• what computer platform the team will use for the final assembly of the functional Components.
Our team has agreed to use C++ and the graphics and sound libraries provided in the course (including fssimplewindow.h, yssimplesound.h, ysglfontdata.h, and yspng.h). This ensures that all members are familiar with the tools and development environment.

• who will be the assembly leader, who takes the lead in integrating all the components into a single assembly.
Jia Xie has been assigned as the assembly leader. He is responsible for integrating all components into the final build.

• what data structures and classes should be defined and shared among the members.
Different data structures are shared among various components. We detailed the data we need from each other in our detailed design. Such as UI will require the TileMap[][], TileSize, Tile Types from the Map system to render the map. Also requires Player position, animation frame, score, lives, Monster positions, states, and animation frames to render Player and Monsters

Detailed Design— UI and Art Assets Component

1. Component Overview

This component is responsible for all visual rendering and user interface elements in the game, including:
- Complete art asset creation (AI creation or free art asset)
- Loading and managing art assets (characters, maze tiles, items, UI images)
- Drawing game UI screens (main menu, pause menu, game over screen)
- Displaying HUD elements (score, lives, level indicators)
- Rendering the maze map according to data provided by the Map component
- Rendering player and monster animations based on state information
- Playing UI-related sound effects when needed

2. Expected interaction with Other Components

Component	Data Provided	Used For Rendering
Map System	TileMap[][], TileSize, Tile Types	Drawing maze tiles and background
Player Controller	Player position, animation frame, score, lives	Rendering player and HUD
Monster AI	Monster positions, states, animation frames	Rendering monsters in different states
Core Engine	Game state (Menu/ Play/ Pause / End)	Deciding which UI screen to display

3. Art Resource Folder Structure

```
assets/
  ui/
    main_menu.png
    pause_overlay.png
    gameover.png
  characters/
    player/
      frame_0.png
      frame_1.png
      …
      frame_n.png
    monsters(all includes different frame like player)/
      red/
      blue/
      yellow/
    items/
      dot.png
      power.png
  tiles/
    wall.png
    road.png
    tunnel.png
```

4. Rendering Architecture

```
Main Game Loop:
  User Input
  Update Game Logic
  UI Component DrawFrame(CurrentState, SharedGameData)
  Swap Buffer
```

```
UI Rendering Logic:
DrawFrame(state):
  if state == MENU:
    DrawMainMenu()
  elif state == PLAY:
    DrawBackground()
    DrawMapTiles()
    DrawPlayer()
    DrawMonsters()
    DrawHUD()
  elif state == PAUSE:
    DrawPauseOverlay()
  elif state == GAMEOVER:
    DrawGameOverScreen()
```

5. Pseudocode

5.1 Asset Manager

```
class TextureManager:
  load(filename)
    if already loaded return
    else decode PNG and store texture ID
```

5.2 Map Rendering

```
DrawMap(mapGrid):
  for each tile (x,y):
    get tile texture
    draw at x*TILE_SIZE, y*TILE_SIZE
```

5.3 Player Rendering

```
DrawPlayer(player):
  select animation frame
  draw player image at player position
```

5.4 Monster Rendering

```
DrawMonster(monster):
  choose monster image set based on monster state
  draw at monster position
```

5.5 HUD Rendering

```
DrawHUD(score, lives, level):
  draw score text
  draw lives icons
  draw level text
```

6. State Transition Overview

MENU -> PLAY -> (PAUSE <-> PLAY) -> GAMEOVER -> MENU

Detailed Design— Monster AI

1. Component Overview

- This component is responsible for all ghost (monster) behavior, including:
- Reading the shared 0/1 map grid (1 = wall, 0 = walkable) and using it as the navigation graph
- Automatically generating a patrol loop from the text-based map for each ghost
- Updating ghost state using a finite-state machine (PATROL / CHASE / RETURN / STUNNED)
- Computing shortest-path distance to the player and triggering chase behavior
- Running A*-based pathfinding over the map grid for CHASE and RETURN states
- Applying local movement rules (only turning at intersections; 90° turns; 180° turn in dead ends)
- Detecting ghost–player collisions, including special “back-hit” detection to trigger STUNNED

2. Expected interaction with Other Components

Component	Data Provided	Used For Rendering
Map System	MapGrid[][] (0 = walkable, 1 = wall), map width/height	Building path planner
Player Controller	Player tile position (px, py) and facing direction pDir	Computing shortest-path distance; chase target; back-hit detection
UI & Art Assets	ghost positions, directions, and states	Rendering ghost according to state
Core Engine	deltaTime & Game state (Menu/ Play/ Pause / End)	Update, Pause, Reset

3. Pseudocode

```
enum GhostState { PATROL, CHASE, RETURN, STUNNED };

updateGhostAI(ghost):

  dist = shortestPathDistance(mapGrid, ghost.pos, playerPos)
```

```
// state update
if ghost.state == STUNNED:
    ghost.stunTimer -= deltaTime
    if ghost.stunTimer <= 0:
        ghost.state = RETURN
        ghost.path  = AStar(mapGrid, ghost.pos, nearestPatrolNode(ghost))
else if dist >= 0 and dist <= ghost.perceptionRange:
    ghost.state = CHASE
    ghost.path  = AStar(mapGrid, ghost.pos, playerPos)
else if ghost.state == CHASE or ghost.state == RETURN:
    if ghost.path.empty():
        ghost.state = RETURN
        ghost.path  = AStar(mapGrid, ghost.pos, nearestPatrolNode(ghost))
    if onPatrolPath(ghost):
        ghost.state = PATROL          // back to patrol loop


// movement (no movement while stunned)
if ghost.state != STUNNED:
    if isDeadEnd(mapGrid, ghost.pos, ghost.dir):
        ghost.dir = turnBack(ghost.dir)          // 180° at dead end
    else if isIntersection(mapGrid, ghost.pos) and not ghost.path.empty():
        desiredDir = directionToNext(ghost.path, ghost.pos)
        if isRightAngleOrStraight(ghost.dir, desiredDir):
          ghost.dir = desiredDir                 // ±90° or straight
    ghost.pos = ghost.pos + ghost.dir            // move one tile
```

```
// collision with player
if ghost.pos == playerPos:
    if playerDir == ghost.dir:                    // back-hit
        ghost.state         = STUNNED
        ghost.stunTimer = STUN_DURATION
    else:
        notifyPlayerHit(ghost)
```

Detailed Design— Player Control

1. Component Overview

This component manages all logic and control related to the player-controll, including keyboard input, movement, item collection, power-up mechanics, life management, and collision interactions with monsters. Parts including:

- Process keyboard input and buffered directions
- Update player movement and alignment within the maze
- Handle item collection (energy dots and power pellets)
- Manage power-up activation, countdown, and deactivation
- Detect and respond to monster collisions
- Maintain life count and respawn sequences
- Communicate with the Map, Monster AI, and Core Engine components

2. Expected interaction with Other Components

Component	Data Provided	Used For
Map System	TileMap data, Tile types, Collision info	Validate walkable paths, detect items (energy dots/power pellets)
Monster AI	Monster position, direction, and collision triggers	Handle player collision outcomes (death or stun)
UI & Art	Player position, state, animation frame, score, lives	Render player, HUD, and special effects

Core Engine	Game state signals (Menu / Play / Pause / Game Over)	Activate or disable input and state updates

3. Internal Data & States

Key internal structures:

enum PlayerState { NORMAL, POWERED, DYING, RESPAWNING, DEAD };
enum Direction { RIGHT, UP, LEFT, DOWN };
Position: int gridX, gridY; double pixelX, pixelY;
Movement: Direction currentDir; Direction bufferedDir; double moveSpeed;
Power-up: bool isPowered; double powerTimer;
Life system: int lives; double deathTimer; double respawnTimer;

4. Logic Summary

On each frame, if the game is active:

Read keyboard input and update intended direction.

Move Player along valid tiles; stop at walls.

On entering a tile, check for dots or power pellets and update score/state.

Update power-up timer and revert to Normal when expired.

Check collision with monsters:

If Powered: notify monster defeat and award bonus.

Otherwise: trigger life loss, death animation, respawn or game over.

5. Pseudo code

```
UpdatePlayer(deltaTime)

{

  if(GameState != PLAY) return;

  HandleInput();              // update bufferedDir

  if(CanTurn(bufferedDir))       // at tile center & next tile walkable

  {

    currentDir = bufferedDir;
```

```
        bufferedDir = NONE;
    }
    if(CanMove(currentDir))
    {
        MoveAlong(currentDir, moveSpeed * deltaTime);
        if(EnteredNewTile())
        {
            HandleItemPickup(gridX, gridY);
        }
    }
    UpdatePowerUp(deltaTime);        // decrease timer, reset state if needed
    CheckMonsterCollision();         // resolve hit: powered kill or lose life
}
```

Detailed Design— Map system

1. Component Overview (Map System)

This component is the foundational data layer for the game's environment. It manages the maze structure, defines the properties of all map tiles, and tracks the location and state of collectible items.

- Map Data Structure: Implement a robust 2D grid structure to represent the maze, storing wall/walkable data.
- Level Management: Load and manage different maze designs/levels, handling transitions and initial setup.
- Tile Definitions: Define and store tile properties (Wall, Path, Energy Dot, Power Pellet, Monster House, Tunnel).
- Walkability/Collision Map: Provide the definitive source of truth for collision validation for Player and Monster entities.

- Game State Tracking: Maintain the current state of the map, primarily tracking remaining energy dots and power pellets to detect level completion.
- Special Zone Logic: Implement logic for features like tunnel wrapping and the impassable barrier of the monster house.

2. Expected Interaction with Other Components

Component	**Data Provided (Map System → Component)**	**Used For**
UI & Art Assets	TileMap[][] (Tile Types), TileSize	Rendering the maze structure, walls, and items (dots/pellets).
Player Controller	TileMap[][] (Walkability), Tile Types	Validating player movement path, detecting tile-based collisions (walls), and checking for item pickup on a tile.
Monster AI	MapGrid[][] (0=walkable, 1=wall), Map Width/Height	Building the navigation graph for A* pathfinding and generating patrol paths.

3. Internal Data & States

```
enum TileType

{

  WALL,

  PATH,

  ENERGY_DOT,

  POWER_PELLET,
  MONSTER_HOUSE_DOOR,
  TUNNEL
};
```

4. Pseudo code

```
MapManager::LoadLevel(levelId):
    // 1. Load map structure from a file or data array based on levelId.
    read mapData from assets (e.g., CSV, text file)

    // 2. Reset game-specific counters.
    remainingEnergyDots = 0
    remainingPowerPellets = 0

    // 3. Populate the core TileMap array and count collectibles.
    for Y from 0 to MAP_HEIGHT - 1:
        for X from 0 to MAP_WIDTH - 1:
            TileMap[Y][X] = ParseTileType(mapData[Y][X]) // Map char/int to enum TileType

            if TileMap[Y][X] == ENERGY_DOT:
                remainingEnergyDots++
            else if TileMap[Y][X] == POWER_PELLET:
                remainingPowerPellets++

            // Store critical locations for other components
            if TileMap[Y][X] == PLAYER_START:
```

```
            StorePlayerStart(X, Y)
        if TileMap[Y][X] == MONSTER_START:
            StoreMonsterStart(X, Y)

MapManager::IsLevelComplete():
    return (remainingEnergyDots == 0) and (remainingPowerPellets == 0)

    // The currentLevelId has already been incremented by CheckForLevelAdvance()

    // 1. Reset all necessary map-related variables and structures.
    ResetMapState(); // Clear monster/player positions, power-up states, etc.

    // 2. Load the data for the new map (Map 2, then Map 3).
    LoadLevel(currentLevelId);

    // 3. Signal to other components to reset their entities
    // Core Engine will handle telling Player/Monsters to move to the new start positions
```