Report Documentation

By Joshua Todd

Programming for games

City 1050

25/04/2018

Table of Contents

Product Specification Version 1 (Before Final Game Version)

Elements highlighted in orange have been changed in version 2

Main aim of the game:

The player will rotate around the earth clockwise or anticlockwise and protect it from incoming meteors for as long as possible.

Sprites

Needed Sprites: Spaceship, projectile, meteor, planet, background with stars, title screen to start the game.

Animated Objects

Meteors will be animated to look like they are rotating, possible animated background.

Collision Detection

Collisions will happen between the player's bullets and the meteors, the meteors will have a certain amount of health.

Collisions will also happen between the meteors and the planet, meteors will take a set amount of health away from the planet upon collision.

The player's ship will not be able to be damaged.

Artificial Intelligence

Meteors will try to crash into the earth.

After every 30 seconds a boss will appear in the form of an alien spaceship and will try to shoot down the planet and avoid the players bullets.

Win Lose State

Player's aim is to get the highest survival time as possible.

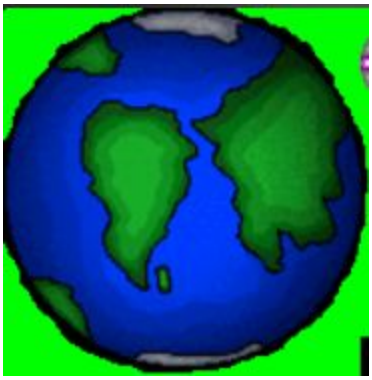
Player loses when the health of the planet reaches zero.

Product Specification Version 2 (Final Game Version Complete)

Main aim of the game:

The player will rotate around the earth clockwise or anticlockwise and protect it from incoming meteors for as long as possible.

Sprites



Planet

Ship

Bullet

Meteor

Life

Menu Bitmaps

Animated Objects

Meteors are animated to look like they are rotating, meteors have 10 frames of animation and animate slowly.

Collision Detection

Collisions happen between the player's bullets and the meteors, the meteors will have a set amount of health and deactivate upon 1 collision with a bullet.

Collisions happen between the meteors and the planet, the meteors take 1 life away from the planet upon collision.

The player's ship is not able to take damage.

Artificial Intelligence

Meteors will try to crash into the earth.

The radius of the meteors is reduced gradually until they either collide with the planet or a bullet. Animation is also implemented into the meteors.

Win Lose State

Player's aim is to live as long as possible while protecting the earth.

Player loses when the health of the planet reaches zero.

Updates Considered for Future Versions

Meteor Updates

Meteors could have the ability to collide with each other and change direction but still get pulled into the planet's gravitational field. The collision would not destroy them.

More animation could be added and different size meteors for more interesting variation.

Big meteors could split into smaller meteors when hit by a bullet.

Bullet Updates

Bullets could change upon special pickups dropped by destroyed meteors, this can change the number of bullets or fire multiple at once. Laser and rocket upgrades would also be interesting.

Planet Updates

Planet may also have an animation to slowly rotate to make it more interesting.

Background Updates

The background is in need of some more colour and animation, stars could twinkle and galaxies in the background could slowly rotate.

Scoring system

A scoring system is desperately needed as there is currently no sense of achievement in the game, each meteor destroyed could give the player a certain amount of points that would be displayed to them in the top corner of the screen.

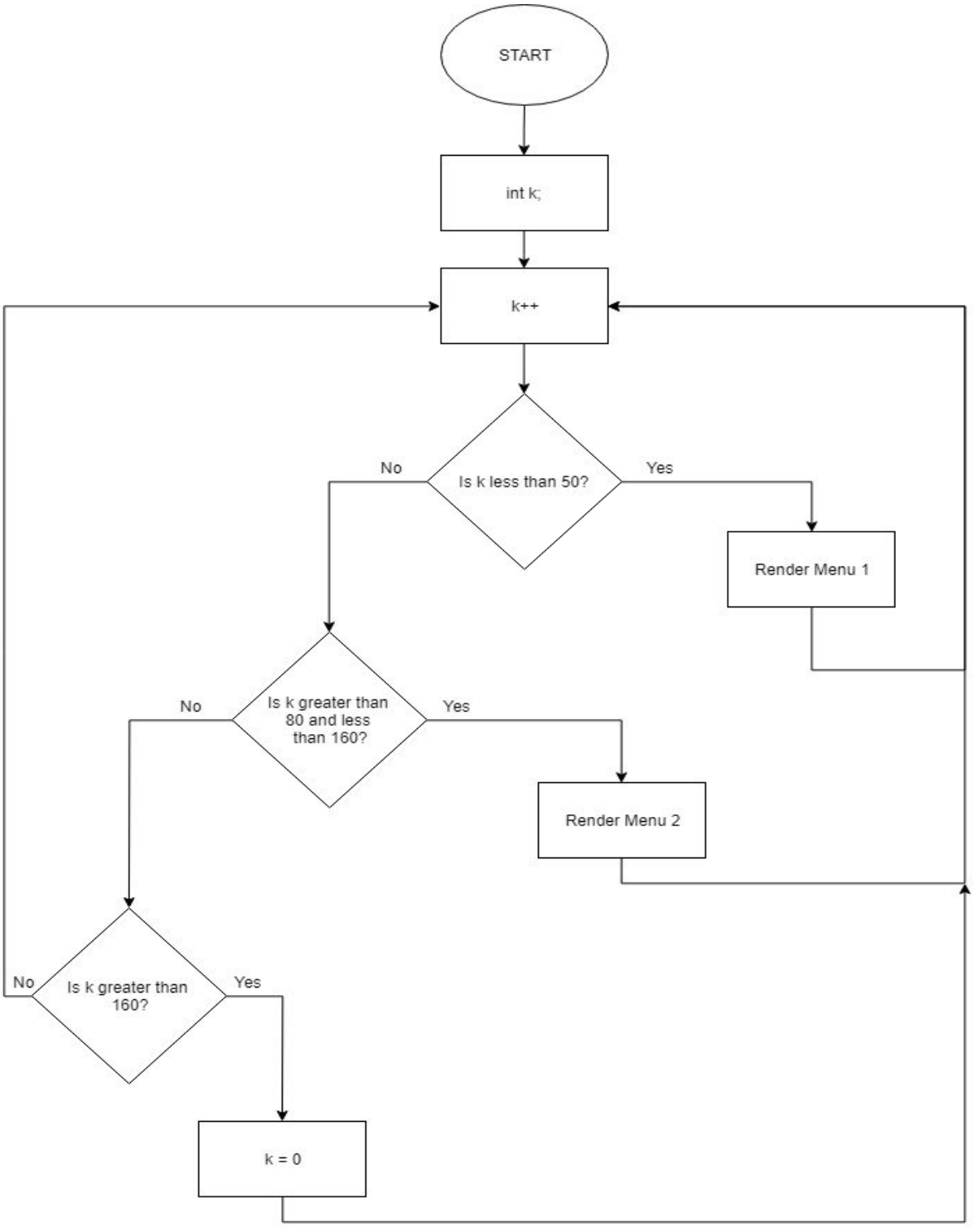
Reset System

The game is currently unable to reset all the meteors and leaves one behind

Flowcharts for Game Development

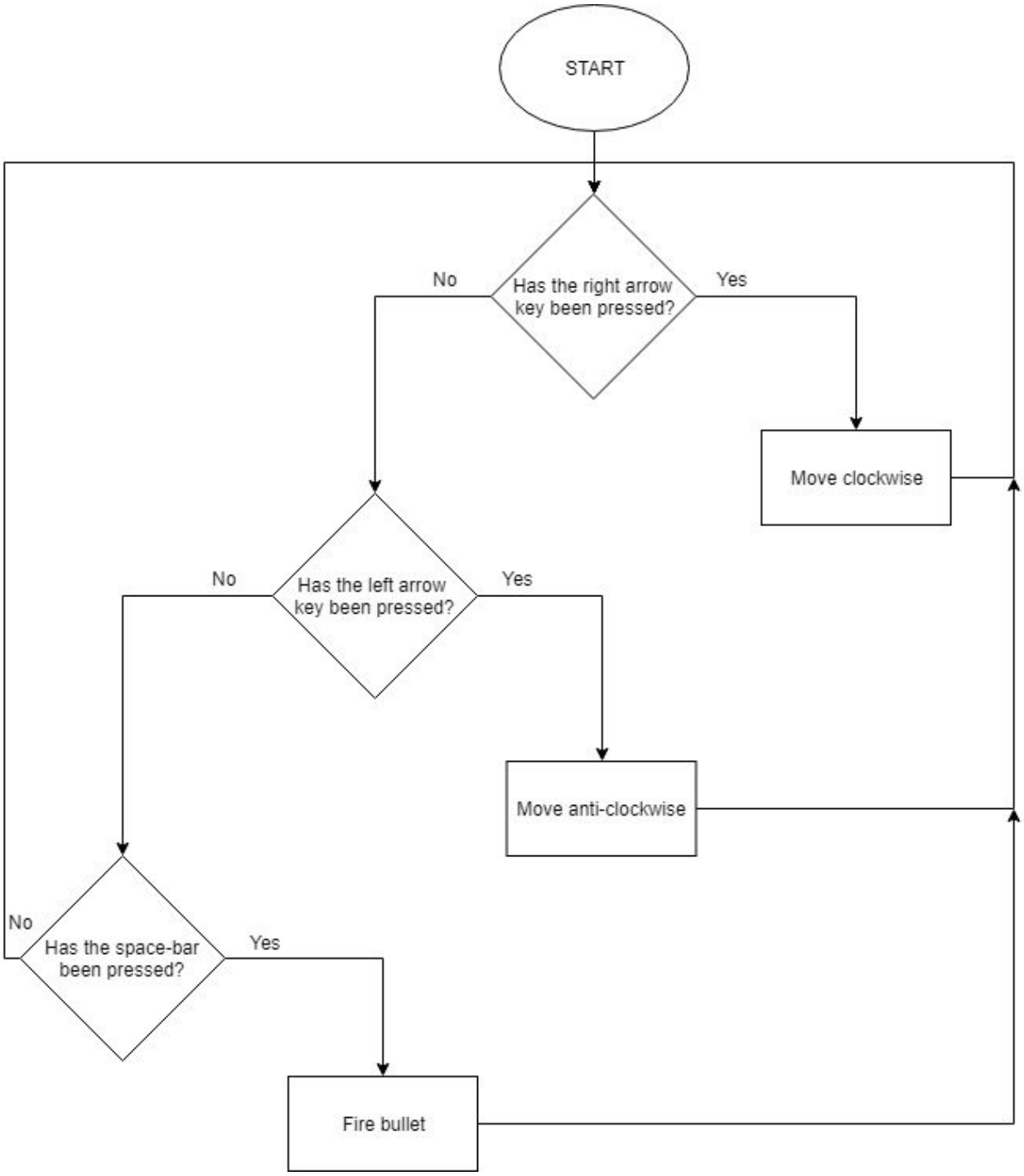
MenuSwitcher();

Flowchart for switching the menu, values were later changed to 80 and 160 to slow the flashing between menus.



Character Movement();

This function was later renamed to CharacterController(); and the code for firing the bullet was written in the BulletLocation(); function.



Explanation of Algorithms

UpdateFrame();

UpdateFrame(); is the main function that runs every frame, in this case it is running at 60 fps.

```
void UpdateFrame(void)
```

Within are 3 variables of type RECT that tell the program the locations of the sprite's that need to be rendered.

```
                                //x  y  x2   y2
RECT            planetSprRect = { 0, 0, 149, 150 }; //Rectangle for grabbing image from sprite sheet
RECT            characterSprRect = { 149, 0, 189, 49 };
RECT            backSprRect = { 300, 300, 1324, 1065 };
```

To make the frames run as smooth as possible a delay is implemented, meaning if the game is stuck on a frame for too long the program will load the next frame.

```
// Milliseconds since program started
thisTickCount = GetTickCount();

//Animation step
if ((thisTickCount - lastTickCount) > delay)
{
    //Move to next frame;
    lastTickCount = thisTickCount;
    currentFrame++;
    if (currentFrame > maxFrame) currentFrame = 0;
}
```

MenuSwitcher(); is called to display the menu when the program first starts, there are then 2 if statements to allow the game to leave the menu and run.

The array start[z] is passed the value of 1 when the enter key is pressed, the following if statement checks the value of start is equal to 1. If true then the rest of the functions will run. Without storing a value in an array the game will only run if the user holds down the enter key, the array allows the game to run until the value of the array changes.

```
MenuSwitcher();

if (KEY_DOWN(VK_RETURN))
{
    start[z] = 1; // pass a value into the array
}

if (start[z] == 1) // runs if the button is pressed once, keeps running
{
    CharacterController();

    BackgroundRender(baspritex, baspritey, backSprRect);

    CharacterRender(chspritex + 4, chspritey + 4, characterSprRect);

    PlanetRender(plspritex, plspritey, planetSprRect);

    BulletLocation();

    MeteorLocation();

    Collision();

    SpriteAnimation();
}
```

SpriteAnimation();

SpriteAnimation(); function allows the sprites to animate.

```
void SpriteAnimation(void)
```

The algorithm to allow sprites to animate is quite simple, the first if statement will only run every 20 frames, this controls the speed of the animation.

Inside the if statement xSprite and x2Sprite are incremented by 32, the size of the sprites to animate are 32x32 so this will grab every sprite accurately.

```
if (currentFrame % 20 == 0) // runs every 20 frames
{

    xSprite += 32; // increment sprite grabbing value by 32
    x2Sprite += 32;
```

xSprite and x2Sprite are used in the MeteorLocation function to render the meteors.

```
RECT meteorSprRect = { xSprite, ySprite, x2Sprite, y2Sprite };
```

The last part of the function is another if statement inside the previous one that checks if x2Sprite is greater than 528. 528 is the end point for the final animation sprite, it then returns to the first one by setting xSprite and x2Sprite back to their first value.

```
if (x2Sprite > 528) // if the renderer goes paast the last sprite on the bitmap
{
    xSprite = 209; // send it back to the first sprite
    x2Sprite = 241;
}
```

Collision();

The Collision function handles the collision of the bullets and the meteors.

```
void Collision(void)
```

Two for loops are used to first check every meteor and bullet currently rendered.

```
for (int i = 0; i < MB; i++) // check for every bullet
{
    if (ba[i] == 1)
    {
        for (int a = 0; a < MC; a++) // check for every meteor
        {
            if (ma[a] == 1) // if both bullet and meteor are active do this
```

If both the meteor and bullet are active it puts the x and y location of the sprites at their current location then using pythagoras theory for circular collision deactivates the current meteor and bullet if they collide.

```
bspritex = screenWidth / 2 - 12 + br[i] * cos(bA[i]); // check the location of the meteors and bullets
bspritey = screenHeight / 2 - 12 + br[i] * sin(bA[i]);
mespritex = screenWidth / 2 - 25 + mr[a] * cos(mA[a]);
mespritey = screenHeight / 2 - 25 + mr[a] * sin(mA[a]);
if ((mespritex - bspritex)*(mespritex - bspritex) + (mespritey - bspritey)*(mespritey - bspritey) < (25 * 25))
{
    ba[i] = 0;
    ma[a] = 0;

    score++; // increment score
}
```

Rendering

All the functions below do the same thing apart from some of them render their sprites from different files.

```
void PlanetRender(int x, int y, RECT cropRect){ ... }
void CharacterRender(int x, int y, RECT cropRect){ ... }
void BackgroundRender(int x, int y, RECT cropRect){ ... }
void Menu1Render(int x, int y, RECT cropRect){ ... }
void Menu2Render(int x, int y, RECT cropRect){ ... }
void MeteorRender(double x, double y, RECT cropRect){ ... }
void BulletRender(double x, double y, RECT cropRect){ ... }
void LivesRender(int x, int y, RECT cropRect){ ... }
```

For example CharacterRender(); uses g_pDDOne to render sprites from the ALL.bmp

```
hRet = g_pDDSBack->BltFast(x, y, g_pDDSOne, &cropRect, DDBLTFAST_SRCCOLORKEY);
```

And Menu1Render(); uses g_pDDSthree to render sprites from the METEORS1.bmp

CharacterController();

The CharacterController(); function handles the movement of the player.

```
void CharacterController()
{
```

It first of all places the character to rotate around the center of the screen by dividing the height and width of the screen by 2 then subtracting 25 to get the centre of the sprite. By subtracting 25 the centre of the sprite acts as if it is in the middle of the sprite, without doing this the centre point of the sprite would be in the top right corner and it would not rotate around the planet in the correct fashion.

```
chspritex = screenWidth / 2 - 25 + radius * cos(angle);
chspritey = screenHeight / 2 - 25 + radius * sin(angle);
```

2 if statements are used asking if the left or right key is pressed, while the conditions are true the character will move clockwise or anticlockwise depending on which key is pressed.

```
if(KEY_DOWN(VK_RIGHT)) // increment angle when pressing right to go clockwise
{
    angle += 0.035;
}

else if(KEY_DOWN(VK_LEFT)) // opposite
{
    angle -= 0.035;
}
```

BulletLocation();

The BulletLocation(); function handles the algorithms for the characters bullets.

```
void BulletLocation()
```

Firstly there is an if statement checking if space bar has been pressed and the current frame is a 10th frame, inside is a for loop that starts a counter from 0 to 10 as the maximum amount of bullets that can be rendered at once is 10. Inside that for loop is another if statement checking if a bullet is active on a 10th frame, if not it passes the angle and radius of the bullet into arrays and makes the bullet active.

```
if (KEY_DOWN(VK_SPACE) && currentFrame % 10 == 0) // is spacebar is pressed every 10 frames
{
    for (int counter1 = 0; counter1 < MB; counter1++) // increment counter until 10
    {
        if (ba[counter1] == 0 && currentFrame % 10 == 0) // if bullet is incative on a 10th frame
        {
            br[counter1] = bulletRadius; // pass values into arrays
            ba[counter1] = 1; // make bullet active
            bA[counter1] = angle;

            counter1 = MB;
        }
    }
}
```

A for loop starts another counter to check if the bullet is active, if so increment the radius of the bullet by 10 pixels each frame, pass values of the sprite's x and y and then render the bullet at these coordinates. Finally there is an if statement to check if the bullet has reached the edge of the screen, if so make that bullet inactive.

```
for (int counter1 = 0; counter1 < MB; counter1++)//first check if bullet is active
{
    if (ba[counter1] == 1) // if bullet is active
    {
        br[counter1] += 10; //move bullet 10 pixels each frame
        bspritex = screenWidth / 2 - 12 + br[counter1] * cos(bA[counter1]); // dput bullet at this location
        bspritey = screenHeight / 2 - 12 + br[counter1] * sin(bA[counter1]);

        BulletRender(bspritex, bspritey, bulletSprRect); // render the bullet

        if (br[counter1] > 650)//if bullet at edge of screen
        {
            ba[counter1] = 0;//make the bullet inactive
        }
    }
}
```

MeteorLocation();

The MeteorLocation(); function handles the movement for the AI of the meteors and the health of the planet.

```
void MeteorLocation()
```

Firstly the random number generator is seeded, then a variable type double is declared as random and is passed a random number from 0 to 360. This value is then passed to another variable called meteorAngle.

```
srand(time(NULL)*GetTickCount()); // seed random number generator

double random = rand() % (360 - 1 + 1) + 1; // pass random a random value each time
double meteorAngle = random; // make meteor angle random
```

This function works in a very similar way to the BulletLocation(); function. It also uses for loops to check every meteor and passes values into arrays to determine the angle and radius of the next meteor that wall spawn. However this is called on every 20th frame.

```
if (currentFrame % 20 == 0) // run every 20 frames
{
    for (int counter2 = 0; counter2 < MC; counter2++) // start a counter to run until 1000
    {
        if (ma[counter2] == 0 && currentFrame % 20 == 0)  // if meteor is inactive and its a 20th frame
        {
            mr[counter2] = meteorRadius; // pass values to arrays
            ma[counter2] = 1;
            mA[counter2] = meteorAngle;

            counter2 = MC;
        }
    }
}
```

In the second for loop it does a similar thing to the second for loop of BulletLocation();, it moves the meteors at a certain number of pixels each frame and renders them.

```
for (int counter2 = 0; counter2 < MC; counter2++)//first check if meteor is active
{
    if (ma[counter2] == 1) // is meteor is active
    {
        mr[counter2] -= 0.35; //move meteor 0.35 pixels each frame
        mespritex = screenWidth / 2 - 25 + mr[counter2] * cos(mA[counter2]); //set location of x and y
        mespritey = screenHeight / 2 - 25 + mr[counter2] * sin(mA[counter2]);

        MeteorRender(mespritex, mespritey, meteorSprRect); // render meteors
```

Below, four if statements are used to determine how many lives the player has. The player has 3 lives to start with and every meteor that hits the planet loses the player 1 life.

```
if (planethealth == 3) // is the health of the planet is 3, display 3 lives
{
    RECT lifeSprRect = { 528, 0, 626, 32 };
    LivesRender(20, 20, lifeSprRect);
}
if (planethealth == 2)// is the health of the planet is 2, display 2 lives
{
    RECT lifeSprRect = { 528, 0, 594, 32 };
    LivesRender(20, 20, lifeSprRect);
}

if (planethealth == 1)// is the health of the planet is 1, display 1 life
{
    RECT lifeSprRect = { 528, 0, 562, 32 };
    LivesRender(20, 20, lifeSprRect);
}
```

The final if statement checks to see if the meteor is within the radius of the planet, if true it makes that meteor inactive and reduces the player lives by 1. If the player runs out of lives it takes the game back to the title screen and resets the lives back to 3.

```
if (mr[counter2] < 100)//if meteor is at planet radius
{
    ma[counter2] = 0;//make the meteor inactive

    planethealth -= 1; // decrement planet health by 1

    if (planethealth < 1) // if plate health is less than 1
    {
        start[z] = 2; // show menu screen
        planethealth = 3; // reset the planets health
    }
}
```

MenuSwitcher();

MenuSwitcher(); is the final function that allows the title screen to flash “Press Enter to Play”.

```
void MenuSwitcher()
{
```

Every frame is increments a variable k as a counter and uses 3 if statements to determine when to switch between bitmaps. If k is less than 80 it will show the menu with the pay text, when k is greater than 80 is will display the menu without the text. Once k reaches 160 it resets k back to zero to start the count again.

```
RECT            menu1SprRect = { 0, 0, 1024, 768 };
RECT            menu2SprRect = { 0, 0, 1024, 768 };

k++; // increment

if (k < 80) // if i is less than 80 display menu 1
{
    Menu1Render(0, 0, menu1SprRect);
}
if (k > 80 && k < 160) // if i is more than 80 and less than 160 display menu 2
{
    Menu2Render(0, 0, menu2SprRect);
}
if (k > 160) // set i back to 0 if it is greater than 160
{
    k = 0;
}
```

Constraints and Weaknesses

Constraints of the Application

Sprite rotation was found to be an issue, no simple method of rotation sprites was found. This is the reason the sprite for the players ship is a symmetrical circle, the same goes for the bullet.

Initially the ship was more plane like looking and the bullets looked more like long lasers, because i could not find a way to rotate the sprites in the application i ended up making everything circular.

Creating a game based on circular movement gave me a much bigger challenge when coding the game, the pythagorean theorem had to be implemented in all functions that contailed movement.

Creating text seems to be very difficult, I created the numbers 0 - 9 and added them to the main bitmap however translating an integer in the code into the correct sprites to be displayed on the screen without hard coding every single number to display the correct sprites will take a very clever algorithm. This is currently beyond my skill set.

Weaknesses in the Program

The program has a few weaknesses that would need to be addressed to update the game for future versions. Right now in my eyes it is as basic as the game can be, you can lose however you have no indication of how well you played. A scoring system will need to be implemented for it to be actually fun.

Currently the game does not get more challenging over time i think the meteors would need to slowly speed up or get greater in numbers.

When you lose all of your 3 lives the game puts you back into the title screen, if you press enter to play again the game seems to allow 1 meteor to still be there from before when the player died

Testing

A Problem with Displaying the Lives

When I was implementing the lives into the code I had a problem displaying them properly to the player, after one meteor collided with the planet the lives went from displaying 3 to 1 then 0.

This was the code that was written:

```
    if (planethealth == 3)
    {
            RECT lifeSprRect = { 528, 0, 626, 32 };
            LivesRender(20, 20, lifeSprRect);
    }

    if (mr[counter2] < 100)//if is at planet radius
    {
        ma[counter2] = 0;//make the meteor inactive

        planethealth--;

        if (planethealth == 2)
        {
            RECT lifeSprRect = { 528, 0, 594, 32 };
            LivesRender(20, 20, lifeSprRect);
        }

        if (planethealth == 1)
        {
            RECT lifeSprRect = { 528, 0, 562, 32 };
            LivesRender(20, 20, lifeSprRect);
        }

        if (planethealth < 1)
        {
            start[z] = 0;
            counter2 = MC;
            mr[counter2] = meteorRadius; // reset meteors if planet health is less than 1?
            planethealth = 3;
        }
    }

}
```

I realised the reason was that the if statements regarding the number of lives should be outside the if statement that checks to see if the meteor has collided with the planet.

Like so:

```
if (planethealth == 3)
{
    RECT lifeSprRect = { 528, 0, 626, 32 };
    LivesRender(20, 20, lifeSprRect);
}
if (planethealth == 2)
{
    RECT lifeSprRect = { 528, 0, 594, 32 };
    LivesRender(20, 20, lifeSprRect);
}

if (planethealth == 1)
{
    RECT lifeSprRect = { 528, 0, 562, 32 };
    LivesRender(20, 20, lifeSprRect);
}
if (mr[counter2] < 100)//if is at planet radius
{
    ma[counter2] = 0;//make the meteor inactive

    planethealth -= 1;



    if (planethealth < 1)
    {
        start[z] = 0;
        counter2 = MC;
        mr[counter2] = meteorRadius; // reset meteors if planet health is less than 1?
        planethealth = 3;
    }
}
```

This way the lives now display correctly to the player.

Resetting the Game After Losing

There was a problem resetting the game after the player had lost all 3 lives. The program would take the player back to the title screen and then prompt to press enter to play again. If enter was pressed the game would resume as it left off acting more like it was paused rather then reset.

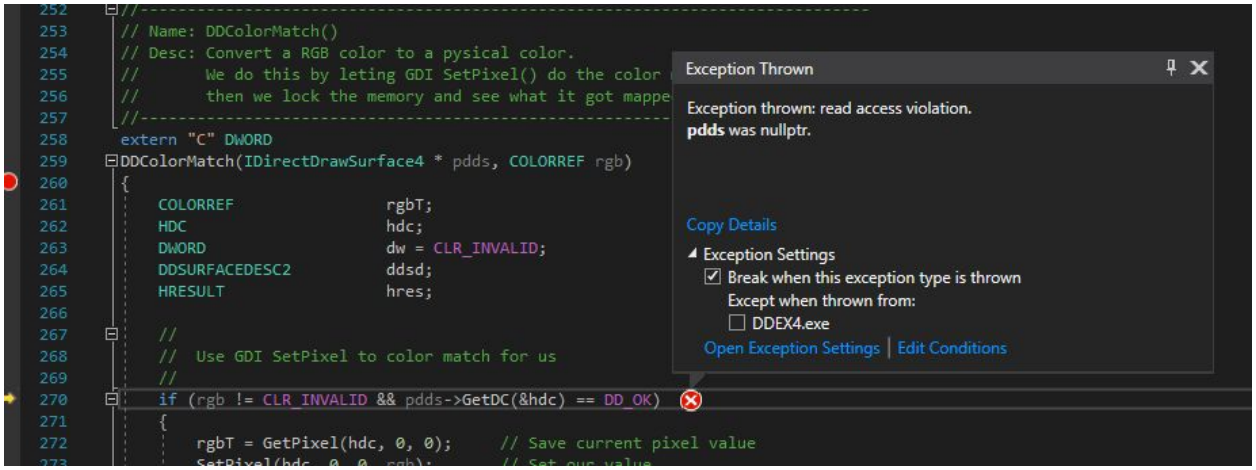
The fix was to make all current meteors inactive when the player lost all their lives.

```
    }
}
if (start[z] == 2)
{
    ma[counter2] = 0;
}
```

However this seems to be only a partial fix, one meteor is always left behind from the previous game when the game is restarted. This needs to be fixed for the next version.

Using Breakpoints to Test the Code

When adding two new bitmaps for the menu to work I had an error in another file that had code I did not recognise. I used breakpoints to check exactly where it had broken.



Truthfully I didn't know how to fix this for a while, but I knew it was something to do with the new bitmaps I had added. Looking through the code I realised that I had an error where the bitmaps were included.

```
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENG)
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_UK
#pragma code_page(1252)

//////////////////////////////////////////////////////////////////////////////
//
// Bitmap
//

ALL                     BITMAP                  "ALL.BMP"

METEORS1                BITMAP                  "METEORS1.BMP"

METEORS2                BITMAP                  "METEORS2.BMP"

TILE                    BITMAP                  "tiles.bmp"


//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems
```

As it turns out the string name for the file it was looking for had a typo and so it could not find the correct file. After this was corrected the programme ran smoothly.