```
        x_hor = ox + depth_hor * cos_a

        delta_depth = dy / sin_a
        dx = delta_depth * cos_a

        for i in range(MAX_DEPTH):
            tile_hor = int(x_hor), int(y_hor)
            if tile_hor == self.map_pos:
                player_dist_h = depth_hor
                break
            if tile_hor in self.game.map.world_map:
                wall_dist_h = depth_hor
                break
            x_hor += dx
            y_hor += dy
            depth_hor += delta_depth

        # verticals
        x_vert, dx = (x_map + 1, 1) if cos_a > 0 else (x_map - 1e-6, -1)

        depth_vert = (x_vert - ox) / cos_a
        y_vert = oy + depth_vert * sin_a

        delta_depth = dx / cos_a
        dy = delta_depth * sin_a

        for i in range(MAX_DEPTH):
            tile_vert = int(x_vert), int(y_vert)
            if tile_vert == self.map_pos:
                player_dist_v = depth_vert
                break
            if tile_vert in self.game.map.world_map:
                wall_dist_v = depth_vert
                break
            x_vert += dx
            y_vert += dy
            depth_vert += delta_depth

        player_dist = max(player_dist_v, player_dist_h)
        wall_dist = max(wall_dist_v, wall_dist_h)

        if 0 < player_dist < wall_dist or not wall_dist:
            return True
        return False

    def draw_ray_cast(self):
        pg.draw.circle(self.game.screen, 'red',
                       (100 * self.x, 100 * self.y), 15)
        if self.ray_cast_player_npc():
```

```
            pg.draw.line(
                self.game.screen,
                'orange',
                (100 * self.game.player.x,
                 100 * self.game.player.y),
                (100 * self.x,
                 100 * self.y),
                2)


class SoldierNPC(NPC):
    def _init_(
            self,
            game,
            path='resources/sprites/npc/soldier/0.png',
            pos=(
                10.5,
                5.5),
            scale=0.6,
            shift=0.38,
            animation_time=180):
        super()._init_(game, path, pos, scale, shift, animation_time)


class CacoDemonNPC(NPC):
    def _init_(
            self,
            game,
            path='resources/sprites/npc/caco_demon/0.png',
            pos=(
                10.5,
                6.5),
            scale=0.7,
            shift=0.27,
            animation_time=250):
        super()._init_(game, path, pos, scale, shift, animation_time)
        self.attack_dist = 1.0
        self.health = 150
        self.attack_damage = 25
        self.speed = 0.05
        self.accuracy = 0.35
        self.points = 300


class CyberDemonNPC(NPC):
    def _init_(
            self,
            game,
            path='resources/sprites/npc/cyber_demon/0.png',
```

```
        pos=(
            11.5,
            6.0),
        scale=1.0,
        shift=0.04,
        animation_time=210):
    super()._init_(game, path, pos, scale, shift, animation_time)
    self.attack_dist = 6
    self.health = 350
    self.attack_damage = 15
    self.speed = 0.055
    self.accuracy = 0.25
    self.points = 600
```

player.py

```
rom settings import *
import pygame as pg
import math


class Player:
    def _init_(self, game):
        self.game = game
        self.x, self.y = PLAYER_POS
        self.angle = PLAYER_ANGLE
        self.shot = False
        self.health = PLAYER_MAX_HEALTH
        self.rel = 0
        self.health_recovery_delay = 700
        self.time_prev = pg.time.get_ticks()

    def recover_health(self):
        if self.check_health_recovery_delay() and self.health < PLAYER_MAX_HEALTH:
            self.health += 1

    def check_health_recovery_delay(self):
        time_now = pg.time.get_ticks()
        if time_now - self.time_prev > self.health_recovery_delay:
            self.time_prev = time_now
            return True

    def check_game_over(self):
        if self.health < 1:
            self.game.object_renderer.game_over()
            pg.display.flip()
            pg.time.delay(1500)
            self.game.new_game()

    def get_damage(self, damage):
        self.health -= damage
        self.game.object_renderer.player_damage()
        self.game.sound.player_pain.play()
        self.check_game_over()

    def single_fire_event(self, event):
        if event.type == pg.MOUSEBUTTONDOWN:
            if event.button == 1 and not self.shot and not self.game.weapon.reloading:
                self.game.sound.shotgun.play()
                self.shot = True
                self.game.weapon.reloading = True

    def movement(self):
        sin_a = math.sin(self.angle)
        cos_a = math.cos(self.angle)
        dx, dy = 0, 0
        speed = PLAYER_SPEED * self.game.delta_time
        speed_sin = speed * sin_a
        speed_cos = speed * cos_a
```

```
keys = pg.key.get_pressed()
        if keys[pg.K_w]:
            dx += speed_cos
            dy += speed_sin
        if keys[pg.K_s]:
            dx += -speed_cos
            dy += -speed_sin
        if keys[pg.K_a]:
            dx += speed_sin
            dy += -speed_cos
        if keys[pg.K_d]:
            dx += -speed_sin
            dy += speed_cos

        self.check_wall_collision(dx, dy)

    def check_wall(self, x, y):
        return (x, y) not in self.game.map.world_map

    def check_wall_collision(self, dx, dy):
        scale = PLAYER_SIZE_SCALE / self.game.delta_time
        if self.check_wall(int(self.x + dx * scale), int(self.y)):
            self.x += dx
        if self.check_wall(int(self.x), int(self.y + dy * scale)):
            self.y += dy

    def draw(self):
        pg.draw.line(self.game.screen, 'yellow', (self.x * 100, self.y * 100),
                     (self.x * 100 + WIDTH * math.cos(self.angle),
                      self.y * 100 + WIDTH * math. sin(self.angle)), 2)
        pg.draw.circle(self.game.screen, 'green', (self.x * 100, self.y * 100), 15)

    def mouse_control(self):
        mx, my = pg.mouse.get_pos()
        if mx < MOUSE_BORDER_LEFT or mx > MOUSE_BORDER_RIGHT:
            pg.mouse.set_pos([HALF_WIDTH, HALF_HEIGHT])
        self.rel = pg.mouse.get_rel()[0]
        self.rel = max(-MOUSE_MAX_REL, min(MOUSE_MAX_REL, self.rel))
        self.angle += self.rel * MOUSE_SENSITIVITY * self.game.delta_time

    def update(self):
        self.movement()
        self.mouse_control()
        self.recover_health()

    @property
    def pos(self):
        return self.x, self.y

    @property
    def map_pos(self):
        return int(self.x), int(self.y)
```

settings.py

```
import math

# game settings
#RES = WIDTH, HEIGHT = 1280, 720
RES = WIDTH, HEIGHT = 1920, 1080
HALF_WIDTH = WIDTH // 2
HALF_HEIGHT = HEIGHT // 2
FPS = 60

PLAYER_POS = 1.5, 5  # mini_map
PLAYER_ANGLE = 0
PLAYER_SPEED = 0.004
PLAYER_ROT_SPEED = 0.002
PLAYER_SIZE_SCALE = 60
PLAYER_MAX_HEALTH = 100

MOUSE_SENSITIVITY = 0.0003
MOUSE_MAX_REL = 40
MOUSE_BORDER_LEFT = 100
MOUSE_BORDER_RIGHT = WIDTH - MOUSE_BORDER_LEFT

FLOOR_COLOR = (30, 30, 30)

FOV = math.pi / 2
HALF_FOV = FOV / 2
NUM_RAYS = WIDTH // 2
HALF_NUM_RAYS = NUM_RAYS // 2
DELTA_ANGLE = FOV / NUM_RAYS
MAX_DEPTH = 20

SCREEN_DIST = HALF_WIDTH / math.tan(HALF_FOV)
SCALE = WIDTH // NUM_RAYS

TEXTURE_SIZE = 256
HALF_TEXTURE_SIZE = TEXTURE_SIZE // 2
```

textinput.py

```
# import sys module
import pygame
import sys
# pygame.init() will initialize all
# imported module
pygame.init()

clock = pygame.time.Clock()

# it will display on screen
screen = pygame.display.set_mode([600, 500])

# basic font for user typed
base_font = pygame.font.Font(None, 32)
user_text = ''

# create rectangle
input_rect = pygame.Rect(200, 200, 140, 32)

# color_active stores color(lightskyblue3) which
# gets active when input box is clicked by user
color_active = pygame.Color('lightskyblue3')

# color_passive store color(chartreuse4) which is
# color of input box.
color_passive = pygame.Color('chartreuse4')
color = color_passive

active = False

while True:
    for event in pygame.event.get():

      # if user types QUIT then the screen will close
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        if event.type == pygame.MOUSEBUTTONDOWN:
            if input_rect.collidepoint(event.pos):
                active = True
            else:
                active = False

        if event.type == pygame.KEYDOWN:

            # Check for backspace
            if event.key == pygame.K_BACKSPACE:
                  # get text input from 0 to -1 i.e. end.
                user_text = user_text[:-1]

            # Unicode standard is used for string
            # formation
            else:
                user_text += event.unicode
```

```
# it will set background color of screen
screen.fill((255, 255, 255))

if active:
    color = color_active
else:
    color = color_passive

# draw rectangle and argument passed which should
# be on screen
pygame.draw.rect(screen, color, input_rect)

text_surface = base_font.render(user_text, True, (255, 255, 255))

# render at position stated in arguments
screen.blit(text_surface, (input_rect.x + 5, input_rect.y + 5))

# set width of textfield so that text cannot get
# outside of user's text input
input_rect.w = max(100, text_surface.get_width() + 10)

# display.flip() will update only a portion of the
# screen to updated, not full area
pygame.display.flip()
```

object renderer.py

```
import pygame as pg
from database import addScore
from settings import *


class ObjectRenderer:
    def _init_(self, game):
        self.game = game
        self.screen = game.screen
        self.font_renderer = game.font_renderer
        self.wall_textures = self.load_wall_textures()
        self.sky_image = self.get_texture(
            'resources/textures/sky.png', (WIDTH, HALF_HEIGHT))
        self.sky_offset = 0
        self.blood_screen = self.get_texture(
            'resources/textures/blood_screen.png', RES)
        self.digit_size = 90
        self.digit_images = [self.get_texture(
            f'resources/textures/digits/{i}.png', [self.digit_size] * 2) for i in range(11)]
        self.digits = dict(zip(map(str, range(11)), self.digit_images))
        self.game_over_image = self.get_texture(
            'resources/textures/game_over.png', RES)
        self.win_image = self.get_texture('resources/textures/win.png', RES)
        self.minimap = pg.Surface((300, 300))

    def draw(self):
        self.draw_background()
        self.render_game_objects()
        self.draw_player_health()

    def win(self):
        self.screen.blit(self.win_image, (0, 0))

    def game_over(self):
        self.screen.blit(self.game_over_image, (0, 0))
        addScore(self.game.name, self.game.score)

    def draw_player_health(self):
        health = str(self.game.player.health)
        self.font_renderer.render(health, (20, HEIGHT - 80), 2)

    def player_damage(self):
        self.screen.blit(self.blood_screen, (0, 0))

    def draw_background(self):
        self.sky_offset = (self.sky_offset + 4.5 *
                           self.game.player.rel) % WIDTH
        self.screen.blit(self.sky_image, (-self.sky_offset, 0))
        self.screen.blit(self.sky_image, (-self.sky_offset + WIDTH, 0))
        # floor
        pg.draw.rect(self.screen, FLOOR_COLOR, (0, HALF_HEIGHT, WIDTH, HEIGHT))

    def render_game_objects(self):
        list_objects = sorted(
            self.game.raycasting.objects_to_render,
            key=lambda t: t[0],
            reverse=True)
```

```
        for depth, image, pos in list_objects:
            self.screen.blit(image, pos)

        self.font_renderer.drawTimeTexts()
        self.font_renderer.render(
            str(self.game.score), (WIDTH / 2, 20), 2, (255, 255, 255))
        self.font_renderer.render(
            f"Enimies left: {self.game.enemies}", (20, 60), 2, (255, 129, 120))
        self.render_minimap()

    def render_minimap(self):
        self.minimap.blit(self.game.map.minimap_texture, (0, 0))
        xFactor = self.minimap.get_width() / len(self.game.map.mini_map[0])
        yFactor = self.minimap.get_height() / len(self.game.map.mini_map)
        for npc in self.game.object_handler.npc_list:
            color = "red" if npc.alive else "black"
            pg.draw.circle(
                self.minimap,
                color,
                (npc.x * xFactor, npc.y * yFactor),
                4)

        x = self.game.player.x * xFactor
        y = self.game.player.y * yFactor
        pg.draw.circle(
            self.minimap,
            'blue',
            (x, y),
            4)

        rad = self.game.player.angle
        playerDirX = x + math.cos(rad) * 25
        playerDirY = y + math.sin(rad) * 25
        pg.draw.line(  # draw player direction
            self.minimap, 'blue', (x, y), (playerDirX, playerDirY), 2)
        self.screen.blit(self.minimap, (WIDTH - 300, HEIGHT - 300))

    @staticmethod
    def get_texture(path, res=(TEXTURE_SIZE, TEXTURE_SIZE)):
        texture = pg.image.load(path).convert_alpha()
        return pg.transform.scale(texture, res)

    def load_wall_textures(self):
        return {
            1: self.get_texture('resources/textures/1.png'),
            2: self.get_texture('resources/textures/2.png'),
            3: self.get_texture('resources/textures/3.png'),
            4: self.get_texture('resources/textures/4.png'),
            5: self.get_texture('resources/textures/5.png'),
        }
```

pathFinder.py

```
from collections import deque


class PathFinding:
    def _init_(self, game):
        self.game = game
        self.map = game.map.mini_map
        self.ways = [-1, 0], [0, -1], [1, 0], [0, 1], [-1, -1], [1, -1], [1, 1], [-1, 1]
        self.graph = {}
        self.get_graph()

    def get_path(self, start, goal):
        self.visited = self.bfs(start, goal, self.graph)
        path = [goal]
        step = self.visited.get(goal, start)

        while step and step != start:
            path.append(step)
            step = self.visited[step]
        return path[-1]

    def bfs(self, start, goal, graph):
        queue = deque([start])
        visited = {start: None}

        while queue:
            cur_node = queue.popleft()
            if cur_node == goal:
                break
            next_nodes = graph[cur_node]

            for next_node in next_nodes:
                if next_node not in visited and next_node not in self.game.object_handler.npc_positions:
                    queue.append(next_node)
                    visited[next_node] = cur_node
        return visited

    def get_next_nodes(self, x, y):
        return [(x + dx, y + dy) for dx, dy in self.ways if (x + dx, y + dy) not in self.game.map.world_map]

    def get_graph(self):
        for y, row in enumerate(self.map):
            for x, col in enumerate(row):
                if not col:
                    self.graph[(x, y)] = self.graph.get((x, y), []) + self.get_next_nodes(x, y)
```

profile.py

```
from button import Button
import sys
import pygame as pg
from database import addScore, getScores, loadScores
from settings import *


class ProfileMenu:
    def _init_(self, mainmenu):
        game = mainmenu.game
        self.mainmenu = mainmenu
        self.game = game
        self.screen = game.screen
        self.font_renderer = game.font_renderer
        self.mouseDown = False
        self.surface = pg.Surface((800, 300))
        self.x = (WIDTH - 800) // 2
        self.y = (HEIGHT - 600) // 2
        self.surface.fill((0, 0, 0))
        self.initButtons()
        self.text = game.name

    def initButtons(self):
        self.buttons = []
        back = Button(
            self.game,
            20 + self.x,
            20 + self.y,
            'X', 'yellow',
            self.back)

        save = Button(
            self.game,
            400 + self.x,
            200 + self.y,
            'Save', 'Green',
            self.save)

        self.add_button(back)
        self.add_button(save)

    def add_button(self, button):
        self.buttons.append(button)

    def back(self):
        self.mainmenu.show_profile = False

    def save(self):
        self.mainmenu.game.name = self.text
        addScore(self.text, 0)
        self.mainmenu.initButtons()
        self.back()

    def draw(self):
        self.surface.fill((0, 0, 0))
```

```
for key in self.mainmenu.keys:
    if key.key == pg.K_BACKSPACE:
        self.text = self.text[:-1]
    elif key.key == pg.K_RETURN:
        self.back()
        break
    elif len(self.text) < 12:
        self.text += key.unicode
for button in self.buttons:
    button.draw()
    self.surface.blit(
        button.surface, (button.x - self.x, button.y - self.y))
    if self.mainmenu.mouseDown:
        button.clicked()

(texture, rect) = self.font_renderer.fontLarge.render(
    self.text, fgcolor='white')
self.surface.blit(texture, (200, 100))
self.screen.blit(self.surface, (self.x, self.y))
```

raycasting.py

```
import pygame as pg
import math
from settings import *


class RayCasting:
    def _init_(self, game):
        self.game = game
        self.ray_casting_result = []
        self.objects_to_render = []
        self.textures = self.game.object_renderer.wall_textures

    def get_objects_to_render(self):
        self.objects_to_render = []
        for ray, values in enumerate(self.ray_casting_result):
            depth, proj_height, texture, offset = values

            if proj_height < HEIGHT:
                wall_column = self.textures[texture].subsurface(
                    offset * (TEXTURE_SIZE - SCALE), 0, SCALE, TEXTURE_SIZE
                )
                wall_column = pg.transform.scale(wall_column, (SCALE, proj_height))
                wall_pos = (ray * SCALE, HALF_HEIGHT - proj_height // 2)
            else:
                texture_height = TEXTURE_SIZE * HEIGHT / proj_height
                wall_column = self.textures[texture].subsurface(
                    offset * (TEXTURE_SIZE - SCALE), HALF_TEXTURE_SIZE - texture_height // 2,
                    SCALE, texture_height
                )
                wall_column = pg.transform.scale(wall_column, (SCALE, HEIGHT))
                wall_pos = (ray * SCALE, 0)

            self.objects_to_render.append((depth, wall_column, wall_pos))

    def ray_cast(self):
        self.ray_casting_result = []
        texture_vert, texture_hor = 1, 1
        ox, oy = self.game.player.pos
        x_map, y_map = self.game.player.map_pos

        ray_angle = self.game.player.angle - HALF_FOV + 0.0001
        for ray in range(NUM_RAYS):
            sin_a = math.sin(ray_angle)
            cos_a = math.cos(ray_angle)

            # horizontals
            y_hor, dy = (y_map + 1, 1) if sin_a > 0 else (y_map - 1e-6, -1)

            depth_hor = (y_hor - oy) / sin_a
            x_hor = ox + depth_hor * cos_a

            delta_depth = dy / sin_a
            dx = delta_depth * cos_a

            for i in range(MAX_DEPTH):
                tile_hor = int(x_hor), int(y_hor)
```

```
            if tile_hor in self.game.map.world_map:
                texture_hor = self.game.map.world_map[tile_hor]
                break
            x_hor += dx
            y_hor += dy
            depth_hor += delta_depth

        # verticals
        x_vert, dx = (x_map + 1, 1) if cos_a > 0 else (x_map - 1e-6, -1)

        depth_vert = (x_vert - ox) / cos_a
        y_vert = oy + depth_vert * sin_a

        delta_depth = dx / cos_a
        dy = delta_depth * sin_a

        for i in range(MAX_DEPTH):
            tile_vert = int(x_vert), int(y_vert)
            if tile_vert in self.game.map.world_map:
                texture_vert = self.game.map.world_map[tile_vert]
                break
            x_vert += dx
            y_vert += dy
            depth_vert += delta_depth

        # depth, texture offset
        if depth_vert < depth_hor:
            depth, texture = depth_vert, texture_vert
            y_vert %= 1
            offset = y_vert if cos_a > 0 else (1 - y_vert)
        else:
            depth, texture = depth_hor, texture_hor
            x_hor %= 1
            offset = (1 - x_hor) if sin_a > 0 else x_hor

        # remove fishbowl effect
        depth *= math.cos(self.game.player.angle - ray_angle)

        # projection
        proj_height = SCREEN_DIST / (depth + 0.0001)

        # ray casting result
        self.ray_casting_result.append((depth, proj_height, texture, offset))

        ray_angle += DELTA_ANGLE

    def update(self):
        self.ray_cast()
        self.get_objects_to_render()
```

sound.py

```
import pygame as pg


class Sound:
    def _init_(self, game):
        self.game = game
        pg.mixer.init()
        self.path = 'resources/sound/'

    def loadGameSounds(self):
        self.shotgun = pg.mixer.Sound(self.path + 'shotgun.wav')
        self.npc_pain = pg.mixer.Sound(self.path + 'npc_pain.wav')
        self.npc_death = pg.mixer.Sound(self.path + 'npc_death.wav')
        self.npc_shot = pg.mixer.Sound(self.path + 'npc_attack.wav')
        self.npc_shot.set_volume(0.2)
        self.player_pain = pg.mixer.Sound(self.path + 'player_pain.wav')
        self.theme = pg.mixer.music.load(self.path + 'theme.mp3')
        pg.mixer.music.set_volume(0.4)

    def loadMenuSounds(self):
        self.menuTheme = pg.mixer.music.load(self.path + 'menu.mp3')
        pg.mixer.music.set_volume(0.6)
```

spriteObject.py

```
import pygame as pg
from settings import *
import os
from collections import deque


class SpriteObject:
    def _init_(
            self,
            game,
            path='resources/sprites/static_sprites/candlebra.png',
            pos=(
                10.5,
                3.5),
            scale=0.7,
            shift=0.27):
        self.game = game
        self.player = game.player
        self.x, self.y = pos
        self.image = pg.image.load(path).convert_alpha()
        self.IMAGE_WIDTH = self.image.get_width()
        self.IMAGE_HALF_WIDTH = self.image.get_width() // 2
        self.IMAGE_RATIO = self.IMAGE_WIDTH / self.image.get_height()
        self.dx, self.dy, self.theta, self.screen_x, self.dist, self.norm_dist = 0, 0, 0, 0, 1, 1
        self.sprite_half_width = 0
        self.SPRITE_SCALE = scale
        self.SPRITE_HEIGHT_SHIFT = shift

    def get_sprite_projection(self):
        proj = SCREEN_DIST / self.norm_dist * self.SPRITE_SCALE
        proj_width, proj_height = proj * self.IMAGE_RATIO, proj

        image = pg.transform.scale(self.image, (proj_width, proj_height))

        self.sprite_half_width = proj_width // 2
        height_shift = proj_height * self.SPRITE_HEIGHT_SHIFT
        pos = self.screen_x - self.sprite_half_width, HALF_HEIGHT - \
            proj_height // 2 + height_shift

        self.game.raycasting.objects_to_render.append(
            (self.norm_dist, image, pos))

    def get_sprite(self):
        dx = self.x - self.player.x
        dy = self.y - self.player.y
        self.dx, self.dy = dx, dy
        self.theta = math.atan2(dy, dx)

        delta = self.theta - self.player.angle
        if (dx > 0 and self.player.angle > math.pi) or (dx < 0 and dy < 0):
            delta += math.tau

        delta_rays = delta / DELTA_ANGLE
        self.screen_x = (HALF_NUM_RAYS + delta_rays) * SCALE

        self.dist = math.hypot(dx, dy)
```

```
        self.norm_dist = self.dist * math.cos(delta)
        if -self.IMAGE_HALF_WIDTH < self.screen_x < (
                WIDTH + self.IMAGE_HALF_WIDTH) and self.norm_dist > 0.5:
            self.get_sprite_projection()

    def update(self):
        self.get_sprite()


class AnimatedSprite(SpriteObject):
    def _init_(
            self,
            game,
            path='resources/sprites/animated_sprites/green_light/0.png',
            pos=(
                11.5,
                3.5),
            scale=0.8,
            shift=0.16,
            animation_time=120):
        super()._init_(game, path, pos, scale, shift)
        self.animation_time = animation_time
        self.path = path.rsplit('/', 1)[0]
        self.images = self.get_images(self.path)
        self.animation_time_prev = pg.time.get_ticks()
        self.animation_trigger = False

    def update(self):
        super().update()
        self.check_animation_time()
        self.animate(self.images)

    def animate(self, images):
        if self.animation_trigger:
            images.rotate(-1)
            self.image = images[0]

    def check_animation_time(self):
        self.animation_trigger = False
        time_now = pg.time.get_ticks()
        if time_now - self.animation_time_prev > self.animation_time:
            self.animation_time_prev = time_now
            self.animation_trigger = True

    def get_images(self, path):
        images = deque()
        for file_name in sorted(os.listdir(path)):
            if os.path.isfile(os.path.join(path, file_name)):
                img = pg.image.load(path + '/' + file_name).convert_alpha()
                images.append(img)
        return images
```

uirenderer.py

```
import pygame as pg


class UIRenderer:
    def _init_(self, game):
        self.game = game
        self.screen = game.screen
        self.font_renderer = game.font_renderer
        self.buttons = []

    def update(self):
        pos = pg.mouse.get_pos()
        for button in self.buttons:
            button.draw()
            button.clicked(pos)

    def add_button(self, button):
        self.buttons.append(button)
```

```
from sprite_object import *


class Weapon(AnimatedSprite):
    def _init_(
            self,
            game,
            path='resources/sprites/weapon/shotgun/0.png',
            scale=0.4,
            animation_time=90):
        super()._init_(game=game, path=path, scale=scale, animation_time=animation_time)
        self.images = deque([pg.transform.smoothscale(img, (self.image.get_width(
        ) * scale, self.image.get_height() * scale)) for img in self.images])
        self.weapon_pos = (
            HALF_WIDTH - self.images[0].get_width() // 2,
            HEIGHT - self.images[0].get_height())
        self.reloading = False
        self.num_images = len(self.images)
        self.frame_counter = 0
        self.damage = 50

    def animate_shot(self):
        if self.reloading:
            self.game.player.shot = False
            if self.animation_trigger:
                self.images.rotate(-1)
                self.image = self.images[0]
                self.frame_counter += 1
                if self.frame_counter == self.num_images:
                    self.reloading = False
                    self.frame_counter = 0

    def draw(self):
        self.game.screen.blit(self.images[0], self.weapon_pos)

    def update(self):
        self.check_animation_time()
        self.animate_shot()
```

database.py

```
import json

data = {}


def loadScores():
    try:
        with open('data.json', 'r') as f:
            global data
            data = json.load(f)
    except BaseException:
        print('Error loading data.json')


def saveScores():
    with open('data.json', 'w') as f:
        json.dump(data, f)


def addScore(name, score):
    if name in data:
        oldScore = data[name]
        if score > oldScore:
            data[name] = score
    else:
        data[name] = score
    data['current'] = name
    saveScores()


def getScores():
    return data


def sortScores():
    filtered = {k: v for k, v in data.items() if k != 'current'}
    return {
        k: v for k,
        v in sorted(
            filtered.items(),
            key=lambda item: item[1],
            reverse=True)}
```

4.3 TOOLS USED

Frontend Tools: Python, Pygame library, texture

Backend Tools: Python Database

5. TESTING

5.1 INTRODUCTION

Since the project is completed, we need to carry out testing to check if it is successful. If each component is working properly in all aspects and gives desired output, then project is said to be successful. So in order to say our project is completely successful, it needs to be tested first.

5.2 FUNCTIONAL TESTING

a) *Testing functionality of the Game Screen*:

Test Case Description:
We will verify whether the controls of the game are working as per the design plan and actions of the player is displayed on the screen.
Test Steps:
Press ‘W’, ‘S’, ‘A’ and ‘D’. Then use mouse right click to fire towards enemies.
Expected Result:
The player moves forward, then backward, then to the left and then to the right. Firing is activated when right mouse button is clicked.
Actual Result:
The player moves according to the keys pressed. First forward, then backward, then to the left and then to the right. Firing is activated when right mouse button is clicked.
Status: Successful

5.3 SYSTEM TESTING

System Testing is checking whether all the requirements were satisfied by the obatined result. The steps involved here are:

a) Unit Testing: Unit testing focuses verification efforts on the smallest unit of the software design, the module. This is also known as “Module Testing”. All the modules are tested separately. This testing is carried out during programming stage itself. Each module is found to be working satisfactorily with regard to the expected output from the module.

b)Integration Testing: The objective is to take unit tested modules and build a program structure. All the modules are combined and tested as a whole. Here, correction is difficult because the isolation of cause is complicated by the vast expense of the entire program. Integration Testing stops all the errors taht gets uncovered and are corrected for the next testing steps. It was also successfully carried out.

c) System Testing: It is the stage of implementation which ensures whether the system works accurately and efficiently when live operation commences. Testing is vital to the success of the system. System testing makes a logical assumption that if all the parts of the system are correct, then goal will be successfully achieved.

d) Validation Testing: Validation succeeds when the software function in a manner that can reasonably expected by the customer. After validation test has been conducted, one of two possible conditions exists. One is that the performance characteristics confirm to specifications and are accepted and the other is that the deviation from specification is uncovered and a deficiency list is created. Proposed system under consideration has been tested by using validation testing and found to be working satisfactorily.

e) Output Testing: After performing validation testing, the next step is output testing of the proposed system. Since no system could be useful if it does not produce the required output in the specified format, asking the users about the format, tests the outputs generated by the system under consideration. Here, the output format is considered in two ways, one is on the screen and other is the printed format. The output format on the screen is found to be correct as the format that was designed in the system design phase.

f) User Acceptance Testing: User Acceptance of a system is the most important key factor of the success of any designed system. As this determines how satisfied a user is and whether the system is accepted finally by the user, it is the most crucial testing.

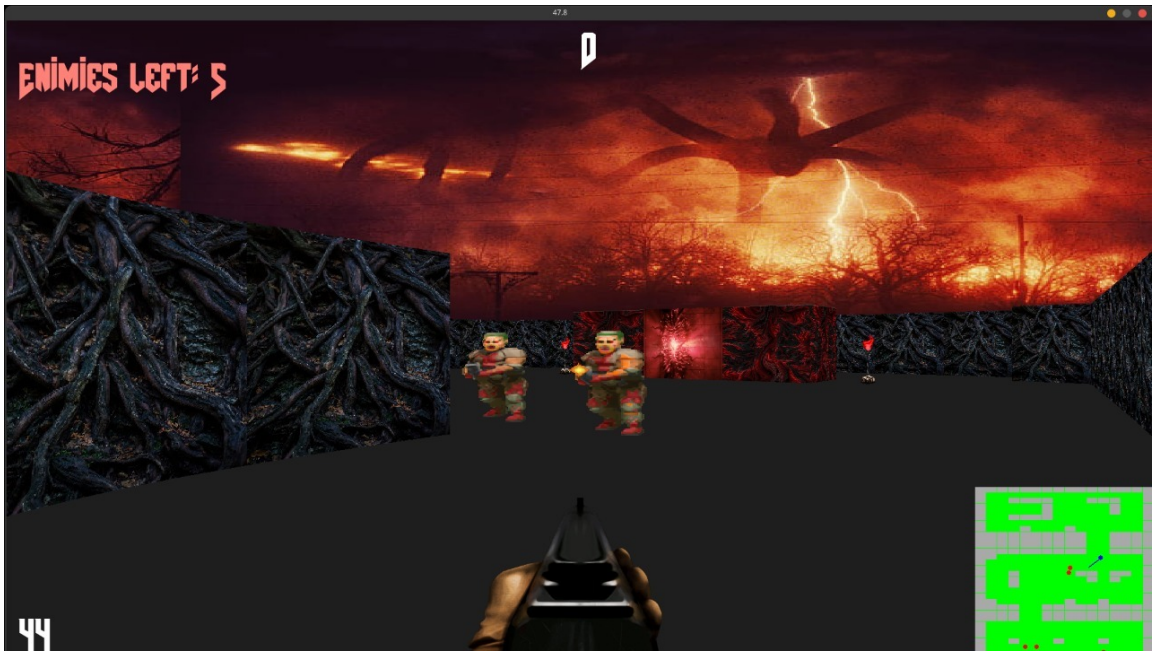
6. RESULT

6.1 OUTPUT

The game is successfully implemented, tested and obtained as per the design specified while in the design phase. All the controls work as per the requirements and result status is successful.

6.2 OUTPUT SCREENSHOTS

SPLASH SCREEN


ENIMIES LEFT: 5
0
44