```c
//
//  main.c
//  Raycaster
//
//  Created by ----- on 1/22/24.
//
// IMPORTANT! There are instances where code using the Simple DirectMedia
 Layer (SDL) package were generated by a language model. Lines of these
 instances are indicated as such.
// Images and music sourced within project not used for commercial purposes or
 redistribution purposes.

#include "raycaster.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_mixer.h>

int main(int argc, const char * argv[])
{
    // INITIALIZATION OF PACKAGES ----------


    // Code initializes SDL package with error handling.
    SDL_Window* Window = SDL_CreateWindow("Raycaster",
     SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED, ScreenWidth,
     ScreenHeight, SDL_WINDOW_SHOWN); // Generated by language model
    SDL_Renderer* Renderer = SDL_CreateRenderer(Window, -1,
     SDL_RENDERER_ACCELERATED); // Generated by language model
    if (!Window || !Renderer)
    {
        printf("ERROR! Some error occured when attempting to open the window
         or renderer!\n");
        return 1;
    }
    // Code initializes SDL mixer to play sounds
    if (SDL_Init(SDL_INIT_AUDIO) < 0)
    {
        printf("ERROR! Some error occured when attempting to open the sound
         player!\n");
        return 1;
    }
    if (Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048) < 0)
    {
        printf("ERROR! Some error occured when attempting to open the sound
         player!\n");
        return 1;
    }
```

```
// SPRITE IMAGES ----------------



// Opens all sprite image files and assigns them to correct sprite
SDL_Texture* SpriteTextures[SPRITE_IMAGES] = {NULL}; // Initializes crap

// Iterates thorugh all images and opens them for being drawn on the screen
for (int i = 0; i < SPRITE_IMAGES; i++)
{
    // Opens the sprite image file
    char Filepath[100];
    sprintf(Filepath, "/Users/REDACTED/Documents/Xcode
     Apps/Raycaster/Sprites/sprite_%d.png", i);
    SDL_Surface* SpriteSurface = IMG_Load(Filepath);
    if (SpriteSurface == NULL)
    {
        printf("ERROR! Couldn't find sprite_%d.png\n", i);
        if (i == 0)
        {
            printf("\nYou may have prevented Xcode from accessing
             files...modify the code then revert changes to reprompt the
             editor\n\n");
        }
        return 1;
    }

    // Creates a list that is assigned to every Sprites.ImageID
    SpriteTextures[i] = SDL_CreateTextureFromSurface(Renderer,
     SpriteSurface); // Generated by language model
    SDL_FreeSurface(SpriteSurface); // Generated by language model
    if (SpriteTextures[i] == NULL)
    {
        printf("ERROR! Some error occured when attempting to create a
         texture for a sprite!\n");
        return 1;
    }
}
/*
 Sprite sources:
 https://minecraft.fandom.com/wiki/Potion
 https://www.behance.net/gallery/35816219/Micro-Monsters
 https://www.nintendo.com/us/store/products/dragon-sinker-switch/
*/

// Loads a specific image: first start menu
```

```
SDL_Texture* FirstTitle = IMG_LoadTexture(Renderer,
 "/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Icons/firsttitlescreen.png");
if (FirstTitle == NULL)
{
    printf("ERROR! Couldn't find firsttitlescreen.png");
    return 1;
}

// Loads a specific image: second start menu
SDL_Texture* SecondTitle = IMG_LoadTexture(Renderer,
 "/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Icons/secondtitlescreen.png");
if (SecondTitle == NULL)
{
    printf("ERROR! Couldn't find secondstitlescreen.png");
    return 1;
}

// Loads a specific image: loss menu
SDL_Texture* LossScreen = IMG_LoadTexture(Renderer,
 "/Users/REDACTED/Documents/Xcode Apps/Raycaster/Icons/gameover.png");
if (LossScreen == NULL)
{
    printf("ERROR! Couldn't find gameover.png");
    return 1;
}

// Loads a specific image: win menu
SDL_Texture* VictoryScreen = IMG_LoadTexture(Renderer,
 "/Users/REDACTED/Documents/Xcode Apps/Raycaster/Icons/victory.png");
if (VictoryScreen == NULL)
{
    printf("ERROR! Couldn't find victory.png");
    return 1;
}

// Loads a specific image: low health indicator
SDL_Texture* BloodOverlay = IMG_LoadTexture(Renderer,
 "/Users/REDACTED/Documents/Xcode Apps/Raycaster/Icons/BloodOverlay.png");
if (BloodOverlay == NULL)
{
    printf("ERROR! Couldn't find BloodOverlay.png");
    return 1;
}

// Loads all gun sprites into play
SDL_Texture* GunTextures[9] = {NULL}; // Initializes crap
for (int i = 0; i < 9; i++)
{
    char filepath[100];
```

```
    sprintf(filepath, "/Users/REDACTED/Documents/Xcode
     Apps/Raycaster/GunIcons/gun_%d.png", i);
    SDL_Surface* GunSurface = IMG_Load(filepath);
    if (GunSurface == NULL)
    {
        printf("ERROR! Couldn't find gun_%d.png\n", i);
        return 1;
    }
    // Creates a list that is assigned to every gun texture
    GunTextures[i] = SDL_CreateTextureFromSurface(Renderer, GunSurface);
     // Generated by language model
    SDL_FreeSurface(GunSurface); // Generated by language model
    if (GunTextures[i] == NULL)
    {
        printf("ERROR! Some error occured when attempting to create a
         texture for a sprite!\n");
        return 1;
    }
}
// Gun source:
 https://www.newgrounds.com/art/view/wronglabel/first-person-rifle-anim-4



// SPRITE MUSIC ---------


Mix_Music *TitleTheme = Mix_LoadMUS("/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Sounds/titletheme.ogg"); // Loads music into use
if (TitleTheme == NULL)
{
    printf("ERROR! Couldn't find titletheme.ogg\n");
    return 1;
}
Mix_Music *MainTheme = Mix_LoadMUS("/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Sounds/maintheme.ogg"); // Loads music into use
if (MainTheme == NULL)
{
    printf("ERROR! Couldn't find maintheme.ogg\n");
    return 1;
}
Mix_Music *VictoryTheme = Mix_LoadMUS("/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Sounds/victorytheme.ogg"); // Loads music into use
if (VictoryTheme == NULL)
{
    printf("ERROR! Couldn't find victorytheme.ogg\n");
    return 1;
}
```

```
Mix_Music *LossTheme = Mix_LoadMUS("/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Sounds/losstheme.ogg"); // Loads music into use
if (LossTheme == NULL)
{
    printf("ERROR! Couldn't find losstheme.ogg\n");
    return 1;
}
Mix_Chunk *FireSound = Mix_LoadWAV("/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Sounds/firesound.ogg");
if (FireSound == NULL)
{
    printf("ERROR! Couldn't find firesound.ogg\n");
    return 1;
}
Mix_Chunk *PotionSound = Mix_LoadWAV("/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Sounds/potionsound.ogg");
if (PotionSound == NULL)
{
    printf("ERROR! Couldn't find potionsound.ogg\n");
    return 1;
}
Mix_Chunk *SpriteDeathSound = Mix_LoadWAV("/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Sounds/spritedeathsound.ogg");
if (SpriteDeathSound == NULL)
{
    printf("ERROR! Couldn't find spritedeathsound.ogg\n");
    return 1;
}
Mix_Chunk *ReloadSound = Mix_LoadWAV("/Users/REDACTED/Documents/Xcode
 Apps/Raycaster/Sounds/reloadsound.ogg");
if (ReloadSound == NULL)
{
    printf("ERROR! Couldn't find reloadsound.ogg\n");
    return 1;
}

/*
 Music sources:
 https://www.youtube.com/watch?v=YB2MbgDsLaA
 https://www.youtube.com/watch?v=qHUOQ5elnFU
 https://www.youtube.com/watch?v=eHzuQyFgJ2E
 https://www.youtube.com/watch?v=xmxof0mQI_Y
 https://www.youtube.com/watch?v=WhFcW6NCMgY
 https://www.youtube.com/watch?v=Zju3DoAoPXo
 https://www.youtube.com/watch?v=1yfQ4Qs9HnI
 https://creatorassets.com/a/8-bit-coin-sound-effects
 */
```

```
// SPRITES INITIALIZATION --------------



Sprite TempSprites[11] = // List of temporary sprites on the stack and is
 starting sprite data
{
    // Level 1 enemies
    {0, 7, 14, 2.5, 0, 85}, // Bounty hunter guy
    {1, 7, 2.5, 2.5, 0, 85}, // Bounty hunter guy

    // Level 2 enemies
    {2, 5, 4.5, 11.5, 0, 150}, // Mechanized robot
    {3, 5, 5.5, 18.5, 0, 150}, // Mechanized robot
    {4, 5, 2.5, 17.5, 0, 150}, // Mechanized robot

    // Level 3 enemies. Victory condition
    {5, 6, 14, 10.5, 0, 375}, // Dragon

    // Health potions
    {6, 4, 13.5, 3.5, 2}, // Health potion
    {7, 4, 19.5, 20, 2}, // Health potion
    {8, 4, 9, 5.5, 2}, // Health potion
    {9, 4, 15.5, 7.5, 2}, // Health potion

    // Fun easter egg: the first sprite I coded onto the game!
    {10, 8, 18.5, 5.5, -1} // Easter egg
};

// Allocate memory for the sprites, initializing with starting sprites
int Size = sizeof(TempSprites) / sizeof(TempSprites[0]);
SpritesCapacity = Size; // Set initial SpritesCapacity to match the size
Sprite* Sprites = (Sprite*)malloc(SpritesCapacity * sizeof(Sprite));
for (int i = 0; i < Size; i++)
{
    Sprites[i] = TempSprites[i];
}

// Make sure SpritesCapacity is good, and if not, allocates more memory
 for it
if (Size >= SpritesCapacity)
{
    SpritesCapacity *= 5; // Increase SpritesCapacity
    Sprite* Temp = (Sprite*)realloc(Sprites, SpritesCapacity *
     sizeof(Sprite));
    Sprites = Temp;
}


// START CODE --------------
```

```
// Memory allocation to handle events in SDL
SDL_Event *event = malloc(sizeof(SDL_Event)); // Generated by language
 model
bool ProgramRunning = true;
bool FirstStartScreen = true;
bool SecondStartScreen = false;
bool EndScreen_Victory = false;
bool EndScreen_Loss = false;
bool GunShot = false;
bool GunReload = false;
int RecoverStaminaTimer = 0;
int GunShotRenderTime = SDL_GetTicks();
int GunReloadRenderTime = SDL_GetTicks();

// Plays the title theme music
Mix_VolumeMusic(MIX_MAX_VOLUME / 3.5); // Sets volume to 35%
if (Mix_PlayMusic(TitleTheme, -1) < 0) // Plays the music!
{
    printf("ERROR! Some error occured when attempting to play
     maintheme.ogg!");
    return -1;
}

// Central loop that constantly runs and draws thing on screen until
 program is closed
while (ProgramRunning)
{


    // OTHER SCREENS LIKE LOSS/WIN --------------------


    // Initiate a start screen to give instructions to the player what to
     do
    if (FirstStartScreen)
    {
        while (SDL_PollEvent(event) != 0) // Generated by language model
        {
            // Handle quit event and stop the program on player demand
            if (event->type == SDL_QUIT) // Generated by language model
            {
                ProgramRunning = false;
            }
            // Generated by language model; if a key is being pressed
            if (event->type == SDL_KEYDOWN)
            {
                // Prints out a welcome message + reminds the user of how
                 to move around
                printf("WELCOME TO\n\n");
                char *ASCII_Art =
```

```
            "  _____                    ____                          _   \n"
            "|  __ \\            / __ \\                | |  \n"
            "| |__) |__ _ _   _| |  | |_   _  ___  ___| |_ \n"
            "|  _  // _` | | | | |  | | | | |/ _ \\/ __| __|\n"
            "| | \\ \\ (_| | |_| | |__| | |_| |  __/\\__ \\ |_ \n"
            "|_|  \\_\\__,_|\\__, |\\___\\_\\\\__,_|\\___||___/\\__|\n"
            "              __/ |                            \n"
            "             |___/                             \n";

            // Print the ASCII Art to the terminal
            printf("%s\n", ASCII_Art);
            printf("By REDACTED\n\n");
            printf("[W] to MOVE FORWARD\n");
            printf("[A] to MOVE LEFT\n");
            printf("[S] to MOVE BACK\n");
            printf("[D] to MOVE RIGHT\n\n");

            printf("[SHIFT] to RUN\n\n");

            printf("[<-] to TURN LEFT\n");
            printf("[->] to TURN RIGHT\n\n");

            printf("[SPACE] to FIRE\n\n");

            printf("OBJECTIVE:\n\n");

            char *ASCII_Art2 =
            "  ____  _____ _____ _____    _  _____   _____ _   _ _____
             \n"
            "|  _ \\| ____|  ___| ____|  / \\|_   _| |_   _| | | |
             ____|\n"
            "| | | |  _| | |_  |  _|   / _ \\ | |     | | | |_| |  _|
             \n"
            "| |_| | |___|  _| | |___ / ___ \\| |     | | |  _  | |___
             \n"
            "|____/|_____|_|   |_____/_/   \\_\\_|     |_| |_|
             |_|_____|\n"
            "|  _ \\|  _ \\    / \\  / ___|/ _ \\| \\ | |
                 \n"
            "| | | | |_) |  / _ \\| |  _| | | |  \\| |
              \n"
            "| |_| |  _ <  / ___ \\ |_| | |_| | |\\  |
              \n"
            "|____/|_| \\_\\/_/   \\_\\____|\\___/|_| \\_|
                 \n";

            printf("%s", ASCII_Art2);
            SecondStartScreen = true;
            FirstStartScreen = false;
        }
    }
```

```
    // If player hasn't pressed a key, just draw title screen
    SDL_Rect TitleRect = {0, 0, 1280, 800};
    SDL_Texture* Title = FirstTitle;
    SDL_RenderCopy(Renderer, Title, NULL, &TitleRect);
    SDL_RenderPresent(Renderer); // Generated by language model
}
else if (SecondStartScreen)
{
    while (SDL_PollEvent(event) != 0) // Generated by language model
    {
        // Handle quit event and stop the program on player demand
        if (event->type == SDL_QUIT) // Generated by language model
        {
            ProgramRunning = false;
        }
        if (event->type == SDL_KEYDOWN) // If a key is pressed, start
         the game!
        {
            SecondStartScreen = false;
            // Plays the main theme music
            Mix_VolumeMusic(MIX_MAX_VOLUME / 3.5); // Sets volume to
             35%
            if (Mix_PlayMusic(MainTheme, -1) < 0)
            {
                printf("ERROR! Some error occured when attempting to
                 play maintheme.ogg!");
                return -1;
            }
        }
    }
    // If player hasn't pressed a key, just draw title screen
    SDL_Rect TitleRect = {0, 0, 1280, 800};
    SDL_Texture* Title = SecondTitle;
    SDL_RenderCopy(Renderer, Title, NULL, &TitleRect);
    SDL_RenderPresent(Renderer);

}
else if (EndScreen_Loss)
{
    while (SDL_PollEvent(event) != 0) // Generated by language model
    {
        // Handle quit event and stop the program on player demand
        if (event->type == SDL_QUIT) // Generated by language model
        {
            ProgramRunning = false;
        }
    }

    // Renders loss screen
    SDL_Rect LossRect = {0, 0, 1280, 800};
    SDL_RenderCopy(Renderer, LossScreen, NULL, &LossRect);
```

```
    SDL_RenderPresent(Renderer); // Generated by language model

}
else if (EndScreen_Victory)
{
    while (SDL_PollEvent(event) != 0) // Generated by language model
    {
        // Handle quit event and stop the program on player demand
        if (event->type == SDL_QUIT) // Generated by language model
        {
            ProgramRunning = false;
        }
    }

    // Renders victory screen
    SDL_Rect VictoryRect = {0, 0, 1280, 800};
    SDL_RenderCopy(Renderer, VictoryScreen, NULL, &VictoryRect);
    SDL_RenderPresent(Renderer); // Generated by language model
}


// ACTUAL CODE --------------------------


else
{
    // If some event is triggered to the SDL: so if the player presses
     a button
    while (SDL_PollEvent(event) != 0) // Generated by language model
    {
        // Handle quit event and stop the program on player demand
        if (event->type == SDL_QUIT) // Generated by language model
        {
            ProgramRunning = false;
        }
        // Generated by language model; if a key is being pressed
        if (event->type == SDL_KEYDOWN)
        {
            // Uses a list of stored scan codes that tracks key being
             pressed down, instead of number of key presses; allows
             movement/rotation to be simultaneous and smooth
            keyState[event->key.keysym.scancode] = true;
        }
        // Generated by language model; if a key is released
        if (event->type == SDL_KEYUP)
        {
            keyState[event->key.keysym.scancode] = false;
        }

        // Handles when we want key to be registered once, not
         continously (so not held down)
```

```
    if (event->type == SDL_KEYDOWN)
    {
        // If the gun is fired
        if (event->key.keysym.scancode == SDL_SCANCODE_SPACE)
        {
            if (PlayerAmmo > 0 && GunReload == false)
            {
                GunShot = true;
                GunShotRenderTime = SDL_GetTicks();
                DirectionAbsoluteRadian = atan2(DirectionY,
                 DirectionX);
                InitializeCreateSprite(Sprites, 3, PlayerX + 1 *
                 cosf(DirectionAbsoluteRadian), PlayerY + 1 *
                 sinf(DirectionAbsoluteRadian), 1,
                 DirectionAbsoluteRadian);
                PlayerAmmo -= 1;
                Mix_VolumeChunk(FireSound, MIX_MAX_VOLUME); //
                 100% volume
                Mix_PlayChannel(-1, FireSound, 0);
            }
        }
        // If the gun is reloaded
        if (event->key.keysym.scancode == SDL_SCANCODE_R &&
         GunReload == false && PlayerAmmo != 10)
        {
            GunReload = true;
            GunReloadRenderTime = SDL_GetTicks();
            Mix_VolumeChunk(ReloadSound, MIX_MAX_VOLUME); // 100%
             volume
            Mix_PlayChannel(-1, ReloadSound, 0);
        }
        // Debugging purposes
        if (event->key.keysym.scancode == SDL_SCANCODE_F1)
        {
            PlayerHealth = 999;
        }
        // No-violence, pure raycaster showcaser
        if (event->key.keysym.scancode == SDL_SCANCODE_F2)
        {
            NoViolence = true; // In case I want to showcase a
             pure raycaster...without the blood and iron. Enabled
             through F2
        }
    }
}

// Adjusts live information and checks things before attempting to
 render frame ---------------

PlayerInput(&RecoverStaminaTimer); // Checks for player movement
AdjustSpriteInfo(Sprites); // Sorts all the sprites.
```

```
HandleSpriteBehavior(Sprites, &NoViolence, FireSound, PotionSound,
 SpriteDeathSound); // Moves sprites

if (PlayerHealth <= 0) // If the player is dead...
{
    EndScreen_Loss = true;
    Mix_VolumeMusic(MIX_MAX_VOLUME / 3.5); // Sets volume to 35%
    if (Mix_PlayMusic(LossTheme, -1) < 0) // Plays the music!
    {
        printf("ERROR! Some error occured when attempting to play
         maintheme.ogg!");
    }
}

bool DragonDead = true; // Check if the player has won yet
for (int i = 0; i < NumberOfSprites; i++)
{
    if (Sprites[i].ImageID == 6)
    {
        DragonDead = false;
        break;
    }
}
if (DragonDead)
{
    EndScreen_Victory = true;
    // Plays victory theme
    Mix_VolumeMusic(MIX_MAX_VOLUME / 3.5); // Sets volume to 35%
    if (Mix_PlayMusic(VictoryTheme, -1) < 0) // Plays the music!
    {
        printf("ERROR! Some error occured when attempting to play
         maintheme.ogg!");
    }
}

// Begins drawing the scene ----------------------------

// Creates roof/ceiling
SDL_SetRenderDrawColor(Renderer, 180, 175, 250, 255);
SDL_RenderFillRect(Renderer, &(SDL_Rect){0, 0, ScreenWidth,
 ScreenHeight / 2 + VerticalView}); // Generated by language model
// Creates floor
SDL_SetRenderDrawColor(Renderer, 90, 150, 90, 255);
SDL_RenderFillRect(Renderer, &(SDL_Rect){0, ScreenHeight / 2 +
 VerticalView, ScreenWidth, ScreenHeight / 2 - VerticalView}); //
 Generated by language model

// Variables to figure out how many rays to calculate
int Pixel = ScreenWidth;
float RayDegreeInterval = (float) FieldOfView / ScreenWidth; //
 Every how many degrees should a ray be cast
```

```
// Iterates through every pixel (vertical stripe) on the screen
for (double RayDegree = -(FieldOfView / 2); RayDegree <=
 (FieldOfView / 2); RayDegree += RayDegreeInterval) // If FOV = 60
 degrees, iterates from -30 to 30 degrees. NOT absolute angles!
{
    double RayRadian = RayDegree * M_PI / 180;
    DirectionAbsoluteRadian = atan2(DirectionY, DirectionX);

    // Calculates the distance from wall to player and stores it
    float Value = CalculateRay(RayRadian);

    // Calculates how long the stripe should be from two points,
     with the line's center being the center of the screen
    // True value of size of the line stripe can be extremely
     large if Value is extremely small; i.e. if Value is 0.01 True
     value could be 5000 while the window's physical dimensions
     would be 800. Only for optimization purposes
    int TrueTopPixel = ScreenHeight / 2 - ScreenHeight / 2 / Value;
    int RenderTopPixel = TrueTopPixel;
    if (RenderTopPixel < 0)
    {
        RenderTopPixel = 0;
    }
    int TrueBottomPixel = ScreenHeight / 2 + ScreenHeight / 2 /
     Value;
    int RenderBottomPixel = TrueBottomPixel;
    if (RenderBottomPixel > 800)
    {
        RenderBottomPixel = 800;
    }

    // Blurs the walls into the ceiling based on distance for
     depth perception
    float DistanceFactor;
    if (Value > 20)
    {
        DistanceFactor = 1;
    }
    else
    {
        DistanceFactor = Value / 20;
    }
    // Calculating the color components with it fading to the sky
     color
    int WallRed = 160 + (int)((180 - 160) * DistanceFactor);
    int WallGreen = 160 + (int)((175 - 160) * DistanceFactor);
    int WallBlue = 160 + (int)((250 - 160) * DistanceFactor);

    // Drawing the wall with the color, and with the line
```

```
if (!NoViolence) // In case I want to showcase a pure
 raycaster...without the blood and iron. Enabled through F2
{
    SDL_SetRenderDrawColor(Renderer, WallRed, WallGreen,
     WallBlue, 255);
    SDL_RenderDrawLine(Renderer, Pixel, RenderBottomPixel,
     Pixel, RenderTopPixel);
}
else
{
    SDL_SetRenderDrawColor(Renderer, WallRed, WallGreen,
     WallBlue, 255);
    SDL_RenderDrawLine(Renderer, Pixel, VerticalView +
     RenderBottomPixel, Pixel, VerticalView + RenderTopPixel);
}

// Iterates through all the sprites and checks if, for this
 specific pixel (in the loop we're in), if a sprite should be
 drawn, and logs that pixel to draw the sprite later after the
 walls
for (int i = 0; i < NumberOfSprites; i++)
{
    // Calculates absolute radian of the ray
    float RayAbsoluteRadian = DirectionAbsoluteRadian +
     RayRadian;
    if (RayAbsoluteRadian > M_PI)
    {
        RayAbsoluteRadian -= 2 * M_PI;
    }
    else if (RayAbsoluteRadian < -1 * M_PI)
    {
        RayAbsoluteRadian += 2 * M_PI;
    }

    // If the ray is at the same angle with the sprite, log it
     to render later
    if (fabsf(Sprites[i].AngleToPlayer - RayAbsoluteRadian) <
     0.005) // Tolerance level.
    {
        Sprites[i].Pixel = Pixel;
        Sprites[i].PlayerToWallDistance = Value; // Logs, for
         that sprite at that pixel, the same distance to a
         wall at that pixel, to see whether the sprite is in
         front of (closer to the player) or behind the wall
         (farther from the player)
    }
}

// Moves to the next pixel.
if (Pixel < 0)
{
```

```
            Pixel = ScreenWidth;
        }
        else
        {
            Pixel--;
        }
    }



    // DRAW SPRITES ----------------


    // Iterates through all the sprites and draws them if able to
    for (int i = 0; i < NumberOfSprites; i++)
    {
        // If sprite should be on screen as logged earlier in loop
        if (Sprites[i].Pixel > -1)
        {
            // If the sprite is actually in front of the wall instead
             of behind it
            if (Sprites[i].DistanceToPlayer <
             Sprites[i].PlayerToWallDistance)
            {
                // Calculates dimensions of sprite to drawn on screen
                float HalfW = ((ScreenHeight / 2) /
                 Sprites[i].DistanceToPlayer);

                float StartDrawX = Sprites[i].Pixel - HalfW;
                float StartDrawY = (ScreenHeight / 2) - (ScreenHeight
                 / 2) / Sprites[i].DistanceToPlayer;
                float HeightOfSprite = 2 * (ScreenHeight / 2) /
                 Sprites[i].DistanceToPlayer;

                if (NoViolence) // In case I want to showcase a pure
                 raycaster...without the blood and iron. Enabled
                 through F2
                {
                    StartDrawY += VerticalView;
                }

                SDL_Rect SpriteRect = {StartDrawX, StartDrawY, 2 *
                 ((ScreenHeight / 2) / Sprites[i].DistanceToPlayer),
                 (2 * (ScreenHeight / 2) /
                 Sprites[i].DistanceToPlayer)};
                SDL_Texture* Texture =
                 SpriteTextures[Sprites[i].ImageID]; // Takes the
                 sprite's image it should use from the list of textures
                SDL_RenderCopy(Renderer, Texture, NULL, &SpriteRect);
```

```
                if (Sprites[i].SpriteType == 0) // If sprite is enemy
                {
                    // Draws sprite's health bar
                    if (Sprites[i].ImageID == 7) // Bounty hunter
                     sprite (draws the health to scale to value 85)
                    {
                        SDL_SetRenderDrawColor(Renderer, 255, 255,
                         255, 100);
                        SDL_RenderFillRect(Renderer,
                         &(SDL_Rect){StartDrawX, StartDrawY,
                         HeightOfSprite, 120 /
                         Sprites[i].DistanceToPlayer});
                        SDL_SetRenderDrawColor(Renderer, 200, 30, 30,
                         255);
                        SDL_RenderFillRect(Renderer,
                         &(SDL_Rect){StartDrawX + 2, StartDrawY + 2,
                         ((float) Sprites[i].SpriteHealth / 85) *
                         (HeightOfSprite - 4), 120 /
                         Sprites[i].DistanceToPlayer - 4});
                    }
                    if (Sprites[i].ImageID == 5) // Mechanized robot
                     sprite (draws the health to scale to value 150)
                    {
                        SDL_SetRenderDrawColor(Renderer, 255, 255,
                         255, 100);
                        SDL_RenderFillRect(Renderer,
                         &(SDL_Rect){StartDrawX, StartDrawY,
                         HeightOfSprite, 120 /
                         Sprites[i].DistanceToPlayer});
                        SDL_SetRenderDrawColor(Renderer, 200, 30, 30,
                         255);
                        SDL_RenderFillRect(Renderer,
                         &(SDL_Rect){StartDrawX + 2, StartDrawY + 2,
                         ((float) Sprites[i].SpriteHealth / 150) *
                         (HeightOfSprite - 4), 120 /
                         Sprites[i].DistanceToPlayer - 4});
                    }
                    if (Sprites[i].ImageID == 6) // Dragon sprite
                     (draws the health to scale to value 375)
                    {
                        SDL_SetRenderDrawColor(Renderer, 255, 255,
                         255, 100);
                        SDL_RenderFillRect(Renderer,
                         &(SDL_Rect){StartDrawX, StartDrawY,
                         HeightOfSprite, 120 /
                         Sprites[i].DistanceToPlayer});
                        SDL_SetRenderDrawColor(Renderer, 200, 30, 30,
                         255);
```

```
                            SDL_RenderFillRect(Renderer,
                             &(SDL_Rect){StartDrawX + 2, StartDrawY + 2,
                             ((float) Sprites[i].SpriteHealth / 375) *
                             (HeightOfSprite - 4), 120 /
                             Sprites[i].DistanceToPlayer - 4});
                        }

                        if (Sprites[i].SeenByPlayer != true)
                        {
                            Sprites[i].SeenByPlayer = true; // The sprite
                             won't interact with the player if the player
                             hasn't rendered it on screen/seen it yet
                        }
                    }

                    Sprites[i].Pixel = -1; // Sprite has been rendered for
                     this pixel
                }
            }
        }

        // Draws other (less cool) overlays UIs ------------------------

        if (!NoViolence) // In case I want to showcase a pure
         raycaster...without the blood and iron. Enabled through F2
        {
            DrawHUD(Renderer); // Draws the health bar/etc.
            DrawWeapon(Renderer, GunTextures, &GunShot, GunShotRenderTime,
             &GunReload, GunReloadRenderTime); // Draws the gun on screen
        }

        // Draws the low health indicator
        if (!NoViolence) // In case I want to showcase a pure
         raycaster...without the blood and iron. Enabled through F2
        {
            if (PlayerHealth < 18)
            {
                SDL_Rect BloodRect = {0, 0, 1280, 800};
                SDL_Texture* Blood = BloodOverlay;
                SDL_RenderCopy(Renderer, Blood, NULL, &BloodRect);
            }
        }

        // Draws the minimap
        DrawMap(Sprites, Renderer);

        // Presents each frame to screen
        SDL_RenderPresent(Renderer); // Generated by language model
    }
}
// Technical shenanigans w/ memory once called for the code to end.
```

```
    SDL_DestroyWindow(Window); // Generated by language model
    SDL_Quit(); // Generated by language model
    Mix_FreeMusic(MainTheme);
    Mix_FreeChunk(FireSound);
    Mix_CloseAudio();
    free(event);
    free(Sprites);
    return 0;
}
```

```
// Functions that are either a) for readability or b) looped. Main function
 code above! ------------------------
```

```
float CalculateRay(float RayRadian)
{
    // Calculates the lowest distance between vertical and horizontal
     traversals
    float LowestDistance = 0;

    // Gets ray,and it's direction  in standard position
    DirectionAbsoluteRadian = atan2(DirectionY, DirectionX);
    float RayX = cosf(DirectionAbsoluteRadian + RayRadian);
    float RayY = sinf(DirectionAbsoluteRadian + RayRadian);

    // Temporary variables that check for iterates up and checks for ray
     collision with wall
    float xTraverse;
    float yTraverse;

    // Meh
    float DistanceFromStart;

    int const MAXCHECK = 20;
```

```
// Using a grid-iteration pattern and trigonometry, attempts to find a
 wall both vertically iterating by a constant integer 1 and horizontally
 iterating by constant integer 1 and the lower distance between the
 vertical and horizontal attempts.

// Checks for horizontal intersection
if (RayX > 0)
{
    // Jumps to the nearest integer grid
    float DistToStart = (1 - (PlayerX - (int) PlayerX));
    xTraverse = PlayerX + DistToStart;
    yTraverse = PlayerY + RayY * (DistToStart / RayX);
    // Iterates up by 1 horizontally and by some value vertically to find
     a wall. I.E. From (0, 3), it may iterate to (1, 3.7), (2, 4.4), etc.
    for (int i = 0; i < MAXCHECK; i++)
    {
        if (xTraverse > (MapWidth - 1) || yTraverse > (MapHeight - 1) ||
         xTraverse < 0 || yTraverse < 0)
        {
            break; // If out of bounds of Map
        }
        if (Map[(int) xTraverse][(int) yTraverse] == 1) // If finds a
         wall...
        {
            DistanceFromStart = sqrtf((xTraverse - PlayerX) * (xTraverse -
             PlayerX) + (yTraverse - PlayerY) * (yTraverse - PlayerY));
            if (DistanceFromStart < LowestDistance || LowestDistance <
             0.001)
            {
                LowestDistance = DistanceFromStart; // Stores the lowest
                 distance
            }
            break;
        }

        // If doesn't find a wall, iterate to next grid
        xTraverse += 1;
        yTraverse += (RayY / RayX);
    }

}
else if (RayX < 0)
{
    float DistToStart = PlayerX - (int) PlayerX;
    xTraverse = PlayerX - DistToStart;
    yTraverse = PlayerY - RayY * (DistToStart / RayX);
    for (int i = 0; i < MAXCHECK; i++)
    {
        if (xTraverse > (MapWidth - 1) || yTraverse > (MapHeight - 1) ||
         xTraverse < 0 || yTraverse < 0)
        {
```

```
            break;
        }
        if (Map[(int) (xTraverse - 1)][(int) yTraverse] == 1)
        {
            DistanceFromStart = sqrtf((xTraverse - PlayerX) * (xTraverse -
             PlayerX) + (yTraverse - PlayerY) * (yTraverse - PlayerY));
            if (DistanceFromStart < LowestDistance || LowestDistance <
             0.001)
            {
                LowestDistance = DistanceFromStart;
            }
            break;
        }

        xTraverse -= 1;
        yTraverse -= (RayY / RayX);
    }
}

// Checks for vertical intersection
if (RayY > 0)
{
    // Jumps to the nearest integer grid
    float DistToStart = (1 - (PlayerY - (int) PlayerY));
    yTraverse = PlayerY + DistToStart;
    xTraverse = PlayerX + RayX * (DistToStart / RayY);
    // Iterates up by 1 vertically and by some value horizontally to find
     a wall. I.E. From (0, 3), it may iterate to (0.7, 4), (1.4, 5), etc.
    for (int i = 0; i < MAXCHECK; i++)
    {
        if (xTraverse > (MapWidth - 1) || yTraverse > (MapHeight - 1) ||
         xTraverse < 0 || yTraverse < 0)
        {
            break;
        }
        if (Map[(int) xTraverse][(int) yTraverse] == 1)
        {
            DistanceFromStart = sqrtf((xTraverse - PlayerX) * (xTraverse -
             PlayerX) + (yTraverse - PlayerY) * (yTraverse - PlayerY));
            if (DistanceFromStart < LowestDistance || LowestDistance <
             0.001)
            {
                LowestDistance = DistanceFromStart;
            }
            break;
        }

        yTraverse += 1;
        xTraverse += (RayX / RayY);
    }
}
```

```
    else if (RayY < 0)
    {
        float DistToStart = PlayerY - (int) PlayerY;
        yTraverse = PlayerY - DistToStart;
        xTraverse = PlayerX - (RayX * (DistToStart / RayY));

        for (int i = 0; i < MAXCHECK; i++)
        {
            if (xTraverse > (MapWidth - 1) || yTraverse > (MapHeight - 1) ||
             xTraverse < 0 || yTraverse < 0)
            {
                break;
            }
            if (Map[(int) xTraverse][(int) (yTraverse - 1)] == 1)
            {
                DistanceFromStart = sqrtf((xTraverse - PlayerX) * (xTraverse -
                 PlayerX) + (yTraverse - PlayerY) * (yTraverse - PlayerY));
                if (DistanceFromStart < LowestDistance || LowestDistance <
                 0.001)
                {
                    LowestDistance = DistanceFromStart;
                }
                break;
            }

            yTraverse -= 1;
            xTraverse -= (RayX / RayY);
        }
    }

    if (LowestDistance < 0.000001)
    {
        return 0.0000001; // Prevent dividing by 0
    }
    else
    {
        return LowestDistance;
    }
}

void PlayerInput(int *RecoverStaminaTimer)
{
    bool Forward;
    bool Left;
    bool Right;
    bool Backward;
    bool Running;

    // Stores the status of if the key is being held down or not and attempts
     to trigger movement in another function
```

```
if (keyState[SDL_SCANCODE_W])
{
    Forward = true;
    // printf("%f, %f\n", PlayerX, PlayerY);
}
else
{
    Forward = false;
}

if (keyState[SDL_SCANCODE_A])
{
    Left = true;
    // printf("%f, %f\n", PlayerX, PlayerY);
}
else
{
    Left = false;
}

if (keyState[SDL_SCANCODE_S])
{
    Backward = true;
    // printf("%f, %f\n", PlayerX, PlayerY);
}
else
{
    Backward = false;
}

if (keyState[SDL_SCANCODE_D])
{
    Right = true;
    // printf("%f, %f\n", PlayerX, PlayerY);
}
else
{
    Right = false;
}
if (keyState[SDL_SCANCODE_LSHIFT])
{
    if (PlayerStamina > 0) // if able to sprint
    {
        Running = true;
        *RecoverStaminaTimer = 0;
        PlayerStamina -= 0.2;
    }
    else
    {
        PlayerStamina = 0; // in case somehow stamina reached a negative
         value
```

```
        Running = false;
        // Resets the stamina recover timer to stop progress of replenish
        if (*RecoverStaminaTimer == 0)
        {
            *RecoverStaminaTimer = SDL_GetTicks(); // Gets current time
        }
    }
}
else
{
    Running = false;
    if (*RecoverStaminaTimer == 0)
    {
        *RecoverStaminaTimer = SDL_GetTicks(); // Gets current time
    }
    // Regenerates stamina over time
    if ((SDL_GetTicks() - *RecoverStaminaTimer) > 1500 && PlayerStamina <
     100)
    {
        PlayerStamina += 0.1;
        if ((SDL_GetTicks() - *RecoverStaminaTimer) > 3000 &&
         PlayerStamina < 100) // regenerates faster if over 3 seconds
        {
            PlayerStamina += 0.2;
        }
    }

}

// Takes all player movement statuses and attempts to use them
PlayerMovement(Forward, Left, Right, Backward, Running);

// Handle player rotations and moves direction vectors
if (keyState[SDL_SCANCODE_LEFT])
{
    DirectionX = DirectionX * cosf(0.5 * M_PI / 180) - DirectionY *
     sinf(0.5 * M_PI / 180);
    DirectionY = DirectionX * sinf(0.5 * M_PI / 180) + DirectionY *
     cosf(0.5 * M_PI / 180);
}
if (keyState[SDL_SCANCODE_RIGHT])
{
    DirectionX = DirectionX * cosf(0.5 * M_PI / 180) + DirectionY *
     sinf(0.5 * M_PI / 180);
    DirectionY = -DirectionX * sinf(0.5 * M_PI / 180) + DirectionY *
     cosf(0.5 * M_PI / 180);
}

if (NoViolence) // In case I want to showcase a pure raycaster...without
 the blood and iron. Enabled through F2
{
```

```
        if (keyState[SDL_SCANCODE_UP])
        {
            if (VerticalView < 400)
            {
                VerticalView += 4;
            }
            else
            {
                VerticalView = 400;
            }
        }
        if (keyState[SDL_SCANCODE_DOWN])
        {
            if (VerticalView > -400)
            {
                VerticalView -= 4;
            }
            else
            {
                VerticalView = -400;
            }
        }
    }
}

void PlayerMovement(bool Forward, bool Left, bool Right, bool Backward, bool
 Running)
{
    float MovementSpeed = 0.008;
    float RunningSpeed = 1.75;

    float NewX = PlayerX;
    float NewY = PlayerY;

    // Sets temporary variables through changing direction, because needs to
     check collision first
    if (Forward)
    {
        NewX += MovementSpeed * cosf(DirectionAbsoluteRadian);
        NewY += MovementSpeed * sinf(DirectionAbsoluteRadian);
    }
    if (Left)
    {
        NewX += MovementSpeed * cosf(DirectionAbsoluteRadian + M_PI / 2);
        NewY += MovementSpeed * sinf(DirectionAbsoluteRadian + M_PI / 2);
    }
    if (Right)
    {
        NewX += MovementSpeed * cosf(DirectionAbsoluteRadian + 3 * M_PI / 2);
        NewY += MovementSpeed * sinf(DirectionAbsoluteRadian + 3 * M_PI / 2);
    }
```

```
    if (Backward)
    {
        NewX += MovementSpeed * cosf(DirectionAbsoluteRadian + M_PI);
        NewY += MovementSpeed * sinf(DirectionAbsoluteRadian + M_PI);
    }
    if (Running)
    {
        NewX = PlayerX + (RunningSpeed * (NewX - PlayerX));
        NewY = PlayerY + (RunningSpeed * (NewY - PlayerY));
    }

    // Handles collision
    if (Map[(int) NewX][(int) NewY] != 0)
    {
        // Checking if we can't move vertically
        if (Map[(int) PlayerX][(int) NewY] != 0)
        {
            NewY = PlayerY;
        }
        // Checking if we can't move horizontally
        if (Map[(int) NewX][(int) PlayerY] != 0)
        {
            NewX = PlayerX;
        }
    }

    PlayerSpeed = sqrtf((NewY - PlayerY) * (NewY - PlayerY) + (NewX - PlayerX)
     * (NewX - PlayerX));

    // If player is moving, shake that gun in the HandleBob function!
    if (PlayerSpeed > 0.0001)
    {
        HandleBob();
    }

    // Update player position
    PlayerX = NewX;
    PlayerY = NewY;
}

void HandleBob(void)
{
    // Modifies variable VerticalView that shakes the gun when moving.

    if (BobCounter > 100 && DirectionBobUp == false)
    {
        DirectionBobUp = true;
        BobCounter = 0;
    }
    else if (BobCounter > 100 && DirectionBobUp == true) // Flip direction
    {
```

```
        DirectionBobUp = false;
        BobCounter = 0;
    }

    if (DirectionBobUp)
    {
        VerticalView += 0.15;
        BobCounter++;
    }
    else // Flip direction
    {
        VerticalView -= 0.15;
        BobCounter++;
    }
}


void AdjustSpriteInfo(Sprite *Sprites)
{
    for (int i = 0; i < NumberOfSprites; i++)
    {
        // Adjusts the distance to player to be used for
        Sprites[i].DistanceToPlayer = sqrtf((Sprites[i].LocationX - PlayerX) *
         (Sprites[i].LocationX - PlayerX) + (Sprites[i].LocationY - PlayerY) *
         (Sprites[i].LocationY - PlayerY));

        // Adjusts angle to player to be used for 1. casting projectiles and
         2. checking if we can even draw the sprite!
        Sprites[i].AngleToPlayer = atan2(Sprites[i].LocationY - PlayerY,
         Sprites[i].LocationX - PlayerX);

        // Initializes the sprite's timer if not already initialized. Yep.
         Sigh.
        if (Sprites[i].FireTimer < 0 || Sprites[i].FireTimer > 99999999)
        {
            Sprites[i].FireTimer = SDL_GetTicks();
        }
    }

    // Sorts the sprites based on their distance to the player
    qsort(Sprites, NumberOfSprites, sizeof(Sprite), Comparator);
}

int Comparator(const void *a, const void *b)
{
    // Simple comparator for qsort that sorts sprites in list based on
     distance to player
    Sprite *SpriteA = (Sprite *)a;
    Sprite *SpriteB = (Sprite *)b;
    if (SpriteA->DistanceToPlayer < SpriteB->DistanceToPlayer)
    {
```

```
        return 1;
    }
    else if (SpriteA->DistanceToPlayer > SpriteB->DistanceToPlayer)
    {
        return -1;
    }
    else
    {
        return 0;
    }
}

void DrawMap(Sprite* Sprites, SDL_Renderer* Renderer)
{
    // Draws the white border outline of the map
    SDL_SetRenderDrawColor(Renderer, 255, 255, 255, 255);
    SDL_RenderFillRect(Renderer, &(SDL_Rect){43, 43, 150 + 4, 150 + 4});
    // Map actual dimensions: 45, 45 -> 195, 195 (150x150)

    // Iterates through 81 squares around the player and checks whether
     there's a sprite or a wall in them
    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            if ((int) PlayerX + j - 5 < 0 || (int) PlayerX + j - 5 >
             (MapHeight - 1) || (int) PlayerY + i - 5 < 0 || (int) PlayerY + i
             - 5 > (MapWidth - 1)) // If the attempted square to check is out
             of bounds
            {
                SDL_SetRenderDrawColor(Renderer, 115, 115, 115, 255);
                SDL_RenderFillRect(Renderer, &(SDL_Rect){45 + i * 15, 45 + j *
                 15, 30, 30}); // draw out of bounds as "wall"
            }
            else if (Map[(int) PlayerX + j - 5][(int) PlayerY + i - 5] == 1)
             // If the attempted square to check is a wall
            {
                SDL_SetRenderDrawColor(Renderer, 115, 115, 115, 255);
                SDL_RenderFillRect(Renderer, &(SDL_Rect){45 + i * 15, 45 + j *
                 15, 30, 30});
            }
            else // if theres nothing there
            {
                SDL_SetRenderDrawColor(Renderer, 90, 150, 90, 255);
                SDL_RenderFillRect(Renderer, &(SDL_Rect){45 + i * 15, 45 + j *
                 15, 30, 30}); // Draw it as an empty space
            }

            // Iterates through all sprites and checks if they're in an
             attempting to be checked square
            for (int k = 0; k < NumberOfSprites; k++)
```

```
            {
                if ((int) PlayerX + j - 5 == (int) Sprites[k].LocationX &&
                 (int) PlayerY + i - 5 == (int) Sprites[k].LocationY)
                {
                    int DrawOnMapX = (Sprites[k].LocationX - (int)
                     Sprites[k].LocationX ) * 10; // Deviation of sprite from
                     integer value, and draws accordingly
                    int DrawOnMapY = (Sprites[k].LocationY - (int)
                     Sprites[k].LocationY ) * 10;

                    if (Sprites[k].SpriteType == 0 && Sprites[k].SeenByPlayer
                     == true) // If is an enemy
                    {
                        SDL_SetRenderDrawColor(Renderer, 130, 30, 15, 255);
                        SDL_RenderFillRect(Renderer, &(SDL_Rect){45 + i * 15 +
                         DrawOnMapY, 45 + j * 15 + DrawOnMapX, 5, 5});
                    }
                    if (Sprites[k].SpriteType == 2) // If is a health potion
                    {
                        SDL_SetRenderDrawColor(Renderer, 30, 185, 70, 255);
                        SDL_RenderFillRect(Renderer, &(SDL_Rect){45 + i * 15 +
                         DrawOnMapY, 45 + j * 15 + DrawOnMapX, 5, 5});
                    }
                }
            }
        }

        // Draws the player on the screen
        int PlayerDrawOnMapX = ((PlayerX - (int) PlayerX)) * 10; // Decimal
         difference of the player from integer values, then draw on the map (so
         always centered on the map but illusion of moving around)
        int PlayerDrawOnMapY = ((PlayerY - (int) PlayerY)) * 10;
        SDL_SetRenderDrawColor(Renderer, 255, 255, 255, 255);
        SDL_RenderFillRect(Renderer, &(SDL_Rect){120 + PlayerDrawOnMapY, 120 +
         PlayerDrawOnMapX, 7, 7}); // Draws in center of map with deviation from
         integer value in mind

}

void DrawHUD(SDL_Renderer* Renderer)
{
    // Ammo bar
    for (int i = 0; i < PlayerAmmo; i++)
    {
        SDL_SetRenderDrawColor(Renderer, 255, 234, 191, 255);
        SDL_RenderFillRect(Renderer, &(SDL_Rect){i * 41 + 45, ScreenHeight -
         147, 35, 22});
        SDL_SetRenderDrawColor(Renderer, 222, 167, 58, 255);
        SDL_RenderFillRect(Renderer, &(SDL_Rect){i * 41 + 47, ScreenHeight -
         145, 31, 18});
```

```
    }

    // Health bar
    SDL_SetRenderDrawColor(Renderer, 255, 255, 255, 100);
    SDL_RenderFillRect(Renderer, &(SDL_Rect){0 + 45, ScreenHeight - 95, 510,
     60});
    SDL_SetRenderDrawColor(Renderer, 200, 30, 30, 255);
    SDL_RenderFillRect(Renderer, &(SDL_Rect){0 + 47, ScreenHeight - 93,
     PlayerHealth * 5 + 6, 56});

    // Stamina bar
    SDL_SetRenderDrawColor(Renderer, 255, 255, 255, 255);
    SDL_RenderFillRect(Renderer, &(SDL_Rect){0 + 45, ScreenHeight - 116, 455,
     14});
    SDL_SetRenderDrawColor(Renderer, 60, 130, 255, 255);
    SDL_RenderFillRect(Renderer, &(SDL_Rect){0 + 47, ScreenHeight - 114,
     PlayerStamina * 4.5 + 1, 10});

}

void DrawWeapon(SDL_Renderer* Renderer, SDL_Texture* GunTextures[], bool
 *GunShot, int GunShotRenderTime, bool *GunReload, int GunReloadRenderTime)
{
    if (*GunShot == true) // If gun has been fired, render based on how much
     time has passed since initial button press
    {
        if (SDL_GetTicks() - GunShotRenderTime > 100) // if the time is over 2
         seconds, stop rendering the gun as "shot"
        {
            *GunShot = false;
            GunShotRenderTime = 0;
        }
        else // Render the gun in a "shot" position briefly
        {
            SDL_Rect GunRect = {ScreenWidth - ScreenWidth / 4.5, 2 *
             VerticalView + ScreenHeight - ScreenHeight / 2.4, ScreenWidth /
             4.2, ScreenHeight / 2.2};
            SDL_Texture* Gun = GunTextures[1];
            SDL_RenderCopy(Renderer, Gun, NULL, &GunRect);
        }
    }
    if (*GunReload == true) // If gun is reloading, render based on how much
     time has passed since initial button press
    {
        int Time = SDL_GetTicks() - GunReloadRenderTime;
        if (Time > 0)
        {
            if ((Time / 150) < 6) // Render the gun in a "taking out magazine"
             with Time / 150 being what frame the gun should be rendered
            {
```

```
                SDL_Rect GunRect = {ScreenWidth - ScreenWidth / 4.5, 2 *
                 VerticalView + ScreenHeight - ScreenHeight / 2.4, ScreenWidth
                 / 4.2, ScreenHeight / 2.2};
                SDL_Texture* Gun = GunTextures[2 + (Time / 150)];
                SDL_RenderCopy(Renderer, Gun, NULL, &GunRect);
            }
            else if ((Time / 150) < 9) // Render the gun in a "inserting
             magazine" with Time / 150 being what frame the gun should be
             inserted
            {
                SDL_Rect GunRect = {ScreenWidth - ScreenWidth / 4.5, 2 *
                 VerticalView + ScreenHeight - ScreenHeight / 2.4, ScreenWidth
                 / 4.2, ScreenHeight / 2.2};
                SDL_Texture* Gun = GunTextures[8 - (Time / 150) + 6];
                SDL_RenderCopy(Renderer, Gun, NULL, &GunRect);
            }
            else // When animation is finished as Time has passed enough,
             actually reload
            {
                PlayerAmmo = 10;
                *GunReload = false;
            }
        }
    }
    if (!*GunShot && !*GunReload) // If gun is doing nothing, just render
     stationary and based on bob effect
    {
        SDL_Rect GunRect = {ScreenWidth - ScreenWidth / 4.5, 2 * VerticalView
         + ScreenHeight - ScreenHeight / 2.4, ScreenWidth / 4.2, ScreenHeight
         / 2.2};
        SDL_Texture* Gun = GunTextures[0];
        SDL_RenderCopy(Renderer, Gun, NULL, &GunRect);
    }
}

void InitializeCreateSprite(Sprite* Sprites, int ImageID, float StartX, float
 StartY, int SpriteType, float AngleIfProjectile)
{
    NumberOfSprites++;
    // If necessary, increase memory allocation for Sprites
    if (NumberOfSprites - 1 >= SpritesCapacity)
    {
        SpritesCapacity *= 5;
        Sprite* temp = (Sprite*)realloc(Sprites, SpritesCapacity *
         sizeof(Sprite));
        Sprites = temp;
    }
    Sprites[NumberOfSprites - 1] = CreateSprite(Sprites, ImageID, StartX,
     StartY, SpriteType, AngleIfProjectile); // Generates a new sprite on
     demand and inserts it into list of Sprites that we have so generously
     allocated
```

```
}

Sprite CreateSprite(Sprite* Sprites, int ImageID, float StartX, float StartY,
 int SpriteType, float AngleIfProjectile)
{
    Sprite NewSprite;
    NewSprite.ImageID = ImageID;
    NewSprite.LocationX = StartX;
    NewSprite.LocationY = StartY;
    NewSprite.SpriteType = SpriteType;
    NewSprite.UniqueID = UniqueIDCounter++;
    NewSprite.AngleIfProjectile = AngleIfProjectile;

    return NewSprite;
}

void HandleSpriteBehavior(Sprite* Sprites, bool *NoViolence, Mix_Chunk*
 FireSound, Mix_Chunk* PotionSound, Mix_Chunk* SpriteDeathSound)
{
    // Initalizing temporary variables to check for collision, similar to
     player collision
    float NewX;
    float NewY;

    for (int i = 0; i < NumberOfSprites; i++)
    {
        // If the sprite type is an enemy
        if (Sprites[i].SpriteType == 0)
        {
            if (!*NoViolence) // In case I want to showcase a pure
             raycaster...without the blood and iron. Enabled through F2
            {
                // Move the sprite!
                if ((Sprites[i].DistanceToPlayer < 8 &&
                 Sprites[i].SeenByPlayer == true) || ((Sprites[i].ImageID == 6
                 && Sprites[i].SpriteHealth < 375) || (Sprites[i].ImageID == 6
                 && Sprites[i].DistanceToPlayer < 4)))
                {
                    // Increments the enemy closer
                    if (Sprites[i].ImageID == 6) // If is dragon, move a bit
                     faster
                    {
                        NewX = Sprites[i].LocationX + -0.0024 *
                         (Sprites[i].LocationX - PlayerX);
                        NewY = Sprites[i].LocationY + -0.0024 *
                         (Sprites[i].LocationY - PlayerY);
                    }
                    else
                    {
                        NewX = Sprites[i].LocationX + -0.0016 *
                         (Sprites[i].LocationX - PlayerX);
```

```
                NewY = Sprites[i].LocationY + -0.0016 *
                 (Sprites[i].LocationY - PlayerY);
            }
            if (Map[(int) NewX][(int) NewY] != 0)
            {
                // Checking if we can't move vertically
                if (Map[(int) Sprites[i].LocationX][(int) NewY] != 0)
                {
                    NewY = Sprites[i].LocationY; // Set new position
                     to original position (don't move!)
                }
                // Checking if we can't move horizontally
                if (Map[(int) NewX][(int) Sprites[i].LocationY] != 0)
                {
                    NewX = Sprites[i].LocationX; // Set new position
                     to original position (don't move!)
                }
            }

            Sprites[i].LocationX = NewX;
            Sprites[i].LocationY = NewY;
        }

        // Attack the player!
        if (Sprites[i].DistanceToPlayer < 7 && Sprites[i].SeenByPlayer)
        {
            if (Sprites[i].ImageID == 7) // if sprite is bounty hunter
            {
                if ((SDL_GetTicks() - Sprites[i].FireTimer) > 1300)
                {
                    InitializeCreateSprite(Sprites, 3,
                     Sprites[i].LocationX + 0.2 *
                     cosf(Sprites[i].AngleToPlayer + M_PI),
                     Sprites[i].LocationY + 0.2 *
                     sinf(Sprites[i].AngleToPlayer + M_PI), 1,
                     Sprites[i].AngleToPlayer + M_PI);
                    Sprites[i].FireTimer = SDL_GetTicks();
                    Mix_VolumeChunk(FireSound, MIX_MAX_VOLUME / 4); //
                     25% volume
                    Mix_PlayChannel(-1, FireSound, 0); // Play fire
                     sound
                }
            }
            if (Sprites[i].ImageID == 5) // if sprite is mechanized
             robot
            {
                if ((SDL_GetTicks() - Sprites[i].FireTimer) > 900)
                {
```

```
                    InitializeCreateSprite(Sprites, 3,
                     Sprites[i].LocationX + 0.2 *
                     cosf(Sprites[i].AngleToPlayer + M_PI),
                     Sprites[i].LocationY + 0.2 *
                     sinf(Sprites[i].AngleToPlayer + M_PI), 1,
                     Sprites[i].AngleToPlayer + M_PI);
                    Sprites[i].FireTimer = SDL_GetTicks();
                    Mix_VolumeChunk(FireSound, MIX_MAX_VOLUME / 4); //
                     25% volume
                    Mix_PlayChannel(-1, FireSound, 0); // Play fire
                     sound
                }
            }
            if (Sprites[i].ImageID == 6) // If sprite is the dragon
             boss
            {
                if ((SDL_GetTicks() - Sprites[i].FireTimer) > 700)
                {
                    InitializeCreateSprite(Sprites, 3,
                     Sprites[i].LocationX + 0.2 *
                     cosf(Sprites[i].AngleToPlayer + M_PI),
                     Sprites[i].LocationY + 0.2 *
                     sinf(Sprites[i].AngleToPlayer + M_PI), 1,
                     Sprites[i].AngleToPlayer + M_PI);
                    Sprites[i].FireTimer = SDL_GetTicks();
                    Mix_VolumeChunk(FireSound, MIX_MAX_VOLUME / 4); //
                     25% volume
                    Mix_PlayChannel(-1, FireSound, 0); // Play fire
                     sound
                }
            }
        }
    }
}

// If the sprite is a projectile
if (Sprites[i].SpriteType == 1)
{
    // Move along designated path
    if (Map[(int) Sprites[i].LocationX][(int) Sprites[i].LocationY] !=
     1)
    {
        Sprites[i].LocationX += 0.10 *
         cosf(Sprites[i].AngleIfProjectile);
        Sprites[i].LocationY += 0.10 *
         sinf(Sprites[i].AngleIfProjectile);
    }
    // Destroy if hits a wall
    else
    {
        // printf("Projectile destroyed by contact with wall\n");
```

```
            DestroySprite(Sprites, Sprites[i].UniqueID);
        }
        // Destroy if hits player
        if (fabs(Sprites[i].LocationX - PlayerX) < 0.2 &&
         fabs(Sprites[i].LocationY - PlayerY) < 0.2)
        {
            // printf("Projectile destroyed by contact with player\n");
            PlayerHealth -= 10;
            DestroySprite(Sprites, Sprites[i].UniqueID);
        }
        // Destroy if hits a sprite
        for (int j = 0; j < NumberOfSprites; j++)
        {
            if (j != i && Sprites[j].SpriteType == 0)
            {
                if (fabs(Sprites[i].LocationX - Sprites[j].LocationX) <
                 0.2 && fabs(Sprites[i].LocationY - Sprites[j].LocationY)
                 < 0.2)
                {
                    // printf("Projectile destroyed by contact with
                     sprite\n");
                    Sprites[j].SpriteHealth -= 25;
                    DestroySprite(Sprites, Sprites[i].UniqueID);
                    // Destroy if sprite health reduced below zero
                    if (Sprites[j].SpriteHealth <= 0)
                    {
                        // printf("Sprite destroyed by health below
                         zero\n");
                        DestroySprite(Sprites, Sprites[j].UniqueID);
                        Mix_VolumeChunk(SpriteDeathSound, MIX_MAX_VOLUME);
                         // 100% volume
                        Mix_PlayChannel(-1, SpriteDeathSound, 0); // Play
                         sprite death sound
                    }
                }
            }
        }
    }
    // If the sprite is a health potion
    if (Sprites[i].SpriteType == 2)
    {
        // Give player health boost when touched
        if (fabs(Sprites[i].LocationX - PlayerX) < 0.2 &&
         fabs(Sprites[i].LocationY - PlayerY) < 0.2)
        {
            // printf("Potion destroyed by contact with player\n");
            if (PlayerHealth < 50)
            {
                PlayerHealth += 50;
            }
            else
```

```
                {
                    PlayerHealth = 100;
                }
                Mix_VolumeChunk(PotionSound, MIX_MAX_VOLUME); // 100% volume
                Mix_PlayChannel(-1, PotionSound, 0); // Play fire sound
                DestroySprite(Sprites, Sprites[i].UniqueID);
            }
        }
    }
}

void DestroySprite(Sprite* Sprites, int UniqueID)
{
    for (int i = 0; i < NumberOfSprites; i++)
    {
        if (Sprites[i].UniqueID == UniqueID)
        {
            // Replaces every element, starting with the one we want to
             destroy, with the next element in the Sprites list
            for (int j = i; j < NumberOfSprites; j++)
            {
                Sprites[j] = Sprites[j+1];
            }
            NumberOfSprites--;
            return;
        }
    }
}
```