VRIJE
UNIVERSITEIT
BRUSSEL

Project Computersystemen

PLANTS VERSUS ZOMBIES IN X86

Tugay Ülger & Vojtech De Coninck

First Term

Academic year 2024-2025

Computerwetenschappen & Ingenieurswetenschappen

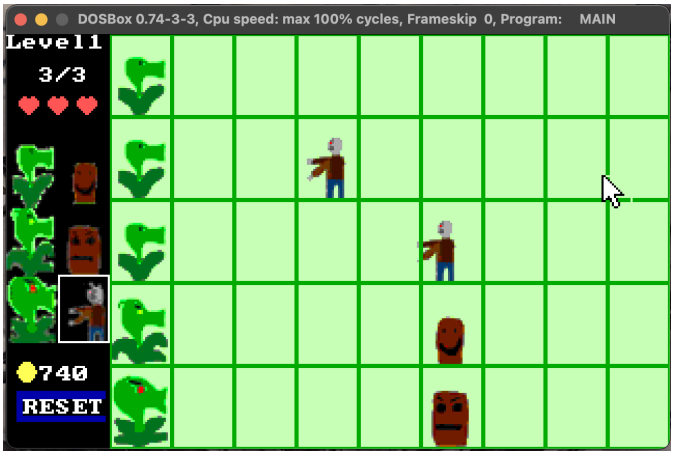


Figure 1: Screenshot of the game.

1 Introduction

We are inspired by the classic mobile game "Plants versus Zombies". It is a tower defense game, but has its own twist to it. Instead of having 'towers', this game has 'plants', all kinds of plants. These plants are not only used to guard a path the enemies need to follow, they are also used to block the path. So the "enemies", in this case the "zombies", have to also clear their path from these plants in order to get to their destination. The game is designed in a grid shape, where the rows are distinct lanes, the zombies spawn from the right side, and the player loses 'hearts' or points when the zombies reach the left side of the board. The zombies come in waves and the game has its difficulty increased, by increasing the zombies' difficulties, the amount of zombies in the waves, and so on. There are also different kinds of plants. Some plants are used to obstruct the path of the enemies, whilst others shoot projectiles. The plants can be placed on a tile, the beauty of the game is placing the plants strategically. As the game progresses, the player faces more difficult zombies and has the opportunity to find strategies to defeat the waves using more capable plants that cost more coins.

2 Manual

How to Play the Game

Note that there is some background music, be sure to activate your headphones. The music is from YouTube[1]. And the graphics are custom made in GIMP and then converted to bin files using the methods provided by the course assistants. The game is played using the mouse only. Here's how to interact with the game:

- **Placing Entities**:
  - On the left side of the screen, you will see buttons corresponding to different entities, such as plants or zombies (note that the zombie is for illustrative purposes only and will not be present in the final game).
  - Click on one of the buttons to select the entity you want to place.
  - Then, click on a grid position where you want to place the entity. The entity will be placed at the selected position on the grid.

Changing the Graphics

To change the game's graphics, follow these steps:

[1] `https://www.youtube.com/watch?v=IhEDW8zJfSk`

- **Graphics Files**: The game's graphics are stored in bin files. To modify the graphics, replace the existing bin files with your custom graphics.
- **File Placement**: Make sure the new graphics are placed in the correct folder and named appropriately to replace the existing files.

Changing the Music

To change the background music, perform the following steps:

- **Music Files**: The music is stored in a bin file. Replace the current music file with your desired audio track.
- **File Placement**: Ensure that the new music file is placed in the correct directory and named properly to replace the original music.
- **Music Loop**: The game's music is set to loop every 52 seconds, which is hardcoded into the game. Changing the music file will update the soundtrack, but the loop duration will remain the same unless the code is modified.

3 Program features

Program Features

The game is designed with a focus on simplicity, performance, and flexibility. The main features include:

Gameplay

The game is controlled entirely through the mouse. The player can select entities, such as plants or zombies, from the toolbar on the left side of the screen. By clicking on the buttons, the player can choose an entity, and then clicking on the grid places the selected entity at the clicked position. The zombie button is used for illustrative purposes and is not part of the final game.

Entity Interaction

The game features a dynamic interaction between entities. Entities are placed within a grid system, which consists of a static grid for allocated entities and a dynamic grid for movable entities. This grid structure optimizes collision detection and interaction by categorizing entities into separate layers, allowing for efficient and accurate movement and interaction handling. The collision detection system uses layer abstraction to determine how entities should interact, whether through collisions, attacks, or movement restrictions.

Level Design and Waves

The game supports multiple levels and wave configurations. Each level consists of a series of waves, which include spawn patterns for entities. The game progression is controlled through wave counters and entity spawn management. The level design is flexible, allowing for various wave configurations that influence gameplay dynamics.

Graphics and Music Customization

The game allows players to modify its visuals and audio:

- **Graphics**: The game's graphics can be changed by replacing the relevant bin files with custom graphics.
- **Music**: Players can change the background music by replacing the music file with their desired audio track. The music is set to loop every 52 seconds.

Limitations

- The zombie entity button in the shop is purely illustrative, and the full game would not include this entity.
- The music is hardcoded to loop every 52 seconds, and customization of this loop duration would require modifying the source code.
- There is a static amount of memory allocated to entities, which may limit scalability for more complex levels.
- in every grid, entities can not be stacked, stacking is done through layering different entity grids, constraining the amount of possible stacked entities to a constant amount, which is equal to the amount of grids. Extending the amount of grids, is costly, since the memory for all cells are statically allocated before hands, making stacking entities not desirable and costly.

4 Program Design

Please note that the code is in one simple file, which is not desired, to improve readability we have sectioned the code using comments that draw a line between every section. The program is structured as follows:

- A set of macros to improve readability.
- Randomizer code, which is copied from the examples.
- Functions that are for general purpose.
- The code that operates on the static grid.
- The code that operates on the dynamic grid.
- The game logic that is centered around the `updateGameState` function.
- The code for graphical purposes is centered around drawing the static and dynamic entities.
- The code to handle mouse input.
- The code to handle sprite drawing is copied from the examples as well.
- The code to handle audio output is also copied from the examples.
- The code that processes levels.

- The code for the user interface.
- The `main` function that orchestrates the program.
- The data segment that defines the respective variables and data structures, such as entities and levels.

The main function contains the following:

4.1 Pseudocode for Game Logic

Algorithm 1: GameLoop

Input: `current_game_state`, `VGAMEM`

```
1 Call updateGameState(current_game_state);
2 Call renderFrame(VGAMEM);
```

Algorithm 2: updateGameState

Input: `current_game_state`

```
1  for i = 0 to ROWS do
2      for j = COLUMNS to 0 do
3          if cell(i, j) contains a zombie then
4              Set spawn_projectile_row[i] = true;
5              Call handleZombieMovement(i, j);
6          if cell(i, j) contains a plant then
7              if spawn_projectile_row[i] = true then
8                  Call handleProjectileSpawning(i, j);
9          if cell(i, j) contains a projectile then
10             Call handleProjectileMovement(i, j);
```

Algorithm 3: handleProjectileMovement

Input: `projectile_cell`

```
1 Move projectile one cell to the right;
2 if projectile_cell reaches the end of the grid then
3     Despawn the projectile;
4 if projectile_cell encounters a zombie then
5     Call inflictDamage(projectile, zombie);
6     Despawn the projectile;
```

Algorithm 4: handleZombieMovement

Input: `zombie_cell`

if *`zombie_cell` to the left is unoccupied* **then**
    Move zombie one cell to the left;
else if *`zombie_cell` to the left contains a plant* **then**
    Call `inflictDamage(zombie, plant)`;
    Zombie waits;
if *`zombie_cell` reaches the end of the grid* **then**
    Call `loseHeart()`;

Explanation

- **GameLoop**: The main loop of the game. It calls two critical functions:
  - `updateGameState`: Updates the state of the game based on current grid conditions.
  - `renderFrame`: Renders the graphical state of the game on the screen.
- **updateGameState**: Iterates over the grid from top to bottom and from right to left.
  - If a zombie is encountered:
    - Sets a flag `spawn_projectile_row` for the row to spawn projectiles later.
    - Handles zombie movement.
  - If a plant is encountered:
    - Spawns a projectile if the row flag is set.
    - Calls `handleProjectileSpawning`.
  - If a projectile is encountered:
    - Calls `handleProjectileMovement`.
- **handleProjectileMovement**: Manages the movement of projectiles.
  - Projectiles move right. If they reach the end, they despawn.
  - If they hit a zombie, they inflict damage and then despawn.
- **handleZombieMovement**: Manages the movement of zombies.
  - Zombies move left if the space is unoccupied.
  - If a plant is in the way, they wait and inflict damage on the plant.
  - If a zombie reaches the end, it triggers the `loseHeart()` function.

4.2 Game Cycle and Entity Timings

All the timings and speeds are dependent on the game cycle, so on the speed of one iteration of the main function. So everything is timed based on the game cycle. Things such as projectile spawning rate, zombie movement speeds, and also zombie damage speed are based on the internal clock of every entity, which is respectively in its data structure. The idea is that the code is object-oriented, in some ways it is, but some other things such as image drawing and its path and so on, which generally would be a method that calls the path to the image in the class, are done in a functional way. This was done because it was more convenient for us to do so at the moment of writing that code.

4.3 Grid Data Structure for Efficient Collision Detection

In our game, the memory on the side of entities is entirely static because we constrained the game to only allow one entity per cell. Even with this constraint, entities can still move over each other, as long as they are stored in a different grid layer, thanks to the use of different grids. The high-level idea of the grids is as follows:

In typical collision detection algorithms, we would need to iterate over all elements, which is not optimal in terms of performance. To optimize this, we section the game state into grids and check for collisions within each cell. In our case, we move entities in discrete positions that define the collision, and we adjust the drawing relative to these positions. When an entity moves enough, we change its grid cell. This significantly simplifies collision detection.

This approach is only feasible because of the initial constraint of one entity per cell in a grid layer. It makes the system both memory- and performance-efficient. The only drawback is that we have a static amount of memory allocated for the maximum number of entities. However, since the memory required is relatively small (approximately $2 \times \text{ROWS} \times \text{COLUMNS} \times$ size of entity structure), the system works quite well. The result is seamless: entities appear to move continuously, even though the underlying mechanics are discrete.

Static Grid and Dynamic Grid

The grid data structure is split into two parts: a static grid and a dynamic grid. This division defines which entities can move and which cannot. Essentially, all entities can stand and move, but static entities are optimally allocated and freed.

Static Grid The static grid is an array of size `ROWS` × `COLUMNS` where each element is an entity data structure. The data structure, explained later, is placed directly in the static grid for static entities, and changes are made to the structures in place. The first `COLUMNS` cells in the array represent row 0, and so on. This creates a mapping from index `i` to (x, y) coordinates in the 2D grid.

Dynamic Grid The dynamic grid separates the memory storage and location of dynamic entities. When initializing a dynamic entity, we search for a place in the statically allocated memory array for dynamic entities and store the relative index of this position in a separate array. This array defines the (X, Y) locations in the abstract 2D grid.

2D Grid Representation

The 2D grid can be seen as a matrix of (X, Y) coordinates, with an additional dimension indicating whether an entity is static or dynamic. This separation allows for more efficient movement: instead of relocating an entire structure in memory when an entity moves, we only adjust the relative index in the location array or grid.

The separation of grids also simplifies defining interactions between entities. For example, if an object in the background collides with an entity from grid `x` but not with one from grid `y`, this behavior can be specified in the game logic. This makes it possible to define interaction hierarchies, which can be used for drawing the scene as well. By drawing from hierarchy 0 to the top, we can create an overlay of images.

Static Grid Implementation

The static grid is simply an array of `ROWS` × `COLUMNS` size, where each element represents an entity's data structure. Each element in this array corresponds to a cell in the abstract 2D grid.

```
struct Entity {
    // Entity properties
};

#define ROWS 10
#define COLUMNS 10
struct Entity staticGrid[ROWS][COLUMNS];
```

Dynamic Grid Implementation

The dynamic grid separates memory allocation from location tracking. When an entity is initialized dynamically, it is placed in the static memory array for dynamic entities, and its position is tracked relative to the static grid.

```
struct DynamicEntity {
    // Dynamic entity properties
};

struct DynamicEntity* dynamicGridMemory[MAX_DYNAMIC_ENTITIES];
int locationGrid[MAX_DYNAMIC_ENTITIES];
```

Movement in the Grid

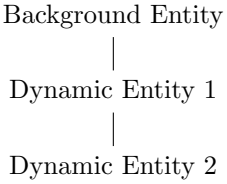
When an entity moves in the dynamic grid, we simply adjust the index in the location grid instead of reallocating memory for the entire entity. This makes movement much more efficient.

```
void moveEntity(int entityID, int newX, int newY) {
    locationGrid[entityID] = newX * COLUMNS + newY;
}
```

This approach ensures that entities appear to move seamlessly, while the actual implementation is based on discrete grid updates. Note that moving in the static grid is also possible, but it would require to move the whole structure, in this approach only moving the identifier is enough, thus saving computational steps at the expense of some extra memory.

Interaction and Hierarchy

The separation into static and dynamic grids also simplifies the definition of interactions between entities. By organizing entities into different grids, we can specify which entities should interact with each other and in what way. This facilitates the creation of interaction hierarchies, which can be used to draw entities in layers, with lower priority entities drawn first.



4.4 Entity Data Structure

The entity data structure is the core representation of game entities in the grid system. Each entity is represented by a structure that holds various properties defining its state and behavior. These properties include vital attributes such as health, damage, speed, position, and timers, which are essential for gameplay mechanics like movement, animation, and collision detection.

Structure Breakdown

The structure of an entity is as follows:

```
STRUC ENTITY
    type            db -1                   ; Entity type, initially set to
        -1 as a placeholder
    health          db ?                    ; Current health of the entity
    damage          db ?                    ; Damage dealt by the entity
    speed           db ?                    ; Movement speed of the entity
    x               db ?                    ; X-coordinate in the 2D grid
    y               db ?                    ; Y-coordinate in the 2D grid
    direction       db ?                    ; Direction in which the entity
        is facing
    animationFrame  db ?                    ; Current animation frame for
       the entity
    animationTimer  db ?                    ; Timer to control animation
       frame updates
    spriteIndex     db ?                    ; Index of the sprite in the
       sprite sheet
    fireInterval    db ?                    ; Time interval between firing
       actions
    fireTimer       db ?                    ; Timer for controlling firing
       actions
ENDS ENTITY
```

Each field in the ‘ENTITY‘ structure serves a specific function:

type Represents the type of the entity, such as zombie, projectile, or player. It is initially set to -1 as a placeholder and updated upon entity initialization.

health Tracks the current health value of the entity. It is modified when the entity takes damage.

damage Specifies the amount of damage the entity can inflict on others. This value is used during combat or collision.

speed Defines the movement speed of the entity, which influences how quickly it can traverse the grid or interact with other entities.

x, y Represent the entity's position on the 2D grid. These coordinates determine where the entity is located within the game world.

direction Stores the current facing direction of the entity, which can be used for things like animations or movement logic.

animationFrame Holds the current frame of the entity's animation, allowing for smooth animation transitions.

animationTimer A timer that controls how often the animation frame should change, ensuring that the animation is updated at regular intervals.

spriteIndex This value stores the index that points to the specific sprite in the sprite sheet used to render the entity.

fireInterval Defines the minimum time required between consecutive firing actions, used to prevent rapid fire.

fireTimer A timer that tracks the elapsed time since the last firing action, determining whether the entity is allowed to fire again.

Usage in the Game

This entity data structure is used within both the static and dynamic grids. In the static grid, entities are placed at fixed locations, and their state is updated as they interact with the environment. In the dynamic grid, entities move within the grid, and their positions are tracked using relative indices.

The properties such as 'health', 'damage', 'speed', and 'direction' directly influence gameplay mechanics such as combat, movement, and AI behavior. Meanwhile, 'animationFrame' and 'animationTimer' are used to handle visual aspects of the game, ensuring that entities are rendered with appropriate animations.

The 'fireInterval' and 'fireTimer' attributes are crucial for managing combat mechanics, specifically in regulating the firing rate of projectiles or attacks, ensuring that entities cannot fire too quickly.

Overall, this data structure provides a compact yet versatile representation of game entities, balancing memory efficiency with gameplay functionality. Note that for some entities some of the members are not used or redundant. We chose to do this in order to have a uniform entity data structure that can be used in the whole game. Everything entity is essentially expressed as the same. If the programming language allowed object oriented programming, we would have created an object hierarchy for that, but this being absent didn't stop us from thinking in OOP terms.

4.5 Drawing the Game State

In order to render the game state, entities are drawn in layers, starting with the static entities and then drawing the dynamic ones. This ensures that entities that are part of the background are rendered first, followed by entities that move or interact with the player. The following steps outline how to draw the game state for each layer of entities.

1. Initialize Drawing

Before drawing any entities, we must initialize the drawing environment. This includes setting up the screen buffer (a memory area where pixel data is stored before being displayed) and defining the color palette used for drawing. The screen buffer holds the graphical representation of the game world, and the palette defines how colors will be mapped in the drawing process.

- Initialize screen buffer.
- Set up the color palette.

2. Iterate Over Entities

For each layer (static and dynamic), we need to loop through the respective entity arrays and access their properties. Specifically, we iterate over the `dynamic_entities_array` for dynamic entities. For each entity in the array, we retrieve its properties such as position, sprite index, and other relevant attributes (e.g., health, damage, etc.).

- Loop through `dynamic_entities_array` and retrieve entity data.
- Retrieve entity attributes such as position (`x`, `y`), sprite index, and others.

3. Calculate Screen Position

Each entity has a position in the game world, denoted by its coordinates `(x, y)`. To display the entity on the screen, we need to convert these game world coordinates to screen coordinates. This step may involve adjusting for the camera position or viewport, which allows the game world to be shifted relative to the screen. The camera's position will typically affect how the entity's position is transformed into a screen coordinate.

- Convert entity's world coordinates (`x`, `y`) to screen coordinates.

4. Retrieve Sprite Data

Each entity has a sprite that visually represents it on the screen. The sprite is identified by its `spriteIndex`, which is an index into a predefined sprite sheet or image database. The sprite data contains information about the visual representation of the entity, including pixel information, dimensions, and transparency.

- Use the `spriteIndex` to retrieve sprite data from memory.
- The sprite data is typically a 2D array or bitmap representing the entity's appearance.

5. Draw Sprite

Once the sprite data has been retrieved, we need to copy the sprite to the appropriate location in the screen buffer. This is done by iterating over the sprite's pixel data and setting the corresponding pixels in the screen buffer.

- Iterate over the sprite's pixel data.
- Set corresponding pixels in the screen buffer.
- Handle transparency if the sprite has transparent pixels (often a specific color index or alpha channel is used for transparency).

6. Update Buffer

After all entities for a given layer (either static or dynamic) have been drawn, the screen buffer must be updated to reflect the changes. This typically involves copying the contents of the screen buffer to the video memory or using a graphics API to render the updated buffer on the screen. This step effectively makes the changes visible to the player.

- After drawing all entities, update the screen buffer.

Example Process for Dynamic Entities

The following pseudocode outlines the general process for drawing dynamic entities:

```
void drawEntities(Entity* dynamicEntities, int numEntities, ScreenBuffer* buffer) {
    for (int i = 0; i < numEntities; i++) {
        Entity* entity = &dynamicEntities[i];

        // Calculate screen position
        // (X, Y) -> (X * cell_width + offset, Y * cell_height + offset)
        int screenX = convertToScreenX(entity->x);
        int screenY = convertToScreenY(entity->y);

        // Retrieve sprite data
        Sprite* sprite = getSprite(entity->spriteIndex); // is actually hardcoded in our code

        // Draw the sprite to the screen buffer
        for (int row = 0; row < sprite->height; row++) {
            for (int col = 0; col < sprite->width; col++) {
                // Get the pixel from the sprite and place it in the buffer
                if (sprite->pixels[row][col] != TRANSPARENT) { // please note that the transparent part is not implemented because of time constraints
                    buffer->pixels[screenY + row][screenX + col] = sprite->pixels[row][col];
                }
            }
        }
    }

    // Update screen buffer to reflect changes
    updateScreen(buffer);
}
```

This pseudocode demonstrates the process for drawing dynamic entities, from calculating the screen position and retrieving sprite data to updating the screen buffer.

Conclusion

Drawing the game state involves iterating over the entities, calculating their screen positions, retrieving sprite data, and updating the screen buffer. The process ensures that both static and dynamic entities are rendered correctly, and the result is a seamless and visually cohesive game world. Also please note that the information about buffering and the sprite index in the sprite table isn't implemented in that exact way, the intended implemenation is different, because we did not get to make that fully operational. Our vision was to create the code towards a reusable code base, that would make it possible to use the same data structures and functions in another game, essentially creating a game engine. There are also some ideas, like integrating file types, level editor ideas and integrating drawing editors and so on, in order to make the creating of new games much smoother. Even though we were aware of the time constraint, this didn't withhold us from making the code as generic as possible, we obviously only succeeded partially doing so.

4.6 Game Logic and Collision Detection

The game logic was designed to be as generic as possible, with the flexibility to extend and modify behaviors through abstraction layers. While not all aspects of the code were fully implemented to achieve maximum genericity, the core ideas are in place, allowing for easy expansion. The central concept is to enable movement in the x and y directions and to detect collisions between static and dynamic entities.

The collision detection system works with an abstraction of layers, where the goal is to detect interactions between different layers (such as static vs. dynamic entities) and the respective positions. Extending this system involves adding new layers and parameterizing the collision detection function to account for the layers and the positions involved.

Collision Detection Logic

The collision detection function returns different values based on the nature of the entities being moved:

- **Return 0**: If moving an entity from position 1 to position 2 (or vice versa) is possible, meaning at least one of the two positions is empty.
- **Return 1**: If both positions are occupied by the same entity, indicating that the entity doesn't need to move or can simply wait.
- **Return 2**: If the two positions are occupied by different entities, signaling that an interaction (such as an attack) is needed.

Handling Movement and Interactions

The `handleMovement` functions utilize the collision detection results to determine the next action for each entity. If the result indicates that the entity can move, the function will update the entity's position. If the result indicates an attack or interaction, the respective functions will be called to inflict damage or remove the entity.

Specifically:

- The `inflictDamage` function is called when a collision between two entities requires damage to be applied.
- The `remove` or `free` function is triggered when an entity is removed from the game. This function frees the occupied position in the grid, marking it as available for other entities.

Layered Abstraction for Behavior Flexibility

The use of layers allows for complex behaviors to be easily added by creating new collision detection functions and extending the game logic. However, implementing these behaviors in a simple way—such as writing a small method composed of just a few smaller ones—is not always possible. This limitation arises from the decision not to make the code fully generic, though the overall architecture is flexible enough to accommodate such extensions. The layering system and collision abstraction provide a solid foundation for defining new entity interactions and behaviors while maintaining clarity and efficiency.

By allowing the creation of custom behaviors through the layering system, new interactions (e.g., entities moving differently depending on their layer or specialized actions) can be implemented by simply extending the current logic without requiring significant changes to the existing codebase.

4.7 Data Structures for Level Design and Waves

The game uses several key data structures to manage levels and waves of entities, allowing for flexible and dynamic level design. These structures are optimized for both performance and memory efficiency.

LEVEL Structure

The `LEVEL` structure is used to represent each level in the game. It includes the following fields:

- **level_name**: A pointer to the name of the level.
- **waves_count**: The number of waves in the level.
- **waves**: A pointer to the waves for the level.

Example of a `LEVEL` structure:

```
STRUC LEVEL
    level_name dd ?      ; Pointer to the level name
    waves_count db ?     ; Number of waves
    waves dd ?           ; Pointer to the waves
ENDS LEVEL
```

This structure is used to define different levels, each of which contains multiple waves. For instance, `level1_instance` stores the details of level 1, including its name and waves.

Level Array and Current Level Tracking

A static array holds references to different level instances, allowing for quick access to each level:

```
level_array dd offset level1_instance, offset level2_instance, ...
current_level_index db 0  ; Index of the current level
current_level dd offset level1_instance  ; Pointer to the current level
current_wave db 0  ; Counter for the current wave
current_entity db 0  ; Counter for the current entity in the current wave
wave_length db 15  ; Length of waves
spawn_counter db 0
```

The current level and wave are tracked using `current_level_index`, `current_level`, `current_wave`, and `current_entity`.

Wave Design

Each level contains several waves of entities. Each wave is defined using a sequence of entity positions, where each entry can represent a specific entity type, spawn location, or behavior. The -1 is an unused type identifier, which can be used to include a break in that specific spawn cycle. Note that in the current code the wave lengths are set to 15. Example for `level1_waves`:

```
level1_wave1 db -1, -1, 0, -1, 0, -1, -1, -1, 0, -1, -1, -1, 0, -1, -1
level1_wave2 db -1, -1, 0, 0, -1, -1, -1, 0, -1, 0, -1, -1, -1, 0, -1
level1_wave3 db -1, -1, -1, 0, -1, -1, -1, -1, 0, -1, -1, -1, 0, -1, -1
```

These waves define the order in which enemies or obstacles will appear, enabling structured progression in the game.

Processing Level Design and Waves

The logic for processing levels and waves follows a series of steps that are executed in the game loop:

- **Initialize Level**: Set up the level by placing the entities and defining the waves.
- **Load Level Data**: Load the predefined level data from structures or external files (if implemented).
- **Spawn Waves**: Based on the wave counter, spawn new waves of enemies at specified intervals or conditions.
- **Update Level**: Continuously update the level state, including entity movements, interactions, and wave progression.
- **Render Level**: Draw the level by iterating over the grid and dynamic entities array to render each entity on the screen.

Extending Logic for Persistent Storage

To extend the logic for saving and loading levels and waves persistently, file I/O operations need to be implemented. This allows the game state, including levels, waves, and player progress, to be saved and restored.

File Format and Serialization

- **Define File Format**: Choose a file format for storing level data and waves. A binary format could be efficient, but for better readability, a format like JSON could be used.
- **Save Level Data**: Serialize the current state of the level, including entities and wave configurations, and write them to a file.
- **Load Level Data**: Read the level data from the file and deserialize it to restore the game state.
- **Save Scores**: Serialize the player's score and progress to a file.
- **Load Scores**: Read the scores and progress from the file and restore the player's status.

Level and Entity Design Editor

In order to enhance the game development process and allow for more flexibility in level creation, we considered the possibility of developing an editor or utilizing an existing one to visually design levels and entities. The primary goal of this editor is to allow users to interactively modify the game's levels and entities by using a simple interface, such as click-and-drag actions.

Level Design Editor

The level editor would allow designers to visually place entities and define wave patterns for each level. By simply clicking and dragging, users could place zombies, define their spawn locations, and adjust the wave configurations, such as timing and entity types. This would enable quick iteration and testing of different level layouts without having to manually adjust values in the game code.

- **Wave Pattern Editor**: A tool for defining the sequence and timing of enemy waves, allowing designers to adjust the difficulty curve easily.
- **Entity Placement**: Entities like zombies can be placed directly on the grid, with the ability to define their type, health, speed, and other properties.

Entity Editor

The entity editor would be responsible for creating and modifying game entities, such as zombies or items. It would provide functionality for editing entity properties, including their appearance, behaviors, and variables.

- **Pixel Art Section**: A tool for drawing and editing the visual appearance of entities, allowing designers to create custom sprites for different entities.
- **Variable Editing**: The editor would allow the designer to adjust entity properties, such as health, damage, speed, and other variables directly through a user-friendly interface.

Benefits

By integrating a graphical editor for both level and entity design, we can provide a more intuitive and efficient workflow for developers. This would also enable easier testing and iteration, as well as reduce the need for manual coding adjustments. Whether creating new levels or tweaking entity properties, the editor would facilitate rapid prototyping and offer a more accessible interface for designing complex game behaviors.

5 Encountered problems

The development process was not without its challenges. The most difficult problem we faced was figuring out how to create structures, arrays of structures, and correctly update the members within such arrays. After thoroughly reading manuals and documentation, we eventually figured it out. We also created a repository to enable parallel work, which helped to streamline the development process.

From my perspective, the biggest challenge was debugging. It was a nightmare. In the early stages, we relied heavily on custom debugging methods, such as print functions, to track issues. While correcting code was difficult, understanding what was actually happening was even harder. There was no straightforward way to print strings, data structures, and other data types in a formatted, readable manner, making debugging much more cumbersome. This frustration made us realize the value of higher-level programming languages, which offer more convenient debugging tools and better abstractions for handling data structures.

Our code might not be the most efficient, as we worked at a low level, but this allowed us to gain a deep understanding of how things work under the hood. We even had to implement counters and timers for the first time, which was a unique learning experience.

Overall, the project was a valuable and enlightening experience. If we had the opportunity to start over with the knowledge we have now, we would attempt to create a more complex game or at least experiment with more advanced graphics, such as 2.5D, to further push the boundaries of our skills and creativity.