Report — CS4303 Practical 4

140013112

November 30, 2018

1 Overview

For this practical we were asked to implement our own game idea of a chosen genre. I chose to create the game "Moonage Daydream" of the genre space-platformer/causual-game. The main technical challenges to this game was implementing smooth gravity-physics, improving Processing performance and tuning the overall theme to fit the casual-game genre.

In this game, the player controls an astronaut travelling between planets in a spacecraft. The player spreads vegetation and life while defeating monsters and collecting stars. The game was successful in its aims and all challenges were solved, even defeating the limitations of the `processing` library (as discussed later in this report). Testing the game on various subjects contributed to the game at various stages of development.

Ideas which were dropped from the original pitch include the "inner rooms" inside planets which would have been entered through mario-like tubes which would require a whole new user-interface as well as a new gravity system and procedurally generated levels, for which there was no time. Also the shapes of planets was limited to spheres, to vastly simplify gravity rules. A small change was made in how grass is generated, as I found it cleaner and easier to generate grass on planet once for the whole planet rather than moving over it. The game name was changed from 'Major Tom' after user feedback revealed how overused this name was in modern culture.

1.1 Genre Context

1.1.1 Space-Platformer

The first and more obvious genre of this game was the theme of space-platformer. The features of classic platformer games which have been reused by me include jump-and-run controls, the 2-D profile view, the scrolling movement (keeping the player in the middle of the screen) and the jumping combat. It relates to other space-flight simulation games as it features gravity, a spaceship and classic space-like controls of forward and turning only.

Although the theme of space-platformer is not its own genre by itself, many games of this kind exist and have been very popular, most notably "Super Mario Galaxy" which was most noted for its graphics, gravity mechanics, soundtrack and setting. I used these key features as the main aims for my game (excluding soundtrack due to time-constraints).

1.1.2 Casual Game

Casual Games are designed to be picked up easily, allowing for potentially short bursts of play time, meant to be a relaxing and fun pastime. Features include a high reward to time ratio, bold colours to excite the senses and easy controls. A good soundtrack as well as frequent events and autosaving levels also corresponds to this genre.

Taking time-constraints into account, I chose to stick to a colourful design, intuitive controls and a self-explaining and easy game.

2 Aims

As mentioned in the context-overview, my aims for this game was developing an easy, fun and colourful game with pleasant graphics and controls. The game is not designed to be a real challenge but draw the player into the setting.

To satisfy these aims, I found that to create a pleasant experience, smoothness was critical. Smoothness includes both instantaneous response to user inputs as well as a fast update rate. Most time was spent fine-tuning gravity mechanics, camera control and improving game performance to keep the frame rate above 24fps which is the minimum human tolerance to perceive motion continuously. Also key-hints are displayed for non-obvious actions possible (like spawning grass or entering and exiting the spacecraft).

The number in the top left of the screen indicates the current frame rate of the game.

The graphics used for this game were taken from the internet and modified to fit a colour-theme (i.e. all shades of red in the different objects are the same red). The font used is meant to be playful and fun. Player stats in the top left are intentionally big and non-technical.

2.1 In-game Objectives and Rules

The in-game objectives of the player are kept very simple. The idea is to explore the constellations of planet-systems in leisure. Secondary objectives include not running out of air and finding all 3 stars per planet-system by jumping on the monsters.

Survival (i.e. not running out of air) is a secondary objective, as the player starts out with the ample amount of air of 300 (3 lives * 100 air). Air depletes very slowly and monsters only make you use 6 air which may be recollected. Collecting air on planets with grass is easy.

Reasons to grow grass on planets is solely to produce air for the player. I thought about letting the player earn badges for growing grass on all planets of a planet-system, but this was dropped to to time constraints. The reason to kill monsters is to stop them from running into you and making you loose air and also to find the 3 stars. The reason to find the 3 stars is to complete a planet-system and progress through the constellation.

I achieved to make this game truly infinite, as the background night-sky is procedurally generated and the game is effectively reset every time the satellite dish is activated. This means there is no way to win this game, there is only a sense of progress through the game.

3 Implementation

This game was implemented in Java using the Processing library. The IntelliJ IDE was used rather than the Processing IDE. This is the reason for `.java` file-endings. Also an instance of the `PApplet` must be passed to each object which wants to make use of Processing functions like drawing. And the game must be started through a `public static void main(){}` method. All drawable game elements have a `display()` method which draws the element when called.

3.1 Code Structure

The code files are divided into the packages `Main`, `Physics`, `UI` and `Elements`.

`Main` includes files related to the starting and running of the game, holding image resources and the background. The `Res.java` class loads all the images used in the game once on startup. This reduces the load of reloading images for each enemy and reduces the game memory footprint. The `Universe.java` class holds the game together and contains references to the current constellation, planet-system, the player objects and handles key inputs.

The `Physics` package was designed to hold code files relating to the gravity mechanics as well as the constellation and planet-system. The physics code for gravity was adapted from lecture slides and examples given on studres. The hierarchy of my universe is as follows: the universe holds the current constellation; the constellation holds a list of planet-systems, a planet system holds planets and all elements on the planets (monsters, air, etc). Whenever the player is travelling between planet-systems in free-space, the universe takes care of forces and displaying the spacecraft.

All other packages include exactly what they are intended to include. Notable is however the `GObject.java` class which is the superclass for all objects influenced by gravity. It takes care of displaying objects correctly, integrating forces and getters and setters for common variables.

4 Physics and Controls

4.1 Gravity

All game objects (except the spacecraft when it is in free-space) are attracted to the nearest planet where the distance is measured by taking the euclidian distance between the center of the object and the planet - the radius of the planet. For the force of gravity an inverse of the vector from the center of the object to the planet is used. This means far objects are less attracted than close objects.

Gravity is then integrated for all objects, together with other forces. The `integrate()` method in `GObject.java` takes care of this. Each object has a `floatFactor` which determines the effect of gravity. Bubbles have a negative float-factor, which inverses gravity. After applying the float-factor, gravity is added directly to the velocity. This approach follows code examples from studres closely.

The astronaut always has its feet pointing towards the nearest planet. This allows it to navigate and land upright. Also for the astronaut any velocities into directions other than the up/down direction of the nearest planet are dampened significantly, such that the astronaut may not go into orbit around planets but will always return to the planet.

All game objects may not come closer to a planet than its radius, thus always stay on the surface. A boolean `onPlanet` indicates if a game object is on the surface. Some extra conditions and checks were needed to allow the astronaut to leave the planet surface again, as initially due to rounding errors, sometimes it stuck to the surface, or left the planet when moving left or right quickly.

4.2 Camera Control

Extensive research into Processing had to be made until camera control code could be found online and condensed into a working `camera()` method. This method may set angle, zoom and translation of the camera. For this practical, only translation and zoom was used. Thus the camera always points at the playable object (astronaut or spacecraft), thus keeping them at the center of the screen. I experimented with always having the player upright and rotating the universe around them, but this ended up making my test subjects sea-sick.

4.3 Movement

For the astronaut – left and right movement works by adding 2 vectors as shown in figure 1 to move right and then down to stay on the surface of the planet. Up movement including jumping and jetpacking was implemented by adding vectors to the velocity, thus indirectly affecting the position. I chose not to use velocity for left/right movement as this gave the user more exact control. When jumping on monsters, using velocity would have meant users would need to guess how much force is needed to arrive at a desired location.

Changing the astronaut from jumping to jetpacking actually proved a challange as I needed to keep track on when the `w` key was released and repressed and when the astronaut came back to the planet surface.

For the spacecraft, only the up-movement adds to the velocity. Left and right turn the craft by incrementing or decrementing the heading.

5 Procedural Generation

5.1 Planet Systems

4 constellations were hard-coded into the `Constellation.java` class. These arrays of vectors encode the relative positions of planet-systems inside the constellation with x and y values between 0 and 12, and the first planet-system always at (0,0). The first planet-system must be around the origin such that spawning of the player works correctly. For real positions of planet-systems, relative positions are magnified by (2 * window_width). The first planet-system is also always the same and is created such that the planet the player initially lands on a planet without enemies.

All other planet-systems are generated as follows: As the available area to generate we take the zoomed-out window size. This is encoded in a `GenerationRegion` object. Each generationRegion may have 2 subregions, thus creating a binary tree like structure. Regions are alternately divided vertically and horizontally by level. I.e. to generate 5 planets: on the first level with 1 generationRegion and no sub regions, the field is divided vertically (shown by the green line in figure 2). Then we get to level 2. At this level, both subregions of the original region are divided horizontally (shown by blue lines in the figure). Lastly, at level 3, one region from level 3 is chosen at random and divided vertically again (shown by the red line). To finally generate the planets, pseudo-random

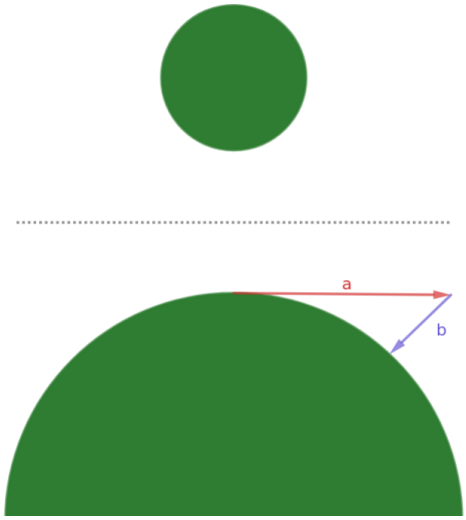

Figure 1: Schematic with vectors a and b showing vectors needed to move right and stay on the surface on the planet; the grey dashed line shows the divide where any objects would switch between the gravity of one planet to the other.

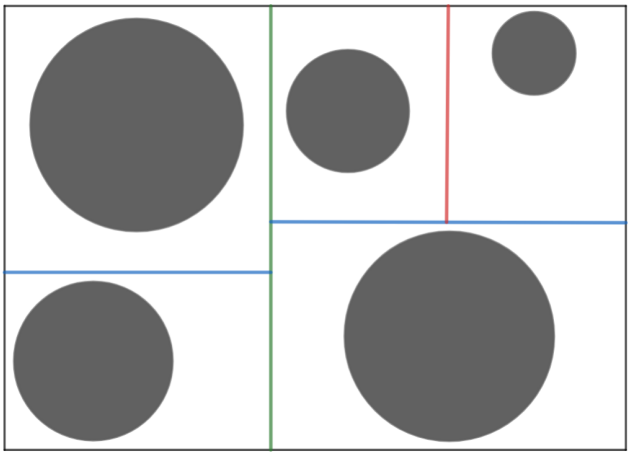

Figure 2: Schematic showing the procedural generation of a planet-system with 5 planets.

circle-centers are chosen inside the regions and pseudo-random radii are chosen to favour larger planets. Regions are also divided somewhat randomly, as the green line in the figure, for example, is not exactly in the center.

The number of enemies per planet corresponds to the planet's size. 3 enemies will always 'bear' a star.

5.2 Night Sky

The night sky background with all the stars is generated in the following way: the play area is divided into `SkyRegions` of 500x500 pixels with the x and y coordinates of the sky regions at multiples of 500. Such a grid pattern is shown in figure 3. The area of the current screen is shown in dark blue and labelled 'SCREEN'. This area moves around as the player travels around.

In the method `generateRegionsAndGetToDisplay()` the array `displayingRegions` holds all regions coloured in light blue – all regions on which `display()` needs to be called. The regions coloured in green (and light blue) are held in the `neededRegions` array. These regions will be generated if they do not yet exists. Generated regions will be held in the hashmap `generatedRegions`. In the figure, the regions on the left marked with '(x)' have not yet been generated, but will be if the screen moves left enough to enter one of the currently green marked regions such that there is always a buffer of 1 region to ungenerated regions.

Once the camera/screen moves away from a region, if the distance between the region and the edge of the screen

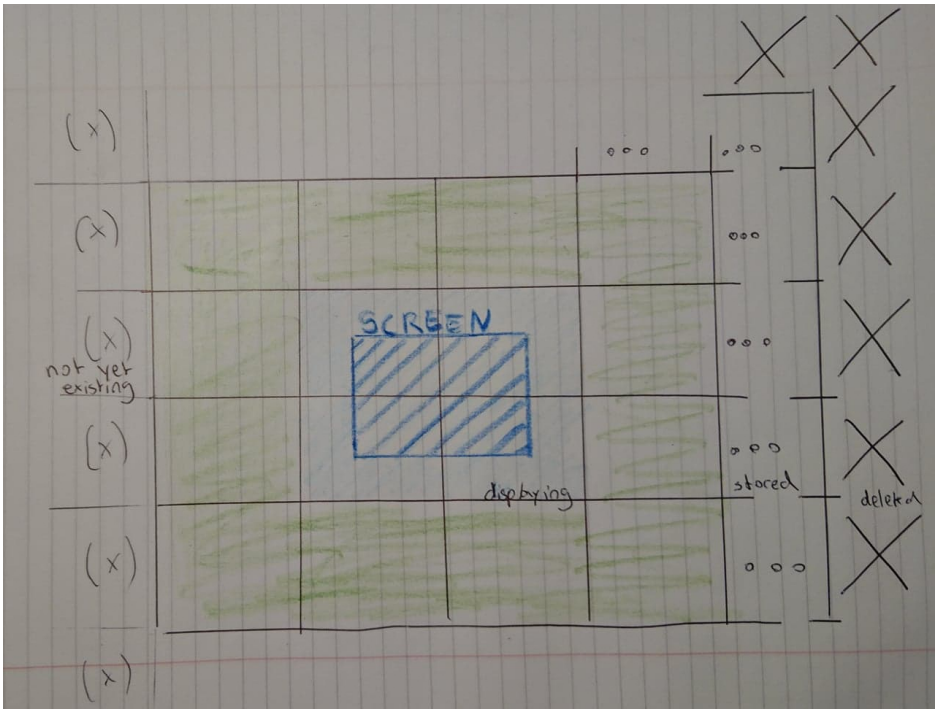

Figure 3: Schematic showing the procedural generation of the night sky.

is past a threshold, the region is deleted (marked by 'X' in the diagram). If the screen were to move back towards the deleted region, it would be regenerated completely fresh, and look different. This is however not noticeable to the user as the position of stars is not usually used as point of reference.

6 Testing

The game was tested on 1 test subject completely unfamiliar to computer science but familiar to casual gaming, as well as 2 fellow computer science students. The first test subject provided valuable feedback during the entire time of development on usability and graphics.

6.1 Issue 1

One of the major issues flagged during testing was smoothness and lagging of the game. Initially, the displaying of 300+ grass images and 1000+ stars slowed the game down quite significantly even on my high-end PC. It was observed that movement seemed sluggish and unresponsive, making the character difficult to control and the game pace slow and boring.

Steps were taken to address this issue to the best of my abilities as described in the next section. I tuned my physics and controls to feel best at a frame-rate of about 30fps, which was the new average on my PC after my improvements *until I implemented the final fix*. After these improvements, test-subjects enjoyed the game more than before.

6.2 Issue 2

Another issue raised during testing were the rather simple game objectives. Subjects were generally impressed by physics and robust design, but never played for more than 5 minutes, as once all objectives and actions were understood, nothing new felt like it needed exploring.

This issue concerned the overall design of the game and could not be addressed appropriately in the time given. I think the implementation of sub-levels inside planets as originally intended would have significantly helped in keeping user interest. Also a greater range of enemies and vegetation types (not just grass) could have helped.

One thing I added as a response to this was the progress bar at the bottom left. It shows the progress through the constellations and shows how many systems were completed (all 3 stars collected). One test subject confirmed that this added a sense of achievement to the player's efforts in the game.

7 Enhancing Game Performance

As flagged by the test-subjects, the Processing framework seems to have a lot of trouble displaying multiple images and shapes for each frame. I was already loading images from disk only once, thus reducing the most obvious source of lag. Research into the framework finally revealed the method `get()`, which allows the creation of a screenshot of part of the screen by copying the pixel values into a PImage object. However, the positions passed to get() are relative to the current window and only currently visible things may be screenshotted.

I used the get-method on the planets once all the grass is generated. To be able to screenshot the complete planet with all the grass, I first move the camera at zoomlevel 1 to center on the planet for 1 frame, effectively moving it into the center of the screen. This is perceived as a short flickering to the user. Then I screenshot the planet including all the grass into the variable `getImg`. Once this is complete and everything is moved back, I only display the getImg instead of all the pictures of grass.

To allow only 1 object to use `get()` per frame (as otherwise the camera gets completely messed up) I created a mutex-like boolean in the `MainGame` class accessed through the method `requestGetterMutex()`. Also the planet which may get this frame is displayed first by the planet-system, so that no other planets are in the background when screenshotting.

I saw dramatic increase in frame rate after implementing this for the planets. When I tried the get-method on the background night sky, I actually saw a decrease in frame rate. It seems that displaying 4-6 PImage objects is more work than displaying 1000+ points (stars are displayed as points of various sizes).

My last attempt at increasing the frame rate was implemented shortly before submitting the practical, thus could not be tested properly, however achieved to get the frame rate back to 60fps!!! I found a way to write directly to the processing pixels array instead of drawing points for all the stars. The method `displayStarAsPixel()` in `SkyRegion` class first gets the relative position of the star in relation to the current screen configuration, then checks if the whole star is within bounds of the screen, then writes the star's colour directly to the pixel location. If a star is of size 3, then an area of 3x3 pixels will be written to. Somehow stars seem bigger with this method and I did not have time to deal with the issue of zoom yet, which makes stars seem the same size at all zoom levels and may cause sea-sickness when travelling in the spacecraft a lot. Because my game was tuned to work best at a lower frame-rate, I ended up limiting the maximum frame rate to 40fps.

8 Conclusion

In conclusion, the implemented game is original, solves several technical challenges, makes use of content learned in this module and made use of testing. It provides a modern science-fiction take on a mixture of the genres - platformer, space-simulator and casual game. The game aims to immerse the player into a setting, through colourful graphics, carefully tuned physics, and procedurally generated worlds.

8.1 Possible Improvements and Known Issues

Given more time, inner levels would have extended the game substantially and added valuable and more diverse content to keep the player entertained. This game also does not make use of an AI or story. Adding a gentle background story could have added some juice to the game, even though game stories are not common with casual games. The monsters AIs are intentionally kept simple and 'stupid', but adding more intelligent bosses would have been nice.

Figure 5 in the appendix highlights a problem when using screenshots of planets (overlapping with other planets). This issue could not be resolved in the time provided, but maybe adding transparency to PImage objects or screenshotting 2 planets at once could have fixed that.

9 Appendix

Screenshots are provided to illustrate gameplay.

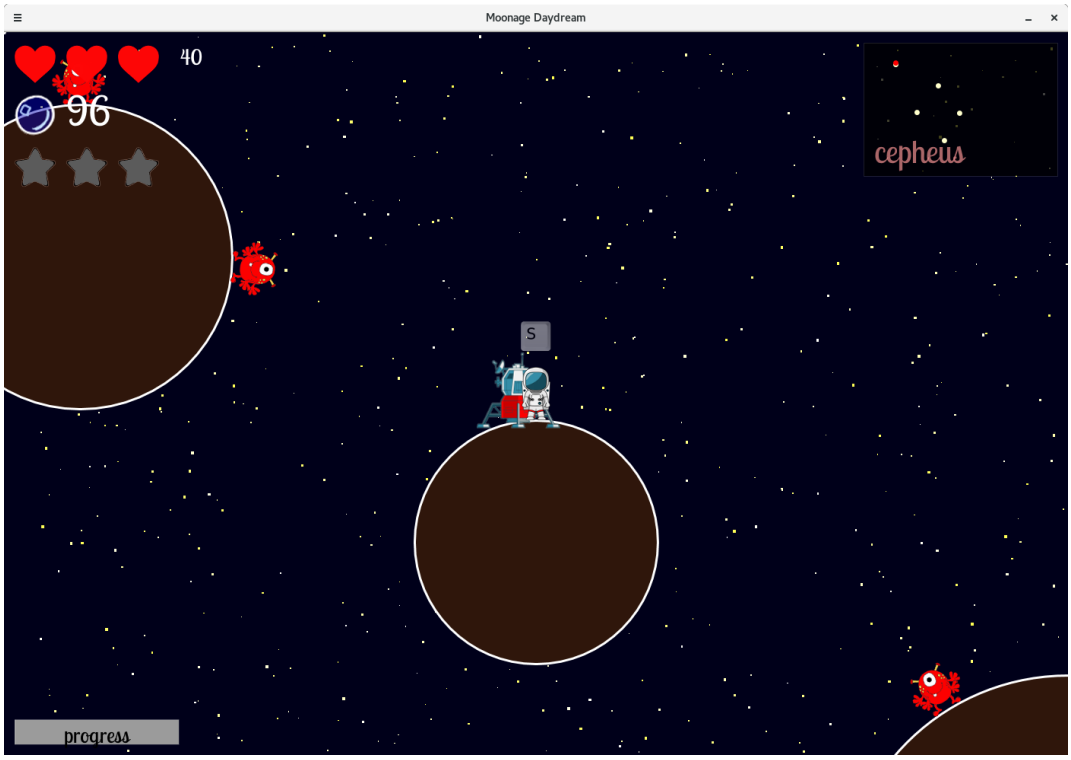

Figure 4: Initial Screen shown to player. Also showing a key-hint above the astronaut and monsters.

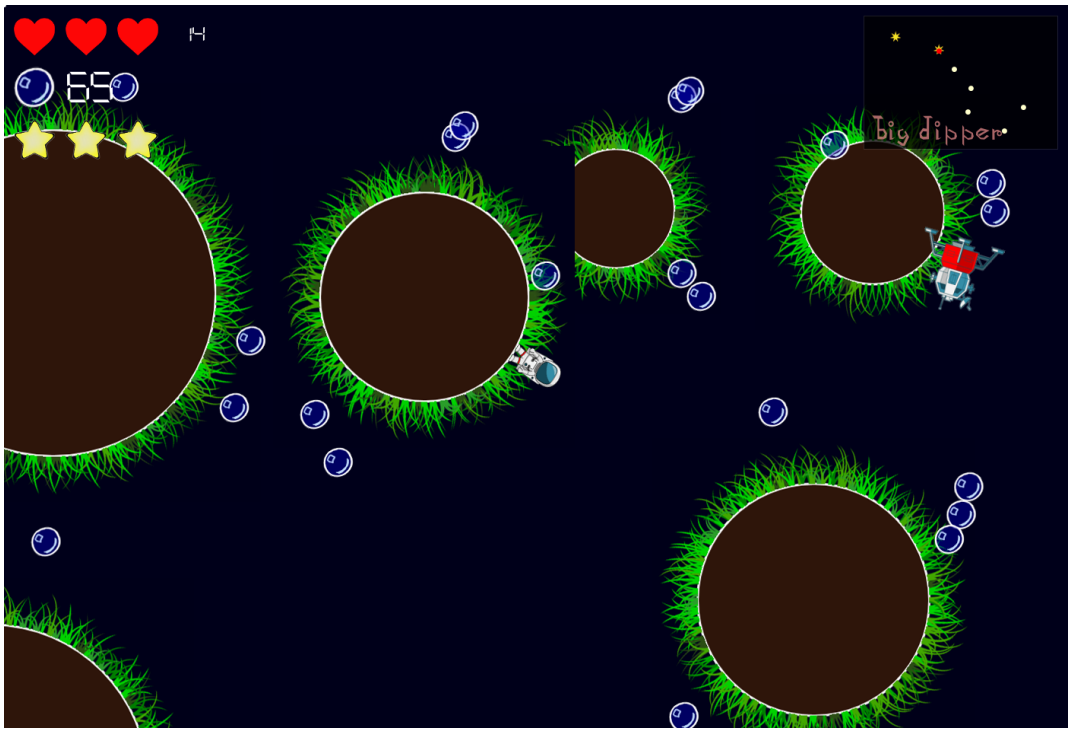

Figure 5: Screen showing an ugly side-effect to using screenshots of planets when planets are close together

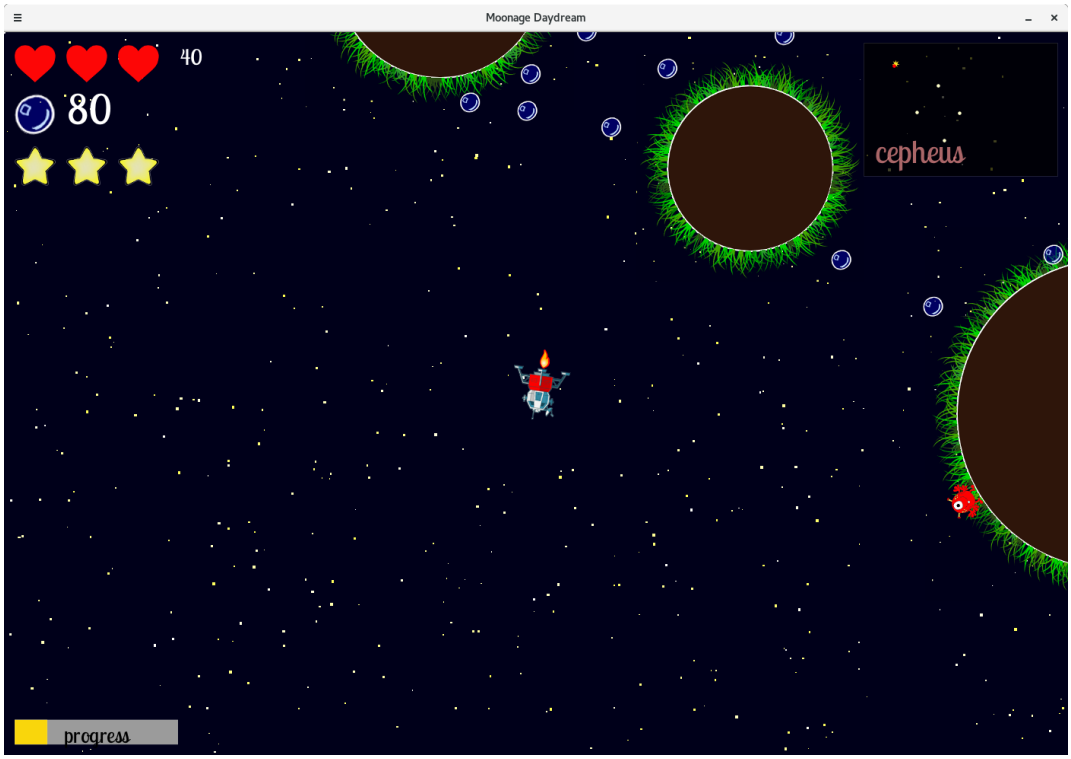

Figure 6: Screen showing the player flying the spacecraft after collecting all 3 stars in the first system and cepheus constellation.