**DON’T FORGET TO CREDIT:**

**STANISLAV PETROV – PYTHON IMPLEMENTATION**

**AMRO IBRAHIM – CORE LOGIC**

1. Set up the basic project structure

* **settings.py**: Contains game constants and configuration
* **README.md**: Project documentation

**Core libraries**

* + **Pygame** For graphics and input handling
  + **numpy** For efficient array operations
  + **numba** For performance optimization
  + **math** For mathematical calculations
  + **struct** Binary data handling
  + **Vector2** 2D vectors
  + **random** Random number generation
  + **sys** System operations

1. Implemented WAD file reading

* **wad\_reader.py:**

Class: **WADReader**

Key methods:

**read\_header:**

Reads the 12-byte WAD header

Returns: wad\_type, lump\_count, init\_offset

**read\_directory:**

Reads the WAD file directory

Returns: list of lump information

**read\_vertex:**

Reads vertex coordinates (x, y)

Returns: Vector2 position

**read\_linedef:**

Reads wall definitions

Returns: Linedef object

**read\_sector**:

Reads sector information

Returns: Sector object

* **wad\_data.py:**

Class: **WADData**

Key methods:

**load\_map:**

Loads a specific map from the WAD file

**load\_textures:**

Loads texture data from the WAD file

**load\_sprites:**

Loads sprite data from the WAD file

1. Created data structures for game elements

* **data\_types.py**

Class: **TextureMap**

Stores texture information

'name', 'flags', 'width', 'height', 'patch\_count', 'patch\_maps'

Class: **Sector**

Represents a map sector

'floor\_height', 'ceil\_height', 'floor\_texture', 'ceil\_texture', 'light\_level'

Class: **Linedef**

Represents a wall line

'start\_vertex\_id', 'end\_vertex\_id', 'flags', 'line\_type', 'sector\_tag'

Class **Node**

BSP tree node

'x\_partition', 'y\_partition', 'dx\_partition', 'dy\_partition', 'bbox'

1. Implemented Binary Space Partitioning for efficient rendering

* **bsp.py**

Class **BSP**:

Key Methods:

**render\_bsp\_node:**

Traverses the BSP tree for rendering

Parameters: node\_id - current node to process

**is\_on\_back\_side:**

Determines if player is behind a partition

Parameters: node - BSP node to check

**check\_bbox:**

Checks if bounding box is in view

Parameters: bbox - bounding box to check

**render\_sub\_sector:**

Renders a subsector

Parameters: sub\_sector\_id - ID of subsector to render

1. Added player movement and controls

* **player.py**

Class: **Player**

Key Methods:

**control:**

Handles player input and movement

**Uses pygame.key.get\_pressed()** for input

**get\_height:**

Manages player height and floor interaction

Updates player position relative to floor

**update:**

Updates player state

Calls **control()** and **get\_height()**

1. Implemented the rendering system

* **view\_renderer.py**

class **ViewRenderer**:

Decorators:

**@staticmethod**

* A static method is a method that belongs to the class rather than an instance of the class
* Doesn't require self parameter
* Can't access or modify class/instance state
* Can be called without creating an instance
* Used for utility functions that don't need class data

**@njit**

* Numba's Just-In-Time (JIT) compiler decorator
* Converts Python code to machine code for faster execution
* Significantly improves performance of numerical computations
* Works best with loops and array operations
* Has some limitations on supported Python features

Key Methods:

**Static njit draw\_column:**

Draws a vertical line

Parameters: framebuffer, x, y1, y2, color

**draw\_flat:**

Draws floor/ceiling

Parameters: texture ID, lighting, coordinates

**draw\_wall\_col:**

Draws a wall column

Parameters: framebuffer, texture, coordinates, lighting

* **map\_renderer.py**

**Class: MapRenderer:**

Key Methods:

**draw\_linedefs:**

Draws wall lines

**draw\_player\_pos:**

Draws player position and FOV

**draw\_vertexes:**

Draws map vertices

* **seg\_handler.py**

Class: **SegHandler**

Key Methods:

**draw\_solid\_wall\_range:**

Renders solid walls between x1 and x2

Handles texture mapping and lighting

**draw\_portal\_wall\_range:**

Renders portal walls (windows, doors)

Handles different ceiling/floor heights

**scale\_from\_global\_angle:**

Calculates wall scaling based on distance and angle

Used for perspective correction

**clip\_portal\_walls:**

Clips portal walls to prevent rendering errors

Handles overlapping walls

**classify\_segment:**

Classifies segments for rendering

Determines if wall is solid or portal

* **asset\_data.py**

Class: **AssetData**:

Key Methods:

**load\_textures:**

Loads wall and floor textures

Handles texture mapping

**load\_sprites:**

Loads game sprites

Manages sprite animations

**load\_palette:**

Loads color palette

Handles color mapping

1. Added map switching and game state management

* **main.py**

Class: **DoomEngine:**

Key Methds:

**change\_map:**

Switches to a different map

Parameters: map\_name - name of map to load

**run:**

Main game loop

Handles events, updates, and rendering

**check\_events:**

Handles user input and events

Includes map switching and exit

**update:**

Updates game state

Calls update methods for all components

* **GameLauncher.java**

Class: GameLauncher extends Jframe

Key Methods:

**launchGame:**

Launches the Python game

Handles input validation

Manages process creation

**GameLauncher:**

Creates GUI window

Sets up input fields

Initializes buttons

**validateInputs:**

Validates user inputs

Shows error messages if invalid

* **launch\_game.py**

Additional methods:

**get\_player\_speed:**

Gets player speed from user input

Validates input values

**get\_rotation\_sensitivity:**

Gets rotation sensitivity from user input

Validates input values

* **if \_\_name\_\_ == '\_\_main\_\_':**

Main entry point

Handles command-line arguments

Launches game with specified parameters