SUPER DANK SPACE ADVENTURE

Eric Smith

Erik Juarez

Perry Huynh

Vincente Lara

Contents

1. REQUIREMENTS

1.1 GitHub

The project will be contained in a Github repository (Master branch), and the instructor will have the URL and access to view or clone it.

1.2 Source Files

Each student will have a C++ source file named correctly as described in lab-6. The source file should have a minimum of 500 lines of your own code. Code copy and pasted from a framework does not count as your own code. A header file may also be included.

1.3 Makefile

The project will have a Makefile that can be used to build your project on the local computers in Room 315. The makefile must compile and link the student files for each student in your group.

1.4 Other

For the development of this game, we opted to use the K&R Indent Style. For our software development process, we used a mix of agile and waterfall. The majority of the functional and nonfunctional requirements are explored in the next section, design.

2. DESIGN

The design of the game was inspired from games such as Agar.io and Katamari Forever, where the player's goal is to traverse around the area and collide with other objects and gain as much mass as they can. The basic idea of the game is to collide with other objects and gain as much mass as you can, while avoiding larger objects.

In order to implement this, we found it best to use and modify the Asteroids framework, to make the game similar to Agar.io, but using X11, OpenGL, and OpenAL in C++.

When gaining mass, we decided to take into account the size of the absorbed object, so absorbing something significantly smaller than you would amount to little to no increase in size while absorbing something nearly as big as you would amount to a relatively large increase in size. We also opted to use an inverse exponential function when increasing the player's score and then using the score in a logarithmic function to increase the player's size, rather than simply incrementing by one. We also wanted the size of the enemy objects that are spawned throughout the game to be relative to the player's current size. The size of object would also affect its own velocity.

One feature that we definitely needed to implement was a sort of spawn protection, so that as the enemy objects are spawned, they do not spawn directly on top of you and immediately kill you.

We wanted to modify the camera, having it so there was a sort of invisible "boundary" around the character, so that when the boundary was passed, the camera would slowly pan towards the direction that the player was moving, simulating a larger area.
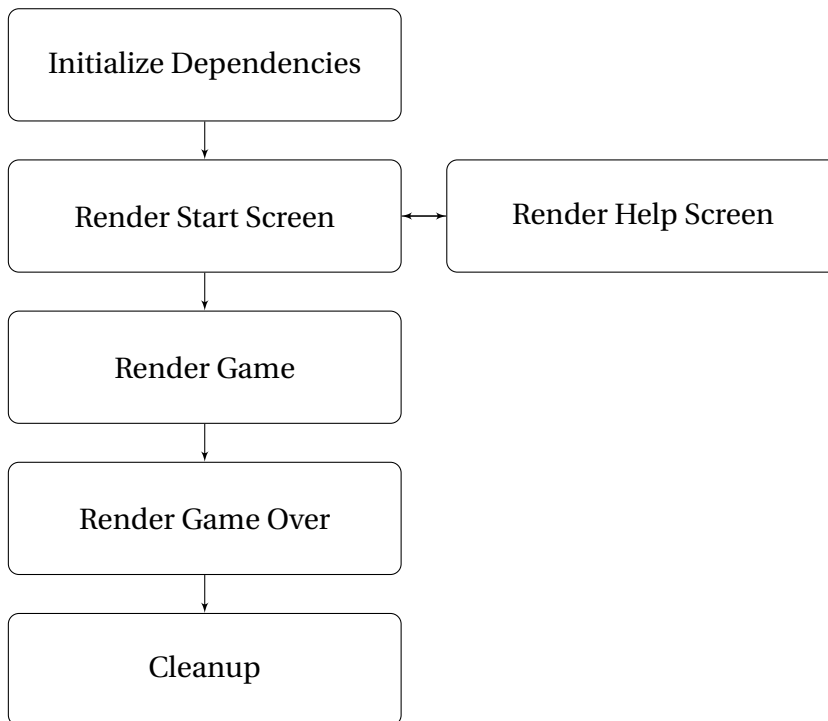
As for the nonfunctional requirements of the game, we also wanted to add features such as sounds and texture in order to make it look more interesting than using static colors. We also wanted to add the option to restart the game once the player loses, so they would not have to close the program and run it again.

3. IMPLEMENTATION

3.1 General

Game States

The game ran inside a loop within the main and followed the general data flow diagram below:

```
┌──────────────────────────┐
│  Initialize Dependencies │
└──────────────────────────┘
              │
              ▼
┌──────────────────────┐        ┌──────────────────────┐
│  Render Start Screen │ ◄────► │  Render Help Screen  │
└──────────────────────┘        └──────────────────────┘
              │
              ▼
┌──────────────────────┐
│     Render Game      │
└──────────────────────┘
              │
              ▼
┌──────────────────────┐
│   Render Game Over   │
└──────────────────────┘
              │
              ▼
┌──────────────────────┐
│       Cleanup        │
└──────────────────────┘
```

After initializing XWindows, OpenGL, OpenAL, and Textures, the game enters a loop where it it would check for key presses and renders the game's current state. The game initially renders the start screen, which can switch to a Help screen given the Player pressed the 'h' key. After pressing the 'Enter' key to start the game, the loop enters the in-game state and continually calculates and update the positions of every object in the game and renders them. Upon colliding with a larger object, the player is brought to the game over screen. After exiting the program, everything is cleaned up.

4

Restart

After the game is over, the player has the option to press the 'r' key in order to restart the game without having to close the game and run the program again.

3.2 Rendering

All rendering was done through various shapes drawn through OpenGL inside a window created by X11. Fonts were rendered using the portion of the framework relating to fonts found in Homework 1 and Lab 2.

Player and enemy objects are rendered using OpenGL's *GL_TRIANGLE_FAN*.

```
glBegin(GL_TRIANGLE_FAN);
float x = (float)g.ship.radius * cos(499 * PI / 180.f);
float y = (float)g.ship.radius * sin(499 * PI / 180.f);

for (int i = 0; i <= 500; i++)
{
    glVertex2f(x, y);
    x = (float)g.ship.radius * cos(i * PI / 180.f);
    y = (float)g.ship.radius * sin(i * PI / 180.f);
}
glEnd();
```

Above is a portion of the code that was used to render the circles before applying textures.

3.3 Textures

Conversion of Images

Since the game utilizes a wide variety of textures, having a mass amount of *.ppm* files was not very feasible, so it was better to store them as *.png* files and temporarily convert them to *.ppm* files and remove them once we are done with them. Hard coding all of the system commands for conversion and deletion would have been very tedious, inefficient, and inflexible so we used a loop with *<dirent.h>* and iterated through the directory containing the images and converted any files with the *.png* extension and converted them to **.ppm* files.

```c
struct dirent *entry;
DIR *dp = opendir("./images/");

if (dp)
{
    while ((entry = readdir(dp)) != NULL)
    {
        if (strstr(entry->d_name, ".png"))
        {
            char t1[256], t2[256];
            char t3[256] = "images/";
            strcpy(t2, entry->d_name);
            int slen = strlen(t2);
            t2[slen - 4] = '\0';
            strcat(t2, ".ppm");
            sprintf(t1, "convert images/%s images/%s", entry->d_name, t2);
            system(t1);
            strcat(t3, t2);
            strcpy(g.imageTemp[g.nt++], t3);
        }
    }
    closedir(dp);
}
```

Above is an a portion of the source code that was used to iterate through the directory and load any *.png* files and convert them to *.ppm* files.

Loading the Textures

The next step in adding textures to the game was to load the converted *.ppm* files and to associate them via OpenGL to their respective objects. The issue with using *<dirent.h>* to load the files was that the order of the files converted was random, so iterating through the array to load certain textures would not work. To solve this issue, the loading of single textures such as the background could be hard coded, but for textures of enemy, we found it was better to utilize a loop and *strcat()* to load the enemy textures, rather than hard code a similar block of code ten times. At the end of the program, we iterate through the array containing the paths for the temporary *.ppm* files and remove them.

```cpp
for (int i = 0; i < 6; ++i)
{
    std::stringstream tempStr;
    tempStr<<(i + 1);
    std::string str = tempStr.str();
    char t1[256] = "images/texture_enemy_";
    const char* t2 = str.c_str();
    const char* t3 = ".ppm";
    strcat(t1, t2);
    strcat(t1, t3);

    ppm6CleanupImage(ppm);
    ppm = ppm6GetImage(t1);
}
```

Above is an a portion of the source code that was used to iterate through the directory and load the textures for the enemy objects.

Rendering the Textures

The Start Screen, Help Screen, Game Over Screen, and Background were rendered by applying a texture to an area of the screen drawn using *GL_QUADS*.

The textures for the player and enemy objects were rendered by simply applying a texture at the same time the points on the circle are calculated, which produced a "swirl" effect.

3.4 Movement

Player Movement

The general implementation for movement in the game was a modified version of the movement function(s) from the Asteroids framework. The player moves towards the location of the mouse cursor on the screen, rather than using the arrow keys. In order to do this we had to add *PointerMotionMask* to the events to check for in the *initXWindows()* function.

```
static int savex = 0;
static int savey = 0;

if (savex != e->xbutton.x || savey != e->xbutton.y)
{
    //Mouse moved
    savex = e->xbutton.x;
    savey = e->xbutton.y;

    int y = yres - e->xbutton.y;
    float dx = savex - g.ship.pos[0];
    float dy = y - g.ship.pos[1];
    float len = sqrt(dx * dx + dy * dy);

    g.ship.vel[0] = dx / len;
    g.ship.vel[1] = dy / len;
    normalize(g.ship.vel);
    shipRadiusSpeed();
    normalize(g.ship.vel);
    return;
}
```

Above is a portion of the source code that was used to have the player move towards the location of the mouse cursor.

Velocity

The speed of an object (player or asteroid) is dependent on the radius of the object. First we initialize each object with a random vector speed. We came up with a math function that took

the radius and scaled it down to an appropriate speed (decreasing speed as the radius grows larger).

```
Flt radiusVel = ( ( 500.0 / ( g.ship.radius + 100.0 ) ) − 2.0 )
```

The first problem encountered with the velocity is that the speed is calculated by adding the squares of the *x* and *y* coordinates of the vector. We take the ratio of the *x* and *y* coordinates of the vector and compare them to the magnitude of that speed. The ratio of the coordinate to the total speed is the same regardless of what you scale it up or down to. We then multiply the ratio of *x-to-speed* and *y-to-speed* by the calculated radius velocity and we get the result of the same directional movement with a properly scaled speed.

```
Flt radiusVel = ( ( 500.0 / ( g.ship.radius + 100.0 ) ) − 2.0 );

if ( radiusVel < 0 )
{
    radiusVel = 0.01;
}

Flt xVel = g.ship.vel[0];
Flt yVel = g.ship.vel[1];
Flt speed = sqrt( ( xVel * xVel ) + ( yVel * yVel ) );
Flt xRatio = ( xVel / speed ) * radiusVel;
Flt yRatio = ( yVel / speed ) * radiusVel;

g.ship.vel[0] = xRatio;
g.ship.vel[1] = yRatio;
```

Above is an a portion of the source code that was used to calculate the player's current velocity.

3.5 Growth

For the player's growth, we decided to use an inverse logarithmic function when increasing the player's score, in order to have it so absorbing a small object amount to little gain in size, and absorbing an object relatively close to the player's size results in a high gain in size.

$$y = \frac{1}{x}$$
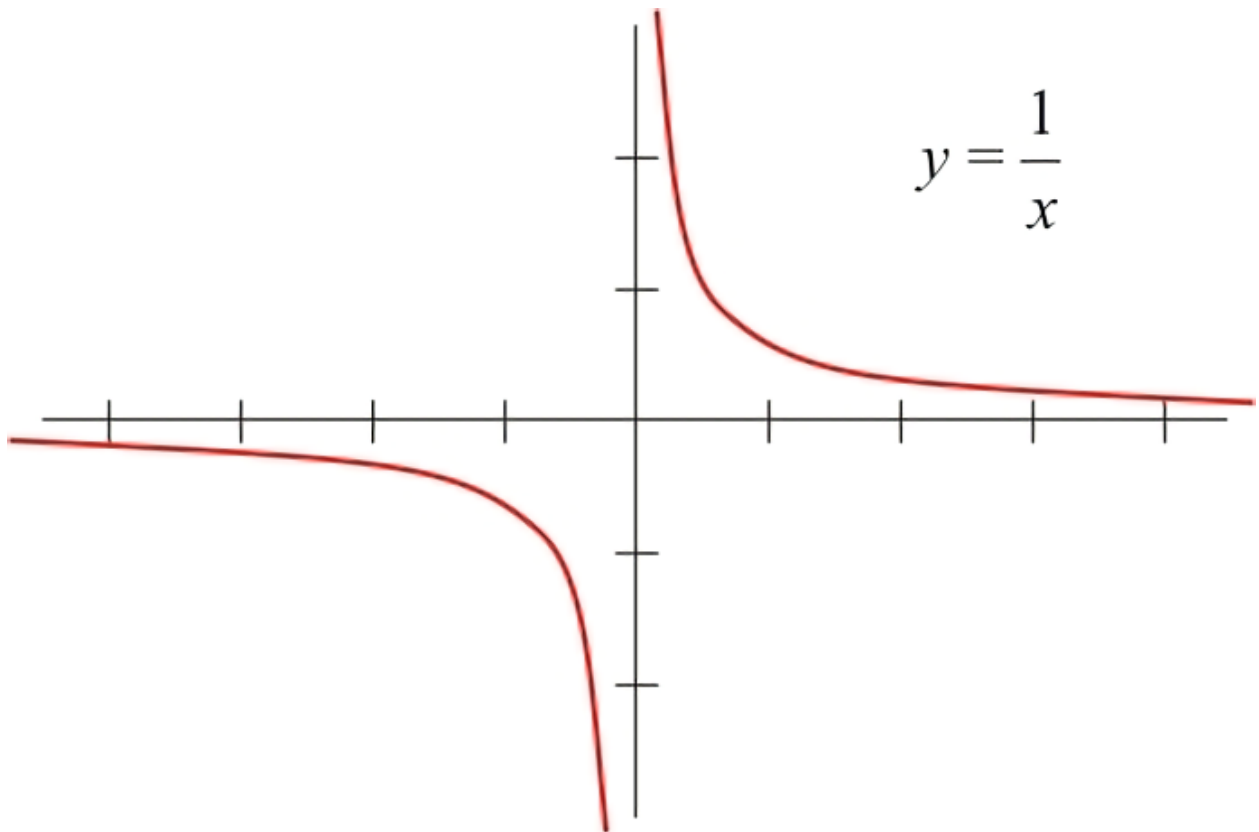
The graph is a relatively close representation of the mathematical principle of an inverse relationship between the differential of radius to growth in radius.

3.6 Camera

Our idea for camera implementation was inspired by the presentation of a game during class. Our first thought was to change the position of the player while having the camera follow them. This ended up being more difficult than we anticipated. we decided to implemented a camera that had the world move around our character.

When our players velocity changes, we subtract that velocity from the objects within our world. This creates the illusion that our players character is moving throughout the world. To make it more apparent that our player is moving, we implemented a dead zone in the middle of the screen.

When the player is within a certain box in the middle of the screen, the camera is static and the player is changing position on the screen. When the player reaches the edges of the box, the camera shifts from static to dynamic and follows the character around the world. To do this, we created boundaries within the world.

If the player's position is within these boundaries, then we change the position of the ship. If the player's position is outside of these boundaries, then we keep the players position constant and change the position of the asteroids.

3.7 Spawning of Enemy Objects

All of the enemy objects were created as nodes in a linked list, each initialized with a random radius that is scaled to be challenging, but not impossible. The random radius is calculated through a random number generator and scaled with the player's current radius and constants that give it a range of values that give the player slightly more asteroids smaller than them, but still enough large ones that need avoiding. Prior to adding the textures, the asteroids were colored depending on their size compared to the player, but after adding the textures, the player has to decide whether the asteroid is small enough to eat, or a game ender.

```
Asteroid *a = new Asteroid;
a->nverts = 8;
a->radius = (rnd() * 2.0 * g.ship.radius) - (rnd() * 0.5 * g.ship.radius);
a->textureId = rand() % 6;
```

Above is a portion of the code used when adding an enemy object. The object is initialized with a size relative to the player's current size, and given a random texture value.

3.8 Spawn Protection

In order to provide some protections to the player, we came up with idea of having an "invisible box" around the player that prevents other objects from spawning inside, making it so a large object cannot simply spawn right on top of the player immediately causing them to lose the game.

```
Flt tmpx = xres >> 2;
Flt tmpy = yres >> 2;

if ( a->pos[0] <= ( 3.0 * tmpx ) &&
     a->pos[0] >= tmpx &&
     a->pos[1] <= ( 3.0 * tmpy ) &&
     a->pos[1] >= tmpy)
{
    if ((a->pos[0] <= (2.0 * tmpx)) && (a->pos[0] > tmpx))
    {
        a->pos[0] -= 1.25 * tmpx;
        a->pos[2] = 0.0f;
    }
    else
    {
        a->pos[0] += 1.25 * tmpx;
        a->pos[2] = 0.0f;
    }
}
```

Above is a portion of the source code used in adding another object to check whether the object will be within the boundary around the player, and if it is move it.

3.9 Sounds

We were initially using the FMOD library for the sounds for our game, but then we found out that we were not allowed to use it, so we had to switch to OpenAL.

The sounds for the game were then reimplemented using OpenAL and the OpenAL Utility Toolkit. The sound files came from Nintendo's Kirby series. There are various sounds and looping songs for different events in the game, including the start screen, leaving the start screen, the main game screen, collision with objects, and losing the game.

3.10 Score

The player's score is based on the amount of mass they have gained since the game started. The score is equivalent to the player object's current radius, which increases relative to the size of the object they absorb. The score is rendered on the screen using the portion of the framework relating to fonts found in Homework 1 and Lab 2, using *fonts.h* and *libggfonts.a*

4. TESTING

The methods of testing utilized in the project included both white-box and black-box testing. For white-box testing, the area being tested was usually delegated to the contributor to that portion of the code, unless they requested help from others as each group member mainly focused on their portion, rather than extensively learn about all aspects of each others' code. Black-box testing was generally done by all members of the group. A majority of the debugging was done through print statements scattered throughout sections of a code to see where an issue might be occurring. We were attempting to determine whether our functions were receiving the proper values of variables and if they were being affected in unintentional ways. Another reason for these output statements was to get hard data from the game objects so we could use that data to assist in our scaling to make the game grow and progress in a smooth manner.

A practice that greatly helped our group throughout the coding and testing process was pair programming. Having another set of eyes parse through the code while the main developer was coding was extremely helpful in spotting any issues that arose during coding and debugging.

Testing was also done by compiling and running the game on various different computers and Linux distributions, such as Ubuntu and Arch Linux. Some issues arose when accessing the OpenAL library files with some of the Linux distributions we tested because they resided in a different directory, so the *Makefile* had to altered to successfully compile. We also found an interesting bugs with certain Linux distrubutions such as Ubuntu, where using the the computer's function keys to change the volume would result in a segfault. Another bug that took us a long time to find was the growth of the player upon absorbing another object. We

5. DELIVERY/MAINTENANCE

Our group decided on using one combined presentation, with each group member talking about their respective contribution to the project. Each member provided a separate document containing their phase of the technical report and contribution for the implementation section, where it was then combined and formatted using LaTeX. Maintenance was not in mind for this game as it was a project for a class so future features would not be implemented and bugs that may arise would not be fixed unless someone wanted to continually work on the game.

6. SCREENSHOTS

6.1 Start Screen
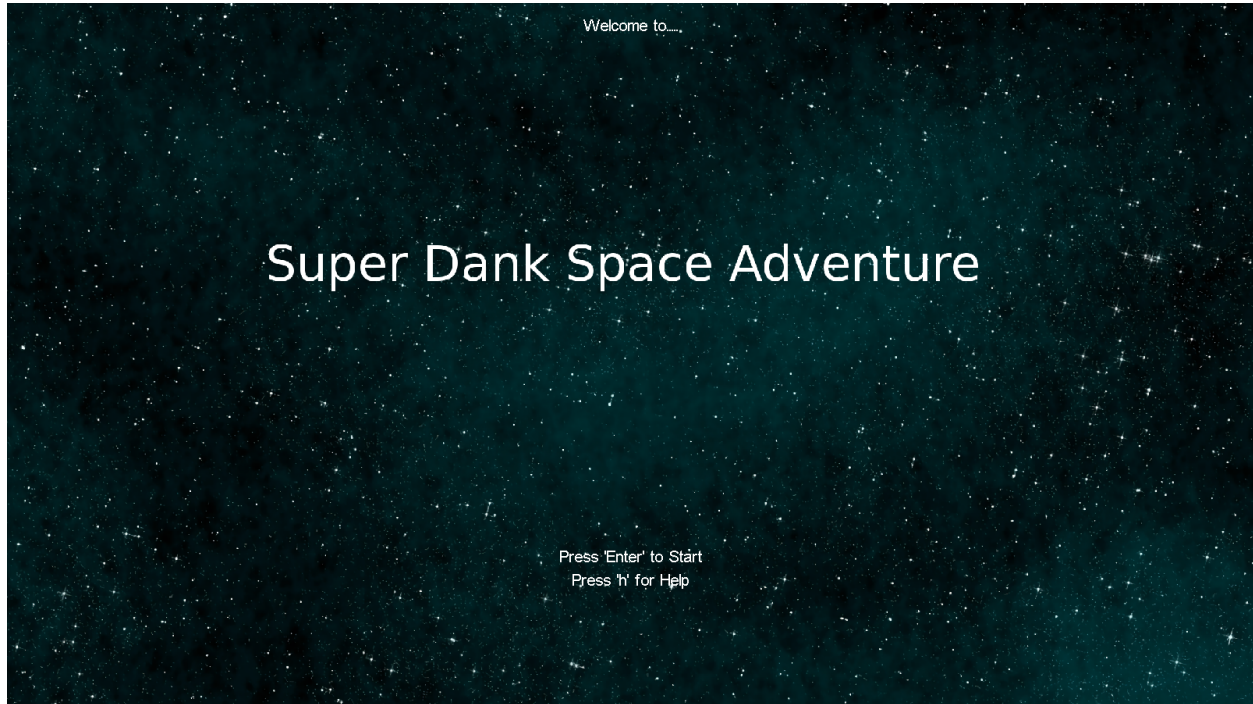
6.2 Help Screen
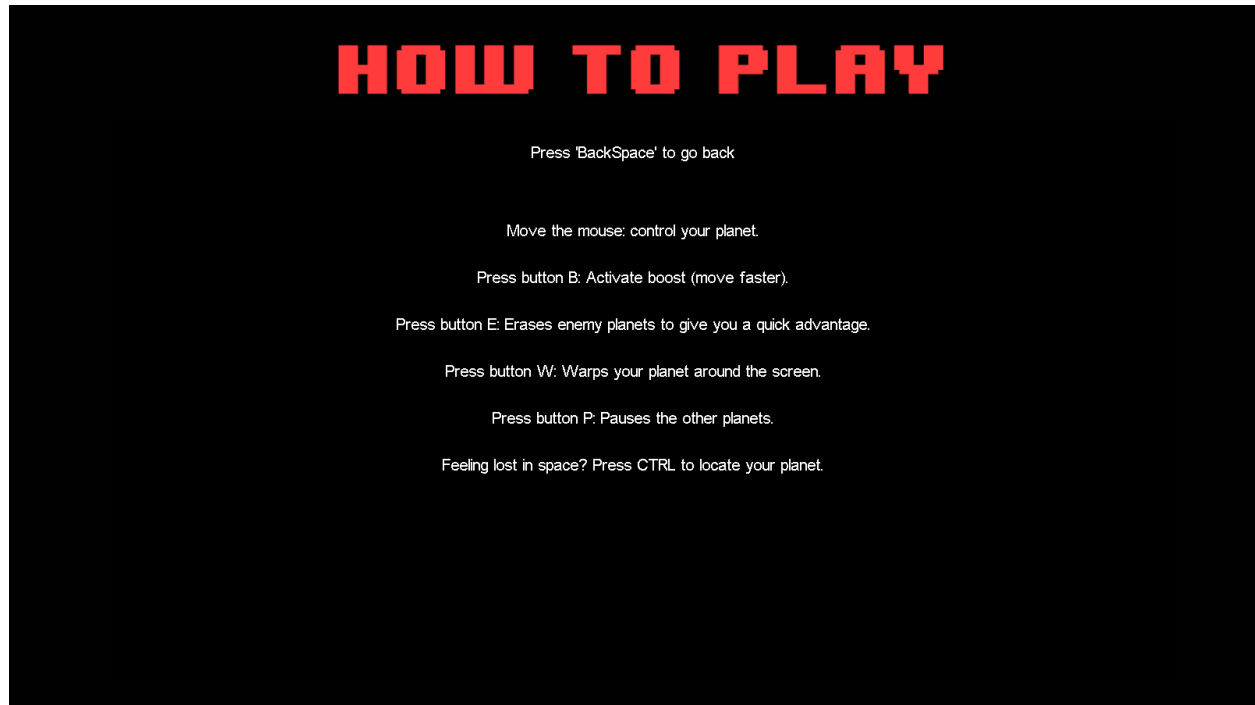
6.3 Main Game Screen

6.4 Game Over Screen