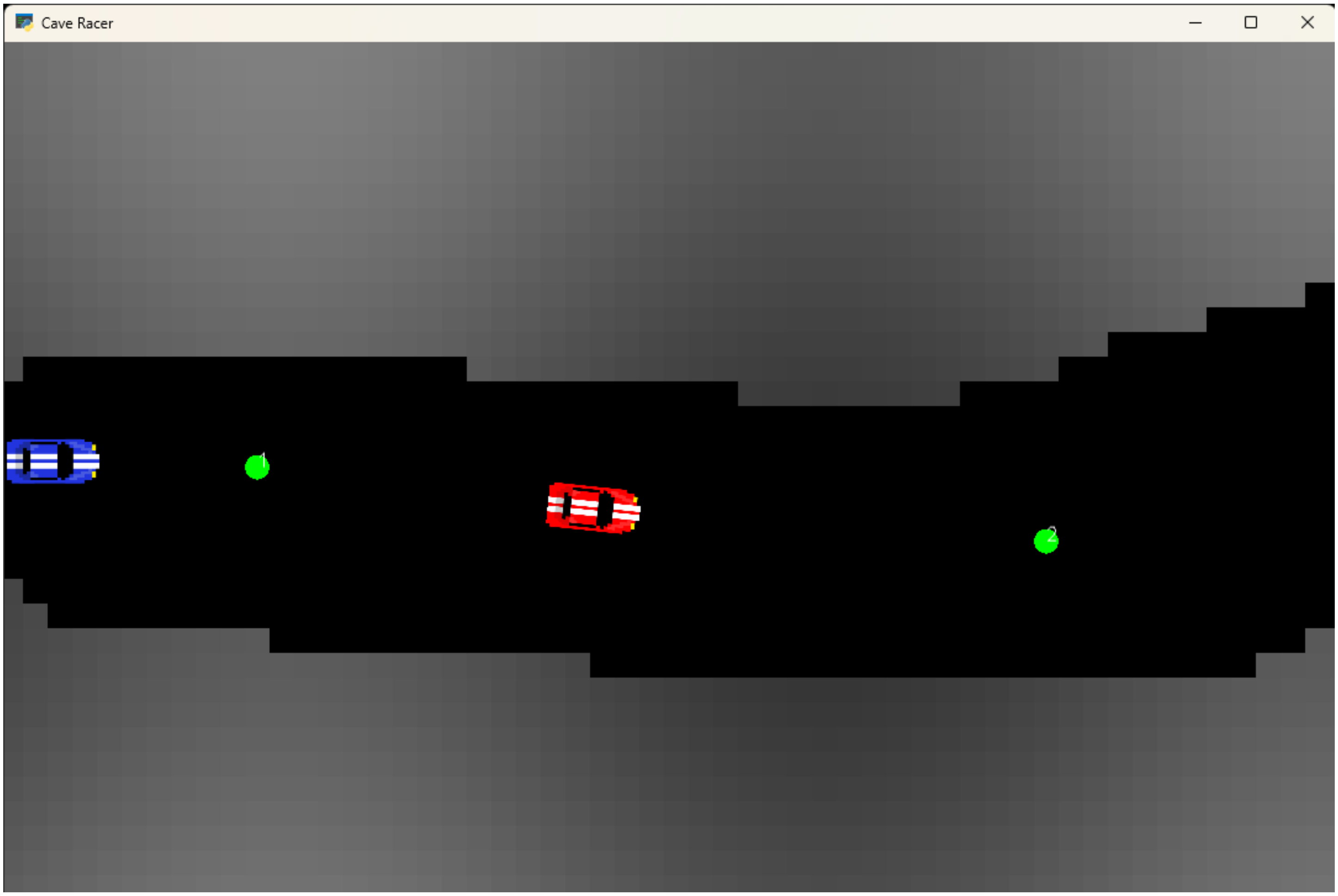
03/21/2023
CAVE RACE CHALLENGE
Team B

# EVER CHANGING CAVE RACE
# Design Document



By Corey Verkouteren and Reece Watson

March 2023

03/21/2023
CAVE RACE CHALLENGE
Team B

## Overview:

"Ever Changing" Cave Race is a top down, single player game where the player races against a computer-controlled opponent. The catch for both is that the players must face a track that changes every time the game starts. If the play wins another race is started with a different track. If the player loses the game ends.

We are using an algorithm known as ***cellular automata*** that enables us to build our racetrack randomly each time the game is played. Which means that the player must adjust to a new track each time. More importantly so must the program that's controlling the computer driven car. The first car to reach the end of the track wins the race. There are power ups on the track. If picked up by the user, it gives them the ability to shoot a drill rock that can create a new path through the cave walls that hopefully is a short cut to the end.

Targeted Audience – Our audience is anyone enjoys playing a game, especially a racing game and is in the 6 – 12-year age range. To appeal to older players, we need to add more features, levels, and additional awards and power ups.

Playtime – There is the ability to play for 10 minutes or more depending on the success of the player. The game ends when the player loses a race.

Selling Points – We think the fact that the track is different each time is a selling point it means that every time you play it, you get a slightly different experience.

Performance – The game play performance is great, there is no lag and, the controls are responsive. From a performance perspective our primary concern was load time. The game generates a random track cell by cell each time it is run. Our performance testing on load times was an average 2.38 seconds which was not a noticeable wait when you start the game.

## Initial Constraints:

The following outlines our initial constraints:

- It must be written in Python.
- It should have a back story.
- We'll use Cellular Automata – To randomly generate a different track every time the game is played.
    - The track must have a beginning and an end.
    - The track must be wide enough for the cars to reasonably navigate the track.
- The game will be able to be played with a game controller or the keyboard.
- It would have a menu and a win screen

## Back Story:

It is 2074 and the oil crisis is over. With an abundance of fuel and clean running technology, America's love affair with cars has soared. Of specific fascination is the driving skill of the racer. There are contests across the country where drivers compete against each other for points. The higher your cumulative point total the more prestige you gain.

However, if you really wanted to have ultimate prestige, and if you have at least 1000 points you can sign up for the "Everchanging Cave Race" If you run this race, you receive an additional 5000 points and the official title of "Cave Master"! However, there are some down sides:

1. The cave track is different every time, so even if this is your 5th attempt you don't know what the track will look like.
2. You are racing in a cave! It is narrow and there is a lot of rock.
3. If you lose, 1000 points are deducted from your total.
4. Your opponent is control by a supercomputer; it'll be hard to beat!

It's a lot to risk, but every driver wants to run this ever-changing track.

## Game Play

Both the player and the opponent start at the beginning of the Track. Once the countdown finishes they can immediately start driving toward the finishes that race. The first one there wins and continues to race.

While the opponent car has the advantage of being computer controlled, there are powerups on the track that the player can get. These power ups equip the car with "Drilling Missiles" that can cut through the cave generating a new path and hopefully a short cut to the end.

## Overall Technical Design

The game uses the python library Arcade for the overall game loop, key and controller input, player movement and collision detection.

We utilized and example of Cellular Automata  from the Arcades Example web site and changed it to meet the needs of our game. It is free to use and modify, no attribution was required. In addition to Arcade, we wrote several libraries ourselves to support this application.

They are:

- Bots.py
- Globals.py
- Levels.py
- Menus.py
- Misc_Functions.py
- Player.py
- World Objects
- Additional Assets – such as images and sound files

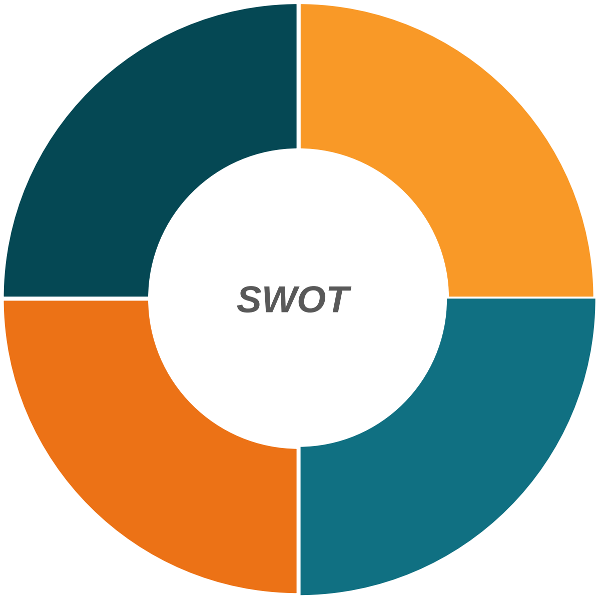
## SWOT Analysis

### STRENGTHS

- Ever-Changing Environment
- Simple Controls

### WEAKNESSES

- Some learning curve to with the controls to develop a smooth path throughout the game



### OPPORTUNITIES

- Additional levels
- Player stats including high score.
- Additional power ups
- Better visual difference between levels
- Add Sounds

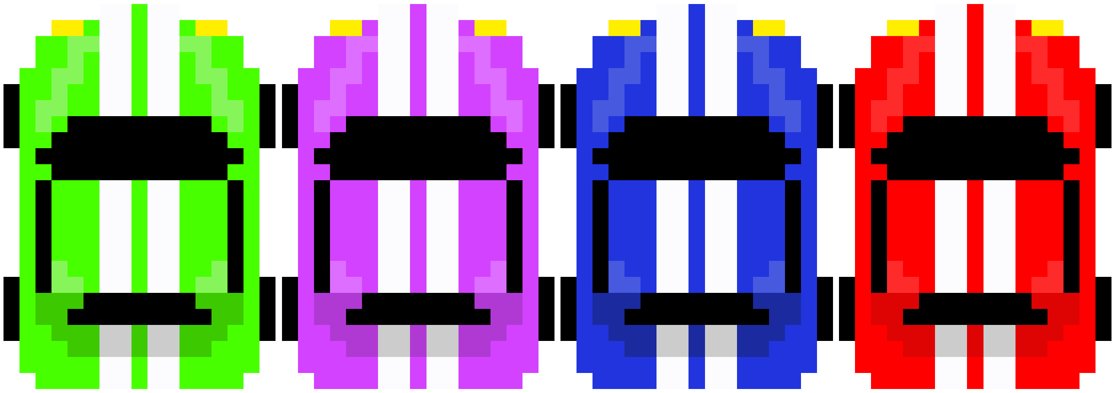
### THREATS

- Staff – As students move on, we may not get to the opportunities as we move on to the next project and next Skills USA

03/21/2023
CAVE RACE CHALLENGE
Team B

## Concept Art

All cars in the race were created by Team Member Corey Verkouteren



# Sample Code

## Main.py

This is the main piece of code from which all

```
import arcade

import arcade as arc


import Globals

import Levels as lvl

from World_Objects import Drill

from Misc_Functions import IsRectCollidingWithPoint, get_turn_multiplier

from Menus import start_menu, controls_menu, win_menu, loss_menu

from Particles import drill_wall_emit

from math import radians, sin, cos




class MainMenu(arc.View):

    def __init__(self):

        super().__init__()

        self.width = Globals.SCREEN_WIDTH

        self.height = Globals.SCREEN_HEIGHT


        self.scene = None


        self.camera = None

        self.button_list = []

        self.text_list = []


    def on_show_view(self):

        start_menu(self)
    def on_resize(self, width: int, height: int):
        self.window.set_viewport(0, width, 0, height)
        Globals.resize_screen(width, height)
        self.__init__()
        self.on_show_view()


    def on_draw(self):
        arc.draw_xywh_rectangle_filled(0, 0, Globals.SCREEN_WIDTH, Globals.SCREEN_HEIGHT, color=arc.color.DARK_SLATE_GRAY)


        for button in self.button_list:
            button.update()
        for text in self.text_list:
            try:
                text.update()
            except:
                text.draw()


    def on_mouse_press(self, mouse_x: int, mouse_y: int, button: int, modifiers: int):
        for button in self.button_list:
            if IsRectCollidingWithPoint(button.get_rect(), (mouse_x, mouse_y)):
                if button.id == "start":
                    game_view = GameView()
                    self.window.show_view(game_view)
                if button.id == "controls":
                    controls_view = ControlsView()
                    self.window.show_view(controls_view)
```

03/21/2023
CAVE RACE CHALLENGE
Team B

```
class ControlsView(arc.View):

    def __init__(self):

        super().__init__()

        self.width = Globals.SCREEN_WIDTH

        self.height = Globals.SCREEN_HEIGHT


        self.scene = None


        self.camera = None

        self.button_list = []

        self.text_list = []


    def on_show_view(self):

        controls_menu(self)


    def on_resize(self, width: int, height: int):

        self.window.set_viewport(0, width, 0, height)

        Globals.resize_screen(width, height)

        self.__init__()

        self.on_show_view()


    def on_draw(self):

        arc.draw_xywh_rectangle_filled(0, 0, Globals.SCREEN_WIDTH, Globals.SCREEN_HEIGHT,
color=arc.color.DARK_SLATE_GRAY)


        for button in self.button_list:

            button.update()

        for text in self.text_list:

            try:

                text.update()
            except:

                text.draw()


    def on_mouse_press(self, mouse_x: int, mouse_y: int, button: int, modifiers: int):

        for button in self.button_list:

            if IsRectCollidingWithPoint(button.get_rect(), (mouse_x, mouse_y)):

                if button.id == "back":

                    menu_view = MainMenu()

                    self.window.show_view(menu_view)




class EndMenus(arc.View):

    def __init__(self):

        super().__init__()

        self.width = Globals.SCREEN_WIDTH

        self.height = Globals.SCREEN_HEIGHT


        self.scene = None


        self.camera = None

        self.button_list = []

        self.text_list = []


    def on_show_view(self):

        controls_menu(self)


    def on_resize(self, width: int, height: int):

        self.window.set_viewport(0, width, 0, height)

        Globals.resize_screen(width, height)

        self.__init__()
        self.on_show_view()


    def on_draw(self):

        arc.draw_xywh_rectangle_filled(0, 0, Globals.SCREEN_WIDTH, Globals.SCREEN_HEIGHT, color=arc.color.DARK_SLATE_GRAY)


        for button in self.button_list:

            button.update()

        for text in self.text_list:

            try:

                text.update()

            except:

                text.draw()


    def on_mouse_press(self, mouse_x: int, mouse_y: int, button: int, modifiers: int):

        for button in self.button_list:

            if IsRectCollidingWithPoint(button.get_rect(), (mouse_x, mouse_y)):

                if button.id == "back":

                    menu_view = MainMenu()

                    self.window.show_view(menu_view)




class WinView(EndMenus):

    def __init__(self):

        super().__init__()


    def on_show_view(self):

        win_menu(self)
```

```
class LossView(EndMenus):

    def __init__(self):

        super().__init__()


    def on_show_view(self):

        loss_menu(self)




class GameView(arc.View):

    def __init__(self):

        super().__init__()

        self.width = Globals.SCREEN_WIDTH

        self.height = Globals.SCREEN_HEIGHT


        self.scene = None


        self.camera = None

        self.gui_camera = None

        self.view_left = 0

        self.view_bottom = 0

        self.end_of_map = Globals.CELL_GRID_WIDTH

        self.map_height = Globals.CELL_GRID_HEIGHT


        self.player = None


        # input stuff

        self.controller = None


        self.right_trigger_pressed = False

        self.left_trigger_pressed = False
        self.w_pressed = False
        self.s_pressed = False
        self.a_pressed = False
        self.d_pressed = False

        self.up_pressed = False
        self.down_pressed = False
        self.left_pressed = False
        self.right_pressed = False

        self.thumbstick_rotation = 0

        self.move_up = False
        self.move_down = False
        self.move_left = False
        self.move_right = False

        self.powerup_pressed = False

        # game stuff
        self.race_num = 1
        self.game_timer = 0
        self.past_time = 0
        self.seconds_timer = 0
        self.start_countdown = None
        self.start_countdown_num = 0
        self.emitters = []
        self.physics_engine = None
        self.bot_physics = []
        # grid/automata stuff
        self.level_update_timer = 0
        self.grid = []

        self.track_points = []

    def process_keychange(self):
        # print(self.controller.x)

        if self.player is None:
            return

        if self.controller:
            self.thumbstick_rotation = self.controller.x
        else:
            self.thumbstick_rotation = 0

        # Process left/right
        if self.w_pressed or self.up_pressed or self.right_trigger_pressed:
            self.move_up = True
        else:
            self.move_up = False

        if self.s_pressed or self.down_pressed or self.left_trigger_pressed:
            self.move_down = True
        else:
            self.move_down = False

        if self.a_pressed or self.left_pressed or self.thumbstick_rotation < -Globals.DEADZONE:
            self.move_left = True
        else:
            self.move_left = False

        if self.d_pressed or self.right_pressed or self.thumbstick_rotation > Globals.DEADZONE:
            self.move_right = True
        else:
            self.move_right = False

        if self.move_up and not self.move_down:
            self.player.accelerate()
        elif self.move_down and not self.move_up:
            self.player.backwards_accelerate()

        controller_rotation_mult = 1

        if self.thumbstick_rotation != 0:
            controller_rotation_mult = abs(self.thumbstick_rotation)

        if self.player.speed == 0 and self.player.speed == 0:
            self.player.change_angle = 0
        elif self.move_right and not self.move_left:
            self.player.change_angle = -Globals.PLAYER_ROTATION_SPEED * get_turn_multiplier(self.player.speed) * controller_rotation_mult
        elif self.move_left and not self.move_right:
            self.player.change_angle = Globals.PLAYER_ROTATION_SPEED * get_turn_multiplier(self.player.speed) * controller_rotation_mult
        elif not self.move_left and not self.move_right:
            self.player.change_angle = 0

        if self.powerup_pressed and self.player.power_up == "drill":
            self.powerup_pressed = False
            self.player.power_up = None
            new_drill = Drill(launch_angle=self.player.angle)
            new_drill.center_x = self.player.center_x
            new_drill.center_y = self.player.center_y
            self.scene.add_sprite("powerups", sprite=new_drill)


    def on_key_press(self, key, modifiers):
        if key == arc.key.W:
            self.w_pressed = True
        if key == arc.key.S:
            self.s_pressed = True
        if key == arc.key.A:
            self.a_pressed = True
        if key == arc.key.D:
            self.d_pressed = True

        if key == arc.key.UP:
            self.up_pressed = True
        if key == arc.key.DOWN:
            self.down_pressed = True
        if key == arc.key.LEFT:
            self.left_pressed = True
        if key == arc.key.RIGHT:
            self.right_pressed = True

        if key == arc.key.ESCAPE:
            quit()
        # DEV INPUTS
        if key == arc.key.SPACE:
            self.powerup_pressed = True



        # run cellular automata for 1 step
        if key == arc.key.N:
            lvl.update_level(self)

        # generate new track
        if key == arc.key.R:
            lvl.new_track(self)

        # clears current grid
        if key == arc.key.C:
            self.scene["cells"].clear()

    def on_key_release(self, key, modifiers):
        if key == arc.key.W:
            self.w_pressed = False
        if key == arc.key.S:
            self.s_pressed = False
        if key == arc.key.A:
            self.a_pressed = False
        if key == arc.key.D:
            self.d_pressed = False

        if key == arc.key.UP:
            self.up_pressed = False
        if key == arc.key.DOWN:
            self.down_pressed = False
        if key == arc.key.LEFT:
            self.left_pressed = False
        if key == arc.key.RIGHT:
            self.right_pressed = False

        self.process_keychange()

    # noinspection PyMethodMayBeStatic
    def on_joybutton_press(self, joystick, button):

        if button == 7:  # Right Trigger
            self.right_trigger_pressed = True
        elif button == 6:  # Left Trigger
            self.left_trigger_pressed = True
        elif button == 3:  # "X" Button
            self.powerup_pressed = True

    # noinspection PyMethodMayBeStatic
    def on_joybutton_release(self, joystick, button):

        if button == 7:  # Right Trigger
            self.right_trigger_pressed = False
        elif button == 6:  # Left Trigger
            self.left_trigger_pressed = False
        elif button == 3:  # "X" Button
            self.powerup_pressed = False

    def on_show_view(self):
        arc.set_viewport(0, self.window.width, 0, self.window.height)

        controllers = arcade.get_game_controllers()
        if controllers:
            self.controller = controllers[0]
            self.controller.open()
            self.controller.push_handlers(self)

        self.load_level()

    def load_level(self):
        lvl.new_track(self)

        # reset timers
        self.game_timer -= self.game_timer

        # start countdown info
        self.start_countdown = arc.Text(f"5", 0, 0, arc.color.WHITE, font_size=60,
                                        font_name="ARCADECLASSIC")
        self.start_countdown.x = (Globals.SCREEN_WIDTH / 2) - (self.start_countdown.content_width / 2)
        self.start_countdown.y = (Globals.SCREEN_HEIGHT / 2) - (self.start_countdown.content_height / 2)

    def on_resize(self, width: int, height: int):
        Globals.resize_screen(width, height)
        self.__init__()
        self.on_show_view()

    def on_draw(self):
        if self.camera is None:
            return

        self.camera.use()
        arc.draw_rectangle_filled(self.camera.position.x + self.camera.viewport_width / 2,
                                  self.camera.position.y + self.camera.viewport_height / 2,
                                  self.camera.viewport_width, self.camera.viewport_height, arc.color.BLACK)
        self.scene["powerups"].update_animation()
        self.scene.draw()

        for emitter in self.emitters:
            emitter.draw()

        '''
        for bot in self.scene["bots"]:
            arc.draw_line(bot.center_x, bot.center_y, bot.center_x + 100 * cos(bot.desired_angle), bot.center_y + 100 * sin(bot.desired_angle), (0, 0, 255), 10)
        '''
        i = 0
        for point in self.track_points:
            i += 1
            arc.draw_circle_filled(point[1] * Globals.CELL_HEIGHT + Globals.GRID_BL_POS[1], point[0] * Globals.CELL_WIDTH + Globals.GRID_BL_POS[0], 10, (0, 255, 0))
            arc.draw_text(str(i), point[1] * Globals.CELL_HEIGHT + Globals.GRID_BL_POS[1], point[0] * Globals.CELL_WIDTH + Globals.GRID_BL_POS[0])

        # gui cam stuff
        self.gui_camera.use()
        if self.start_countdown:
            self.start_countdown.draw()

    def center_camera_to_player(self):
        # Scroll left
        left_boundary = self.view_left + Globals.LEFT_VIEWPORT_MARGIN
```

```
if self.player.left < left_boundary:
    self.view_left -= left_boundary - self.player.left


# Scroll right
right_boundary = self.view_left + self.width - Globals.RIGHT_VIEWPORT_MARGIN
if self.player.right > right_boundary:
    self.view_left += self.player.right - right_boundary


# Scroll up
top_boundary = self.view_bottom + self.height - Globals.TOP_VIEWPORT_MARGIN
if self.player.top > top_boundary:
    self.view_bottom += self.player.top - top_boundary


# Scroll down
bottom_boundary = self.view_bottom + Globals.BOTTOM_VIEWPORT_MARGIN
if self.player.bottom < bottom_boundary:
    self.view_bottom -= bottom_boundary - self.player.bottom


# keeps camera in left bound of map
if self.view_left < 0:
    self.view_left = 0


# keeps camera in right bound of map
if (self.view_left + self.width) > self.end_of_map:
    self.view_left = self.end_of_map - self.width


# keeps camera in bottom bound of map
if self.view_bottom < 0:
    self.view_bottom = 0
```

```
        # keeps camera in top bound of map
        if self.view_bottom + self.height > self.map_height:
            self.view_bottom = self.map_height - self.height

        # Scroll to the proper location
        position = self.view_left, self.view_bottom
        self.camera.move_to(position, Globals.CAMERA_SPEED)

        # OLD
        """
        screen_center_x = self.player.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player.center_y - (
            self.camera.viewport_height / 2
        )

        self.camera.move_to((screen_center_x, screen_center_y))
        """

    def on_update(self, delta_time: float):
        self.game_timer += delta_time
        self.seconds_timer = int(self.game_timer)

        self.process_keychange()
        if self.seconds_timer < 4:
            self.start_countdown.text = f"Race {self.race_num} of {Globals.RACE_NUM}"
            self.start_countdown.x = Globals.MID_SCREEN - (self.start_countdown.content_width / 2)
            self.start_countdown.y = (Globals.SCREEN_HEIGHT / 2) - (self.start_countdown.content_height / 2)
        elif 4 <= self.seconds_timer <= 8.5:
            self.start_countdown_num = -((self.seconds_timer - 4) - 5)
            self.start_countdown.text = f"{self.start_countdown_num}"
            self.start_countdown.x = Globals.MID_SCREEN - (self.start_countdown.content_width / 2)
            self.start_countdown.y = (Globals.SCREEN_HEIGHT / 2) - (self.start_countdown.content_height / 2)
        elif 8.5 < self.seconds_timer == 9:
            self.start_countdown_num = -(self.seconds_timer - 5)
            self.start_countdown.text = f"GO"
        else:
            if self.start_countdown:
                self.start_countdown = None
            self.scene.update()
            self.physics_engine.update()
            for phy in self.bot_physics:
                phy.update()
        for emitter in self.emitters:
            emitter.update()
        self.center_camera_to_player()


        # player-power up box interaction
        collisions = arc.check_for_collision_with_list(self.player, self.scene["power_boxes"])
        for box in collisions:
            self.player.power_up = "drill"
            box.kill()


        # bot-exit interaction
        for bot in self.scene["bots"]:
            bot_exit_collisions = arc.check_for_collision_with_list(bot, self.scene["exit"])
            if bot_exit_collisions:
                l_view = LossView()
                self.window.show_view(l_view)


        # player-exit interaction
        exit_collisions = arc.check_for_collision_with_list(self.player, self.scene["exit"])

        if exit_collisions:

            self.race_num += 1

            if self.race_num > Globals.RACE_NUM:

                win_view = WinView()

                self.window.show_view(win_view)

            else:

                Globals.randomize_wall_color()

                self.load_level()


        # powerup interactions

        for powerup in self.scene["powerups"]:

            if powerup.type == "drill":

                for cell in powerup.collides_with_list(self.scene["cells"]):

                    emitter_label, new_emitter = drill_wall_emit((cell.center_x, cell.center_y),

                                                                 cell.texture.image.getcolors()[0][1])

                    self.emitters.append(new_emitter)

                    cell.kill()



def main():

    """Main function"""

    window = arc.Window(Globals.SCREEN_WIDTH, Globals.SCREEN_HEIGHT, Globals.SCREEN_TITLE, fullscreen=False,
resizable=True)

    start_view = MainMenu()

    window.show_view(start_view)

    arc.run()

             if __name__ == "__main__":

    main()
```

03/21/2023
CAVE RACE CHALLENGE
Team B

## Bot.py

This is the module that controls the computer car

```
#bot.py
#Corey Verkouteren, Reece Watson
#3/22/23
#Cave Race Challenge
#The is a 2D Top Down car race game that takes place in a cave

import arcade as arc
from Globals import *
from Misc_Functions import get_closest_wall
from math import sin, cos, radians, degrees, sqrt, atan2, pi


class Car(arc.Sprite):
    def __init__(self):
        super().__init__()

    def update(self):
        self.center_x = -self.change_y * sin(radians(self.angle))

        self.center_y = self.change_y * cos(radians(self.angle))


class BasicBot(arc.Sprite):
    def __init__(self, walls, track_points):
        super().__init__("./Assets/Player/Audi.png")

        self.scale = .3
        self.angle = -90

        self.desired_angle = 0

        self.walls = walls
        self.track_points = track_points
        self.last_track_point = -1

        self.wall_closeness = CELL_HEIGHT * 1.5

        self.speed = 0

        self.max_speed = BOT_MAX_SPEED

    def accelerate(self):
        '''
        if self.change_y < self.max_speed:
            self.change_y += 2
        '''
        self.change_y = self.max_speed

    def update(self):

        if self.last_track_point + 1 == len(self.track_points):
            self.kill()
            return

        next_track_point = self.track_points[self.last_track_point + 1]
        next_track_point_pos = (next_track_point[1] * CELL_HEIGHT + GRID_BL_POS[1], 
next_track_point[0] * CELL_WIDTH + GRID_BL_POS[0])

        if sqrt(abs(next_track_point_pos[0] - self.center_x)**2 + 
abs(next_track_point_pos[1] - self.center_y)**2) < 5 * CELL_HEIGHT:
            self.last_track_point += 1

        self.angle %= 360

        desired_angle = atan2(next_track_point_pos[1] - self.center_y, 
next_track_point_pos[0] - self.center_x) - (pi/2)

        '''
        angle_diff = degrees(desired_angle)

        cw_y_dist = self.center_y - closest_wall.center_y
        cw_x_dist = self.center_x - closest_wall.center_x

        direction = 0
        if angle_diff > 0:
            direction = -1
        elif angle_diff < 0:
            direction = 1

        self.angle -= direction
        '''

        self.desired_angle = desired_angle + (pi/2)  # for debugging
        self.angle = degrees(desired_angle)

        self.accelerate()

        self.center_x += -self.change_y * sin(radians(self.angle))
        self.center_y += self.change_y * cos(radians(self.angle))

        self.change_x = 0
        self.change_y = 0
```

03/21/2023
CAVE RACE CHALLENGE
Team B

## Particles.py

This module pixelate the cave ground as the drill passes through it.

```
import arcade as arc
from PIL import Image


particle_color_num = 0



class WallParticle(arc.FadeParticle):
    def __init__(self, wall_color):
        global particle_color_num
        texture = arc.Texture(f"Wall_Particle{particle_color_num}", Image.new("RGBA", (32, 32), wall_color), hit_box_algorithm=None)
        particle_color_num += 1
        print(particle_color_num)
        super().__init__(filename_or_texture=texture, change_xy=arc.rand_in_circle((0, 0), 1),
                         lifetime=.5)



class DrillEmitter(arc.Emitter):
    def __init__(self, center_xy, wall_color=(50, 50, 50)):
        super().__init__(center_xy=center_xy, emit_controller=arc.EmitBurst(1),
                         particle_factory=lambda emitter: WallParticle(wall_color))



def drill_wall_emit(center_xy, wall_color=(50, 50, 50)):
    e = DrillEmitter(center_xy, wall_color)
    return drill_wall_emit.__doc__, e
```