Two Interesting Games: *Little Inferno* and *Human Resource Machine*

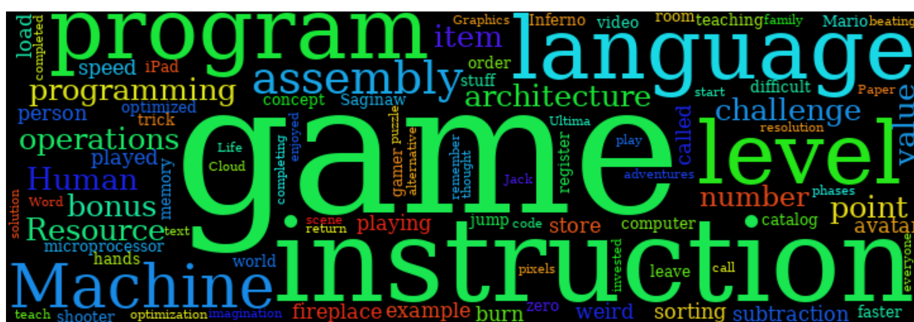

Figure 1: Word Cloud

As a guy who remembers a world before *Pac-Man*, I admit to having gone through phases in which I played a lot of video games. As a child my family had a game console for a time; not an Atari, but a Sears "Tele-Games Electronic Games Motocross Sports Center IV," which had *Pong*-style games out the wazoo, and a deeply dumb motorcycle jump game. Back then you had to use a lot of imagination to translate the low-resolution, barely-animated game representation into something more vivid, but I had fun with it. It wasn't like there were a lot of great video game alternatives. During the home computer years I was a big fan of Infocom text adventures, completing many of them and dabbling in programming my own text adventures (fortunately all lost). Graphics games were in their infancy when I first played them. Graphics games on the TRS-80, in black and white with a resolution of 48 pixels by 128 pixels, were notably crude; remember that bit about having to use a lot of imagination?

Over the years I mostly was not a big gamer, although I went through some phases when I played a lot of games. In the early nineties, I played *Doom*. I had a brief infatuation with the PC-based *Ultima* games and played the hell out of several of them, including the two *Ultima Underworld* games. A few years later, I was a fan of some Nintendo 64 titles and I'll never forget conquering the tough boss "Mad Jack," an evil jack-in-the box, in *Donkey Kong 64*. I gradually migrated away from first-person shooter games towards racing and puzzle games,

enjoying *Mario Kart* and beating *Paper Mario* and the sequel, *Paper Mario: The Thousand Year Door* as well as *The Legend of Zelda: The Wind Waker.*

When I moved to Saginaw and was living alone for a while I bought a few games from Steam, including *Half-Life 2.* It was *Half-Life 2*, particularly the "We Don't Go to Ravenholm" episode, that put me off first-person shooter games, probably forever, but I enjoyed (and completed) *World of Goo.*

I'm glossing over many other games that were significant to me over the years, but the point of this introduction is to show that I've gravitated more towards puzzle games and physics games. I have never gotten into MMPORGs and modern, realistic first-person shooter games leave me cold. That makes me what hard-core gamers might call a "casual gamer." This term seems to be pejorative, but I embrace it, because it seems to me that the alternative is "someone who wastes far too much time playing games." I don't believe time spent playing games is actually entirely *wasted* — recreation is important for everyone. And some games, for example the second game I'll discuss below actually can hone teach useful skills. But the return on time invested into playing video games is more limited than the return on time invested in other hobbies, such as learning to build things with your hands, or learning to play an instrument, or doing pretty much anything outside.

Since purchasing an iPad, I never really found iPad games I thought were worth playing; adaptations of other games didn't always work well on the iPad touch-screen hardware. But recently I was browsing the App Store and found two games that I really like.

Little Inferno

The first is an oddball game called *Little Inferno.* The setting and plot is bizarrely simple (and simply bizarre). You've just received a brick fireplace, called a "Little Inferno Entertainment Fireplace," made by the Tomorrow Corporation (which is the name of the actual company that made the game, by the way). You don't have anything to burn in it, but you can order things from the company. They include a starter catalog. Everything is delivered right to you. The catalog is a mishmash of weird parody objects: broken toys, expired and discontinued food items, weird junk-shop finds, and the occasional amusingly explosive, infested, or toxic item. It's first-person, and for most of the game you sit in front of the fireplace and burn stuff up, and order new stuff. Stuff is delivered in just a few seconds, with bigger-ticket items taking tens of seconds or minutes, but the game allows you to spend your bonus points on rush delivery. So you can indulge your impatience, and you quickly find that indulging your impatience makes you even less patient. That's about it for the basic gameplay.

There's a story arc that starts to emerge; you get mail, and you get weather reports. Apparently your community is suffering from terrible winter storms and they've been going on for some time. In fact, you never seem to actually leave your home, or eat or drink anything. To me, it's a wonderfully weird satire of

Amazon Prime. You earn money by burning things in your fireplace, which you can use to buy more things. When you burn "combos," weird groupings of items with some symbolic or metaphorical relationship, you get bonus points. You can use the bonus points to order more and more things, and they arrive ever-faster. As you level up you get more catalogs, with weirder and more expensive items in them. The items burn in entertaining, and sometimes horrible, ways. For example, there's a stuffed cat called Kitty Kitty Poo Poo. When you ignite it, it poops. A lot. It seems to be filled entirely with poop, which burns.

However demented and gross the game's sense of humor, it was never quite as dark as I expected. For example, one of the required combos is called "Sorority Party." I thought it involved the "Low Self-Esteem Action Doll" and the chainsaw, but that may be a remnant of watching one too many low-budget horror movies back in the eighties. It actually involves balloons, not chainsaws.

Eventually you level up enough to bring the game to a satisfying and surreal conclusion. I won't spoil it for you. This game isn't for everyone, but it doesn't take all that long to complete it, and I had fun playing it.

Human Resource Machine

I discovered that the same developers have released another game, *Human Resource Machine*. *Human Resource Machine* is a bit difficult to describe if you aren't already familiar with very low-level computer programming. In particular, if you've ever studied assembly language programming, it will seem familiar. You are an office drone, starting out in the mail room. You work your way through a series of rooms. In each room, your supervisor gives you an assignment involving sorting or filing or rearranging boxes, moving them from an incoming conveyor belt to an outgoing conveyor belt. But you don't move the boxes yourself in real time; you don't have direct control of your avatar. You can't drag him around or make him pick up a box. Instead, you write a program, in a very simple assembly language, to control your avatar. Then you run the program. When you run it your avatar zooms around, executing the instructions. You can make your avatar run the program step-by-step, even stepping backwards, and change the speed.

Learning Assembly Language

The assembly language you are learning starts out very simply. Initially you have only two commands, one to take an item from the inbox, and one to put an item in the outbox. As the challenges get more difficult you get addition and subtraction, and jump operations, and conditional jump operations, and then load and store operations, and then indexed load and store operations, and then indexed indirect load and store operations. If you make it through the whole game, you will have mastered the basics of assembly language programming.

The Human Resource Machine instruction set is not a real instruction set architecture (that is, it isn't taken from an existing microprocessor), but at their

heart, instruction set architectures are very similar. So once you've mastered this one, you could presumably learn any real instruction set architecture, just as after I learned to program the Z-80 (the microprocessor inside the Radio Shack TRS-80), it was relatively easy to learn to program the 6502 (the microprocessor inside the Apple II). In fact the Human Resource Machine architecture is a bit like that of the 6502, or the Atmel AVR family.

I have some small criticisms to make of the Human Resource Machine instruction set architecture as a teaching tool. It is a little more irregular than I think it should be. Some instructions leave objects in your hands, and some don't (your "hands" represent a sort of accumulator, or common register used to do most operations including math operations). In real instruction sets, when you load something from one register into another, it is always copied, and a register is never "empty" — they always have a value. A "null" or empty value is a concept imposed at a higher level of abstraction in languages such as C. But real instruction set architectures all have their quirks, so it's not that surprising to come across some oddities in this architecture as well.

Gamification of Low-Level Programming

If this game had existed when I was teaching the System Software course I taught at Saginaw Valley State University, and I could have arranged for the students to have access to it, I would have assigned it. It is pretty much the best attempt at "gamification" of programming I've ever seen. I especially like the fact that it teaches assembly language, which is the lowest in abstraction, only one level above the raw bits in memory that microprocessors actually execute. Most programming environments you see online teach Python, or Lua, or JavaScript, or some other high-level language. That's valuable, but it's refreshing to see a teaching tool that teaches programming at a lower level.

Most of the basic problems are simple, but they get progressively harder. You have to invent some techniques to map high-level concepts to your low-level assembly language instructions. For example, you don't have an instruction to determine if two numbers are equal. But you have a subtraction instruction, and you can compare a value to zero. So you have to figure out that you can check to see if two numbers are equal by subtracting one from the other. If the result is zero, they were equal, otherwise, they weren't. This is just one of many such techniques you have to work out to solve the problems. It's good practice because when you are implementing programs in assembly language, you have to come up with tricks like these, mapping high-level concepts to low-level implementation, all the time.

To get to the end of the game, you have to get through a whole series of challenges. In the final challenge, you must implement a sorting algorithm in assembly language. Sorting a list of numbers can be accomplished with a trivial library call in Java or Python, but in assembly language it makes a fairly challenging program. I chose a maximally-inefficient version of a sorting

algorithm known as "bubble sort." It contains a lot of redundant operations, which made my little avatar go berserk for five minutes when I ran it. Some easy optimizations are possible, but the first challenge is just to get a program working.

Optimization

When I completed all the required challenges, I started doing the side challenges, a parallel series of graded exercises provided for additional practice. In addition to just solving each challenge, you can try to get one or two bonus points for each challenge. You can obtain one point by reaching (or beating) a target value for the minimum program length, and another point by reaching (or exceeding) a target value for execution speed. In other words, the first bonus point comes from writing a sufficiently short program. The second bonus point comes from writing a sufficiently fast program.

In some simple cases one solution will meet both objectives. But in many of them, the size-optimized program looks *very* different than the speed-optimized program. A speed-optimized program might be much longer, removing loops in favor of repeated sequences of instructions, and pulling out all kinds of tricks to make it run faster. For example, earlier I described a trick to compare two numbers. What if you still need the first number, after you have subtracted the second number from it? You could store it in memory, then do the subtraction and comparison with zero, then load it from memory again. That takes three instructions. Or, you could just do the subtraction and comparison with zero, then "undo" the subtraction by adding back the second value, which takes only two instructions. It may seem silly, but it's faster. And real-world code that has to run as fast as possible often uses similar tricks.

Perhaps not surprisingly, an enterprising person has set up a GitHub repository with optimized solutions. See https://github.com/atesgoral/hrm-solutions.

I've been programming computers since 1977. Although I met the requirements to complete the game, I still have not managed to beat all the optimization challenges. As the game says, some of them are quite difficult.

My Own Implementation

But because I enjoyed the concept so much, I did start make my own little virtual machine that implements the Human Resource Machine instruction set, written in C, which runs on an ATtiny 441 microcontroller. It is a skeletal project, and only includes enough sample code in the HRM instruction set to test the basics, but if you want to play with it, the code is here: https://github.com/paulrpotts/TinyHRM.

I had the idea, originally, that I might communicate with the tiny microcontroller running the virtual machine via a serial port, uploading a little Human Resource Machine program and letting it run. I'm not sure I will ever get around to

completing that, since I am getting paid to write other programs and I have more than enough paid work to keep me busy, but at least I had some fun with it.

The game has cut scenes, and a concluding scene. Again, I won't spoil it for you, but as a programmer about to celebrate my 49th birthday, I found the concluding scene slightly unnerving. It's a great game and I highly recommend it, and I am looking forward to more releases from Tomorrow Corporation.

Saginaw, Michigan
September 26th, 2016

This work by Paul R. Potts is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/.