A Simple Lunar Lander Clone

Juraj Borza

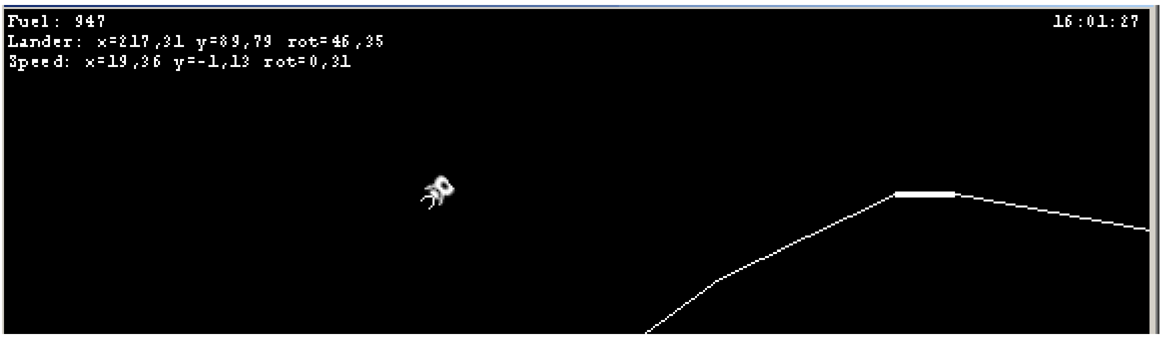
8 Dec 2009 Public Domain

Rate this: ★★★★★ 4.80 (14 votes)

This article describes how a Lunar Lander game was made with simple C# and GDI+ code

Download demo binary - 8.51 KB

Download source code - 14.84 KB



```
Fuel: 947                                                        16:01:27
Lander: x=217,31 y=89,79 rot=46,35
Speed: x=19,36 y=-1,13 rot=0,31
```

Introduction

This article describes the development of a small Lunar Lander clone. Lunar Lander is an arcade game by Atari, and it's one of the games with a simple, but great idea, that people loved to play over and over again.

Lunar Lander is a game where you are given control of a landing craft that attempts to land on the Moon. Lunar gravity is applied, so the lander gradually falls and would crash against the rough surface. The player must apply thrust from the main thruster and rotate the spacecraft to navigate to a landing pad.

The game is controlled by arrow keys - Up, Left and Right, and X, which controls afterburner (increased thrust at increased fuel usage).

A limited supply of fuel which burns at operation of thrusters is an element of difficulty, as is limited tolerance for lander's velocity and rotation at the landing. Attempt to land too fast, and the fragile vehicle would crash.

Elements of Design

I have decided to separate game objects into these classes:

- `Game` - handles game update and events
- `Ground` - holds information about the level

- `Lander` - contains information about the lunar lander and updates it

The game uses Windows Form for displaying the rendered `Bitmap` on a `PictureBox` and capturing keyboard events. The `KeyDown` and `KeyDown` events from Windows Form are sent to the Game, which toggles these keys as *pressed* in the Game. Then for example, the `Lander` object can check if an Up key is pressed and apply thrust via the main thruster.

The game runs in timesteps managed by Windows Form's `Timer`.

Terrain is randomly generated. It's internally stored as a list of connected points; in the rendering they are connected by lines.

Graphics

Game drawing is done in GDI+, via the `Graphics` object. In each timestep the game is drawn, first drawing the ground, the lander and then some statistics (speed, position). As the lander can be rotated, we need to apply rotation transformation to the `Graphics`:

Hide Copy Code

```
Matrix m = new Matrix();
m.RotateAt((float)MathFuncs.RadiansToDegrees(Lander.Rotation), Lander.Location);
g.Transform = m; //assigning the matrix to Graphics object
g.DrawImage(Lander.Sprite, Lander.LocationGraphics);
m.Reset();
g.Transform = m; //restore original matrix
```

Game Updating and Simulation

The game updating is done in timesteps in the `Game` object. The game first updates the lander's position via calling the `Lander`'s `Update` method and then we check for collisions.

The physical simulation is quite simple. We apply forces to the lander craft - first the gravity, which is:

Hide Copy Code

```
sY += Gravity * ts.TotalSeconds;
```

Then we check for each keypress (main or rotating thruster) and apply force according to thrust and rotation:

Hide Copy Code

```
sY -= Math.Sin(MathFuncs.NormalizeAngle(Rotation + Math.PI / 2))
    * totalSeconds * ThrustSpeed;
sX -= Math.Cos(MathFuncs.NormalizeAngle(Rotation + Math.PI / 2))
    * totalSeconds * ThrustSpeed;
```

Then we need to check for collision with the ground - we just check if each corner of lander's bounding rectangle is 'under' a terrain line. As the ground/terrain is just a sequence of joined lines - slopes, that only goes from left to right, then no line can be above or below another one. So if we found out that after timestep, one of the bounding rectangle's points is located under the line, the lander must have either landed or crashed.

Checking if a `Point` is under the `line` (in 2D) consists of calculating Y-coordinate of a point on the `line` with the same X-coordinate of the `Point` and determining if `Point.Y` > line's Y coordinate at `Point.X`. So something like `Line.GetYCoordAtX(point.X) > point.Y`.

Conclusion

The game was hastily designed and implemented, so the class design isn't as good as it could be. If you'd like to add something, here are additional ideas or exercises:

- Some kind of scoring functionality could be implemented, based on final landing velocity, rotation and fuel left.
- Sound effects could be added to enhance the experience.

- Analog control via mouse or gamepad could provide finer grade of control.
- Multiplayer, where two players would try to land on the same screen.

History

- 8th December, 2009: Initial post

License

Share

About the Author

Juraj Borza
Software Developer MicroStep
Slovakia 🇸🇰

Juraj Borza is currently working as a full-time C# developer located in Bratislava, Slovakia. He likes coding for fun and releases some projects under open-source license.

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

your work
Mithileshforcode 2-Jan-14 19:21

My vote of 5
conmajia 5-Jul-12 7:07

Fun game, different than the Win 3.11 clone
Bojan Sala 11-Oct-10 3:37

that's fun
milkplus 26-Dec-09 10:47

Re: that's fun
Juraj Borza 27-Dec-09 4:06

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.