Ayo Dele-Adeniyi

DODGE RACER

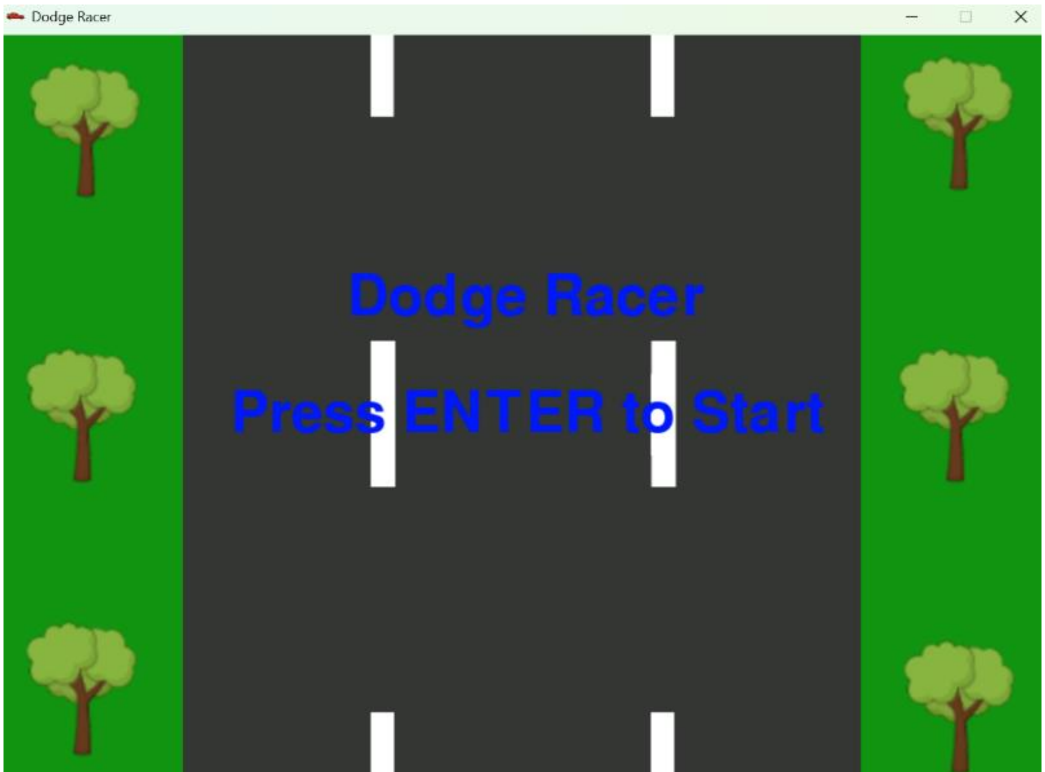
Contents

INTRODUCTION 2
ANALYSIS 3
ALL FEATURES THE MAKE THE PROBLEM SOLVABLE THROUGH COMPUTATIONAL METHODS 3
STAKEHOLDERS 6
RESEARCH THE PROBLEM 7
LIMITATIONS OF PROPOSED SOLUTION 8
HARDWARE AND SOFTWARE REQUIREMENTS OF SOLUTION 9
SUCCESS CRITERIA 11
DESIGN 12
BREAKING DOWN THE PROBLEM SYSTEMATICALLY 12
TESTING 14
POST-DEVELOPMENT DATA 15
DEVELOPING THE CODED SOLUTION 17
EVIDENCE OF EACH STAGE OF THE ITERATIVE DEVELOPMENT PROCESS 17
TESTING TO INFORM DEVELOPMENT 24
PROVIDE EVIDENCE OF TESTING AT EACH STAGE 24
EVALUATION 31
TESTING TO INFORM EVALUATION 31
USER FEEDBACK 31
GAMEPLAY METRICS 35
PREFORMANCE DATA 35
BUG REPORT AND ERROR LOGS 36
DATA ANALYSIS REPORT 37
EVALUATION OF SOLUTION 38
TEST EVIDENCE AGAINST SUCCES CRITERIA 38
USABILITY FEATURES 40
CODE LISTING 44

Ayo Dele-Adeniyi

INTRODUCTION

The problem my game aims to solve is the repetitive and predictable nature of many 2D car games and endless runners like Subway Surfers. Unlike traditional games where movement is restricted to fixed lanes and obstacles appear in fixed patterns, my game introduces dynamic movement, randomized obstacle placement, and adaptive difficulty scaling. These features make it solvable by computational methods, as they rely on real-time player input handling, random number generation, and time-based difficulty adjustments



ANALYSIS

ALL FEATURES THE MAKE THE PROBLEM SOLVABLE THROUGH COMPUTATIONAL METHODS

This game makes good use of the Pygame library to handle images, user input, and game logic. It comes with built-in tools for rendering pictures, finding collisions, handling keyboard input, and controlling the game loop. This makes making an interactive game a lot easier. Pygame makes complex tasks easier, like drawing objects on the screen and changing where they are, so I can focus on the mechanics of the game instead of the basics. This method can be computed because it uses optimised algorithms to handle graphics and input processing, which makes sure that everything runs smoothly. Pygame's structured system also lets you write code that can be used again and again, which makes it easier to build, test, and keep up with the game.

The game window and display setup set the game's resolution, title, and icon, as well as the visual area where it takes place. Pygame's display functions are used to make this happen on the computer. These functions give you exact control over the screen size and graphics. everything looks the same on all devices by setting fixed dimensions and changing parts on the fly to fit the screen. It also lets images, the background, and the user interface (UI) change in real time, which makes the game more interactive and visually appealing. This method is flexible because it lets changes be made without having to redraw or move parts by hand.

Implementing Dynamic Player Movement allows the player to move freely omnidirectionally rather than being restricted to preset lanes. The game continuously detects key presses and adjusts the player’s position accordingly, ensuring smooth and responsive movement. This is efficiently managed using arithmetic operations and condition checks, which a computer can process rapidly within each frame. Since computers can handle these calculations at high speeds, they provide a reliable way to create interactive and engaging gameplay.

Boundary enforcement ensures that the player’s car stays within the playable area by limiting movements within set limits on the screen. This prevents the car from moving off-screen, which could cause gameplay issues or visual glitches. In this game, boundary enforcement would be implemented using mathematical constraints, such as the max() and min() functions, to keep the player's position within a valid range. This approach is well-suited for computation because it relies on simple numerical comparisons and is efficient. It also prevents errors and maintains visual consistency. This makes it a reliable method for managing movement within the game environment.

To make obstacles in the game, they are placed at random, moved lower, and checked to see if they will hit the player. Spacing rules are used when obstacles are created to keep them from being placed unfairly, and they are removed once they leave the screen to improve performance. Structured steps like loops, conditional lines, and math calculations can be used to handle it quickly and easily. As the game progresses it can change how obstacles behave, making sure that the player has a difficult but fair experience. Hitboxes and logical comparisons are also used to handle collision recognition, which makes it possible to do accurate and quick calculations to figure out when a crash happens.

The game loop is the main structure that updates the state of the game, handles user input, moves items, checks for collisions, and shows graphics on the screen. It keeps the frame rate steady while doing these jobs in a controlled order to make sure the game runs smoothly. This method can be used for computation because iterative execution lets the program handle many jobs at once. By using a loop, the game only does the calculations it needs to do every frame, which cuts down on work that isn't needed. It also lets you make changes that are dynamic, like making the game harder over time, which makes it more interesting and flexible.

Dynamic difficulty scaling adjusts the game's challenge level based on the player's progress, ensuring a balanced and engaging experience. In the game, the obstacle speed increases over time, making it progressively harder as the player survives longer. This is implemented by checking elapsed time and increasing speed at regular intervals. A computational approach is well-suited for this because it allows real-time adjustments based on predefined conditions, ensuring smooth and automated difficulty management. This method enhances replay ability by adapting to the player's skill level without requiring manual intervention, creating a more engaging and personalized gameplay experience.

Randomisation is used in the game to decide where and how often obstacles show up, making sure that each game is different and fun for the player. When the random module is used, obstacles appear at random times and places, which stops trends that would make the game too easy or boring. Computers can quickly come up with and use random numbers, which ensures variety while keeping the rules of fair play in place. This method makes it easier to play again and again because each session has a different challenge. This keeps players interested without having to create multiple different levels.

The game over system knows when a player hits an obstacle and takes them to a special screen that shows their score and lets them decide to start over. Collision recognition checks to see if the player's hitbox overlaps with an obstacle's hitboxes. A computer-based method works well for this because it can accurately find conditions that end the game in real time without any help from a person. To make sure there are smooth changes between game sessions, the system can also reset game variables, remove obstacles, and restart the game quickly.

The scrolling background makes it look like the visual is moving by moving it down all the time and putting it back where it was when it goes off-screen. This effect makes players feel like they are driving their car, but they aren't actually moving it. The background can be moved automatically every frame, which makes sure the animation runs smoothly. The speed of scrolling can also be changed automatically based on game conditions, such as making the game harder over time.

Some important UI features and menus for the player are the start menu, the game over screen, and the score display. Text and images are shown on the screen based on the current state of the game to apply these parts. dynamic changes like showing different messages based on how the player is doing or how the game turns out are all included. Event handling can also be used to track player input, like hitting "Enter" to start or restart the game. This makes sure that the game flows smoothly from one state to the next. This way of doing things makes the UI essential if you want a smooth experience and not have to run the code again to restart it

Different parts of the game are controlled by game state management, like the start menu, the actual game, and the screen that says "game over." To do this on a computer, variables and conditional reasoning are used to keep track of and switch between states. It can smoothly switch between states, which lets the game stop, reset, or restart without having to restart the program. It also lets you add new states quickly, which makes it scalable, and it lets you make changes in real time based on what the player does. A structured management system is necessary for game creation because it stops mistakes and makes sure the experience runs smoothly.

STAKEHOLDERS

Stakeholder 1: Casual Computer Gamers

Casual mobile gamers play games for quick bursts of entertainment and relaxation. They like games with easy-to-understand controls, fun tasks, and no need to learn a lot. Unlike in other games, where obstacles appear in the same limited lane format, my game keeps each session fresh by randomly generating obstacles that the player can dodge using the full omnidirectional movement. This keeps players engaged without feeling like they are repeating the same experience. Due to the timely difficultly increase and score count, the game also has a strong replay ability factor for casual players who want to improve/see how far they've come. My experience also has and easy to understand and processes gui and controls so casual gamers can simply jump into the experience without much knowledge or prior understanding

Stakeholder 2: Competitive Players & Speed runners

Fans of competitive gaming and speedrunning are looking for games that award skill, strategy, and accuracy over luck. My game provides an ideal experience for them by introducing dynamic movement, allowing players to optimize their dodging techniques instead of being limited to predefined lanes. This gives skilled players more freedom to improve their movement and develop advanced strategies. In addition, obstacles show up at random instead of following a set plan, so memorising is not enough to win. High scores require quick reflexes and exact movement, which makes the game more fun for people who like to learn how to play and beat records. Competitive players and speed runners seek games that continuously challenge their skills, my game caters to this by ensuring that as the player survives longer, the game gradually increases in difficulty.

Sources

According to research from Startquestion, “casual gamers tend to prioritize relaxation, accessibility, and short play sessions, while competitive gamers seek challenges, mastery, and opportunities to improve their skills. Casual players often prefer simple controls and forgiving mechanics, whereas competitive players value increasing difficulty, precise mechanics, and skill-based progression. Taking this into account, my game balances both audiences by incorporating intuitive controls for accessibility while progressively increasing difficulty to engage competitive players."

According to gamedeveloper, “Understanding the preferences of casual and competitive gamers is crucial for designing a game that appeals to both audiences. Casual gamers often seek games that are easy to learn, provide immediate enjoyment, and allow for short, flexible play sessions. They may prefer games with simple mechanics and forgiving difficulty levels, enabling them to engage in a relaxed and leisurely manner. In contrast, competitive gamers are driven by the desire to overcome challenges and achieve mastery. They are more tolerant of frustration, viewing significant obstacles as opportunities to gain a sense of accomplishment and "bragging rights" upon success.”

RESEARCH THE PROBLEM

To ensure that my game remains fun and engaging, I extensively researched existing 2D car games and endless runners, identifying key weaknesses that make their gameplay feel repetitive. One of the most common limitations is restricted movement, as seen in games like *Subway Surfers*, which only allows players to switch between three fixed lanes. This constraint makes gameplay predictable since players always have the same movement options, reducing the need for strategic decision-making. Additionally, obstacle placement tends to be repetitive, with the same patterns appearing frequently, allowing players to rely on memory rather than quick reactions and adaptability.

Temple Run is one game that does a good job of trying to deal with these problems. Temple Run adds free horizontal movement by letting players tilt their devices to change their place on the fly, unlike lane-based movement. This makes the game more interesting and stops it from being too predictable like in fixed-lane games. But Temple Run still has some problems. As a mobile game, it relies on an accelerometer to track your movements, which can feel rough or hard to control at times. Furthermore, the game lets you freely move left and right, but not up and down. This makes it harder for the player to change their speed in response to barriers.

Building on this research, my game aims to improve upon these solutions by introducing full omnidirectional movement, allowing players to move both horizontally and vertically. The ability to move up and down adds more strategic layers because players can speed up to avoid danger or slow down to reposition themselves before a barrier appears. This adds a new level of skill expression and decision-making that isn't present in other endless races. Using arrow keys instead of motion controls also makes handling more exact and responsive, since my game is meant to be played on a PC. This is because tilting a device to move can be awkward.

My game fixes the problems with current control schemes, the predictability of lane-based movement, and the repetitiveness of obstacle patterns. It also makes the experience more dynamic and skill-based, which keeps players interested for longer.

LIMITATIONS OF PROPOSED SOLUTION

My proposed game is better than standard endless runners because it lets you move in any direction and doesn't have fixed lanes, but it also has some problems that come with it. One big problem is that it's harder to keep control. For lane-based movement, players can quickly swipe left or right to move between positions that have already been set. For omnidirectional movement, however, players need to be more exact with their inputs. This means that new players might find it hard to control their movements well, especially when they need to react quickly in a small area. In order to fix this, I have made sure that obstacles gradually increase in speed at set 10 second intervals so that players have enough time to make choices without getting too stressed out.

The chance that players will get lost is another problem. The structured three-lane method in many endless runners makes it easy for players to guess what will happen next. But in my game, letting players move around freely means they have to constantly check where they are in relation to obstacles, which can make it harder to see gaps or safe zones. To keep this from happening, I make sure that obstacles are always spread out in a sizeable distance away from each other and that their placement is always fair for the gamer. This keeps things from getting impossible, which could be frustrating.

Full vertical movement also makes balance issues more likely. Some players might try to stay in one place or move too slowly, which would slow down the game since they can move forward and backward. This might make people less interested, since part of the fun of endless runners is the idea of always moving forward. To stop this from happening, the speed of my game slowly speeds up over time. This forces players to keep moving and stops them from being too defensive (finding a loophole in getting a higher score).

Lastly, using keyboard controls instead of motion-based input makes sure that you are accurate, but it may take away from the immersion that some players look for in endless runs. Tilting is used in games like Temple Run to make the experience more physical, while arrow keys are more traditional and straight. However, since my game is meant to be played on a PC and not a phone, keyboard keys are more useful and allow for better responsiveness, so the trade-off is fair.

Even with these problems, the suggested answer improves the depth and unpredictable nature of the game while still being fair and easy to use. Careful changes to the game's design, like slowing down movement and placing obstacles in a planned way, have kept it difficult and fun.

HARDWARE AND SOFTWARE REQUIREMENTS OF SOLUTION

To ensure my game runs smoothly and efficiently, I have identified the necessary hardware and software requirements. These requirements are chosen based on the game's mechanics, graphical demands, and input handling.

Hardware Requirements

1. Processor (CPU) – Minimum: Dual-Core, Recommended: Quad-Core or higher
    - The game constantly changes in real time how the player moves, where obstacles are placed, when collisions happen, and how the background scrolls. A multi-core central processing unit makes sure that the game runs smoothly and without lag by processing different parts of the game quickly.
2. RAM – Minimum: 4GB, Recommended: 8GB or higher
    - Since Pygame handles real-time rendering and input detection, sufficient memory is required to store active game elements like icons, packages, and background images. More RAM prevents stuttering and allows for seamless performance.
3. Graphics Processing Unit (GPU) – Integrated Graphics (Minimum), Dedicated GPU (Recommended)
    - Since the game mostly uses 2D graphics, you don't need a very powerful GPU. Having at least an integrated graphics card, on the other hand, makes sure that animations and picture rendering work well. A specialised GPU would make things even better, especially when frame rates are high.
4. Storage – Minimum: 500MB Free Space
    - The icons, Python code, and background pictures that make up the game's assets need a place to be stored. The game files don't take up much room (500MB is enough) but you should have more space for updates and data that is created during the game.
5. Input Devices – Keyboard and Mouse (Mandatory)
    - The game is designed for PC gameplay, using arrow keys for movement rather than motion-based controls. A keyboard is essential for precise input and a mouse is needed for necessary clicks like running and ending the code

Software Requirements

1. Operating System – Windows, macOS, or Linux
    - The game can run on any major PC OS because Pygame is a cross-platform library. But Windows is what most casual players use, so the game is made to work best on that platform.

2. Python 3.x

    - The game is developed using Python 3 and relies on Pygame, which is compatible with all Python 3 versions. Older versions of Python (2.x) are not supported due to syntax and feature differences.

3. Pygame Library

    • Pygame comes with built-in functions for rendering images, detecting collisions, handling events, and creating game loops. These functions make development easier and boost speed. Therefore, the most recent package must be loaded.

4. Random Library (for Obstacle Placement)

- The random module in Python is used to make new obstacles in the game all the time, so each time you play it is different. This makes sure that the game is unpredictable, which keeps it interesting.

5. Frame Rate Management (60 FPS Recommended)

- The game is designed to run at 60 frames per second so that the graphics stay smooth. The Clock method in Pygame helps control the frame rate, which makes sure that the experience is consistent and responsive.

Justification for Requirements

The chosen hardware provides smooth performance without using too many resources, so a lot of people can play the game. The software requirements focus on making sure the game works well on multiple platforms, is easy to create, and has the best performance possible. This makes sure the game runs smoothly while still being fun.

Making sure that both the hardware and software needs are met lets the game run quickly and smoothly, without any lag. This makes it fun to play and safe for your computer.

SUCCESS CRITERIA

1. Engaging and Satisfying Game Length

- Measurement: Average session time should be at least 3–5 minutes for casual players and longer for skilled players. This will be tracked via in-game timers.
- Justification: If the game is too short, it may feel unsatisfying; if it is too long, it can become tedious. A balanced session time promotes replay ability and maintains player interest.

---

2. Accurate Score Tracking System

- Measurement: The game must display the player's score in real-time during gameplay and show the final score at the end of the session. The score should reset correctly upon restarting.
- Justification: A real-time, responsive score system gives players immediate feedback, rewarding progress and motivating replay in an endless runner game.

---

3. Dynamic Difficulty Scaling

- Measurement: The game should gradually increase in difficulty over time — for example, by increasing obstacle speed or spawn frequency every 30 seconds or after certain score thresholds.
- Justification: Scaling difficulty maintains challenge for all player skill levels, preventing the gameplay from feeling too easy or repetitive.

---

4. Randomised Obstacle Generation

- Measurement: Obstacles should be randomly generated with varied patterns each session, while still ensuring fair and possible paths for the player.
- Justification: Randomised obstacle patterns increase replay ability and prevent the game from feeling predictable or stale.

DESIGN

BREAKING DOWN THE PROBLEM SYSTEMATICALLY

Firstly, Using Pygame to make a game means doing basic things like making a game screen and adding a background picture. With the game screen as the main viewing surface, all visual elements are shown, making it possible to play. Adding a background picture improves the visual appeal and immersion, giving players a sense of place and time that keeps them interested. For this to work, you must first initialise Pygame and set the display window to the right size. Then, load the background picture you want and draw it onto the screen at the right spot, which is usually the upper left corner (0, 0). This process makes sure that the background is drawn before any other game features, which keeps the layers in the right place.

Drawing a car icon and figuring out how it moves are two of the most important steps in using Pygame to make a car-based game. The car icon is the player's avatar, giving them a visual image that makes the game more fun and real. To do this, you use pygame.image to load the picture of the car then use screen.blit() to put it on the game screen at certain points. This process makes sure that the car is shown correctly in the game world so that players can find it and interact with it properly. Player movement is one of the most important things that can be done to improve interactivity and user interest. In Pygame, one popular way to do this is to monitor for keyboard inputs and move the player on the screen to match. Starting with Pygame's pygame.key.get_pressed() function, which gives you a list of key states. It returns a list of Boolean values (True or False), with each position in the list representing a specific key. If a key is pressed, the corresponding position will be True; if it's not pressed, it will be False. The game can figure out which keys are being pressed by looking at this order. This means that if the right arrow key is hit, the player can move the character to the right. The figure would also move to the left if you pressed the left arrow key. This method makes it easy and quick to control player movement, making sure that the game responds to what the user does. As long as you hold down a key, you can keep moving, making the game play smooth and easy to understand.

It's important to set player limits in games so that the player's character stays in the playable area on the screen. If there were no limits, the player could move the figure off-screen, which would make the game less enjoyable or allow cheating. By putting these lines down, we can stop the player's character from going where they shouldn't. During this step, we set the player's limits for moving left and right and up and down. For example, the player's car shouldn't be able to go past the left or right sides of the road I made. It should also be able to stay inside the top and bottom lines. We use mathematical functions that match the player's position to the limits that have been set to make sure this happens. If a player moves too far in one way, the position is changed so that the player stays within the lines. This method saves a lot of computer time because it only uses easy number comparisons (like the max and min functions) to make sure the player's position stays correct. We can be sure that the player's character will always be in the right place by doing this at every frame of the game. This keeps the game's graphics and features from breaking. This also helps keep the player's experience constant, since they won't be able to do anything that would make the game stop working.

To create obstacles that fall down the screen at random positions, we first randomly generate the horizontal position of each obstacle using 'random.randint()', ensuring the obstacle stays within the screen's width. The vertical position starts off-screen at the top, placing the obstacle just above the visible area. This allows it to fall into view as the game progresses. A crucial part of this process is maintaining a minimum horizontal gap between obstacles. We check the distance between the new obstacle and existing ones by comparing their horizontal positions. If the gap is smaller than the predefined `min_horizontal_gap`, the obstacle is not placed, and the program tries again, for a maximum of 10 times. This ensures that obstacles are spaced out evenly, preventing them from being too close and making the game fairer for the player. The obstacles move down the screen each frame by increasing their `y` position based on the speed. Once an obstacle moves off-screen, it's removed from the list to maintain performance. New obstacles are generated randomly, with a 5% chance per frame and a 25% chance for an additional obstacle. This randomness keeps the game dynamic and unpredictable. Overall, the combination of random generation, proper spacing, and obstacle movement ensures the game is challenging while providing players with enough room to navigate.

I'm going to make the obstacles move a little faster every ten seconds so that the game doesn't always offer the same level of challenge. The longer a player stays alive, the harder the game gets, which makes the game interesting. If this wasn't there, the task would stay the same, and skilled players might get bored after a while. I will use a timer to keep track of how long the person has been in the game to do this. Once ten seconds have passed, the game will slightly speed up the objects that are falling. I'll also use the modulus method to make sure this only happens once every ten seconds. Mod 10 should always equal zero. This method makes sure that the level of difficulty doesn't rise all at once, but instead rises slowly and steadily. It's easy to use this method, and it adds a useful progression system that pushes players to stay focused and act faster over time. It speeds up the game and makes players feel better about themselves the longer they last.

In order to detect when the player crashes into an obstacle, I will include a collision detection system using hitboxes. A hitbox is an invisible shape usually a rectangle that surrounds a visual object in the game and is used to check whether it overlaps with another object's hitbox. This is a very common technique in 2D games for handling things like damage, item pickups, or in this case, ending the game. For my game, I plan to define a hitbox for the player's car and for each obstacle. However, rather than using the full size of the car image, I will shrink the player's hitbox slightly. This means the rectangle used for collisions will be a bit smaller than the actual image shown on screen. This technique helps the game feel fairer to the player - so the game doesn't count a collision just because the very edge of the car brushes against an obstacle. It allows a little forgiveness and gives the player more control. Each time the game updates, it will check whether the player's hitbox overlaps with any of the obstacle hitboxes. If they do overlap, this will count as a crash, and the game will trigger the game over process. This approach is efficient, easy to implement in code, and provides a good balance between accuracy and fairness for the gameplay.

I'm going to add a score feature that goes up as the player lives. This will give the player feedback on how well they did and give them a sense of accomplishment. This will help players see how far they've come and motivate them to keep getting better every time they play. A score that can be seen also makes the game more competitive, pushing players to beat their old high scores or fight with other players. To do this, I will use the time that has passed since the game began to keep track of how long it has been going. The player will get more points the longer they can stay away from obstacles. It's fair and easy to understand how to score because the longer you stay living, the higher your score. This number will also be shown clearly on the screen while the game is being played, so the player can see it change as the game goes on. The method is easy to use and reliable because the score is based on time. It keeps you from having to track more complicated things like avoiding obstacles or how far you've driven, but it still rewards skilful play. This method also works for my fast-paced game, where survival and quick responses are the most important things.

I am going to add both a start menu and a game over screen to make the game easier to use and give it structure. These features help guide the player through the game and make it more polished and easy to understand. After starting the game, players will see the start screen. The title of the game and a clear sign telling the player to press a key (like ENTER) to start will be on it. This gives the player time to get ready before the game starts and keeps them from being thrown into it without warning. You can make the menu feel more professional and open by adding the game road as a background, maybe in a blurry way. If the player hits an object, the game over screen will show up. The player will see a message like "Game Over!" and their end score on this screen, which tells them how well they did. Also, there will be a message that tells the player how to start the game over, such as by pressing ENTER again. So, the game loop stays smooth, and you don't have to restart the whole app or get confused. A loop will be used to make both screens work. The loop will wait for the right input before moving on. The event handling system in Pygame makes this easy to handle, and it lets the game pause at important points without freezing or crashing. These screens make the game easier for people to use and manage, and they bring the quality closer to that of professional games.

TESTING

I carefully picked scenarios that cover the most important parts of the game so that I could find and explain test data for the iterative development of my game. With these test cases, I can check the game's main features, like how the player moves, how obstacles are created, and how speeds change. This will help me make sure the game works, is fair, and is fun to play.

For player movement, I plan to test it in all four directions (left, right, up, and down) to make sure the player can move easily and stay inside the game area's lines. I will also test edge cases, like trying to move the player past the edge of the screen to see if the limits are being followed properly. This data will be used to see how the game responds to computer input, making sure that the speed of movement is right and that there are no lags or delays.

For collision detection, I'll use a number of test cases to make the player hit objects in a number of different ways. I will interact with obstacles that are at different heights and across the screen, and I will make sure that the game correctly identifies collisions. Besides that, I'll try edge cases where the player is right on the edge of an obstacle's hitbox. That way, you can be sure that the hitboxes are the right size and that the collision recognition system doesn't give you false positives or negatives.

Because collisions are so important to the game, this is a very important place to test.

Another important feature is the ability to increase speed. To test the system, I will check every 10 seconds to see if the obstacle speed increases properly. I'll keep an eye on the time while you play to make sure the speed boost happens as planned and stays the same throughout the game. I will also try extreme cases where the game runs for longer periods of time to see if the faster speed causes difficulty spikes that make the game too easy or impossible to play.

For obstacle creation, I will make sure that obstacles appear at random across the screen, leaving enough space between them so that players can avoid them. To be more specific, I will test how obstacles are placed at random by observing them as they fall and making sure they don't touch. I will also check the minimum horizontal gap limit to make sure that obstacles are not too close to each other. If they are, the game would not be fair or fun for the player. In addition, I will check to see if the game consistently adds new obstacles at the right rate, so players can plan for them but still find the game difficult.

These test cases were picked to cover both normal games and very rare situations that might not be clear at first glance. By trying every part of the game, I can find and fix any problems early on in the development process. This makes sure that the game works well and gives players a good experience. With each round of testing, I'll learn more about how the rules work and how to make the game better for the user. By checking and developing my game in small steps, I can keep it interesting and free of major bugs.

POST-DEVELOPMENT DATA

The focus moves to improving the game based on feedback from real users and performance analysis in the post-development phase. More test data will be gathered to look at the game's long-term playability, user engagement, and general performance. This will be done to make sure the game is not only practical but also fun. This information will help improve the game, fix any problems that come up, and make it run better.

User Feedback: Once the game is ready for everyone to play, user feedback will be very important for finding places where players are having trouble or getting frustrated. This information will be gathered by talking to people who playtest the game and watching how they interact with it. It will give you qualitative information about how hard the game is, how it works, and how much fun it is generally. Some of the things that will be tried are how responsive the controls are, how well the difficulty of the obstacles is balanced, and how well the visual signals for obstacles and score are understood.

Gameplay Metrics: After the game is finished being developed, I will keep track of things like the average score, the average player retention, and how many rounds have taken place in the average gaming session. This numerical data will show how interesting the game is, how hard it is to move forward, and if there are any parts that could be made harder or easier to balance the experience for the players. For instance, if most players are always getting to a certain point or time, you might need to change how hard it is. Also, if players are dying too quickly or too slowly, it could mean that the speed boosts need to be rebalanced or the obstacles need to be placed more evenly.

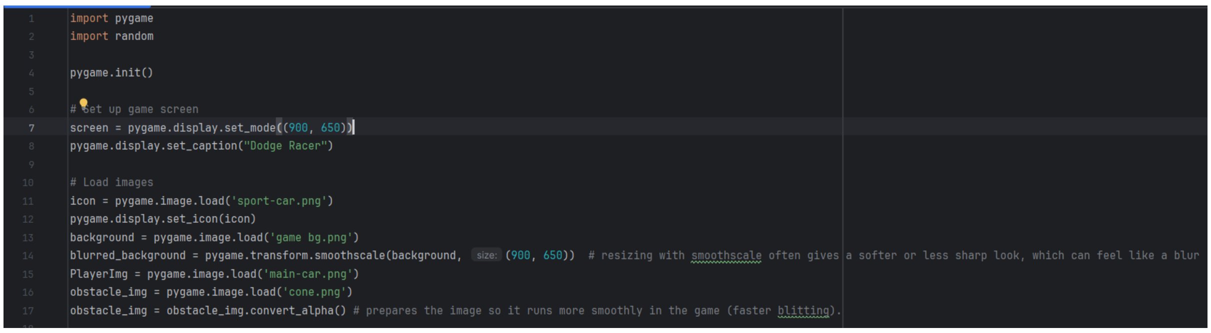
Performance Data: After the game comes out, it's important to keep an eye on how it works on different platforms, especially if it's played on different hardware. This means keeping track of frame rates, load times, and any crashes or drops in speed. This kind of information will help you figure out if the game works well on a number of different systems. This can help make things work better, make games more stable, and make sure that everyone has a good time playing.

Bug Reports and Error Logs: Bug reports and error logs will be very important after the game has been made and people have played it. This information will show you any bugs, crashes, or other issues that were missed during the testing and development stages. It's important to keep an eye on these reports on a regular basis and fix problems like failed collision recognition, graphical glitches, or performance issues that could affect how the player feels.

In the end, both qualitative and quantitative data will be used a lot in the post-development process to make the game even better. Feedback from users, measurements for gameplay, performance data, bug reports, and long-term engagement data will all help make the game better and make sure it stays fun and useful for players.

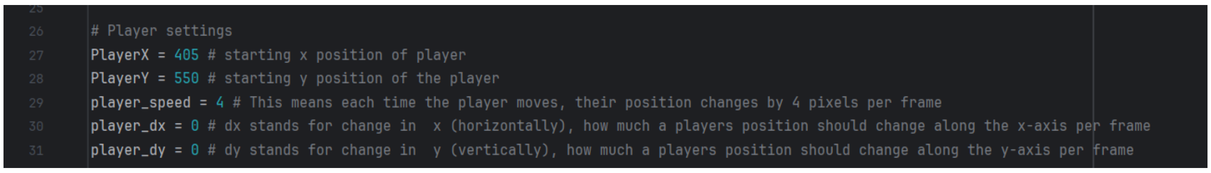
DEVELOPING THE CODED SOLUTION

EVIDENCE OF EACH STAGE OF THE ITERATIVE DEVELOPMENT PROCESS



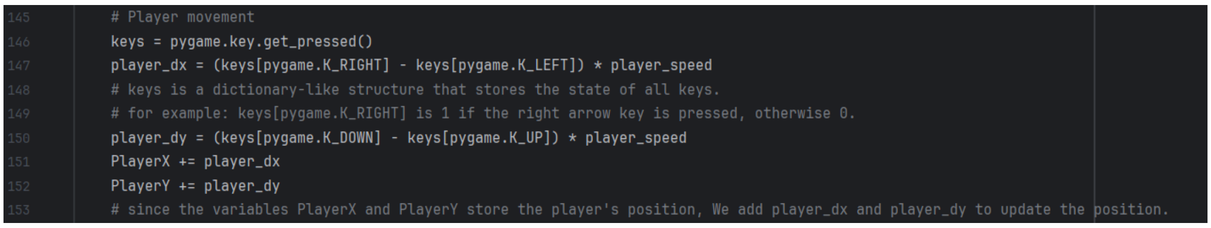
```python
import pygame
import random

pygame.init()

# Set up game screen
screen = pygame.display.set_mode((900, 650))
pygame.display.set_caption("Dodge Racer")

# Load images
icon = pygame.image.load('sport-car.png')
pygame.display.set_icon(icon)
background = pygame.image.load('game bg.png')
blurred_background = pygame.transform.smoothscale(background,  size: (900, 650))  # resizing with smoothscale often gives a softer or less sharp look, which can feel like a blur
PlayerImg = pygame.image.load('main-car.png')
obstacle_img = pygame.image.load('cone.png')
obstacle_img = obstacle_img.convert_alpha() # prepares the image so it runs more smoothly in the game (faster blitting).
```

To set up the game visually, I first defined the game screen size using pygame.display.set_mode((900, 650)). This creates a fixed window for the game to run in, ensuring everything fits and looks consistent. I also set a title for the game window with pygame.display.set_caption("Dodge Racer"), which gives the game a clear name when running. Then, I loaded all the necessary images, such as the background, player, and obstacle images, using pygame.image.load(). For the background, I used pygame.transform.smoothscale() to resize it smoothly to match the screen size, giving it a soft, blurred effect that helps the text stand out better. I also used convert_alpha() on the obstacle image to make it render faster and support transparency properly, which improves both performance and visual quality during gameplay.



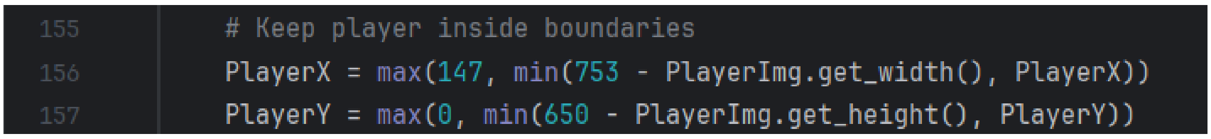
```python
# Player settings
PlayerX = 405 # starting x position of player
PlayerY = 550 # starting y position of the player
player_speed = 4 # This means each time the player moves, their position changes by 4 pixels per frame
player_dx = 0 # dx stands for change in  x (horizontally), how much a players position should change along the x-axis per frame
player_dy = 0 # dy stands for change in  y (vertically), how much a players position should change along the y-axis per frame
```

To set up the player character, I defined starting coordinates for the car using PlayerX = 405 and PlayerY = 550, which places it near the bottom centre of the screen. I also set player_speed = 4 so the player moves 4 pixels each time a key is pressed, ensuring smooth and noticeable movement. I used player_dx and player_dy to store how much the player should move horizontally and vertically in each frame. These values are updated later based on which arrow keys are pressed, allowing the player to move the car in any direction during gameplay.



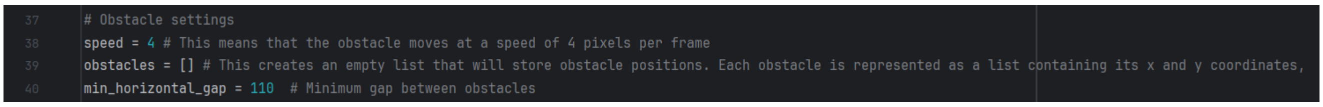
```python
    # Player movement
    keys = pygame.key.get_pressed()
    player_dx = (keys[pygame.K_RIGHT] - keys[pygame.K_LEFT]) * player_speed
    # keys is a dictionary-like structure that stores the state of all keys.
    # for example: keys[pygame.K_RIGHT] is 1 if the right arrow key is pressed, otherwise 0.
    player_dy = (keys[pygame.K_DOWN] - keys[pygame.K_UP]) * player_speed
    PlayerX += player_dx
    PlayerY += player_dy
    # since the variables PlayerX and PlayerY store the player's position, We add player_dx and player_dy to update the position.
```

To allow the player to move the car, I used pygame.key.get_pressed() to detect which arrow keys are being held down. This function returns a list of all keys, where each key is marked as pressed (1) or not pressed (0). I calculated the horizontal movement (player_dx) by subtracting the value of the left key from the right key and multiplying it by the player's speed. I did the same for vertical movement (player_dy) using the up and down keys. Finally, I updated the player's position by adding these values to PlayerX and PlayerY, which keeps the player moving smoothly in response to the key inputs. This method gives the player full directional control while keeping the movement smooth and responsive.



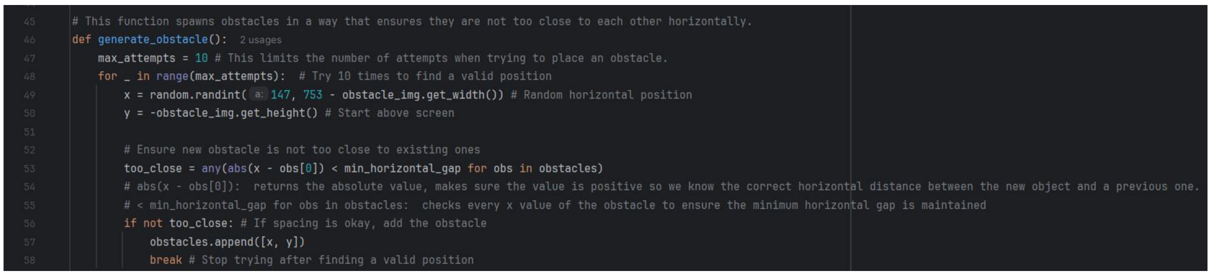
```python
    # Keep player inside boundaries
    PlayerX = max(147, min(753 - PlayerImg.get_width(), PlayerX))
    PlayerY = max(0, min(650 - PlayerImg.get_height(), PlayerY))
```

To make sure the player stays within the game screen and doesn't move off-screen, I used boundary checks for both the X and Y positions. The line PlayerX = max(147, min(753 - PlayerImg.get_width(), PlayerX)) ensures the player's car stays between the left and right edges of the road, accounting for the car's width. Similarly, PlayerY = max(0, min(650 - PlayerImg.get_height(), PlayerY)) keeps the car from moving above the top or below the bottom of the screen. This helps maintain a consistent gameplay area and prevents graphical glitches or unfair gameplay.



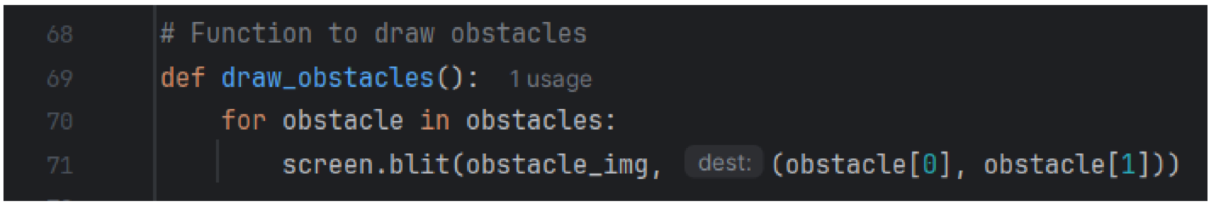
```python
# Obstacle settings
speed = 4 # This means that the obstacle moves at a speed of 4 pixels per frame
obstacles = [] # This creates an empty list that will store obstacle positions. Each obstacle is represented as a list containing its x and y coordinates,
min_horizontal_gap = 110  # Minimum gap between obstacles
```

I set up the obstacle system by first defining a speed variable, which controls how fast each obstacle moves down the screen — in this case, 4 pixels per frame. I also created an empty list called obstacles, which will store the position of each obstacle as a pair of X and Y coordinates. This makes it easier to update and draw multiple obstacles on the screen. To prevent obstacles from spawning too close to each other, I introduced a min_horizontal_gap value. This ensures there is a minimum space between obstacles horizontally, which helps maintain fair gameplay by giving the player enough room to move and react.



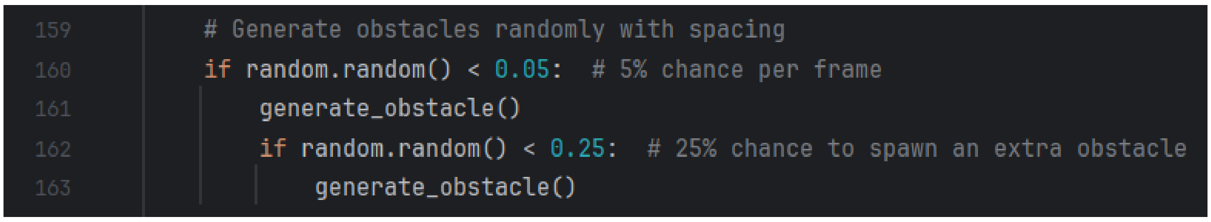
```python
# This function spawns obstacles in a way that ensures they are not too close to each other horizontally.
def generate_obstacle():  2 usages
    max_attempts = 10 # This limits the number of attempts when trying to place an obstacle.
    for _ in range(max_attempts):  # Try 10 times to find a valid position
        x = random.randint( a: 147, 753 - obstacle_img.get_width()) # Random horizontal position
        y = -obstacle_img.get_height() # Start above screen

        # Ensure new obstacle is not too close to existing ones
        too_close = any(abs(x - obs[0]) < min_horizontal_gap for obs in obstacles)
        # abs(x - obs[0]):  returns the absolute value, makes sure the value is positive so we know the correct horizontal distance between the new object and a previous one.
        # < min_horizontal_gap for obs in obstacles:  checks every x value of the obstacle to ensure the minimum horizontal gap is maintained
        if not too_close: # If spacing is okay, add the obstacle
            obstacles.append([x, y])
            break # Stop trying after finding a valid position
```

To control how obstacles appear on the screen, I created a function called generate_obstacle. This function randomly chooses a horizontal (X) position for each new obstacle, making sure it starts just above the visible area of the screen. To keep the game fair and not too difficult, I included a check to make sure new obstacles don't spawn too close to others already on screen. I did this using a min_horizontal_gap value, which helps maintain enough space between each obstacle. The function tries up to 10 times to find a good spot — if it finds one that's far enough from existing obstacles, it adds the new obstacle to the list and stops. This method ensures obstacles are spaced out well and makes the game more balanced and playable.



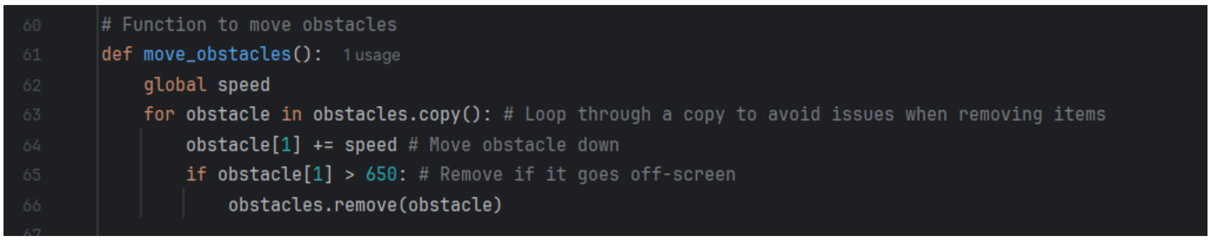
```python
# Function to draw obstacles
def draw_obstacles():  1 usage
    for obstacle in obstacles:
        screen.blit(obstacle_img,  dest: (obstacle[0], obstacle[1]))
```

The draw_obstacles() function is responsible for displaying each obstacle on the screen. It loops through the obstacles list and uses screen.blit() to draw the obstacle_img at the specified x and y coordinates of each obstacle. This ensures that all current obstacles are visually rendered at their correct positions in the game window, allowing the player to see and interact with them.



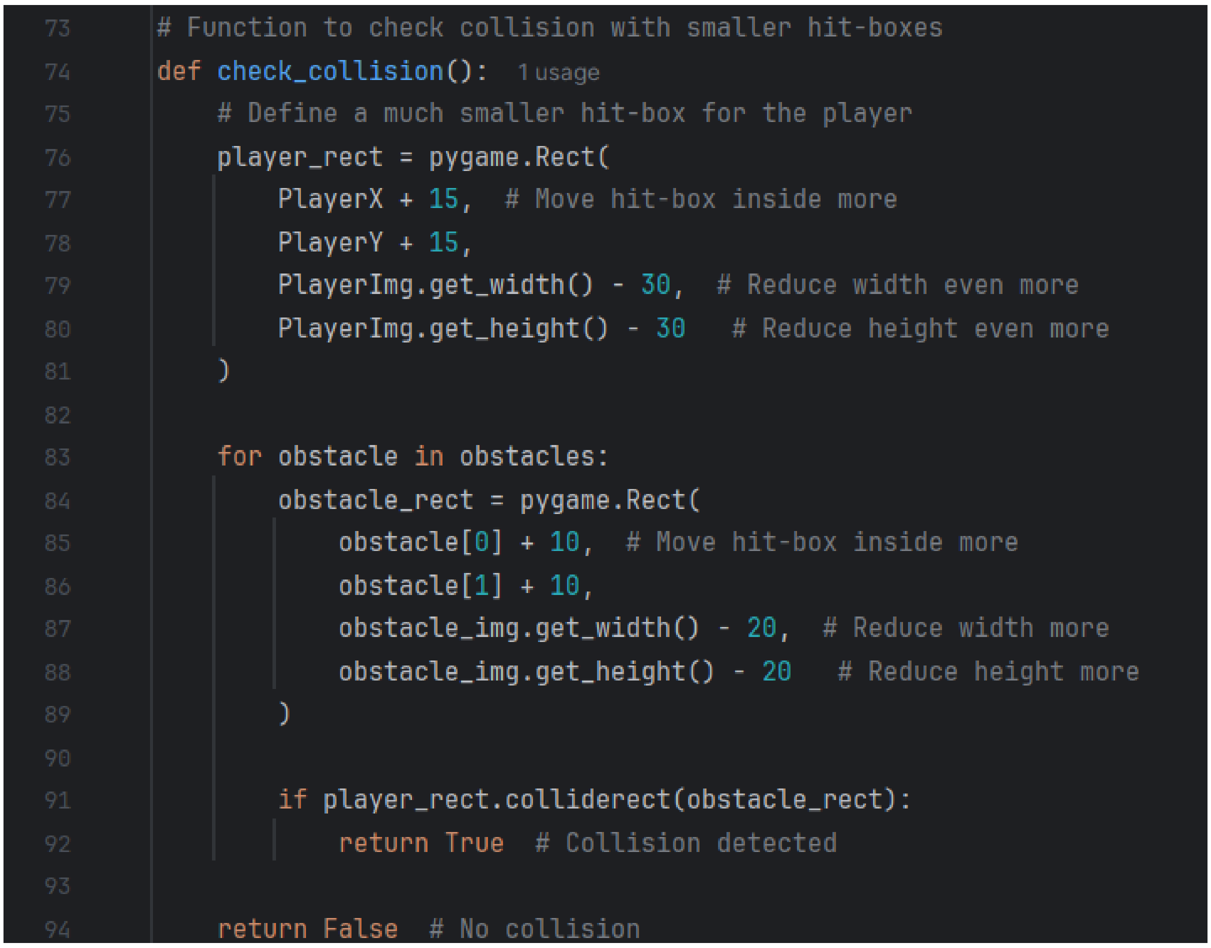
```python
    # Generate obstacles randomly with spacing
    if random.random() < 0.05:  # 5% chance per frame
        generate_obstacle()
        if random.random() < 0.25:  # 25% chance to spawn an extra obstacle
            generate_obstacle()
```

This section handles the random generation of obstacles during the game. On each frame, there is a 5% chance that a new obstacle will be created using the generate_obstacle() function. Additionally, there's a 25% chance to generate a second obstacle immediately after the first. This random approach adds variety and unpredictability to the game, making it more challenging and engaging for the player. The use of spacing and chance helps prevent the obstacles from becoming too predictable or overwhelming.



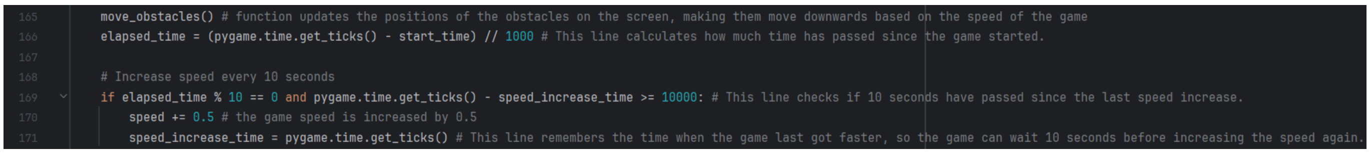
```python
# Function to move obstacles
def move_obstacles():  1 usage
    global speed
    for obstacle in obstacles.copy(): # Loop through a copy to avoid issues when removing items
        obstacle[1] += speed # Move obstacle down
        if obstacle[1] > 650: # Remove if it goes off-screen
            obstacles.remove(obstacle)
```

The move_obstacles() function moves obstacles down the screen by updating their vertical position using the global speed variable. It loops through a copy of the obstacles list to avoid issues when removing items during iteration. For each obstacle, it increases the y-coordinate by speed, moving the obstacle down. If an obstacle moves off the screen (y-coordinate greater than 650), it is removed from the list to prevent unnecessary tracking and improve performance. This ensures that only visible obstacles are kept in the game, maintaining both accuracy and efficiency.



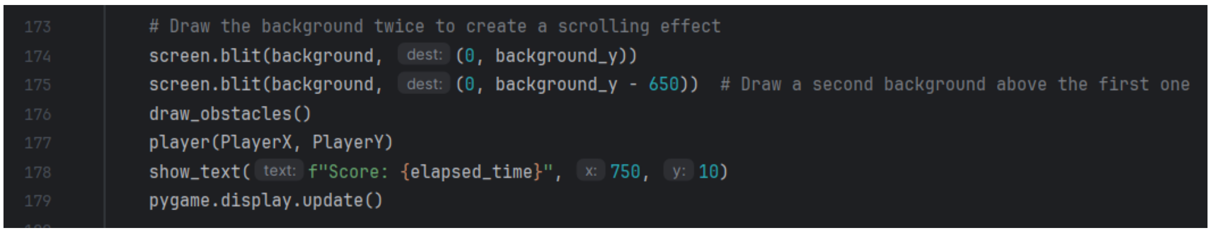
```python
# Function to check collision with smaller hit-boxes
def check_collision():  1 usage
    # Define a much smaller hit-box for the player
    player_rect = pygame.Rect(
        PlayerX + 15,  # Move hit-box inside more
        PlayerY + 15,
        PlayerImg.get_width() - 30,  # Reduce width even more
        PlayerImg.get_height() - 30   # Reduce height even more
    )

    for obstacle in obstacles:
        obstacle_rect = pygame.Rect(
            obstacle[0] + 10,  # Move hit-box inside more
            obstacle[1] + 10,
            obstacle_img.get_width() - 20,  # Reduce width more
            obstacle_img.get_height() - 20   # Reduce height more
        )

        if player_rect.colliderect(obstacle_rect):
            return True  # Collision detected

    return False  # No collision
```

The check_collision() function checks if the player has collided with any obstacles, using smaller hit-boxes to make the gameplay feel fairer. It first creates a reduced-size rectangle around the player by adding padding to the position and shrinking the width and height of the image. Then, it loops through each obstacle and creates a similarly reduced hit-box for them. By using colliderect(), it checks if the player's hit-box overlaps with any obstacle's hit-box. If a collision is detected, it returns True; otherwise, it returns False. This method avoids unfair collisions by focusing only on the central part of the images, improving the accuracy and playability of the game.



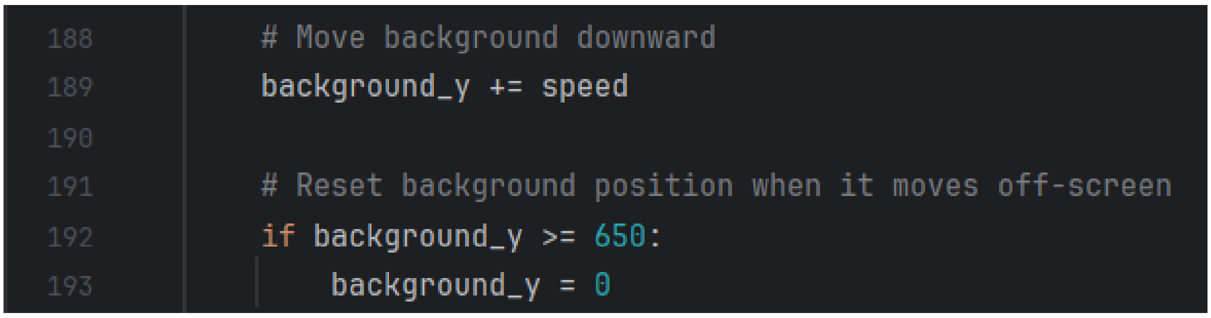
```python
    move_obstacles() # function updates the positions of the obstacles on the screen, making them move downwards based on the speed of the game
    elapsed_time = (pygame.time.get_ticks() - start_time) // 1000 # This line calculates how much time has passed since the game started.

    # Increase speed every 10 seconds
    if elapsed_time % 10 == 0 and pygame.time.get_ticks() - speed_increase_time >= 10000: # This line checks if 10 seconds have passed since the last speed increase.
        speed += 0.5 # the game speed is increased by 0.5
        speed_increase_time = pygame.time.get_ticks() # This line remembers the time when the game last got faster, so the game can wait 10 seconds before increasing the speed again.
```

This section of the code updates the game's difficulty over time. The move_obstacles() function is called to move all the obstacles downward based on the current game speed, making the game feel active and challenging. The elapsed_time variable calculates how many seconds have passed since the game started by subtracting start_time from the current time. To make the game progressively harder, the code checks if 10 seconds have passed since the last speed increase. If so, it raises the speed by 0.5 and updates the speed_increase_time to the current time. This gradual increase keeps the gameplay exciting and tests the player's skill as time goes on.



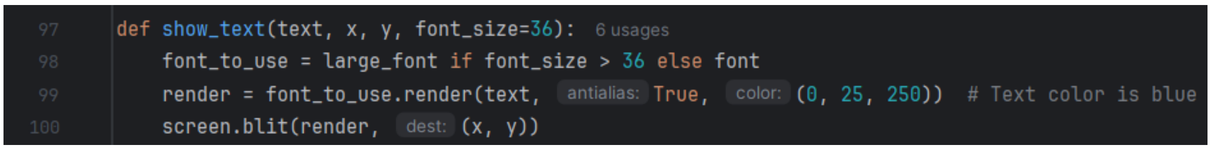
```python
    # Draw the background twice to create a scrolling effect
    screen.blit(background, dest: (0, background_y))
    screen.blit(background, dest: (0, background_y - 650))  # Draw a second background above the first one
    draw_obstacles()
    player(PlayerX, PlayerY)
    show_text( text: f"Score: {elapsed_time}",  x: 750,  y: 10)
    pygame.display.update()
```

This part of the code is responsible for drawing everything on the game screen each frame. The background is drawn twice — once at its current position and once just above it — to create a smooth vertical scrolling effect, giving the illusion of movement. After that, the draw_obstacles() function displays the current obstacles, and the player() function draws the player at the correct position. The show_text() function is used to display the score in the top-right corner, keeping the player informed of their progress. Finally, pygame.display.update() refreshes the screen so that all the new drawings appear in the game window.



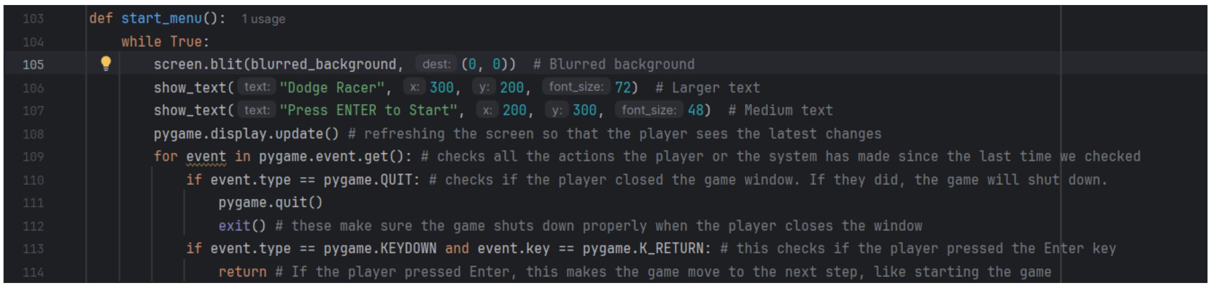
```python
    # Move background downward
    background_y += speed

    # Reset background position when it moves off-screen
    if background_y >= 650:
        background_y = 0
```

This section of the code handles the movement of the background to create a scrolling effect. The background's vertical position (background_y) is increased by the speed value, making it move downward on the screen. When the background moves off the bottom of the screen (i.e., when background_y reaches or exceeds 650 pixels), it resets the position back to the top (background_y = 0). This creates a continuous loop of scrolling, making the game environment feel dynamic and immersive.



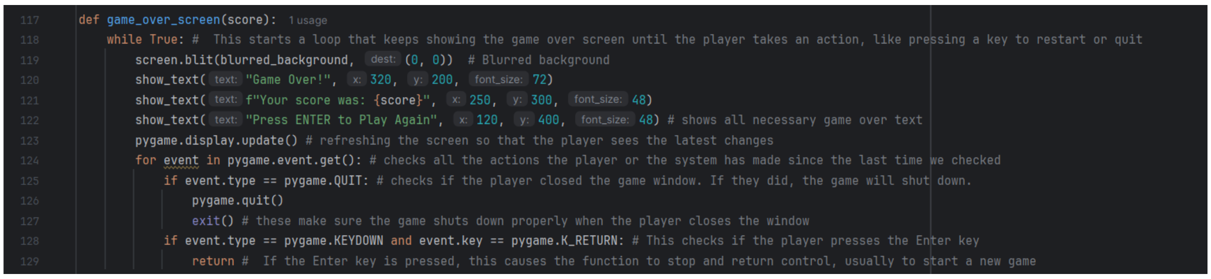
```
def show_text(text, x, y, font_size=36):  6 usages
    font_to_use = large_font if font_size > 36 else font
    render = font_to_use.render(text,  antialias: True,  color: (0, 25, 250))  # Text color is blue
    screen.blit(render,  dest: (x, y))
```

The show_text() function displays text on the screen at a specified position. It takes the text, x and y coordinates, and an optional font size. The text is rendered in a blue color (0, 25, 250) and then drawn on the screen using blit(). This function is useful for showing important information like the score or instructions to the player in a clear and readable way.



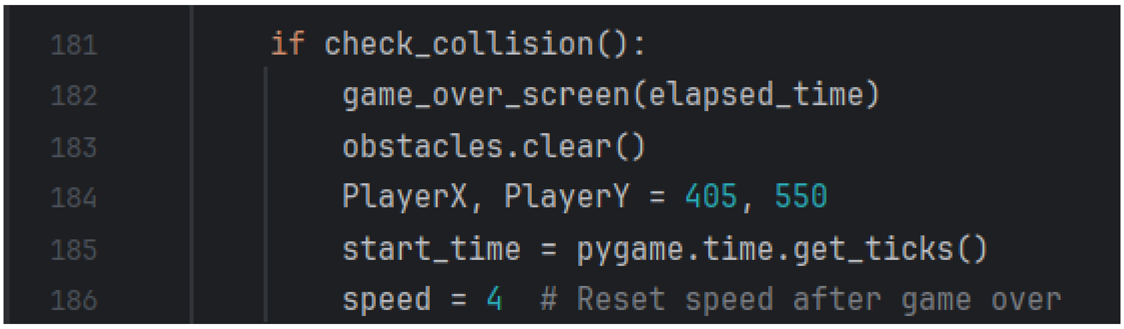
```
def start_menu():  1 usage
    while True:
        screen.blit(blurred_background,  dest: (0, 0))  # Blurred background
        show_text( text: "Dodge Racer",  x: 300,  y: 200,  font_size: 72)  # Larger text
        show_text( text: "Press ENTER to Start",  x: 200,  y: 300,  font_size: 48)  # Medium text
        pygame.display.update() # refreshing the screen so that the player sees the latest changes
        for event in pygame.event.get(): # checks all the actions the player or the system has made since the last time we checked
            if event.type == pygame.QUIT: # checks if the player closed the game window. If they did, the game will shut down.
                pygame.quit()
                exit() # these make sure the game shuts down properly when the player closes the window
            if event.type == pygame.KEYDOWN and event.key == pygame.K_RETURN: # this checks if the player pressed the Enter key
                return # If the player pressed Enter, this makes the game move to the next step, like starting the game
```

The start_menu() function creates the game's starting screen, displaying a blurred background along with the game title and instructions to press ENTER to start. It continuously updates the screen using pygame.display.update() so that the player sees the latest visuals. It listens for events like closing the window or pressing a key. If the player closes the game window, the function shuts the game down properly using pygame.quit() and exit(). If the player presses the ENTER key, the function returns, allowing the game to start. This start menu gives the player a clear and smooth introduction before gameplay begins.



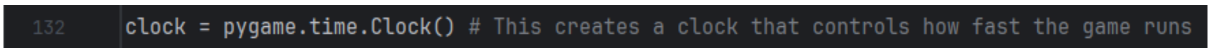
```
def game_over_screen(score):  1 usage
    while True: #  This starts a loop that keeps showing the game over screen until the player takes an action, like pressing a key to restart or quit
        screen.blit(blurred_background,  dest: (0, 0))  # Blurred background
        show_text( text: "Game Over!",  x: 320,  y: 200,  font_size: 72)
        show_text( text: f"Your score was: {score}",  x: 250,  y: 300,  font_size: 48)
        show_text( text: "Press ENTER to Play Again",  x: 120,  y: 400,  font_size: 48) # shows all necessary game over text
        pygame.display.update() # refreshing the screen so that the player sees the latest changes
        for event in pygame.event.get(): # checks all the actions the player or the system has made since the last time we checked
            if event.type == pygame.QUIT: # checks if the player closed the game window. If they did, the game will shut down.
                pygame.quit()
                exit() # these make sure the game shuts down properly when the player closes the window
            if event.type == pygame.KEYDOWN and event.key == pygame.K_RETURN: # This checks if the player presses the Enter key
                return #  If the Enter key is pressed, this causes the function to stop and return control, usually to start a new game
```

The game_over_screen() function displays a game over screen when the player loses. It enters a loop that keeps the screen visible until the player either presses ENTER to restart or closes the game. It shows a blurred background and displays the message “Game Over!”, the player’s final score, and instructions to play again. The screen is refreshed continuously using pygame.display.update() so that any changes are visible. It listens for player actions, and if the window is closed, it safely exits the game. If the ENTER key is pressed, the function ends, allowing the game to restart. This screen gives the player a clear ending point and a smooth way to start over.



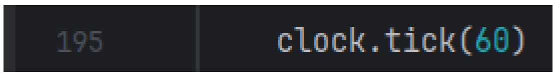
```
    if check_collision():
        game_over_screen(elapsed_time)
        obstacles.clear()
        PlayerX, PlayerY = 405, 550
        start_time = pygame.time.get_ticks()
        speed = 4  # Reset speed after game over
```

This section checks if a collision occurs by calling the check_collision() function. If a collision is detected, it triggers the game over screen using the game_over_screen() function, passing the player's score (elapsed time) as an argument. It then clears all obstacles from the screen with obstacles.clear() and resets the player’s position to the starting coordinates (PlayerX, PlayerY = 405, 550). The game’s timer is reset by storing the current time in start_time, and the speed is set back to its initial value of 4. This ensures that after a game over, the game state is properly reset for a fresh start.



```
clock = pygame.time.Clock() # This creates a clock that controls how fast the game runs
```

The line clock = pygame.time.Clock() creates a clock object that helps control the speed of the game. It allows the developer to manage the frame rate by specifying how many frames per second the game should run, typically using clock.tick(). This ensures the game runs smoothly and consistently across different devices, preventing it from running too fast or too slow depending on the computer's performance. It's an essential part of maintaining stable and predictable game timing.

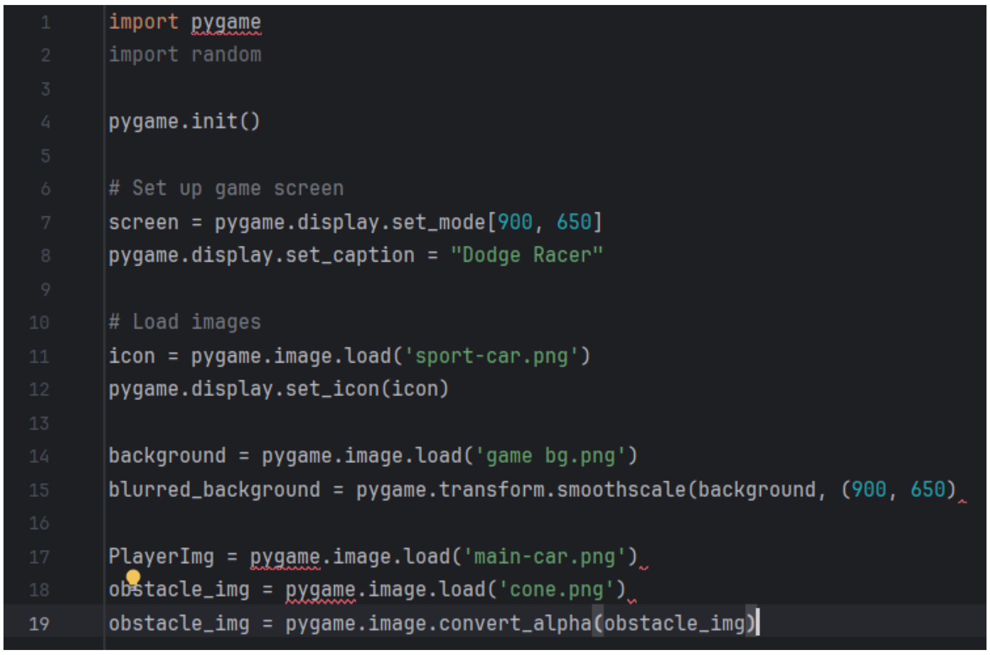


```
    clock.tick(60)
```

The clock.tick(60) function is used to control the frame rate of the game. It ensures that the game runs at a consistent speed by limiting the number of frames per second (FPS) to 60. This means the game will update 60 times per second, which helps maintain smooth gameplay and prevents the game from running too fast or too slow on different systems. By setting the FPS to 60, it also helps manage the game's performance and responsiveness.

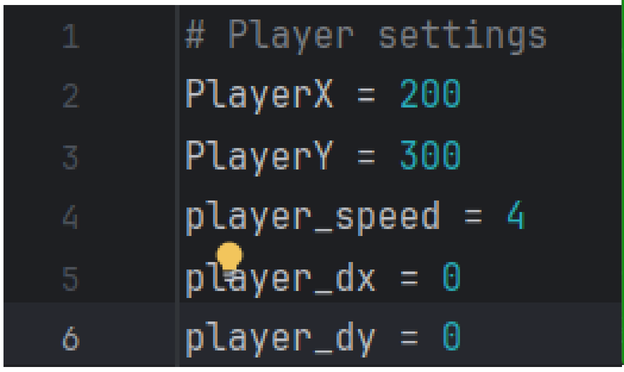
TESTING TO INFORM DEVELOPMENT

PROVIDE EVIDENCE OF TESTING AT EACH STAGE

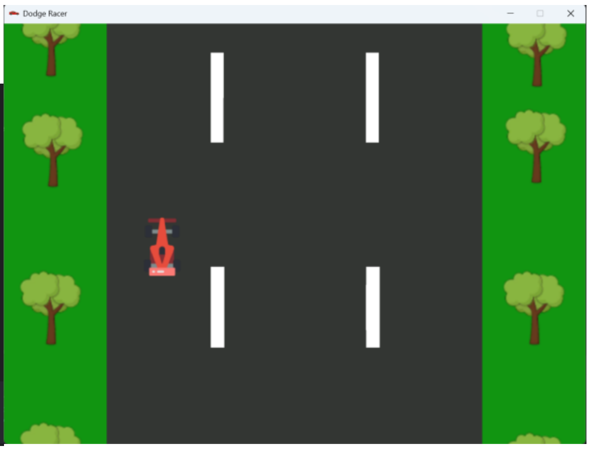


```
import pygame
import random

pygame.init()

# Set up game screen
screen = pygame.display.set_mode[900, 650]
pygame.display.set_caption = "Dodge Racer"

# Load images
icon = pygame.image.load('sport-car.png')
pygame.display.set_icon(icon)

background = pygame.image.load('game bg.png')
blurred_background = pygame.transform.smoothscale(background, (900, 650)

PlayerImg = pygame.image.load('main-car.png')
obstacle_img = pygame.image.load('cone.png')
obstacle_img = pygame.image.convert_alpha(obstacle_img)
```

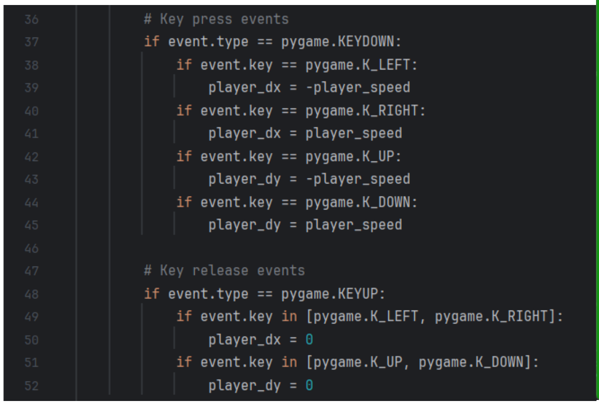
The first time I tried to write the setup code for my game, I made a few errors in syntax that made the program fail instantaneously. How I tried to set the screen size was one of the main problems. In the line pygame.display.set_mode[900, 650], I should have used brackets instead of square brackets. Python wants function calls to be made with brackets, not square brackets, which are used for indexing but not calling functions. Another problem was how I set the title of the game window. I used an assignment line instead of the set_caption() function by accident. I did, pygame.display.set_caption = "Dodge Racer", but his doesn't run the code; it just changes the name of set_caption to a string, which stops it from working at all. Also, the line where I used smooth scale to change the size of the background had a syntax mistake. The Python interpreter gave me a syntax error because I forgot to close the brackets at the end of the line. This stopped the code from running at all. Lastly, I used the convert_alpha() method in the wrong way. I wrote pygame.image.convert_alpha(obstacle_img) because I thought it would take an image as an option. while the whole time , you should call the convert_alpha() method on an image object, like this: obstacle_img.convert_alpha(). Python gave an error message saying that the pygame.image package doesn't have a function called convert_alpha() because of this mistake.



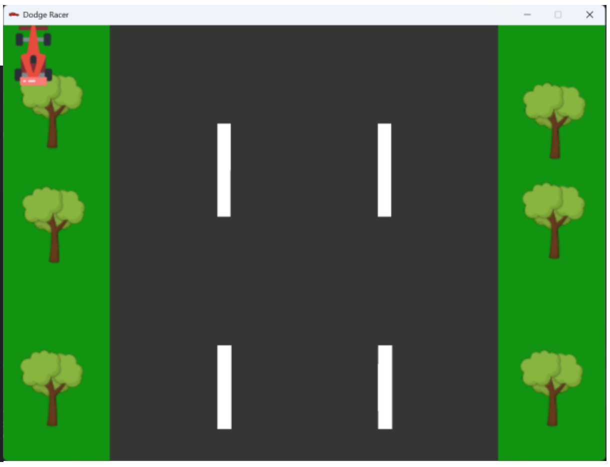
```python
# Player settings
PlayerX = 200
PlayerY = 300
player_speed = 4
player_dx = 0
player_dy = 0
```



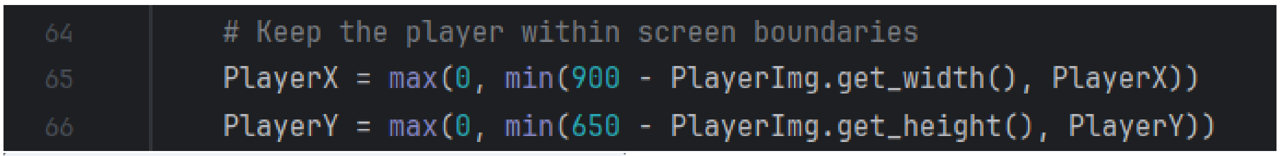
I quickly realised that (200, 300) is not the best place to start the player's car because it puts it too far to the left and maybe too high on the screen. Most racing or dodging games are played from above, with the player's car starting in the middle or near the bottom. This doesn't give the player enough room or sight to respond to obstacles coming at them. It makes more sense to start closer to the middle horizontally and lower down vertically, like PlayerX = 450 and PlayerY = 550 on a 900x650 screen. This way, you can see what's coming and it feels more natural to play.



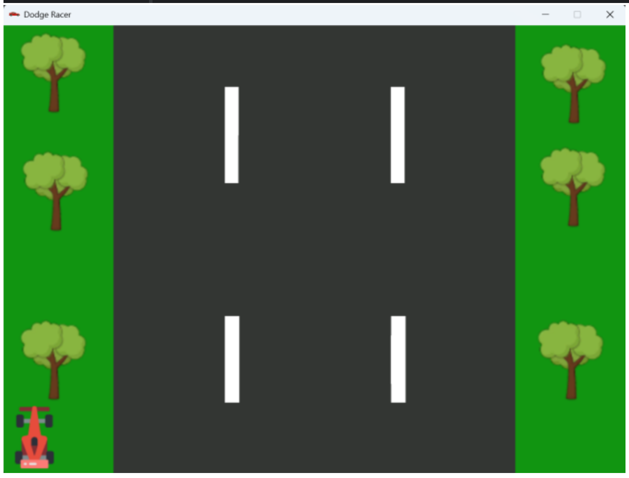
```python
        # Key press events
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                player_dx = -player_speed
            if event.key == pygame.K_RIGHT:
                player_dx = player_speed
            if event.key == pygame.K_UP:
                player_dy = -player_speed
            if event.key == pygame.K_DOWN:
                player_dy = player_speed

        # Key release events
        if event.type == pygame.KEYUP:
            if event.key in [pygame.K_LEFT, pygame.K_RIGHT]:
                player_dx = 0
            if event.key in [pygame.K_UP, pygame.K_DOWN]:
                player_dy = 0
```



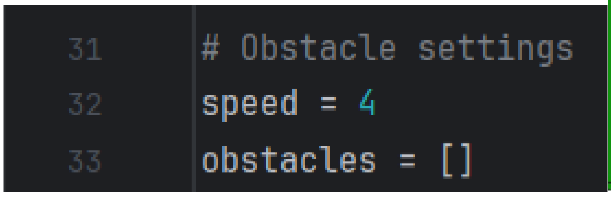
The first method using KEYDOWN and KEYUP events updates the movement only when a key is pressed or released. This can cause issues if multiple keys are pressed quickly or if a key is held down for a long time, as the game may miss key events or experience delays. This results in choppy or unresponsive movement since the car's direction only updates when the key events occur. In contrast, the second method with pygame.key.get_pressed() checks the state of all keys every frame. This method continuously updates the car's movement as long as the key is held down, making the movement smooth and responsive. Since it doesn't rely on event detection, it ensures more consistent control, even when multiple keys are pressed or a key is held down for a longer period. This approach leads to smoother and more precise movement.



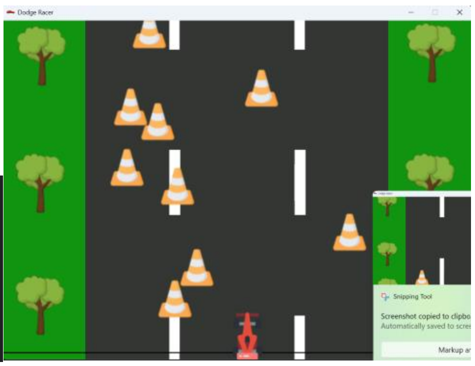
```
    # Keep the player within screen boundaries
    PlayerX = max(0, min(900 - PlayerImg.get_width(), PlayerX))
    PlayerY = max(0, min(650 - PlayerImg.get_height(), PlayerY))
```



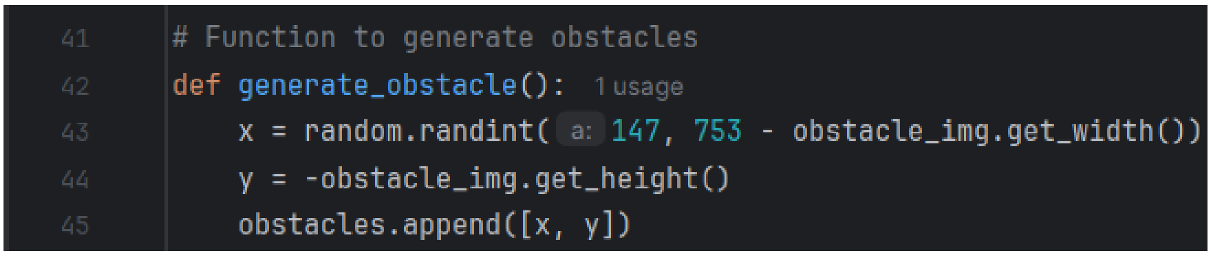
In my first plan for boundary enforcement, the method wasn't working well because it didn't take into account how the road and sidewalks were built or how big the icons really were. Since the player icon is slightly bigger than expected, the fixed screen boundaries were causing the icon look like the wheels were far from even touching the pavement. The car would just fly off the road when playing because the hardcoded numbers for the right and bottom boundaries (900 and 650) didn't take into account the size of the background I made in photo editing software. To fix this, I changed the code to take this into account, which made the placement in the road area easier and more accurate. By dynamically changing the edges, I made sure that the icon would fit better within the layout and look like it was lined up with the sidewalk, which fixed the problem where the wheels seemed to float and barely touch the edge. This fix makes sure that the visual experience is more accurate and smooth.



```
# Obstacle settings
speed = 4
obstacles = []
```

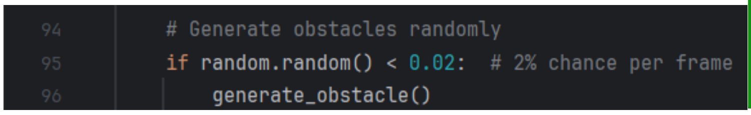


My first idea wasn't really the best because it didn't take into account how hurdles should be placed and handled in the game. It set the speed and made a list of empty spaces for hurdles, but it was missing important parts that would have made the game great. If you don't think about a sufficient horizontal distance between obstacles, the game could end up with obstacles that are too close to each other, making it too hard or unfair for the player. The game is now more fair and better because I added the min_horizontal_gap. This will keep hurdles from being too close to each other, making it easier for the player to get through them. This makes the game more fair by giving players enough room to move around while keeping the level of difficulty at a good level.

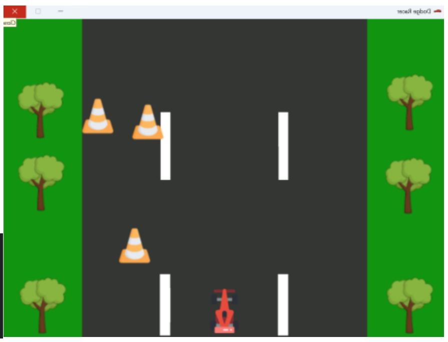


```
# Function to generate obstacles
def generate_obstacle():  1 usage
    x = random.randint( a: 147, 753 - obstacle_img.get_width())
    y = -obstacle_img.get_height()
    obstacles.append([x, y])
```

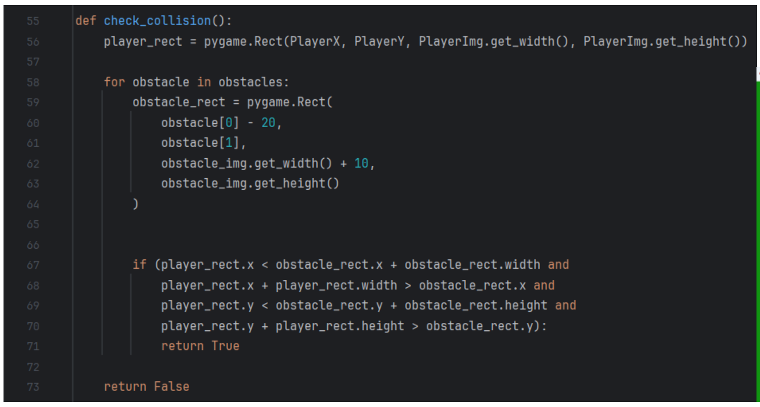
It became clear to me when I looked at the initial draft of my generate_obstacle() code that the way I placed obstacles was too simple and didn't support fair gameplay. The function picked an x-coordinate between 147 and 753, minus the width of the obstacle picture, to put the obstacle just above the screen and add it to the list without checking it first. In this case, obstacles could easily appear next to each other with no space between them, making it impossible for the player to avoid them. Also, because there was no control over where the obstacles were placed, the game felt fast and rough. I fixed this by adding a system that checks for the right amount of space between obstacles before adding a new one. In the new version, I used a for loop to try up to ten times to find a valid point and added a max_attempts limit to stop the loop from running forever. I also put in a condition that compares the x-values of any existing obstacles to the new one to see how far apart they are horizontally. The obstacle would not be added if the new position was too close to an existing one, or if the horizontal gap was less than the number I set with min_horizontal_gap. The new obstacle wouldn't be added to the list until a valid spot with enough room was found. It changed the way the game felt a lot because it stopped unfair spawns and gave players enough room to get around barriers. Also, the game felt better and more consistent because the placement now followed rules that took both difficulty and justice into account.



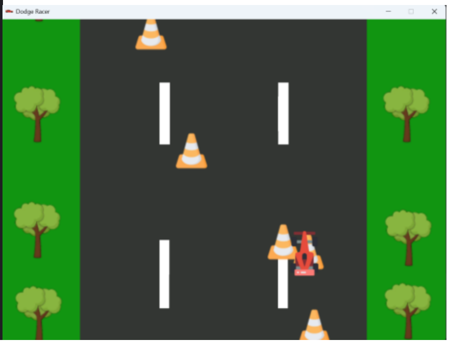
```
# Generate obstacles randomly
if random.random() < 0.02:  # 2% chance per frame
    generate_obstacle()
```



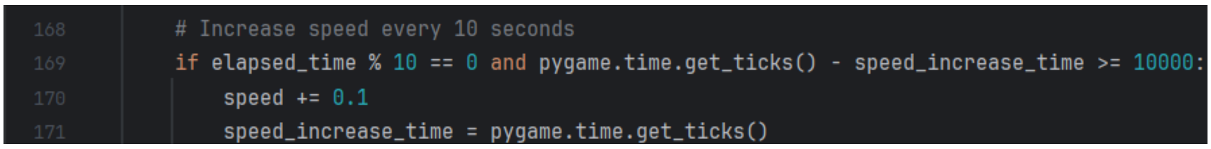
I recognised that I had written my random obstacle spawning code in a very simple and limited way when I looked back at an older version of it. It only gave me a 2% chance per frame to create an obstacle when I used - if random.random() < 0.02:. This worked in theory, but it made the game feel slow and inconsistent . At times, obstacles wouldn't show up for a long time and other times, they'd come out of nowhere, with no clear pattern. The game lacked energy and challenge because hurdles didn't appear very often and there was no way to change how many could show up at once. It also didn't have the fast-paced feel I was going for. After some time, I realised I could make this better by slightly raising the base spawn rate and adding some controlled randomness to make the game more interesting. I changed it in the new version to - if random.random() < 0.05: This made it 5% more likely that an obstacle would show up on any given frame. But adding that extra check inside that condition, if random.random() 0.25, made the difference though. This made it 25% more likely that a second obstacle would appear in the same frame. It was very helpful to have this small addition because it changed the game so that sometimes only one obstacle would show up and other times two would show up close together. With that extra bit of chance, the spawning felt more natural and kept the player on their toes without being too much. It also made the game more fair and harder at the same time, since I still had control over the general odds but left enough room for variation so that each playthrough felt a little different. This small change had a big effect on how fast the game went and how good it was altogether.



```python
def check_collision():
    player_rect = pygame.Rect(PlayerX, PlayerY, PlayerImg.get_width(), PlayerImg.get_height())

    for obstacle in obstacles:
        obstacle_rect = pygame.Rect(
            obstacle[0] - 20,
            obstacle[1],
            obstacle_img.get_width() + 10,
            obstacle_img.get_height()
        )


        if (player_rect.x < obstacle_rect.x + obstacle_rect.width and
            player_rect.x + player_rect.width > obstacle_rect.x and
            player_rect.y < obstacle_rect.y + obstacle_rect.height and
            player_rect.y + player_rect.height > obstacle_rect.y):
            return True

    return False
```

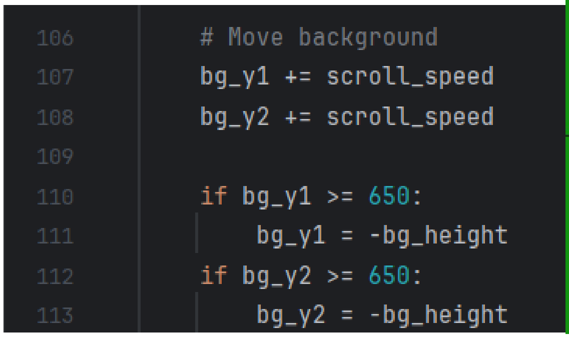


looking back at the collision code I used earlier in my project, I realised it had a lot of problems that made the game feel unfair and buggy. I was using the pygame.Rect function to store and manipulate rectangular areas for both the player and obstacles, which was a step in the right direction, but I failed to shrink the hitboxes. The player’s hitbox was the exact size of the image, and the obstacle’s hitbox was even bigger than the image, which led to unfair and overly sensitive collisions. I also accidentally moved the obstacle’s hitbox to the side a bit, while trying to centre everything, which made it even more out of line with the visuals. Instead of using the proper colliderect() function that Pygame provides to determine if two rectangular shapes (represented by Pygame.Rect objects) are overlapping, I tried to write my own way of checking if rectangles were overlapping, which was a lot more complicated and harder to get right. This made the collisions feel random and sometimes frustrating to the player. Finally, I also forget to add the consequences for when a collision occurs, so instead of the game ending, collisions did nothing. Later on, I fixed this by using smaller rectangles that sit nicely inside the car and obstacle images, so collisions only happen when it really looks like they should. I also used colliderect() to make the code simpler and more accurate. This made the game feel fairer and more enjoyable to play.

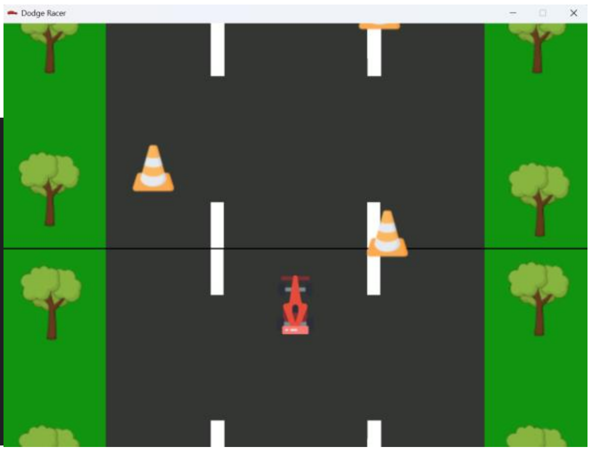


```
    # Increase speed every 10 seconds
    if elapsed_time % 10 == 0 and pygame.time.get_ticks() - speed_increase_time >= 10000:
        speed += 0.1
        speed_increase_time = pygame.time.get_ticks()
```

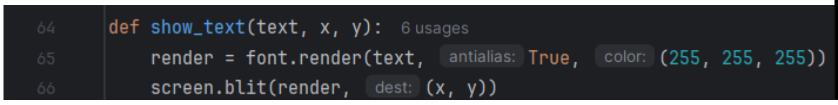
When I first tried to increase the game's speed over time, I used a very small increment of 0.1. At first, this seemed like a safe and controlled way to slowly increase difficulty, but in practice, it didn't make much of a difference during gameplay. The change was so small that players barely noticed the speed increasing at all, even after a long time. This caused the game to feel boring and too easy for too long, which made it harder to keep the player engaged or challenged. I later changed the increment to 0.5, which created a more noticeable and effective difficulty curve. With this adjustment, the speed increase felt meaningful after each time interval, helping the game gradually become more intense without feeling too sudden or unfair. This made the gameplay more dynamic and rewarding, while still giving the player time to adjust to the faster pace.



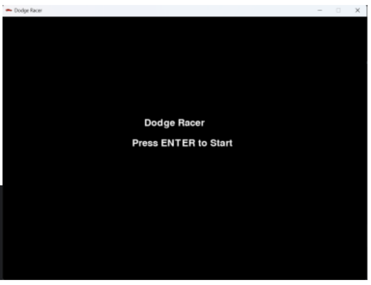
```
    # Move background
    bg_y1 += scroll_speed
    bg_y2 += scroll_speed

    if bg_y1 >= 650:
        bg_y1 = -bg_height
    if bg_y2 >= 650:
        bg_y2 = -bg_height
```



When I first tried to make the background scroll, I only used one picture and tried to move it down, resetting its place when it went off the screen. At first, this seemed to work, but when the background changed, there was a visible jump or flicker that stopped the smooth scrolling effect I was going for. The movement felt off because there was a visible break between when the background went away and when it came back. This could be distracting for the player. I learnt later that I needed to draw the background picture twice, once just below the screen and once just above it, in order to make a real endless scrolling background. I used two vertical positions that changed together as the screen scrolled instead of resetting the position of a single picture. This meant that when one picture went off the screen, the next one was already there to keep the motion going smoothly. By going back and forth between them and setting them to the right time, I was able to make a loop that played smoothly without any stuttering or breaks. This made the motion feel smoother and more professional.

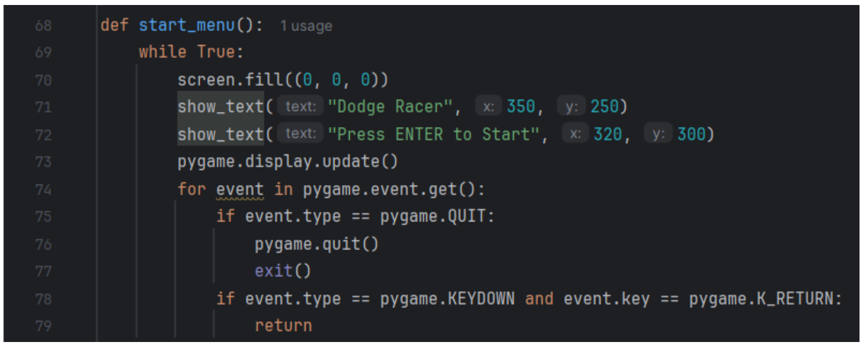


```
def show_text(text, x, y):  6 usages
    render = font.render(text,  antialias: True,  color: (255, 255, 255))
    screen.blit(render,  dest: (x, y))
```



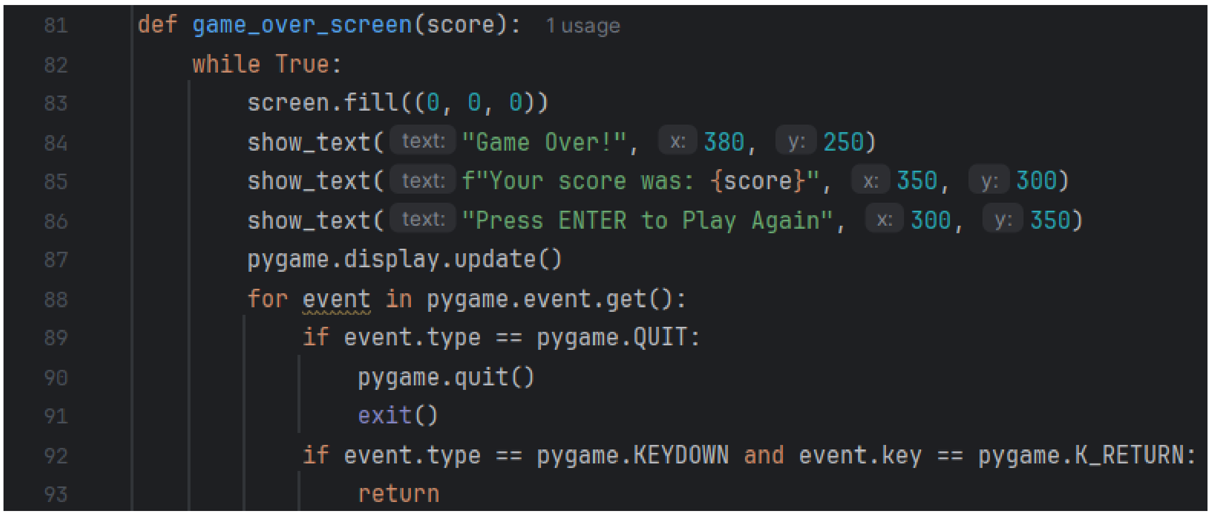
Dodge Racer
Press ENTER to Start

I used a fixed font and plain white text for everything on the screen when I first wrote the show_text software. While it technically displayed the text, the result didn't look very good . In bright areas, and also places with a lot of white, the white text often clashed with the background, making it hard to read. It also didn't fit the style of the game and looked pretty simple. Plus, the writing didn't change fonts, so it looked the same whether it was the score, the title, or the messages. This made the screen look dull and disorganised. Later, I made this better by changing the text colour to a blue shade that went better with the background and the style of the game as a whole. I also included a font_size option so I could change the text's size based on its use. For instance, I made the fonts bigger for the names and smaller for the game stats. To keep the look constant and make it easier to read, I even switched between different font objects based on the size. These changes made a big difference in how the game felt overall. They made the interface clearer and more polished and fixed the problems with style and visibility in my older version.



```
def start_menu():  1 usage
    while True:
        screen.fill((0, 0, 0))
        show_text( text: "Dodge Racer",  x: 350,  y: 250)
        show_text( text: "Press ENTER to Start",  x: 320,  y: 300)
        pygame.display.update()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()
            if event.type == pygame.KEYDOWN and event.key == pygame.K_RETURN:
                return
```

When I initially created the start_menu function, I used a very simple approach to display the text and trigger the start of the game. I just filled the screen with a solid black colour, displayed the game title and "Press ENTER to Start" in the centre, and checked for the ENTER key to move on. While this approach worked, it felt very plain and lacked any visual appeal. The black background was stark and didn't do much to engage the player right from the start. The text was also a bit too simple, with no differentiation in size to give the game more visual hierarchy. I then realised that adding some stylistic improvements could greatly enhance the player's experience. So, I switched to using a blurred background image to give the screen a more dynamic and polished look. I also increased the font size for the title to make it stand out more, and adjusted the size of the "Press ENTER to Start" text to create a better visual balance. These changes not only made the menu look more professional, but they also helped set the tone for the game and made the title feel more important. The function became much more visually appealing and created a better first impression for the player.



```
def game_over_screen(score):  1 usage
    while True:
        screen.fill((0, 0, 0))
        show_text( text: "Game Over!",  x: 380,  y: 250)
        show_text( text: f"Your score was: {score}",  x: 350,  y: 300)
        show_text( text: "Press ENTER to Play Again",  x: 300,  y: 350)
        pygame.display.update()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()
            if event.type == pygame.KEYDOWN and event.key == pygame.K_RETURN:
                return
```

In my initial game_over_screen function, I followed a straightforward approach. I filled the screen with a black background and displayed basic text messages informing the player of their game over status and score. Even though this worked, this design was quite dull and lacked any engagement. Along with the simple black background, the text sizes were all the same, which made the screen feel a bit unbalanced. There was also no sense of style or atmosphere to reflect the game's tone at that point. Later, I realized that a more visually interesting game over screen could elevate the player's experience. To achieve this, I added a blurred background, which gave the screen a dynamic, slightly faded look and helped the text stand out more. I also increased the size of the "Game Over!" text and adjusted the size of the score and "Press ENTER to Play Again" text to create a clearer visual hierarchy. This way, the most important information was more prominent. The added background and varied font sizes made the screen feel much more polished and engaging. These changes contributed to a much smoother and more visually appealing transition to the end of the game, giving it a more professional and immersive finish.

EVALUATION

TESTING TO INFORM EVALUATION

USER FEEDBACK

How often do you, or would you consider, coming back to play this game?

Daily	
Weekly	\\\
Occasionally	\
Never	\

How would you rate the difficulty of this game?

Too Hard	
Hard	\
Just Right	\\\
Easy	\
Too Easy	

How satisfied are you with the graphics and visual design?

Very Satisfied	\\
Satisfied	\\\
Neutral	
Dissatisfied	
Very Dissatisfied	

How would you rate the ease of controlling your player's omnidirectional movement?

Too Hard	
Hard	\
Neutral	\
Easy	\\\
Too Easy	

How would you rate the random placement of the obstacles across the road?

Too Hard	
Just Right	\\\\\
Too Easy	

How engaging was the dynamic difficulty scaling (obstacles get faster as you play)?

Very Engaging	\\\\\\\\
Somewhat Engaging	\\
Not Engaging	

How would you rate the accuracy of the collision detection?

Very Accurate	\\\\\\\\\\
Neutral	
Not Accurate	

Did the scrolling background give you a sense of movement and immersion?

Yes, it felt smooth	\\\\\\\\\\
Neutral	
No, it was distracting	

How easy was it to understand the controls and rules of the game?

Too Difficult	
Difficult	
Neutral	\\
Easy	\\\\\\\\
Too Easy	

What aspect of the game did you find most enjoyable?

The Controls	\
The Difficulty	
The Graphics	\\
The Score System	
The Obstacles	\\

Do you think the game could use more features or challenges?

Yes	\\\\\
No	

How would you rate the game overall?

Excellent	\\
Good	\\\
Neutral	
Poor	
Very Poor	

Would you recommend this game to others?

Yes	\\\\\
No	

GAMEPLAY METRICS

Average Score per Run	63
Average Number of Rounds Played per Session	4
Average Session Length	4.2 minutes
Player Retention Rate	50%

PREFORMANCE DATA

Platform	Frame Rate (FPS)	Average Load Time	Crash Occurrences	Performance Issues	Resolution
PC (High-End)	60 FPS	0.5 seconds	None	Smooth performance at 60 FPS with high graphics settings.	N/A
PC (Mid-Range)	50-60 FPS	1.5 seconds	None	Smooth performance at 50-60 FPS on medium settings.	Adjusted graphics settings for smoother play.
PC (Low-End)	30-45 FPS	3 seconds	Occasional crashes	Lower frame rate and occasional stuttering in intense scenes.	Reduced texture quality and resolution.

BUG REPORT AND ERROR LOGS

Issue	Severity	Status	Player Impact
Minor delay in collision detection after quick player movement	Low	Resolved	Rare, minimal impact on gameplay.
Occasional frame stutter during background scrolling on older systems	Medium	Resolved	Minor drop in performance, but not noticeable to most players.
Score not updating correctly during the final seconds of gameplay	Low	Resolved	Affects final score display, but not gameplay.
Obstacles spawning slightly off-screen in rare cases	Low	Resolved	Minimal visual glitch, does not affect gameplay.
Unresponsive controls on certain keyboards in rare cases	Medium	Resolved	Occasional frustration, does not affect gameplay significantly.

DATA ANALYSIS REPORT

The majority of user feedback about the game has been good, with many players saying they liked how it was designed and how it worked. The graphics and visual style were one of the things that people liked most about the game. Many players liked how the background sliding made them feel like they were moving and immersed them in the game, which made the experience better. People also liked how the visual cues for obstacles helped players act quickly and get through the game easily. The way the game looks and feels as a whole seems appealing and many players said they were happy with the graphics and visuals. The omnidirectional movement controls were another good thing because most users said it was simple and easy to control how the player moved, feeling smooth and responsive. The score system was liked by everyone; players liked that it gave them instant feedback on how they were doing. This encourages them to keep playing and try to get better scores. The dynamic challenge scaling also made the game more fun to play, as many people found it interesting and challenging that the obstacles got faster over time, especially skilled players. It kept the game interesting and pushed players to their limits, which was a big part of keeping them interested over long rounds.

Even with these good points, feedback also showed places where things could be better. Difficulty balance was one of the main concerns, as some players felt that the game was either too hard or too easy, depending on their skill level. The dynamic difficulty progression was liked, but it could be improved to make the task rise more gradually and in a way that fit the player better. This would keep players of all skill levels interested without letting them get upset or bored. Additionally, players reported that the game became somewhat repetitive after a few rounds. Randomisation made each playthrough somewhat different, but the hurdles and game mechanics could use more variety to keep people interested for longer. It might help to keep the game interesting by adding new types of obstacles, power-ups, or changing the surroundings. Some users also said that the game could use more progress feedback, like sound effects, animations, or visual signs that show players what they've done well, like avoiding obstacles or reaching certain goals. This would make the experience more satisfying by giving people real results for the things they do. Some players also had small problems with how well the controls worked, especially when moving quickly. Most users found the settings easy to use, but some did notice a small lag when things were moving quickly. It will be important to fine-tune the control system so that it stays smooth and quick at all speeds. Lastly, a number of users said that the game could be made more fun to play again and again. Some players found that the game became less fun after a few plays because the content didn't change. To fix this, I'm going to add new features that players can unlock, like new cars, more obstacles, or an upgrade system. This will keep the game interesting and give players more reasons to come back over time.

The data from tracking bugs after the game came out shows that all problems were fixed, and none of them had a big effect on how it was played. Most of the problems were small, like a slight delay in collision recognition and some visual glitches, like obstacles appearing off-screen. There were a few moderately bad problems, like frame stuttering on older systems and controls that wouldn't work on some computers, but they were fixed and didn't really affect the experience for most players. The game was mostly stable and playable, with only a few small issues mentioned here and there.

The game works well on both high-end and mid-range PCs, as shown by the stable frame rates and quick load times. Some crashes and lower FPS happen on low-end PCs, but these problems could be fixed by lowering the quality of the textures. If this keeps up, adding more tweaks like a "low-spec" mode might make it easier for people with less powerful systems to play. You can use these ideas again if the game ever comes out on other devices. Long-term stability will be ensured by ongoing speed monitoring, which will also help support new features and updates.

In conclusion, the comments shows that the game is well-built, with fun mechanics, nice graphics, and a satisfying scoring system. But the level of difficulty, the range of obstacles, the responsiveness of the controls, and the number of times you can play the game could all be better. By making these changes, I can make the experience more polished and interesting for a wider group of players, which will keep them coming back for more. The comments you give will be very helpful in making future changes to the game that will make it more fun and last longer.

EVALUATION OF SOLUTION

TEST EVIDENCE AGAINST SUCCES CRITERIA

1. Game Length That Keeps You Interest
This success criteria was met in part. According to what players said, the game was fun and kept them interested for a few minutes at a time, which is the right amount of time for casual players (3–5 minutes). As the game got harder, skilled players were able to stay in it even longer, which made them want to keep playing. But some players said that the game felt like it was being played over and over again after a while. This means that the length of the game was usually good, but the lack of different kinds of content made it less fun to play for a long time. To fully meet this requirement, the game could have more content that can be unlocked or new tasks that will make people want to play it again. To fix this in future versions, I could add more material to unlock, like new vehicles, levels, or rewards. These plans would help players reach their targets and keep them playing for longer.

2. System for keeping accurate score
This requirement was fully met. The players always said that the scoring system worked well and let them know right away how they were doing. During play, the score was shown clearly, and at the end of the session, the total score was shown. The game also reset correctly every time it was played again. People didn't report any bugs or problems in this area, which shows that the score system worked as it should have. The real-time changes made players want to beat their old scores, which made the game more fun and satisfying overall.

3. Dynamic Scaling of Difficulty
This success condition was met in part. The game was harder by making obstacles move faster over time, which many players liked because it made the game more difficult. Skilled players said it made the game more fun and exciting to play for longer. Some users, though, thought it got too easy too quickly or too hard for too long. This shows that the function did work, but the rate at which the difficulty level rose could have been better. The game could have adaptive difficulty that changes based on how well players do, or players could be able to pick the amount of difficulty they want to play at the beginning. I could make this better in the future by adding adaptive challenge that changes based on how well or how skilled the player is. Gamers could also pick a level of effort (easy, medium, or hard) before the game starts. This would level the playing field and make the game more fun for everyone.

4. Generating Obstacles at Random
Part of this was met. There were random obstacles in the game, which helped make each practice feel a little different. The players noticed that the patterns changed and liked that each time it wasn't exactly the same. There were, however, reports from some players that the game became dull after a few rounds. It seems that the types of obstacles and game rules weren't varied enough, even though randomness was used. More types of obstacles, power-ups, and game designs could be added to fully meet this requirement. This would make the game more fun to play again and again and keep it from getting boring over time. Some things I could do to fix this are add more types of obstacles, new mechanics or power-ups, and different backgrounds or environments between games. These changes would make each run feel more special and make the game more fun to play again and again in the future.

USABILITY FEATURES

The Controls

Most of the comments about the controls was positive. Users liked how easy and responsive it felt to move in any direction. But some players, especially those with older systems, felt a little lag when they moved quickly. This problem is probably caused by hardware limits. When movements happen faster, they use more of the system's processing power, which makes control input take longer. The control system could be made better by making some of the visual effects that use a lot of resources simpler when the game detects lower system performance. Adding a "low-spec" mode could make sure that settings work well on devices with less powerful processors. Also, changing the graphics settings in real time depending on the system's capabilities could help make sure that performance is the same on all platforms. The game could use changeable settings that change based on the player's hardware to work with less powerful systems. When a lower-end device is found, the program could change the graphics (by lowering the quality of textures, getting rid of some visual effects, or making animations simpler) while keeping the controls available. Adding a "low-spec" mode would let users change settings by hand if they are having speed problems. It will also be important to keep an eye on how different systems are performing on a regular basis so that changes can be made as needed. To make sure that the control system works well with all devices, it needs to be tested often on a range of hardware setups. As the game changes with new updates, it should be a top priority to keep the controls smooth without adding lag, especially when new hardware or features are added. When controls are optimised for lower-end systems, graphical quality or visual effects may suffer, which could change how fun the game is to play on those systems as a whole. To keep these things in balance without making the experience worse for people with better hardware, a lot of testing is needed.

The Difficulty

The dynamic difficulty scaling worked well, especially for skilled players who liked how the challenge grew. But some users thought the level of difficulty was either too fast or too slow, which made them frustrated or bored, based on how skilled they were. It looks like the game's difficulty curve wasn't always right for all players. So that doesn't happen, I could make the game harder over time, based on player performance as well as time or score limits. For instance, the game could have an adaptive challenge system that only speeds up obstacles or increases the rate at which they appear when the player has done really well in previous rounds and slows down or changes things when the player has been having a hard time even getting the average score. Adding different levels of difficulty could also accommodate both new and experienced players, making sure that everyone can find the task they're looking for. An adaptive difficulty algorithm that tracks the player's progress in real time and changes the game as needed could make the difficulty scale better. For example, the game could make it harder faster if the player's score is going up quickly. But if the person is having trouble (for example, if they can't avoid obstacles), the level of difficulty could be lowered a bit to keep them interested without being too hard. It would also be great to have different levels of challenge, like easy, medium, and hard. Players could pick their favourite level at the beginning of the game or change it while they're playing. It will be important to keep an eye on what players say and change how hard things get after updates. Over time, keeping the game's challenges just right so that players don't get frustrated or bored will be a constant job. Complex difficulty scaling might need more testing and development time to make sure the method stays fair and doesn't make players feel like they're being punished for their skill level. It might also make the game's logic more complicated, which could slow it down.

The Graphics

One of the best things about the game was its graphics. Players liked how the background scrolled and how smoothly the characters moved. But some people thought the game could use more ways to let players know when they've done a good job, like visual cues, movements, or sound effects. To improve the input, I could add animations or sound effects when a player avoids an obstacle on a close call or sets a new personal high score. This kind of feedback on progress would let players know right away when they've done well, which would make their wins more satisfying and encourage them to keep playing. I could also add extra visual effects or changes to the environment that happen automatically as the player moves. Adding graphics or sound effects could be made easier by adding a feedback system that responds to what the player does by showing or hearing certain things. For instance, a successful dodge could lead to a happy image or sound, and reaching a certain number of points could be rewarded with effects or flashing lights. This could be easily done with if statements programmed for certain occurrences in the game and play the appropriate effects. As players move through the game, the surroundings could change by switching out the background images or the obstacles they face. It is very important to make sure that adding new movements and visual effects doesn't slow things down. As updates are made to the game, it should be a top goal to keep the best balance between graphics quality and performance on all devices. Adding new images and animations could slow down older computers, especially if the game wasn't already set up to work well on those kinds of computers. Adding high-quality images could make the game's file size bigger, which could be a problem for people who don't have a lot of space on their computers.

The Score System

It was said that the score system gave players immediate feedback, which encouraged them to keep playing. But players said that there should be more specific ways to see progress, like specific awards or bonuses, so that players could get more information about how they're doing. I could make the scoring system better by adding score multipliers and score dividers. These collectibles would show up on the screen in the same way that obstacles do, but based on whether they were multipliers or dividers, they would change the rate at which your score changed. It would be more fun for players to change their scores and see how they compare by adding these collectibles. Once these items were gathered, they would briefly change the rate at which score is gained, with multipliers making the rate at which points are earned faster and dividers making it slower. To do this, we would have to add new types of objects, update the collision detection system to change the score state, and change the scoring logic to give brief bonuses or penalties based on the effect of the collectible. Changes in colour or icons, as well as UI markers, could be added to let players know when a score modifier is at work. This feature would add more variety and movement to scoring, which would make it more interesting and encourage players to plan ahead and communicate more. To make sure the game is fair, the spawn rates would need to be carefully balanced, and the game would need to be tested to stop exploits and keep its identity over time. When thinking about maintenance, it's important to make sure that new features like score multipliers and dividers are written in a way that makes them easy to update and change. Also, feedback from players and performance data can become very important as more features are added to make sure the game stays fair and balanced. One problem with these collectibles is that they can throw off the balance of the game if they are not carefully planned. For instance, frequent multipliers could make it too easy for players to get high scores, which would make growth less challenging and meaningful. On the other hand, dividers could make players frustrated if they think they are being punished unfairly. If players depend too much on how these collectibles appear, it could lead to inconsistent experiences where high scores depend more on random item placements than actual performance. This lack of consistency could make people less likely to play competitively or be happy with their progress.

The Obstacles

The randomness of the obstacles kept the game interesting, but many players said that after a few rounds, the game felt like it was being played over and over again. This means that the random generation did its job, but the obstacles weren't different or engaging enough to keep people's attention for a long time. To keep the game interesting and difficult, I could add new kinds of obstacles that affect the player in different ways. For example, oil slicks could make the car spin out for a short time, and sticky areas could slow movement for a short time. This gives the player a way to be punished and make things harder without stopping the game. Arrays can be used to keep where each obstacle is in the game, which will make the obstacle system better. A set of coordinates (x, y) would be used to describe each obstacle, and these coordinates would be saved in an array. As the game goes on. The game could check to see if the player is too close the player before a collision happens before the debuff takes effect. You can do this by using simple conditional checks to see if the player's position overlaps with the coordinates of an obstacle. If it does, a collision has occurred. This method is adaptable and makes it easy to make changes, like adding more types of obstacles or habits over time. Each new type of obstacle would need to be fully tested before it could be added so that bugs don't show up or the level of difficulty goes up too quickly. To better control how obstacles interact with each other, a hashtag system could be used. It would also be important to keep an eye on performance to make sure the game works well on all devices after updates. Adding new obstacles that change how the player controls the game could throw off the balance if they are used too much or put too often. This could make players dissatisfied or make the game feel less skill-based. This also makes level creation and testing more difficult, which takes more time to make and might make the game harder for people who aren't very good at it.

CODE LISTING

```
import pygame

import random


pygame.init()


# Set up game screen

screen = pygame.display.set_mode((900, 650))

pygame.display.set_caption("Dodge Racer")


# Load images

icon = pygame.image.load('sport-car.png')

pygame.display.set_icon(icon)

background = pygame.image.load('game bg.png')

blurred_background = pygame.transform.smoothscale(background, (900, 650))
# resizing with smoothscale often gives a softer or less sharp look, which
can feel like a blur

PlayerImg = pygame.image.load('main-car.png')

obstacle_img = pygame.image.load('cone.png')

obstacle_img = obstacle_img.convert_alpha() # prepares the image so it
runs more smoothly in the game (faster blitting).


# Font settings

font = pygame.font.Font(None, 36)  # In-game text (small)

large_font = pygame.font.Font(None, 72)  # Larger text for menus


background_y = 0  # Track background position



# Player settings

PlayerX = 405 # starting x position of player

PlayerY = 550 # starting y position of the player
```

```
player_speed = 4 # This means each time the player moves, their position changes by 4 pixels per frame

player_dx = 0 # dx stands for change in  x (horizontally), how much a players position should change along the x-axis per frame

player_dy = 0 # dy stands for change in  y (vertically), how much a players position should change along the y-axis per frame


# If Right Arrow is pressed, player_dx = 4 → Moves right.

# If Left Arrow is pressed, player_dx = -4 → Moves left.

# If both or neither are pressed, player_dx = 0 → No movement.


# Obstacle settings

speed = 4 # This means that the obstacle moves at a speed of 4 pixels per frame

obstacles = [] # This creates an empty list that will store obstacle positions. Each obstacle is represented as a list containing its x and y coordinates,

min_horizontal_gap = 110  # Minimum gap between obstacles


def player(x, y):

    screen.blit(PlayerImg, (x, y))


# This function spawns obstacles in a way that ensures they are not too close to each other horizontally.

def generate_obstacle():

    max_attempts = 10 # This limits the number of attempts when trying to place an obstacle.

    for _ in range(max_attempts):  # Try 10 times to find a valid position

        x = random.randint(147, 753 - obstacle_img.get_width()) # Random horizontal position

        y = -obstacle_img.get_height() # Start above screen


        # Ensure new obstacle is not too close to existing ones

        too_close = any(abs(x - obs[0]) < min_horizontal_gap for obs in obstacles)
        # abs(x - obs[0]):  returns the absolute value, makes sure the
value is positive so we know the correct horizontal distance between the
new object and a previous one.

        # < min_horizontal_gap for obs in obstacles:  checks every x value
of the obstacle to ensure the minimum horizontal gap is maintained

        if not too_close: # If spacing is okay, add the obstacle

            obstacles.append([x, y])

            break # Stop trying after finding a valid position


# Function to move obstacles

def move_obstacles():

    global speed

    for obstacle in obstacles.copy(): # Loop through a copy to avoid
issues when removing items

        obstacle[1] += speed # Move obstacle down

        if obstacle[1] > 650: # Remove if it goes off-screen

            obstacles.remove(obstacle)


# Function to draw obstacles

def draw_obstacles():

    for obstacle in obstacles:

        screen.blit(obstacle_img, (obstacle[0], obstacle[1]))


# Function to check collision with smaller hit-boxes

def check_collision():

    # Define a much smaller hit-box for the player

    player_rect = pygame.Rect(

        PlayerX + 15,  # Move hit-box inside more

        PlayerY + 15,

        PlayerImg.get_width() - 30,  # Reduce width even more

        PlayerImg.get_height() - 30   # Reduce height even more

    )
    for obstacle in obstacles:
        obstacle_rect = pygame.Rect(
            obstacle[0] + 10,  # Move hit-box inside more
            obstacle[1] + 10,
            obstacle_img.get_width() - 20,  # Reduce width more
            obstacle_img.get_height() - 20   # Reduce height more
        )

        if player_rect.colliderect(obstacle_rect):
            return True  # Collision detected

    return False  # No collision



def show_text(text, x, y, font_size=36):
    font_to_use = large_font if font_size > 36 else font
    render = font_to_use.render(text, True, (0, 25, 250))  # Text color is
black
    screen.blit(render, (x, y))



def start_menu():
    while True:
        screen.blit(blurred_background, (0, 0))  # Blurred background
        show_text("Dodge Racer", 300, 200, 72)  # Larger text
        show_text("Press ENTER to Start", 200, 300, 48)  # Medium text
        pygame.display.update() # refreshing the screen so that the player
sees the latest changes
        for event in pygame.event.get(): # checks all the actions the
player or the system has made since the last time we checked
            if event.type == pygame.QUIT: # checks if the player closed
the game window. If they did, the game will shut down.
                pygame.quit()
                exit() # these make sure the game shuts down properly when the player closes the window

            if event.type == pygame.KEYDOWN and event.key == pygame.K_RETURN: # this checks if the player pressed the Enter key

                return # If the player pressed Enter, this makes the game move to the next step, like starting the game




def game_over_screen(score):

    while True: #  This starts a loop that keeps showing the game over screen until the player takes an action, like pressing a key to restart or quit

        screen.blit(blurred_background, (0, 0))  # Blurred background

        show_text("Game Over!", 320, 200, 72)

        show_text(f"Your score was: {score}", 250, 300, 48)

        show_text("Press ENTER to Play Again", 120, 400, 48) # shows all necessary game over text

        pygame.display.update() # refreshing the screen so that the player sees the latest changes

        for event in pygame.event.get(): # checks all the actions the player or the system has made since the last time we checked

            if event.type == pygame.QUIT: # checks if the player closed the game window. If they did, the game will shut down.

                pygame.quit()

                exit() # these make sure the game shuts down properly when the player closes the window

            if event.type == pygame.KEYDOWN and event.key == pygame.K_RETURN: # This checks if the player presses the Enter key

                return #  If the Enter key is pressed, this causes the function to stop and return control, usually to start a new game




clock = pygame.time.Clock() # This creates a clock that controls how fast the game runs

start_menu() # This runs the start screen of the game
```

```
running = True # This starts the game and keeps it running. It ensures the
game loop continues

start_time = pygame.time.get_ticks() # helps keep track of how long the
game has been running

speed_increase_time = start_time # makes sure that the speed boost happens
as soon as the game starts.  As time goes on, this number will be used to
tell the game when 10 seconds have passed. When 10 seconds are up, the
speed will speed up.


while running:

    screen.fill((0, 0, 0)) # Clear screen

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            running = False


    # Player movement

    keys = pygame.key.get_pressed()

    player_dx = (keys[pygame.K_RIGHT] - keys[pygame.K_LEFT]) *
player_speed

    # keys is a dictionary-like structure that stores the state of all
keys.

    # for example: keys[pygame.K_RIGHT] is 1 if the right arrow key is
pressed, otherwise 0.

    player_dy = (keys[pygame.K_DOWN] - keys[pygame.K_UP]) * player_speed

    PlayerX += player_dx

    PlayerY += player_dy

    # since the variables PlayerX and PlayerY store the player's position,
We add player_dx and player_dy to update the position.


    # Keep player inside boundaries

    PlayerX = max(147, min(753 - PlayerImg.get_width(), PlayerX))

    PlayerY = max(0, min(650 - PlayerImg.get_height(), PlayerY))


    # Generate obstacles randomly with spacing

    if random.random() < 0.05:  # 5% chance per frame
```

```
        generate_obstacle()

        if random.random() < 0.25:  # 25% chance to spawn an extra
obstacle

            generate_obstacle()


    move_obstacles() # function updates the positions of the obstacles on
the screen, making them move downwards based on the speed of the game

    elapsed_time = (pygame.time.get_ticks() - start_time) // 1000 # This
line calculates how much time has passed since the game started.


    # Increase speed every 10 seconds

    if elapsed_time % 10 == 0 and pygame.time.get_ticks() -
speed_increase_time >= 10000: # This line checks if 10 seconds have passed
since the last speed increase.

        speed += 0.5 # the game speed is increased by 0.5

        speed_increase_time = pygame.time.get_ticks() # This line
remembers the time when the game last got faster, so the game can wait 10
seconds before increasing the speed again.


    # Draw the background twice to create a scrolling effect

    screen.blit(background, (0, background_y))

    screen.blit(background, (0, background_y - 650))  # Draw a second
background above the first one

    draw_obstacles()

    player(PlayerX, PlayerY)

    show_text(f"Score: {elapsed_time}", 750, 10)

    pygame.display.update()


    if check_collision():

        game_over_screen(elapsed_time)

        obstacles.clear()

        PlayerX, PlayerY = 405, 550

        start_time = pygame.time.get_ticks()

        speed = 4  # Reset speed after game over
    # Move background downward
    background_y += speed

    # Reset background position when it moves off-screen
    if background_y >= 650:
        background_y = 0

    clock.tick(60)

pygame.quit()
```