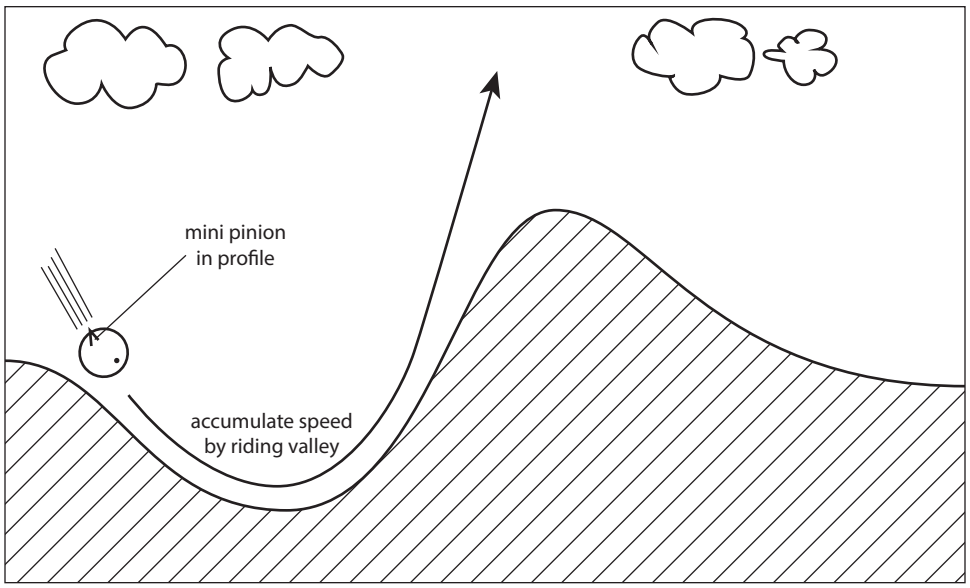
ICS4U Summative Proposal

Mitchell Kember

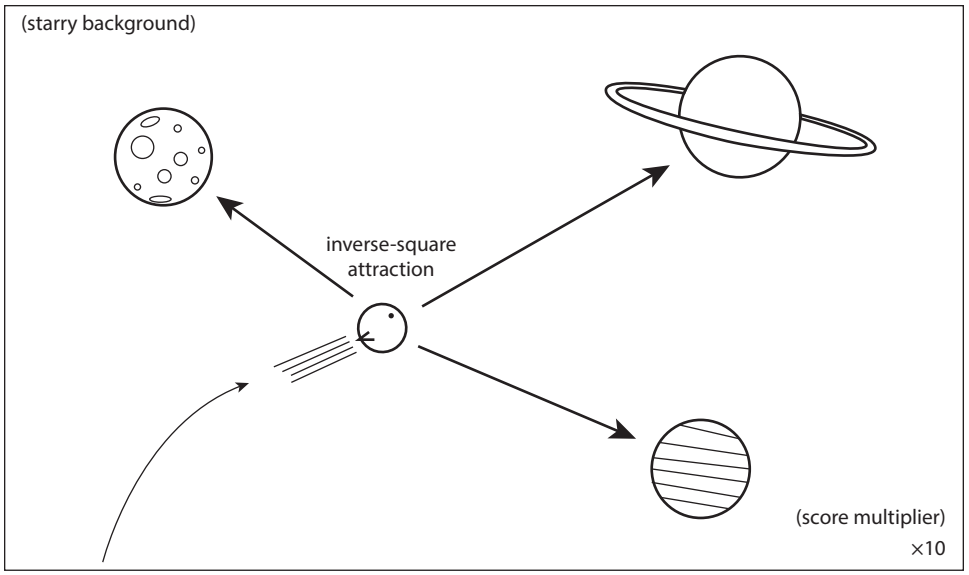
16 October 2013

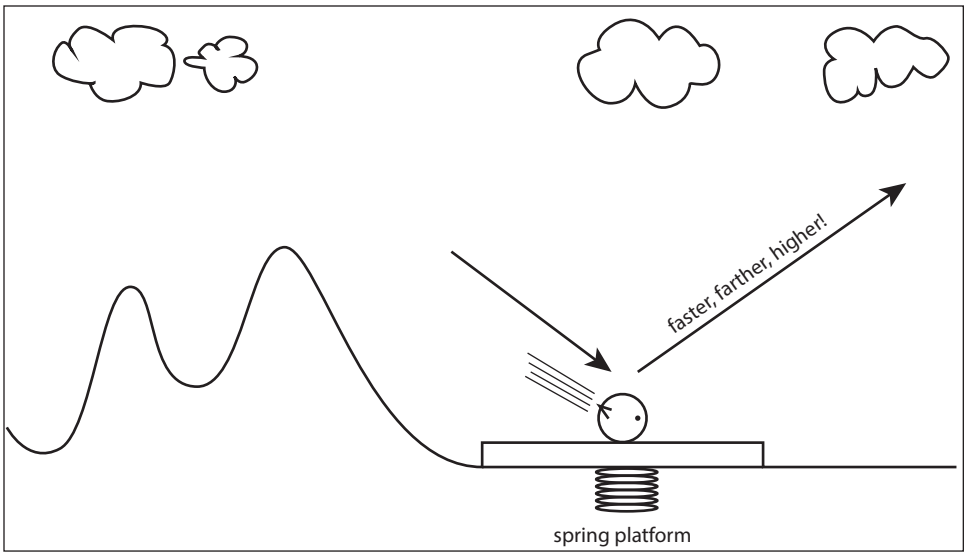
DESCRIPTION

Mini Pinions is a game that (some might say) is similar in concept to *Tiny Wings* for iOS. It borrows the basic game mechanism: a bird must accumulate speed to earn more points and reach the end of the level. The bird can't really fly, so it must dive towards the ground with the right timing such that it slides along a smoothly curved valley, shoots up on the other side, and reaches a higher altitude, thus defying the law of conservation of energy. The player is only in control of when the bird follows a regular ballistic trajectory and when it drops like a deadweight. The game is made more interesting with the addition of the features listed in the next section.



The game will also incorporate planetary gravity and spring physics. When the bird reaches very high altitudes, the player will have to dodge planets which attract the bird using the inverse-square law. Spring-loaded platforms will make it easier for the bird to reach space.





FEATURES

I have tried to only put the features I am sure I will be able to complete here. The others are under "If I have time." This all really depends on time; I am pretty sure that I could do all of them given enough of it. The distinction is only a rough guess, so I could end up being off on either side (I might easily finish them all, or I might barely finish the ones listed here). I may also decide that different features are more appropriate to spend time on as

I progress through the project. I could have added more, but these should be good to get me started.

- main menu and level selector
- instructions screen with high scores
- smooth-feeling collision with ground when the bird's direction is close enough to parallel to begin sliding instead of rebounding
- a variety of terrain types
- visually appealing ground and sky
- score multiplier
- score algorithm (weighting of different things)
- speed-up strips of land
- frenzy mode (increased points, flashing colours, etc.)
- coins and power-ups
- automatic scaling to always show the landscape
- combination of multiple different physics simulations
  - uniform gravity (obvious)
  - planetary gravity (go really high and you reach space?)
  - springs (land on bouncy surfaces)

IF I HAVE TIME

- choose between multiple birds (different appearance and different mass)
- surfaces with different coefficients of friction
- achievements & statistics
- ghost trail of previous run
- smooth transitional animations
- animation of flapping mini pinions
- dynamic background (something other than day/night cycle)
- background music & sound effects
- custom level designer (Bézier curve tool for smooth hills)
- cheat codes (or cheats that are unlocked after beating the game)
- race against computer

SKILLS & CONCEPTS

- project organization (splitting up files, classes, functions)
- clear documentation
- debugging skills

- game design (making it fun)
- graphic design (textures)
- drawing/rendering to screen
- mouse & keyboard input handling
- variables & arrays
- conditionals & loops
- functions & classes
- Object Oriented Programming (OOP)
  - abstraction, encapsulation, polymorphism, inheritance
- or functional programming (see next section)
- vector quantities
- physics simulation: position, velocity, acceleration
- regular (uniform) gravity
- planetary gravity (inverse-square law)
- spring physics

TOOLS

I will use Processing to create *Mini Pinions*. However, I may not use it directly in its Java environment. There are a few reasons for this. First, it will be more interesting and educational to use a different language. Second, I did the same thing for both my summatives in ICS2O (I tried a different language that I was not necessarily as comfortable in), so why not do it again? Third, I think Processing is great for quick sketches, but as a language and as an IDE for relatively large projects, it is painful. This is because, in my experience, it makes it impossible to debug programs by highlighting a seemingly arbitrary line of code that isn't even in the same file as the one causing the error. This still seems to be happening with the latest version.

Instead, to make things more interesting and to learn more, I am thinking of using something completely different: Lisp. My favourite dialect of Lisp is Scheme, but it is impractical for most real-world applications. I have learned a lot about Lisp and Scheme by reading *The Little Schemer* (I also started *Structure and Interpretation of Computer Programs*, but didn't finish). Rather than Scheme, I will use Clojure, a dialect that runs on the Java Virtual Machine (JVM). I found a project on GitHub called Quil that provides painless integration for Clojure and Processing.

If this turns out to be more trouble than it's worth, I will probably just switch to using regular Java Processing. I hope I don't have to do that, though.

STEPS

The halfway checkpoint is step number 7.

1. Figure out how the basics of whatever environment I end up using: drawing to screen, handling input, and doing this within a main loop.
2. Create a skeleton of the project with files for each logical section (e.g., one for each screen, maybe several for different aspects of the main game).
3. Code the basics of the GUI so that it will be easy to connect screens via buttons. The draw loop will always draw the current screen, and maybe free the resources used by the screens that are no longer displayed.
4. Flesh out the structure of the main game screen. Start with the essentials: modelling the physics of the bird. Don't focus on scores or pretty graphics for now.
5. Create a basic level to test the bird in. Don't spend too much time on it, but it should have a variety of different hill shapes for the bird to slide on.
6. Perfect the bird sliding simulation. Determine the threshold angle that separates near-parallel landing angles (slide) from all the others (bounce off) – or maybe interpolate between these two extremes somehow (perfectly elastic collision when perpendicular, less so as angle decreases?).
7. [**Halfway**] Finish the basic level. It doesn't need to have all the features, but it should be playable and it should at least show a score. It doesn't need to look amazing, but it shouldn't look like a testing environment any more.
8. Quickly implement the other screens before improving the main game: main menu, instructions, high scores, level selector.
9. Incorporate planetary gravity somehow in the game. One idea: get high enough and you will go into space, where you get more points but must avoid crashing into planets. It should be a diversion to add variety to the game, not a separate part of the game.
10. Incorporate spring physics by adding bouncy platforms that behave according to Hooke's law. This helps the bird gain altitude much quicker than well-timed sliding does.
11. Add little things to make the game more fun: speed-up areas, power-ups, more complex score mechanism (multipliers, frenzy mode, etc.).
12. Add all of the features listed in the "Features" section that haven't already been mentioned.
13. Polish everything as much as possible before moving on to the next step.

Make sure the code is well-documented, the game has no glitches, and everything is working well in general. Have a friend try it to get a second opinion on everything.

14. Add as many features in the "If I have time" subsection as possible. Choose features based on which will add the most value to the game and based on the time they will require.