Universidad Nacional de Ingeniería

DACTIC

Ingeniería en computación

World Legacy 3D

Integrantes

Burgos Silva Rugby Josué // 2022-0251U

Darce Mairena Yasser José // 2022-0378U

Díaz Hernández Leonardo Rosalío // 2022-0246U

Barahona Engel Jasmir // 2022-0319U

Grupo: 3T1-CO

Docente: Danny Oswaldo Chavez Miranda

Viernes 5 de Julio del 2024

Objetivos Generales

Desarrollar un ambiente virtual en 3D que recree distintos patrimonios de la humanidad

Objetivos Específicos

1. Implementar un motor gráfico en 3D utilizando Python y OpenGL

2. Integrar un sistema de navegación intuitivo que permita a los usuarios moverse libremente

Introducción

El proyecto consiste en la creación de un entorno 3D que recrea un museo virtual, cuyas principales atracciones son diversos patrimonios de la humanidad, incluyendo monumentos y esculturas icónicas. La elección de este proyecto se fundamenta en su significativo valor cultural y educativo, permitiéndonos preservar y mostrar estos patrimonios de manera accesible para todo público.

Además World Legacy brinda la capacidad de explorar diferentes ángulos y detalles de los modelos 3D, lo que lo convierte en una herramienta de gran relevancia en diversas disciplinas educativas, tales como arquitectura, historia, arqueología y arte. Este proyecto no solo facilita la accesibilidad de los diferentes patrimonios de la humanidad, sino que también enriquece el aprendizaje interactivo y multidimensional de estudiantes.

Este proyecto también incorpora distintas características interactivas como una pantalla de inicio la cual cuenta con opciones de “Inicio”, “Créditos” y “Ayuda” lo cual facilita y enriquece la interacción del usuario. Entre estas distintas características interactivas este proyecto también cuenta con un menú donde el usuario puede ajustar el sonido del entorno a su preferencia.

Un aspecto importante a destacar es la facilidad para el usuario de iniciar su recorrido en el interior o exterior del museo desde el menú. Esto proporciona una experiencia personalizada, permitiendo una exploración más libre y atractiva del entorno 3D.

Algunos de los modelos implementados en el proyecto fueron La Torre Eiffel, El Big Ben, La Catedral de San Basilio, El Coliseo Romano, La Estatua de David de Miguel Angel, la Esfinge, La Torre Inclinada de Pisa, la Estatua Moai y como parte del ambiente externo del museo se incluyó la estatua de Venus de Milo.

Desarrollo

World Legacy se desarrolló utilizando Python y OpenGL, mediante la biblioteca ModernGL, que permite la creación de aplicaciones gráficas avanzadas utilizando la API de OpenGL. ModernGL facilita la integración con otras bibliotecas y marcos de trabajo de Python, como PyGLM, que proporciona tipos y funciones matemáticas similares a los de GLSL (OpenGL Shading Language). Además, se utilizó Pygame para manejar la reproducción de sonidos dentro del proyecto, y NumPy para proporcionar soporte para arrays y matrices multidimensionales, junto con una colección de funciones matemáticas que permiten operar con estos datos de manera eficiente.

La estructura del proyecto se desarrolló de manera extensa mediante usos de diferentes archivos a partir del cual se aplicaron distintas funciones para el proyecto, a partir de **main.py** implementa un motor gráfico utilizando Python con las bibliotecas Pygame y ModernGL para crear una aplicación gráfica 3D. Este código configura una aplicación gráfica en 3D que utiliza Pygame para manejar la ventana y el sonido, y ModernGL para renderizar gráficos 3D. La aplicación crea un entorno 3D interactivo con luces, cámara, una escena que se renderiza continuamente y también maneja el volumen del sonido ambiente con las letras Z y X.

```
        # Light
        self.light = Light(position=(50, 50, -10), color=(1, 1, 1))
        self.additional_lights = [
            AdditionalLight(position=(0, -3, -10), color=(1, 1, 1)),
            AdditionalLight2(position=(0, 20, -10), color=(1, 0.5, 0.5))
        ]

        # Camera
        self.camera = Camera(self)
        self.mesh = Mesh(self)
        self.scene = Scene(self)

        # Audio Manager
        self.audio_manager = AudioManager(self.camera)
        # Load and play sound
        self.sound = pg.mixer.Sound('Audios/ambiente.mp3')
        self.sound.play(-1)  # Loop the sound

        # Info Manager
        self.info_manager = InfoManager(self.camera)  # Inicializar InfoManager

    def check_events(self):

        # Procesar eventos de Pygame


        for event in pg.event.get():
            if event.type == pg.QUIT or (event.type == pg.KEYDOWN and event.key == pg.K_ESCAPE):
                self.mesh.destroy()
                pg.quit()
                sys.exit()
            elif event.type == pg.KEYDOWN:
                if event.key == pg.K_z:
                    self.audio_manager.adjust_volume(-0.1)  # Disminuir el volumen
                elif event.key == pg.K_x:
                    self.audio_manager.adjust_volume(0.1) # Aumentar el volumen
```

Para cargar las luces se hace uso de un archivo **light.py** el cual define clases sobre las diferentes luces que se están usando en el entorno 3D, estableciendo sus propiedades y calculando las matrices de vista asociadas a cada luz para su correcta representación. Para la configuración sobre la cámara se usa un archivo **camera.py** el cual representa la visión del usuario sobre el entorno 3D se implementan el uso de constantes las cuales son necesarias para definir el campo de visión en grados, la distancia desde la cámara al plano de recorte más cercano, la distancia desde la cámara al plano de recorte más lejano, la velocidad de movimiento de la cámara y la sensibilidad de rotación de la cámara. También se implementan matrices de vista y proyección, además de utilizar el método move(self) permite mover la cámara hacia adelante, atrás y hacia los lados, utilizando las teclas W, S, A y D.

```python
def move(self):
    velocity = SPEED * self.app.delta_time
    keys = pg.key.get_pressed()
    validate = self.verifyGrande() is not True and self.verifyPequeño() if not self.stateCamera else self.verifyGrande()
    #print(self.verifyGrande() is not True)
    print(self.position)

    if keys[pg.K_w]:
        self.z = self.position[2] + self.forward[2] * velocity
        self.x = self.position[0] + self.forward[0] * velocity
        if validate and self.collisions.check_limits():
            self.position[2] = self.z
            self.position[0] = self.x

    if keys[pg.K_s]:
        self.z = self.position[2] - self.forward[2] * velocity
        self.x = self.position[0] - self.forward[0] * velocity
        if validate and self.collisions.check_limits():
            self.position[2] = self.z
            self.position[0] = self.x

    if keys[pg.K_a]:
        self.x = self.position[0] - self.right[0] * velocity
        self.z = self.position[2] - self.right[2] * velocity
        if validate and self.collisions.check_limits():
            self.position[0] = self.x
            self.position[2] = self.z

    if keys[pg.K_d]:
        self.x = self.position[0] + self.right[0] * velocity
        self.z = self.position[2] + self.right[2] * velocity
        if validate and self.collisions.check_limits():
            self.position[0] = self.x
            self.position[2] = self.z

    if keys[pg.K_1]:
        self.stateCamera = False
        self.position = glm.vec3(-9.65, 10, -128.883)
    if keys[pg.K_2]:
        self.stateCamera = True
        self.position = glm.vec3(0, 10, 10)
```

Dentro del mismo **camera.py** trabajamos las colisiones, las cuales abarcan las colisiones con los modelos 3D, y las colisiones con el museo. Las colisiones se manejan a través de la clase

collisions de **collisions.py**, que se inicializa con una lista de modelos que representan objetos en el entorno 3D. La clase **camera** utiliza esta instancia de **collisions** para verificar si la posición actual de la cámara colisiona con alguno de estos modelos antes de permitir que la cámara se mueva hacia esa posición.

```
models = [

    {'position': (26, -136), 'size': (6, 6)},    #Coliseo
    {'position': (-33, 137), 'size': (5, 5)},    #Eiffel
    {'position': (-126, 60), 'size': (4, 4)},    #PizzaTower
    {'position': (116, -73), 'size': (7, 7)},    #Catedral
    {'position': (-39, -136), 'size': (4, 4)},   #Estatua (Esfinge)
    {'position': (-111, -81), 'size': (5, 5)},   #Estatua2 (Michelangelo)
    {'position': (120, 63), 'size': (4, 4)},     #BigBen
    {'position': (25, 136), 'size': (5, 5)},     #Moai

    # Elementos fuera del museo
    {'position': (0, 0), 'size': (8, 8)},     #Venus de Milo
    {'position': (-4, 73), 'size': (14, 14)},  #Banco 1
    {'position': (-4, -73), 'size': (14, 14)}, #Banco 2
    {'position': (69, 27), 'size': (14, 14)},  #Banco 3
    {'position': (-70, 25), 'size': (14, 14)}, #Banco 4

    {'position': (40, 50), 'size': (5, 5)},      #Arbol 1
    {'position': (-40, 50), 'size': (5, 5)},     #Arbol 2
    {'position': (-35, -68), 'size': (5, 5)},    #Arbol 3
    {'position': (27, -68), 'size': (5, 5)},     #Arbol 4
    {'position': (-68, 1), 'size': (5, 5)},      #Arbol 5
    {'position': (68, 1), 'size': (5, 5)},       #Arbol 6
```

Posteriormente tenemos las funciones **verifyGrande** y **verifyPequeño**, Estas funciones verifican si la posición actual de la cámara (**self.x** y **self.z**) está dentro de los límites de dos polígonos definidos por sus vertices (**vertices** y **verticesDentro**), estos polígonos corresponden a la figura del museo, la cual es un hexágono. Utilizan el método de ray casting para determinar si el punto está dentro del polígono en el plano xz.

```
    def verifyGrande(self):
        vertices = ([
            [-43.8833, -76.0611 ],
            [43.8734, -76.213],

            [88.1478, 0.119104],

            [43.8008, 76.1551],
            [-44.1507, 76.1202],

            [-87.9573, -0.117065]])

        # Definir el punto a verificar
        P = ([self.x, self.z])
```

```
    def is_point_in_polygonFuera(point, vertices):
        n = len(vertices)
        inside = False
        x, z = point
        p1x, p1z = vertices[0]
        for i in range(n + 1):
            p2x, p2z = vertices[i % n]
            if z > min(p1z, p2z):
                if z ≤ max(p1z, p2z):
                    if x ≤ max(p1x, p2x):
                        if p1z ≠ p2z:
                            xinters = (z - p1z) * (p2x - p1x) / (p2z - p1z) + p1x
                        if p1x == p2x or x ≤ xinters:
                            inside = not inside
            p1x, p1z = p2x, p2z
        return inside

    # Verificar si el punto está dentro del hexágono en el plano 'xz'
    if is_point_in_polygonFuera(P, vertices):
        return True
    else:
        return False
```

Esta devolverá **True** si la cámara se encuentra dentro del polígono.

En el archivo **collisions.py**, la clase **Collisions** tiene métodos para verificar los límites de los modelos en relación con la posición de la cámara. **check_limits** es un método que itera sobre los modelos definidos y verifica si la posición **x** y **z** de la cámara cae dentro de los límites del modelo. Calcula los límites del modelo basados en su **position** y **size**, y devuelve **False** si la cámara colisiona con algún modelo, de otro modo devuelve **True**, impidiendo así que la cámara se mueva a esa posición

```
def check_limits(self):
    for model in self.models:
        position, size = model['position'], model['size']
        limits = (size[0] / 2, size[1] / 2)
        bool_x = position[0] - limits[0] < self.app.x < position[0] + limits[0]
        bool_z = position[1] - limits[1] < self.app.z < position[1] + limits[1]

        if bool_x and bool_z:      D-Maysena, la semana pasada • Merge with Ram
            return False
    return True
```

En la función **move** de **camera.py** hacemos uso de las funciones antes mencionadas, ya que **move** se encarga de gestionar el movimiento de la cámara en función de las teclas presionadas por el usuario. Luego encontramos el nuevo valor de las coordenadas **z** y **x** de la posición de la cámara en función de su dirección hacia adelante (**self.forward**) y la velocidad de movimiento (**velocity**), posteriormente se verifican si hay colisiones con los modelos 3D del entorno o colisiones con el museo, para ello hacemos uso de una variable **validate**

(booleana), y de **self.stateCamera**, la cual se inicializa como **True** y se encargará de manejar el estado de la camara, osea si se encuentra dentro o fuera del museo.

```
validate = self.verifyGrande() is not True and self.verifyPequeño() if not self.stateCamera else self.verifyGrande()
if keys[pg.K_w]:
    self.z = self.position[2] + self.forward[2] * velocity
    self.x = self.position[0] + self.forward[0] * velocity
    if validate and self.collisions.check_limits():
        self.position[2] = self.z
        self.position[0] = self.x
```

Si no hay colisiones, el nuevo valor de **z** y **x**, se le asignan a la **position** de la cámara, permitiendo que la cámara se desplace, de otro modo la cámara no podrá moverse.

```
    if keys[pg.K_1]:
        self.stateCamera = False
        self.position = glm.vec3(-9.65, 10, -128.883)
    if keys[pg.K_2]:
        self.stateCamera = True
        self.position = glm.vec3(0, 10, 10)
```

Si se presiona la tecla **1** el estado de la cámara cambia a **False**, que sucede cuando la cámara entró al museo, si se toca la tecla **2** se cambia a **True**, manteniendo una sintonía con las verificaciones anteriores.

Para definir cómo se renderizan los elementos del museo se hace uso de **shader_program.py** el cual libera recursos de la GPU asociados con cada programa de shader, cabe destacar que para renderizar los objetos normales se define shader default, para renderizar el entorno del cielo se hace uso de shader skybox y posteriormente el shader advanced_skybox para recursos adicionales del skybox. Para procesar los vértices se hace uso de **default.vert** el cual su propósito principal es transformar las posiciones de los vértices desde el espacio local del objeto hasta el espacio de pantalla y pasar información adicional necesaria para el procesamiento de fragmentos, el cual se realiza mediante el uso de **default.frag** el cual determina el color final de cada fragmento que se renderiza, también aplica efectos de iluminación y texturización para simular como la luz interactúa con las superficies del museo.

Para cargar los objetos se hace uso de distintos archivos, primeramente **VBO.py** define varias clases para manejar VBO, usando ModernGL y PyWavefront, todos estos modelos se encuentran en una carpeta objects en extensión .obj, para el cual se necesita aplicar su ruta de

ubicación para cada modelo.

```
class Estatua2VBO(BaseVBO):
    def __init__(self, app):
        super().__init__(app)
        self.format = '2f 3f 3f'
        self.attribs = ['in_texcoord_0', 'in_normal', 'in_position']

    def get_vertex_data(self):
        objs = pywavefront.Wavefront('objects/Statue_v1_L2.123cc93d694a-81fb-4c81-8a75-7fa010dfa777/12330_Statue_v1_L2.obj', cache=True, parse=True)
        obj = objs.materials.popitem()[1]
        vertex_data = obj.vertices
        vertex_data = np.array(vertex_data, dtype='f4')
        return vertex_data
```

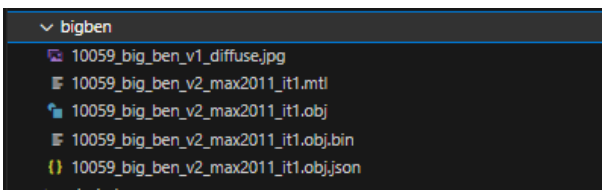
Posteriormente en **VAO.py** encapsula los vbo y los shaders, permitiendo la configuración de los atributos de los vértices.

```
# Estatua 2
self.vaos['estatua2'] = self.get_vao(
    program=self.program.programs['default'],
    vbo=self.vbo.vbos['estatua2'])
```

Una vez obtenidos los vbos y vaos, el archivo **model.py** establece una estructura de clases para crear y manejar los modelos donde se manejan transformaciones, renderización y texturas para los distintos modelos desde las clases específicas para cada uno de los modelos.

```
class Estatua2(ExtendedBaseModel):
    def __init__(self, app, vao_name='estatua2', tex_id='estatua2',
                 pos=(0, 0, 0), rot=(-90,0,0), scale=(0.025,0.025,0.025)):
        super().__init__(app, vao_name, tex_id, pos, rot, scale)
```

En **texture.py** facilita la carga y procesamiento de las texturas para cada modelo agregado, para lograr cargar la textura de cada modelo se necesita establecer la ruta de las imágenes. Cada modelo .obj cuenta con un archivo .jpg en su carpeta la cual es la que se utiliza para establecer la textura de cada objeto, el cual también cuenta con otros archivos dentro de la carpeta de cada modelo como .mtl, .json y .bin.



Cada uno de estos archivos es importante para la implementación del modelo, el archivo .obj contiene la geometría, el archivo .mtl describe los materiales, el archivo .json contiene configuraciones adicionales del modelo y el archivo.bin contiene datos binarios relacionados con el renderizado.

Luego **mesh.py** se crea una instancia de la clase vao y texture donde se incluye la definición de vértices, posiciones y texturas asociadas al objeto. También hace uso de destroy() que se encarga de liberar recursos utilizados por el vao y texture cuando ya no son necesarios.

El archivo **scene_renderer.py** se encarga del proceso de renderizado, primeramente renderiza las sombras, luego el renderizado principal de los objetos y por último el skybox. También gestiona la configuración y limpieza de los recursos necesarios.Para montar la escena que se muestra en este proyecto utiliza un archivo llamado **scene.py** el cual gestiona la carga y el renderizado de todos los objetos y luces dentro de una escena 3D. Inicializa y carga los modelos y luces en el método load, y proporciona métodos para agregar objetos y luces adicionales. El método render se encarga de renderizar todos los objetos en la escena y el skybox para completar el proceso de visualización de la escena.

```python
class Scene:
    def __init__(self, app):
        self.app = app
        self.objects = []
        self.lights = []
        self.load()
        self.skybox = AdvancedSkyBox(app)

    def add_object(self, obj):
        self.objects.append(obj)

    def add_light(self, light):
        self.lights.append(light)

    def load(self):
        app = self.app
        add = self.add_object

        n, s = 100, 2
        for x in range(-n, n, s):
            for z in range(-n, n, s):
                add(Cube(app, pos=(x, -s, z)))
        add(Coliseo(app, pos=(25, 4.5, -150), scale=(0.00035, 0.00035, 0.00035)))
        add(Eiffel(app, pos=(-33, 4, 130), scale=(0.00030, 0.00030, 0.00030)))
        add(PisaTower(app, pos=(-126, 5, 60)))
        add(Catedral(app, pos=(118, 4, -80)))
        add(Estatua(app, pos=(-33, 4, -150)))
        add(Estatua2(app, pos=(-125, 4, -80)))
        add(Bigben(app, pos=(116, 4, 60)))
        add(moai(app, pos=(25, 4, 130)))
        add(museo4(app,pos=(-4,-0.99,-10), scale=(0.2,0.2,0.2)))

```

Siguiendo con el desarrollo del ambiente , nos encontramos con el skybox ,el cual está ubicado en archivos previamente trabajados en este caso son: **texture.py** con la siguiente líneas y funciones.

```
self.textures['skybox'] = self.get_texture_cube(dir_path='textures/skybox1/', ext='png')
```

```
def render(self):
    for obj in self.objects:
        obj.render()
    self.skybox.render()
```

esta funcion se encuentra en **scene.py** , siguiendo nos encontramos con la creación de una clase muy importante para renderizar nuestro skybox, tenemos la classe **SkyBoxVBO** ,la cual consiste en la toma de los índices de la imágenes que cubren el espacio determinado para el skybox y que sean acorde a dicho espacio ,esto se trabajó con vértices, y arrays, para un mejor renderizado y facilidad

```
class SkyBoxVBO(BaseVBO):
    def __init__(self, ctx):
        super().__init__(ctx)
        self.format = '3f'
        self.attribs = ['in_position']

    @staticmethod
    def get_data(vertices, indices):
        data = [vertices[ind] for triangle in indices for ind in triangle]
        return np.array(data, dtype='f4')

    def get_vertex_data(self):
        vertices = [(-1, -1, 1), ( 1, -1,  1), (1,  1,  1), (-1, 1,  1),
                    (-1, 1, -1), (-1, -1, -1), (1, -1, -1), ( 1, 1, -1)]

        indices = [(0, 2, 3), (0, 1, 2),
                   (1, 7, 2), (1, 6, 7),
                   (6, 5, 4), (4, 7, 6),
                   (3, 4, 5), (3, 5, 0),
                   (3, 7, 4), (3, 2, 7),
                   (0, 6, 1), (0, 5, 6)]
        vertex_data = self.get_data(vertices, indices)
        vertex_data = np.flip(vertex_data, 1).copy(order='C')
        return vertex_data
```

Esto seria todo por parte del skybox, puesto que al pasarse tanto a **vbo.py y vao.py** , no hay necesidad de llamarlos en el main ,puesto que para que funcione este skybox se pasa a **scene.py** , la cual importa todo lo de la escena del programa sin necesidad de estar haciendo tantos llamados al main.

Al implementar sonidos en el entorno donde se encuentran los modelos 3D, hacemos uso de la biblioteca pygame. Nos auxiliamos de una clase nueva que engloba tanto las referencias a los sonidos como la lógica que implementa el cálculo de la proximidad de la cámara con respecto a un modelo 3D específico, esta se encuentra en el archivo **audios.py**. Para enriquecer la experiencia inmersiva del usuario se utilizaron sonidos que representen o hagan referencia al período, o país de las esculturas, un sonido diferente por cada uno de ellos.

Iniciamos definiendo un diccionario que contenga cada sonido con respecto a cada escultura.

```
def load_sounds(self):
    # Cargar los sonidos de los archivos y almacenarlos en un diccionario
    sound_files = {
        "Pedestal_bigben": 'bigben.mp3',
        "Pedestal_eiffel": 'eiffel.mp3',
        "Pedestal_coliseo": 'coliseo.mp3',
        "Pedestal_pisatower": 'pisatower.mp3',
        "Pedestal_catedral": 'catedral.mp3',
        "Pedestal_estatua": 'estatua.mp3',
        "Pedestal_estatua2": 'estatua2.mp3',
        "Pedestal_moai": 'moai.mp3'
    }
```

Posteriormente tomamos las coordenadas de cada objeto y realizamos el cálculo de la proximidad de la cámara mediante un algoritmo.

```
def check_proximity(self):
    # Verificar la proximidad de la cámara a los pedestales
    positions = {
        "Pedestal_bigben": (120, -1, 63),       #Big Ben
        "Pedestal_eiffel": (-33, -1, 137),      #Torre Eiffel
        "Pedestal_coliseo": (26, -1, -136),     #Coliseo
        "Pedestal_pisatower": (-126, -1, 60),   #Torre de Pisa
        "Pedestal_catedral": (116, -1, -73),    #Catedral de San Basilio
        "Pedestal_estatua": (-39, -1, -136),    #Esfinge
        "Pedestal_estatua2": (-111, -1, -81),   #David de Miguel Ángel
        "Pedestal_moai": (25, -1, 136)          #Moai
    }
    cam_pos = self.camera.position
    threshold = 33.0  # Define un umbral de proximidad en unidades

    # Verificar la distancia de la cámara a cada pedestal
    for key, pos in positions.items():
        distance = ((cam_pos[0] - pos[0]) ** 2 + (cam_pos[1] - pos[1]) ** 2 + (cam_pos[2] - pos[2]) ** 2) ** 0.5
        if distance < threshold:
            print(f"Proximity detected for {key} at distance {distance}")
            self.play_sound(key)
            return

    self.stop_current_sound()  #Detener el sonido actual si no se detecta proximidad
```

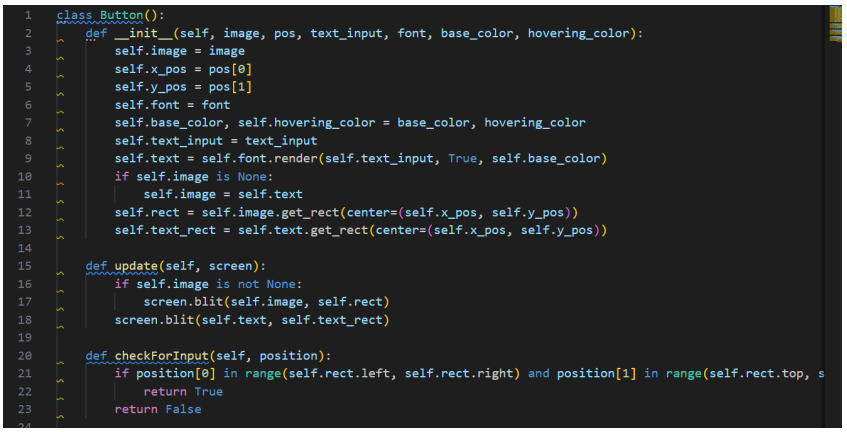
Luego definimos una función que permita bajar o subir el volumen mediante las teclas Z y X. Función a la que hacemos referencia en el archivo **main.py** donde tenemos definido un sonido de ambiente general que simula el ruido de fondo que podemos encontrar en lugares públicos.

```
def adjust_volume(self, change):
    # Ajustar el volumen tanto del sonido de fondo como del sonido de proximidad
    self.background_volume = max(0.0, min(self.background_volume + change, 1.0))
    self.proximity_volume = max(0.0, min(self.proximity_volume + change, 1.0))

    # Ajustar el volumen del sonido de fondo y de proximidad
    self.background_sound.set_volume(self.background_volume if self.current_sound is None else self.background_volume * 0.1)
    if self.current_sound is not None:
        self.current_sound.set_volume(self.proximity_volume)

    print(f"Adjusted volumes - Background: {self.background_volume}, Proximity: {self.proximity_volume}")
```

Siguiendo en el proceso encontramos la creación de la interfaz principal de “World Legacy 3D” , la cual consiste en inserción en el main, y en la creación de un archivo llamado **button.py**, la cual es para el manejo de los eventos de los botones y también su posición ,tamaño y estilo . constituida en la clase **Button ,** en la cual se puede observar la declaración de varias funciones la más destacadas están el update que permite la toma de la imagen que estará haciendo como de fondo en los botones. También está el checkforinput que permite que solo admita el teclado estando en la zona determinada del botón.

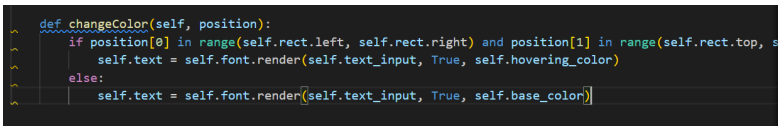


```
class Button():
    def __init__(self, image, pos, text_input, font, base_color, hovering_color):
        self.image = image
        self.x_pos = pos[0]
        self.y_pos = pos[1]
        self.font = font
        self.base_color, self.hovering_color = base_color, hovering_color
        self.text_input = text_input
        self.text = self.font.render(self.text_input, True, self.base_color)
        if self.image is None:
            self.image = self.text
        self.rect = self.image.get_rect(center=(self.x_pos, self.y_pos))
        self.text_rect = self.text.get_rect(center=(self.x_pos, self.y_pos))

    def update(self, screen):
        if self.image is not None:
            screen.blit(self.image, self.rect)
        screen.blit(self.text, self.text_rect)

    def checkForInput(self, position):
        if position[0] in range(self.rect.left, self.rect.right) and position[1] in range(self.rect.top, s
            return True
        return False
```

tambien encontramos con funciones como la que simula un hover en html ,que es que al momento que el usuario pase el mouse sobre el boton este muestre un efecto como de cambio de color.



```
def changeColor(self, position):
    if position[0] in range(self.rect.left, self.rect.right) and position[1] in range(self.rect.top, s
        self.text = self.font.render(self.text_input, True, self.hovering_color)
    else:
        self.text = self.font.render(self.text_input, True, self.base_color)
```

Para la realización de la información que contiene cada estatua se realizó con un nuevo archivo llamado , **info.py ,** Para hacer su llamado al main,se trabajó con la creación de la clase InfoManager,a cual contiene tanto como una función de proximidad ,así como el estilo de la ventana ,una ventana de mensaje ,entre otras cosas.

```python
class InfoManager:
    def __init__(self, camera):
        self.camera = camera
        self.current_key = None
        self.positions = {
            "Object_bigben": (120, 0, 63),
            "Object_eiffel": (-33, 0, 137),
            "Object_coliseo": (26, 0, -136),
            "Object_pisatower": (-126, 0, 60),
            "Object_catedral": (116, 0, -73),
            "Object_esfinge": (-39, 0, -136),
            "Object_david": (-111, 0, -81),
            "Object_moai": (25, 0, 136)
        }
        self.info = {
            "Object_bigben": (
                "El Big Ben\n\n"
                "Ubicación: Londres, Inglaterra\n\n"
                "Altura: 96,3 metros\n\n"
                "Descripción: El Big Ben, oficialmente conocido como Torre del Reloj del Palacio de Westmi
                "es una torre neogótica que alberga el reloj de cuatro caras más grande del mundo. Es uno
                "símbolos más emblemáticos de la ciudad y del Reino Unido."
            ),
            "Object_eiffel": (
                "La Torre Eiffel\n\n"
                "Ubicación: París, Francia\n\n"
                "Altura: 324 metros\n\n"
                "Descripción: La Torre Eiffel es una torre de hierro forjado situada en el Campo de Marte.
                "Es el monumento más emblemático de la ciudad y uno de los más famosos del mundo."
```

```python
    def check_proximity(self):
        cam_pos = self.camera.position
        threshold = 20.0  # Define un umbral de proximidad

        for key, pos in self.positions.items():
            distance = ((cam_pos[0] - pos[0]) ** 2 + (cam_pos[1] - pos[1]) ** 2 + (cam_pos[2] - pos[2]) **
            if distance < threshold:
                if self.current_key != key:
                    self.current_key = key
                    self.show_info_dialog(key)
                return

        self.current_key = None

    def show_info_dialog(self, key):
        root = Tk()
        root.withdraw()  # Ocultar la ventana principal de Tkinter
        if messagebox.askyesno("Información", "¿Desea ver la información sobre este objeto?"):
            info_window = Toplevel(root)
            info_window.title("Información")
            info_window.geometry("400x400")
            info_window.configure(bg='white')
            info_window.overrideredirect(True)  # Quita la barra de título

            font_body = tkFont.Font(family="Helvetica", size=12)

            info_label = Label(info_window, text=self.info[key], font=font_body, bg='white', fg='blue', wr
            info_label.pack(fill='both', expand=True)
```

También es importante destacar que se utilizó la librería tkinter de python ,la cual fue una gran solución para la creación de esto ,por las herramientas eficaces que posee para el diseño de ventanas y darle estilos y posicionamiento, sintiéndose como trabajar con HTML.

Conclusión

World legacy se logró desarrollar utilizando algunas de las herramientas mencionadas como Python + OpenGL, además de lograr aplicar técnicas aprendidas a lo largo del semestre. Esto permitió desarrollar un entorno 3D el cual destaca por su interacción y valor cultural, permitiendo la accesibilidad a distintos monumentos y esculturas.

Bibliografía

Pygame - https://www.pygame.org/docs/

ModernGL - https://moderngl.readthedocs.io/en/5.10.0/

Numpy - https://numpy.org/doc/1.26/

Matplotlib - https://matplotlib.org/stable/index.html

PyWavefront - https://pypi.org/project/PyWavefront/

Torre Eiffel - https://free3d.com/es/modelo-3d/tour-eiffel-237685.html

Big ben - https://free3d.com/es/modelo-3d/-big-ben-v2--15945.html

San Basilio - https://free3d.com/es/modelo-3d/saint-basil-cathedral-v2--40415.html

Coliseo - https://free3d.com/es/modelo-3d/colosseum-v1--937409.html

Esfinge - https://free3d.com/es/modelo-3d/-egypt-sphinx-v2--884713.html

Cabezas Moai - https://free3d.com/es/modelo-3d/moai-v3--808476.html

Estatua de Venus - https://free3d.com/es/modelo-3d/statue-v1--541832.html

David de Miguel Angel - https://free3d.com/es/modelo-3d/statue-v1--372946.html

Torre Inclinada de Pisa - https://free3d.com/es/modelo-3d/-pisa-tower-v1--434896.html

Palmera - https://free3d.com/es/modelo-3d/-palm-tree-v1--283200.html

Pedestal - https://free3d.com/3d-model/-square-pedastal--241087.html

Banco - https://free3d.com/3d-model/concretebench--425396.html

Museo - https://sketchfab.com/3d-models/vr-gallery-showcase-presentation-building-6dc365ac66884e5aa51f8302c7dc299d