Title: Development of a 2D Shooter Game in C++ Using Win32 API and GDI+

Name: MD Arafat Islam Roche

Student ID:202101009045

Abstract

This report documents the creation of a 2D shooter game developed in C++ using the Win32 API and GDI+ for rendering graphics. The project aimed to build a basic game where a player controls a spaceship that can shoot bullets to destroy alien invaders. We faced several challenges during the development process, including issues with rendering performance, optimizing game speed, handling screen flicker, and implementing custom assets. This report outlines the step-by-step process of development, the errors encountered, the solutions applied, and insights gained from the experience.

Introduction

The goal of this project was to create a simple, playable 2D shooter game using native C++ and Win32 API, without relying on modern game development libraries. This approach allowed for a deeper understanding of rendering graphics with GDI+, handling game logic, and implementing custom timers. The game features a player-controlled spaceship, shooting mechanics, custom alien enemies, and a scrolling background. The project was divided into multiple stages: setting up a basic Win32 window, rendering sprites, implementing game mechanics, optimizing performance, and adding custom assets.

Development Process

Stage 1: Initial Setup of the Game Window

The first step was setting up a basic Win32 window for the game. This window served as the foundation for rendering game objects and handling player input. The Win32 `WNDCLASSEX` structure was defined, and `WM_PAINT` was used to refresh the window with each game frame. The primary goal was to establish a clear, responsive game window that could handle continuous input and rendering without lag.

Code Example: Initializing the Window

```
WNDCLASSEX wcex;
wcex.cbSize = sizeof(WNDCLASSEX);
wcex.style = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc = WndProc;
wcex.cbClsExtra = 0;
wcex.cbWndExtra = 0;
wcex.hInstance = hInstance;
wcex.hIcon = LoadIcon(nullptr, IDI_APPLICATION);
wcex.hCursor = LoadCursor(nullptr, IDC_ARROW);
wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wcex.lpszMenuName = nullptr;
```

```
wcex.lpszClassName = L"MainWndClass";
wcex.hIconSm = LoadIcon(nullptr, IDI_APPLICATION);
```

Explanation: This code initializes the window class, specifying details such as the cursor, icon, and background color. Setting `lpfnWndProc` to `WndProc` ensures that the `WndProc` function will handle all incoming messages, including rendering, input, and timer events.

Stage 2: Adding GDI+ for Rendering Graphics

To draw 2D objects in the game, GDI+ was chosen as the rendering library, which allows for relatively easy drawing of images and shapes. We loaded custom images for the spaceship, alien enemies, and the background using `Gdiplus::Image`. We used a rendering function to handle these images and drew bullets as simple colored ellipses.

Code Example: Rendering the Game

```
void Render(HDC hdc) {
    Graphics graphics(hdc);

    // Draw background
    if (backgroundImage) {
        graphics.DrawImage(backgroundImage, 0, 0, 800, 600);
    }

    // Draw player
    if (spaceshipImage) {
        graphics.DrawImage(spaceshipImage, static_cast<int>(playerX),
static_cast<int>(playerY), 40, 40);
    }

    // Draw bullets
    SolidBrush bulletBrush(Color(255, 255, 0, 0));
    for (auto &bullet : bullets) {
        graphics.FillEllipse(&bulletBrush, static_cast<int>(bullet.x),
static_cast<int>(bullet.y), 5, 10);
    }

    // Draw enemies
    if (alienImage) {
        for (auto &enemy : enemies) {
            if (enemy.alive) {
                graphics.DrawImage(alienImage, static_cast<int>(enemy.x),
static_cast<int>(enemy.y), 30, 30);
            }
        }
    }
}
```

Explanation: This function uses GDI+ to render each game element: background, player spaceship, bullets, and alien enemies. By creating separate sections for each element, we could easily manage which objects appeared in each frame, ensuring that the graphics update smoothly.

Stage 3: Implementing Player Controls

Player control was achieved by using the `WM_KEYDOWN` message to track arrow key inputs, enabling horizontal movement of the spaceship. A `VK_SPACE` key press was used to trigger shooting, allowing the player to fire bullets toward the alien enemies. This laid the groundwork for the game mechanics, where each bullet would act as a projectile intended to hit the aliens.

Code Example: Handling Player Input

```
case WM_KEYDOWN:
    switch (wParam) {
    case VK_LEFT:
        playerX -= playerSpeed;
        break;
    case VK_RIGHT:
        playerX += playerSpeed;
        break;
    case VK_SPACE:
        ShootBullet();
        break;
    }
    break;
```

Explanation: This code in the `WndProc` function handles player input for left and right movement as well as shooting. By associating specific key presses with each action, the player can control the spaceship intuitively. The `ShootBullet` function generates a new bullet that travels upward from the spaceship's current position.

Stage 4: Implementing Shooting and Collisions

Bullets were implemented as individual objects, each moving upward with a specified speed. Collisions were handled by calculating the Euclidean distance between each bullet and alien. When this distance fell below a certain threshold, the bullet would be removed, and the alien would be marked as "dead." This simple collision detection allowed for basic interactivity and challenge in the gameplay.

Challenges Faced and Solutions

Challenge 1: Screen Flickering

Issue: The game window experienced noticeable flickering due to the immediate rendering approach, where `InvalidateRect` and `WM_PAINT` would refresh the screen without buffering. This was especially visible as more sprites and background elements were added, creating an inconsistent and distracting experience for the player.

Solution: Double buffering was introduced to solve this issue. By creating an off-screen buffer using a compatible memory device context (`CreateCompatibleDC` and `CreateCompatibleBitmap`), we rendered the game objects to the memory DC first and then transferred the rendered image to the main device context using `BitBlt`. This eliminated the flickering and provided a smoother visual experience.

Code Example: Implementing Double Buffering

```
void RenderDoubleBuffer(HDC hdc) {
    HDC memDC = CreateCompatibleDC(hdc);
    HBITMAP memBitmap = CreateCompatibleBitmap(hdc, 800, 600);
    SelectObject(memDC, memBitmap);

    Graphics graphics(memDC);
    Render(memDC);

    BitBlt(hdc, 0, 0, 800, 600, memDC, 0, 0, SRCCOPY);

    DeleteObject(memBitmap);
    DeleteDC(memDC);
}
```

Explanation: This function sets up a memory DC (`memDC`) as an off-screen buffer, where we draw the entire game frame. After completing the rendering, `BitBlt` copies this buffer to the actual device context (`hdc`). This approach prevented flickering by eliminating the need to render directly to the screen.

Challenge 2: Slow Performance

Issue: As more elements were added, including multiple bullets, alien sprites, and continuous background rendering, the game began to slow down. This slowdown was exacerbated by the use of `Sleep` for frame control, which can create significant delays and make gameplay less responsive.

Solution: We replaced `Sleep` with a timer-based update system using `SetTimer`. By setting a timer with a 16-millisecond interval (to achieve around 60 FPS), the game loop could refresh consistently without freezing or delays. This allowed the game to maintain a steady frame rate while updating bullet positions, enemy spawns, and player input more fluidly.

Challenge 3: Alien Spawning

Issue: Initially, aliens were not appearing as intended due to issues in the spawning logic. Aliens would sometimes overlap, fail to spawn consistently, or appear off-screen, making the game too easy or unplayable.

Solution: To address this, we created a timer-based enemy spawning mechanism with `SetTimer`. Every 800 milliseconds, an alien would spawn at a random horizontal position near the top of the screen. The addition of a `y` position and a movement speed for each alien allowed them to move downward, making the gameplay more dynamic.

Game Optimizations

After addressing the primary issues, we further optimized the game by increasing the speed of player movement, bullet velocity, and alien downward movement speed. This was done to enhance gameplay and create a more challenging experience. Adjustments included:

1. **Bullet Speed**: Increased from 10 to 20 to make shooting more responsive.
2. **Player Movement**: Increased the spaceship speed to make dodging aliens easier.
3. **Enemy Speed**: Added a downward movement to enemies to make gameplay more challenging.

Code Example: Bullet and Enemy Speed Optimization

```
for (auto &bullet : bullets) {
    bullet.y -= bullet.speed;
}

for (auto &enemy : enemies) {
    if (enemy.alive) {
        enemy.y += enemy.speed;
    }
}
```

Explanation: In the `Update` function, each bullet's `y` position is decreased by a higher speed value, making them travel faster. Additionally, the enemy's `y` position is incremented by a speed factor, allowing enemies to move downward at a rate that challenges the player's reactions.

Errors Encountered

1. **Rendering Artifacts**: Early attempts at rendering caused some graphical artifacts due to incomplete double buffering. This was resolved by carefully implementing the memory DC and ensuring it was copied entirely to the main DC.
2. **Timer Conflicts**: With multiple timers (one for frame update and one for alien spawning), there were occasional conflicts. We used separate timer IDs and ensured each timer controlled only one aspect of the game (one for frame refresh, the other for enemy spawning).
3. **Collision Detection Errors**: In the initial collision detection, small errors in the calculation caused bullets to miss aliens or register hits incorrectly. By refining the collision threshold, we achieved more accurate collision results.

Final Game Features & Code

The final version of the game includes:

- **Spaceship Control**: Smooth horizontal movement with responsive shooting.
- **Alien Invasion**: Aliens spawn periodically and move down the screen, providing a consistent challenge.
- **Bullet Firing and Collision**: Bullets move quickly and effectively register hits on aliens.
- **Custom Assets**: Spaceship, alien, and background images provide a unique look.
- **Double-Buffered Rendering**: Smooth visuals without flickering or tearing.

The complete code is given below:

```
#include <windows.h>
#include <gdiplus.h>
#include <vector>
#include <cmath>
#include <ctime>

#pragma comment (lib,"Gdiplus.lib")

using namespace Gdiplus;

// Global Variables
HINSTANCE hInst;
HWND hWnd;
bool isRunning = true;

struct Bullet {
    float x, y;
    float speed = 50.0f;  // Increase bullet speed
};

struct Enemy {
    float x, y;
    float speed = 2.0f;  // New variable to move enemies downwards
    bool alive = true;
};

std::vector<Bullet> bullets;
std::vector<Enemy> enemies;

// Player Variables
float playerX = 400.0f;
float playerY = 500.0f;
const int playerSpeed = 50;  // Increase player movement speed

// Images
Image* spaceshipImage = nullptr;
Image* alienImage = nullptr;
Image* backgroundImage = nullptr;

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

void LoadAssets() {
    spaceshipImage = new Image(L"spaceship.png");
    alienImage = new Image(L"alien.png");
    backgroundImage = new Image(L"background.png");
}

void ReleaseAssets() {
    delete spaceshipImage;
    delete alienImage;
    delete backgroundImage;
}
```

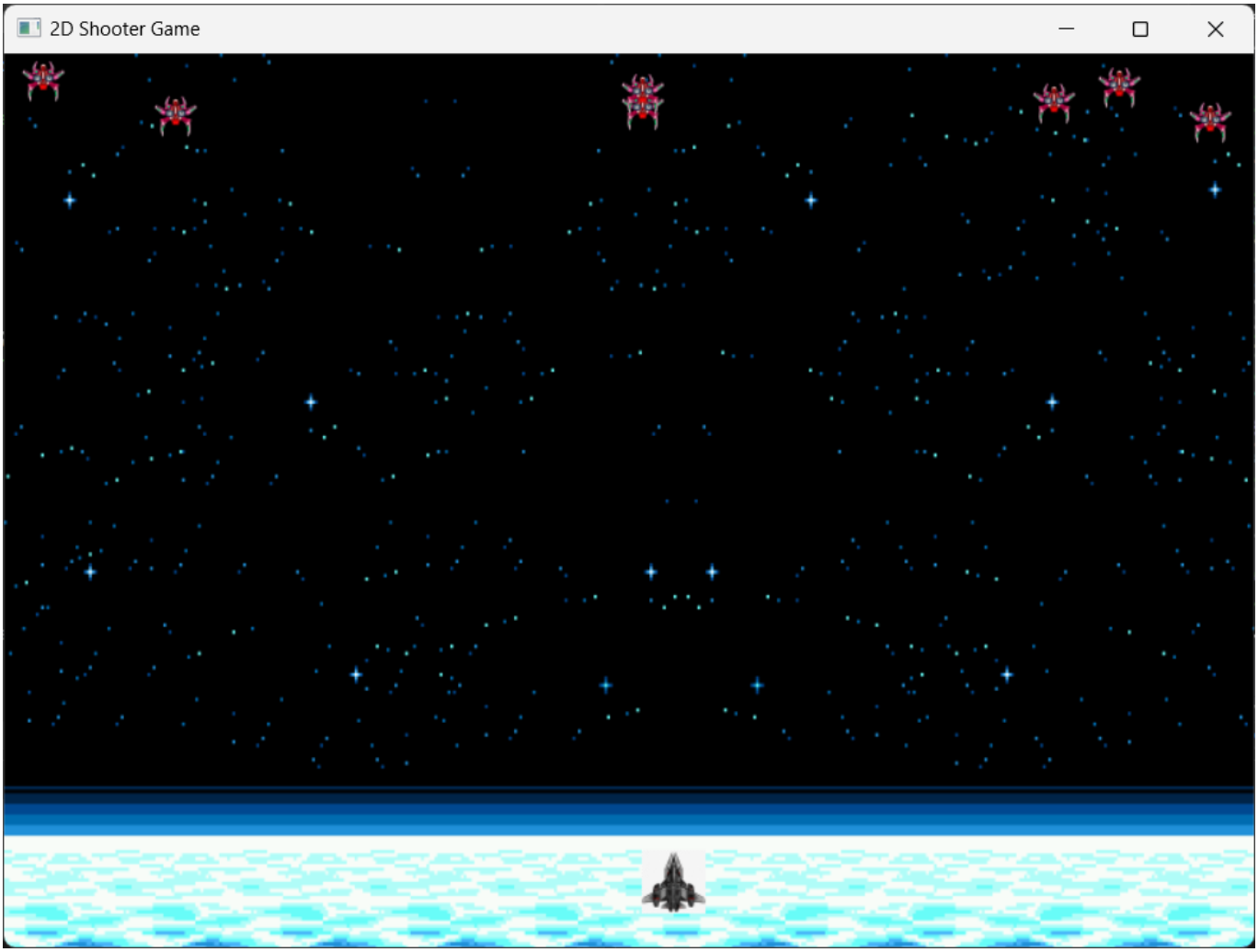
```
void Update() {
    // Update bullets
    for (auto& bullet : bullets) {
        bullet.y -= bullet.speed;
    }

    // Update enemies and check for collisions with bullets
    for (auto& enemy : enemies) {
        if (enemy.alive) {
            enemy.y += enemy.speed;  // Move enemy downwards

            for (auto& bullet : bullets) {
                float dx = bullet.x - enemy.x;
                float dy = bullet.y - enemy.y;
                float distance = std::sqrt(dx * dx + dy * dy);

                if (distance < 20.0f) {  // Collision radius
                    enemy.alive = false;
                    bullet.y = -1000;  // Remove bullet by moving it off-screen
                }
            }
        }
    }

    // Remove off-screen bullets and inactive enemies
    bullets.erase(std::remove_if(bullets.begin(), bullets.end(), [](const Bullet&
b) { return b.y < 0; }), bullets.end());
    enemies.erase(std::remove_if(enemies.begin(), enemies.end(), [](const Enemy&
e) { return !e.alive || e.y > 600; }), enemies.end());
}

void Render(HDC hdc) {
    Graphics graphics(hdc);

    // Draw background
    if (backgroundImage) {
        graphics.DrawImage(backgroundImage, 0, 0, 800, 600);
    }

    // Draw player
    if (spaceshipImage) {
        graphics.DrawImage(spaceshipImage, static_cast<int>(playerX),
static_cast<int>(playerY), 40, 40);
    }

    // Draw bullets
    SolidBrush bulletBrush(Color(255, 255, 0, 0));
    for (auto& bullet : bullets) {
        graphics.FillEllipse(&bulletBrush, static_cast<int>(bullet.x),
static_cast<int>(bullet.y), 5, 10);
    }

    // Draw enemies
    if (alienImage) {
```

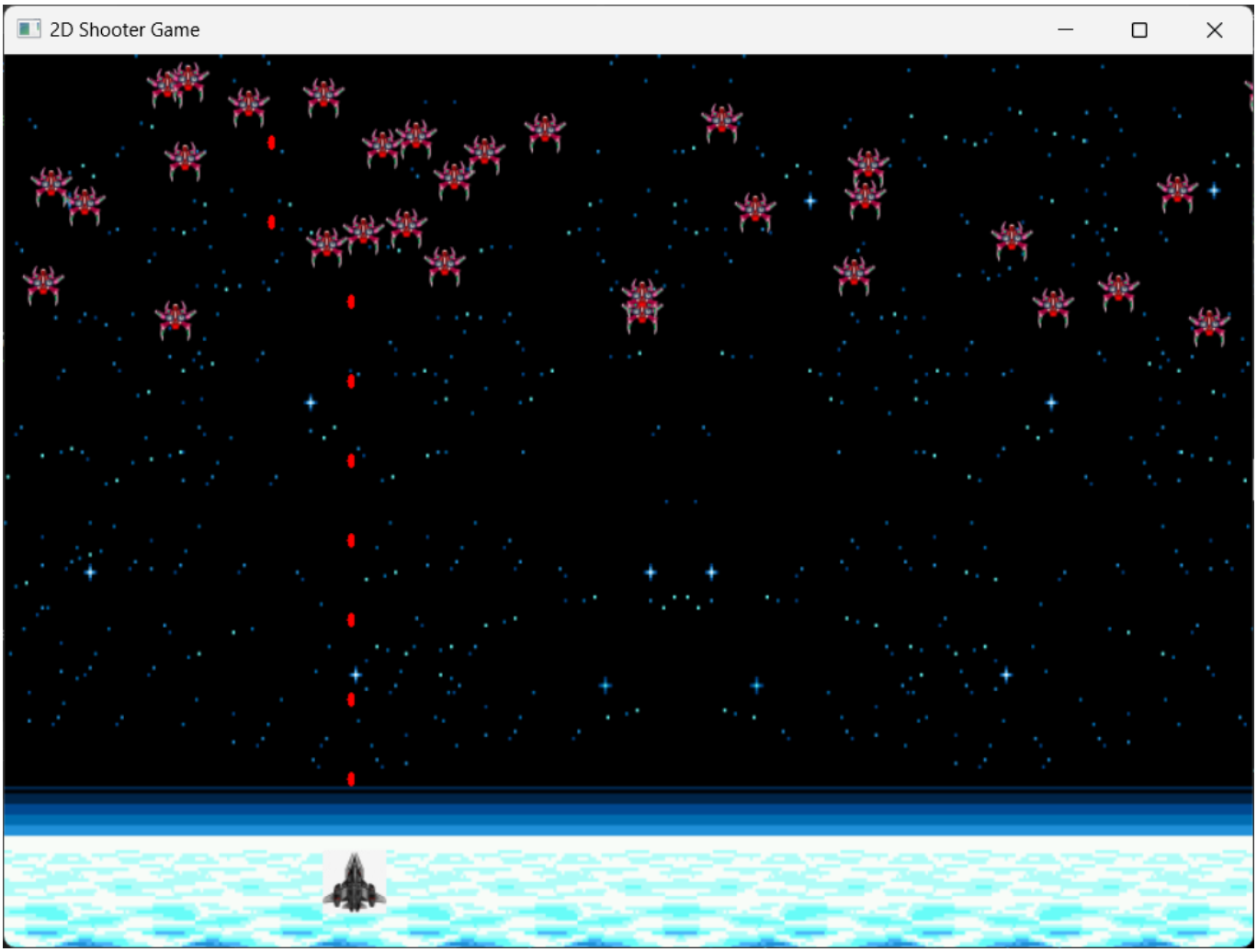
```
        for (auto& enemy : enemies) {
            if (enemy.alive) {
                graphics.DrawImage(alienImage, static_cast<int>(enemy.x),
static_cast<int>(enemy.y), 30, 30);
            }
        }
    }
}

void SpawnEnemy() {
    enemies.push_back({ float(rand() % 800), 0 });  // Start enemies from the top
(y = 0)
}

void ShootBullet() {
    bullets.push_back({ playerX + 15.0f, playerY });
}

void RenderDoubleBuffer(HDC hdc) {
    // Create a compatible memory device context
    HDC memDC = CreateCompatibleDC(hdc);

    // Create a compatible bitmap for double buffering
    HBITMAP memBitmap = CreateCompatibleBitmap(hdc, 800, 600);
    SelectObject(memDC, memBitmap);

    // Fill the background of the memory DC
    HBRUSH hBrush = CreateSolidBrush(RGB(0, 0, 0));
    RECT rect = { 0, 0, 800, 600 };
    FillRect(memDC, &rect, hBrush);
    DeleteObject(hBrush);

    // Use GDI+ to render to the memory DC
    Graphics graphics(memDC);
    Render(memDC);

    // Copy the memory DC to the actual device context
    BitBlt(hdc, 0, 0, 800, 600, memDC, 0, 0, SRCCOPY);

    // Cleanup
    DeleteObject(memBitmap);
    DeleteDC(memDC);
}

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow) {
    hInst = hInstance;
    MSG msg;

    // Register Window Class
    WNDCLASSEX wcex;
    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = WndProc;
    wcex.cbClsExtra = 0;
```

```
    wcex.cbWndExtra = 0;
    wcex.hInstance = hInstance;
    wcex.hIcon = LoadIcon(nullptr, IDI_APPLICATION);
    wcex.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wcex.lpszMenuName = nullptr;
    wcex.lpszClassName = L"MainWndClass";
    wcex.hIconSm = LoadIcon(nullptr, IDI_APPLICATION);

    RegisterClassEx(&wcex);

    // Create Window
    hWnd = CreateWindow(L"MainWndClass", L"2D Shooter Game", WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT, CW_USEDEFAULT, 800, 600, nullptr, nullptr, hInstance, nullptr);
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // Initialize GDI+
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, nullptr);

    // Load game assets
    LoadAssets();
    srand(static_cast<unsigned>(time(0)));

    // Set up timers
    SetTimer(hWnd, 1, 16, nullptr);  // Timer for the game loop (roughly 60 FPS)
    SetTimer(hWnd, 2, 800, nullptr);  // Timer to spawn enemies more frequently
(800ms)

    // Game loop
    while (isRunning) {
        if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE)) {
            if (msg.message == WM_QUIT) {
                isRunning = false;
            }
            else {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }
    }

    // Clean up GDI+ resources
    ReleaseAssets();
    GdiplusShutdown(gdiplusToken);
    KillTimer(hWnd, 1);
    KillTimer(hWnd, 2);

    return (int)msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
```

```
    switch (message) {
    case WM_KEYDOWN:
        switch (wParam) {
        case VK_LEFT:
            playerX -= playerSpeed;
            break;
        case VK_RIGHT:
            playerX += playerSpeed;
            break;
        case VK_SPACE:
            ShootBullet();
            break;
        }
        break;

    case WM_TIMER:
        if (wParam == 1) {  // Game update timer
            Update();
            InvalidateRect(hWnd, nullptr, FALSE);
        }
        if (wParam == 2) {  // Enemy spawn timer
            SpawnEnemy();
        }
        break;

    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hWnd, &ps);
        RenderDoubleBuffer(hdc);  // Render using double buffering
        EndPaint(hWnd, &ps);
    }
    break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Here is a glimpse what the game looks:





Conclusion

The development of this 2D shooter game in C++ with Win32 API and GDI+ was a valuable learning experience in low-level game programming and rendering optimization. By troubleshooting issues related to screen flicker, performance, and object spawning, we gained a deep understanding of how to manage a game loop, implement double buffering, and handle real-time game updates. Although simple, this game serves as a solid foundation for understanding more complex game development principles and techniques. Future work could involve adding sound, more intricate enemy behavior, and refining the game's collision handling further.

References

- Microsoft Documentation on Win32 API
- MSDN Resources on GDI+ Graphics
- Real-Time Game Loop Concepts