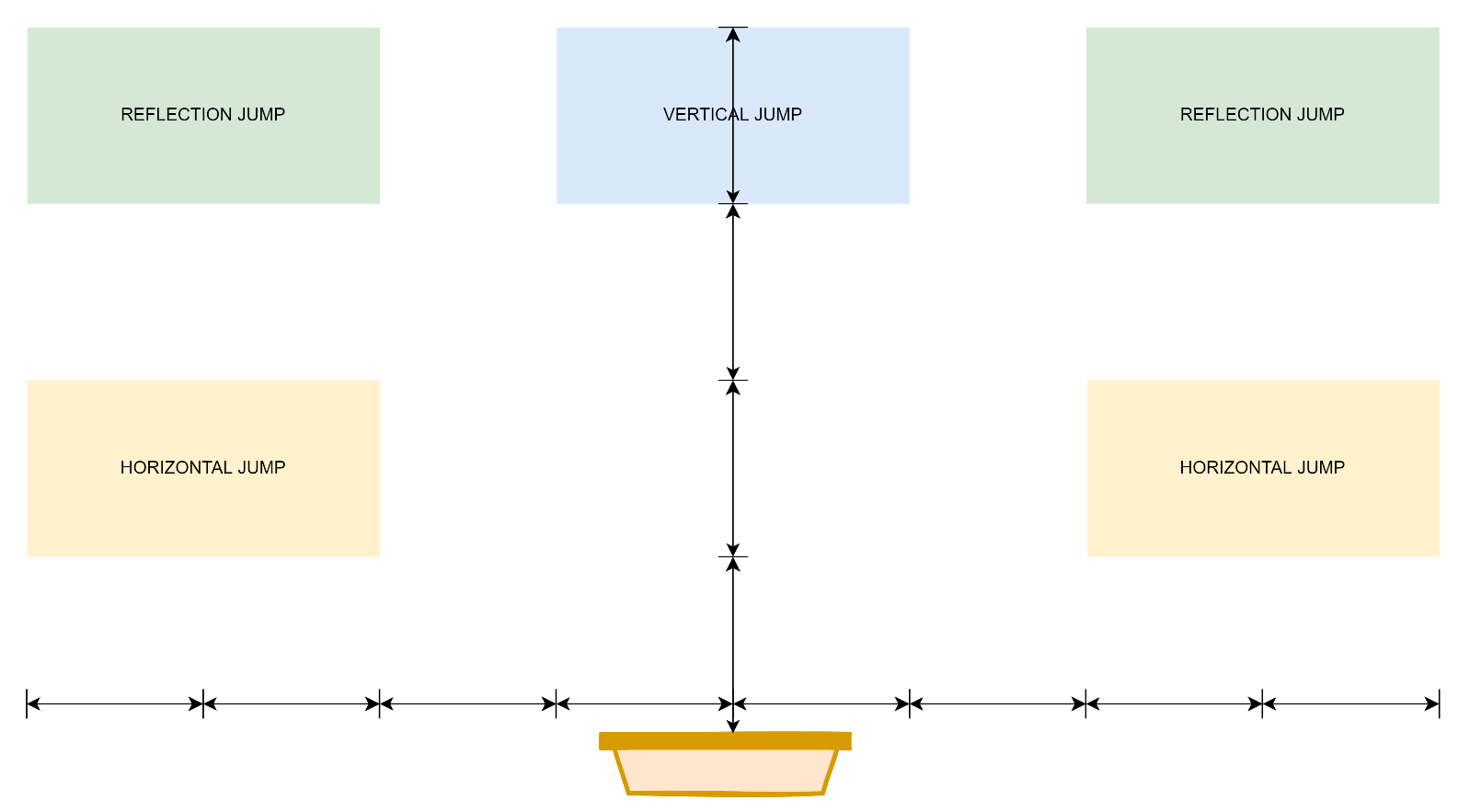
Levels are generated randomly at runtime. I originally had intentions of being able to save levels between different runs of the program, but creating serializable versions of the involved Processing classes would have taken too much time. The compromise I reached is that levels are consistent per execution of the program, which in honesty I do not find to really be a weakness: 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 add 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 and 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). It can also affect the maximum rate at which tokens are spawned (in terms of the minimum number of platforms there are vertically between tokens) but in practice I found that this was not necessary.

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 absolute 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 must spend less time between jumps in order to stay ahead.

In level generation, jumps between platforms are split into three categories: a reflection jump, a vertical jump, and a horizontal jump (see also Figure 1, 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):

* 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 window.
* A horizontal jump places a token 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

Platforms are generated as a series of jumps until a platform is placed within 0.75 jump heights of a top limit which is 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.

Concurrent with a horizontal jump immediately after a vertical 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, 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.

### Panning

To handle levels being taller than the application window, panning is implemented. The top and bottom of the screen are padded by, 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.

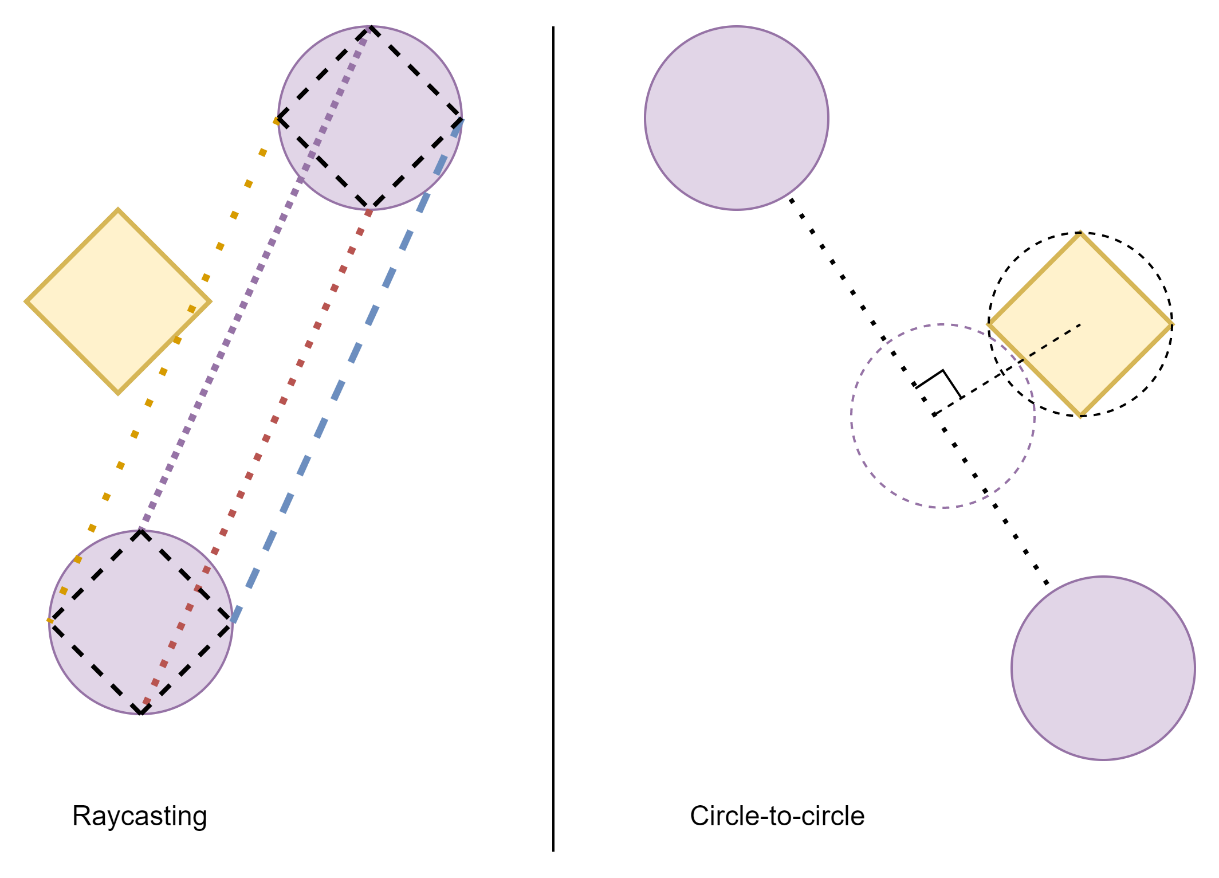
### Collision Detection

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 of the PC between the last set 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 vertical 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 as they start to descend, 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 pull it off, 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 liens of Figure 2, but this was complex to calculate, surprisingly error-prone, and even theoretically not much more accurate than approximating tokens as circles: where ray-casting underestimated the area of the PC, circle-to-circle overestimates the area of tokens (see Figure 2).



Figure

## 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, and make the background music play faster, to ensure that the player is as stressed as the PC.

### 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 would be easy to change and 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.

### Steering

### Rendering Style

### Music

### Debuffs

### Extensions and Repercussions

## Menus

* Jump
  + Single ~~or double~~?
  + ~~Press and hold to increase height?~~
  + Timings
    - Climb
    - Hang time
    - Fall
    - Coyote-time (jumping just after going off the edge of a platform)
    - Delayed jump frames (if you try to jump just before touching the ground)
    - Floaty or grounded?
  + Mid-air velocity change?
  + Maintained horizontal velocity from ground movement?
  + ~~Wall jumping?~~
* Water
  + Start immediately?
  + Constant speed
  + Instant death, ~~or breath holding mechanic~~?
    - Instant death in Rainbow Islands
    - Restart level upon death
  + Makes tokens inaccessible
  + Opaque ~~or translucent~~?
* Movement
  + Acceleration
    - Slow = heavy
  + Top speed
  + Deceleration
    - Slow = slippery, hard to control
  + Acceleration and deceleration frames do not need to be equal
  + Direction change time
* Camera
  + Vertical scrolling
  + Move when player in top (and bottom?) third (?) of screen
  + First level, start with player in bottom of screen, to encourage moving upwards
* Platform types
  + Single-side pass through
  + ~~No pass through~~
  + ~~Drop-away~~
  + ~~Spiked~~
  + ~~Mobile (side to side etc)~~
  + ~~Slippery~~
  + Colour-coding
  + Trampolines
  + ~~Are the sides of the screens walls that can be jumped off/climbed?~~ 
    - ~~Mario, Super Meat Boy, Celeste~~
* De-buffs
  + Loss of colour contrast (makes ~~distinguishing platforms~~ tracking stress difficult)
  + Tunnel vision, but only stress when water within sight
  + Constant stress
  + ~~Powerups increase stress~~
  + Unpredictable stress spikes
  + Become sluggish when not stressed
  + How should de-buffs be communicated to the player?
    - Descriptions are a bit boring, but they may not be intuitive to understand
    - ~~Tutorial level~~
    - Visual or ~~auditory~~ feedback
* ~~Powerups~~ 
  + Temporarily stop the water, but it speeds up afterwards
  + ~~Cap the impact of stress~~
  + ~~Temporarily reduce the impact of stress~~
  + ~~Higher jumps~~
  + ~~In-level pickups, or between-level purchases?~~ 
    - ~~In-level (levels should be re-playable for completionism, the same each attempt)~~
* Stress effects
  + Jerkier movement
    - Decreased acceleration time
    - Increased deceleration time
    - ~~Reduced hang time~~ 
      * ~~Could increase rise and fall time to maintain overall airtime~~
  + Faster music
  + ~~Vignette effect~~
  + ~~Directly proportional to how close the water is~~, or increasing/decreasing over time when the distance is within certain thresholds?
    - The former could be quite jerky, but easier to understand
* Difficulty curve
  + ~~Introduction of new platform types~~
  + Introduction of new de-buffs ~~(and powerups)~~
  + Increased water speed
  + Greater density of tokens in a horizontal space
  + ~~How close platforms are to one another (i.e. platforming difficulty)~~
  + Are levels grouped into “worlds”?
    - By de-buff?
    - ~~By platforming difficulty?~~

# Context

Don’t Drown is a vertical scrolling platformer, of which there are many others. 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.

# Evaluation

Table

Description automatically generatedA picture containing text, receipt, screenshot, document

Description automatically generatedTable

Description automatically generated with low confidence