Gravity Golf

Sean Lewis

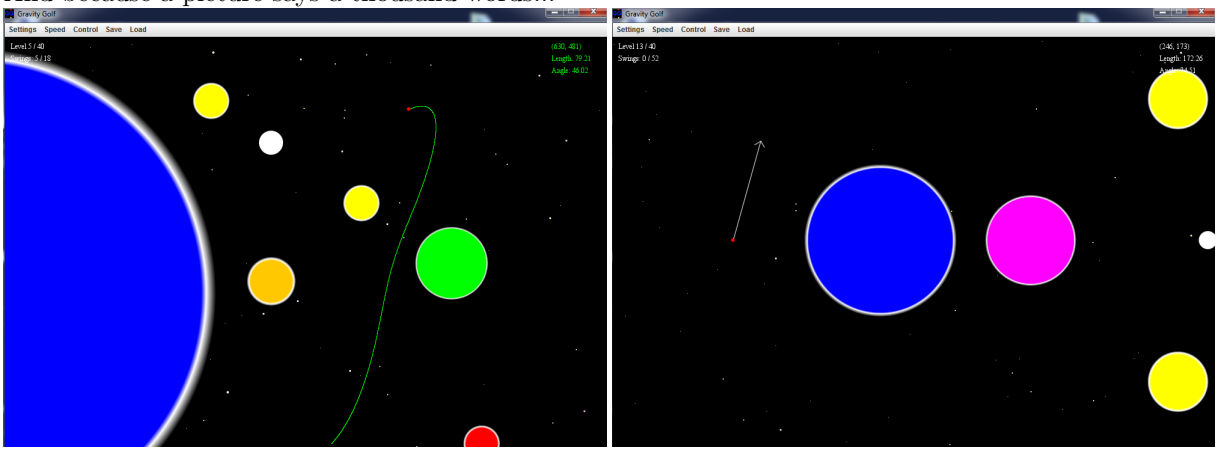
1 Overview

The purpose of this file is to provide a succinct description of the Gravity Golf game and its associated utilities. The base game can be described simply; it is a 2-d gravitational simulation where the player tries to "launch" the ball with an initial velocity so it reaches a goal.

The project has grown far beyond that old goal, however. Additional parts include:

- Many graphical effects, most notably the shaking and particle effects on collisions
- A useful Level Editor GUI for easy creation and testing of new level designs
- A Level Solver that concurrently computes all solution points to a level

And because a picture says a thousand words...



2 Program Structure

The entire project is divided into five packages: `structures`, `graphics`, `game`, `editor`, and `tests`. I take a bottom-up description of the structure.

2.1 Structures

The lowest level of control is in the objects being simulated: the planets, the ball, the moon, etc.. Each of these is modeled by its own class. Two abstract classes, `CircularShape` and `MovableCircularShape`, make much of the implementation trivial.

Important aspects of this package include the final `CalcHelp` class, which provides many often used methods (especially the `getAngle` method) and the `Vector2d` class, which provides a convenient representation of two-dimensional vectors that the objects use to represent their velocity.

The most important part of the entire package is the `updateLevel` method of the `Level` class. This method performs all computations needed to advance the simulation by one tick, including summing forces on the ball and moving it, checking for any collisions, advancing the moons' positions, and checking if the ball has reached its goal.

2.2 Graphics

The `graphics` package is a small package meant to implement the various graphical effects used in the game. Most implementations are trivial, but these include:

1. `CollisionEffect` implements the screen shaking (simulated by dampened harmonic motion) and randomized particle effects when the ball collides with a planet
2. `TrailEffect` draws the trail of the ball
3. `WarpDrawer` draws arrows between all paths that the warps can make the ball travel between
4. `GravityVectorsEffect` draws the gravitational forces influencing the ball
5. `ResultantDrawer` draws the total resultant of the gravitational forces influencing the ball
6. `InfoDisplay` displays information to the user, including the angle and magnitude they shot at
7. `MenuScreen` displays a intro menu and information to the user about keyboard shortcuts

2.3 Game

The `game` package serves as the front-end for displaying the game and tracking the game state.

The `DataHandler` class handles the reading in of the level file (`levels/levels.txt`), as well as the settings file (`settings.txt`).

A `GameManager` is a higher-level container for the game state that tracks the overall information being used by the game, including a `List` of all of the `Level` objects, the current level, and the number of swings the user has taken.

`GamePanel` is the class where everything comes together; it drives the animation and calls the routines from every other structure to advance and display the game.

The `GravityGolf` class is the class that contains the main method and is run to launch the game.

3 Points of Interest

3.1 Solving a Level

A common question when playing the game is "how the hell do I do this level?" Wanting to answer this as thoroughly as possible, I implemented a tool for finding every solution point to a level. The low-level implementation is in the `Level` class' `possibleWin(Point2d p)` method.

Finding all solutions is simply a matter of iterating over every possible point the user could click and checking if the point would be a possible win. Unfortunately, the standard game allows the user a total of $\pi * 300^2 \approx 280000$ different possible points. Thus, a complete solution check would frequently take upwards of 10 minutes. Thankfully, the task can be made parallel with few changes, since it divides into those 280000 different tasks.

The `editor` package contains the `LevelSolver` class, which provides methods for computing a Level's solution set, and a main method for calling the methods and printing all of the data to a file. `GamePanel` can be modified (by setting `DRAW_SOLUTIONS = true`) to display these points in-game, given that the solution sets have already been computed and put in files.

3.2 Random Level Generation

A intriguing idea is to compute random levels for the user, allowing them to play a different set of levels every time they start the game. A primitive method for computing randomized levels is implemented in the `editor.Randomizer` class.

Impediments to fully implement this include: ensuring that a level is solvable (even in parallel, one level can take up to 5 minutes to find all solutions) and intelligent placement of the goal.

A useful test for observing the `Randomizer` in action is `tests.RandomLevelTest` which rapidly generates a new level each time the user presses any keyboard button.