Doom PyEngine: Doom-Inspired Raycasting Simulation Using Object-Oriented Programming in Python

A Final Project
presented to the faculty of the
School of Arts, Science and Technology
National Teachers College
Quiapo, Manila

In Partial Fulfillment of the Requirements
in Object-Oriented Programming
for the Degree of
Bachelor of Science in Information Technology

Submitted by:
Marco Antonio B. Tecson

Submitted to:
Ms. Sittie Jehan M. Panda-ag

June 2025

Introduction

This project, titled “Doom-Inspired Raycasting Simulation Using Object-Oriented Programming in Python,” is a 2.5D rendering simulation that draws inspiration from the original Doom game released in 1993. Its primary goal is to recreate the core rendering logic of Doom using a modern programming mindset—specifically through object-oriented programming (OOP) principles. The system is broken into modular classes such as Player, Renderer, Engine, Map, and AssetManager, each handling a specific responsibility within the simulation. This allows for a cleaner, more organized codebase that can be easily expanded or modified, compared to the monolithic C-style structure of the original engine. By structuring the logic this way, the project not only recreates Doom's look and feel but also teaches solid software design.

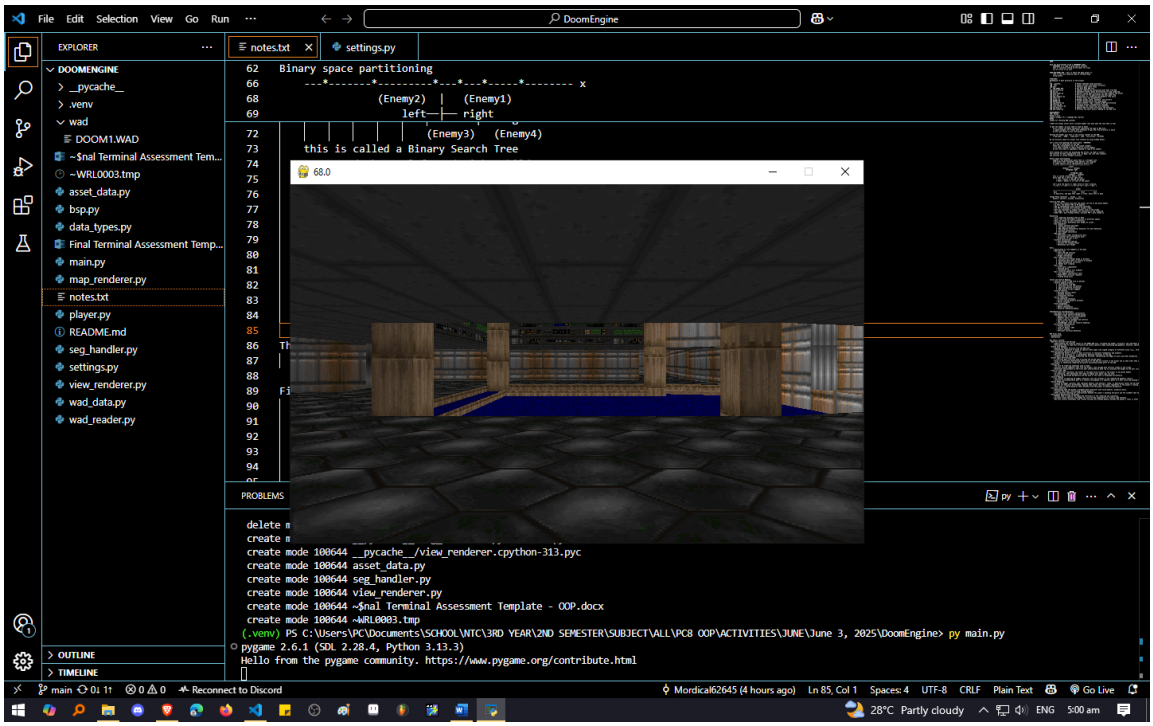
The original Doom engine was written in C and closely interacted with hardware-level memory and performance management. While powerful, such low-level code can be difficult to understand for new developers. This project takes the opposite approach—using Python, a high-level language, and emphasizing OOP to make the system easier to follow. Each component is encapsulated within a class, using methods and attributes to keep behavior consistent and manageable. For example, the Renderer class handles raycasting and field of view logic, while the Player class manages movement, position, and camera angle—all separated cleanly to follow the single responsibility principle.

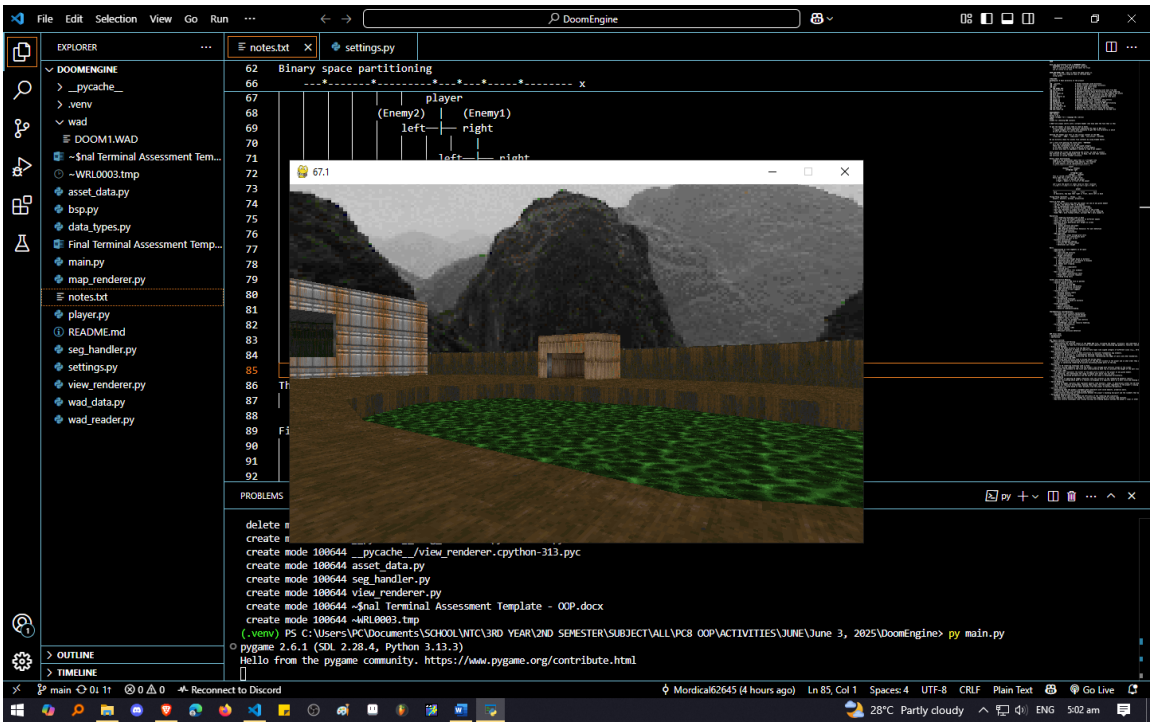
Using OOP not only simplifies code reuse and debugging but also makes it easier to explain complex concepts like raycasting, BSP (Binary Space Partitioning), and texture mapping. When each feature is modeled as its own object, it's easier to visualize how parts of a game engine interact—just like in real-world game development. The modular structure also opens doors for future expansion, such as adding enemy sprites, animation systems, or physics. By showing that a game engine inspired by Doom can be recreated in Python with an OOP mindset, this project serves as both a technical demo and a learning tool. Ultimately, it proves that advanced game concepts can be broken down into simpler, reusable parts through the power of object-oriented programming.

Project Description

This project is a Python-based simulation that recreates the rendering engine of the classic first-person shooter game, Doom. It focuses on implementing the revolutionary raycasting technique that provided the illusion of a 3D environment from 2D map data in the original game. By parsing the original DOOM1.WAD file, the simulation accesses and utilizes authentic game assets like maps, textures, and sprites. The goal is to build a functional engine capable of displaying Doom levels and allowing basic player navigation within them. This endeavor serves as an educational exercise in graphics programming, binary file handling, and game development principles using object-oriented programming in Python.

The system incorporates several key features and functions to achieve its goals. It includes a robust WAD file reader and data handler to extract vital map geometry, texture information, and other assets directly from the original game file. The core functionality lies in the raycasting and BSP rendering pipeline, which processes the map data to draw the environment from the player's perspective with correct depth and texture mapping. A player module manages movement, rotation, and basic collision detection against the rendered world geometry. Supporting features include custom data structures for efficient data handling and the planned use of Numba for optimizing performance-critical sections of the rendering code. This combination of features allows the project to simulate the fundamental visual and interactive experience of classic Doom.


File Edit Selection View Go Run
DoomEngine
EXPLORER
notes.txt
settings.py
DOOMENGINE
__pycache__
.venv
wad
DOOM1.WAD
~$nal Terminal Assessment Tem...
~WRL0003.tmp
asset_data.py
bsp.py
data_types.py
Final Terminal Assessment Temp...
main.py
map_renderer.py
notes.txt
player.py
README.md
seg_handler.py
settings.py
view_renderer.py
wad_data.py
wad_reader.py
Binary space partitioning
(Enemy2) | (Enemy1)
left—┼— right
(Enemy3) (Enemy4)
this is called a Binary Search Tree
68.0
PROBLEMS
create mode 100644 __pycache__/view_renderer.cpython-313.pyc
create mode 100644 asset_data.py
create mode 100644 seg_handler.py
create mode 100644 view_renderer.py
create mode 100644 ~$nal Terminal Assessment Template - OOP.docx
create mode 100644 ~WRL0003.tmp
(.venv) PS C:\Users\PC\Documents\SCHOOL\NTC\3RD YEAR\2ND SEMESTER\SUBJECT\ALL\PC8 OOP\ACTIVITIES\JUNE\June 3, 2025\DoomEngine> py main.py
pygame 2.6.1 (SDL 2.28.4, Python 3.13.3)
Hello from the pygame community. https://www.pygame.org/contribute.html
OUTLINE
TIMELINE
main
Reconnect to Discord
Mordical62645 (4 hours ago)
Ln 85, Col 1
Spaces: 4
UTF-8
CRLF
Plain Text
Go Live
28°C Partly cloudy
ENG
5:00 am


File Edit Selection View Go Run
DoomEngine
EXPLORER
notes.txt
settings.py
DOOMENGINE
Binary space partitioning
player
(Enemy2) | (Enemy1)
left—┼— right
67.1
PROBLEMS
create mode 100644 __pycache__/view_renderer.cpython-313.pyc
create mode 100644 asset_data.py
create mode 100644 seg_handler.py
create mode 100644 view_renderer.py
create mode 100644 ~$nal Terminal Assessment Template - OOP.docx
create mode 100644 ~WRL0003.tmp
(.venv) PS C:\Users\PC\Documents\SCHOOL\NTC\3RD YEAR\2ND SEMESTER\SUBJECT\ALL\PC8 OOP\ACTIVITIES\JUNE\June 3, 2025\DoomEngine> py main.py
pygame 2.6.1 (SDL 2.28.4, Python 3.13.3)
Hello from the pygame community. https://www.pygame.org/contribute.html
OUTLINE
TIMELINE
main
Reconnect to Discord
Mordical62645 (4 hours ago)
Ln 85, Col 1
Spaces: 4
UTF-8
CRLF
Plain Text
Go Live
28°C Partly cloudy
ENG
5:02 am

Objectives

- To apply object-oriented programming principles for modular engine design.
- To simulate the rendering engine of classic Doom using Python.
- To implement a raycasting system that creates a 3D illusion from 2D map data.
- To read and extract game assets (maps, textures, sprites) from the DOOM1.WAD file.
- To render authentic Doom levels with proper depth and texture mapping.
- To enable basic player navigation, including movement and rotation.
- To explore Binary Space Partitioning (BSP) for efficient rendering.
- To provide an educational tool for learning game engine concepts in Python.

Target Users

- **Aspiring game developers** – who want to understand how early 3D game engines like Doom were built.
- **High-level programmers** – looking to apply their skills in game development and graphics programming.
- **Computer science and IT students** – especially those from The National Teachers College, who are studying topics like OOP, data structures, and graphics programming.
- **Tech hobbyists and tinkerers** – interested in how classic games work under the hood.
- **Educators and mentors** – who need a hands-on example to teach binary file handling, OOP, or rendering concepts.
- **Graphics programming learners** – especially those looking for an approachable intro before moving to OpenGL or Vulkan.
- **Retro game enthusiasts** – curious about how old-school games like Doom were technically possible.
- **Developers exploring engine architecture** – who want to study and build lightweight, customizable engines in Python.

Scope and Limitations

Scope

- Recreate Doom’s core rendering engine using Python.
- Parse DOOM1.WAD to extract maps, textures, and sprites.
- Implement raycasting with BSP for pseudo-3D rendering.
- Include:
- Textured wall rendering
- Player movement and rotation
- Basic collision detection

Limitation

- No full game logic (AI, weapons, pickups, progression).
- No advanced rendering (floor/ceiling textures, dynamic lighting).
- Limited to DOOM1.WAD support
- May not match original Doom's performance and visual precision.

Methodology

Topics

- WAD File Structure and Parsing
- Binary File Reading
- Data Structures (Vertices & Linedefs)
- Binary Space Partitioning (BSP)
- Raycasting
- Field of View (FOV)
- Texture Mapping
- Sprite Rendering
- Collision Detection
- Performance Optimization Techniques
- Object-Oriented Programming (OOP)
    - Encapsulation: This is evident in the design where data (like player position, map vertices, texture data) and the methods that operate on that data (like moving the player, rendering the map, loading assets) are bundled together within distinct classes (Player, Map, AssetManager, Renderer). This helps in managing complexity and protecting data by controlling access.

    - Abstraction: The project uses abstraction by modeling different aspects of the game as simplified objects. For example, the Player class abstracts the complex details of player state and movement into a manageable object with properties like position and orientation, and methods for movement and rotation. Similarly, Map abstracts the level geometry and structure.

Tools

- Slade 3
- VSCode (IDE)

Software

- Pygame
- Numba
- Python 3.13.3

Programming Language

- Python

Output

The project culminates in a functional Doom-inspired raycasting engine implemented in Python. The final product is a 3D-like game engine that can render and display levels from the original Doom game using the DOOM1.WAD file. The engine successfully recreates the iconic visual style and gameplay mechanics of the classic game, allowing users to navigate through the game world in first-person perspective.

Visit my project GitHub link for the codes of this engine: https://tinyurl.com/33m6wac3

Main Features for Defense/Submission:

1. **WAD File Integration**
   - **wad_reader.py**: Core file for reading and parsing the WAD file
   - **wad_data.py**: Handles WAD data structures and management
   - **asset_data.py**: Processes and manages game assets from WAD
   - **data_types.py**: Defines data structures for WAD elements

2. **3D Rendering System**
   - **view_renderer.py**: Implements raycasting and rendering
   - **bsp.py**: Implements Binary Space Partitioning

- **seg_handler.py**: Handles segment rendering and clipping
- **map_renderer.py**: Manages map visualization

3. **Player Movement and Controls**
   - **player.py**: Handles player movement, rotation, and physics
   - **settings.py**: Contains player-related constants (speed, height, etc.)

4. **Visual Features**
   - **view_renderer.py**: Texture mapping and sprite rendering
   - **asset_data.py**: Texture and sprite management
   - **seg_handler.py**: Wall and portal rendering

5. **Technical Implementation**
   - **main.py:** Core engine class and game loop
   - **settings.py**: Global settings and constants
   - All files demonstrate OOP principles with clear class structures

6. **Interactive Demo**
   - **main.py:** Game loop and event handling
   - **player.py**: Player controls and movement
   - **view_renderer.py**: Real-time rendering
   - **bsp.py**: Efficient rendering optimization

- **Customizable Player Speed and Rotation Sensitivity**
  - The engine now prompts the user to input player movement speed and rotation sensitivity before the simulation starts.
  - Input values are relative to 50, which represents normal speed/sensitivity.
  - Example: 100 = 2x speed, 25 = 0.5x speed.
  - Invalid inputs (non-numeric or ≤ 0) are gracefully handled, and users are re-prompted until a valid number is entered.
  - These runtime parameters enhance user experience by allowing flexible testing, debugging, and personalized control responsiveness.

Conclusion

This project set out to recreate the core rendering engine of the classic Doom game using modern object-oriented programming (OOP) principles in Python. The original Doom engine, while groundbreaking, was built using low-level C code that can be difficult to understand and modify. By contrast, this simulation takes a high-level, modular approach to break down complex graphics programming concepts into manageable, reusable components such as Player, Renderer, Map, and AssetManager.

Through this project, we addressed the challenge of simulating 2.5D rendering using raycasting and Binary Space Partitioning (BSP), while reading and utilizing authentic assets from the original DOOM1.WAD file. The engine successfully displays Doom-like levels and allows interactive navigation, effectively capturing the visual essence and logic of the original game in a more accessible format.

Beyond the technical implementation, this project has been an invaluable learning experience. It deepened my understanding of OOP, data encapsulation, abstraction, graphics programming, binary file handling, and real-time rendering techniques. The process of translating low-level game architecture into high-level Python code provided hands-on insight into how classic game engines operate under the hood.

The value of this project lies in its dual nature: it serves both as a tribute to retro game development and as an educational tool for aspiring developers. Whether you're a student, hobbyist, or educator, this simulation offers a simplified yet faithful model of how a game engine can be designed using modern programming techniques.

Ultimately, this project demonstrates that even complex systems like Doom's rendering engine can be reimagined using OOP in Python—making advanced game development concepts more approachable, modular, and easier to understand.