VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY

THE INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DSA REPORT

Game: Minimos

Name	ID	Contribution
Khương Minh Triết	ITITDK22041	100%

Ho Chi Minh City, Vietnam

2024-2025

Table of Contents

1. Introduction

1.1. Topic motivation

- My first game development project is about making a colony simulation game, inspired by resource management and life simulation games like Going Medieval and Stardew Valley. These types of games are popular around the world because they let players plan, build, and manage a group of people. The goal of my game is to mix simulation, task control, and pathfinding so that players can manage settlers who gather resources, build things, and react to different problems.

- Colony sim games are known for characters that act on their own, systems for collecting and using resources, and open-ended gameplay. Instead of controlling each person directly, players give orders or assign jobs. This style lets players be creative and think ahead. Every new game can feel different because of random maps and changing situations, which makes the game fun to play many times.

- This project helped me learn about how games are made, using object-oriented programming (OOP) and data structures and algorithms (DSA) with C# in Unity. I worked on features like task lists, pathfinding (A*), animations, and user interface (UI) messages. These parts help make the game fun and show my programming skills.

- Because I didn’t have enough time to finish everything, I focused on small but important features. I made sure characters can move,

do basic jobs like collecting wood, and show actions with text or animations. These basic parts show the idea of a colony sim game and give a good starting point to add more features in the future.

1.2. Developer team

Team name: Last man standing

Name	ID	Contribution
Khương Minh Triết https://github.com/triek	ITITDK22041	Report writing Game developing Graphic designing Presentation

1.3. References

- Going Medieval - Foxy Voxel


- Stardew Valley - Eric Barone


- Unity Documentation (docs.unity3d.com)

2. Software requirements

2.1. Working tools, platforms:

- Unity (Game Engine)
- C# (Programming Language)
- Visual Studio (Code Editor)
- GitHub (Version Control)
- Microsoft Visual Studio: Coding environment
- Adobe Photoshop: Game level map, background graphic, character, UI buttons

2.2. Choice of Game engine

- I chose to use Unity with C# instead of a simple Java IDE because Unity makes it easier to build games with graphics, animations, and user interaction.
- While Java is good for learning programming and writing basic logic, it does not have built-in tools for creating game worlds, placing objects, or adding sound and animation.
- Unity has a visual editor and many ready-to-use features that helped me create a better-looking and more interactive game.

- This choice allowed me to focus more on game design and logic, rather than spending too much time building tools from scratch.

3. Game Rule and Game Loop

3.1. Game Rule

- Players do not control settlers directly.
- Players click on tasks or areas, or press a button to trigger an action. The settlers then automatically move and perform the action.
- Each settler has a queue and performs one task at a time in the order it was added.
- Tasks include picking up flowers for now.

3.2. Game Loops

The game loop is based on selecting settlers and giving them tasks using a queue system:

- When a settler is not selected, they wander randomly around the map.
- Player can click on a settler to select them.
- Once selected, the player can click on the ground to move the settler there directly.
- The player can also click on a flower, then press a UI button to queue a pickup flower task for the selected settler.
- Multiple tasks can be queued in order using the task queue.
- The settler follows A* pathfinding to reach the target of each task.

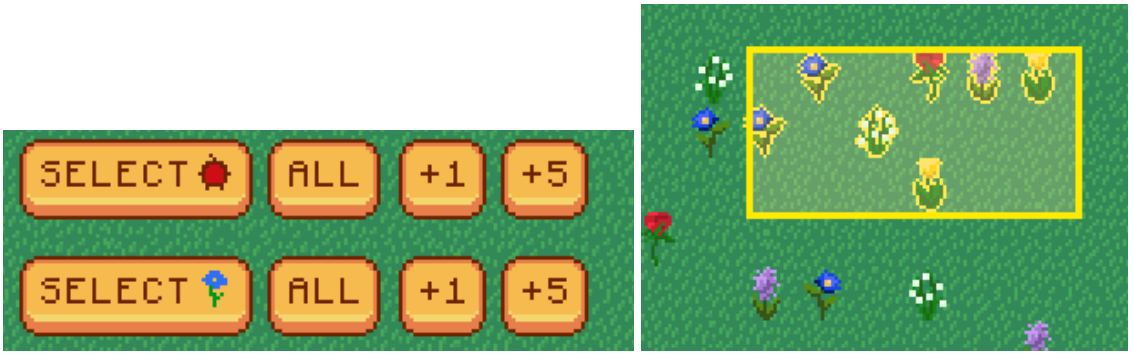
- When the settler arrives, they perform the action (e.g., pick up the flower).
- The next task in the queue starts automatically until the queue is empty.
- If no tasks are assigned, the settler will return to random wandering.

3.3. Basic Controls:

- Left-mouse press: Choose characters, flower, press UI buttons, drag to area select
- Right-mouse press: Choose moving destination

.

UI buttons:



There are two main UI rows for different object types (characters and flowers), each with the following buttons:

- **SELECT**: Selects an area inside the dragged square for objects of that type (e.g., settlers or flowers).
- **ALL**: Selects all objects of that type currently visible on screen.
- **+1 / +5**: Spawns 1 or 5 more of the selected object type (e.g., more settlers or more flowers).

4. System Design

4.1. List of Classes and their Responsibilities:

Folder	Script Name	Description
Interaction	Action	Defines types of actions a settler can perform.
	PickupButtonController	Handles logic when the pickup button is pressed.
	PickupFlower	Handles the logic for picking up flowers.
Movement	SettlerMovement	Controls settler movement and animation.
	MouseClickMovement	Moves settler to clicked ground location.
	MovementManager	Handles and coordinates settler movement states.
Pathfinding	AStarManager	Manages the A* pathfinding system for navigation.
	Node	Represents individual nodes used in A* calculations. Calculates A* paths for settler navigation.
ResourceGenerator	FlowerSpawner	Spawns flower resources at random or set positions.
	FlipSync	Synchronizes object orientation based on direction.
	PrefabsChecker	Checks for valid prefab configurations.
	ResourceGenerator	Handles general resource spawning and management.
	SettlerSpawner	Spawns settler characters into the game.
Selecting	CharacterSelector	Allows player to select/deselect settlers.
	DrawRect	Draws a rectangle area for selecting multiple objects.
	FlowerSelector	Handles logic for selecting flower objects.

	SelectButton	Manages selection button functionality.
	SettlerManager	Oversees all settlers and their registration.
	UIManager	Coordinates UI interactions with selection events.
Storage	ResourceManager	Tracks collected resources (e.g., flower count).
TaskManager	Task	Defines a task including target and type.
	TaskExecutor	Manages settler task queues and execution.
	TaskManager	Coordinates task creation, tracking, and removal.
UI	DebugConsole	Displays debug logs and messages on screen for testing.

Fig 4.1.1.: List of Classes and their Responsibilities

4.3. Program breakdown

1. Task System:

The task system is central to settler behavior. It is built using the `Task`, `TaskExecutor`, and `PickupFlower` classes:

- **Task** stores the task type (e.g., PickupFlower), the world position, and an optional reference to a target object. Each task has a status (`Pending`, `InProgress`, `Completed`, or `Failed`).
- **TaskExecutor** holds a queue of tasks per settler. When a task is added with `EnqueueTask()`, it is executed one-by-one. The settler moves to the target using A* pathfinding and performs the task.
- **PickupFlower** handles the execution of flower pickup tasks, including animation and attaching the flower to the settler.

Tasks are triggered either from the UI button (e.g., pickup command) or automatically on right-click via the `Action` class. After each task completes, the next one begins until the queue is empty.

2. Movement System:

Settlers switch between idle wandering and goal-directed movement:

- `SettlerMovement` makes idle settlers wander randomly within a defined area.
- `MouseClickMovement` moves settlers along a calculated A* path toward a user-clicked location or task target.
- `MovementManager` handles switching between wandering and click-based control, depending on player selection.

3. Player and Level:

The A* algorithm is implemented in `AStarManager`, which provides pathfinding between any two walkable nodes. Each node is represented by the `Node` class, which stores its neighbors and cost data.

4. Selection and UI:

- `CharacterSelector` and `FlowerSelector` allow the player to select settlers or flowers using clicks or drag-box.
- `SelectButton` and `UIManager` show/hide selection controls and allow selecting all units with one click.
- UI buttons are used to assign tasks or spawn units. These interact with the selection system and task logic.

5. UI Buttons:

The game includes a button-based interface that allows players to manage characters and assign tasks efficiently. There are two main UI rows for different object types (e.g., settlers and flowers), each with the following buttons:

- **SELECT**: Selects an area around the clicked location for objects of that type (e.g., settlers or flowers).
- **ALL**: Selects all objects of that type currently visible on screen.
- **+1 / +5**: Spawns 1 or 5 more of the selected object type (e.g., more settlers or more flowers).

For example, the red icon row controls settler selection and spawning, while the blue flower icon row is used to select flowers and queue pickup tasks. This visual grouping helps players easily identify which type of object they are interacting with.

These buttons connect to the `TaskExecutor`, `SettlerSpawner`, and `UIManager` to perform selection, spawning, and task assignment logic.

4.4. DSA Concepts

1. OOP

The entire game is structured using **Object-Oriented Programming (OOP)**:

- Each settler, task, flower, or UI element is a class.
- Scripts like `MovementManager`, `PickupFlower`, and `TaskExecutor` encapsulate logic and interact via method calls and events.
- DSA (Data Structures and Algorithms) concepts are applied to implement real-time systems like pathfinding, selection, and task queuing.

2. Queue

The game uses the **Queue** data structure to manage settler actions:

- In `TaskExecutor.cs`, each settler has a `Queue<Task>` to store tasks in the order they are given (FIFO).
- When a player queues a flower pickup, it's added to this queue.
- `ExecuteNextTask()` handles each task sequentially, maintaining task order.

Example:

```
Queue<Task> taskQueue = new Queue<Task>();
taskQueue.Enqueue(new Task("Pickup Flower", ...));
```

3. List

`List<T>` is widely used as a flexible collection type:

- `CharacterSelector.cs` and `FlowerSelector.cs` use `List` to track selected units or objects.
- `Node.cs` uses `List<Node>` to store connected neighbors in the A* pathfinding graph.
- `FlowerSpawner.cs` and `ResourceGenerator.cs` use `List<SpawnData>` to manage spawnable prefabs with weighted probabilities.

Example:

```
List<MovementManager> selectedCharacters;
List<Node> connections;
List<FlowerSpawnData> flowerSpawnDataList;
```

4. Hash Table (HashSet)

HashSet is used to efficiently prevent duplicate or overlapping positions:

- `SettlerMovement.cs` keeps a `HashSet<Vector3>` of occupied grid spaces to avoid settlers stacking.
- `ResourceGenerator.cs` uses a `HashSet<Vector2>` to avoid spawning resources on top of one another.

Example:

```
private static HashSet<Vector3> occupiedPositions = new();
if (occupiedPositions.Contains(nextPos)) { ... }
```

5. Graph

The map is treated as a **graph** of walkable nodes:

- Each `Node` connects to nearby nodes using `List<Node> connections`.
- Nodes are linked during scene initialization using `ConnectNearbyNodes()`.

Example:

```
public List<Node> connections = new List<Node>();
public void ConnectNearbyNodes(float radius) { ... }
```

6. Graphs Advanced – A Algorithm*

The game uses the **A*** algorithm to find the optimal path from a settler to a task location:

- Implemented in `AStarManager.cs` using `gScore`, `hScore`, and `fScore()` for cost tracking.
- Settlers query `GeneratePath()` and receive a list of `Node` steps to follow.
- This allows pathfinding to handle obstacles and long distances effectively.

Example:

```
public List<Node> GeneratePath(Node start, Node end) {
  // A* logic using openSet, gScore, and hScore
}
```

5. Game methods

1. Summary

Method Name	Description
`EnqueueTask(Task task)`	Adds a task (e.g., PickupFlower) to a settler's queue.
`ExecuteNextTask()`	Starts the next task after the previous one is completed.
`PickupFlower.Execute()`	Executes pickup logic with movement, animation, and attachment.
`GeneratePath(Node start, end)`	Runs A* algorithm to compute the optimal path between nodes.
`FindNearestNode(Vector2 pos)`	Finds the node closest to the click or settler position.
`SetTargetPosForTask(Vector3 pos)`	Tells a settler to move to a position as part of a task.

`MoveTo(Vector3 targetPos)`	Called from random or task systems to start movement to a target.
`SelectCharacter(bool)`	Selects or deselects a settler and updates its animation/UI.
`List<FlowerSpawnData>`	Manage different flower prefabs and their spawn chances for weighted random selection.
`QueuePickupTasks(IEnumerable)`	Called from UI or right-click to assign multiple flower tasks.
`SelectAll() / DeselectAll()`	Used by selection button to control all settlers at once.
`OnMouseClick()`	Starts A* movement when player right-clicks ground.
`SelectCharactersInRectangle()`	Selects all settlers within a drawn rectangle using drag selection.
`RegisterSettler()`	Registers settler to central manager for selection and tracking.

```
    public static int[][] GetLevelData() {  2 usages   triek
        BufferedImage img = GetSpriteAtlas(LEVEL_ONE_DATA);
        int[][] lvlData = new int[img.getHeight()][img.getWidth()];

        System.out.println("Map size: " + img.getHeight() + " : " + img.getWidth());

        for (int j = 0; j < img.getHeight(); j++)
            for (int i = 0; i < img.getWidth(); i++) {
                Color color = new Color(img.getRGB(i, j));
                int value = color.getRed();
                if (value >= 70)
                    value = 69;
                lvlData[j][i] = value;
            }
        return lvlData;

    }
}
```

Fig 4.4.1.1.: GetLevelData method

This method will get the level data such as size, RGB color codes of the pixels.

```
public void draw(Graphics g, int lvlOffset) {  triek *

    for (int j = 0; j < Game.TILES_IN_HEIGHT; j++)
        for (int i = 0; i < levelOne.getLevelData()[0].length; i++) {
            int index = levelOne.getSpriteIndex(i, j);
            g.drawImage(levelSprite[index],  x: Game.TILES_SIZE * i - lvlOffset,
                    y: Game.TILES_SIZE * j, Game.TILES_SIZE, Game.TILES_SIZE,  observer: null);

        }
}
```

Fig 4.4.1.2.: draw method

This method will draw the map tiles by tiles using the data from the GetLevelData method

Fig 4.4.1.3.: Blocks graphics

Fig 4.4.1.4.: Block drawing guide

Each tile is specified by the value of Red color. For instance, the tile 10 has a value of R:10, G:any, B:any with “any” being any number from 0-255.

Fig 4.4.1.1.: Game level design

Each pixel in this map will have the color value corresponding to its block.

2. Orb method

Fig 4.4.2.1.: Boost orb animation

```
    private void drawBoosts(Graphics g, int xLvlOffset) {  1 usage   triek *
        for (Boost c : boosts) {
            g.drawImage(boostArr[c.getOrbState()][c.getAniIndex()],  x: (int) c.getHitbox().x - xLvlOffset - 14,
                    y: (int) c.getHitbox().y - 14, BOOST_WIDTH, BOOST_HEIGHT,  observer: null);
//              Draw hitbox
//              c.drawHitbox(g, xLvlOffset);
        }
    }
```

Fig 4.4.2.2.: Draw orb method

3. Character method

Fig 4.4.3.1.: Character animation

```
private void loadAnimations() {  1 usage   triek
        BufferedImage img = LoadSave.GetSpriteAtlas(LoadSave.PLAYER_ATLAS);

        animations = new BufferedImage[6][9];
        for (int j = 0; j < animations.length; j++)
            for (int i = 0; i < animations[j].length; i++)
                animations[j][i] = img.getSubimage( x: (i+1)*135,  y: (j+1)*135,  w: 135,  h: 135);

}
```

Fig 4.4.3.2.: Draw character method

4. Physics methods

```
if(!inAir)
    if(!IsEntityOnFloor(hitbox, lvlData))
        inAir = true;

if(inAir) {
    if(CanMoveHere(hitbox.x, y: hitbox.y + airSpeed, hitbox.width, hitbox.height, lvlData)) {
        hitbox.y += airSpeed;
        airSpeed += gravity;
        updateXPos(xSpeed);
    } else {
        hitbox.y = GetEntityYPosUnderRoofOrAboveFloor(hitbox, airSpeed);
        if((gravity > 0 && airSpeed > 0) || (gravity < 0 && airSpeed < 0))
            resetInAir();
        else
            airSpeed = fallSpeedAfterCollision;
        updateXPos((xSpeed));

    }
} else
    updateXPos(xSpeed);

}
```

Fig 4.4.4.1.: Check if in air

This method will check if the character is in the air to prevent jumping while in air. It will be true when the character is not on the floor or immediately set to true when the character jumps.

```
} else
    upda
}

private void checkBoostAttack() {  1 usage   triek
    if (attackChecked || aniIndex != 1)
        return;
    attackChecked = true;
    playing.checkBoostHit(attackBox);
    boolean isHit = orbManager.isBoostHit(attackBox);
    if (isHit) {
        jump();
    }
}
```

Fig 4.4.4.2.: Boost orb animation

This will check if the player executes attack action, which is also the jump button, while touching the orb. If it is true then the player jumps.
It will ignore the fact that the character is in the air.

```
private static boolean IsSolid(float x, float y, int[][] lvlData) {
    int maxWidth = lvlData[0].length * Game.TILES_SIZE;
    if (x < 0 || x >= maxWidth)
        return true;
    if (y < 0 || y >= Game.GAME_HEIGHT)
        return true;

    float xIndex = x / Game.TILES_SIZE;
    float yIndex = y / Game.TILES_SIZE;

    int value = lvlData[(int) yIndex][(int) xIndex];

    if (value != 69)
        return true;
    return false;

}
```

Fig 4.4.4.3.: Check if the ground is solid method

```
public static boolean CanMoveHere(float x, float y, float width, float height, int[][] lvlData) {

    if(!IsSolid(x, y, lvlData))
        if(!IsSolid( x: x+width,  y: y+height, lvlData))
            if(!IsSolid( x: x+width, y,lvlData))
                if(!IsSolid(x,  y: y+height, lvlData))
                    return true;
    return false;

}
```

Fig 4.4.4.4.: Check if the player can move method

5. Game over methods

```
private void updateXPos(float xSpeed) {  3 usages  triek
    if(CanMoveHere( x: hitbox.x + xSpeed, hitbox.y, hitbox.width, hitbox.height, lvlData)) {
        hitbox.x += xSpeed;

    } else
        dying = true;
}
```

Fig 4.4.5.1.: If the player can not move then game over

```
public void update() {  triek
    if (dying) {
        if (playerAction != DYING) {
            playerAction = DYING;
            aniTick = 0;
            aniIndex = 0;
            playing.setPlayerDying(true);
            System.out.println("Died");
        } else if (aniIndex == GetSpriteAmount(DYING) - 1 && aniTick >= aniSpeed - 1) {
            playing.setGameOver(true);
        } else {
            updateAnimationTick();
        }
        return;
    }
```

Fig 4.4.5.2.: Methods when the game is over

Fig 4.4.5.3.: Game over situation

In this situation, the player could have jumped if they pressed jump when touched the orb, but they missed and fell to the blocks below. Because the player approached the blocks from the left side, they could not move further to the left, therefore they would be game over.

6. Button methods

```
} else if (isIn(e, menuB)) {
    if (menuB.isMousePressed()) {
        playing.resetAll();
        Gamestate.state = Gamestate.MENU;
        playing.unpauseGame();
    }

} else if (isIn(e, replayB)) {
    if (replayB.isMousePressed()) {
        playing.resetAll();
        playing.unpauseGame();
    }

} else if (isIn(e, unpauseB))
    if (unpauseB.isMousePressed())
        playing.unpauseGame();
```

Fig 4.4.6.1.: Menu, restart and continue button methods

5. Conclusion

In this project, I developed a simplified colony simulation game that brings together settler control, task queuing, pathfinding, and resource collection in a dynamic 2D environment. Players interact with settlers indirectly through a UI and selection system, assigning actions like picking up flowers using intuitive buttons and real-time feedback. This structure reflects core simulation mechanics seen in full-scale games but remains focused and manageable for learning.

The primary goal of this project was to apply Data Structures and Algorithms (DSA) in a practical context. I used core structures like queues for task management, lists for storing game objects, and dictionaries for resource tracking. The A* Pathfinding algorithm was implemented to calculate settler movement paths efficiently across the game grid. Additionally, logical flow

structures and event callbacks were used to model settler behavior in a modular and extensible way.

5.1 What can be improved next time?

While the project demonstrates a solid foundation, several areas can be improved or added in future iterations:

- **Add More Task Types**: Currently, only picking up flowers is implemented. Adding more task types such as chopping trees, mining, farming, or crafting would greatly expand gameplay variety.
- **Smarter AI and Collaboration**: Settlers could automatically decide what to do when idle or work together on larger tasks.
- **Building and Farming System**: Introduce the ability to construct structures and grow crops.
- **Visual Feedback and Animations**: Add clearer task progress indicators, better animations, and visual polish.
- **Task Prioritization and Roles**: Enable players to assign settler roles or task priority queues.
- **Save/Load System**: Allow persistent world states between game sessions.
- **Collaborative Development**: Working with others would allow for better testing, faster progress, and learning from peers.

This project has laid a solid groundwork for future development and has given me valuable experience designing systems that are both functional and extensible.

This project helped me learn how games work under the hood, especially how to break down systems like movement, task queues, and pathfinding learned in DSA course. Even though I could not finish all planned features due to time

limits, I successfully created a working foundation of a colony simulation game. The settlers can move, take orders, and gather resources, which proves that the core systems are functional and expandable. In the future, I plan to improve UI, add farming and building systems, and make the AI smarter.