PI from collisions

Motivation

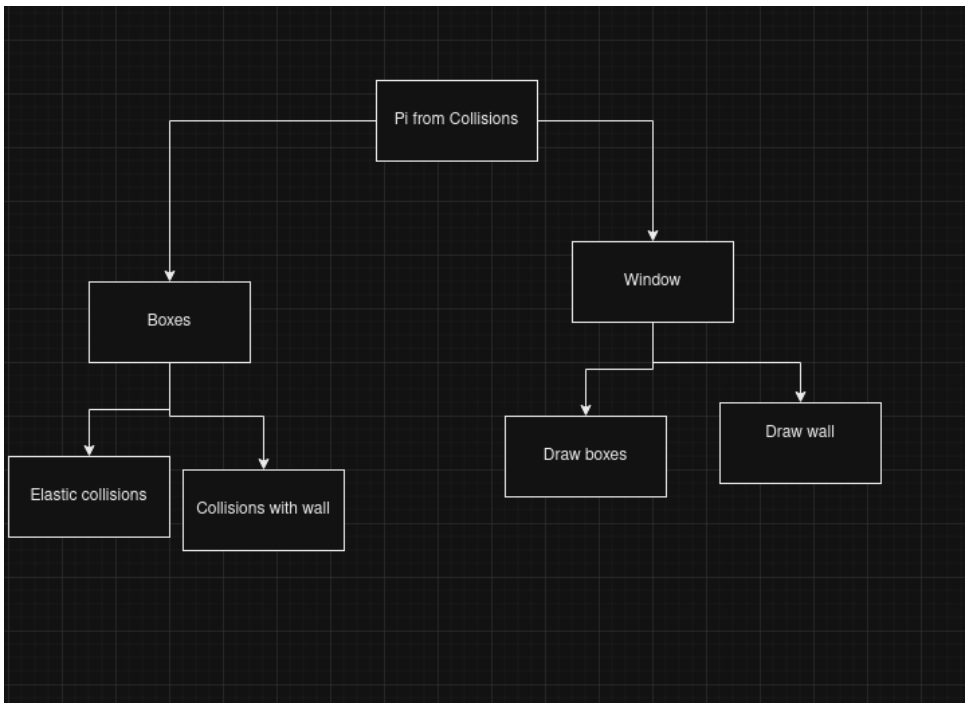
This project was motivated by the idea of creating a simulation that approximates PI (for certain definitions of reasonable), and this specific project arises from a interesting problem postulated by a maths YouTuber named 3b1b.

The premise behind this simulation is as follows. Imagine two blocks, A and B, where the ratio between their masses, denoted m_A and m_B, be 100^N (m_B / m_A). Push the latter block towards the other, assuming a frictionless elastic world, where of A's left hand side there is a wall perpendicular to the ground. From this, the number of collisions within this system must be N digits of π.

I won't delve deep into *why* this occurs, but rather focus on how to *make it occur* (so to speak). In order to learn more, please read the original paper, and said video.

I will be using Python, as well the excellent Manim community edition. I use the former due to its simplicity, and the latter because it might be the only decent "graphics" library available in Python. *Fun fact: The original author of the Manim library was also the person who made the video!*

Ideally, I would like this project to be simple, yet elegant. I do not wish for this program to be a giant *monad of sorts.* What I mean is that, I don't what this program to be a "machine" with many distinct moving parts. Yes, there will obviously be separate classes, functions, but I want them to be self-contained, rather than a "spaghetti" of sorts.



Note that our first prototype will include Drawing boxes, and moving the boxes around. These are chosen, since these are an antecedent to collision detection and resolution, and testing it makes it easier. However, I do talk about resolution of the collision.

Module 1: Rendering An Screen

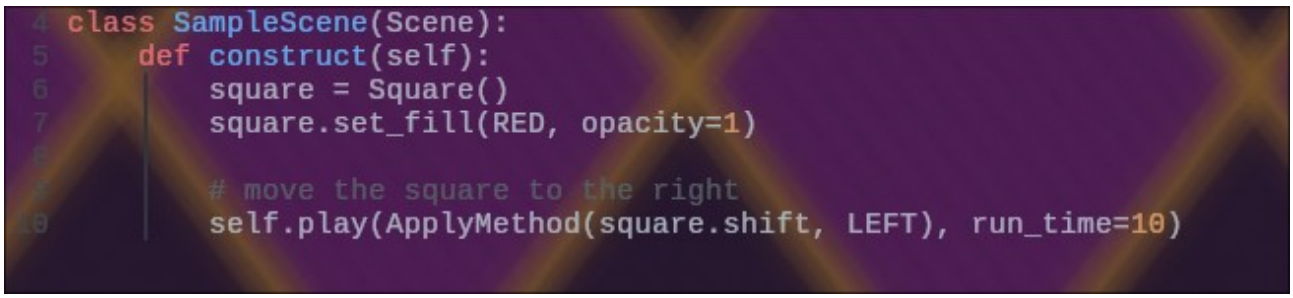
Module 1: Test plan

Now ideally we want the window to remain open until closed, as well as for it to exit when escape it pressed. For a basic window, that is all we want.

TC(n)	Test data	Expected Output	Evidence	Passed?
1	Start window	Should remain open		
2	Start window, press escape	Should close		

Module 1: Development

We would like a window (preferably an openGL instance) of a window. Fortunately, Manim has a feature that allows us to render to a window (rather than to a video, as default):



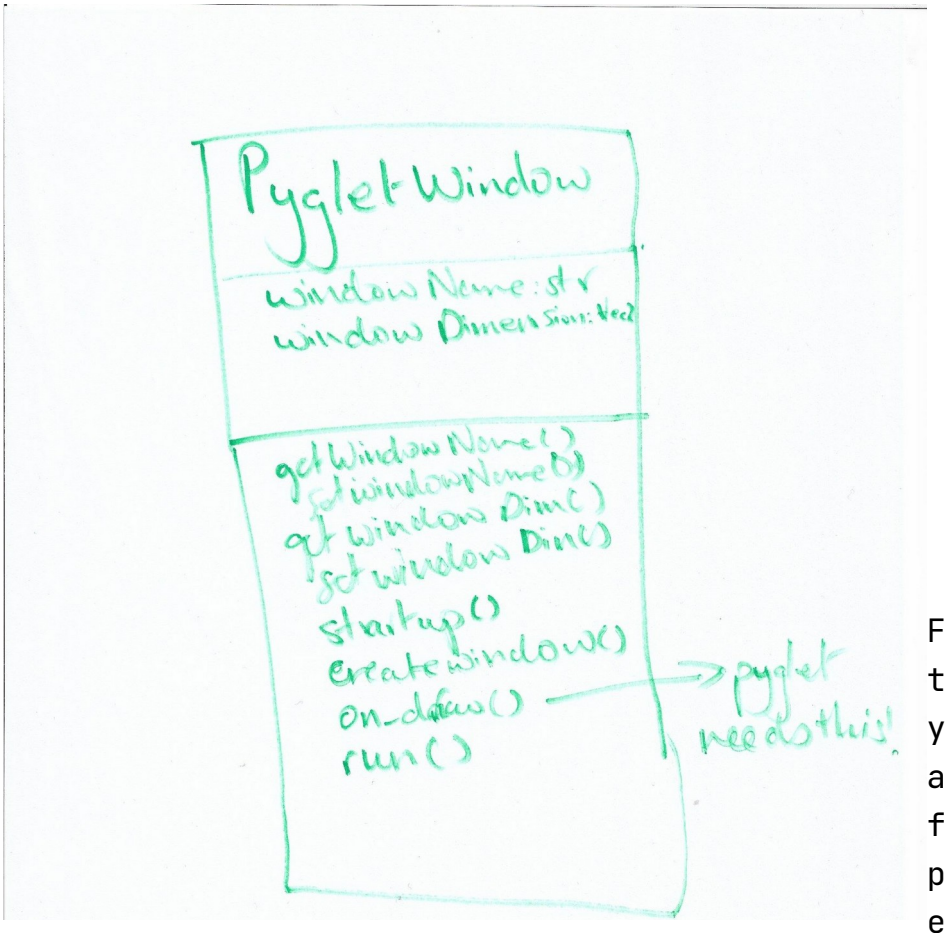
Where we create a sample scene class, and “play” anything that needs to be moved, animating a video, that is then played on top of the opengl window.

However all test cases for this module fail. The reason being manim is not event-driven, rather (as explained), it simply plays a video*. Therefore it is not possible for us to do anything related to events and capturing said events. Maybe we could run the video on a loop constantly (like in a separate pygame window), however, as discussed before, that would mean breaking the simplicity rule, as this would be a crummy solution to a *dead* simple problem. Therefore, I elected to use PyGlet, a Pythonic graphics library.

Pyglet is built in a similar vein to PyGame, however, I chose the former over the latter, due to the former’s better control for delta-time, something which will prove useful in later modules.

Since Pyglet (along with most of python’s libraries) are purely imperative, I will create a window class, in order for ease of

use, as well as clean code. Also, Pyglet's documentation recommend sub classing the window.



Firstly, we have the initializer. As you can see, I have added the ability for the client programmer (me), to expand the input handling, as well as expanding the drawing, since I feel like that is a good way to abstract Pyglet out of the equation in order to progress to the more math heavy code later on.

```
def handleKeyboard(self, key, modifiers):
    if key == pyglet.window.key.ESCAPE:
        self.windowHandle.close()
    if self.keyboardInputHandler is not None:
        self.keyboardInputHandler(key, modifiers)
```

```
class PygletOverride:
    def __init__(
        self,
        windowName: str = "Pyglet Window",
        windowSize: pmath.Vec2 = pmath.Vec2(500, 500),
        keyboardInputHandler: callable = None,      ● Expected type expressi
        drawHandler: callable = None,      ● Expected type expression but ru
    ):
        self.windowName: str = windowName
        self.windowSize: pmath.Vec2 = windowSize
        self.windowHandle: pyglet.window.Window = None      ● Cannot assign
        self.keyboardInputHandler: callable = keyboardInputHandler      ● Ex
        self.drawHandler: callable = drawHandler      ● Expected type expres
```

And this is how we (for example), allow the client to expand their keyboard handling, and of course we have included the criteria for the window to close when escape is pressed as part of our test plan.

```
def createWindow(self) -> bool:
    self.windowHandle = pyglet.window.Window(
        width=self.windowSize.x,
        height=self.windowSize.y,
        caption=self.windowName,
    )
    return True if self.windowHandle is not None else False
```

This is how we create a window. Note that we also have checks to see if Pyglet failed with its initialisation.

Module 1 Testing

For the test code, I have written only 3 lines, one to initialise the class, one to initialise the window, and one to start pyglet proper.

```
69 testWindow = PygletOverride()
70
71 testWindow.startUp()
72 testWindow.run()
```

TC(n)	Test data	Expected Output	Evidence	Passed?
1	Start window	Should remain open, until closed		
2	Start window, press escape	Should close		

Note: evidence needs adding

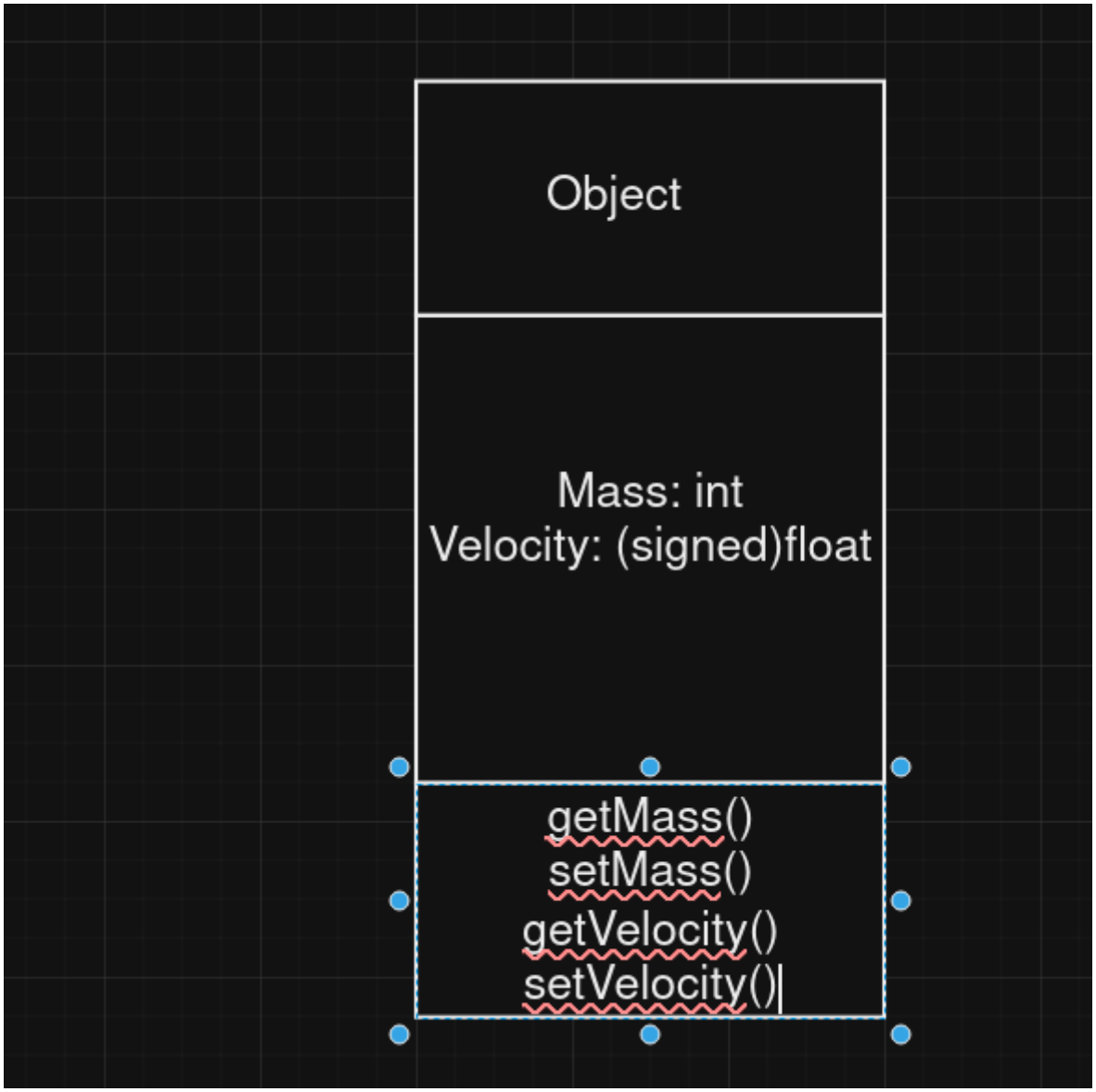
Module 2: Conserving Momentum

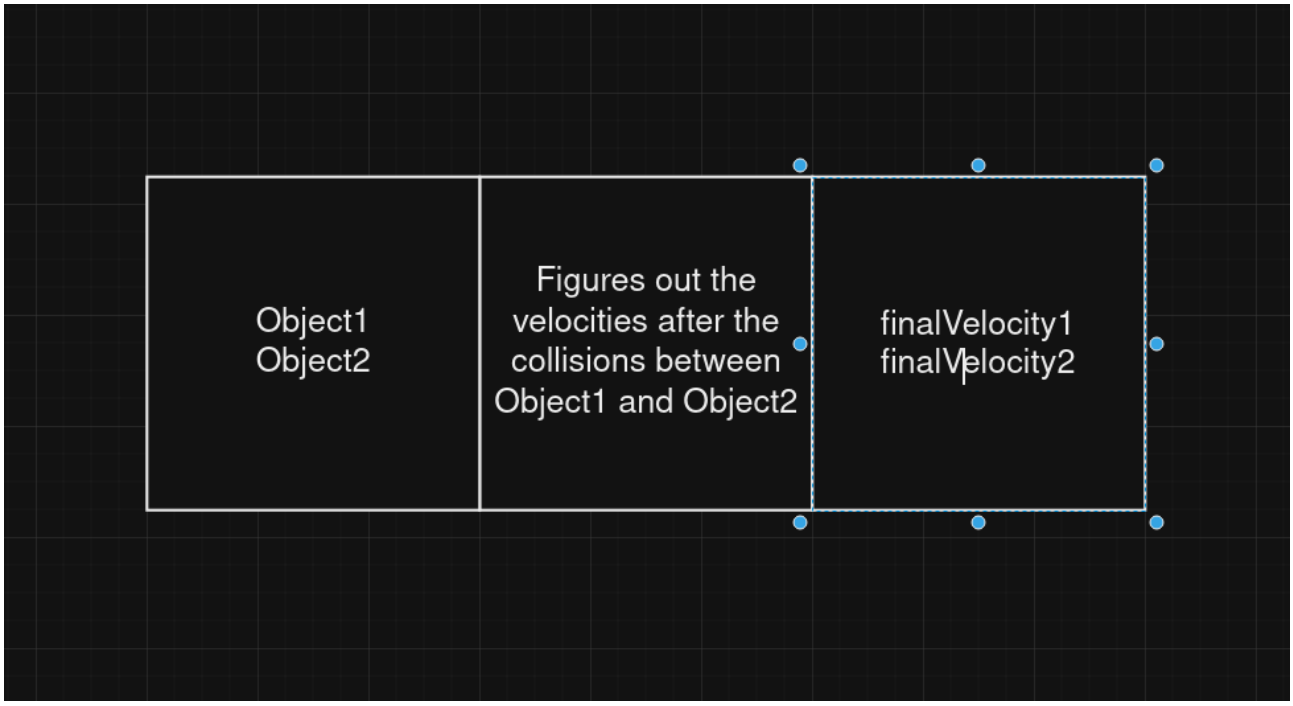
Now, we enter the fun part of the project, by conserving momentum. We need to do so, in order to satisfy the first premise of our problem.

$$
\begin{aligned}
v_1 &= \frac{m_1 - m_2}{m_1 + m_2} u_1 + \frac{2m_2}{m_1 + m_2} u_2 \\
v_2 &= \frac{2m_1}{m_1 + m_2} u_1 + \frac{m_2 - m_1}{m_1 + m_2} u_2.
\end{aligned}
$$

These equations are used in an elastic environment, where both kinetic energy and momentum are equal before and after any collision. V_i describes the final velocity (after a collision) of the i^{th} particle, m_i the same for mass, and u_i for initial velocity.

In foresight, it might be prudent for us to represent the **current** velocities and masses of our particles in some sort of structure, in order to make legibility and further expansion easier.





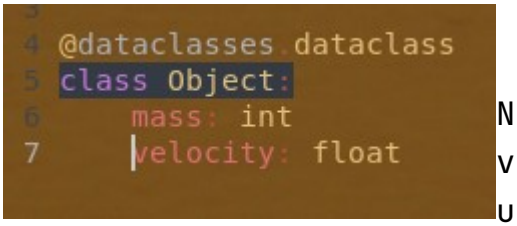
Module 2: Test plan

Therefore, our test plan is as follows:

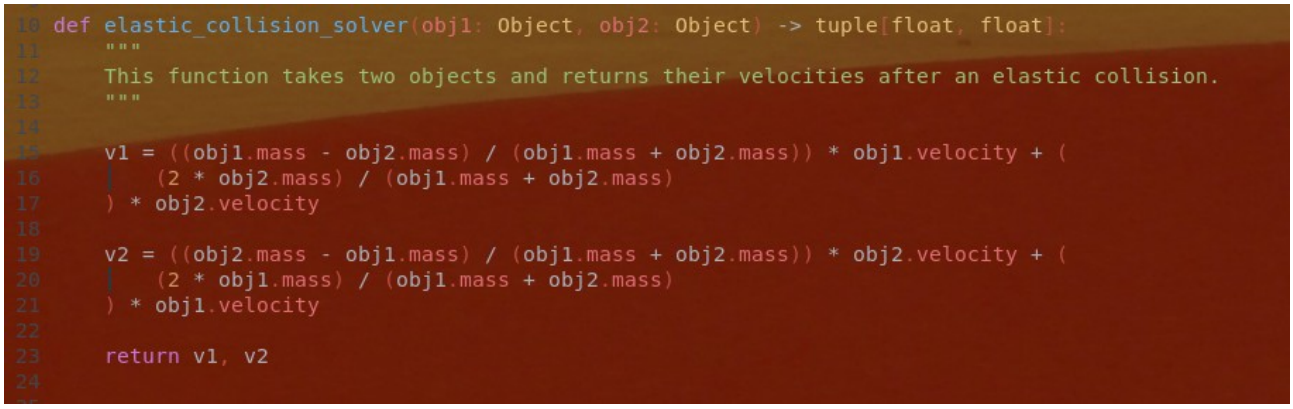
TC(n)	Test name	Test data	Expected Output
3	test_collision_with_equal_mass	Object(1, 2), Object(1, -3)	(-3, 2)
4	test_collision_with_different_mass	Object(2, 4), Object(1, -3)	(0.5, 3.5)
5	test_collision_with_one_stationary_object	Object(2, 0), Object(1, -3)	(-3, 0)
6	test_collision_with_opposite_directions	Object(1, 2), Object(1, 3)	(3, 2)
7	test_collision_with_same_directions	Object(1, 2), Object(1, 2)	(2, 2)
8	test_collision_with_large_mass_difference	Object(100, 5), Object(1, -3)	(-3, 5)
9	test_collision_with_zero_velocity	Object(2, 0), Object(1, 0)	(0, 0)

Module 2: Development

I decided to program the Object as a data structure rather than a class, since I felt that since this object is really here to ***store data***, there isn't an advantage of creating a class with a constructor, as this is only here to store data in an organised container. If this object had more than getters and setters as its method, a class would have been beneficial. Using a data structure not only is faster, but is also more legible.



Note that we only use one dimensional vectors, as our boxes will only exist upon a line, and won't be flying. If we needed the y component, we could use sin(velocity) to yield our y component.



Then we have the solver following the equation above. Note that we use type hinting of a tuple in order to return both velocities.

Module 2: Testing

I used unittests for this module, as firstly, they provide comprehensive tools for testing in a orderly manner. Secondly, we are testing a (mathematical) function, which requires more thorough testing.

Writing the test was rather difficult with python's module structure, therefore I ended up with this structure (ignore pycache):

where we need __init__.py in each subfolder to indicate its module status. Very annoying initially, and will remain annoying.

TC(n)	Test name	Test data	Expected Output	P?
3	test_collision_with_equal_mass	Object(1, 2), Object(1, -3)	(-3, 2)	
4	test_collision_with_different_mass	Object(2, 4), Object(1, -3)	(0.5, 3.5)	
5	test_collision_with_one_stationary_object	Object(2, 0), Object(1, -3)	(-3, 0)	
6	test_collision_with_opposite_directions	Object(1, 2), Object(1, 3)	(3, 2)	
7	test_collision_with_same_directions	Object(1, 2), Object(1, 2)	(2, 2)	
8	test_collision_with_large_mass_difference	Object(100, 5), Object(1, -(-3, 5) 3)		
9	test_collision_with_zero_velocity	Object(2, 0), Object(1, 0)	(0, 0)	

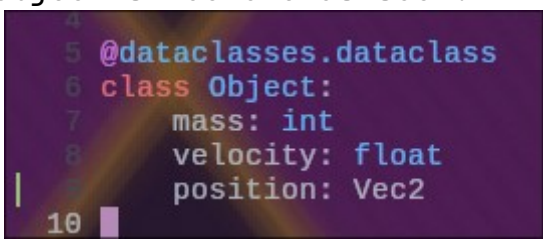
The evidence is in the appendix :)

Module 3: Drawing some boxes

Before we detect and solve collisions, I will take a detour to render the objects.

In this module I would like to create a subroutine that can effectively render a rectangle based upon the object's position. I will cover how to render and compute the movement of the object in the next module.

In order to do so, we need to add a position variable onto our Object structure as such:



Now we could have had a 2d position variable, which is a ratio of how far across the object is from the screen, however computing the collisions of the boxes would get needlessly complicated. Regardless, this is an unnecessary undertaking, since we are only saving around two bytes of memory, and it is not like python is known for its glorious lack of memory use.

I also would want a way to change the colour of the boxes, maybe depending

Module 3: Test plan

TC(n)	Test Name	Test data	Expected output
10	Box with red colour	Object(1, 0, Vec2(400, 600)), (255, 0 , 0)	Red box at 400, 600
11	Same Box but blue	Object(1, 0, Vec2(400, 600)), (0, 0, 255)	Blue box at 400, 600
12	Blue Box but different location	Object(1, 0, Vec2(200, 700)), (0, 0, 255)	Blue box at 200, 700
13	Blue Box but different mass	Object(10, 0,	Larger blue box

Vec2(200, 700)), at 200, 700
(0, 0, 255)

Module 3: Development

Module 3.1: Slight Detour

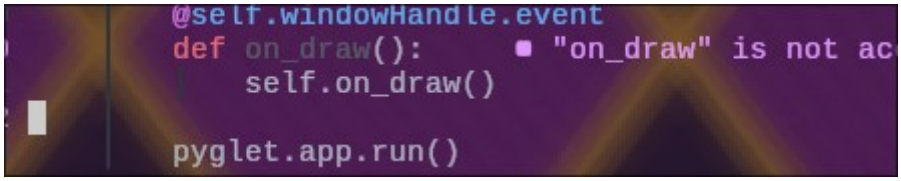
While writing code for this project, I realised that I had written the *PygletOverride* class wrong. Here's why:

Above the is the way I wrote the run function, where I call the pyglet function to essentially finish its overhead checks, and start rendering the window.

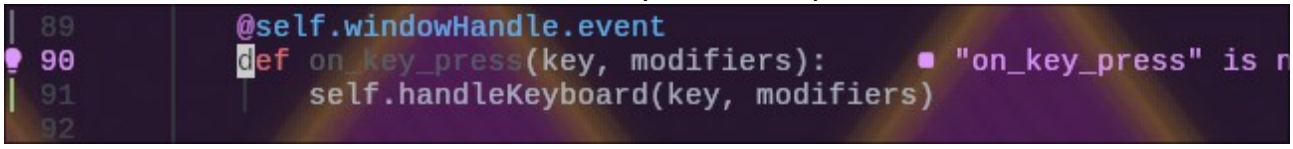
What I didn't realise was that pyglet had no way of knowing what my draw function was (as it was a separate function is the class (as shown previously)), as they say in their own documentation:

> The Window dispatches an `on_draw()` event whenever it's ready to redraw its contents. pyglet provides several ways to attach event handlers to objects; a simple way is to use a decorator:

So in order to remedy this issue, I changed the run function to as follows:



This needs to also be done for keyboard input.



For same reasons as above.

Yes, its a function inside a function, however this works since we have a safe way to setting the decorators for the on_draw and on_key_press event.

Module 3.0: Continuing On

Creating a subroutine within utility.py should suffice, as this function only really is that, a utility program.

Here is our function signature as dictated by our tests.

```
8 def draw_object(obj: Object, colour: tuple):
```

```
    shapes.Rectangle(
        obj.position.x, obj.position.y, obj.mass * 100, obj.mass * 100, colour
    ).draw()
```

Then we have the actual drawing. Note that we multiply the mass by 100, this is because our mass is in Kg (adhering to SI units), therefore in order for the object to be visible, we need to multiply by some constant. This could be changed if the constant is too big.

Module 3: Testing

To test the code I have created a sample window in the tests directory, and wrote the following code for *testing*.

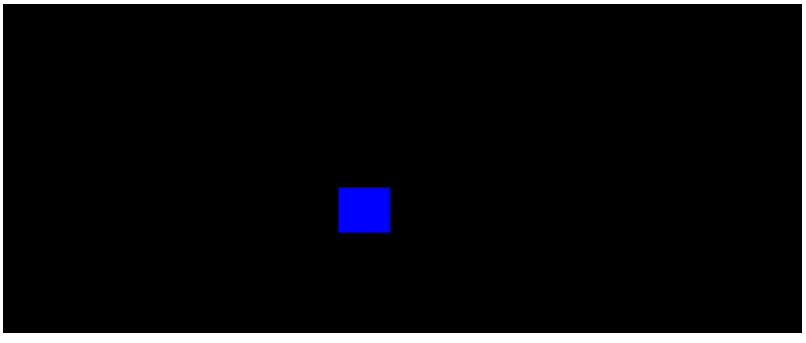
```
window = PygletOverride("Tests for boxes", Vec2(1920, 1080))


boxes = [
    [Object(1, 0, Vec2(400, 600)), (255, 255, 0)],
    [Object(1, 0, Vec2(400, 600)), (0, 0, 255)],
    [Object(1, 0, Vec2(200, 700)), (0, 0, 255)],
    [Object(10, 0, Vec2(800, 300)), (0, 0, 255)],
]


counter = 0


def customDraw():
    draw_object(boxes[counter][0], boxes[counter][1])


def customInput(key, modifiers):     • "modifiers" is not accessed
    global counter

    if key == pyglet.window.key.RIGHT:
        counter += 1
        if counter >= len(boxes):
            counter = 0


window.drawHandler = customDraw
window.keyboardInputHandler = customInput
window.run()
```

It will cycle through the boxes needed, in order for us to do our testing.

TC(n)	Test Name	Test data	Expected output	P?
10	Box with red colour	Object(1, 0, Vec2(400, 600)), (255, 0 , 0)	Red box at 400, 600	
11	Same Box but blue	Object(1, 0, Vec2(400, 600)), (0, 0, 255)	Blue box at 400, 600	
12	Blue Box but different location	Object(1, 0, Vec2(200, 700)), (0, 0, 255)	Blue box at 200, 700	
13	Blue Box but different mass	Object(10, 0, Vec2(200, 700)), (0, 0, 255)	Larger blue box at 200, 700	

As always, the evidence is in the appendix.

As shown in the evidence, the last box is far too big, meaning our constant for mass needs to be changed. I sought to use some mathematical function to yield this constant. I ended up with $f(x)=|(\log(x+10))|$, as it gives a large enough value in the beginning for the box to look meaningfully large, but also then plateaus out for large values of x, giving a good visual.



Of course, this also allows us to change the constant in a smart manner if we so wish later on.

Module 4: Velocity

In this module, I will add a method that will move an Object instance, using its velocity with respect to time.

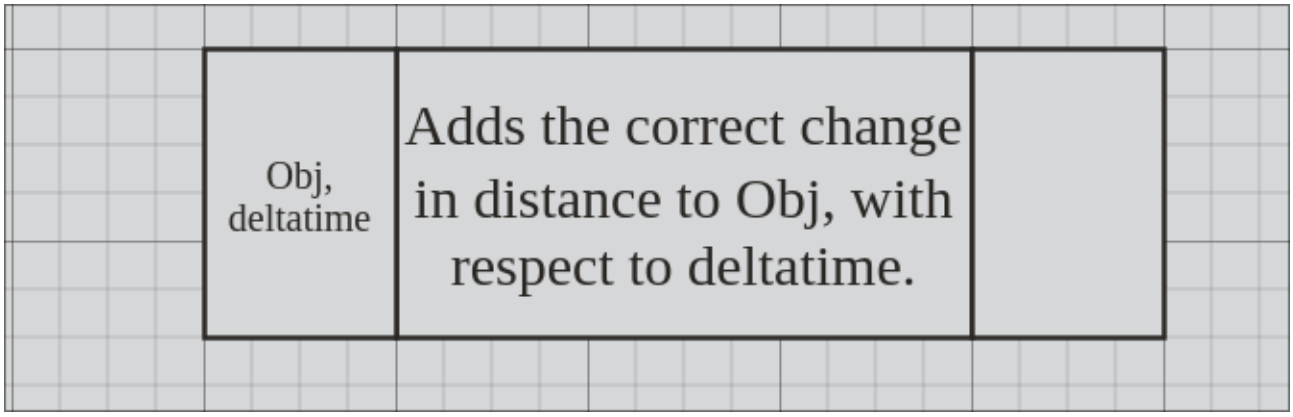
In this simple project, I will use the SUVAT system of equations to solve for position, however do note, that better, more robust numerical integration methods exist, such as Verlet or Leapfrog integration. I did not use those, since they require more complex systems set up as a prerequisite, notably, acceleration. In our elastic world, we do not have any decay of acceleration (since we assert that the forces inside our system will remain constant), therefore, acceleration will be instantaneous and constant (for a given timestep)

Module 4.1: The Maths

$$\Delta s_i = s_i - s_{i-1}$$
$$\because s = \frac{u+v}{2}\Delta t$$
$$\therefore \Delta s_i = \frac{u+u}{2}\Delta t$$
$$\Delta s_i = u \cdot \Delta t$$
$$\therefore s_i = s_{i-1} + (u \cdot \Delta t)$$

For a given timestep (Δt), we calculate the change in distance Δs_i from s_i, where i is our i^{th} timestep. Since as said before, acceleration, is instantaneous, therefore, we have no change in velocity*.

Module 4.2: Implementing the Math



Now that we have the math needed to solve this module, we can come up with a algorithm plan:

Given a Object with mass and velocity, and current position:

1. Calculate the change in distance by the equation calculated above
2. Accrete this to the current x component of the Object's velocity.

Module 4: Sidenote on delta-time

The reason we use delta-time as our time-step, is because it makes the program consistent across framerates, as our timestep is the inverse of the frames per second we run our program at. Therefore, we ensure the mathematics will remain consistent across computers, as well as that, a higher framerate (thus a smaller delta-time), will yield the most accurate value, compared to if we had done analytical integrals.

Pyglet automates this process by making the client schedule the update function to run at a desired framerate, by using this function call (from documentation):

```
pyglet.clock.schedule_interval(update, 0.1)
```

TC(n)	Test name	Test input	Expected Output
14	Positive Velocity	Dt:0.1, obj.pos.x = 10, obj.vel.x = 5	Obj.pos.x = 10.5
15	Negative Velocity	Dt: 0.2, obj.pos.x = 15, obj.vel.x = -3	Obj.pos.x = 14.4
16	Zero Velocity	Dt: 0.5, obj.pos.x = 20, obj.vel.x = 0	Obj.pos.x = 20
17	Large time Step	Dt: 2.0, obj.pos.x = 25, obj.vel.x = 4	Obj.pos.x = 33
18	Obj with zero Vel. Dt is 0	Dt: 0, obj.pos.x = 10, obj.vel.x = 0	Obj.pos.x = 10

Note that we set custom timesteps for testing purposes, however to note, in the next module, we will use pygame's own system for setting a timestep.

Module 4: Development

This can be implemented within two lines, one where we calculate the change in displacement, and one where we accrete this value to the object's position, like so:

```
10 def updateObject(dt: float, obj: Object):
       delta_x = obj.velocity * dt
       obj.position.x += delta_x
```

Module 4: Testing

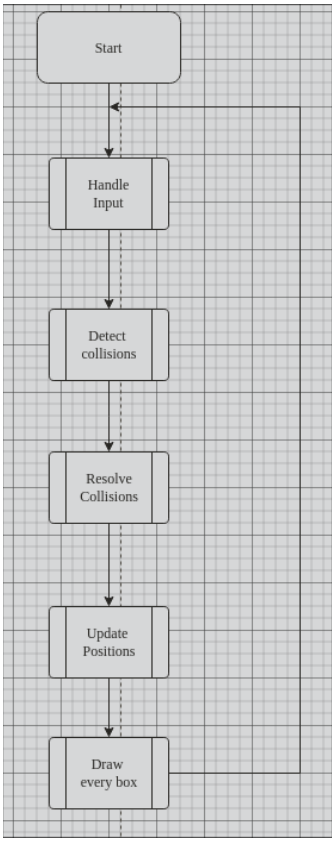
Once again, I will use unittests, in order to standardise my testing procedures for mathematical functions.

```
    def test_positive_velocity(self):
        obj = Object(5, 5, Vec2(10, 0))
        updateObject(0.1, obj)
        self.assertAlmostEqual(obj.position.x, 10.5, places=2)
```

This is test for the first test case, the rest are repeated.

TC(n)	Test name	Test input	Expected Output	P?
14	Positive Velocity	Dt:0.1, obj.pos.x = 10, obj.vel.x = 5	Obj.pos.x = 10.5	
15	Negative Velocity	Dt: 0.2, obj.pos.x = 15, obj.vel.x = -3	Obj.pos.x = 14.4	
16	Zero Velocity	Dt: 0.5, obj.pos.x = 20, obj.vel.x = 0	Obj.pos.x = 20	
17	Large time Step	Dt: 2.0, obj.pos.x = 25, obj.vel.x = 4	Obj.pos.x = 33	
18	Obj with zero Vel. Dt is 0	Dt: 0, obj.pos.x = 10, obj.vel.x = 0	Obj.pos.x = 10	

Testing Prototype 1: Moving boxes around



Rendering and having the ability to move the box is good, however we need to combine the power of the two, by rendering moving objects.

Therefore, we need to establish a main loop, that pyglet will run every frame.

We would want the following test plan:

TC(n)	Test data	Expected output
19	Run the	Objects should

program move.

Note that boxes 1, 2, 4, 5 have been done, and box 3 will be implemented next module, however for the purposes of this module we will only consider the contributions of boxes 4 and 5.

Therefore, we can write on_draw function (which pyglet considers a main loop), as follows:

```
19 def customDraw():
20     # update position
21     # draw the boxes
22
23
```

and let our window know as follows:

```
33 window.drawHandler = customDraw
```

So, we can have a global representation of the boxes by having such a list:

```
10 boxes = [
11     [Object(1, 2, Vec2(2, 0)), (255, 255, 0)],
12     [Object(1, 0, Vec2(400, 600)), (0, 0, 255)],
13 ]
14
```

where the first part of each entry describes the object, the second part describes the colour used by module 3.

Going back to our main loop, we can implement the updates of positions as follows:

```
21     for box in boxes:
22         pyglet.clock.schedule_once(updateObject, 1 / 60, box[0])
```

Note that our update is called by pyglet, and reasons were described in module 4.

Then, we can similarly, render our objects:

```
24     for box in boxes:
25         draw_object(box[0], box[1])
```

Note that we won’t render our objects by calling pyglet’s clock, as pyglet will schedule this at the same rate as our object, since we also write this code in the window’s initialisation:

```
def run(self, frameTime: float = 1 / 60)
```

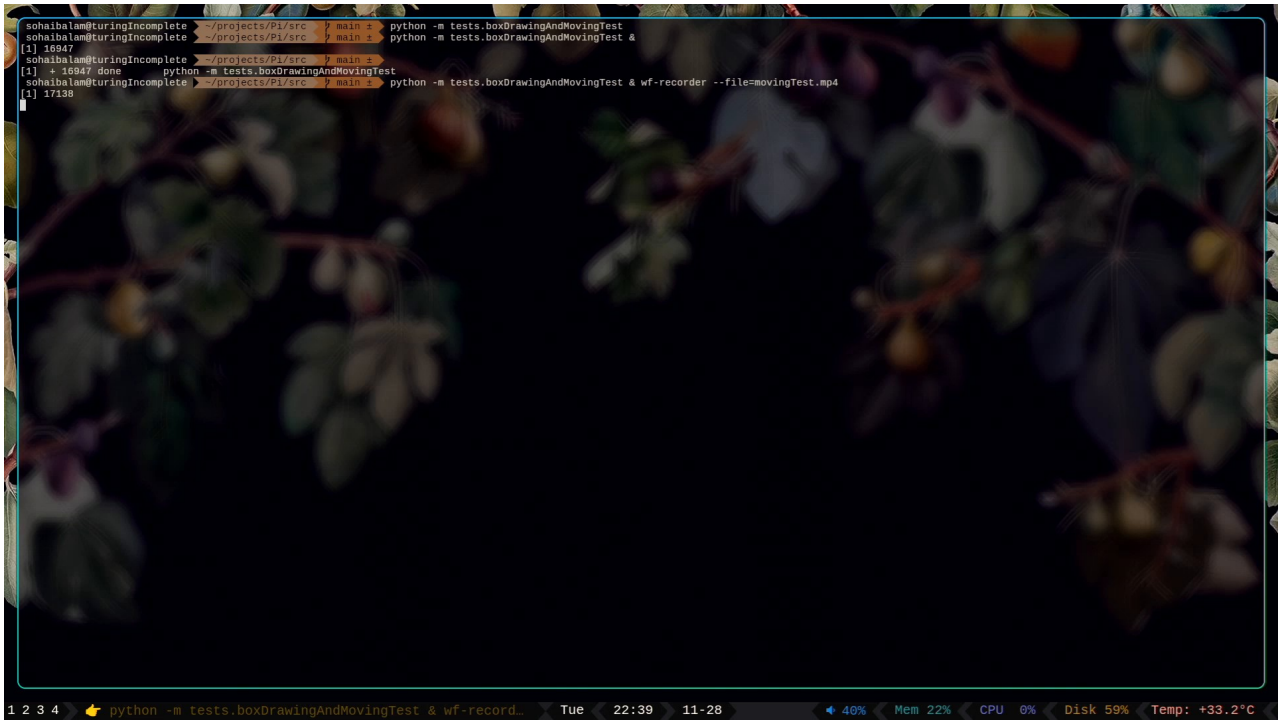
As described in our class, which feeds into:

```
    pyglet.app.run(frameTime)
```

Note that this does not necessarily make our updating positions code is redundant, as it still makes it clear to pyglet, and us, what frame time we use.

This should be able to run now…

TC19:



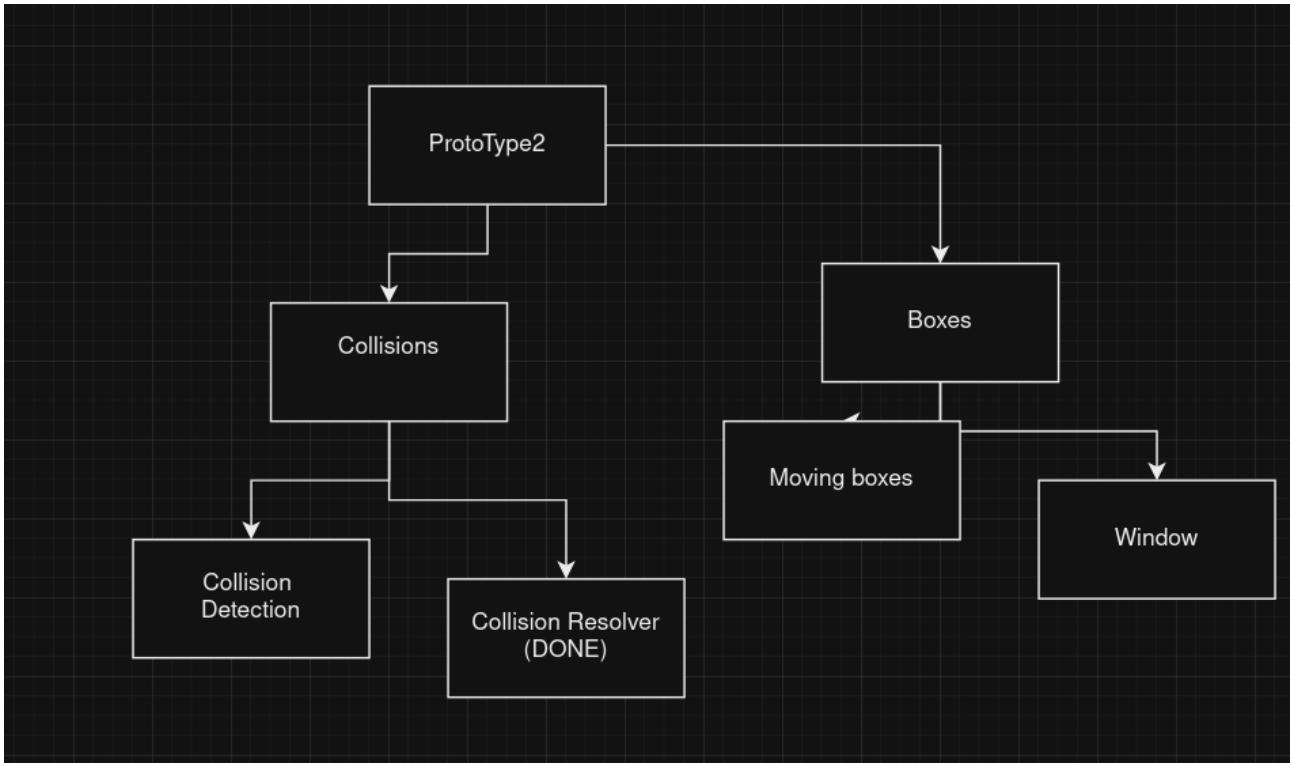
TC(n)	Test data	Expected output	P?
19	Run the program	Objects should move.	

Prototype 1 Evaluation:

All of our tests passed for this prototype, after all, we are simply rendering moving boxes. There wasn’t much deviation from the design philosophy that I had in mind. This simplicity, also allows for a “plug ‘n play” attitude to the components of main loop. Say for example, I decide SUVAT position resolution is not good enough, and change it to leapfrog integration. Since the

module is independent, I only need to change the main loop to accommodate the changes.

Next, I will work on getting collision detection working, and since we have already written the resolution code, after this module, the MVP should be ready, and then we can move on to work on superfluous items like audio, and simulation controls…

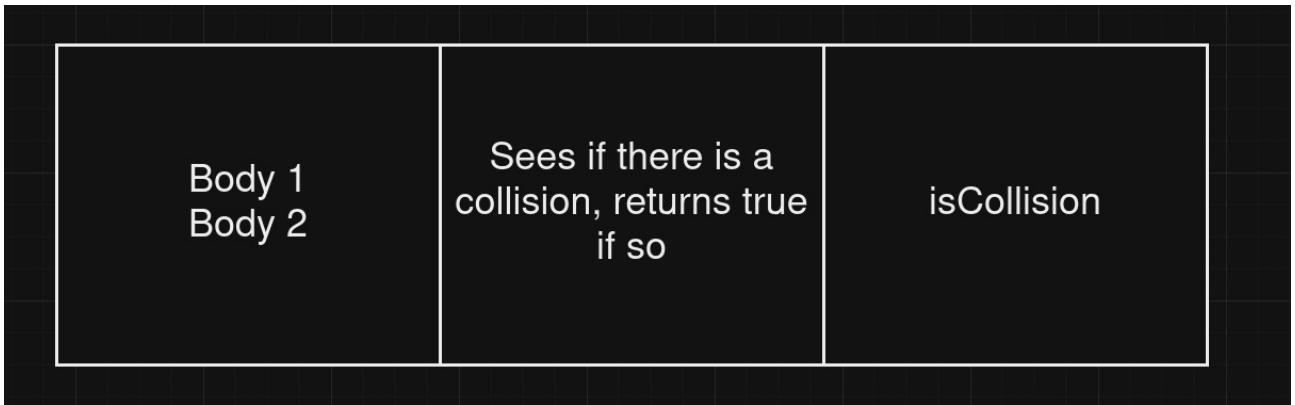


Note, that the right hand side of the tree, has already been developed.

Module 6: Detecting Collisions

Now that we have a method of drawing our boxes, we should work on detecting collisions working on rectangles.

I intend to use a function to detect collision, given two bodies:



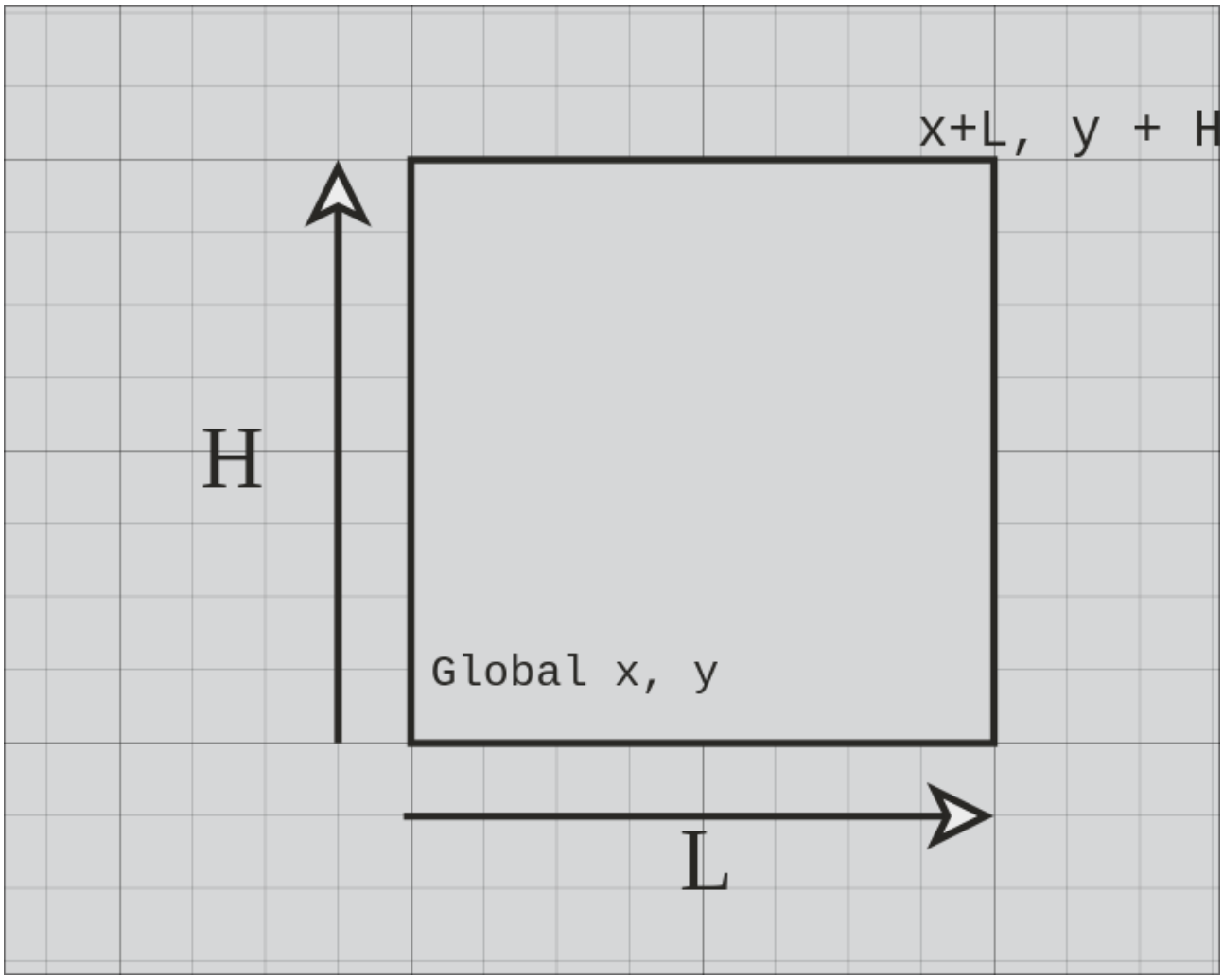
Module 6: Data table

Variable name	Variable type	Description
Body 1	Object	First object to detect collisions with
Body 2	Object	Second object to detect collisions with
Width1	Float	Length of the first body
Width2	Float	Length of the second body
IsCollision	Bool	True if collision, false otherwise.

Module 6: Test plan

TC(n)	Test name	Test input	Expected Output
20	Test no collision	Obj1((0, 0), 1); Obj2((5, 0), 1);	False
21	Test normal collison	Obj1((0, 0), 1); Obj2((4, 0), 1);	True
22	Test same position collision	Obj1((0, 0), 1); Obj2((0, 0), 1);	True

Module 6: Development



As it turns out, pyglet draws its quadrilaterals from the bottom left corner, as shown in the diagram above. Therefore, in order to do a *x-axis only* collision detection, we can craft the following algorithm plan:

1. Check if obj1.pos.x + length1 ⩾ obj2.pos.x
2. If so, return true, if not false.

Which can be implemented in this manner:

```
25 def collisionDetection(obj1: Object, obj2: Object) -> bool:
26     W1 = 10 * mass_factor(obj1.mass)
27
28     if obj2.position.x >= obj1.position.x + W1:
29         return True
30
31     return False
```

Note that we don’t need to calculate Width2, as that would unnecessary.

However there is a problem with the above logic. What if x1 is greater than x2? Won’t that ensure (through the math), that there will be no collision (according to our logic)?

A very naive solution could be to simply raise an exception whenever that happens:

```
26      if obj1.position.x >= obj2.position.x:
27          raise ValueError("obj1 must be to the left of obj2")
```

However, I am not very happy with this solution, as I would like to make a more “general” solution. However to abide by the tenets of prototype testing, I will come back and make a more robust solution as an extra. This is because, due to the nature of our system obj1 **will never** be to the right of obj2, but simply saying that doesn’t make the solution any better.

Module 6: Testing

Tests are again conducted using unittests

TC(n)	Test name	Test input	Expected Output	P?
20	Test no collision	Obj1((0, 0), 1); Obj2((5, 0), 1);	False	
21	Test normal collision	Obj1((0, 0), 1); Obj2((4, 0), 1);	True	
22	Test same position collision	Obj1((0, 0), 1); Obj2((0, 0), 1);	True	

Module 6: Evaluation

Personally, I feel that this module was implemented in a manner of what British people would call a “bodge”, meaning that it is only a temporary solution to a ever-permanent problem. However, I do see myself thinking that this module may not need the amount of over engineering as discussed in the development of this module.

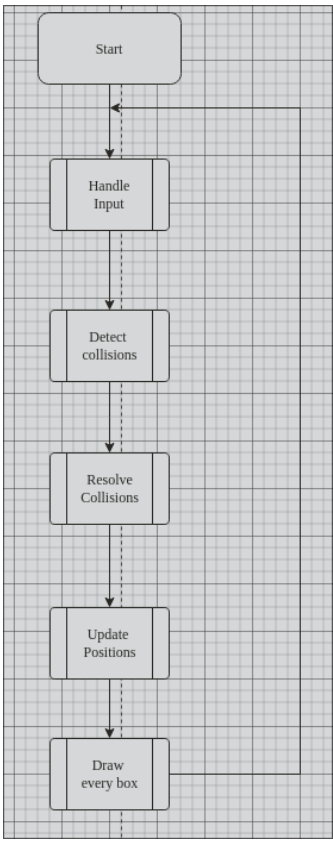
However, this exception could also be very useful in testing the code later on, as it could be a good pointer for when the program's physics engine has been overwhelmed. Regardless, I will keep in mind to come back to it afterwards.

Next up, I will start testing the next chapter of modules in the moniker of Prototype 2, which will combine all of the physics into one neat package, before we move on to further "fluff".

Prototype 2: Electric Boogaloo

In this module I will conduct the second (and by far the most important) test of all, whether the collisions work with moving boxes.

I will implement this flowchart from before, though unlike before, all procedures will run in harmony.



Var name	Var type	Description
Pygletoverride	PygletOverride(module 1)	Main way of communicating with pyglet window ctx
Obj1	Object	First object to test collisions with
Obj2	Object	Second (larger) object to constructively interfere with obj1

Note that all of out test input is the same. Just running the program as described in the flowchart.

TC(n)	Test name	Expected output
23	Boxes movement and collision	Boxes move and collide
24	Collision being counted	Each collision is accounted for.

Prototype 2: Development

We will begin by creating a pyglet context:

```
pygletHandle = PygletOverride("Prototype 2", Vec2(800, 600))
```

Next up, we will add the boxes:

```
obj1 = Object(1, 0, Vec2(50, 200))
obj2 = Object(100, -50, Vec2(750, 200))
```

Next up, we will add a collision counter, as expected by the tests:

```
collisionCount = 0


def customDraw():
    global collisionCount
```

Next, we will check for collisions (both from the wall, and from each object):

```
    if collisionDetection(obj1, obj2):
        obj1.velocity, obj2.velocity = elastic_collision_solver(obj1, obj2)
        collisionCount += 1

    if obj1.position.x < 0:
        obj1.velocity *= -1
        collisionCount += 1
```

Next we will update the objects, note that I ditched the pyglet scheduler, as it proved to be awkward when dealing with smaller timesteps:

```
    updateObject(timestep, obj1)
    updateObject(timestep, obj2)
```

Timestep is defined to be 1 / 144.

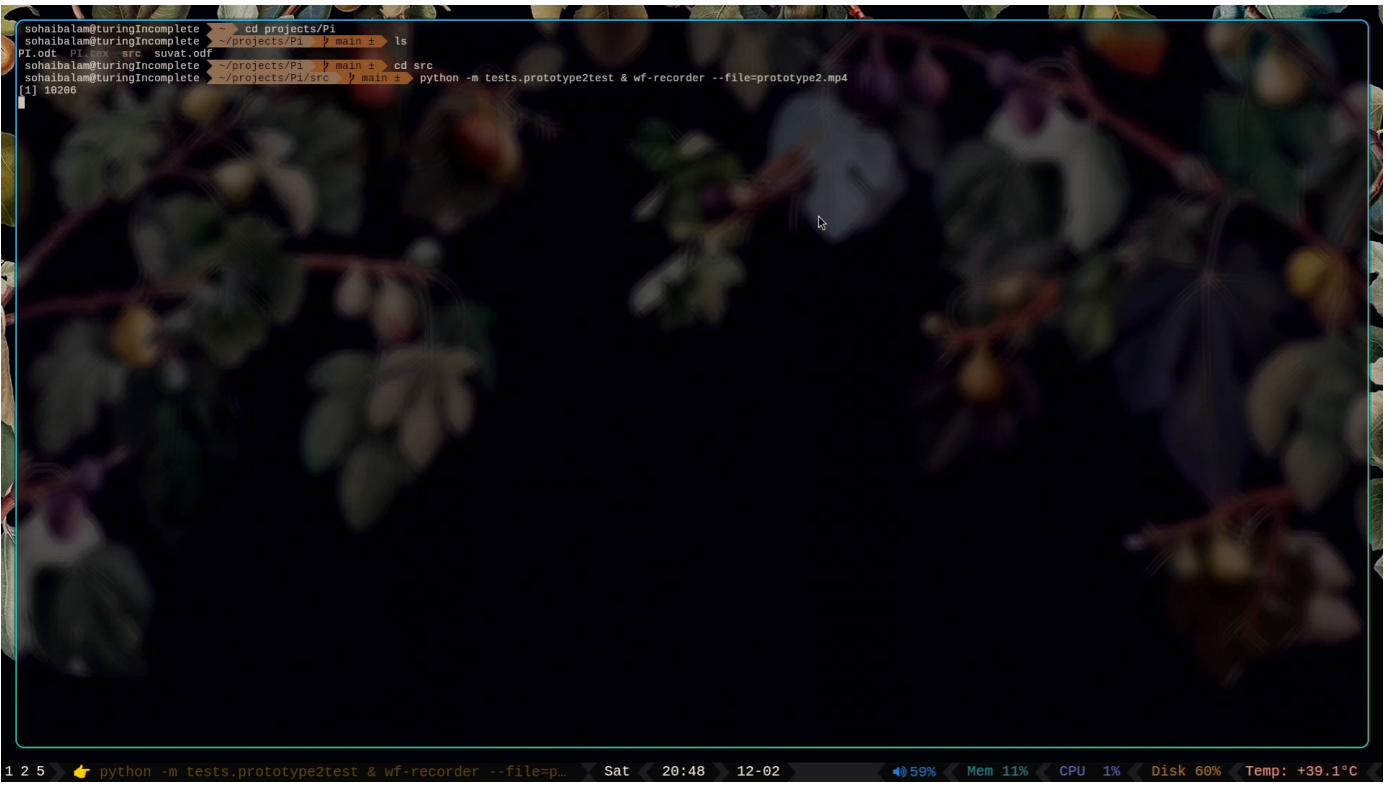
Next we draw the objects:

```
    draw_object(obj1, (255, 0, 0))
    draw_object(obj2, (0, 0, 255))
```

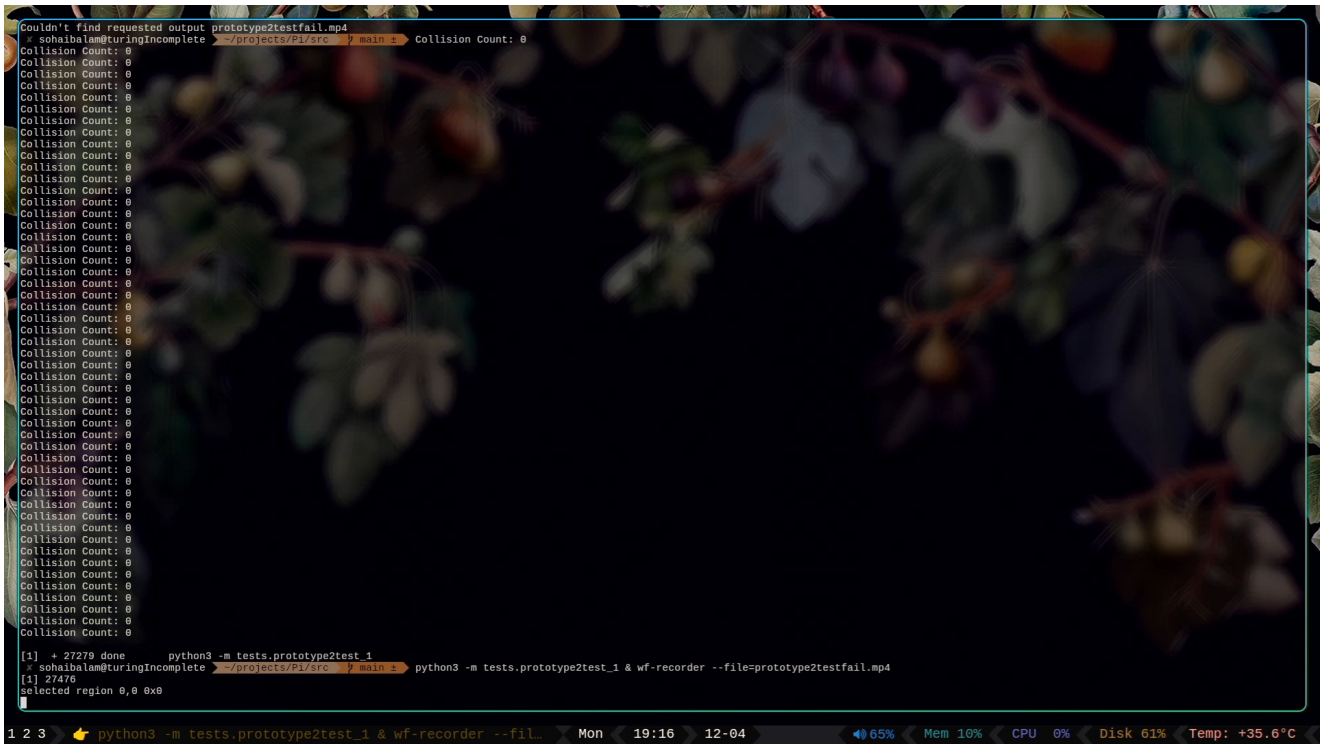
And let our pygletOverride class know what our draw function is, and let it run:

```
pygletHandle.drawHandler = customDraw
pygletHandle.run()
```

TC 23-24:



As with the nature of the problem, we need to be able to give large values of mass for obj2, in order to get a large enough approximation of PI, however when run, the program yields:



This is because, we do not run the simulation enough, for the physics engine to get closer to the actual result. In order to fix this, we implement sub-stepping. Which essentially breaks down the current simulation steps into further simulation steps, which

causes more accurate physical calculation. However, as the sub-steps are increased, the computation time also increases linearly, therefore it should be imperative for us to find the sweet spot.

Therefore I will change the tests to as follows:

TC(n)	Test name	Expected output
23	Boxes movement and collision	Boxes move and collide
24	Collision being counted	Each collision is accounted for.
25	Large values of mass	Simulation has no hiccups.

To implement the sub-stepping algorithm, we firstly change the initial velocity, to stop it from exploding under sub-stepping criteria:

```
digits = 3
timestep = 10 ** (digits - 1)
obj1 = Object(1, 0, Vec2(50, 200))
obj2 = Object(100 ** (digits - 1), -1 / timestep, Vec2(700, 200))
```

```
for i in range(timestep):
```

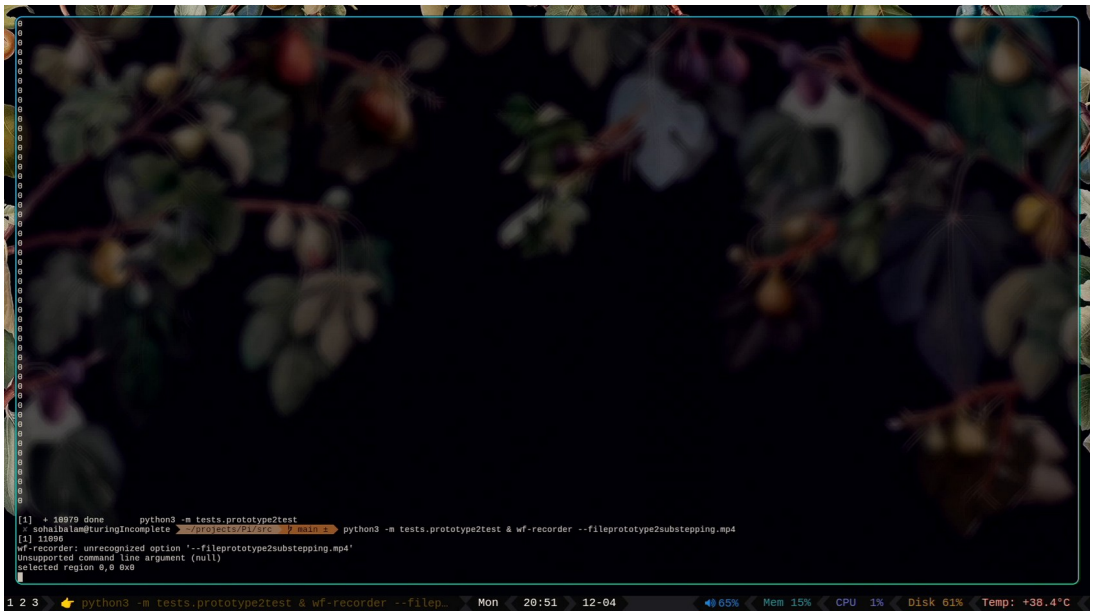
Then we have a for loop running for however many sub-steps we need.

The last change we need get rid of the timestep variable in updating the position. This is because, mathematically speaking, doing a numerical sum with a tiny delta-time, is the same as a integral in the continuous world, whereby we can ignore the contribution of delta-time at all. Therefore, I set it to 1, negating any effect.

```
    # update positions
    updateObject(1, obj1)
    updateObject(1, obj2)
```

The rest of the algorithm is the same. Do note that we draw the boxes at the end of the sub-stepping algorithm.

TC 23-25:



TC(n)	Test name	Expected output	P?
23	Boxes movement and collision	Boxes move and collide	
24	Collision being counted	Each collision is accounted for.	
25	Large values of mass	Simulation has no hiccups.	

Prototype 2: Evaluation

I am quite pleased with the results of this prototype, especially now that we can handle larger number of PI. However, a big concern I had with this prototype, was firstly, the jittering of the left-most box (at the end, in the TC 23-25). My guess it has to do

something with v-sync, however I will confirm my suspicions later on.

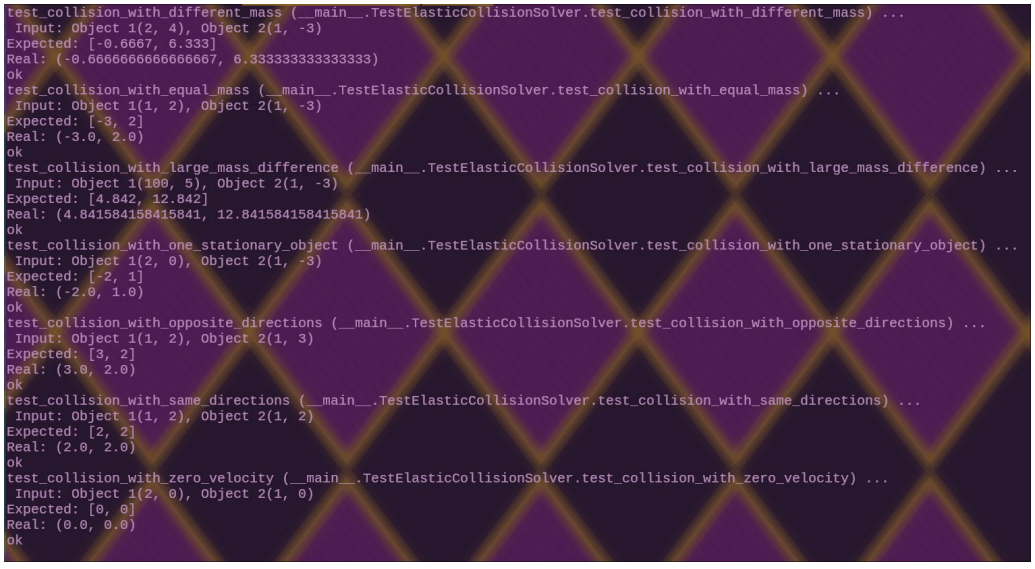
Next up, I will start adding the “fluff” modules, these modules provide no benefit to the end user, however still are useful indeed. The biggest contender is obviously a collision counter, which has to be the most important part of the problem.

After that, I might optimize the code to work for even larger digits (and subsequently the number of sub-steps per frame), by multi-processing, and implement ridiculous mathematical optimisations.

Appendix

Module 2

TC 3-9:



Module 3

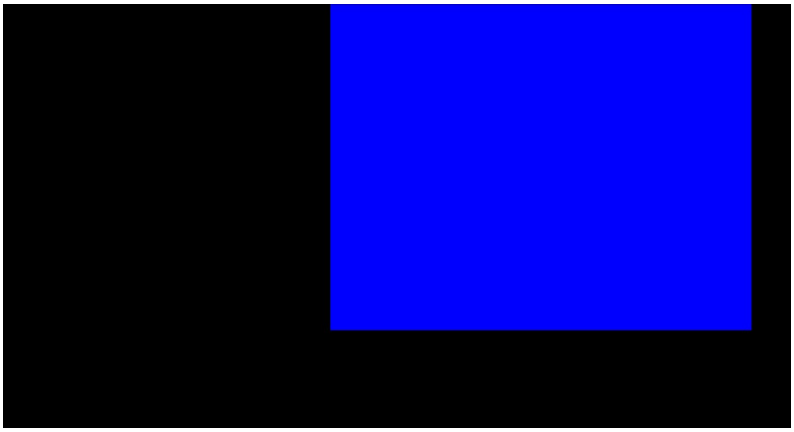
Note that all of these sizes are relative :)

TC10:

TC11:

TC12:

TC13:



Module 4:

TC14 - 18:

```
test_large_time_step (__main__.TestUpdateObject.test_large_time_step) ... ok
test_negative_velocity (__main__.TestUpdateObject.test_negative_velocity) ... ok
test_positive_velocity (__main__.TestUpdateObject.test_positive_velocity) ... ok
test_zero_velocity (__main__.TestUpdateObject.test_zero_velocity) ... ok
test_zero_velocity_zero_time_step (__main__.TestUpdateObject.test_zero_velocity_zero_time_step) ... ok
```