LUNAR LANDER GAME

By Andrew Yaros
CS171-069 - Winter 2017
Homework 5

USER MANUAL

This game is a simulation of a lunar landing.

Each time you play, a log will be kept of the game. When you first open the program, it will ask you what name you want to give the log. Type in a name and press enter.

The game will ask you to name your spacecraft. Type in a name and press enter. Next, you will be asked if you want to see the game's instructions. Press y / Y (for yes) or n / N (for no).

Now the game will start. For each second of your decent, you must decide how much fuel will be burned. Type a number between 0 and 30 and press enter. The simulation will move forward in time by one second. After each second passes, you are given updated information about your situation, including your height, downwards velocity, and remaining fuel. This process will repeat until you reach the ground or run out of fuel.

The game ends once your height reaches 0. You will be shown how fast you hit the ground and how long it took. To win, your landing should be at a velocity less then 2 ft/second.

SYSTEM MANUAL

This program is a simulation of a lunar landing written in C++ and uses the following libraries:

- iostream
- string
- fstream
- cmath

The program requires an external text file containing the game instructions. (The location for this file, which should be titled “input.txt”, may vary depending on your development environment.) The user will be prompted to give their ship a name. Instances of the text “$SPACECRAFT” in input.txt will be replaced with the spacecraft’s name.

The following global variables are used:

- const double GRAVITY – the force of gravity, which is 5 ft/sec
- int minFuelUnits – used to check user input during the game – should be 0
- int maxFuelUnits – same as above – set to 30 (although this is arbitrary)

The following procedures are straightforward; they are used to report information to the user through an output stream:

void show_title_text(ostream &os);
This procedure outputs title text to a given output stream. The only parameter is ostream &os, the destination stream.

void reportStatus(ostream &os, double time, double height, double velocity, double fuel, string name);
This procedure takes the name of the ship and values for time, height, velocity, and fuel, and outputs them in a neat format to a stream, &os.

void finalAnalysis(ostream &os, double velocity);
This procedure outputs a message to the user. The content of the message is based on the velocity of the spacecraft at touchdown (if/else statements are used to do this).

void touchDownStatus(ostream &os, double time, double velocity, double fuel);
This procedure takes the final values for time, velocity, and fuel, and displays them to the user via an output stream.

The next several procedures are more complex:

void introduction

istream &is – the input stream
ostream &os – the output stream
string target – specific text to replace
string replacement – the replacement text

This procedure reads from a istream, replaces all instances of a target string with a replacement string, and outputs the text to an ostream. While the end of the file hasn't been reached, the procedure reads text word by word from an input stream. Each time a word is read:

- The current word is stored in a string, current_word.
- If current_word is equal to target, the program outputs replacement to the ostream (instead of current_word).
- Otherwise, current_word is sent to the ostream.

The eof flag is removed using the clear function, and the position of the stream is moved back to the beginning of the file so the input stream can be re-read the next time the procedure is used.

void updateStatus

double &velocity – downward speed of the spacecraft

double burnAmount – how many units of fuel to burn

double &fuelRemaining – the number of fuel units remaining

double &height – the height of the spacecraft

Given burnAmount, this procedure modifies values for velocity, height, and fuelRemaining.

First, the current velocity is stored in a double, originalVelocity, which will be used later to calculate height.

Next, a new velocity is assigned:

Because gravity is 5 ft/second, and it has been only 1 second since the last update, we simply add 5 (GRAVITY) to the velocity. However, since we are burning fuel, we must account for a decrease in downwards velocity. For this simulation, each unit of fuel burned decreases speed by 1 ft/second. Thus, we subtract burnAmount from current velocity.

Thus:

$$velocity = velocity + (GRAVITY - burnAmount)$$

Now, we calculate the new height. This is done by taking the average of the original and new velocities and subtracting it from the current height.

Thus:

$$\text{height} = \text{height} - \frac{originalVelocity + velocity}{2}$$

Finally, fuelRemaining is updated by subtracting burnAmount from it.

void touchdown

double &elapsedTime – the elapsed time

double &velocity – the downward speed of the spacecraft

double &burnAmount – the number of units of fuel burned

double &height – the height of the spacecraft

This procedure calculates the exact time and velocity values during touchdown.
First, previous values for velocity, height, and time are calculated (*undoing calculations made during the last updateStatus*):

- prev_velocity: velocity: (GRAVITY – burnAmount)
- prev_height: height + ((prev_velocity + velocity) / 2)
- old_time: elapsedTime – 1

A double, delta, is created. It represents the fraction of a second (since the last update) the spacecraft takes to land. *As long as currentBurn is not equal to GRAVITY*, delta is calculated as follows:

$$\frac{\sqrt{prev\_velocity^2 + prev\_height\ (10 - (2 * burnAmount))} - prev\_velocity}{GRAVITY - burnAmount}$$

If currentBurn *is equal to* GRAVITY, delta is calculated as follows:

$$\frac{prev\_height}{prev\_velocity}$$

Lastly, we need to update our original variables. delta is added to elapsedTime, height is set to 0, and velocity is set to:

prev_velocity + (GRAVITY – burnAmount) * delta

Main program

After title text is displayed, the program prompts the user for a title for their log file. This is stored in a string, logFileName. ".txt" is appended to logFileName. An ostream object, logstream, is created. logFileName is created/opened for writing. An initial header is added to the log.

The program prompts the user for the name of their ship and stores the input in a string, shipName.

Now, the program prompts the user, asking if they want to see instructions. User input is stored in a string, instructionPrompt. A do/while loop is used to check if the user inputs y, Y, n, or N; otherwise the prompt repeats itself.

If instruction prompt is Y or y, an input stream for the instructions is created and input.txt is opened for reading. A string, spacecraft_input_target, is created with the text "$SPACECRAFT".

Now, the function introduction is used twice (once for log output and once for console output). The input stream is instructStream, the target text is spacecraft_input_target, and the string to replace it is shipName. After the instructions are displayed, instructStream is closed.

Now the game is ready to start. The game's initial conditions are set; the following doubles are used in the game:

- height – set to 1000
- velocity – set to 50 (in the downwards direction)
- fuel – set to 150
- elapsedTime – set to 0
- currentBurn – to be set later

The game takes place within a while loop which continues as long as height is greater than 0.

First, reportStatus reports the current status to the console and log.

Next, if there is fuel left, we need to know how much to burn, which requires user input, collected within a do/while loop. Before the start of the loop, a string, userQuestion, is created with the text “How much fuel do you want to burn?” At the end of the loop, it is changed to an error message. Thus, future iterations of this do/while loop, which only appear if the input is out of bounds, show the error message instead of the initial question. The user’s input is stored in currentBurn and output to the log. The loop continues if current burn is less than minFuelUnits (0) or greater than maxFuelUnits (30).

Next, if currentBurn is greater than fuel, it is changed to fuel. This prevents fuel from dropping below zero; if the user tries to burn more fuel than they have, all their remaining fuel will be burned. Note; this is only relevant in situations when the user is running low on fuel (fuel < 30).

Now that currentBurn is set, updateStatus can be called using velocity, currentBurn, fuel, and height. At the end of the game loop, elapsedTime is incremented by 1.

When the loop finishes, height should be <= 0. The touchdown procedure is used to calculate the exact status of the spacecraft at height = 0 as well as the exact amount of time the landing took. These values are displayed using touchDownStatus, and finalAnalysis prints a message for the user rating their landing (based on velocity). The game is now over; logStream is closed and the program ends.