Mars Lander

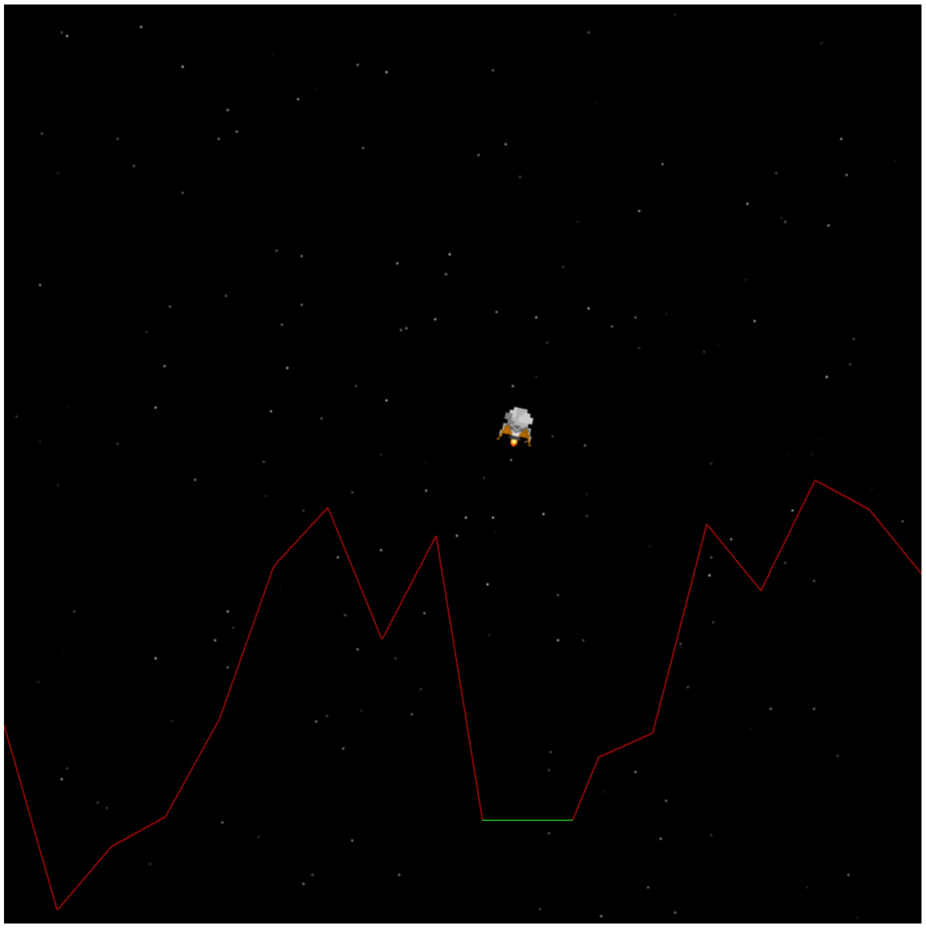

Figure 1: Mars Lander

Getting started

The goal of this lab is to build a 2D mars lander game.

In this game, you control a lander that you need to safely land in the surface of mars.

First of all, let's get started by setting up the project:

- Windows users: double click on the *start_env.bat* file. This opens a terminal in which you should run `gps -P mars_lander.gpr`
- Linux and Mac users: run a terminal and `cd` inside the *tp* directory. From here, run *source env.sh*. Then you can run `gps -P mars_lander.gpr`.

Question 1

Create some variables (`X`, `Y`, `Direction`, `Speed`). Initialize the values of the parameters with the following semantics:

- `X` and `Y`: The position of the object, can be initialized to `0`.
- `Direction`: The current direction of the object. This is an angle in degrees.
- `Speed`: Speed of the object in the given direction.

Question 2

Create an infinite loop that moves the object according to it's speed and direction. To be able to move the object in the given direction, you can transform the angle into a vector that represents the direction of the movement.

Given the angle `a` in radian, the formula is:

- `x = -sin(a)`
- `y = cos(a)`

The object should be drawn as a rectangle and should be rotated according to the given direction. You can use the `Draw_Rect` function in `Display.Basic` to do so.

Refine the values chosen in the previous question in order to have a smooth movement.

Question 3

Use `Get_Key_Status`, from `Display.Basic` to change the speed and direction of the object depending on which key is pressed:

- if **UP** is pressed, we want to set the speed to some value

- if **LEFT** or **RIGHT** is pressed, we want to change the direction of the object.

Question 4

Before going further, we should orgnanize our code using packages and types.

You might have noticed that the position and velocity are vectors. Thus, we want to make a vector package that defines:

- The type for a vector
- A function to add two vectors:

  ```
  function "+" (Left, Right : Vector_Type) return Vector_Type;
  ```

- A function to multiply a vector by some value:

  ```
  function "*" (Left : Vector_Type; Right : Float) return Vector_Type;
  ```

- A function that converts an angle to a vector:

  ```
  function From_Angle (Angle : Angle_Type) return Vector_Type;
  ```

Now we can create our package `Mars_Lander` that defines a private type for our mars lander. We can use a record here that contains the direction of our lander and vectors for its current position and velocity.

The lander type being private, you need to provide some way to update it. Try to find a good way to update the values without providing setters for every parameters. For example, instead of directly modifying the velocity of our lander, we want to set some variable to true, then if this variable is set to true, we want to compute the velocity of our lander depending on the value of the `Direction` angle. We should do this update in a `Step` procedure that is called at each iteration of our simulation, and updates the internal values of our lander.

In this package, you can also add a `Draw` procedure that should draw the lander on the screen.

Update your main program to use our newly created packages and procedures. You should not directly update the position, velocity or direction of the lander, but instead, call the procedures that you defined in `Mars_Lander`

Question 5

We are going to implement the gravity that applies to our lander. To do so, we need to add an acceleration vector to our lander type. We can implement the gravity as a constant acceleration applied to the object.

In terms of our simulation, an acceleration is a value that updates the velocity at each `Step` of our simulation.

Question 6

Changing directly the velocity of our object when hitting **UP** is not what we want. Instead we would like to simulate some propulsion towards the direction of our lander.

To do so, instead of modifying the velocity, we want to apply an acceleration to the object towards its current direction in the `Step` procedure.

Refine the different constants to make it looks good.

Question 7

Now that we have our mars lander, you can notice that we are able to modify it's direction without any constraint.

What we want here is to limit the angle in which the lander can change its direction. To achieve this, use a range type in Ada for the value of the angle and do not permit the angle to be updated further than the max and min values of this newly created type.

Since the lander type is private, this invariant should be easy to keep inside the `Mars_Lander` package.

Question 8

Now, we are going to implement some rules to our game. To win the game, we want to safely land in the surface of mars. To do so, we have some constraints:

- The lander should stop when it is in the ground.
- If the velocity of the lander is to high when hitting the ground, it crashes.
- If the angle of the lander is not close to zero when hitting the ground, it crashes.

For simplicity, to detect the collision between our lander and the ground, let's assume that our lander is a circle. Thus, the only thing we need to do is to check the length between the lander and the ground. There is a collision if the length is less than the radius of the circle.

Implement the rule of our game and print a message on the screen when the lander successfully landed, or crashed. You can use `Display.Basic.Draw_Text` to do so. To center the text in the middle of the screen, you can use `Get_Text_Size` that will return the width and heigh of the given text.

Question 9

Controlling a colored rectangle arround doesn't seem very appealing…

We want to add some textures to our game !

To display an image instead of our colored rectangle we will be using the following sprites:

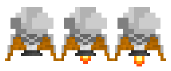

Figure 2: Mars lander sprites

To load an image, you can use the `Load_BMP` function inside the `Display.Image` package.

This returns an access to a two dimensional array type:

```
type RGBA_Array is array (Positive range <>, Positive range <>) of RGBA_T;
type Image_T is access RGBA_Array;
```

Where:

- The first dimension of the array is the **vertical** pixels.
- The second dimension of the array is the **horizontal** pixels.

Notice that the *mars_lander.bmp* image contains three different textures that we want to display depending on the state of the lander:

- The lander is falling without being propulsed, we want to display the first one
- The lander is being propulsed, we want to display the second one
- The lander has been propulsed for more that a fixed amount of simulation steps, we want to display the third one.

The heigh of the image is **22 pixels**.

Each individual sub-images are **20 pixels** large.

Your goal is to create a new `Image_T` object that only contains the desired pixels to be passed to `Draw_Image`. You can pass other values than `22` and `20` for the `Width` and `Height` for `Draw_Image`, the image is automatically scaled. The suggested values are 40 for the width and 44 for the height, but if you want, you can try other values.

Question 10

Let's draw a background with some stars.

Use the image *sky.bmp* for this.

To draw the background behind the lander, we can use the z coordinate passed to Draw_Image so that the background appears behind the lander. This works because even if we are drawing a 2d scene, the engine we are using is OpenGL which is a 3d engine. The z coordinate here corresponds the depth axis.

Question 11

Let's refactor the code to use tasks.

We want at least three different tasks:

- The main task (implemented directly in the main.adb file) is responsible of drawing everything on the screen
- A task that reads the inputs from the user.
- A task that computes each step of the simulation.

You will need to refactor the mars lander to use protected objects instead of regular types.