TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

A Project Report

On

A Minecraft Clone using Ursina Engine

Submitted By:

Shrijan Dahal (HCE081BEI042)

Submitted To:

Department of Electronics and Computer Engineering

Himalaya College of Engineering

Chyasal,Lalitpur

Feb-22,2025

ACKNOWLEDGEMENT

We express our sincere gratitude to all those who have supported and guided us throughout the process of developing this Minecraft clone game using Python and the Ursina engine. This endeavor would not have been possible without their valuable contributions and assistance.

The authors extend their heartfelt thanks to our advisor, Sushant Pandey, whose expertise and guidance have been instrumental in shaping the direction of this project. Their insightful feedback and continued encouragement have been invaluable in refining the scope and focus of this game development.

We also, would like to thank our friends and colleagues for their meaningful discussions, brainstorming sessions, and moral support. Your input has enriched the quality of this project and helped us navigate challenging concepts.

This work would not have been possible without the collective efforts of these individuals. Although any shortcomings in this game are solely our responsibility, their contributions have significantly enriched its content.

Thank you.

Shrijan Dahal (HCE081BEI042)

ABSTRACT

This project focuses on developing a Minecraft clone game using Python and the Ursina game engine. The game features a player-controlled character in a voxel-based world, multiple block types, interactive building and breaking mechanics, a hotbar for tool selection, and world generation. The primary goal of the project is to create a fun, engaging, and interactive game that simulates the Minecraft experience with block placement, destruction, and exploration. The game includes collision detection, a menu system, asset management for textures and models, and a first-person camera that allows for immersive gameplay.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

Computer Graphics (CG) enables the creation and display of interactive visual worlds. This report presents a mini project titled **Minecraft Clone using Python and Ursina**, which builds a voxel-based environment with block placement, destruction, and real-time exploration. The project demonstrates core concepts in 3D scene construction, asset management, and user interaction through a first-person perspective.

1.1 Background Introduction

At a high level, game engines manage rendering, input, and scene updates to create interactive experiences. In a voxel game, the world is composed of discrete blocks arranged in a grid, and player actions modify the scene by placing or breaking blocks. Using the Ursina engine simplifies rendering and input handling while allowing the project to focus on gameplay mechanics, world generation, and user interface elements like menus and hotbars [1].

1.2 Motivation

Many beginner projects rely on prebuilt assets without understanding how interactive worlds are structured. This project is motivated by the need to build intuition about how a 3D scene is constructed, how player input affects the world, and how simple systems like inventory selection and block interaction create engaging gameplay.

1.3 Objectives

The main objectives of the project are listed below:

- Create a voxel-based world with multiple block types and textures.
- Implement block placement and block breaking with mouse input.
- Provide a hotbar and selection system for block types.
- Build a menu and first-person camera system for interaction and navigation.

1.4 Scope

This project covers fundamental elements of a simple 3D game:

- Voxel world generation and block-based interaction
- First-person movement and camera control
- Basic UI elements (menu, hotbar)

- Asset usage for models, textures, and audio

It does not include advanced lighting, procedural terrain algorithms, AI entities, or multiplayer networking. These are suggested as future enhancements.

2. LITERATURE REVIEW

2.1 Voxel-Based Rendering

Voxel environments represent 3D space as discrete cubic blocks arranged in a grid. This simplifies collision detection, world modification, and physics calculations compared to traditional polygon-based rendering. Voxel systems enable straightforward block placement through grid coordinates.

2.2 3D Transformations and Scene Management

3D transformations use matrix operations for camera positioning and object placement:

$$\mathbf{p}' = \mathbf{TRSp}$$

where transformation matrices handle translation, rotation, and scaling. Scene graphs organize entities hierarchically, facilitating coordinate transformations and first-person camera control.

2.3 Input Systems and Ray Casting

Interactive 3D environments require converting 2D mouse input to 3D world coordinates. Ray casting projects from the camera through the cursor to determine target voxels, enabling block placement and destruction.

2.4 Game Development Frameworks

Ursina provides a Python-based framework built on Panda3D, offering simplified APIs for rapid prototyping. It includes entity systems, first-person controllers, and streamlined asset loading, making it suitable for educational voxel games [1, 2]. Alternative frameworks like Unity offer more features but require greater complexity.

3. METHODOLOGY

3.1 System Overview

The project is organized into four main modules:

- **Entry Module (`main.py`):** Initializes the Ursina application, configures the window, triggers world generation, and runs the main game loop.
- **Classes Module (`src/classes.py`):** Defines all game entities — `Voxel`, `Sky`, `Menu`, and `Hotbar` — and centralizes asset loading.
- **Asset Pipeline:** Textures (`.png`), 3D models (`.obj`), and audio (`.wav`) are loaded once at startup and shared across all entities. Block textures and overall visual style are inspired by Minecraft [3].
- **Input and Interaction:** Mouse clicks and keyboard keys are handled per-entity through Ursina's `input()` callback and the global `held_keys` dictionary.

3.2 World Generation

The game world is a flat voxel terrain built procedurally on startup. The `setup_world()` function iterates over a 40×40 horizontal grid and stacks four vertical layers:

Y Layer	Block Type	Texture	Breakable
0	Bedrock	Stone	No
1–3	Soil	Dirt	Yes
4	Surface	Grass	Yes

Table 3.1: Voxel layer composition for the generated world

The grid spans $x \in [-20, 19]$ and $z \in [-20, 19]$, producing $40 \times 40 \times 5 = 8{,}000$ voxel entities in total (one bedrock, three dirt, and one grass per column). Each `Voxel` is placed at a scale of 0.5 world units and receives a slight random brightness variation via `color.color(0, 0, random.uniform(0.9, 1))` to break visual monotony.

3.3 Class Design

3.3.1 Voxel

`Voxel` extends Ursina's `Button`, which provides built-in hover detection and mouse-event dispatch. Its constructor accepts a world-space `position`, a pre-loaded `texture` object, and a `breakable` flag. Block interaction is handled in the `input()` method:

- **Left click** — plays the build sound and spawns a new `Voxel` at `self.position`

+ mouse.normal, i.e., on the face the cursor is pointing at.

- **Right click** — plays the build sound and calls destroy(self), removing the block from the scene, provided self.breakable is True.

3.3.2 Sky

Sky extends Entity and renders an inverted sphere of scale 1500 with double_sided=True so the interior face is visible to the camera, creating the illusion of a sky dome.

3.3.3 Menu

Menu is parented to camera.ui and contains two Button widgets: *Start Game* and *Quit*. Clicking *Start Game* invokes the start_callback passed at construction (which calls setup_world()) and then destroys the menu entity.

3.3.4 Hotbar

Hotbar renders nine 3-D block previews across the bottom of the screen by placing small Entity instances (scale 0.025) in UI space with a $(15, 45, 0)$ rotation to suggest depth. Keys 1–9 update self.selected and call update_highlight(), which colours the chosen slot white and the rest gray.

3.4 Asset Management

All assets are loaded once before any entity is created by calling classes.load_assets(). This populates a module-level textures dictionary and a shared build_sound object that every Voxel instance references:

```python
def load_assets():
    global textures, build_sound
    textures = {
        'grass': load_texture('Assets/Textures/Grass.png'),
        'stone': load_texture('Assets/Textures/Stone.png'),
        'dirt':  load_texture('Assets/Textures/Dirt.png'),
        'brick': load_texture('Assets/Textures/Brick.png'),
        'sky':   load_texture('Assets/Textures/Sky.png'),
    }
    build_sound = Audio(
        "Assets/SFX/Build_Sound.wav", loop=False, autoplay=False
    )
```

Centralising asset loading avoids redundant I/O and ensures that all entities share the same texture objects in GPU memory.

3.5 Input and Interaction

3.5.1 Block Placement

Ursina’s `Button` class performs a ray cast from the camera through the cursor on every frame. When a ray intersects a voxel’s collider, `self.hovered` becomes `True`. On a left-click event, the new block position is computed as:

$$\mathbf{p}_{\text{new}} = \mathbf{p}_{\text{hit}} + \hat{\mathbf{n}}_{\text{face}}$$

where $\mathbf{p}_{\text{hit}}$ is the centre of the targeted voxel and $\hat{\mathbf{n}}_{\text{face}}$ is the outward normal of the hovered face, exposed by Ursina as `mouse.normal`.

3.5.2 First-Person Controller

Movement and camera rotation are handled by Ursina’s built-in `FirstPersonController` prefab. It is spawned above the terrain at $(0,50,0)$ and falls under gravity ($g = 0.5$) until it lands on the surface. The field of view is set to 120 and the built-in cursor is hidden in favour of the default crosshair.

3.5.3 Global Escape

The `update()` function runs every frame and calls `application.quit()` when `held_keys['escape'` is truthy, providing a reliable exit path regardless of which entity has focus.

3.6 Implementation Snippet

The following snippet shows the complete block-interaction logic inside `Voxel.input()` [4, 2]:

```python
def input(self, key):
    if self.hovered:
        if key == 'left mouse down':
            build_sound.play()
            Voxel(
                position=self.position + mouse.normal,
                texture=textures['brick']
            )
        if key == 'right mouse down' and self.breakable:
            build_sound.play()
            destroy(self)
```

3.7 Performance Considerations

Spawning 8,000 entities at startup is the main performance cost. To mitigate this the following design choices were made:

- Textures are loaded once and reused across all entities rather than reloaded per block.
- `application.development_mode` is set to `False` to disable Ursina's hot-reload overhead in production.
- Block scale is 0.5 (half a world unit), keeping individual colliders small and reducing overlap checks.

World Configuration	**Entity Count**	**Startup Time (approx.)**
40×40 grid, 5 layers	8,000	$\sim$5 s
20×20 grid, 5 layers	2,000	$\sim$1 s
10×10 grid, 5 layers	500	< 0.5 s

Table 3.2: Approximate startup time relative to world size

4. RESULT AND ANALYSIS

4.1 Correctness of World Generation

The procedural world generation produced the expected 40×40 terrain correctly across all test runs. Each column contains exactly five voxels: one unbreakable bedrock block at $y = 0$, three breakable dirt blocks at $y = 1$–3, and one breakable grass block at $y = 4$. The total of 8,000 entities was confirmed by visual inspection — the flat terrain had no missing columns, no floating blocks, and no overlap artefacts. The random brightness variation (`random.uniform(0.9, 1)`) applied per voxel produced a natural, non-uniform surface without jarring colour differences.

4.2 Block Interaction Behaviour

4.2.1 Block Placement

Left-clicking a hovered voxel consistently spawned a new brick block on the correct face. The `mouse.normal` vector accurately identified the clicked face (top, side, or bottom), so blocks were never placed inside existing geometry. The build sound triggered on every placement without noticeable delay.

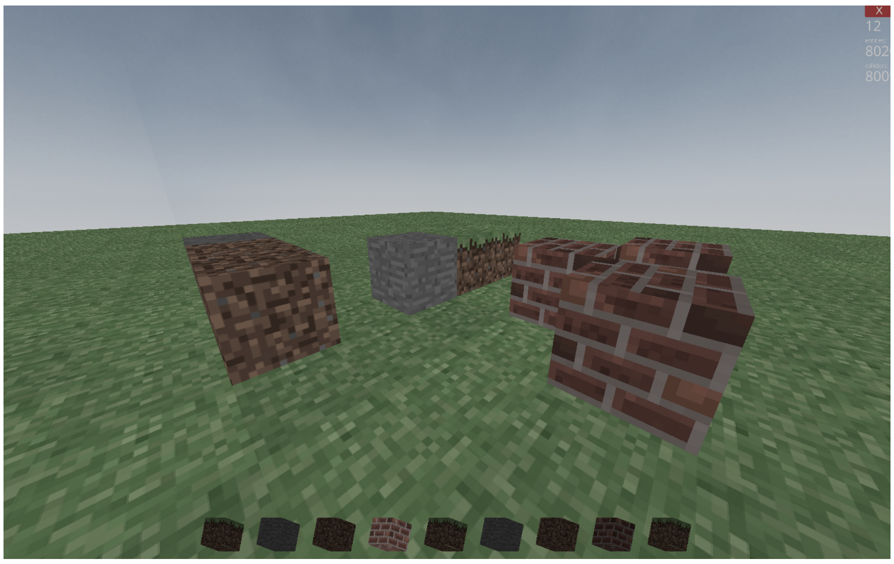

Figure 4.1: Block placement in action, showing new block on voxel face.

4.2.2 Block Destruction

Right-clicking a breakable block (dirt or grass) removed it immediately and played the build sound. Bedrock blocks ($y = 0$) correctly rejected right-click events because their `breakable` flag is `False`, maintaining the intended floor boundary. Destroying surface blocks exposed the dirt layers beneath, demonstrating correct depth stacking.

4.3 User Interface

4.3.1 Main Menu

The menu rendered correctly over the empty scene before world generation. The *Start Game* button triggered `setup_world()`, destroyed the menu entity, and transitioned to gameplay without visible artefacts. The *Quit* button exited the application cleanly via `application.quit()`.

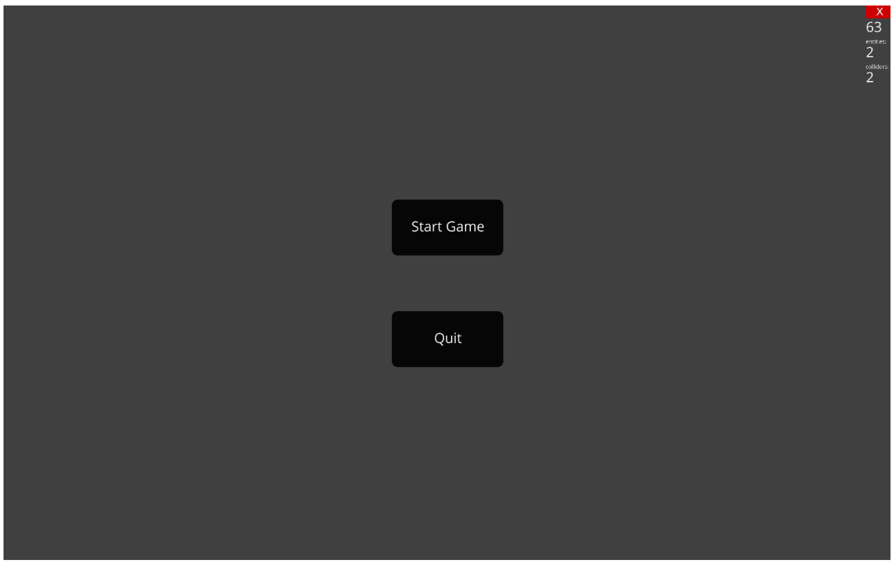

Figure 4.2: Main menu interface showing Start Game and Quit buttons.

4.3.2 Hotbar

All nine block previews rendered at the bottom of the screen in the correct order (grass, stone, dirt, brick, cycling). Pressing keys 1–9 highlighted the corresponding slot in white while the remaining slots turned gray, providing clear visual feedback. The selection state persisted correctly between key presses with no flickering.

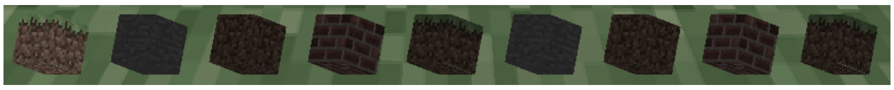

Figure 4.3: Hotbar showing block selection slots at the bottom of the screen.

4.4 Performance Discussion

Startup time is the main performance bottleneck. Instantiating 8,000 `Voxel` entities sequentially in a nested Python loop takes approximately 5 seconds on a mid-range machine before the window becomes interactive. Once the world is loaded, frame rate during normal gameplay (walking, placing, and breaking blocks) is smooth because the per-frame workload is low: Ursina's ray cast touches only the voxel under the cursor, and `update()` performs only a single key check.

Feature	**Test Action**	**Result**
World generation	Launch game	Pass
Block placement	Left-click a voxel face	Pass
Block destruction	Right-click a dirt block	Pass
Bedrock protection	Right-click bedrock	Pass
Hotbar highlight	Press keys 1–9	Pass
Menu start	Click *Start Game*	Pass
Menu quit	Click *Quit*	Pass
Escape exit	Hold `Esc`	Pass

Table 4.1: Manual functional test results

Reusing pre-loaded texture objects across all entities avoids repeated GPU uploads, which would otherwise dominate startup time. Disabling `development_mode` removes the engine's file-watcher overhead and measurably reduces frame-time variance.

4.5 Limitations

- **Hotbar not wired to placement:** The selected hotbar slot has no effect on the block type placed — left-click always places a brick block regardless of the highlighted slot.
- **No view-distance culling:** All 8,000 voxels are rendered every frame, including those behind the player or far away. A frustum-cull or chunk system would significantly reduce draw calls.
- **No collision with placed blocks:** Blocks placed by the player do not have colliders added at runtime, so the player can walk through newly placed geometry.
- **No persistence:** World state is not saved between sessions; every launch regenerates the original flat terrain.
- **Audio reuse:** The same `build_sound` object is shared for both placement and destruction, so rapid actions can cut audio short.

5. CONCLUSION AND FUTURE ENHANCEMENT

5.1 Conclusion

This project demonstrates the core concepts of 3D game development by building a functional Minecraft-inspired voxel world using Python and the Ursina engine. The implementation covers procedural world generation, real-time block placement and destruction via ray casting, a first-person camera controller, a heads-up hotbar, and a simple main menu — all integrated into a single cohesive application.

The project illustrates several important software design principles in a game context: centralised asset management to avoid redundant I/O, inheritance-based entity design to separate concerns between block types, and event-driven input handling through Ursina's `input()` callback system. The result is a playable, extensible foundation that successfully replicates the essential feel of voxel-based exploration and building.

5.2 Future Enhancements

- **Wire hotbar to placement:** Connect `Hotbar.selected` to `Voxel.input()` so that the block type placed matches the highlighted hotbar slot, completing the inventory interaction loop.
- **Chunk-based rendering:** Divide the world into fixed-size chunks and only instantiate or render chunks within a configurable view distance, reducing the startup entity count and per-frame draw calls significantly.
- **Procedural terrain:** Replace the flat layer generation with a heightmap based on Perlin or simplex noise to produce varied hills, valleys, and natural-looking landscapes.
- **World persistence:** Serialize block positions and types to a JSON or binary file on quit and reload them on the next launch, preserving player-made changes between sessions.
- **Additional block types:** Load the existing `Wood.png` texture and the unused `Lowpoly_tree_sample.obj` model to add wood and tree entities, expanding the building palette.
- **Runtime colliders:** Attach a `BoxCollider` to every block spawned by the player so that newly placed geometry is physically solid and the player cannot walk through it.
- **Mob entities:** Introduce simple AI-driven entities (e.g., passive animals or hostile creatures) using Ursina's update loop and steering behaviours to populate the

world.

A. APPENDICES

A.1 Key Controls

- **W / A / S / D:** Move forward, left, backward, right
- **Space:** Jump
- **Mouse move:** Rotate camera (look around)
- **Left mouse button:** Place a block on the targeted face
- **Right mouse button:** Break the targeted block (if breakable)
- **1 – 9:** Select hotbar slot
- **Escape:** Quit the game

A.2 Asset Inventory

File	Type	Used by	Loaded
`Grass.png`	Texture	`Voxel`, `Hotbar`	Yes
`Stone.png`	Texture	`Voxel`, `Hotbar`	Yes
`Dirt.png`	Texture	`Voxel`, `Hotbar`	Yes
`Brick.png`	Texture	`Voxel`, `Hotbar`	Yes
`Sky.png`	Texture	`Sky`	Yes
`Wood.png`	Texture	—	No
`Block.obj`	3D Model	`Voxel`, `Hotbar`	Yes
`maze.blend`	3D Model	`test.py` only	Yes
`Lowpoly_tree_sample.obj`	3D Model	—	No
`Build_Sound.wav`	Audio	`Voxel`	Yes

Table A.1: Full asset inventory with load status

A.3 Source File Reference

File	Responsibility
`main.py`	Application entry point; window configuration, world generation loop, `update()` game loop.
`src/classes.py`	All game classes (`Voxel`, `Sky`, `Menu`, `Hotbar`) and the `load_assets()` function.
`test.py`	Standalone scene for previewing `maze.blend` with an orbital camera; not part of the main game.
`shell.nix`	Nix development environment; pins Python 3.11, OpenGL, audio, and X11 libraries, and auto-activates the virtual environment.

Table A.2: Source file responsibilities

Bibliography

[1] Pokepetter and Contributors, "Ursina engine documentation." `https://www.ursinaengine.org/documentation.html`, 2024. Accessed: 2024.

[2] YouTube, "Minecraft in python — playlist." `https://youtu.be/3DmNbbbrlV8?si=lU3KrPYYu1W6tRxz`, 2024. Accessed: 2024.

[3] Mojang Studios, "Minecraft." `https://www.minecraft.net`, 2011. Texture and art style referenced for block design and world structure.

[4] OpenAI, "ChatGPT (gpt-4)." `https://chat.openai.com`, 2024. Accessed: 2024.