SPACEWARDUINO

Kaivan Wadia (Department of EECS) – Massachusetts Institute of Technology

Advisor: Prof. Philip Tan

ABSTRACT

Spacewar! was one of the first ever computer games developed in the early 1960s. Steve Russell, Martin Graetz and Wayne Witaenem of the fictitious "Hingham Institute" conceived of the game in 1961, with the intent of implementing it on a DEC PDP-1 computer at MIT. Alan Kotok obtained some sine and cosine routines from DEC following which Steve Russell started coding and the first version was produced by February 1962. Spacewar! was the first computer game written at MIT. In the duration of this project we hope to implement the same game on an Arduino Microcontroller with a Gameduino shield while examining the original assembly code and staying as true to the original version of the game as possible.

TABLE OF CONTENTS

1. INTRODUCTION

The major aim of this project is to replicate the first computer game written at MIT, Spacewar!. In the process we aim to learn a lot about the way video games were written for the machines in the pioneering days of computing.

Spacewar! is game of space combat created for the DEC PDP-1 by hackers at MIT, including Steve Russell, Martin Graetz, Wayne Witaenem, Alan Kotok, Dan Edwards, and Peter Samson. The game can be played on a Java emulator at http://spacewar.oversigma.com/. We also obtained the base version of the Spacewar! source code which was written in assembly language for the DEC PDP-1. Our goal in this project is to make a version of the game that runs on an Arduino microcontroller with a Gameduino shield that we can demonstrate in the foyer of the GAMBIT Lab.

In the duration of this project we understood what's going on in the PDP-1 source code that the emulators use. This has meant learning the PDP-1 assembly language MACRO from old manuals written in 1962. These manuals were obtained from the bitsaver's archive[1] . We also got hold of a PDP-1 emulator through the Multiple Emulator Super System[2][3]. The process itself has been slow as a lot of information is buried in the technical documentation or is no longer available as it was assumed knowledge at that point of time. It has also been amazing to go through the old manuals of the PDP-1 and MACRO and understand and share the experience of those hackers back in the 60s.

2. UNDERSTANDING THE SOURCE CODE

There isn’t much information of the Spacewar! source code available online. To understand the source code we had to use the old PDP-1 and MACRO manuals[1]. It also greatly helped to look at the compiled (processed) version of the source code so as to compare it with the clean un-compiled (pre-processing) version and get the translation of each instruction in machine code, which was very useful for understanding exactly what is going and the layout of the memory. This in turn was very helpful in understanding the low level manipulations performed by the code while running the game.

MACRO inlines all macro definitions. The MACRO equivalent of functions is called macros. When we went through the source code we noticed that wherever a macro was used in the source code the compiled version had gone through and expanded it out inline by introducing temporary variables for each of the macro’s arguments. This explained what all the labels starting with "ZZ" meant in the source code.

The numbers mentioned in the source code are all in octal. This took a long time to figure out although we should have figured out much earlier. We were expecting it to be in either decimal or hexadecimal but were surprised to find that the numbering was done in the octal system which made a lot of sense once we discovered that the a word in memory was 18 bits long instead of the usual power of 2.The only case in which the assembler interprets a number as decimal is when the MACRO directive “decimal” has been issued in a macro definition.

We also discovered that there was no stack in the PDP-1. As we were used to working at C level of abstraction this was very confusing at first until we figured out that the macros which look like functions get inlined when processed by the MACRO compiler. Everything is just one big sequential amalgamation of data and instructions.

Labels are used both for naming variables and controlling flow. This was very hard for us to wrap our heads around at first. As mentioned before, there is no separation in memory between data and instructions. As a programmer you would have to make sure that you never let the program counter point to a place in memory that contains data rather than an instruction. The Spacewar! programmers commonly used a label to name an address in memory that they want to jump the flow control to and they also used a label to refer to a memory location that they were only going to store data in. As far as we can tell there is no naming convention which distinguishes them. Sometimes data can be stored immediately next to instructions in memory. However, the majority of the game object data is stored in a big contiguous block of memory after the instructions, rather than being completely interleaved.

Including the symbol "i" after an instruction makes it indirect. We never found an explicit statement of this, but my teammate Owen Macindoe inferred it from looking at the opcodes in the original source code. Indirect instructions take a memory address as an argument which tells them where to find the real argument. This idiom is very commonly used in Spacewar!

The parentheses define constants. MACRO automatically allocates some memory to hold the constant, stores its value there, and then replaces the constant symbols with the memory address of the constant. For instance when MACRO reads the instruction "lac (4000" in the preprocessing phase, it causes some memory to be allocated, say address 047403, and then stores 4000 in that address, replacing the original instruction with "lac 047403". This is all done prior to execution.

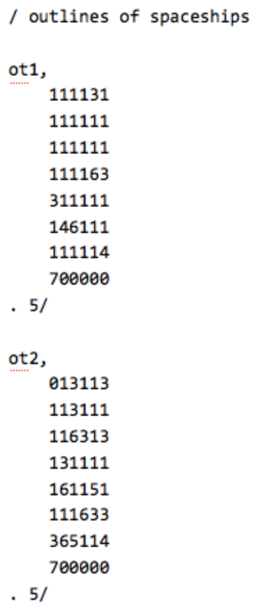
One of the most frequently seen instructions was “dap”. The instruction "dap x" deposits the accumulator contents into the argument portion of the memory address x. This lets you change the argument of the instruction stored at that memory address. One of the most common idioms that the Spacewar! programmers use is to manipulate the flow of the program by writing "jmp .", which is a command that says "jump to this address" (which would result in a tight infinite loop if it were actually execute) and then to use the dap instruction to overwrite the target of the jump from "." to some other address that they load into the accumulator. This makes it hard to look at the code and immediately tell what the flow control is going to look like without tracing the execution, since you need to know what the "." is going to be replaced by when the program is actually running. They use a similar idiom for loading data, for example writing "lac ." which if executed straight up would load the contents of the current address (i.e. the "lac ."

instruction itself) into the accumulator, but they then use dap to conditionally change the target from "." to some other location that contains data that they want to operate on.

Spacewar! is written very close to the machine. The programmers have used a wide range of bitwise manipulation tricks that are not often used in modern programming techniques. They rotate bits in memory using bit-shifting so as to store two short numbers in space normally used for one long number. Number representations are shifted around so that they are in the left or right side of the memory address depending on where a particular instruction call requires that it needs to be. This comes up quite frequently with the display instruction “dpy”. They use MACRO to repeatedly double numbers in the preprocessing stage so that they get bit-shifted to the location in memory that they want before execution. They add together the bit codes of instructions to create combined instructions. These tricks have not been commented very well in the source code and were difficult to pick up and understand why they were being used. This required a lot of poking around and playing with the emulator in order to figure out the use of these tricks.

3. VECTOR GRAPHICS AND SPACESHIPS

After understanding the basic workings of the Spacewar! source code and going through the PDP-1 and MACRO manuals the first major block of code we tried to analyze was the one that was drawing the game state, i.e. drawing the two spaceships. A section of code had been marked by the comment "outlines of spaceship", followed by two blocks of code with the labels ot1 and ot2. This seemed to be the logical starting place to try and understand how the "Wedge" and "Needle" were drawn in Spacewar!. The two blocks of code just contained a sequence of numbers shown in the image below. Understanding what these numbers meant was integral to figuring out how the spacehips were drawn.



```
/ outlines of spaceships

ot1,
	111131
	111111
	111111
	111163
	311111
	146111
	111114
	700000
. 5/

ot2,
	013113
	113111
	116313
	131111
	161151
	111633
	365114
	700000
. 5/
```

To the naked eye these numbers seemed to be an encoding of some kind. The simplest solution seemed to be that the numbers were a bitmap encoding of the spaceships. This idea fell through as the numbers gave us nothing recognizable when used as a bitmap encoding. Looking through past interviews with the developers of Spacewar! it became apparent that the spaceships were drawn using vector graphics techniques. This led us to the process of taking consecutive pairs of numbers as the X and Y coordinates of a series of vectors that would be drawn one after the other. This technique also did not produce any positive results putting us back at square one in terms of understanding the method used to draw the game state.

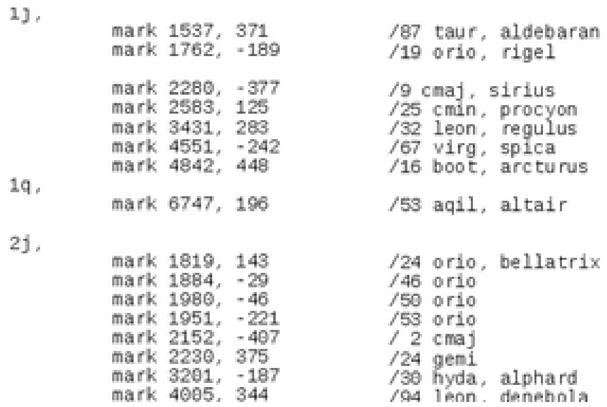
On analyzing different sections of the code in hope of the answer to our problem we came across a section of the code bearing the comment “outline compiler”. This portion of the program was writing out machine instructions into an empty block of memory at the end of the main program. Some of these instructions were display instructions giving us evidence that this section of the code had something to do with the drawing of the spaceships. Our next discovery was a section of the code that took the heading of a spaceship and computed the sine and cosine of the heading. It stored linear combinations of the result in memory. The same memory was being referenced by the code generated by the “outline compiler”. This seemed to match the code that would be needed to multiply vectors using a rotation matrix. We had finally come across the code which controlled the rotation of the graphics coordinate system in Spacewar!.

Armed with the new found knowledge of the rotation algorithm used by Spacewar! we used an excel spreadsheet to analyze the code in the outline compiler and track the values of the variables used by that portion of the code as the program was executed. After analyzing the code we realized that every octal digit in the outline’s encoding was specifying a source code section to execute, and each of these source code sections wrote new instructions into the memory. This newly generated code would later be executed and result in displaying the ships on the screen. These instructions updated the drawing location by moving it in one of 5 directions relative to the last point drawn. A sequence of such instructions would result in the tracing of the outline of a spaceship in the game.

On gathering all this knowledge we decided to trace out the spaceships ourselves following the code in the program. In doing so we were disappointed mid-way as it just seemed to produce a straight line until it finally deviated and produces a diagonal line on the screen. After going through the entire code step by step we realized that only half of each spaceship’s outline was being encoded and that the other half was drawn by reflecting the first half.

4. SPACEWAR! NIGHT SKY

After understanding the drawing of spaceships in Spacewar! we now moved our attention to the star field background of the game and how it was drawn and controlled. The Spacewar! star field is a view of the night sky as seen from MIT. The original code was called Expensive Planetarium and was written by Peter Sampson between 1962 and 1963[4]. Originally the code was independent of the game but was later included in the game to replace the random dots drawn as stars. The end section of the Spacewar! source code was entirely made up of repeated instructions which called a macro "mark". A section of this code is shown in the image below.


```
1j,
        mark 1537, 371          /87 taur, aldebaran
        mark 1762, -189         /19 orio, rigel

        mark 2280, -377         /9 cmaj, sirius
        mark 2583, 125          /25 cmin, procyon
        mark 3431, 283          /32 leon, regulus
        mark 4551, -242         /67 virg, spica
        mark 4842, 448          /16 boot, arcturus
1q,
        mark 6747, 196          /53 aqil, altair

2j,
        mark 1819, 143          /24 orio, bellatrix
        mark 1884, -29          /46 orio
        mark 1980, -46          /50 orio
        mark 1951, -221         /53 orio
        mark 2152, -407         / 2 cmaj
        mark 2230, 375          /24 gemi
        mark 3201, -187         /30 hyda, alphard
        mark 4005, 344          /94 leon, denebola
```

The Spacewar! programmers came up with a macro (function in MACRO) called mark which seemed to convert the star coordinates into a format that makes them easier to process for displaying. The X and Y coordinates of a star were stored in the form of a table in order of their X coordinates. Stars of similar brightness were grouped together in the table. Each call to the macro mark was replaced with the inline code, substituting the arguments for the X and Y coordinates for each star. The macro mark then multiplies the Y coordinate by 2^8, which is equivalent to bit-shifting it left by 8 bits, but will be performed before run time. Also the X coordinate is subtracted from 8192.

Looking at the values of the X and Y coordinates it came to our notice that while the Y coordinates were in the range of -512 to +512 the X coordinates had a range from 0 to 8192. This seemed to be an odd manipulation till we analyzed the macro "dislis". The dislis macro takes the X and Y coordinates of a star after they have been put through the macro mark and uses them to display a star. The Y coordinate is bit shifted by 8 as the display instruction of the PDP-1 looks in the 0-9 bits of the accumulator and the IO

Register for the X and Y coordinates respectively. By bit shifting the Y coordinate it was placed in the correct location needed by the display instruction.

The X coordinate was a little more tricky since certain stars would not be displayed depending on which region of the night sky was visible. As mentioned before the stars were ordered according to their X coordinates and the code went through the stars in the same order. The source code kept track of where the right margin of the screen was in respect to the world space and moved it along in the game loop. After subtracting the X coordinate from 8192 the dislis macro examined the X coordinate to see whether it was in the viewable portion of the screen. If it was in the viewable portion it converted the X coordinate from the world space to the screen space in and placed it in the 0-9 bits of the accumulator. Upon encountering a star whose X coordinate was not in the screen space it exited the loop and moved to the next block of stars. The ordering of stars in this manner ensured that unnecessary computation was avoided.

After figuring out how the star field was drawn in the game we used the information to implement it in the game we were creating on the Arduino/Gameduino combination. The device we used did not have the same screen size, but after obtaining the original coordinate system it was easy to convert it to make it work for the display that we were going to be using.

5. THE ARDUINO/GAMEDUINO IMPLEMENTATION

After figuring out the manner in which the star field and spaceships were drawn in Spacewar! we sat down to discuss as to how we wanted to implement the same in our version of the game. We first had to go through the Arduino and Gameduino reference manuals in order to understand the workings and capabilities of the two devices[5][6]. The Gameduino itself has the technical capabilities of a scrolling background using a bitmap image[7] and also the possibility of using sprites and the Gameduino sprite collision mechanism for detecting collisions between the torpedos and the spaceships[8]. The decision to be made was whether to stay true to the original code or use the technical capabilities offered by the Gameduino and make the code easier to implement. Also the processing capabilities of the Arduino were one of the main considerations while deciding the manner in which we would implement the star field and spaceships.

Regarding the star field we decided that we would stay true to the source code and display each star individually rather than use the scrolling background offered by the Gameduino. We did this by creating our own artificial margins to set the portion of the screen that is viewable and individually checked whether a star was present in this region and print it if it was. The Gameduino did not offer any brightness controls so we had to use different colored sprite to implement the brightness of the various stars, white being the brightest and grey being the dullest.

Each individual star printed on the screen was designated its own individual sprite control word to control the star and its movement across the screen. This led to a few problems as we did not have enough sprite control words to print all the stars that were in the viewable region at a given point of time. To solve this problem we extended the star field space by a factor of 2 so as to have fewer stars being displayed at a single point of time. This worked well and also turned out to be a much better representation of the original star field as there were not many stars displayed at a single point of time in the original game either.

The next decision which we had to make was whether to implement the spaceships using the original code, i.e. using vector graphics techniques, or use the sprite capabilities of the

Gameduino. The problem of implementing the spaceships using vector graphics techniques was that it would take up a lot of the computational capabilities offered by the Arduino and would no longer be efficient. Also the collision detection mechanism would have to be hand written and would be equally computationally heavy. Besides one of the main advantages of using the Gameduino is its pixel perfect collision detection mechanism which we wanted to incorporate into the final version of the game. Therefore, we decided to use two individual sprites for each of the spaceships and use the collision detection mechanism of the Gameduino for detecting collisions.

The second problem was the rotation of the spaceships. The Gameduino has an in-built sprite rotation algorithm which is very effective and precise [9]. The problem though was that we were using two sprites as one spaceship and the rotation algorithm rotated one sprite at a time. This led us to come up with an algorithm which used the rotation mechanism of the Gameduino for the first sprite and compute an offset to place the second sprite and rotate it accordingly. Also in the original game the spaceships rotated around their noses whereas the Gameduino rotation algorithm rotated the spaceships around their centers. This problem was overcome by using two individual sprites to define each spaceship rather than a single sprite for each of the spaceship. In this manner we could draw them together and rotate the two sprites about their centers to create the illusion of them rotating around their noses.

6. CONCLUSION

The Arduino/Gameduino implementation of Spacewar! on the whole has been a fascinating experience. The programming style in the original source code is one that we haven't seen before and portrays the difficulties faced by the programmers in the early days of computing.

There is still a lot of work yet to be done in terms of going through the source code and figuring out the different mechanics the original game used. There are various features in the game which can be turned on and off by controlling the switches of the PDP-1. For example, the gravity of the star can be turned off along with the star field.

The game as a whole still need to be implemented but the fact that we have understood some of the most important characteristics of the original game such as the drawing of spaceships and stars is very important in the implementation of the game. The design decisions taken by the team in regard to the implementation for the various game mechanics has also made the process of implementing the game faster and easier.

On the whole I have got a lot of hands on experience on coding with the Arduino/Gameduino in the duration of this project. I have also learnt the extensive capabilities of the Arduino/Gameduino hardware. The analyzing of the original source code has helped me immensely to understand the working of a computer from a machine level point of view and the difficulties as well as advantages of assembly code. Finally, the completion of the project will be a testament to the fact that gaming has been and will always be a part of MIT's culture.

7. REFERENCES

1. PDP-1 and MACRO manuals available through the Bitsaver's archive, http://www.bitsavers.org/pdf/dec/pdp1/
2. Multiple Emulator Super System (MESS) to emulate the PDP-1, http://www.mess.org/
3. Spacewar.rim file to be loaded on the PDP-1 to emulate the game, http://www.computerspacefan.com/SpaceWarSim.htm
4. Expensive Planetarium source code, http://en.wikipedia.org/wiki/Expensive_Planetarium
5. Arduino technical reference, http://www.arduino.cc/
6. Gameduino technical reference, http://excamera.com/sphinx/gameduino/
7. Gameduino scrolling background, http://excamera.com/sphinx/gameduino/samples/scroll/index.html
8. Gameduino sprite collision mechanism, http://excamera.com/sphinx/gameduino/samples/collision/index.html
9. Gameduino sprite rotation algorithm, http://excamera.com/sphinx/gameduino/samples/rotate/index.html