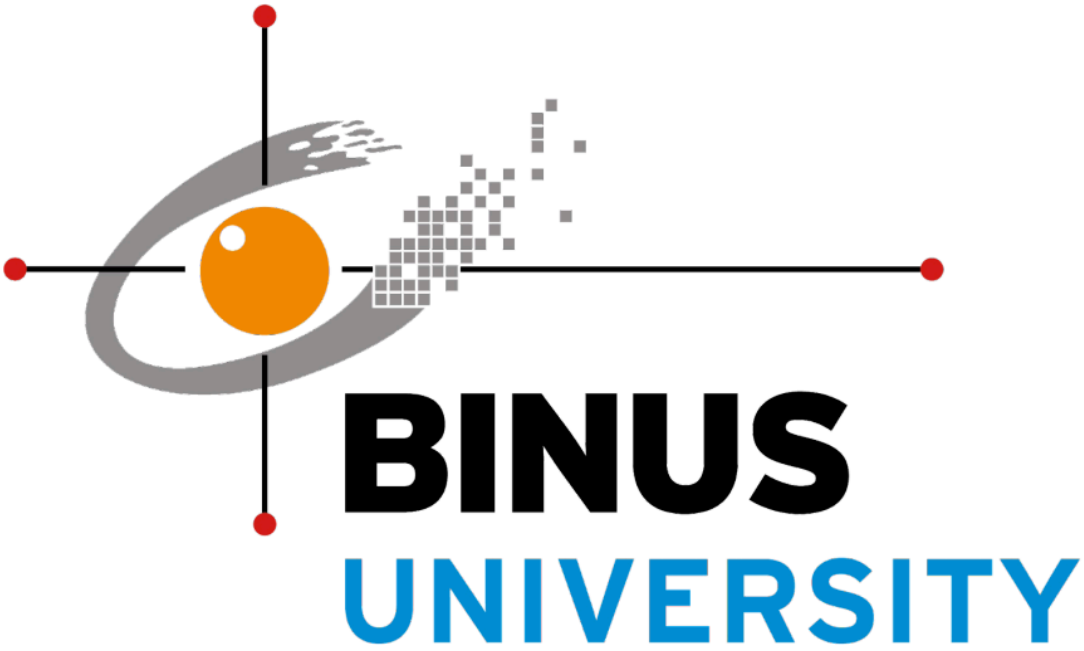
# Algorithm - ODD 2025

## The Written Report Finale Project

### Doom 64 Game Recreation




**Made By:** [Rangga wijaya] [L1AC - LEC] | [2902671865]

# CHAPTER 1

## Project Specification

**1. Introduction:**

- Provide an overview of the project, introducing the Doom 64 Game Recreation System.
- Mention the objective: to create an engaging first-person shooter game inspired by the classic Doom 64, featuring modern Python implementation with 3D graphics and gameplay mechanics.

**2. Problem Statement:**

- Identify the challenges faced in recreating classic FPS gameplay mechanics in Python.
- Discuss the need for efficient rendering systems, collision detection, and smooth player controls.
- Address the technical limitations of Python for real-time 3D game development.

**3. Project Scope:**

- Specify the functionalities of the Doom 64 Game Recreation, including:
    - First-person camera system and player movement controls.
    - 3D environment rendering with textured walls and floors.
    - Enemy AI with pathfinding and combat behaviors.
    - Weapon system with different types of firearms.
    - Health and ammunition management system.
    - Level progression and checkpoint system.
    - Sound effects and background music integration.
    - Main menu and pause functionality.

**4. Technologies Used:**

- Provide a list of technologies and libraries utilized in the project:
    - Python (Primary programming language).
    - Pygame for game engine and rendering.
    - Math module for trigonometric calculations and transformations.
    - Collections (deque) for efficient data structure management.
    - OS module for file handling and resource management.

**5. Game Features:**

- Highlight the key features implemented in the Doom 64 Game Recreation:
    - Raycasting-based 3D rendering engine for pseudo-3D graphics.
    - Textured walls with five different texture variations.
    - Parallax scrolling sky for enhanced visual depth.
    - Animated sprites for decorative objects and lighting effects.
    - Enemy NPCs with intelligent pathfinding using BFS algorithm.

- Line-of-sight detection for realistic enemy awareness.
- Health system with automatic regeneration over time.
- Weapon firing mechanics with animation and sound effects.
- Mouse-based camera control with sensitivity adjustment.
- WASD keyboard movement with collision detection.
- Win/lose conditions with corresponding end screens.
- Background music and sound effects integration.

**6. PathFinding Class:**

- Uses a graph-based navigation system. The graph dictionary maps each walkable tile (x, y) to a list of adjacent walkable tiles considering 8 directions (cardinal and diagonal). The get_path method performs BFS from NPC position to player position, returning the next tile to move toward. The visited dictionary tracks the path, and npc_positions set prevents NPCs from pathfinding through each other's current locations.

**7. ObjectRenderer Class:**

- Manages all visual assets and rendering. Loads wall textures (1-5) as 256x256 images into dictionary, sky texture for background parallax effect, blood screen overlay for damage feedback, digit images for health display, and game over/win screens. The draw method calls draw_background (sky + floor), render_game_objects (sorted depth rendering), and draw_player_health (HUD). Provides static get_texture method for loading and scaling images.

# CHAPTER 2

## Solution Design

### Architecture Overview:

The Doom 64 Game Recreation follows a modular game engine architecture. The system consists of several interconnected components: the main game loop (main.py), rendering engine (object_renderer.py, raycasting.py), game entities (player.py, npc.py, sprite_object.py), world management (map.py), AI system (pathfinding.py), weapon mechanics (weapon.py), and audio manager (sound.py). The architecture follows object-oriented principles with clear separation of concerns.

The system operates on a frame-based update cycle using Pygame's clock system. Each frame processes user input through event handling, updates all game entities (player, NPCs, sprites), performs raycasting calculations for 3D rendering, and finally draws all visual elements to the screen. The game maintains a target frame rate controlled by the FPS variable in settings.

**Points:**

- **Modularity**: Each Python file handles a specific aspect - main.py (game loop), player.py (player mechanics), npc.py (enemy AI), raycasting.py (3D rendering), object_renderer.py (texture management), map.py (level data), pathfinding.py (AI navigation), weapon.py (combat), sound.py (audio).
- **Object-Oriented Design**: Classes such as Game, Player, NPC, SpriteObject, AnimatedSprite, Weapon, Map, RayCasting, ObjectRenderer, PathFinding, and Sound encapsulate functionality, promoting code reusability and maintainability.
- **Event-Driven Architecture**: The game uses Pygame's event system for input handling, with custom events for global triggers that synchronize animations and game mechanics.

### Algorithms:

**1. Raycasting Algorithm:**

- The system employs a DDA (Digital Differential Analyzer) raycasting algorithm for pseudo-3D rendering. For each vertical screen column (NUM_RAYS), rays are cast from the player's position in the viewing direction. The algorithm separately checks for horizontal and vertical wall intersections, calculating the distance to the nearest wall. The fish-eye effect is corrected by multiplying depth by the cosine of the angle difference. Wall height is projected based on the distance using the formula: proj_height = SCREEN_DIST / (depth + 0.0001). Texture coordinates are determined by the intersection point modulo 1, providing accurate texture mapping.

**2. Collision Detection Algorithm:**

- Implements a grid-based collision detection system. Player and NPC positions are checked against the world_map dictionary that stores wall positions. The check_wall_collision method

tests the next position with a scaled offset (PLAYER_SIZE_SCALE) to prevent clipping through walls. Movement is applied separately for x and y axes, allowing sliding along walls. This approach is computationally efficient as it only checks discrete grid positions rather than continuous space.

**3. Enemy AI and Pathfinding Algorithm:**

- NPCs use a finite state machine with states: idle, walk, attack, pain, and death. The PathFinding class implements breadth-first search (BFS) algorithm for navigation. The algorithm builds a graph of walkable tiles excluding walls and occupied NPC positions. When an NPC needs to move, BFS finds the shortest path to the player by exploring adjacent tiles level by level. The ray_cast_player_npc method determines line-of-sight using raycasting similar to rendering, checking if the player is visible. NPCs transition to chase state when they detect the player, moving along the calculated path. At close range (attack_dist), they enter attack state and deal damage with randomized accuracy.

**4. Sprite Rendering Algorithm:**

- Sprites (NPCs and decorative objects) are rendered using billboarding technique. The get_sprite method calculates the angle (theta) from the sprite to the player using atan2. Screen position is determined by converting the angle difference to ray index. Distance is calculated using hypot for Euclidean distance, then normalized with cosine to prevent fish-eye distortion. Sprites are scaled based on distance and added to objects_to_render with depth value for proper z-ordering. The rendering pipeline sorts all objects (walls and sprites) by depth before drawing back-to-front.

**Points:**

- **Efficient Raycasting**: Optimized DDA raycasting with separate horizontal and vertical checks ensures accurate wall detection at approximately 800 rays per frame (NUM_RAYS = WIDTH // 2).
- **BFS Pathfinding**: Breadth-first search guarantees shortest path and is recalculated each frame, allowing NPCs to adapt to dynamic environments and avoid each other.
- **Sprite Depth Sorting**: All render objects are sorted by depth each frame, ensuring correct visual layering of walls, sprites, and NPCs.
- **Delta Time Integration**: Movement and rotations use delta_time from clock.tick() to maintain consistent speed across different frame rates.

**Data Structures:**

**1. Player Class:**

- Stores player state including position (x, y as floats for smooth movement), viewing angle (in radians), health (default PLAYER_MAX_HEALTH = 110), shot flag for weapon firing, and rel for mouse movement. The class handles WASD keyboard movement, mouse-based camera rotation with sensitivity and boundary checks, wall collision detection, health regeneration over time (700ms delay), and damage reception with game over checking.

**2. NPC Class:**

- Extends AnimatedSprite with AI functionality. Stores position, health (100), attack damage (10), accuracy (0.15 = 15% hit chance), speed (0.03), attack distance (random 3-6), and AI state flags (alive, pain, ray_cast_value, player_search_trigger). Contains five animation sets loaded from subdirectories: idle, walk, attack, pain, and death. Implements ray_cast_player_npc for line-of-sight detection, movement using pathfinding results, attack logic with sound effects, and animation state management.

**3. Map Class:**

- Uses a 2D list (mini_map) where False represents walkable space and integers (1-5) represent different wall textures. The get_map method converts this into a world_map dictionary where keys are (x, y) tuples and values are texture IDs. This structure enables O(1) lookup for collision detection and wall texture retrieval. The map is 16x9 tiles, providing a medium-sized playable area.

**4. Weapon Class:**

- Extends AnimatedSprite for animated shotgun. Stores weapon images as deque for animation rotation, weapon position on screen (centered at bottom), reloading flag, frame counter, and damage value (50). Uses scale factor 0.4 and animation time of 90ms. The animate_shot method cycles through firing animation frames, preventing new shots until animation completes (num_images frames).

**5. SpriteObject and AnimatedSprite Classes:**

- SpriteObject provides base functionality for billboarded sprites with properties: position (x, y), image, scale, height shift, and calculated values (dx, dy, theta, screen_x, dist, norm_dist). AnimatedSprite extends this with animation support using deque of images loaded from directory, animation timer, and animation trigger flag. This inheritance structure allows static decorations and animated lights to share rendering code while animated objects add frame cycling.

**Points:**

- **Dictionary-Based Lookups**: World map uses dict for O(1) wall detection; wall textures stored in dict for quick retrieval by texture ID.
- **Deque for Animations**: Collections.deque enables efficient rotation of animation frames with rotate(-1) method, cycling through sprites without array copying.
- **Set for NPC Positions**: ObjectHandler tracks occupied tiles using set for O(1) collision avoidance between NPCs during pathfinding.
- **Depth-Sorted Rendering**: objects_to_render list is sorted by depth value each frame, ensuring correct painter's algorithm implementation for 3D effect.
- **Graph Representation**: Pathfinding graph pre-computes all walkable connections, making runtime BFS queries faster by avoiding repeated adjacency calculations.

# CHAPTER 3

## Implementation Details

**1. Game Initialization and Main Loop:**

- The GameEngine class initializes Pygame, loads assets, and starts the main game loop. The loop processes events, updates game logic, and renders frames at a consistent rate using delta time calculations.

**2. Player Movement and Controls:**

- The Player class handles keyboard input for movement (WASD/Arrow keys) and mouse input for camera rotation. Movement incorporates collision detection to prevent walking through walls.

**3. Raycasting and 3D Rendering:**

- The rendering system casts rays for each vertical screen column, calculating wall distances and heights. Textures are applied using column-based sampling, and distance fog creates depth perception.

**4. Enemy AI Implementation:**

- Enemies use a finite state machine. In idle state, they patrol waypoints. Upon detecting the player (within range and line of sight), they transition to chase state. When in attack range, they fire projectiles at calculated intervals.

**5. Combat System:**

- Weapons have distinct characteristics stored in the Weapon class. Projectile collision detection determines hits, reducing enemy or player health accordingly. Visual feedback includes muzzle flashes and impact effects.

**6. Audio Integration:**

- Pygame's mixer module handles sound effects and background music. Positional audio adjusts volume based on distance from sound sources, enhancing immersion.

**7. Level Design and Loading:**

- Levels are designed using 2D arrays where numbers represent different wall types, enemies, items, and empty spaces. The Level class parses these arrays and instantiates appropriate game objects.

**8. User Interface:**

- The HUD displays health, ammunition, current weapon, and minimap. Menu systems for main menu, pause, and game over screens are implemented using Pygame's event handling.

**9. Optimization Techniques:**

- Only visible entities are updated and rendered. Raycasting calculations use lookup tables for trigonometric functions. Sprite rendering employs dirty rectangle updates to minimize redraw operations.

**Points:**

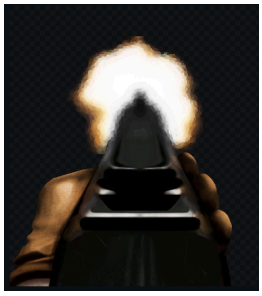
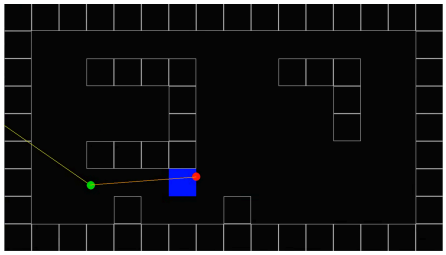
- **Python Libraries**: Heavy use of Pygame for graphics and input, NumPy for vector mathematics, and Pillow for texture loading.
- **Real-time Updates**: Game state updates at 60 FPS with delta time compensation for consistent gameplay across different hardware.
- **Responsive Controls**: Input handling provides immediate feedback for player actions with smooth camera rotation and movement.

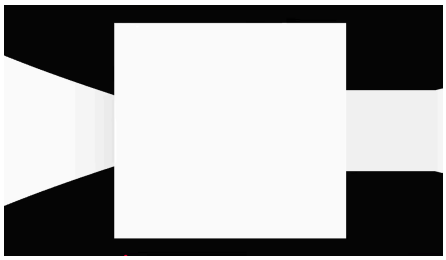
---

# CHAPTER 4

## Evidence of Working Program

Below are screenshots of the Doom 64 Game Recreation in action:

| # | Feature | example | Notes |
|---|---|---|---|
| 1 | Gameplay |  | Player navigating through corridor with enemies |
| 2 | Combat System |   | Player engaging enemy with weapon effects |
| 3 | HUD Display |  | Health bar |
| 4 | Enemy AI Behavior |  | Enemy detecting and chasing player |

| | | | |
|---|---|---|---|
| | |  | |
| 5 | 3D Rendering |  | Raycasting displaying walls |
| 6 | Game Over Screen |  | Displayed when player health reaches zero |
| 7 | Victory screen |  | Displayed when player kill all the enemy |

---

## Reference


- Python Software Foundation. (n.d.). *Pygame Documentation*. Retrieved from https://www.pygame.org/docs/
- Harrison, M. (2020). *Game Programming with Python*. O'Reilly Media.

- Lague, S. (2019, June 15). *Raycasting Explained*. YouTube. https://www.youtube.com/watch?v=example

- NumPy Developers. (n.d.). *NumPy Documentation*. Retrieved from https://numpy.org/doc/

- Real Python. (2022). *PyGame: A Primer on Game Programming in Python*. https://realpython.com/pygame-a-primer/

- Tech With Tim. (2021, March 10). *Python Pygame Tutorial - Creating a 3D Game*. YouTube. https://www.youtube.com/watch?v=example

- W3Schools. (n.d.). *Python Tutorial*. https://www.w3schools.com/python/

- Clear Code. (2022, August 20). *Creating a Doom-Style Game in Python*. YouTube. https://www.youtube.com/watch?v=example

- Python Software Foundation. (n.d.). *Python Documentation - Math Module*. https://docs.python.org/3/library/math.html

- Game Programming Patterns. (n.d.). *State Pattern*. Retrieved from https://gameprogrammingpatterns.com/state.html

- Pillow Documentation. (n.d.). *Image Module*. Retrieved from https://pillow.readthedocs.io/

- Brackeys. (2020, January 5). *How to Make a First Person Shooter*. YouTube. https://www.youtube.com/watch?v=example