Exam No: Y3857545

Intro to Programming – Written report

1 - Abstract

The game created presents a player with a scenario in which they must launch a rocket from Earth to Mars in two-dimensional space. A randomly located moon offers an obstacle, with its own gravitational field five times that of its radius, which will affect the rocket's flight path. The user sets a launch angle and velocity and then the scenario plays out. The user is then presented with a score inversely proportional to the distance travelled by the rocket.

The task was broken down into a number of steps, which if carried out chronologically, will allow the programmer to test their new code using the old code, making this an iterative process. The steps identified were the creation of the menus, the arena, the rocket, the main game loop and finally the moon's gravitational area of influence. These were then broken down again into functions, which are segments of code that carry out a predefined task.

The program contains a number of structures for storing the characteristics of the more complicated aspects of the game, such as the moon and rocket. It also has several dependencies which are files needed to run the program, namely stdio.h, stdlib.h, math.h and graphics_lib.h. The game was produced iteratively, so few bugs can be found.

An ideal solution would fix all the issues stated, as well as carrying out several other relatively simple tasks, such as high score tracking and landing speed monitoring.

2 - Problem Description and Analysis

Requirements

The company requested that an imaginative and engaging Earth to Mars rocket launch scenario was to be created in such a way that it could be ported over to different applications, such as an Android app. The rocket must launch at a chosen angle and velocity and then travel across space towards the target, Mars. A randomly generated moon will then present a mathematically accurate gravitational effect on the rocket inside an artificial area of influence which is five times greater than moons radius. The moons location must be random for each game and must be between the source and destination of the rocket. There should also be a scoring system, which is inversely proportional to the distance travelled by the rocket. Another undefined feature must also be implemented, such as additional animations.

Limitations

Programming constrictions required this to be a two-dimensional game. This is because the GFX library, that was requested by the client, had no three-dimensional support. This created a mathematical limitation. If three-dimensional physics could not be applied, they could not be used in calculations. This meant that only x and y travel would be possible.

Another limitation was the refresh rate of the rocket. As this is only a mini game, it would not be played solely on high spec PCs or consoles and will need to run optimally on devices with less power, such a smart phone. Thus, a refresh rate of one second was selected to ensure slower devices could play the game without struggling.

Iterative design

It is evident that applying the laws of physics to a computer game is difficult. Therefore, using an iterative approach when creating the game made the most sense. By working in steps, the physics can be tackled chronologically, starting by processing launch and proceeding to the effect of the moon on the rocket. This helps the programmer to think logically and determine if calculations are working, allowing issues to be fixed on the spot.

Iteration enables the game to be tested during the development stages, allowing adjustments to be made throughout the development process. Any compatibility issues could also be dealt with during the building of the game rather than at the end, which would be the case if the program was fully segmented and combined into one long program.

Exam No: Y3857545

Game description

By the end of the development process, a two-dimensional game should be fully playable, with the player able to launch a rocket from Earth to Mars and see the effect of the moon's gravity in real time. When the game launches the user should be greeted with a welcome screen. Upon success or failure, the user should be presented with their score

Initial approach

It was found that the best method to develop such a complex game was to compartmentalize the task, so that the problem was split into several sections:

- The menu screens, this includes the welcome screen and the game over screen.
- Designing an arena in which the game can be played.
- Creating a rocket that can launch at varying angles.
- Implementing a main game loop which can move the rocket, keep track of its score and take care of collision detection.
- Making a gravitational area of influence around the moon.

There was some overlap between collision detection and the gravitational area of influence, so the main game loop used its collision detection code to detect when the rocket enters the moons area of gravitational influence and trigger the 5th point stated above.

At this point, it became clear that the game would almost definitely need to be two-dimensional as the processing required for three-dimensional physics appeared impossible for one person to create in the given time frame.

The menus

A welcome screen should greet the user, giving an outline of the challenge posed by the game. This can then be cleared with a key press and the game can begin. After the game, the user should be represented with their score and options to replay or quit in the form of click boxes.

The arena

The arena must contain:

- Earth, Mars and a background
  This is simple to implement; three bitmaps are placed accordingly across the arena. Their positions and sizes can then be altered to suit the screen.
- Moon
  The moon must be created at a random position and of a random diameter, this covers the extra game dynamic requested. To keep up the high standard set by using bitmaps as planets, the moon should also be a bitmap.
- Game controls
  Click to increment boxes with velocity and the launch angle should be made as well as a tutorial paragraph telling the user how to play. The program can then wait for a launch command before proceeding, such as pressing enter.

This will give a good base on which to place the "moving parts" of the game and then play it, such as the rocket. Furthermore, as there is only limited screen space, collision detection should be used to detect if the rocket goes off screen. The game can then end.

The rocket

Visually, the rocket should also be a bitmap to continue the game's aesthetics. As a data structure, the rocket should be globally accessible for the "Moon's gravity" segment so that its angle of travel can be altered in and outside of the moons influence.

The main game loop

The main game loop starts when the user wishes to launch the rocket and comes to an end when the user either succeeds or fails, which is dictated by what they collide with. Should the main game loop continue past either of these points, the game will be rendered unplayable, as it will never end.

For simplicity, each execution of the loop will be one second of in game time as this drastically simplifies many calculations which in turn makes for a smoother running and shorter program. This leads directly onto the movement of the rocket which, put simply, is vector addition. In vector addition, two different vectors are combined using trigonometry to produce a singular, resultant vector. However, in this case the resultant rector will be provided by the user namely, the launch angle. This is called the hypotenuse.

The other two sides of the triangle will represent the x and y resultant movements as the opposite and adjacent sides of the triangle respectively.

Thus, the rocket's x movements can be found using the cosine rule of trigonometry, let R equal the hypotenuse...

$$opp = \cos cos\,(hyp)$$

The same can be said for the rocket's y movement which can be found using the sine rule of trigonometry, let R equal the hypotenuse...

$$adj = sin(hyp)$$

The users chosen acceleration must be taken into account by multiplying both x and y vectors by the acceleration. As one execution is one second and the velocity is in meters per second this will move the rocket as many meters, which are represented by pixels, as required for one second of movement.

$$movement\ (in\ m/s) = \ adj * acceleration$$

$$movement\ \ (in\ m/s) = \ opp * acceleration$$

These can then be added to the rockets original position in order to move it.

The main game loop must also handle collision detection, as after each execution of the loop objects will be moved. For the planets and moon, this is done by taking a distance between the rocket and the object in question and comparing it to the object's radius. If the distance is less than the radius, then a collision has occurred and appropriate actions can then be taken. If the rocket leaves the screen this can be detected by setting boundaries, then if the rocket's coordinates exist outside it will trigger an action. If a collision with Mars is detected, then the game has been won, but anything else results in failure for the player.

The moon's gravity

Accurately implementing the moons gravity requires several steps:

- Entering the field of effect, taking into account the angle of entry.
- Accurately navigating the rocket through the area of influence.
- Leaving the area, considering the rockets angle of flight.

As the rocket enters the area of influence, it will do so at the launch angle, this is already a known variable. The game then progresses and the rockets angle of travel changes this variable can be used to store the new angle of travel. Therefore, when the rocket leaves the area of influence, the function which moves the rocket outside the area of influence will still function correctly.

As for moving the rocket inside the area, this will need to take into account the distance between the rocket and the moon, as it was requested in the brief that the force applied on the rocket increases as it gets closer to the moon. This is modelled with the equation...

$$f = G * \frac{m_1 m_2}{r^2}$$

G is the gravitational constant, therefore it is known. m(1) and m(2) are the masses of the moon and rocket. These will also remain as constants and to begin with the real-world values of the weight of the space shuttle (30,000 Kg) and the moon (7.34767309 * 10^22 Kg). These values can be altered during development to change the balance of the game for smoother running. Finally, r is the distance between the centre of the two masses. This is calculated by taking the x and y locations of the rocket and subtracting the moons x and y coordinates from their respective counterparts.

In order to apply this force, it must be turned into an acceleration value. This is done using the equations…

$$v = at \text{ and } a = \frac{f}{m}$$

With force (f) and mass (m) now known, the acceleration can be found using the second equation. Therefore, using the first equation, a velocity can be found for the execution of the main game loop. As mentioned before, one execution is equal to one second to keep things simple. Thus, the first equation can be changed too...

$$v = a\,(1)$$

Making the first equation equal…

$$v = \frac{f}{m}$$

This makes calculating the velocity of the rocket, due to the moon's gravitational pull, much more efficient as it requires less lines of code to implement.

If the magnitude of the vector has been found (the velocity), the direction must be found. This is the angle at which the rocket is travelling with regard to the x axis. The rocket's angle of travel is calculated with regard to the moon, as shown below. However, it is needed with regard to the x axis. In order to make this change this the area of influence around the moon was split into four quadrants, identified by combinations of upper, lower, left and right. The angles can then be altered accordingly, one quadrant remains the same and the others take additions in multiples of 90 degrees.

Once this is complete, a vector has been found for the rocket's travel towards the moon and this needs to be added to the previous rocket vector as this ensures that the angle at which the rocket entered the area is taken into account, making the equations look like this…

$$x\ movement = \cos cos\ (previous\ angle) * previous\ velocity + \cos cos\ (new\ angle) * previous\ angle$$

$$y\ movement = \sin sin\ (previous\ angle) * previous\ velocity + \sin sin\ (new\ angle) * previous\ angle$$

The rocket can now be moved by adding these resultant values to the rocket's current coordinates. The velocity should also be adjusted accordingly.

Technical specification

In order to meet the design brief, a comprehensive list of needed and extended features was assembled.

Prescribed features

A comprehensive list of all the features incorporated into the program because of the design brief are as follows…

- Earth and Mars
- The moon
- Launch controls
- A scoring system
- The rocket
- The gravity

Extended features

- A welcome screen

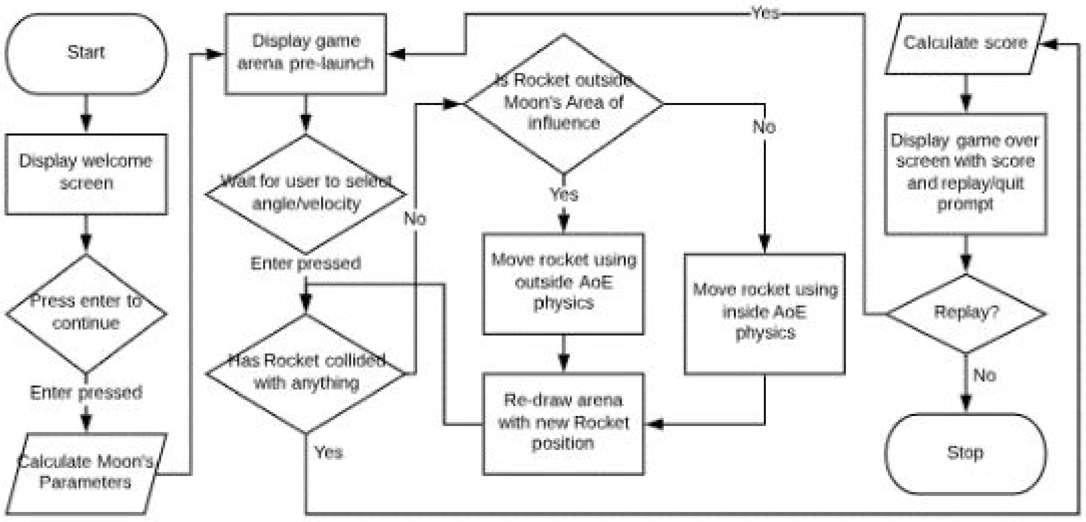
- Moon resizing
- Bitmaps
- A replay
- A background

Iterative design

As discussed previously, the problem was then split into several sections, each to be iteratively developed. The sections are as follows…

- The menu screens.
- Designing an arena.
- Creating a rocket.
- Implementing a main game loop.
- Making the gravitational area of influence.

Using the analysis carried out in the previous section on these points a flowchart can be deduced…


Start
Display welcome screen
Press enter to continue
Enter pressed
Calculate Moon's Parameters
Display game arena pre-launch
Wait for user to select angle/velocity
Enter pressed
Has Rocket collided with anything
Yes
No
Is Rocket outside Moon's Area of influence
Yes
No
Move rocket using outside AoE physics
Move rocket using inside AoE physics
Re-draw arena with new Rocket position
Yes
Calculate score
Display game over screen with score and replay/quit prompt
Replay?
No
Stop

This was then translated into C code, with testing occurring upon the completion of each of the predefined steps. More about this can be found in the testing section.

Libraries

Only four libraries are needed for the program to run. Firstly “stdio.h” and “stdlib.h” are required to do some of the simplistic tasks required by most C programs. These include defining variable types, loops and pointers. All of which feature in the program. “math.h” was included to handle various sin, cos and tan functions as well as running the random integer generator. Finally, “graphics_lib.h” was needed to run all of the graphical parts of the game, such as creating a window and handling bitmaps within the window.

Functions, structures and features

As the development process continued, the segmentation of different features and their associated code became clear. This meant that a comprehensive list of functions and features associated with them could be assembled. The reason for each feature’s assignment to a function could also be justified.

- main(void)
  The main function is a necessity in every c program and will always run first. As a result, it contains all the initial setup required to making the GFX window where the game will be displayed, as well as creating the event queue and calling all of the other functions. At the end of main, the welcome screen function is called, then the wait for enter function and finally the play game function which is recursive. Therefore, main will not run after its first execution.
- welcomeScreen(void)

Extended features – Welcome screen
The first thing the user will see when booting the game up is the result of this function. It creates a background and displays an introductory message to the user.

- waitForEnter(void)
  This function simply hangs (holds execution of the program) until the user presses enter to give them time to read the introductory message. This is a separate function and was not included with the welcome screen function to increase segmentation within the program.
- calculateMoon(struct moon_struct moon)
  Features – Moon, Gravity
  Extended Features – Moon resizing
  Random coordinates are created using the random number function and then stored in the moon structure as “x” and “y”. One of five random radiuses is then selected using the random number generator and stored under “d”. The area of influence is then calculated and stored as “i”.
- drawPlanets(struct moon_struct moon)
  Features – Earth, Moon, Mars
  Extended Features – Bitmap handling
  Earth and Mars are drawn at set locations defined in this function. The moon is drawn at the location stored in the structure “moon”, and if tree had to be used for each location as bitmaps cannot be resized and the function would not allow for a file path stored in a variable. Drawing is done using the BITMAP function of the GFX library. This was used to make the game more visually appealing. because just having white, green and red circles as planets did not look particularly appealing. The three bitmaps and their file paths are below…
- drawBackground(void)
  Extended Features – Background
  A bitmap (file path and example below) is drawn over the entire screen. This was done to make the game more visually appealing as the alternative is GFX’s blank black background.
- drawControlBoard(struct rocket_struct * rocket)
  Features – Adjustable launch angle and velocity, ability to launch
  The launch controls appear on the main screen before the rocket is launched and the values change as the rocket’s properties are altered by the program itself. This keeps the user in the loop, providing more feedback than just that of the rockets path itself. User input for these input boxes is handled by the playGame function so details on key press handling can be found in the associated section. The controls design is intuitive, with the up, down, left and right keys being utilized. Key presses were chosen as this could allow for a future in game feature of the rocket carrying fuel that could be used to speed it up, slow it down or alter its trajectory slightly upon a key press.
- drawRocket(struct rocket_struct rocket)
  Features – Provides a rocket to travel from Earth to Mars
  Extended features – Bitmap handling
  Draw rocket is simplistic in nature, it just uses GFX’s bitmap handling function to draw a bitmap of a rocket at the location stored under “x” and “y” and the rocket function. The coordinates are translated into integers because this is the only input the GFX library will accept as a coordinate.
- getRandom(int max_number, int min_number)
  A random number generator is needed for several features of this program, thus it made sense to create a single function to handle it. A single integer is returned, which is in-between the two integer inputs. This needs the math library to run, making it a dependency.
- moveInsideAoE(struct rocket_struct rocket, struct moon_struct moon)
  Features – Accurate gravity
  A resultant force is calculated using vector addition of the two sperate entities. These entities being a vector for movement outside the area of gravitational influence and a vector for inside it. This final vector is split into x and y resultants and applied to the coordinates currently stored in the rocket structure, which is accessed by the draw rocket function. The velocity is translated into KM/s as this made the game run at a

much more reasonable speed which, in turn, made the rocket 'fly' much more smoothly. This was because the rocket was travelling less distance per refresh.

- moveOutsideAoE(struct rocket_struct rocket)
  Features – Accurate gravity
  The rockets angle of launch is divided up into its x and y components using trigonometry. The hypotenuse is the launch angle, then sin and cos operations can be carried out to find the other side of the triangle. These values are then multiplied by the velocity and then converted to KM/s. The reasoning for the unit change is justified in moveOutsideAoE. The new coordinates can then be written to the structure rocket ready to be written to the screen.
- playGame(void)
  Features – Adjustable launch angle and velocity, scoring
  This function contains several loops and the bulk of the code that links various parts of the game together. Firstly, the arena is drawn and the first loop begins, allowing the user to input the desired velocity and launch angle using key presses. The loop waits for a hardware event, checks if it is valid and alters the settings accordingly. When enter is pressed this loop ends and the flight loop begins. This loop effectively flies the rocket. Collision detection is handled here by comparing the rockets current coordinates to a set of boundaries and if triggered the appropriate action is taken. The location of the rocket can also be checked using this method and the correct function can then be called to move the rocket appropriately. The only way this loop and therefore this function can be left, is if there is a collision. Scoring is handled by incrementing the score counter by one per execution of loop, if a collision is detected with anything other than Mars, this score is then set to 0 again.
- endGame(struct rocket_struct rocket)
  Features – Scoring
  Extended features – Replay functionality
  After the game is over the users score is displayed and two click boxes. The first box is replay and if clicked will call the play game function allowing the user to try again. The quit box will end the entire program and quit the user interface.
- struct rocket_struct rocket
  The rocket has several properties that several different functions need to access and occasionally alter, therefore a class is required. Several points are stored in this class.
  - x – The rocket's x coordinate.
  - y - The rocket's y coordinate.
  - v – Its velocity in KM/s.
  - a – The angle of launch.
  - m – The rocket's mass in KG.
  - d – The distance travelled by the rocket in KM for scoring.
  - f – The force of attraction acting on the rocket from the moon in N.
  - ar – The angle of travel in radians.

  The rocket structure started out as a location to easily store and retrieve coordinates, but during development it grew into a repository to find everything related to the rocket quickly and efficiently.

- struct moon_struct moon
  The Moon structure only has four values stored in it.
  - x – The x coordinate of the centre of the Moon.
  - y – The y coordinate of the centre of the Moon.
  - d – The diameter of the moon.
  - i – The radius of the area of influence of the Moon.

Flowchart of functions

For simplicity and efficiency the program has been split into three sections…

1. Setup

The setup is handled by the main function and runs all the code required to carry out repeated attempts at the game. This will run once.

2. Play game
   Play game is mainly associated with the playGame loop. It sets up and runs a single iteration of the game, right up until a score has been calculated.
3. End game
   The endgame function handles this section of the program. It can call playGame again or end the program.

The functions interact as shown in fig 1.

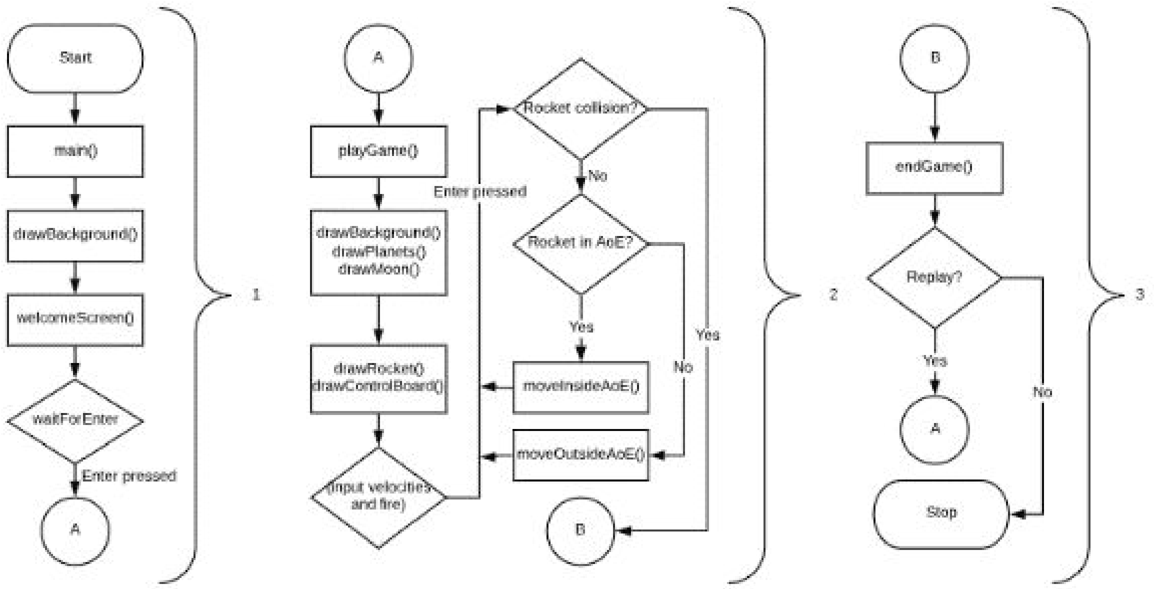


fig 1.

Extended features

Although extended features can be found listed under functions, structures and features, a full list can be found below.

- Moon resizing
  The resizing of the moon adds a new game dynamic, making the game more entertaining for a longer period of time, as the user now has many more different scenarios to attempt.
- Bitmap handling
  This was implemented to make the game more visually appealing.
- Replay function
  This is the most useful addition to the program. If the user likes the game and wants to play again, they will not have the task of needing to restart the application.

Inputs

The command line was not used for any inputs. This is to ensure the game can be ported over to devices with smaller screens that do not have space for both the game window and the touch screen.

Input	Location	Output	Justifcation
Press enter to continue	Welcome screen	Continues program to game screen	Implemented so users who know how to play can skip quickly
Velocity	Game screen	Alters value for rocket's velocity in rocket struct	Required by design brief
Launch Angle	Game screen	Alters value for rocket's travel angle in rocket struct	Required by design brief

Press enter to launch	Game screen	Triggers the "fly rocket" loop	Required by design brief
Replay	Score screen	Calls the playGame() function	Allows users to play again quickly, making for quickly accessible gameplay
Quit	Score screen	Ends the program in its entirety	All good programs need ways to quit.

Outputs

The only output to the user directly is the score, which is placed on the GFX window so it can be viewed easily. It can be seen on the post-game screen after the rocket has completed its journey as this is when it first becomes available.

User interaction

Once the program has been launched, the user will only interact with the GFX window, right up until they choose to quit the game. This is to ensure that all interaction is kept clearly in one place, removing the need for any command line interaction. All the controls are designed to be intuitive, but a small section of text can be found on the arena screen, pre-launch, which explains to the user how to operate the controls. This makes it completely clear how the game should be played.

Testing plan

Testing each step during development involved checking for syntax errors, functionality and then balance. Checking for syntax errors meant trying to build the program and checking the build log for errors. Secondly, checking for functionality meant testing all the newly implemented features to see if they behaved as expected. Finally, a balance check involves things that can be altered to change the 'feel' of the game, these can be anything from changes aesthetically to alterations to the strength of the moon's gravitational pull. This should not be confused with the final testing of the complete product, as this is only the testing required by iterative development. This type of testing is key to the success of a program built iteratively, as the previous step must be correctly implemented to allow for the development of the next step.

When the game is thought to be complete, it can be evaluated using the evaluation criteria and user testing. The evaluation criteria is as follows, some of the points have been taken from the design brief.

- Can the user select a desired velocity and launch angle?
- Does the rocket then travel in the chosen direction at the chosen velocity?
- Is the rocket accurately affected by the moon's gravitational pull as it would be mathematically?
- When the rocket lands on Mars is a score presented to the user that is inversely proportional to the distance travelled?
- Are all the extended features listed fully functional without error?
- Was user testing a success?

User testing would involve giving the game to a varied group of individuals, such as a young child, a teenager and an adult. They would then be allowed to play the game and provide feedback, which could be used to represent their demographic. If negative feedback outweighs positive feedback the game, in its current form, could be considered a failure.

Evaluation of implementation

Utilizing the iterative design process was a success because it meant that the aesthetics, balance of the game and gameplay could be adjusted during development, when the code was less complex and easier to work on, making adjusting the game a quick process. Aesthetically, the design of the game is excellent, with bitmap handling making the game appear more professional than the simplistic two-dimensional shapes that would have had to be used otherwise. which are reminiscent of Microsoft paint more than a successful mini-game.

The layout of the code and therefore the performance of the code are not fully optimized. However, they are not a failure either, there is just room for improvement. A notable issue is the structure rocket, which contains several variables, including two angles, one in degrees and one in radians. Both are not necessary, and this is the sort of issue some more optimization would fix. Despite these small issues the code runs smoothly, especially with the conversion from meters per second to kilometres per second, which means that the rocket moves less distance with each new position, making it appear less jumpy and more natural.

Evaluation criteria

Most of the evaluation criteria stated in the previous section has been met, with the acceptance of the following points…

1. Is the rocket accurately affected by the moon's gravitational pull as it would be mathematically?
   The rocket veers of course wildly when traversing the moon's gravitational area of influence. This is because of a miscalculation between finding the rocket's angle of travel towards the moon and altering it to adhere to the requirement of the program that the angle of travel must relate to the x axis.
2. When the rocket lands on Mars is a score presented to the user that is inversely proportional to the distance travelled?
   Number two is partially a success, the rocket moves correctly until it enters the moon's gravitational influence where its behaviour is unpredictable, but one quadrant is correct. This could not be fixed due to the iterative development process, as this issue was uncovered in the last design step, which left no time to rectify it before the deadline.

Limitations, potential and unsolved problems

- One limitation is the wide array of input options used. On one screen the mouse is needed, the next the arrow keys are. This is not a massive limitation as such, it just removes from the refined feel of the game.
- The game itself is not resizable, this is because all of the objects, such as the bitmaps, appear at fixed locations inside a fixed size window.
- Potentially the moon could be too large to allow the rocket to reach Mars however this is not testable without a fully functional program.
- The most serious issue is the moons gravity, it does not work fully and without it the game does not fully function. This is examined in point two of the evaluation criteria.

Conclusion on evaluation

So far, the game is a success, it is visually appealing and the parts that work, do so without fail. However, it is not complete. The gravity issue needs to be corrected before the game is playable. When this is done it will work well, but not offer a lot of depth to gameplay, more features will need to be implemented to keep users entertained.

Ideas for future improvements

- The rocket's bitmap representation should be able to turn with the direction of travel.
- The rocket could have an accurate fuel counter, some of which burns off at launch, with the amount depending on the launch velocity and the rest left available to make small course and velocity corrections.
- High scores could be saved to the device and tracked, allowing users to compete for better scores.
- The speed at which the rocket hits Mars could be monitored and if the user approaches to fast, they could crash land.

Appendix A

Exam No: Y3857545

Bitmaps and other assets

/Bitmaps/Background.bmp

/Bitmaps/Earth.bmp

/Bitmaps/Mars.bmp

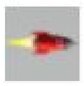
/Bitmaps/Moon'SIZE'.bmp

Moon sizes available 75, 100, 125, 150 and 175 pixels. All images are the same, they just differ in quality.

/Bitmaps/Quit.bmp

/Bitmaps/Replay.bmp

Bitmaps/Rocket.bmp



Exam No: Y3857545

Header files

- stdio.h
- stdlib.h
- math.h
- graphics_lib.h

main.c – The source code

```c
/*Include the generic C libaries for loops, pointers ect*/
#include <stdio.h>
#include <stdlib.h>

/*Include the C maths libaries*/
#include <math.h>

/*Contains the graphical package*/
#include <graphics_lib.h>

/*Defines the moon structure for stopring details related to the moon*/
struct moon_struct
{

    /*
    x - x coodinate of centre of moon
    y - y coordinate of centre of moon
    d - diameter of moon
    i - radius of area of gravitational infuence
    */

    int x;
    int y;
    int d;
    int i;

};

/*Defines the rocket struct for stopring details related to the rocket*/
struct rocket_struct
{

    /*
    x - x coordinate of the rockets current location
    y - y coordinate of the rockets current location
    v - the rocket's current velocity
    a - the rockets launch angle as chosen by the user
    m - the mass of the rocket
    d - the distance travelled by the rocket
    f - the force of attraction acting on the rocket
    ar - the launch angle in radians
    */

    double x;
    double y;
    double v;
    double a;
    double m;
    double d;
    double f;
    double ar;

};

/*Sets up the GFX window and runs the first game*/
int main(void)
{

    /*Open 640x480 GFX window*/
    GFX_InitWindow(1280, 720);

    /*Initiate text, event queue and bitmap handling
    The order of this cannot be changed*/
    GFX_InitFont();

    /*Register mouse and keyboard events*/
    GFX_CreateEventQueue();
    GFX_InitMouse();
    GFX_RegisterMouseEvents();
    GFX_InitKeyboard();
    GFX_RegisterKeyboardEvents();

    /*Initiate bitmap handling*/
    GFX_InitBitmap();


    /*Call to activate random number generator*/
    srand (time(NULL));

    /*Draw the background*/
    drawBackground();

    /*Show the welcome screen text*/
    welcomeScreen();

    /*Write these changes to the screen*/
    GFX_UpdateDisplay();

    /*Wait for the user to press enter, as displayed by the welcome screen
    This is useful for creating in program breakpoints*/
    waitForEnter();

    /*Run the main game loop, this calls the endgame function post gameplay*/
    playGame();

    /*Quit the whole program, the 0 represents a succsessful execution*/
    return 0;

}

/*Presents the user with a logo/press key to continue screen*/
void welcomeScreen(void)
{
    /*Draw the title on screen*/
    GFX_DrawText(590, 130, "Interstellar");

    /*Draw the how to play instructions*/
    GFX_DrawText(540, 200, "You must travel to Mars");
    GFX_DrawText(470, 250, "This means selecting an appropriate velocity");
    GFX_DrawText(420, 300, "and angle of launch, remember to factor in the effect");
    GFX_DrawText(550, 350, "of the moons gravity");

    /*Draw the continue prompt onscreen*/
    GFX_DrawText(540, 480, "Press enter to continue...");

    /*Leave the function*/
    return;

}

/*Waits for the user to press enter, the returns nothing*/
void waitForEnter(void)
{

    /*Create an local int variable called keyPressed to store which key the user presses*/
    int keyPressed;

    /*Create a while loop to run while done = 0*/
    while (true)
    {

        /* Wait for an keyboard or mouse event*/
        GFX_WaitForEvent();

        /*Check the event queue to see if the event registered is a key press*/
        if (GFX_IsEventKeyDown())
        {

            /*If the event was a key press get the allegro assigned integer for that key*/
            GFX_GetKeyPress(&keyPressed);

            /*Check to see if the key pressed is 67, as assigned by allegro*/
            if (keyPressed == ALLEGRO_KEY_ENTER)
            {

                /*If it is enter, return to main()*/
                return;

            }

        }

    }

}

/*Calculate random properties for the moon - stored in struct moon, needs the moon
structure*/
void calculateMoon(struct moon_struct * moon)
{
    /*Get random numbers for x, y and r for the creation of the moon*/
    moon->x = GFX_RandNumber(350, 930);
    moon->y = GFX_RandNumber(350, 370);

    /*Create an local int variable to store the randomly generated diameter option*/
    int diameterOption;

    /*Get a random bumber between 1 and 5 to choose between the 5 .bmp files for the 5
different moon sizes*/
    diameterOption = GFX_RandNumber(1, 5);

    /*Translate the randomly generated diameter option */
    switch (diameterOption)
    {

        /*
        This whole switch statement follows the same outline as below
        Moon diameters are...

        Ran Int - Diameter

        1 - 75
        2 - 100
        3 - 125
        4 - 150
        5 - 175
        */

        /*If the randomly generated number is 1*/
        case 1:

            /*Set the diameter in the class to 75*/
            moon->d = 75;

            /*Leave the switch statement*/
            break;

        case 2:
            moon->d = 100;
            break;

        case 3:
            moon->d = 125;
            break;

        case 4:
            moon->d = 150;
            break;

        case 5:
            moon->d = 175;
            break;

    }
    /*Calculate and store the moon's area of influence's radius - the diameter is useless
in calculation*/
    moon->i = ((moon->d /2) * 5);

    /*Return*/
    return;

}

/*Draws the Earth, Mars and the Moon, needs the moon structure*/
void drawPlanets(struct moon_struct * moon)
{

    /*Bitmaps all follow the same structure, as outlined on the earth bitmap,
    Individual detals can be found abouve each decleration statement
    Bitmap's must have clear backgrounds as background management was not implemented*/

    /*Identifier - Size in pixels - Coordinates*/
    /*Earth - 300x300 - 130,130*/

    /*BITMAP - typedef for bitmaps
      earthBitmap - variable name of bitmap
      GFX_LoadBitmap - call GFX librarys bitmap loading function
      which takes the file path from the parent directory where
      the code blocks project is stored
    */
    BITMAP earthBitmap = GFX_LoadBitmap("Bitmaps/Earth.bmp");

    /*Use the GFX_DrawBitmap function to draw the bitmap - has to be loaded prior to this
    DrawBitmap takes the arguments "Bitmap variable name" and the coordinates where the
centre of the bitmap
    will be drawn as comma seperated variables*/
    GFX_DrawBitmap(earthBitmap, 130, 130);

    /*Free up some memory by "unloading" the bitmap, takes the variable name of the loaded
bitmap as an argument*/
    GFX_FreeBitmap(earthBitmap);

    /*Mars - 150x150 @ 590,450*/
    BITMAP marsBitmap = GFX_LoadBitmap("Bitmaps/Mars.bmp");
    GFX_DrawBitmap(marsBitmap, 1200, 640);
    GFX_FreeBitmap(marsBitmap);

    /*Position the moon at the randomly generated cordinates
    The diameter is checked against the size of the bitmaps
    And then the correctly sized bitmap is implemented
    Following the same layout as above*/
    if (moon->d == 75)
    {

        /*Moon - 75x75 @ moon->x, moon->y*/
        BITMAP moonBitmap = GFX_LoadBitmap("Bitmaps/Moon75.bmp");
        GFX_DrawBitmap(moonBitmap, moon->x, moon->y);
        GFX_FreeBitmap(moonBitmap);

    }
    if (moon->d == 100)
    {

        /*Moon - 100x100 @ moon->x, moon->y*/
        BITMAP moonBitmap = GFX_LoadBitmap("Bitmaps/Moon100.bmp");
        GFX_DrawBitmap(moonBitmap, moon->x, moon->y);
        GFX_FreeBitmap(moonBitmap);

    }

    if (moon->d == 125)
    {

        /*Moon - 125x125 @ moon->x, moon->y*/
        BITMAP moonBitmap = GFX_LoadBitmap("Bitmaps/Moon125.bmp");
        GFX_DrawBitmap(moonBitmap, moon->x, moon->y);
        GFX_FreeBitmap(moonBitmap);

    }

    if (moon->d == 150)
    {

        /*Moon - 150x150 @ moon->x, moon->y*/
        BITMAP moonBitmap = GFX_LoadBitmap("Bitmaps/Moon150.bmp");
        GFX_DrawBitmap(moonBitmap, moon->x, moon->y);
        GFX_FreeBitmap(moonBitmap);

    }

    if (moon->d == 175)
    {

        /*Moon - 175x175 @ moon->x, moon->y*/
        BITMAP moonBitmap = GFX_LoadBitmap("Bitmaps/Moon175.bmp");
        GFX_DrawBitmap(moonBitmap, moon->x, moon->y);
        GFX_FreeBitmap(moonBitmap);

    }

    /*Set the GFX libaries pen colour of white - integer value 11*/
    GFX_SetColour(11);

    /*Draw a circle, colour specified above, to represent the moons maximum area of
influence*/
    GFX_DrawCircle(moon->x, moon->y, moon->d/2*5, 5);

    GFX_DrawText(20, 500, "Use the arrow keys");
    GFX_DrawText(20, 520, "to control the ");
    GFX_DrawText(20, 540, "launch parameters");
    GFX_DrawText(20, 560, "Press enter to launch...");

    /*Draw the angle promt at the top of the screen*/
    GFX_DrawLine(200, 20, 300, 20, 2);
    GFX_DrawLine(200, 20, 290, 60, 2);
    GFX_DrawArc(200, 20, 40, 0, 27, 2);
    /*Leave the function*/
    return;

}

/*Draws the background on the GFX screen*/
void drawBackground(void)
{

    /*Background BITMAP*/
    /*For more information on the drawing of bitmaps see drawPlaets()*/
    BITMAP backgroundBitmap = GFX_LoadBitmap("Bitmaps/Background.bmp");
    GFX_DrawBitmap(backgroundBitmap, 640, 360);
    GFX_FreeBitmap(backgroundBitmap);

    /*Leave the function*/
    return;
}

/*Draw the users current velocity and launch angle, needs the rocket structure*/
void drawControlBoard(struct rocket_struct * rocket)
{

    /*Create an array to store the rockets current angle of travel
    and after that, the velocity*/
    char text[20];

    /*Convert the rockets angle of travel into a string from a double
    and store in the string "text".
    This has to be done as the angle is stored as a double and cannot
    be printed to the GFX screen as text.*/
    sprintf(text, "%lf", rocket->a);

    /*Draw a black rectange to provide a background for the data, this improves
readability
    and prevents clashes with the moons AoE ring - also white*/
    GFX_DrawFilledRectangle(20,655,320,700,0);

    /*Draw the text shown below at the sppecified coordinates*/
    GFX_DrawText(25, 660, "Launch Angle in degrees:");

    /*Draw the contents of the string "text"*/
    GFX_DrawText(230, 660, text);

    /*Convert the double (rockets velocity) into a string and store in "text"*/
    sprintf(text, "%lf", rocket->v);

    /*Draw the contents of the string "text"*/
    GFX_DrawText(230, 680, text);

    /*Draw the text shown below at the sppecified coordinates*/
    GFX_DrawText(25, 680, "Velocity in KM/s:");

    /*Leave the function*/
    return;

}
```

```c
/*Draws the rocket using the location stored in the rocket struct, needs rocket passing
too it*/
void drawRocket(struct rocket_struct * rocket)
{

    /*Convert the double values of the rockets location to int so
    they are compatable with the GFX lib's bitmap writer*/
    int mapX = rocket->x;
    int mapY = rocket->y;

    /*WRite the rocket to the screen - more details on bitmap writing can be found in
drawPlanets()*/
    BITMAP rocketBitmap = GFX_LoadBitmap("Bitmaps/Rocket.bmp");
    GFX_DrawBitmap(rocketBitmap, mapX, mapY);
    GFX_FreeBitmap(rocketBitmap);

    /*Leave the function*/
    return;
}

/*Creates and stores a random integer, needs a max and min number passing to it*/
int getRandom(int max_number, int min_number)
{

    /*Declare the local variable to store the random number*/
    int random;

    /*Calculate the random number using rand()*/
    random = rand() % (max_number + 1 - min_number) + min_number;

    /*Return the newly calculates the random number, ending the function*/
    return random;

}

/*Move the rocket inside the moons ring of gravitational influence*/
void moveInsideAoE(struct rocket_struct * rocket, struct moon_struct * moon)
{

    /*A resultant force is calculated from the previous force acting on the rocket and the
current force being applied to it by the moon*/

    /*Create variables to process the x and y legnths of the triangle's used for
calculating vectors
    and create variables to store the new angle and velocity being applied by the moon*/
    double x, y, angle, velocity;

    /*Calculate the force acting on the rocket and add it to the previous force*/
    /*pow runs with compiler and does not play well with variables*/
    /*v = a t(1) and a=f/m*/

    /*Calculate the distance as x and y resultants between the rocket and the moon*/
    x = moon->x - rocket->x;
    y = moon->y - rocket->y;
    /*Calculate the force acting on the rocket, created by the moon using f =
(G*m1*m2)/r^2*/
    rocket->f = (6.67408 * pow(10.0, -11.0) * rocket->m * 7.34767309 * pow(10.0, 8.0) /
sqrt(x*x+y*y));

    /*Calculate the velocity which needs to be added because of the moon using v = at with
t as 1 as one loop is 1 second in game time*/
    velocity = rocket->f / rocket->m;

    /*Detect if the rocket is to the left of the moon*/
    if (moon->x > rocket->x)
    {

        /*Calculate the resultant angle between the rocket and the moon*/
        angle = atan(y/x);

        /*Detect if the rocket is above of the moon*/
        if (rocket->y < moon->y)
        {

            /*Alter the angle acordingly*/
            angle = angle + 3.14159265359;

        }

        /*Detect if the rocket is below of the moon*/
        if (rocket->y > moon->y)
        {

            /*Alter the angle acordingly*/
            angle = 3.14159265359/2 - angle;

        }

    }

    /*Detect if the rocket is to the right of the moon*/
    if (moon->x < rocket->x)
    {

         /*Calculate the resultant angle between the rocket and the moon*/
        angle = atan(x/y);

        /*Detect if the rocket is above of the moon*/
        if (rocket->y < moon->y)
        {

            /*Alter the angle acordingly*/
            angle = 3.14159265359*1.5 - angle;

        }

        /*Detect if the rocket is below of the moon*/
        if (rocket->y > moon->y)
        {
```

Exam No: Y3857545

```
            /*The angle does not need altering, however this is here incase the maths is
wrong as the gravity is erratic*/
            angle = angle;

        }

    }

    /*Calculate the resultant vector x/y components and move the rocket accordingly*/
    rocket->x += (cos(angle) * velocity) + (cos(rocket->ar) * rocket->v) * pow(10, -3);
    rocket->y += (sin(angle) * velocity) + (sin(rocket->ar) * rocket->v) * pow(10, -3);

    /*Increase the rockets velocity due to the effect of the moon*/
    rocket->v += velocity;

    /*Calculate and store the rockets ne angle of travel*/
    rocket->ar = (angle + rocket->a) /2;

    /*Return to the play game loop*/
    return;

}

/*Move the rocket for 1 second when it is outside the moon's area of infuence*/
void moveOutsideAoE(struct rocket_struct * rocket)
{

    /*If the problem is split up into a triangle, the hypotonuse represents the resultant
movement (rockets velocity).
    Therefore, the other two sides can be used to represent the x and y components of the
rockets movement.
    Using simple trig the "legnth" of the other two sides represnts the rockets x and y
velocities.
    These are multiplied by 10^-3 to make them KM/S rather than M/S
    This method of execution means that one cycle of the games main while loop is
equivilant to 1 second in game time*/

    /*Calculate the rockets movement along the x axis and write it to the rocket struct*/
    rocket->x += cos(rocket->ar) * rocket->v * pow(10, -3);

    /*Calculate the rockets movement along the y axis and write it to the rocket struct*/
    rocket->y += sin(rocket->ar) * rocket->v * pow(10, -3);

    /*Return to the function that called this one*/
    return;

}

/*Runs the game, including mathmatical processing of rocket path*/
void playGame(void)
{
    /*Declare the structure moon and rocket from the global template definition*/
    struct moon_struct moon;
    struct rocket_struct rocket;

    /*Wipe any previous games/the welcome screen off the GFX window*/
    GFX_ClearWindow();
    /*Calculate a new moon position game as struict changes do not show till leaving 
function */
    calculateMoon(&moon);

    GFX_ClearWindow();
    drawBackground();

    /*Draw the newly calculated game*/
    drawPlanets(&moon);

    /*Set the rockets initial values*/
    rocket.x = 200.0;
    rocket.y = 200.0;
    rocket.m  = 30000;
    rocket.d = 0;
    rocket.v = 3000;
    rocket.a = 20;
    rocket.f =0;
    rocket.d = 0;

    /*Draw the rocket in the starting position*/
    drawRocket(&rocket);

    /*Draw the rocket's control board with the set values stored in struct rocket*/
    drawControlBoard(&rocket);

    /*Write to the screen*/
    GFX_UpdateDisplay();

    /*Set fire to 0, so the loop below will run until this is changed*/
    int fire = 0;

    /*Create an local int variable called keyPressed to store which key the user presses*/
    int keyPressed;

    /*Run the loop while fire is equal to 0*/
    while (!fire)
    {

        /* Wait for an keyboard or mouse event*/
        GFX_WaitForEvent();

        /*Check the event queue to see if the event registered is a key press*/
        if (GFX_IsEventKeyDown())
        {

            /*If the event was a key press get the allegro assigned integer for that key*/
            GFX_GetKeyPress(&keyPressed);

            /*Check if the key pressed was the up arrow*/
            if (keyPressed == ALLEGRO_KEY_UP)
            {

                /*Check if the angle displayed currently is greater than 90 deg*/
                if (rocket.a < 90.0)
                {
```

```
                /*Add 5 degrees to the angle*/
                rocket.a += 5;

                }

            }

            /*Check if the key pressed was the down arrow*/
            if (keyPressed == ALLEGRO_KEY_DOWN)
            {

                /*Check if the angle displayed currently is less than -90 deg*/
                if (rocket.a > -90.0)
                {

                    /*Take 5 degrees from the angle*/
                    rocket.a -= 5;

                    }

            }

            /*Check if the key pressed was the enter key*/
            if (keyPressed == ALLEGRO_KEY_ENTER)
            {

                /*Make fire = 1, so the loop ends and the rocket "fires"*/
                fire = 1;

            }

            /*Check if the key pressed was the left arrow*/
            if (keyPressed == ALLEGRO_KEY_LEFT)
            {

                /*Take 100 km/s from the displayed velocity*/
                rocket.v -= 100;

            }

            /*Check if the key pressed was the right arrow*/
            if (keyPressed == ALLEGRO_KEY_RIGHT)
            {

                /*Add 100 km/s to the velocity*/
                rocket.v += 100;

            }

        }

        /*Clear the window to prevent flicker*/
        GFX_ClearWindow();

        /*Draw an entirely new game screen with updated locations*/
        drawBackground();
        drawPlanets(&moon);
        drawRocket(&rocket);
        drawControlBoard(&rocket);

        /*Write changes to the GFX window*/
        GFX_UpdateDisplay();

    }

    /*Check if the users chosen velocity is above the escape velocity*/
    if (rocket.v < 200)
    {

        /*Set the distance travelled to -1 to trigger the excape velocity message*/
        rocket.d = -1;

        /*Run the endgame loop*/
        endGame(&rocket);

    }

    /*Convert the launch angle to radians for processing*/
    rocket.ar  = rocket.a / 180 * 3.14159;

    /*Create variables to store the distance from the moon and mars*/
    double distFromMoon, distFromMars;

    /*While fire = 1 run the loop to fly the rocket*/
    while (fire == 1)
    {

        /*Add 1 to the rockets distance travelled to use as a score*/
        rocket.d += 1;

        /*Use pythag to calculate the distance between the rocket and the moon/mars
        pow(x,y) has not been used as it is calulated when the program is complied,
meaning
        the value of it does not change if the variables inside do*/
        distFromMoon = sqrt((moon.x - rocket.x)*(moon.x - rocket.x)+(moon.y -
rocket.y)*(moon.y - rocket.y));
        distFromMars = sqrt((1200 - rocket.x)*(1200 - rocket.x)+(640 - rocket.y)*(640 -
rocket.y));

        /*Check if the rocket is inside the moons area of gravitational influence (AoE
from here on)*/
        if (distFromMoon <= moon.i)
        {

            /*Call the function to move the rocket along its path inside the AoE*/
            moveInsideAoE(&rocket, &moon);

        }

        /*Check if the rocket has hit mars and landed*/
        if (distFromMars <= 75)
        {
            /*Set fire to 0, ending the fire loop*/
            fire = 0;

        }

        /*Check if the rocket has hit the moon*/
        if (distFromMoon <= moon.d/2)
        {

            /*Set the score to 0, as the user did not make it to mars*/
            rocket.d = 0;

            /*Set fire to 0, ending the fire loop*/
            fire = 0;

        }

        /*Check if the rocket has left the screen*/
        if (rocket.x < 0 || rocket.x > 1280 || rocket.y < 0 || rocket.y > 720)
        {

            /*Set the score to 0, as the user did not make it to mars*/
            rocket.d = 0;

            /*Set fire to 0, ending the fire loop*/
            fire = 0;

        }

        /*If the roockets position does not meet any of the above parameters, it needs to
be moved normally*/
        else
        {

            /*Move forward under with no influence from the moon*/
            moveOutsideAoE(&rocket);

        }

        /*Clear the window so the old rocket position cannot be seen*/
        GFX_ClearWindow();

        /*Draw an entirely new game screen with updated locations*/
        drawBackground();
        drawPlanets(&moon);
        drawRocket(&rocket);
        drawControlBoard(&rocket);

        /*Write changes to the GFX window*/
        GFX_UpdateDisplay();

    }

    /*Run the endgame function, as the game has ended, passing the rocket struct so a
score can be displayed*/
    endGame(&rocket);
    /*Should endGame fail for some reason the program will still return and quit*/
    return;

}

/*Shows the user their score post game and offers replay/quit*/
void endGame(struct rocket_struct * rocket)
{

    /*Post game window wipe*/
    GFX_ClearWindow();

    /*Draw the background*/
    drawBackground();

    /*Create an array to store the score*/
    char text[20];

    /*Convert the score into a string from a double
    and store in the string "text".
    This has to be done as the score is stored as a double and cannot
    be printed to the GFX screen as text.
    The score is the distanced travelled/amount of loops performed*/
    sprintf(text, "%lf", rocket->d);

    /*Check if the user did not aceive escape velocity, checked in playGame()*/
    if (rocket->d == -1)
    {

        /*Notify the user of their mistake*/
        GFX_DrawText(500, 210, "You did not acheive escape velocity");

    }

    /*Run if the user acheived escape velocity*/
    else
    {

        /*Show the user their score*/
        GFX_DrawText(560, 220, "You're score was...");
        GFX_DrawText(590, 230, text);

    }

    /*Draw the replay button at 640,180 details on bitmap drawing can be found in
drawPlanets()*/
    BITMAP replayBitmap = GFX_LoadBitmap("Bitmaps/Replay.bmp");
    GFX_DrawBitmap(replayBitmap, 640, 180);
    GFX_FreeBitmap(replayBitmap);

    /*Draw the replay button at 640,380 details on bitmap drawing can be found in
drawPlanets()*/
    BITMAP quitBitmap = GFX_LoadBitmap("Bitmaps/Quit.bmp");
    GFX_DrawBitmap(quitBitmap, 640, 380);
    GFX_FreeBitmap(quitBitmap);

    /*Update the display*/
```

```c
    GFX_UpdateDisplay();

    /*Create local variables to end the users loop when they make a valid choice, x click 
coordinate and y click coordinate respectively*/
    int choice,x,y;

    /*Set choice to 0 so the loop will run*/
    choice = 0;

    /*Run the following while loop when choice is not equal to 0*/
    while (!choice)
    {

        /*Wait for a mouse/keyboard event to be added to the event queue*/
        GFX_WaitForEvent();

        /*Check if the last event in the queue was a mouse button click, if it was not 
listen for a new event*/
        if (GFX_IsEventMouseButton())
        {

            /*Get the coordinates of the mouse click as integers stored as x and y*/
            GFX_GetMouseCoordinates(&x, &y);

            /*Check if the mouse click is within the x bounds of the buttons*/
            if (x >= 565 && x <= 715)
            {

                /*Check if the click is within the y bounds of the replay button button*/
                if (y >= 152 && y <= 208)
                {

                /*Replay the game by running the playGame() function*/
                playGame();

                }

                /*Check if the click is within the y bounds of the quit button*/
                else if (y >= 352 && y <= 408)
                {

                    /*Close the window and leave the IF statement tree*/
                    GFX_CloseWindow();

                }

            }

        }

    }

    /*Should the program not choose the replay option, it will run this and end the 
program (Failsafe close)*/
    return;

}
```