Stick Hero

LCOM Project

Daniel Filipe Santos Marques, up201503822

João Francisco Barreiros de Almeida, up201505866

01/01/2017

Index

User Instructions

Before starting 'Stick Hero' make sure you know which relative path you are running it from, since it is asked of the user to input this path as a mandatory argument of *service run*, otherwise the project will not run, example of path *"/home/lcom/proj"*.

```
# service run `pwd`/proj -args "/home/lcom/proj/"
```

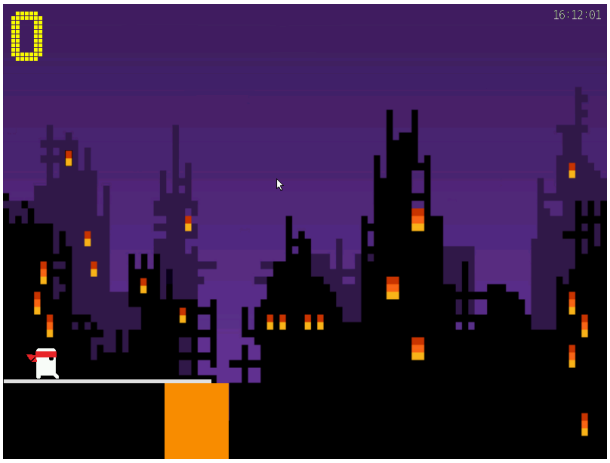
Picture 1: Service run example

When 'Stick Hero' starts, it displays a simple menu with two options: *Play* and *Exit*. The *Play* option shows a yellow color around the button, meaning it's selected (by default). An option can be selected and pressed with not only the mouse movement and left button click, but also with the up/down arrow keys to select and enter key to proceed.

Picture 2: Main menu

If the *Play* option is pressed the game starts! Our 'Hero' appears in a platform and he's ready for the user to hold the mouse's left button making him grow his stick to the perfect size to reach the next platform, without exaggerating and making him go into the abyss, which would also be his faith had he grown it smaller than it should.

Picture 3: Growing Hero's stick

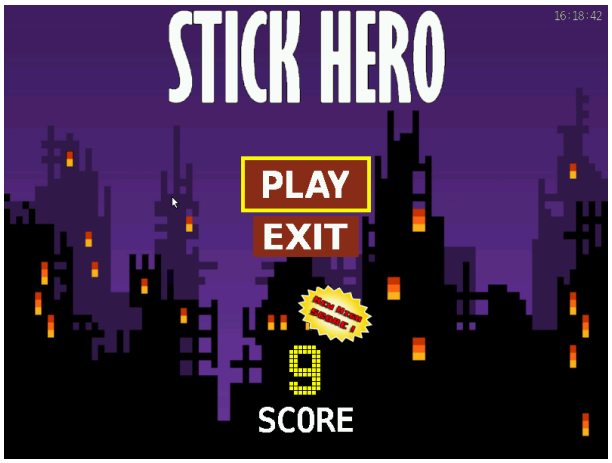


Picture 4: Hero walking

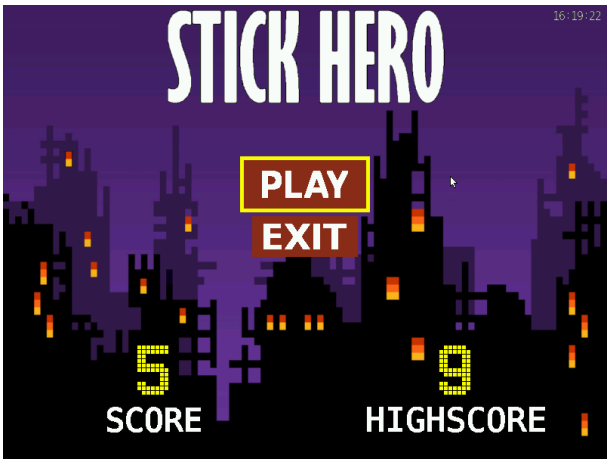
For every platform reached, a point is given to the user and the game ends when the platform is not reached successfully. This dynamic makes the game, theoretically, infinite.

Picture 5: Hero reached a platform

When the player eventually loses, his final score is displayed in addition to the highest score. Next, it's up to the user to keep playing or close the game.



Picture 6: New High Score



Picture 7: Normal Score

Project Status

This project takes advantage of Minix 3's I/O functionalities and documentation to implement the following I/O devices: timer 0, keyboard, mouse, video card and RTC.

Device	Functionality	Type
Timer 0	Frame rate control	Interrupt
Keyboard	Menu navigation	Interrupt
Mouse	Menu navigation and gameplay	Interrupt
Video Card	Screen display	Polling
RTC	Calculate elapsed time and local time display	Polling

- Timer 0

The timer 0 was implemented in this project and designed to receive its interrupts. In the function runGame, there's a loop that receives the interrupts from the various I/O devices, including timer 0's. It's main usage in this project was to assure consistent frame rate and animations, as well as a way to know when to get time/date information from the RTC.

- Keyboard

The keyboard was implemented in this project and designed to receive its interrupts in the so-called driver_receive loop (in the runGame function). The purpose of this implementation was to have a way to navigate through the menu options.

Therefore, the menuHandler function would be called every time the keyboard sent interrupts and the game status was *Main_Menu* or *Replay_Menu*. This function would then call the kbdReadByte function which reads the scan code from the keyboard output buffer. This code is then processed by the menuHandler which would change the selected option in case the scan code belonged to a up/down arrow key or it would return a different integer, meaning the scan code belonged to the enter key. As that happens, the runGame function decides what to do next, based on what option was selected before the enter key was pressed.

- Mouse

The mouse was implemented in this project and designed to receive its interrupts in the so-called driver_receive loop (in the runGame function). The main purpose of this implementation was to have a way for the user to grow the hero's stick. In addition, we built a custom cursor to make it possible to use the mouse a way to navigate through the menu options.

Every time there was an interrupt, a byte was read by the readMouseByte function. When a packet was fully read, we could handle it.

First, the bit 0 of the first byte of the packet was checked as it holds the information regarding the mouse's left button. If set, various actions could happen, depending on the game status. If in a menu, then it would be checked if the cursor was inside an option. Else, if it was in game, then it would be used to dictate when to start the stick growing. As soon as it's released (this bit returns to zero) the stick stops growing and falls.

Secondly, the 'x' and 'y' movements are taken care of, and appropriately added/subtracted to the cursor's coordinates.

Finally, the cursor is drawn again in the updated coordinates.

- Video Card

The video card was implemented in this project and set to mode 0x114. This mode uses 16bit 800x600 resolution with direct color mode.

To ensure a smooth gameplay, we implemented a double-buffer to where we paint every object and then simply copy it to the video memory. In order to achieve this, we have created the function updateBuffer, which based on the parameters it receives paints different objects on the buffer.

- RTC

The RTC was implemented in this project. Its main functionality is to get the local time, which is then displayed onto the screen in a user-friendly way. That way the player knows when it's time to leave the game!

As the RTC is set by default to BCD mode and converting the data from the RTC to binary every time we needed to update the time was time consuming and could affect performance, it was decided to create a function (RTC_SetBinary) which sets the RTC mode to binary.

The time is read once a second (thanks to timer 0's interrupts) using a function (RTC_GetTime). In this function, we disable the RTC interrupts in the beginning and re-enable them at the end. Every time we read something from the RTC we make sure the UIP flag is zero, otherwise we would try to read while it's updating.

In addition, the enable/disable RTC interrupts used are programmed in assembly, using the 'sti' and 'cli' instructions, respectively.

Code Organization/Structure

- game (35%)

In the main function, the game is set up and then run, calling setGame and runGame, respectively. The first one, loads all the bitmaps and sets up the default values for the game objects. The second one, subscribes to the I/O devices interrupts and handles them. Because different situations generate different actions in the game, there was a need to implement a state machine. This way an interrupt from the same device in different stages of the game would be handled differently and accordingly.

The code for each interrupt was kept to the shortest possible, making use of handler functions. This improved readability and shortened the driver_receive loop.

- graphics (30%)

Each time there is the need to move something, be it mouse, platforms or the ninja, the graphics module is used as all the animations are handled in here. To assist in this task there are also some auxiliary functions inside this module, mostly used to either ensure that the high-score is written in the correct spot or to correct the Bitmap images read.

Most of the functions developed in this module behave differently based on their parameters. This allowed us to cut back on the amount of code we had to write, consecutively we believe we maintained a reasonable amount of code that does not make it too tiresome to read.

- menus (15%)

This module allows the user to navigate through the menus using the keyboard. Because of consistency reasons, it was decided to use the break codes to select the options. This way, everything related with the menus would start and end in the game status reserved for the menus.

- files (15%)

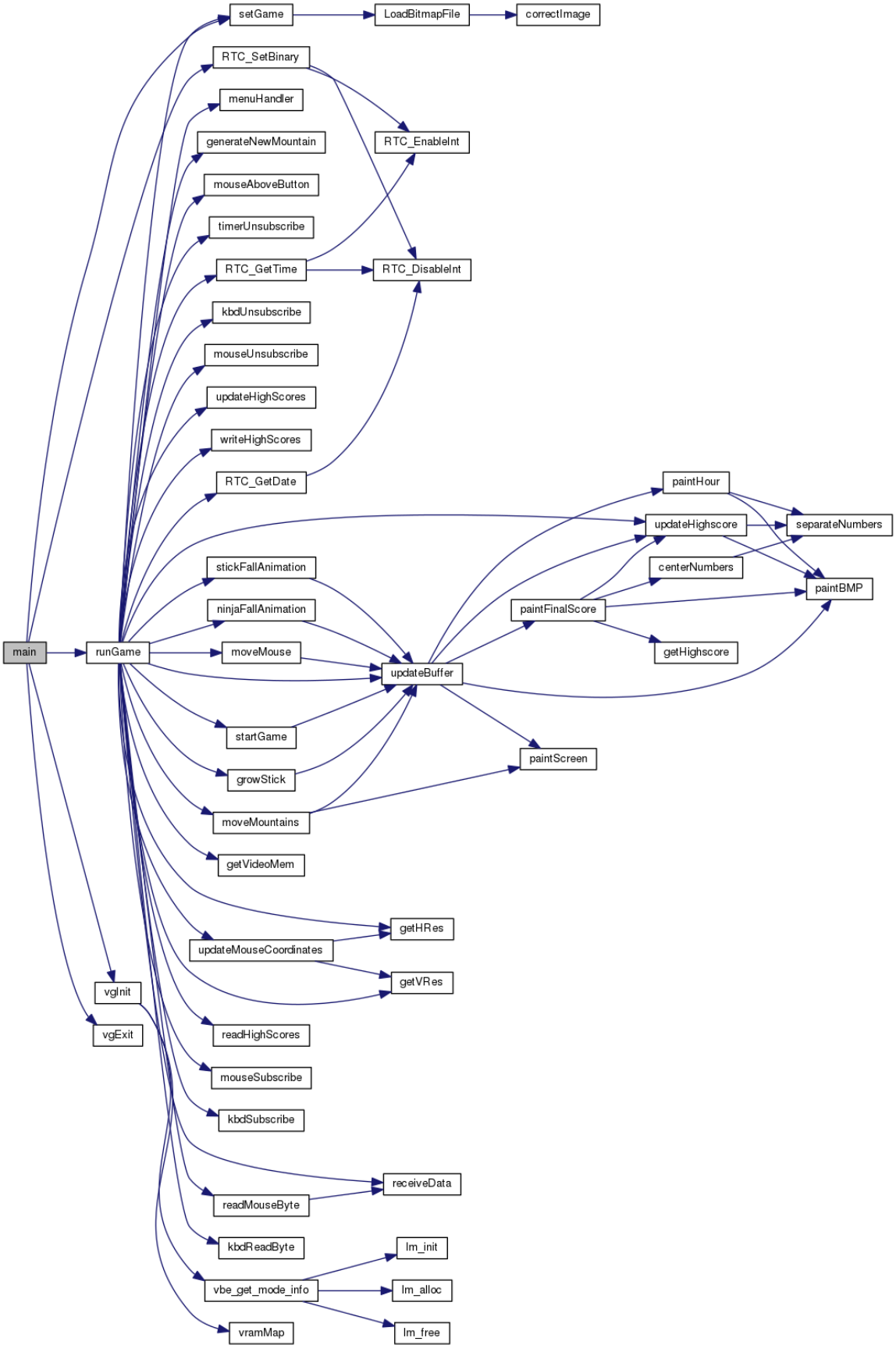
This module was necessary to load, update and save the high scores achieved in the game. It would be very frustrating if a very high score was reached and as soon as the game exited, the high score would be lost!

The file operations were done using the standard C file functions and data types. The file extension used was .txt and if there was no such file yet, a new one would be created. Although the functions are prepared to take any number of high scores, it was decided to keep only the highest of them all, as the resolution was too small to showcase more of them.

- vbe (5%)

This module was completely imported from the lab5 assignment containing only the function vbe_get_mode_info, which is used to find out the capabilities of the graphics system in which the project is running.

- Function Call Graphic



Implementation Details

For the sake of legibility of code and saving some heavy typing, we decided to create seven structs that were all packed inside a single struct - *game_t*. This struct contains all the relevant information to the project, thus allowing us to effortlessly pass relevant information to functions of the project.

As said before, there are two state machines in the project. They help determine what to do based on the current status. The first one is the *game_status_t* and holds the stage where the game is. The seconds is the *options_t* and holds the menu option currently selected. Another big advantage of using state machines is that if it were needed to add, for example, another menu option, there was no need no re-do all the code. It would only be needed to add it to the *options_t* enum and make a handler for that option.

To add more fun to the project we decided to include a high-score, that was implemented using I/O operations on .txt files. As it stands, the functions used to do that can easily be reused for some other project that involves reading and writing multiple numbers to a text file, as we made sure that it would not just simply input and output a single number.

The RTC was an I/O device that initially was not planned to be used in this project. However, there was a necessity for it and, therefore, it was implemented. To enable/disable the RTC interrupts, we've developed two functions in C which only contained assembly code. To be precise, the 'sti' and 'cli' instructions.

The bitmaps were, mainly, done from scratch by the group members using an image editor. As most of those were very small (but detailed), they were done pixel by pixel. Likewise, the code to load the bitmap images information into an intuitive data structure was done mostly from scratch, based on a Stack Overflow answer[1]. In spite of this answer containing multiple errors on the code and some ambiguity, we were able to reverse engineer it and modify it as we pleased.

All the graphics related functions where created to be easily reused on a different implementation of the graphics module. This is achieved by not having the functions paint to a specific buffer, but to a buffer passed on as a parameter, which allows to easily integrate them into a single, double or even triple buffering implementation.

1 http://stackoverflow.com/a/14279511

Retrospect

Now that we have completed this project there are some things we would like to have done differently both during the development process and the planning phase.

For starters, we believe none of us ever really got used to working with Subversion which led to some nuisances where the work of a colleague was completely erased from the most recent revision, as well as every time we needed to import something to the repository, we had to do it multiple times because more often than not we would import it wrongly.

Another setback we had was “swimming against the tide”, regarding the aspect of the bitmap images. Whereas most of our colleagues used a preexisting library, we created our own way of loading and painting a Bitmap, which on one hand allowed us to get a better grasp of how image processing is done, but on the other hand was very time costly which means we could not implement everything we planned for, namely a more complete high-score table and the ability to make the ninja jump based on the mouse gesture performed by the user.

Known-Bugs

Currently we are aware of two possible bugs in the final version of the project.

One of them is related to the graphics module, more specifically the part where the ninja walks from one platform to another. From the testing we've done, we believe it happens whenever the size of the stick is slightly bigger than the next platform, we believe only by 1 or 2 pixels, when that happens the ninja is not falling has it was supposed to but instead it is considered the ninja landed on the platform correctly, which was not the case.

The other we are not sure if it's a false positive or not since we have only seen it once and never again. Very recently as we were testing the game, there was the case where the user lost the first round of the game, then clicked *Play* again and neither the ninja or the first platform was rendered. We do not know why it happened or under which circumstances it has happened since it was very spontaneous.

Conclusions

Despite the setbacks we had on the development process and not being able to fully accomplish what we had envisioned, we are satisfied with the work we have done and are confident we have created an enjoyable game with the required features.

Regarding the “Laboratório de Computadores” course unit we have a few aspects we would like to mention.

On the negative side, we felt like we should have been given more time for the labs work, especially the students whose class is in the beginning of the week, since they have less time between the theoretical class and the labs deadline than the other students whose class is in the end of the week. As suggestion, we think all students should have the same deadline.

On the positive side, we think LCOM was an interesting course unit. We were able to take our idea of interrupt/polling given by MPCP and take it to the next level. It was interesting to see how the hardware and software would combine and how the engineers and programmers that designed the devices and OS’s made programming possible through documentation. In other words, we wouldn’t feel completely lost if a documentation of a new operating system was given and we had to implement the different I/O devices into an application.

Appendix

To run the game, in addition to the standard *'make'* in the "…/src/" folder, the *'service run <path>/proj'* will need an extra argument in order to avoid absolute paths. Therefore, the user should write in front of this *'-args "<path>"'*. Also, a lot of functions are privileged and, therefore, require the config file attached.