Program Design Methods and Intro to Programming Python Final Project: Macet



**Student Information:**

Jason Christian Hailianto / 2301891714

**Class Information:**

Class: L1AC

Lecturer’s name: Ida Bagus Kerthyayana / D5757

**Binus University International**

**School of Computer Science**

**2020**

**Chapter I: Introduction**

In the first semester at Binus International, the Python programming language is taught to the students of Computer Science. After learning said language for approximately 2 months, and understanding the basics of programming and external libraries, students must complete their course by doing a final project with contents of their own choosing, also with specified requirements. Students are given creative freedom on what they want to make and can choose to use any of the python external libraries on the internet. After developing the program, students must demo and present their program to the lecturer.

After the announcement of the final project, I came up with the idea to make a physics simulator in Python complete with weights, gravity, and custom object creation. But, several days after that, I came up with a better idea in my opinion, that will test and improve my coding skills. I came up with an idea to make a traffic simulator in python; complete with road infrastructure, traffic lights and many cars. I thought this is a good idea because, I will be making most of the systems from scratch and try not to use any external libraries for the complex systems. This is also my first time taking on a real project in Python, so I am looking forward to doing this endeavor.

This project officially began in 30th of December 2019. Visual Studio Code was chosen as my IDE for this project. Also, as this project’s nature is open source, all progress will be uploaded to my GitHub account, <https://github.com/nakamarusun/SimMacet> .

**Chapter II: Project Specification**

**Project purpose:**

To help visualize and simulate real traffic simulations in a program and can be used to help solve real problems and how to solve them efficiently. To learn python programming and solve computing problems efficiently.

**Project audience:**

People who would like to simulate traffic simulations, and programmers who would like to learn and understand algorithms.

**Project aim:**

To create a functional traffic simulator. With support of a large amount of car objects without lagging, roads that can connect and intersect with others, and with a stop light system. The simulation needs to have a large area, that can be navigated onto by the user, and needs to sustain a lot of car objects and simulation with minimal computing power.

**Project requirements:**

* Functioning map with coordinate systems,
* Road objects that can connect to each other and intersect,
* Car objects that snaps to road and have real physics variables (Acceleration, speed),
* Smart car collision system that can detect and process collisions with other cars,
* Working traffic light system, with cars that can stop at command.

**Chapter III: Solution Design**

1. **Overview**

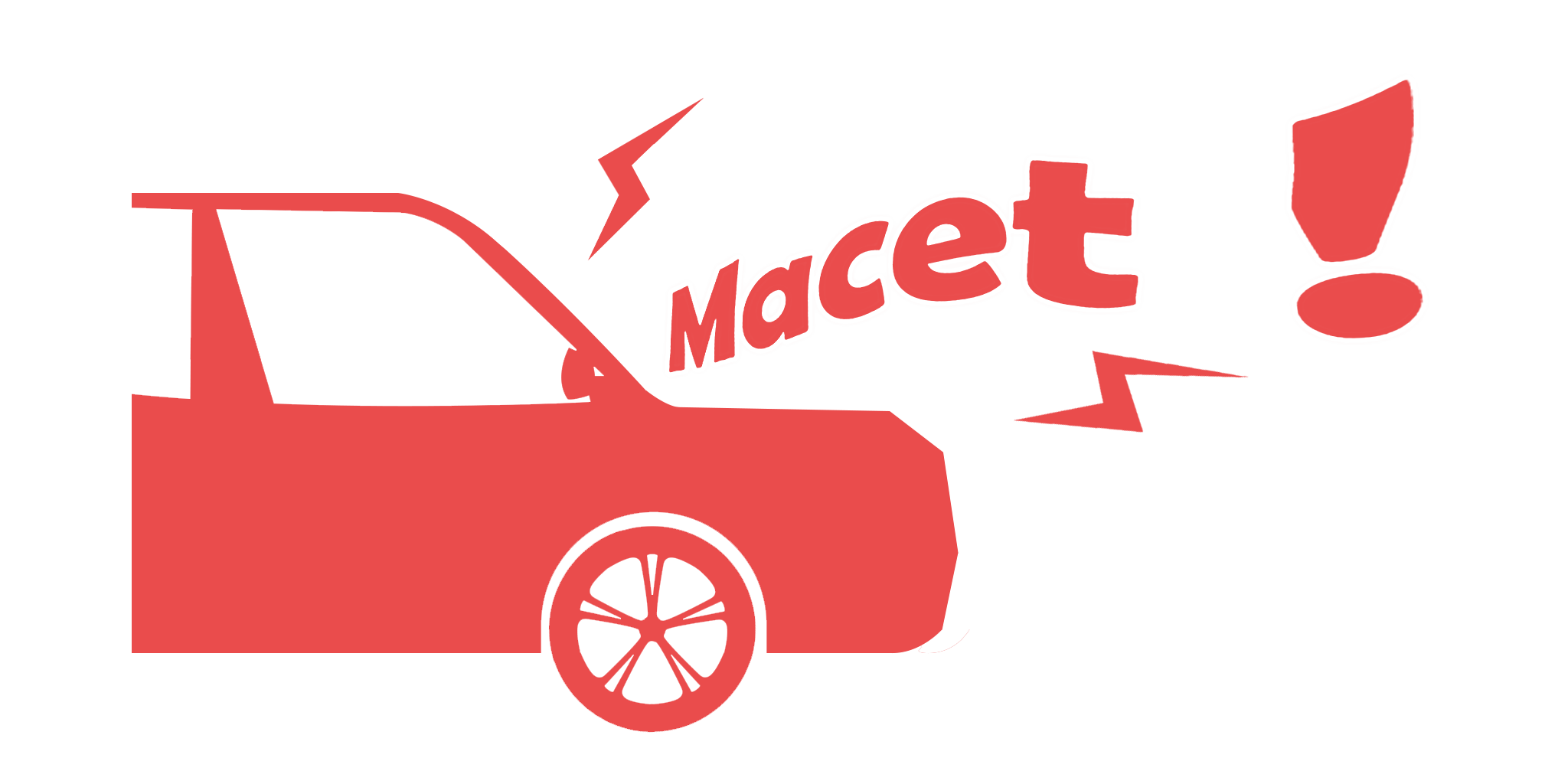


Image 1. “Macet !” logo

“Macet !” is a traffic simulator, designed to help solve and visualize real traffic simulation in a Python application. Also to help other people study algorithms from the application, which is open source.

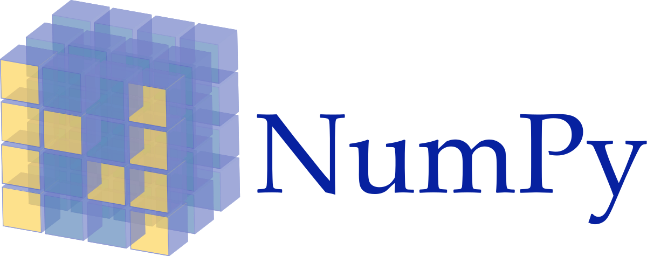
 

Image 2, 3. NumPy, and Pygame logo.

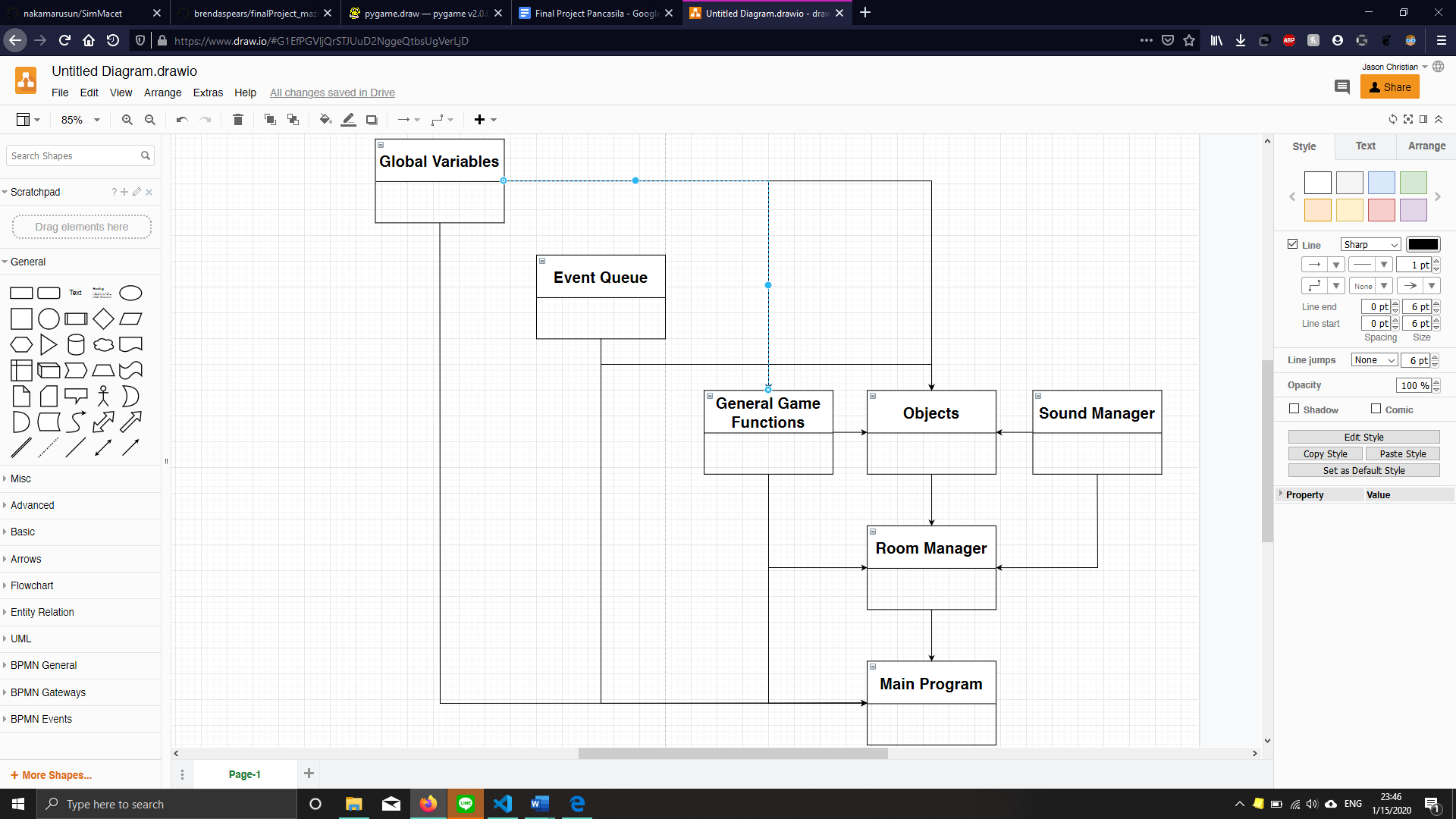
(<http://www.pygame.org/docs/logos.html> , <https://github.com/numpy/numpy/blob/master/branding/icons/numpylogo.svg>)

“Macet !” uses Python 3.8.0 with the following as external libraries:

* NumPy 1.18.1
* Pygame 1.9.6
* Shapely 1.6.4.Post2

The Pygame library is used to visualize and draw things onto the application screen, and as a basic foundation of the simulator. Meanwhile, NumPy and Shapely is used for trigonometry, vector, and numerous other calculations needed. All these libraries are used to make the program working properly.

1. **Program file dependencies**



Explanation:

* **Global variables** are where all the variables needed to be accessed everywhere from the program are stored. This is especially versatile because of that very reason and is essential. For example, the main screen variable, program resolution, current room, fonts, mouse positions are stored.
* **Event queue** is the place where all of the user input events are gathered and processed. For example, mouse movements, keyboard clicks, etc. All of the events processed, are sent to global variables.
* **General game functions** is the place where commonly needed functions, but is not native in python nor Pygame is stored. This is versatile too as it can be accessed from many places at once.
* **Objects** is the place where game objects are defined and stored. This is one of the most important places, as it controls the individual objects of the game. Each objects here, has a built-in function called update(), that will be run every frame of the game.
* **Sound manager** is the place where all sounds are processed and played. Currently unused.
* **Room manager** is a class, where each room can contain a number of objects. The program can only run one room at once, and it functions as a place, where objects interact with only each other in that room.
* **Main program** is the root of all the files. Everything is processed and executed here. First, all the needed files, libraries, and variables are imported and initialized here, based on the settings. Then, comes the main loop. The main loop updates the current room and that room updates every object individually in that room, making everything run properly like a symphony.

1. **What the main loop does**

Basically, every iteration of the main loop runs like this:

    # Get time at the start of the frame

    startTime = time.time()

    # Clear screen buffer

    GMvar.mainScreenBuffer.fill( (255, 255, 255) )

    # Load all events to GMque.currentEvents list

    GMque.loadEvents()

    # Update and get mouse pos

    GMvar.latestMouse = pygame.mouse.get\_pos()

    GMvar.mouseDelta = pygame.mouse.get\_rel()

    GMvar.mouseState = pygame.mouse.get\_pressed()

    GMvar.update()

    # Update each object in current room

    GMvar.curRoom.updateRoom()

    # Draw things that is top most

    GMfun.drawTopMost()

    # FPS Calculator

    FPS = GMfun.displayFps(startTime)

    # Copyright

    GMvar.mainScreenBuffer.blit(GMvar.credit, (GMvar.resolution[0] - 110, 2))

    # Update display

    pygame.display.flip()

    # Delta Timing

    GMvar.deltaTime = GMfun.deltaTiming(startTime)

1. The time of the start of the frame is recorded into a variable,
2. mainScreenBuffer (application window) is filled with a white color (clear)
3. Update all of the events in **Event queue**
4. Update all of the variables in **Global variables** from **Event queue**
5. Update every object in the current room
6. Draw everything that needs to be drawn at the very top to the mainScreenBuffer
7. Calculate current FPS based on the time from the start of the main loop
8. Update display
9. And finally, update delta time with from the start of the main loop (delta time is defined as how long it took to finish a frame’s processing)

The longest part the main loop takes is updating every object in the current room (5). This is because many objects mean a lot of code to be run and of course, this will take some time to take.

The FPS (Frame per Second) is used as a guide, to make sure that the program is not hogging up too much computing power and is still efficient. The frame per second is defined as ( 1 / ( endFrameTime - startFrameTime ) ).

This is still the very basic principle of how the program handles events and updates every object in the game. All the real calculations and processing are done in the individual objects. Next, we will take a look at some principles.

1. **mainScreenBuffer**

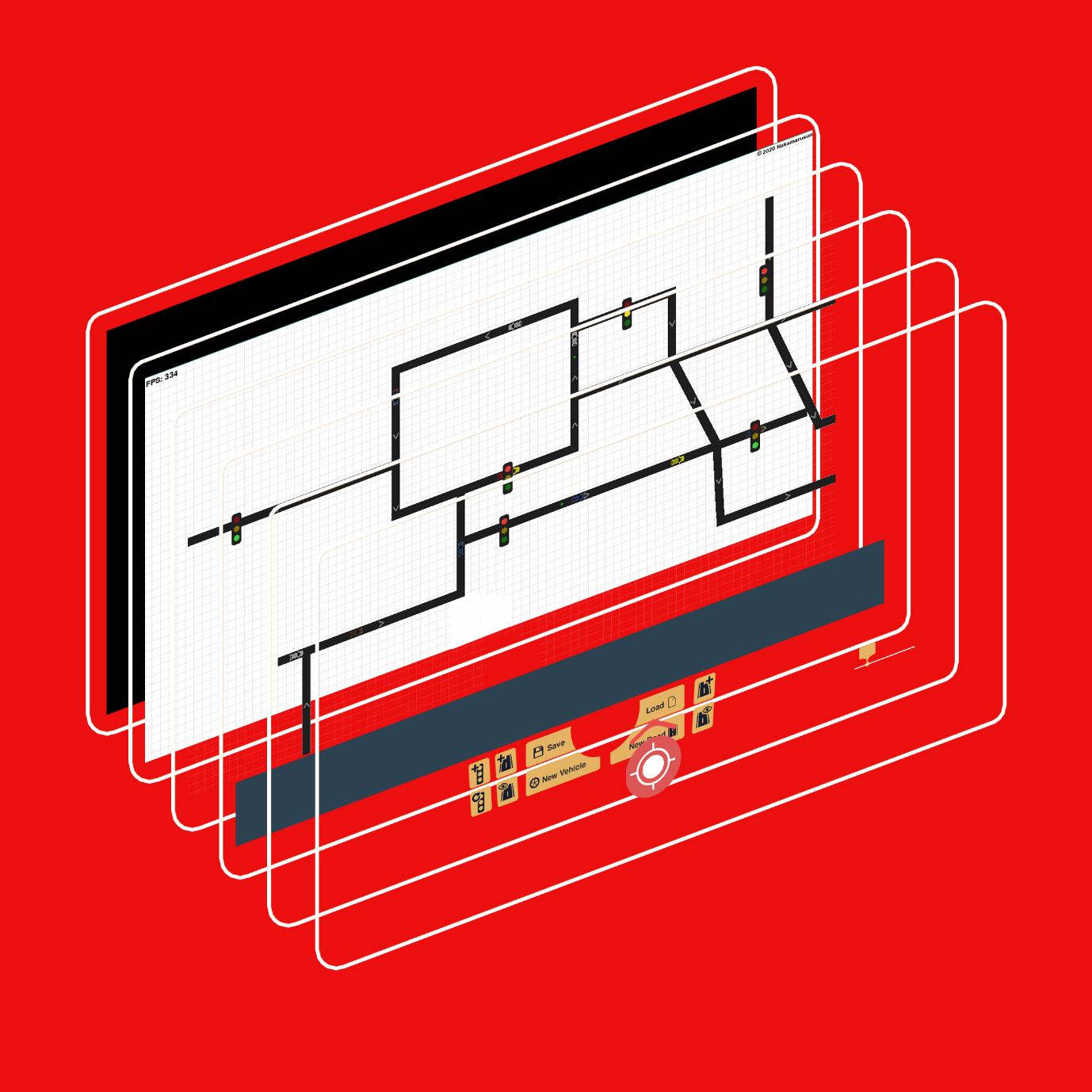
****

Image 4. Layered application view

In a room, each class’ order of update is important. As it governs what is the order of the drawn image. The first object to be drawn, is drawn at the very bottom of the screen, while the last object to be drawn is drawn topmost. Therefore, care must be put when putting object update order.

As seen in the graphics, the canvas’ grid is drawn first, then the roads, cars. Then stop lights. After all the Canvas elements have been drawn, the GUI is drawn next. With the order of GUI background, and then buttons. Note that all the GUI elements is all combined in one surface, so that management will be easier.

1. **Coordinates system**

Every visible object in the program has a “coords” variable. The coords variable is a positioning system which allows the object to be drawn in a correct position in the program, or to check position with others. Every object has the same coords format:

Self.coords: list = [x, y]

Note that the coords is purely the coordinates of the object, and not affected by the map camera position or anything else.

1. **Map camera systems**

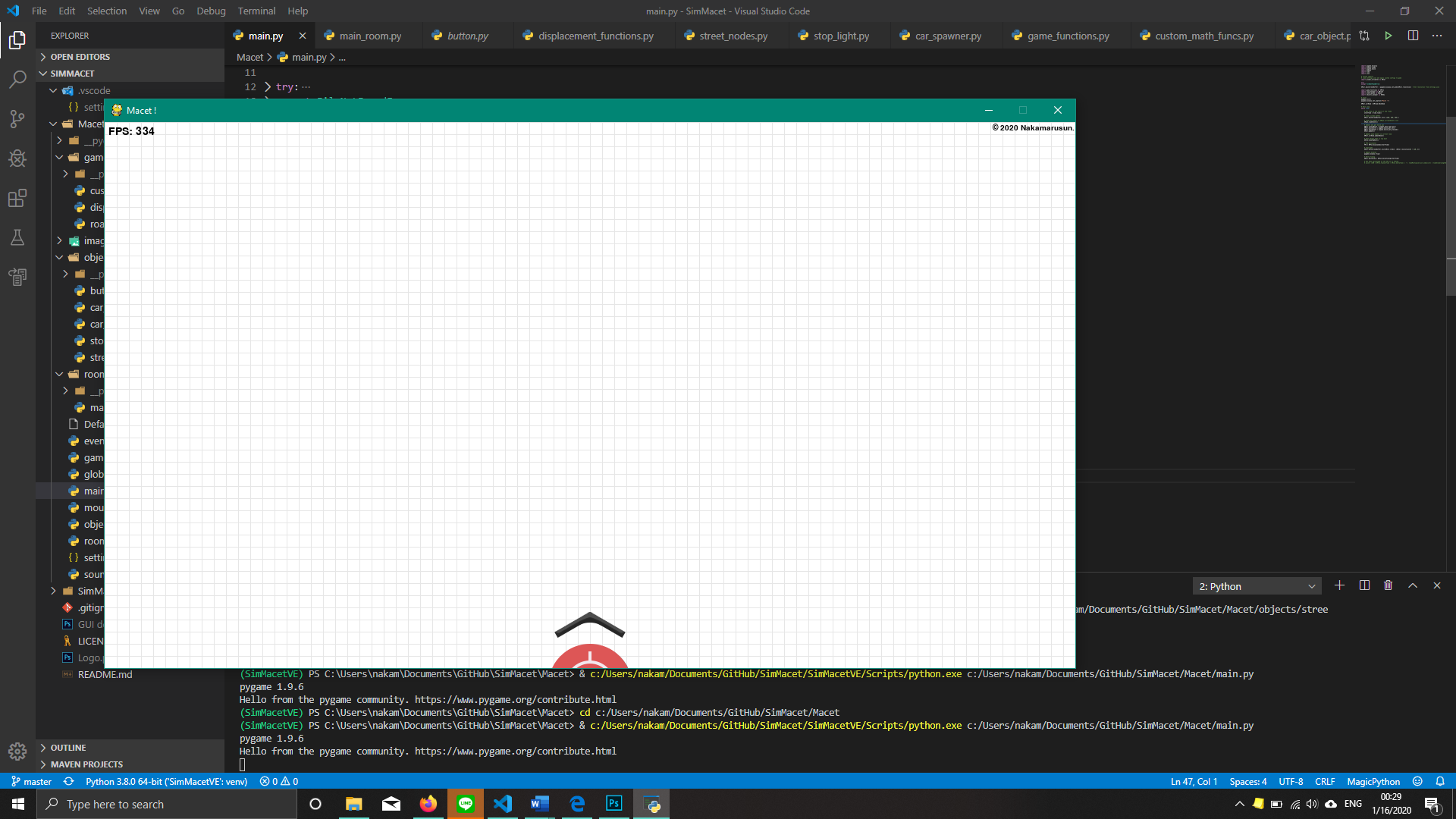


Image 5. Main camera surface

The camera system in the program allows the user to right click and drag the map and navigate to other places that are not seen currently in the screen. The principle is quite basic. Here is how it works:

CameraSurface

+ cameraCoords: list = [0, 0]

+ objectsQueue: object = []

+ drawToMainSurface()

+ update()

This is the simplest class diagram of the camera surface. Explanation for each variable:

* cameraCoords = the current coordinates of the map camera
* objectsQueue = every object that needs to be drawn in the camera surface

Functions:

* drawToMainSurface() = for every item in objectsQueue, draw the item onto main screen, with the item’s coordinates subtracted by cameraCoords.
* update() = when mouse is right clicked, add the delta mouse coordinates to cameraCoords

\*delta mouse coordinates is defined as current mouse coordinates subtracted by previous’ frame mouse coordinates. So basically, change in mouse coordinates per frame of the game.

Therefore, when right click mouse is held and dragged, cameraCoords will change based on the coordinates of the mouse, and the new objects are drawn as the cameraCoords changes.

1. **Buttons and toggle buttons**

Button

+ idleState: pygame.Surface = #Idle button

+ clickedState: pygame.Surface = #Clicked button

+ currentImage: pygame.Surface = idleState

+ coords: list = [#coords]

+ rect: list = [#width, #height]

+ surface = #surfaceToBeDrawn

+ checkState() -> bool

+ update()

Every button object have two images, Normal state, and clicked state. The update() function checks left mouse click and mouse coordinates from **global variables** and processed it based on the coordinate of the button and added by the width and height. If update detects a mouse click on the right coordinates, it will change the current image into its’ clickedState else, it will be normalState. Then, the function will draw the currentImage to the surface it is assigned to. checkState() function checks whether currentImage is clickedState or idleState. If currentImage is clickedState, then it will return True, vice versa.

ToggleButton: Button

+ clicked = False

+ newlyClicked = False

+ conflictButtons = []

+ addConflictButtons(buttons: buttons)

+ update()

Toggle buttons behave a little bit differently than normal button, in the sense that it stays down when it is clicked, and you have to click it again to make it back to normal state. The update function is updated so that a mouse click will change the state, but not return it to the original. A new feature in this is conflict buttons. This is useful to make sure the user can’t physically activate two buttons at the same time, and making your code explode.

1. **Bottom GUI**

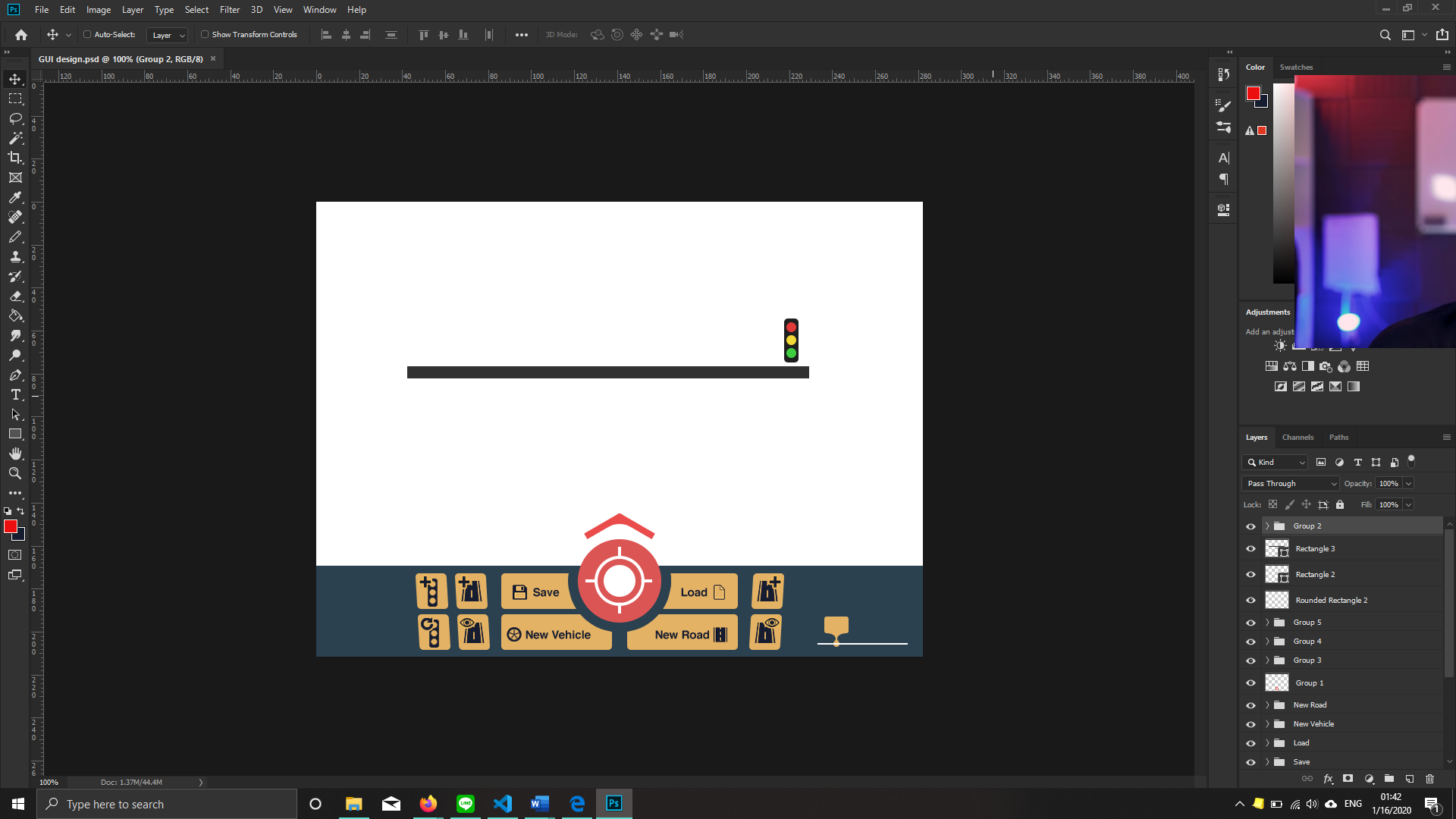


Image 6. BottomGUI

The bottom GUI has its own pygame.surface object. Every button and GUI elements is drawn to its own surface object, before drawing it to the mainScreenBuffer. By doing this, It is easier to move the GUI object as one to hide and make visible the Bottom GUI.

The bottom GUI is dynamic, in the sense that it adapts to the resolution set for the application in settings.json. bottomGUI element positioning can be helped by accessing the application’s resolution variable from **Global variables.** All the button objects are created with surface assigned with the bottom GUI surface.

The GUI arrow can be rotated by using the pygame.transform.rotate() function.

To make GUI positioning easier, the width and height of pygame.surface objects can be accessed by using pygame.surface.get\_rect() function.

The center red button is to move the camera back to [0, 0]. This can be achieved by using the button’s checkState function, then accessing CameraSurface’s cameraCoords and changing it to [0, 0].

1. **StreetNodes**

StreetNodes

+ Id: str = random string with length of 8

+ coords: list = #coords

+ backNodes: [StreetNodes] = #nodes behind

+ connectedNodes: {StreetNodes: [#vector, #angle, #length, #width, #color]}

+ connectTo(newNode: StreetNodes)

+ changeColorWidth(node: StreetNodes, color: tuple, width: int)

+ drawSelf(surface)

Now let’s get into the action.

StreetNodes are objects that is basically a point that connects to another point. This connection, when drawn a line between them with a certain width, creates the road as we know it. A standard StreetNodes object has a coordinate, a list of object which it connects back to (backNodes), and most importantly, connectedNodes, which is a dictionary which key is another StreetNodes object self is connected to, and the value is the mathematical properties, and colors.

The function connectTo(), connects self to the desired StreetNodes object by creating a new key in the connectedNodes variable and defining all the values.

changeColorWidth() is to change color or width of the node to every connectedNode.

The drawSelf(Surface) function is to draw a thick line(road) from self.coords to every key.coords in connectedNodes to the designated surface(CameraSurface), making it visible to the user.

1. **Car**

Car

+ Id: str = random string with length of 8

+ originalImage: pygame.surface = # Load from file

+ nodeAnchor: StreetNodes = #Origin StreetNode

+nodeDestination: StreetNodes = #Destination

+ maxSpeed: int = # maximum speed

+ acceleration: int = #accel

+ scalarSpeed: int = 0

+brakeDeacceleration: int= 200

+oldScalarSpeed: int = 0

+ carInFront: bool = False

+ direction: int = # Current direction

+ image: pygame.surface = # rotated originalImage using pygame.transform.rotate()

+ gridPos: tuple = #current collision grid position

+ update()

The car is an especially special object because, it has a very particular collision systems to detect other cars or StopLights in front of it.

Outside of the car \_\_init\_\_, define some variables which contains the specifications of the car’s collision triangle.

fov = 60 # Degrees

triangleHeight = 87 # Pixels, This is also the start-brake distance Original = 87Px = 16m. 300Px = 56m

triangleBase = math.tan( fov / 2 \* math.pi/180 ) \* triangleHeight \* 2

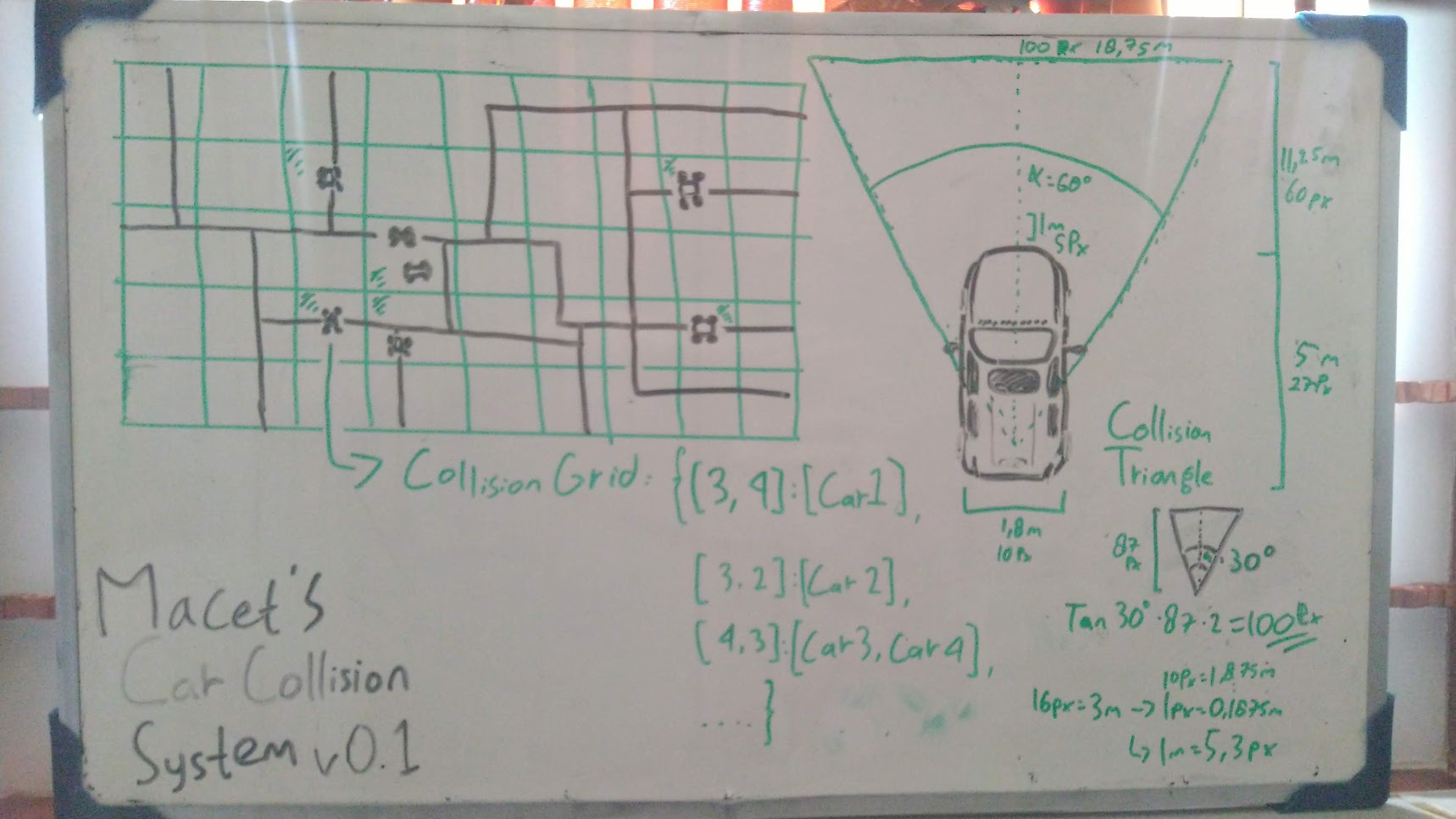


Image 7. Collision detection schematics

The collision system is as follows:

* The map is divided into grids, with relatively big cell sizes (e.g. 160x160).
* Car’s current coordinates is divided by grid’s cell size, to get what is the position of the car in the grid. Put the number into gridPos tuple.
* A dictionary of key gridPos and value list of cars in that same gridPos is created

This system is so that a car does not check for collision with other car that’s too far away and is impossible to have a collision with. Therefore, car only check with its’ gridPos, and the neighboring gridPos for collision.

After putting every cars in dictionary key gridPos and its’ neighbors into a list, for every car in that list, check for collision triangle with the car.

The collision triangle of the car is made from the variables defined outside of the \_\_init\_\_ using math. 3 coordinates of the triangle should be the output.

Check the triangle made from the 3 coordinates collision with the car’s coords in the list

Well, how do you check collision with a triangle ? Like this:

Suppose you know the 3 coordinates of the triangle, and you have a coordinate of a point you want to check for collision. Let’s visualize it like this:

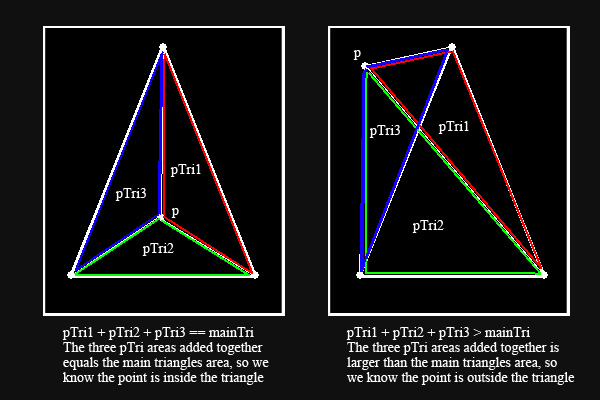


Image 8. Triangle, and it’s area

(<https://www.braynzarsoft.net/viewtutorial/q16390-24-triangle-to-triangle-collision-detection>)

* Suppose that “p” is the point you want to check.
* Calculate the area of the triangle using 3 of the coordinates.
* Calculate the area of pTri1, pTri2, pTri3
* If pTri1 + pTri2 + pTri3 == area of main Triangle then, the point is inside the triangle, making the collision true.
* But if pTri… + pTri3 > area of main Triangle, then it is outside of the triangle.

If a collision is detected, then self.carInFront = True, and self-car should be braking using the built-in variables.

The update() function of the car should be checking for next road to traverse, if the current road is already done, and delete itself if there is no next road. After that, it should check for collisions like the above, and adjust the speed based on the acceleration and collisions.

1. **StopLight**

StopLight

+ Id: str = random string with length of 8

+ Image: pygame.Surface = #Stoplight

+ coords: list = coords

+ nodeAnchor: StreetNodes = nodeAnchor

+ surface: pygame.Surface = surface

+ greenDuration: int = greenDuration \* 1000

+ redDuration: int = redDuration \* 1000

+ go: bool = True

+ timer: Timer = Timer(greenDuration \* 1000)

+ update()

To make StopLights work, it needs to be put inse of the collisionGrid dictionary of the car. Therefore, self StopLight’s gridPos has to be calculated by dividing self coords with the grid’s cell size. Then, StopLights have to append itself to the dictionary and define itself as an collision object for other cars.

Using the timer class, green light and red light duration can be timed and by assigning go = True if green light and False if red light, a functioning StopLight can be made properly.

1. **Road creation**

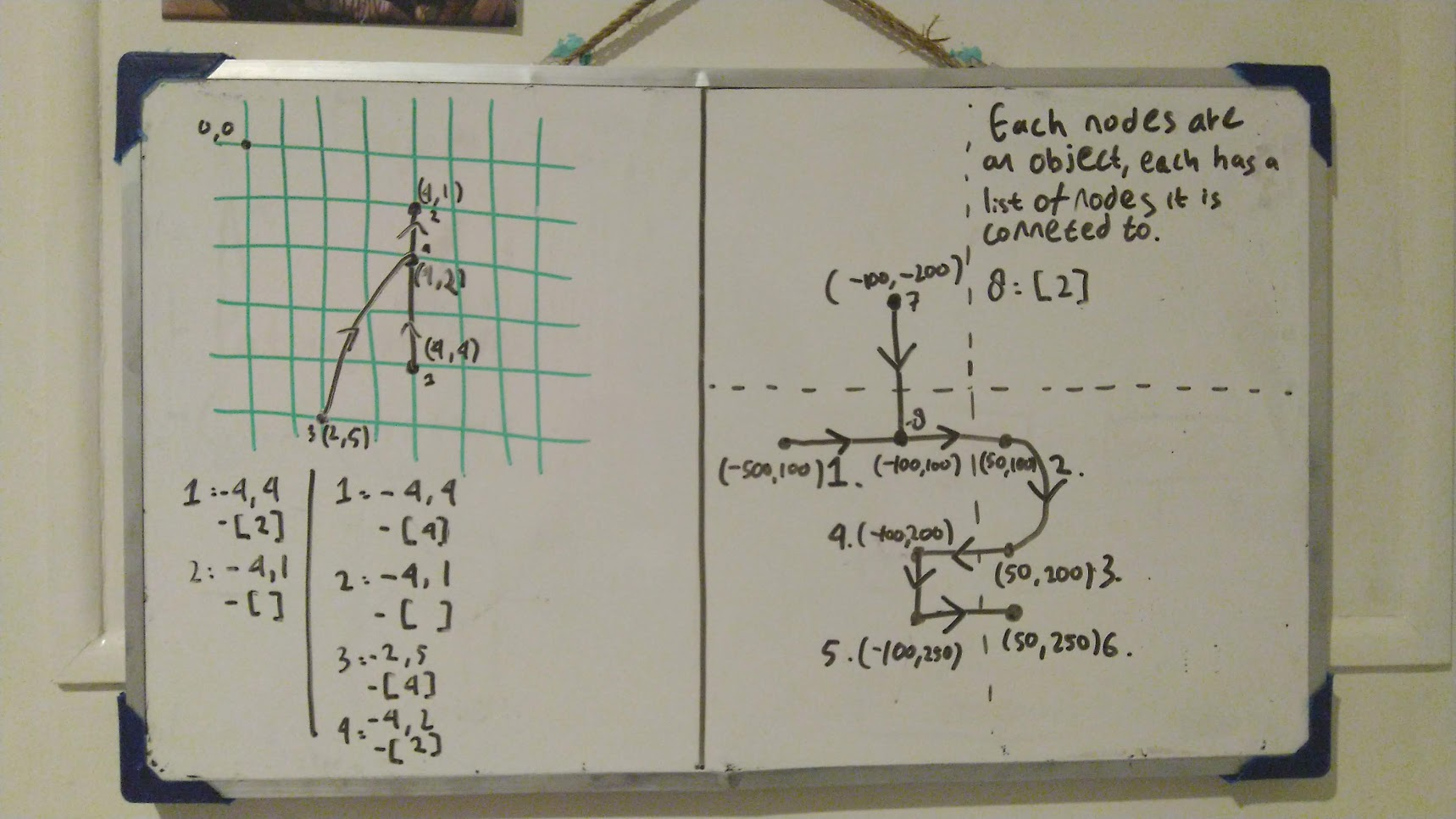


Image 9. Road creation schematics.

Now road creation is a little bit tricky. It can get out of hand really fast if you don’t plan it accordingly. There is a lot of vector mathematics involved in making roads connect to each other with proper data. Also, at times, it can be quite hard to code efficiently. Therefore, you have to be careful to avoid bugs in the future of your code.

Creating the first node is relatively not too hard. First, you need to check whether you are creating the first node on an empty space, or directly above another road. To know that, you can check whether there is an intersection between the point you are clicking, and every road -line coordinates on the visible map. If a intersection is detected, you can get the point of intersection by projecting the point onto the line, then interpolating the projected scalar. Or, if you are too lazy, just use the shapely library to check the intersection AND get the coordinates at the same time.

For the next node, you are starting to create real roads in the map. For this, you have to check the road-line intersection, with the other roads in the visible map. If an intersection is detected, let the player have a choice, whether to connect the roads together, to create a new node on an empty space, or not at all. Connecting the roads can be done by calculating the length from the current mouse position, to the road intersection point.

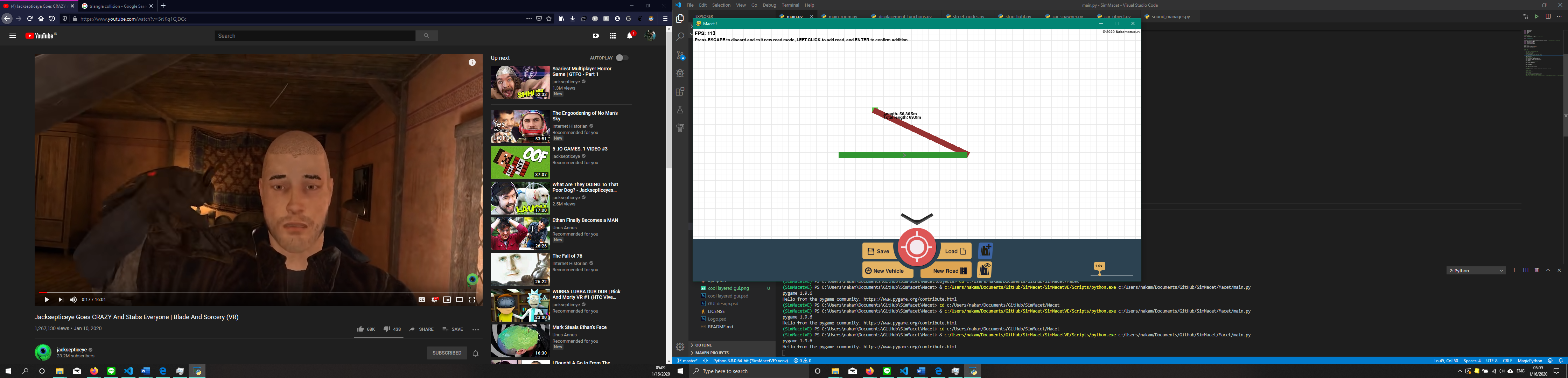


Image 10. Road cannot be created in such extreme angles.

It is also recommended for the program to check the angles made by the current road you are controlling with your mouse with the previous road made. The angles then are checked so that it does not make an extreme angle, in which the car has to do very sharp turns to get to another road, as that is bad road design.

1. **Saving and loading files**

{

    "StreetNodes": {

        "id": [[coordinates], ["Backnodes id"], ["frontNodes id"]]

    },

    "StopLights": {

        "id": [[coordinates], str:"NodeAnchor id", int:greenDuration, int:redrDuration]

    },

    "CarSpawners": {

        "id": [[coordinates], str:"NodeAnchor id", int:maxSpeed, int:interval]

    }

}

Using the internal .json and TkInter library, saving and loading map files with street nodes, stop lights, and car spawners are possible.

This is possible by giving every object that needs to be saved unique id’s, so that every object has a unique identifier so that everything can be connected to each other, especially for the street nodes object.

For every object that needs to be saved in the screen, get their important variables, and add them into the json file. Variables that needs to be saved into the file, are the variables that is needed in initializing each object.

Using the TkInter library, it is possible to create a new window that is a prompt where to save or open the wanted file with a specified name.

To load the files into the game, the json library is used to load it into a variable in the program. Then, every needed variable is read. Furthermore, using the variables, new objects are initialized and put into the game.