Republic of the Philippines

Laguna State Polytechnic University

Province of Laguna

COLLEGE OF COMPUTER STUDIES

[Void Invader]

NAME OF PROJECT

SUBMITTED BY:

ALCAZAR, JOHN ALEX B.

DELATORRE, EMMANUEL S.

DATUIN, JHAZTINE LAITRELL

BS IN COMPUTER STUDIES – 1A

JANUARY 06, 2025

Contents

PROJECT OVERVIEW

Project Name: VOID INVADER

Date Started: [OCTOBER 23, 2024]

Date Ended: [End Date]

Project Team Leader: ALCAZAR, JOHN ALEX B.

Project Team Members: DELATORRE, EMMANUEL S.

DATUIN, JHAZTINE LAITRELL

PURPOSE:

The primary purpose of the "Void Invader" project is to create an engaging and interactive gaming experience that allows players to navigate through various game modes, manage player attributes, and enjoy dynamic gameplay through a well-structured menu and rendering system. The project focuses on implementing essential game mechanics, such as player health management, enemy interactions, and a responsive user interface, to enhance overall gameplay enjoyment and performance.

KEY FEATURES:

1. Game Structure

- **Main Game Loop**: The `Program` class contains a main game loop that continuously updates and renders the game state, allowing for real-time gameplay.
- **Game States**: The game can switch between different states (e.g., main menu, single-player, two-player, scores) using a state management system.

2. Player Management

- **Player Class**: The codebase includes a `Player` class that manages player attributes, actions, and rendering. It supports both single-player and multiplayer modes.
- **Input Handling**: The game captures keyboard input to control player actions, such as movement and attacks.

3. Menu System

- **Dynamic Menus**: The `MainMenu` class provides a dynamic menu system that allows users to navigate through different game options (e.g., single-player, two-player, scores).
- **Selector Navigation**: The menu includes a visual selector that users can move up and down to choose options.

4. Rendering System

- **Custom Rendering**: The game uses a custom rendering system to draw game elements on the console, including borders, player characters, and menus.

- **Debug Information**: The code includes functionality to render debug information, which can be useful for development and testing.

5. Timer and Frame Rate Control

- **Frame Rate Management**: The game uses a timer to control the frame rate, ensuring smooth gameplay at a target frame rate (e.g., 60 FPS).
- **Update Mechanism**: The UpdateScreen method is called at regular intervals to refresh the game state and render the current frame.

6. Input Management

- **Keyboard Input**: The game utilizes a keyboard input system to capture player actions, with cooldowns to prevent rapid input.
- **Direct Input Handling**: The use of the DirectInput class allows for more responsive input handling.

7. Game Configuration

- **Customizable Dimensions**: The game allows for customizable screen dimensions (width and height), enabling flexibility in how the game is displayed.
- **Palette and Visual Settings**: The game engine can set visual palettes and other graphical settings to enhance the visual experience.

8. Exit and Cleanup

- **Graceful Exit**: The game includes functionality to clean up resources and exit gracefully when the player chooses to quit.

CODE DOCUMENTATION

CODING STANDARDS

1. **Naming Conventions**: The code uses PascalCase for class names (e.g., Player, MainMenu) and camelCase for method and variable names (e.g., playerOnePosition, attackCooldownFramesOne).

2. **Commenting**: There are comments throughout the code that explain the purpose of methods and sections.

3. **Use of Access Modifiers**: The code appropriately uses access modifiers (e.g., private, public) to encapsulate class members, promoting good object-oriented design principles.

4. **Separation of Concerns**: The project separates different functionalities into distinct classes (e.g., Player, MainMenu).

5. **Error Handling**: The code includes basic error handling (e.g., try-catch blocks) to manage exceptions that may arise during execution, enhancing the robustness of the application.

FUNCTIONS/METHODS

1. Function: Create()

Purpose: The Create() method is responsible for initializing the game environment. It sets up the console window, configures the game engine, and prepares the main menu and player instances. This method is called once at the start of the game to establish the initial state.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

2. Function: UpdateScreen()

Purpose: The UpdateScreen() method is responsible for updating the game state and rendering the current frame. It checks whether the game is in a playing state or in the menu state, updates the player and if the game is active, and renders the appropriate visuals to the console. This method is called repeatedly at a set interval to ensure smooth gameplay.

Parameters:

- object state

Return Value:

- This method does not return a value (void).

3. Function: Update

Purpose: The Update method is responsible for updating the player's state, including handling player movement, attacks, and bullet updates. It ensures that the player's actions are processed each frame, allowing for real-time interaction.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

4. Function: CanAttack

Purpose: The CanAttack method checks if the player is able to perform an attack based on the cooldown frames. It manages the attack timing and prevents the player from attacking too frequently.

Parameters:

- ref int attackTime
- int cooldownFrames
- ref bool attackPressed

Return Value:

- Returns a boolean value (true or false), indicating whether the attack can be performed.

5. Function: UpdateBullets

Purpose: The UpdateBullets method updates the position of the bullets fired by the player. It checks if the bullets are still within the screen bounds and removes any that have gone off-screen.

Parameters:

- List<Point> bullets

Return Value:

- This method does not return a value (void).

6. Function: Render

Purpose: The Render method is responsible for drawing the player and their bullets on the console screen. It calls other rendering methods to display the player character and their projectiles.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

7. Function: RenderPlayer

Purpose: The RenderPlayer method draws the player character on the console based on its position and color. It calculates the hitbox for the player and renders each segment of the player character.

Parameters:

- Point[] player
- Point position
- int color

Return Value:

- This method does not return a value (void).

8. Function: RenderBullets

Purpose: The RenderBullets method is responsible for drawing the bullets on the console screen. It iterates through the list of bullets and renders each one at its current position using the specified color.

Parameters:

- List<Point> bullets
- int color

Return Value:

- This method does not return a value (void).

9. Function: RenderBorder

Purpose: The RenderBorder method draws the borders of the game area on the console. It creates a visual boundary by rendering the top, bottom, left, and right edges of the game screen using a specified border color.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

10. Function: Dispose

Purpose: The Dispose method is responsible for releasing any resources used by the Player class. It ensures that the keyboard and direct input resources are properly cleaned up to prevent memory leaks and other resource-related issues.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

11. Function: Render

Purpose: The Render method is responsible for displaying the main menu on the console. It clears the buffer, loads the necessary fonts, renders the borders, and displays the game title and menu options based on the current page.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

12. Function: LoadFonts

Purpose: The LoadFonts method loads the font files required for rendering text in the game. It initializes the font objects that will be used to display the game title and menu options in a stylized format.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

13. Function: RenderBorder

Purpose: The RenderBorder method draws the borders of the game area on the console. It creates a visual boundary by rendering the top, bottom, left, and right edges of the game screen using a specified border color.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

14. Function: RenderSelector

Purpose: The RenderSelector method visually indicates the currently selected menu option. It draws a selector (cursor) next to the selected option to provide feedback to the player about their current choice.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

15. Function: GameTitle

Purpose: The GameTitle method displays the title of the game on the console using a stylized font. It positions the title at a specific point on the screen and uses the loaded font to render it.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

16. Function: RenderMainMenu

Purpose: The RenderMainMenu method displays the main menu options on the console. It creates an array of menu options and iterates through it to render each option at the specified positions.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

17. Function: Render1PlayerMenu

Purpose: The Render1PlayerMenu method displays the 1-player menu, prompting the user to enter their name and providing options for gameplay. It shows the player's name input and the available game mode (Survival) along with a back option.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

18. Function: Render2PlayerMenu

Purpose: The Render2PlayerMenu method is intended to display the 2-player menu.

(Not Implemented)

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

19. Function: RenderSurvival

Purpose: The RenderSurvival method is intended to display the survival game mode interface.

(Not Implemented)

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

20. Function: RenderScores

Purpose: The RenderScores method is designed to display the scores or leaderboard information.

(Not Implemented)

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

21. Function: RenderDebugInfo

Purpose: The RenderDebugInfo method outputs debugging information to the console. It provides insights into the current state of the application, such as the current page and the position of the selector, which can be useful for troubleshooting.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

23. Function: Update

Purpose: The Update method is responsible for updating the game state. It checks the current keyboard state, reduces cooldown timers, and handles input based on the current page of the menu.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

24. Function: HandleMainMenuInput

Purpose: The HandleMainMenuInput method processes user input for the main menu. It checks for key presses to navigate through the menu options and handles the selection of different pages.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

25. Function: Handle1PlayerMenuInput

Purpose: The Handle1PlayerMenuInput method processes user input specifically for the 1-player menu. It allows the player to navigate options and enter their name, as well as handle the selection of gameplay modes.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

26. Function: Handle2PlayerMenuInput

Purpose: The Handle2PlayerMenuInput method is intended to process user input for the 2-player menu.

(Not Implemented)

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

27. Function: CanType

Purpose: The CanType method checks if the player can type based on the cooldown timer. It ensures that input is not registered too quickly, providing a smoother user experience.

Parameters:

- ref int moveTime:
- int moveCooldown:

Return Value:

- Returns a boolean indicating whether the player can type (true) or not (false).

28. Function: ResetToMainMenu

Purpose: The ResetToMainMenu method resets the menu state back to the main menu. It clears player names and resets the selector position.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

29. Function: ExitGame

Purpose: The ExitGame method handles the cleanup and termination of the game. It releases any acquired resources and exits the application.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

30. Function: HandleKeyboardInput

Purpose: The HandleKeyboardInput method processes general keyboard input. It checks for key presses and executes corresponding actions based on the current context of the game.

Parameters:

- This method does not take any parameters.

Return Value:

- This method does not return a value (void).

31. Function: GetCharacterFromKey

Purpose: The GetCharacterFromKey method is designed to convert a key press into a character representation. This would be useful for inputting player names or other text-based interactions.

Parameters:

- Key key

Return Value:

- This method would return a character corresponding to the key pressed

TESTING

Provide a few examples of test cases and expected results.

TEST CASES

[Test Case 1]

- Description: [Briefly describe the test case]
- Expected Result: [Specify the expected outcome]

[Test Case 2]

...

Test Results

[Summarize the results of executed test cases]

USER GUIDE

USER INTERFACE

1. Main Menu

- **Options**: The main menu presents several options for the user, including:
  - **1-Player**: Starts a single-player game.
  - **2-Player**: Initiates a multiplayer game.
  - **Scores**: Displays the leaderboard or scores.
  - **Exit**: Exits the game.

2. Selector

- **Navigation**: A visual selector (cursor) allows users to navigate through the menu options. The selector's position changes based on user input (W/S keys) to highlight the currently selected option.

3. 1-Player Menu

- **Input Field**: Prompts the user to enter their name for the single-player mode.
- **Options**: Includes options to start the "Survival" mode or go back to the main menu.

4. 2-Player Menu

- **Placeholder**: Currently, there is no detailed implementation for the 2-player menu, but it is intended to allow for multiplayer game setup.

5. Scores Display

- **Leaderboard**: A section dedicated to displaying player scores and rankings, although the rendering details are not provided.

6. Visual Elements

- **Borders**: The interface includes borders rendered around the console window to enhance visual structure.
- **Fonts**: Custom fonts are loaded for rendering text in a stylized manner, enhancing the overall aesthetic of the menu.

7. Interaction

- **Keyboard Input**: Users interact with the interface primarily through keyboard input, using keys to navigate the menu and select options.

TROUBLESHOOTING

Rendering Issues -

FREQUENTLY ASKED QUESTIONS (FAQ)

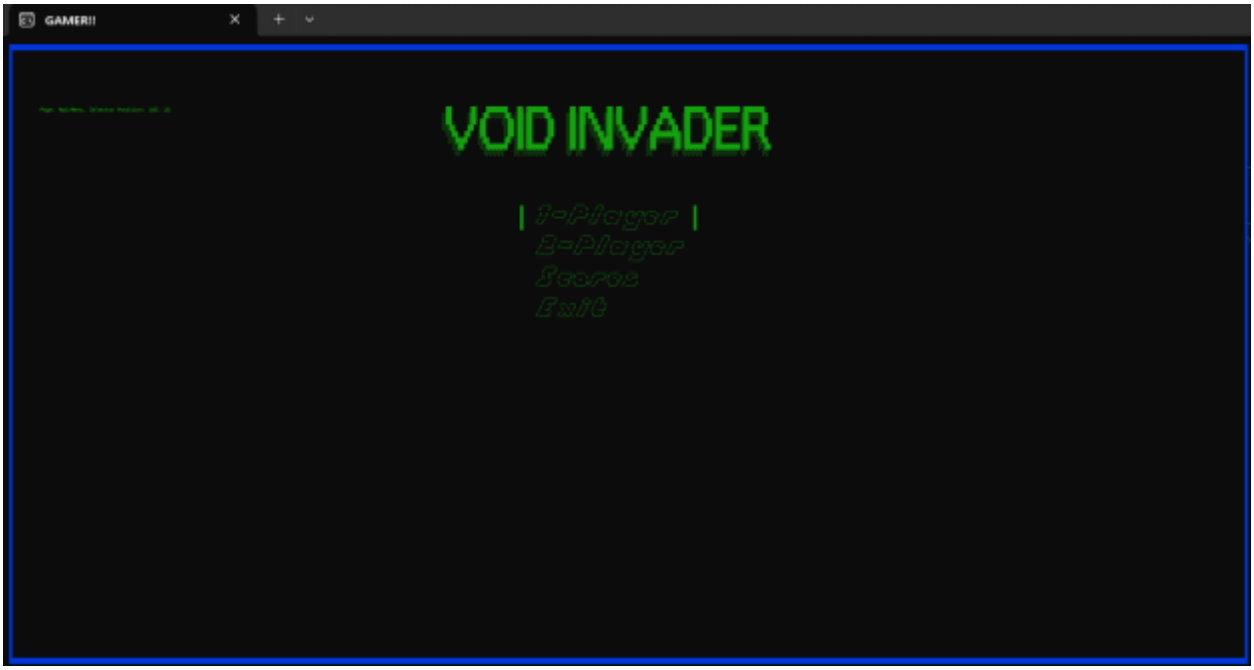
[Question 1]

Answer: [Provide the answer]

[Question 2]

...

CONSOLE SCREENSHOTS



The provided image is a screenshot from a console game project titled "VOID INVADER." The interface employs a retro, pixelated design inspired by classic arcade games. The title, displayed prominently at the center, is complemented by a set of interactive menu options:

1-Player – Starts a single-player game.

2-Player – Initiates a two-player game.

Scores – Displays high scores or game statistics.

Exit – Closes the game.

The screen layout includes a minimalistic blue border, enhancing the visual structure of the menu. Additionally, there is developer-related information, such as "Page: NullMenu, Selector Position," displayed in the top-left corner, suggesting that the game is still under development or debugging.

FUNDAMENTALS OF PROGRAMMING APPLIED

1. Variables and Data Types:

- String Variables

```
private string player1Name = "";
private string player2Name = "";
private string currentPage = "MainMenu";
string debugInfo = $"Page: {currentPage}, Selector Position: {selectorPosition.X}, {selectorPosition.Y}";
```

- Integer Variables

```
private int borderColor = 1;
private int screenWidth = 400;
private int screenHeight = 100;
private int moveCooldown = 10;
private int moveTime = 0;
private int typeCooldown = 8;
private int typeTime = 0;
private int enterCooldown = 8;
private int enterTime = 0;
private int delCooldown = 8;
private int delTime = 0;
public int width = 400;
public int height = 100;
private int borderColor = 1;
private int screenWidth = 400;
private int screenHeight = 100;
public int playerOneColor = 4;
public int playerTwoColor = 3;
public int playerOneLife = 5;
public int playerTwoLife = 5;
private int attackCooldownFramesOne = 30;
private int attackTimeOne = 0;
private int attackCooldownFramesTwo = 30;
private int attackTimeTwo = 0;
```

- Boolean Variables

```
public bool isPlaying = true;
private bool inputName1 = false;
private bool inputName2 = false;
public bool isSinglePlayer = true;
public bool isOnePlayer;
public bool isTwoPlayer;
```

- Point Variables

```
private Point selectorPosition = new Point(165, 25);
public Point playerOnePosition;
public Point playerTwoPosition;
Point newBullet = new Point(bullets[i].X + 1, bullets[i].Y);
Point playerHitBox = new Point(item.X + position.X, item.Y + position.Y);
```

2. Control Structures (if statements, loops):

```
if (isPlaying == true)
{
    player.Update();

    player.Render();
}
else
{
    menu.Render();
    menu.Update();
}


switch (currentPage)
{
   case "MainMenu":
      RenderMainMenu();
      RenderSelector();
      break;
   case "1Player":
      Render1PlayerMenu();
      if (!inputName1) RenderSelector();
      break;
   case "2Player":
      Render2PlayerMenu();
      if (!inputName1 && !inputName2) RenderSelector();
      break;
   case "Scores":
      break;
   case "Survival":
      RenderSurvival();
      break;
}

for (int x = 0; x < screenWidth; x++)
{
   engine.SetPixel(new Point(x, 0), borderColor, ConsoleCharacter.Full);
   engine.SetPixel(new Point(x, screenHeight - 1), borderColor, ConsoleCharacter.Full);
}

for (int y = 0; y < screenHeight; y++)
{
   engine.SetPixel(new Point(0, y), borderColor, ConsoleCharacter.Full);
   engine.SetPixel(new Point(screenWidth - 1, y), borderColor, ConsoleCharacter.Full);
}

foreach (var item in selector)
{
```

```
    engine.SetPixel(new Point(item.X + selectorPosition.X, item.Y + selectorPosition.Y), 2,
ConsoleCharacter.Full);
}

for (int i = 0; i < menuOptions.Length; i++)
{
    engine.WriteFiglet(new Point(170, 25 + (i * 5)), menuOptions[i], font1, 2);
}

if (engine.GetKey(ConsoleKey.Enter) && player1Name != "" && enterTime == 0)
{
    enterTime = enterCooldown;
    inputName1 = false;
}

if (enterTime > 0) enterTime--;

switch (currentPage)
{
    case "MainMenu":
        HandleMainMenuInput();
        break;
    case "1Player":
        Handle1PlayerMenuInput();
        break;
}

if (engine.GetKey(ConsoleKey.W) && selectorPosition.Y > 25 && CanType(ref moveTime,
moveCooldown))
{
    selectorPosition.Y -= 5;
}
else if (engine.GetKey(ConsoleKey.S) && selectorPosition.Y < 40 && CanType(ref moveTime,
moveCooldown))
{
    selectorPosition.Y += 5;
}

if (engine.GetKey(ConsoleKey.Enter) && enterTime == 0)
{
    enterTime = enterCooldown;
    switch (selectorPosition.Y)
    {
        case 25: currentPage = "1Player"; inputName1 = true; break;
        case 30: currentPage = "2Player"; inputName1 = inputName2 = true; break;
        case 35: currentPage = "Scores"; break;
        case 40: ExitGame(); break;
    }
}
```

```
if (engine.GetKey(ConsoleKey.W) && selectorPosition.Y > 30 && CanType(ref moveTime,
moveCooldown))
{
   selectorPosition.Y -= 5;
}
else if (engine.GetKey(ConsoleKey.S) && selectorPosition.Y < 35 && CanType(ref moveTime,
moveCooldown))
{
   selectorPosition.Y += 5;
}

if (engine.GetKey(ConsoleKey.Enter) && enterTime == 0)
{
   enterTime = enterCooldown;
   if (selectorPosition.Y == 30)
   {
      currentPage = "Survival";
   }
   else if (selectorPosition.Y == 35)
   {
      ResetToMainMenu();
   }
}

if (moveTime == 0)
{
   moveTime = moveCooldown;
   return true;
}
if (moveTime > 0) moveTime--;
return false;

if (!inputName1 || (!inputName1 && inputName2)) return;

if (keyboardState.IsPressed(Key.Back) && CanType(ref delTime, delCooldown))
{
   try
   {
      player1Name = player1Name.Substring(0, player1Name.Length - 1);
   }
   catch (System.ArgumentOutOfRangeException) { }
}

for (int i = 0; i < 256; i++)
{
   if (keyboardState.IsPressed((Key)i) && CanType(ref typeTime, typeCooldown))
   {
      char keyChar = GetCharacterFromKey((Key)i);
```

```
        if (keyChar != '\0' && player1Name.Length < 10)
        {
            player1Name += keyChar;
        }
    }
}

switch (key)
{
    case Key.A: return 'A';
    case Key.B: return 'B';
    case Key.C: return 'C';
    case Key.D: return 'D';
    case Key.E: return 'E';
    case Key.F: return 'F';
    case Key.G: return 'G';
    case Key.H: return 'H';
    case Key.I: return 'I';
    case Key.J: return 'J';
    case Key.K: return 'K';
    case Key.L: return 'L';
    case Key.M: return 'M';
    case Key.N: return 'N';
    case Key.O: return 'O';
    case Key.P: return 'P';
    case Key.Q: return 'Q';
    case Key.R: return 'R';
    case Key.S: return 'S';
    case Key.T: return 'T';
    case Key.U: return 'U';
    case Key.V: return 'V';
    case Key.W: return 'W';
    case Key.X: return 'X';
    case Key.Y: return 'Y';
    case Key.Z: return 'Z';

    default: return '\0';
}

if (isSinglePlayer)
{
    isOnePlayer = true;
    playerOnePosition = initialPosition;
}
else
{
    isOnePlayer = true;
    isTwoPlayer = true;
```

```
    playerOnePosition = initialPosition;
    playerTwoPosition = new Point(initialPosition.X + 10, initialPosition.Y);
}

if (keyboardState == null)
    return;
if (keyboardState.IsPressed(Key.W) && playerOnePosition.Y > 1) playerOnePosition.Y--;
if (keyboardState.IsPressed(Key.S) && playerOnePosition.Y < screenHeight - 5) playerOnePosition.Y++;
if (keyboardState.IsPressed(Key.A) && playerOnePosition.X > 1) playerOnePosition.X--;
if (keyboardState.IsPressed(Key.D) && playerOnePosition.X < screenWidth - 5) playerOnePosition.X++;

if (keyboardState.IsPressed(Key.Space) && CanAttack(ref attackTimeOne, attackCooldownFramesOne,
ref attackPressedOne))
{
    Point newBullet = new Point(playerOnePosition.X, playerOnePosition.Y + 3);
    playerOneBullets.Add(newBullet);
}

if (isTwoPlayer)
{
    if (keyboardState.IsPressed(Key.Up) && playerTwoPosition.Y > 1) playerTwoPosition.Y--;
    if (keyboardState.IsPressed(Key.Down) && playerTwoPosition.Y < screenHeight - 7)
playerTwoPosition.Y++;
    if (keyboardState.IsPressed(Key.Left) && playerTwoPosition.X > 1) playerTwoPosition.X--;
    if (keyboardState.IsPressed(Key.Right) && playerTwoPosition.X < screenWidth - 7)
playerTwoPosition.X++;

    if (keyboardState.IsPressed(Key.RightControl) && CanAttack(ref attackTimeTwo,
attackCooldownFramesTwo, ref attackPressedTwo))
    {
        Point newBullet = new Point(playerTwoPosition.X, playerTwoPosition.Y + 3);
        playerTwoBullets.Add(newBullet);
    }
}

if (attackTime == 0)
{
    attackTime = cooldownFrames;
    attackPressed = true;
    return true;
}

if (attackTime > 0) attackTime--;

for (int i = bullets.Count - 1; i >= 0; i--)
{
    Point newBullet = new Point(bullets[i].X + 1, bullets[i].Y);

    if (newBullet.Y >= 0)
```

```
    {
        bullets[i] = newBullet;
    }
    else
    {
        bullets.RemoveAt(i);
    }
}

foreach (var item in player)
{
    Point playerHitBox = new Point(item.X + position.X, item.Y + position.Y);
    playerHitbox.Add(playerHitBox);
    engine.SetPixel(playerHitBox, color, ConsoleCharacter.Full);
}

foreach (var bullet in bullets)
{
    engine.SetPixel(bullet, color, ConsoleCharacter.Full);
}

for (int x = 0; x < screenWidth; x++)
{
    engine.SetPixel(new Point(x, 0), borderColor, ConsoleCharacter.Full);
    engine.SetPixel(new Point(x, screenHeight - 1), borderColor, ConsoleCharacter.Full);
}

for (int y = 0; y < screenHeight; y++)
{
    engine.SetPixel(new Point(0, y), borderColor, ConsoleCharacter.Full);
    engine.SetPixel(new Point(screenWidth - 1, y), borderColor, ConsoleCharacter.Full);
}
```

3. Functions/Methods:

```
public override void Create()
{
    Console.SetBufferSize(width, height);
    Engine.SetPalette(Palettes.Pico8);
    Engine.Borderless();
    Console.Title = "GAMER!!";
    TargetFramerate = 60;
    menu = new MainMenu(Engine);
    player = new Player(Engine, new Point(10, 10), menu.isSinglePlayer);


    // Set a timer for the game loop (running every frame)
    timer = new Timer(UpdateScreen, null, 0, 1000 / TargetFramerate);
```

```
}

private void UpdateScreen(object state)
{
    if (isPlaying == true)
    {
        // Update the player and enemies
        player.Update();

        // Render the player and enemies
        player.Render();
    }

    menu.Render();
    menu.Update();
}

public void Render()
{
    LoadFonts();

    engine.ClearBuffer();

    RenderBorder();
    GameTitle();

    switch (currentPage)
    {
        case "MainMenu":
            RenderMainMenu();
            RenderSelector();
            break;
        case "1Player":
            Render1PlayerMenu();
            if (!inputName1) RenderSelector();
            break;
        case "2Player":
            Render2PlayerMenu();
            if (!inputName1 && !inputName2) RenderSelector();
            break;
        case "Scores":
            // Additional rendering for other pages can go here
            break;
        case "Survival":
            RenderSurvival();
            break;
    }
```

```
    RenderDebugInfo();
    engine.DisplayBuffer();
}

private void LoadFonts()
{
    font = FigletFont.Load("C:\\Users\\Styx\\Desktop\\ITEC102FinalMain\\_Game_Main\\3d.flf");
    font1 = FigletFont.Load("C:\\Users\\Styx\\Desktop\\ITEC102FinalMain\\_Game_Main\\smslant.flf");
}

private void RenderBorder ()
{
    for (int x = 0; x < screenWidth; x++)
    {
        engine.SetPixel(new Point(x, 0), borderColor, ConsoleCharacter.Full);
        engine.SetPixel(new Point(x, screenHeight - 1), borderColor, ConsoleCharacter.Full);
    }

    // Render left and right borders
    for (int y = 0; y < screenHeight; y++)
    {
        engine.SetPixel(new Point(0, y), borderColor, ConsoleCharacter.Full);
        engine.SetPixel(new Point(screenWidth - 1, y), borderColor, ConsoleCharacter.Full);
    }
}

private void RenderSelector()
{
    foreach (var item in selector)
    {
        engine.SetPixel(new Point(item.X + selectorPosition.X, item.Y + selectorPosition.Y), 2,
ConsoleCharacter.Full);
    }
}

private void GameTitle()
{
    engine.WriteFiglet(new Point(140, 10), "VOID  INVADER", font, 2);
}

private void RenderMainMenu()
{
    string[] menuOptions = { "1-Player", "2-Player", "Scores", "Exit" };
    for (int i = 0; i < menuOptions.Length; i++)
    {
        engine.WriteFiglet(new Point(170, 25 + (i * 5)), menuOptions[i], font1, 2);
    }
}

private void Render1PlayerMenu()
```

```
{
    engine.WriteFiglet(new Point(85, 25), "Enter your name:", font1, 2);
    engine.WriteFiglet(new Point(180, 25), player1Name, font1, 2);
    engine.WriteFiglet(new Point(170, 30), "Survival", font1, 2);
    engine.WriteFiglet(new Point(170, 35), "Back", font1, 2);
    if (engine.GetKey(ConsoleKey.Enter) && player1Name != "" && enterTime == 0)
    {
        enterTime = enterCooldown;
        inputName1 = false;
    }
}

private void Render2PlayerMenu()
{
    // nothing to see here
}

private void RenderSurvival()
{
    // nothing to see here

}

private void RenderScores()
{
    // nothing to see here
}

private void RenderDebugInfo()
{
    string debugInfo = $"Page: {currentPage}, Selector Position: {selectorPosition.X}, {selectorPosition.Y}";
    engine.WriteText(new Point(10, 10), debugInfo, 2);
}

public void Update()
{
    keyboardState = keyboard.GetCurrentState();

    // Reduce the cooldown timer for Enter key
    if (enterTime > 0) enterTime--;

    switch (currentPage)
    {
        case "MainMenu":
            HandleMainMenuInput();
            break;
        case "1Player":
            Handle1PlayerMenuInput();
            break;
```

```
    }

    HandleKeyboardInput();
}

private void HandleMainMenuInput()
{
    if (engine.GetKey(ConsoleKey.W) && selectorPosition.Y > 25 && CanType(ref moveTime,
moveCooldown))
    {
        selectorPosition.Y -= 5;
    }
    else if (engine.GetKey(ConsoleKey.S) && selectorPosition.Y < 40 && CanType(ref moveTime,
moveCooldown))
    {
        selectorPosition.Y += 5;
    }

    if (engine.GetKey(ConsoleKey.Enter) && enterTime == 0)
    {
        enterTime = enterCooldown;
        switch (selectorPosition.Y)
        {
            case 25: currentPage = "1Player"; inputName1 = true; break;
            case 30: currentPage = "2Player"; inputName1 = inputName2 = true; break;
            case 35: currentPage = "Scores"; break;
            case 40: ExitGame(); break;
        }
    }
}

private void Handle1PlayerMenuInput()
{
    if (engine.GetKey(ConsoleKey.W) && selectorPosition.Y > 30 && CanType(ref moveTime,
moveCooldown))
    {
        selectorPosition.Y -= 5;
    }
    else if (engine.GetKey(ConsoleKey.S) && selectorPosition.Y < 35 && CanType(ref moveTime,
moveCooldown))
    {
        selectorPosition.Y += 5;
    }

    if (engine.GetKey(ConsoleKey.Enter) && enterTime == 0)
    {
        enterTime = enterCooldown;
        if (selectorPosition.Y == 30)
```

```
        {
            currentPage = "Survival";
        }
        else if (selectorPosition.Y == 35)
        {
            ResetToMainMenu();
        }
    }
}

private void Handle2PlayerMenuInput()
{
    // nothing to see here
}

private bool CanType(ref int moveTime, int moveCooldown)
{
    if (moveTime == 0)
    {
        moveTime = moveCooldown;
        return true;
    }
    if (moveTime > 0) moveTime--;
    return false;
}

private void ResetToMainMenu()
{
    currentPage = "MainMenu";
    selectorPosition = new Point(165, 25);
    player1Name = "";
    inputName1 = false;
}

private void ExitGame()
{
    keyboard.Unacquire();
    directInput.Dispose();
    Environment.Exit(0);
}
private void HandleKeyboardInput()
{

    if (!inputName1 || (!inputName1 && inputName2)) return;

    if (keyboardState.IsPressed(Key.Back) && CanType(ref delTime, delCooldown))
    {
        try
        {
```

```
            player1Name = player1Name.Substring(0, player1Name.Length - 1); // Remove last character
        }
        catch (System.ArgumentOutOfRangeException) { }
    }

    // Iterate over all keys to see if they are pressed
    for (int i = 0; i < 256; i++)
    {
        if (keyboardState.IsPressed((Key)i) && CanType(ref typeTime, typeCooldown))
        {
            char keyChar = GetCharacterFromKey((Key)i);
            if (keyChar != '\0' && player1Name.Length < 10)
            {
                player1Name += keyChar;
            }
        }
    }
}

private char GetCharacterFromKey(Key key)
{
    switch (key)
    {
        case Key.A: return 'A';
        case Key.B: return 'B';
        case Key.C: return 'C';
        case Key.D: return 'D';
        case Key.E: return 'E';
        case Key.F: return 'F';
        case Key.G: return 'G';
        case Key.H: return 'H';
        case Key.I: return 'I';
        case Key.J: return 'J';
        case Key.K: return 'K';
        case Key.L: return 'L';
        case Key.M: return 'M';
        case Key.N: return 'N';
        case Key.O: return 'O';
        case Key.P: return 'P';
        case Key.Q: return 'Q';
        case Key.R: return 'R';
        case Key.S: return 'S';
        case Key.T: return 'T';
        case Key.U: return 'U';
        case Key.V: return 'V';
        case Key.W: return 'W';
        case Key.X: return 'X';
        case Key.Y: return 'Y';
```

```
            case Key.Z: return 'Z';

            default: return '\0'; // Return null character for unrecognized keys
        }
    }

    public void Update()
    {
        keyboardState = keyboard.GetCurrentState();
        if (keyboardState == null)
            return;

        // Player One Controls
        if (keyboardState.IsPressed(Key.W) && playerOnePosition.Y > 1) playerOnePosition.Y--;
        if (keyboardState.IsPressed(Key.S) && playerOnePosition.Y < screenHeight - 5) playerOnePosition.Y++;
    // Adjusted for player height
        if (keyboardState.IsPressed(Key.A) && playerOnePosition.X > 1) playerOnePosition.X--;
        if (keyboardState.IsPressed(Key.D) && playerOnePosition.X < screenWidth - 5) playerOnePosition.X++;
    // Adjusted for player width

        // Player One Shooting
        if (keyboardState.IsPressed(Key.Space) && CanAttack(ref attackTimeOne, attackCooldownFramesOne,
    ref attackPressedOne))
        {
            Point newBullet = new Point(playerOnePosition.X, playerOnePosition.Y + 3);
            playerOneBullets.Add(newBullet);
        }

        // Player Two Controls (if two players are enabled)
        if (isTwoPlayer)
        {
            if (keyboardState.IsPressed(Key.Up) && playerTwoPosition.Y > 1) playerTwoPosition.Y--;
            if (keyboardState.IsPressed(Key.Down) && playerTwoPosition.Y < screenHeight - 7)
    playerTwoPosition.Y++;
            if (keyboardState.IsPressed(Key.Left) && playerTwoPosition.X > 1) playerTwoPosition.X--;
            if (keyboardState.IsPressed(Key.Right) && playerTwoPosition.X < screenWidth - 7)
    playerTwoPosition.X++;

            // Player Two Shooting
            if (keyboardState.IsPressed(Key.RightControl) && CanAttack(ref attackTimeTwo,
    attackCooldownFramesTwo, ref attackPressedTwo))
            {
                Point newBullet = new Point(playerTwoPosition.X, playerTwoPosition.Y + 3);
                playerTwoBullets.Add(newBullet);
            }
        }

        UpdateBullets(playerOneBullets);
```

```
    UpdateBullets(playerTwoBullets);
}

private bool CanAttack(ref int attackTime, int cooldownFrames, ref bool attackPressed)
{
    if (attackTime == 0)
    {
        attackTime = cooldownFrames;
        attackPressed = true;
        return true;
    }
    attackPressed = false;
    if (attackTime > 0) attackTime--;
    return false;
}

private void UpdateBullets(List<Point> bullets)
{
    for (int i = bullets.Count - 1; i >= 0; i--)
    {
        Point newBullet = new Point(bullets[i].X + 1, bullets[i].Y);

        // Check if the bullet is still within the screen bounds
        if (newBullet.Y >= 0)
        {
            bullets[i] = newBullet;
        }
        else
        {
            bullets.RemoveAt(i); // Remove bullet if it goes off-screen
        }
    }
}
public void Render()
{
    engine.ClearBuffer();
    RenderBorder();
    RenderPlayer(playerOne, playerOnePosition, playerOneColor); // for Player One
    RenderPlayer(playerTwo, playerTwoPosition, playerTwoColor); // for Player Two
    RenderBullets(playerOneBullets, playerOneColor); // for Player One Bullets
    RenderBullets(playerTwoBullets, playerTwoColor); // for Player Two Bullets
    engine.DisplayBuffer();
}
public void RenderPlayer(Point[] player, Point position, int color)
{
    List<Point> playerHitbox = new List<Point>();

    // Add the player's segments to the hitbox
```

```
    foreach (var item in player)
    {
        Point playerHitBox = new Point(item.X + position.X, item.Y + position.Y);
        playerHitbox.Add(playerHitBox); // Add each calculated segment to the hitbox list
        engine.SetPixel(playerHitBox, color, ConsoleCharacter.Full); // Render the segment on the screen
    }
}
private void RenderBullets(List<Point> bullets, int color)
{
    foreach (var bullet in bullets)
    {
        engine.SetPixel(bullet, color, ConsoleCharacter.Full);
    }
}
private void RenderBorder()
{

    // Render top and bottom borders
    for (int x = 0; x < screenWidth; x++)
    {
        engine.SetPixel(new Point(x, 0), borderColor, ConsoleCharacter.Full);          // Top border
        engine.SetPixel(new Point(x, screenHeight - 1), borderColor, ConsoleCharacter.Full);   // Bottom
border
    }

    // Render left and right borders
    for (int y = 0; y < screenHeight; y++)
    {
        engine.SetPixel(new Point(0, y), borderColor, ConsoleCharacter.Full);          // Left border
        engine.SetPixel(new Point(screenWidth - 1, y), borderColor, ConsoleCharacter.Full);    // Right
border
    }
}
public void Dispose()
{
    keyboard.Unacquire();
    keyboard.Dispose();
    directInput.Dispose();
}
```

4. Arrays or Collections:

```
    string[] menuOptions = { "1-Player", "2-Player", "Scores", "Exit" };
    public List<Point> playerOneBullets = new List<Point>();
    public List<Point> playerTwoBullets = new List<Point>();
```

```
   private Point[] selector = {new Point(0, 1), new Point(0, 2), new Point(0, 3), new Point(0, 4), new
Point(56, 1), new Point(56, 2), new Point(56, 3), new Point(56, 4)};
   public Point[] playerOne = {
   new Point(0,0),
   new Point(0,1), new Point(1,1), new Point(2,1),
   new Point(0,2), new Point(1,2), new Point(2,2), new Point(3,2),
   new Point(0,3), new Point(1,3), new Point(2,3), new Point(3,3), new Point(4,3),
   new Point(0,4), new Point(1,4), new Point(2,4), new Point(3,4),
   new Point(0,5), new Point(1,5), new Point(2,5),
   new Point(0,6)
         };
   public Point[] playerTwo = {
   new Point(0,0),
   new Point(0,1), new Point(1,1), new Point(2,1),
   new Point(0,2), new Point(1,2), new Point(2,2), new Point(3,2),
   new Point(0,3), new Point(1,3), new Point(2,3), new Point(3,3), new Point(4,3),
   new Point(0,4), new Point(1,4), new Point(2,4), new Point(3,4),
   new Point(0,5), new Point(1,5), new Point(2,5),
   new Point(0,6)
         };

   List<Point> playerHitbox = new List<Point>();
```

CONCLUSION

SUMMARY

Summarize the main points covered in the documentation.

FUTURE ENHANCEMENTS

1. Implement Enemy

This feature involves creating the enemy characters that will interact with the player within the game.

- **Enemy Class**: Define an Enemy class that encapsulates properties such as position, health, speed, and behavior (e.g., movement patterns, attack methods).
- **Rendering**: Implement methods to visually represent enemies on the screen, using graphics or console characters.
- **Collision Detection**: Implement logic to detect collisions between the player and enemies, triggering appropriate responses (e.g., reducing player life).

2. Implement Enemy Spawning

This feature focuses on creating a system for generating enemies at specific intervals or locations within the game. Key components include:

- **Spawning Logic**: Develop a method to spawn enemies at random or predetermined locations on the game map.
- **Spawn Timing**: Implement a timer or counter that controls how often enemies are spawned, allowing for increasing difficulty as the game progresses.
- **Spawn Limits**: Set limits on the number of enemies that can be active at any given time to manage performance and gameplay balance.
- **Variety of Enemies**: Consider implementing different types of enemies with varying attributes and behaviors to enhance gameplay diversity.

3. Player Life

This feature involves managing the player's health and lives throughout the game. Key aspects include:

- **Health System**: Implement a health system that reduces the player's life when they take damage from enemies or obstacles.
- **Game Over Logic**: Develop logic to handle what happens when the player runs out of lives, such as displaying a game over screen and offering options to restart or exit.
- **Health Recovery**: Optionally, implement items or power-ups that allow players to regain health or extra lives.

4. Implement All Not Implemented Functions/Methods

ACKNOWLEDGMENTS

SharpDx
MonoGame.Framework.DesktopGL
MonoGame.Framework.WindowsDX
ConsoleGameEngine (https://github.com/ollelogdahl/ConsoleGameEngine)

https://www.youtube.com/@javidx9 //nice tutorials regarding consoles :)