Space Invaders – Lite Edition

Henry Bishop
Department of Electrical and Computer Engineering
University of Virginia
Charlottesville, VA, USA
hlb9ce@virginia.edu

Bradley Hallier
Deparment of Electrical and Computer Engineering
University of Virginia, Charlottesville, VA, USA
bth3fb@virginia.edu

Abu Hasnat Mohammad Rubaiyat
Department of Electrical and Computer Engineering
University of Virginia
Charlottesville, VA, USA
ar3fx@virginia.edu

***Abstract*— As a final project, the RTOS created in previous mini-projects was extended to create a Space Invaders mini game. The game implemented many features of the original Space Invaders game, including moving blocks of aliens, a movable player, fireable projectiles, and depletable shields. Visual elements were represented by bitmaps similar to those in the original game. Player movement was controlled by the tilt of the board measured by an accelerometer. The player wins the game by defeating all aliens, and loses if the aliens reach the shields or loses all life due to alien bullet contacts.**

Keywords—space invaders, TM4C123G, MKII boosterpack, RTOS

I. Introduction

Countless video games have emerged over the past several decades spanning from small handheld bitmap adventures to wearable and immersive VR experiences. The motivation for this Space Invaders Lite Edition project was to recreate a throwback classic arcade game, Space Invaders, with a TI TM4C123G launchpad and MKII boosterpack to fulfill the requirements for the final project of the 2017 Advanced Embedded Systems course at UVa.

Previously, the group designed a different game that required the player to move a cursor around the screen which then interacted with cubes also on the display. This game was the culmination of a set of "mini projects" as well as homeworks that were periodically assigned throughout the course. These assignments required students to become familiar with important real time operating system (RTOS) concepts by applying them in real software implementations while also using those implementations to build up the cube game. With the background of designing the game and understanding the RTOS that had been implemented, Team Delta was able to take certain pieces of this software to use in its Space Invaders Lite game.

The general theme behind Space Invaders is revealed in denotation of the game title. The player moves their spaceship left and right at the bottom of the screen behind a set of shields which protect the spaceship from alien invaders' attacks. The player and aliens fire back at one another attempting to beat the other. The aliens formation is in a block pattern that gradually moves from top to bottom on the screen. The aliens can win by either depleting the player of all their life points or by invading Earth, or in other words getting past the shields. The implemented form of this game is very similar. Arcade style graphics and movements are used to emulate the feel of the original experience. The objectives are for the player to continuously shoot back at the attacking aliens without getting killed or allowing the aliens to get to the line of shields at the bottom of the screen. Points are accumulated by the player based on how many aliens they kill and life points are based on how many times aliens have shot the player's spaceship.

By providing a common software framework, as developed through the many homework and mini project assignments, Team Delta demonstrated that the extensible RTOS could be implemented for entirely other games, and very popular ones at that, without an immense amount of extra work.

II. System Description

As previously mentioned, the software developed to run the Space Invaders game utilized previously created code, but there was a substantial amount of new software written because the entire game behavior was different from the original cube game. The fundamental RTOS capabilities were left intact as well as the set of threads dealing with exchanging and manipulating player position information. Several new threads that were created for the purpose of this new game: alienThread, collisionThread, bulletThread, and a revised Display thread. The functionality of the boosterpack's Button 2 interrupt was also changed. The high level description of these threads are given below and will reference the data flow diagram shown in Figure 1.

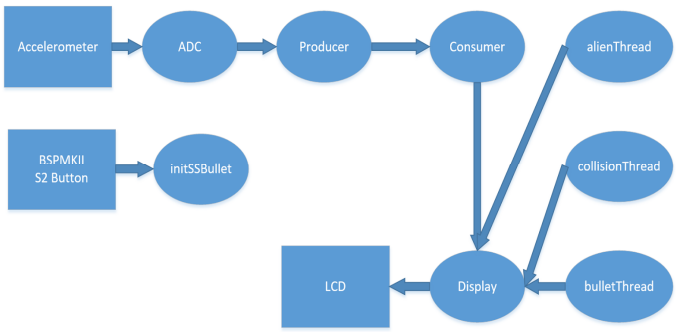


Fig. 1. Data flow diagram of the game illustrating the interactions among the different threads

On the system level, there are quite a few things occurring "at the same time" due to the RTOS's capability to perform thread scheduling. The player's position that is determined by the left right tilt of the accelerometer is initially read by the ADC, which was repurposed from its original use of retrieving information based on the joystick available on the boosterpack. This information is passed on to a thread that appropriately scales the data so it can be used to display position on the LCD screen. This thread interfaces to the Consumer thread which interacts with the Producer via a data FIFO with player position information in it. This thread assigns the player's position to the actual object representing the player. This thread behavior is shown in Diagram 1 at the top left.

In parallel, the three main threads new to this framework, shown on the right side of Diagram 1, control the behavior of the aliens, bullets, as well as results of collisions between the different items in the game. This will be elaborated upon later. Each of these threads communicate to the common Display thread as well. By allowing the Display thread to be the only process that utilizes the LCD resource, there isn't need for mutual exclusion principles. Also, deadlock no longer remains an issue because LCD characters won't be overlapping as would be the case in the previous cube game when two cubes could try to gain access to each other's location simultaneously.

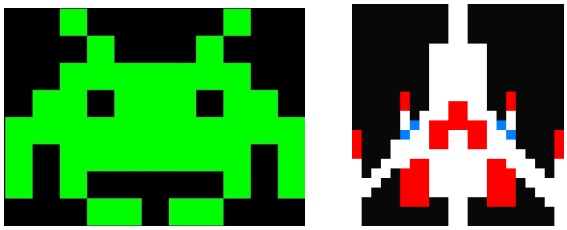
III. Design and Implementation

A. *Game Characters Design*

From the system description explained in previous section, we can have some ideas about the characters used in this game. The main characters of this game are spaceship (controlled by the player), aliens, and shields. Beside these, two types of bullets have been used generated from spaceship and aliens respectively. Figure 2 shows a screenshot of the game illustrating the major characters, i.e. spaceship, three rows of aliens, and two shields (another one has already been destroyed by the aliens).

Fig. 2. A screenshot of the game illustrating spaceship, aliens, and two shields

For creating the aliens and spaceship, bitmap images have been used. First, the images are converted to the corresponding hex codes, i.e. representation of each pixel in hexa number (here, 16 bits for each pixel according to the LCD functionality) using an online tool [1]. Size of each alien is 10x10 (i.e. 10 pixels in each horizontal and vertical directions), and a 15x15 image has been used for spaceship. Figure 3 illustrates the images used to represent the alien and spaceship. The shields are drawn as 8x16 rectangular boxes. Initial color of the shields is green, but each time a shield gets hit by a bullet it changes its color. Each shield can take two hits (changes its color from green to yellow and then red) and after third hit it is destroyed. The bullets are 3x2 boxes. Green has been used for spaceship bullets, and the bullets shot by aliens are of orange color.



(a) (b)

Fig. 3. Images of the two major characters of the game, (a) alien [2], (b) spaceship [3]

B. *Controlling Spaceship*

For the spaceship, we used a 15x15 bitmap image shown in Figure 3(b). Only horizontal movement has been implemented for the controls of the player's icon, similar to the original Space Invaders. As described before, the accelerometer was used to control the movement of this spaceship so that the player can move the spaceship by tilting the device left and right. Since use of the joystick is kept out of this game, the "joystick.c" file is repurposed to support the gathering of data from the accelerometer. By using the sample sequencer 2 (SS2) along with editing some of the control registers meant for manipulating the ADC, retrieving data from the accelerometer could be accomplished with ease. The data coming from the ADC is then manipulated in the Producer function. The scaling factor originally in there was to support the joystick but was altered to support the data range supplied by the ADC. Then similarly to how the player was controlled in the previous game, the software FIFO feeds data to the Consumer thread ultimately controlling the character's movement on the screen, once Display updates.

C. *Controlling Alien Block*

Unlike the spaceship (controlled by the player), aliens should move independently. A separate thread has been implemented to initiate the aliens and control their movements. A block of 18 aliens (6 in each of three rows) has been generated at the beginning. The alien object is defined in 'Aliens.h' header file, which contains the variables for XY position of the corresponding alien, its immediate previous position. It also provides the information of whether the alien is active or not, and what is the score the player will get if he can kill this alien.

All three rows of aliens move sequentially. First, the lowest row of aliens starts moving horizontally, and remaining two rows follow. All the three rows continue moving horizontally till all of them reaches a sidewall, where all the aliens go one step downward and start moving horizontally towards opposite sidewall. This movement continues till any alien reaches the line of shields (i.e. till it invades), or the spaceship does not kill all of them. Figure 4 shows the directions followed by the alien block while moving. The broken line represents the shield line. The game will be over if any of the aliens can pass this line. The spaceship always stays below this line and shoots to the aliens. Before moving to the next position, the previous position of each alien has been stored to make sure display thread can erase the residue of the alien in previous location.

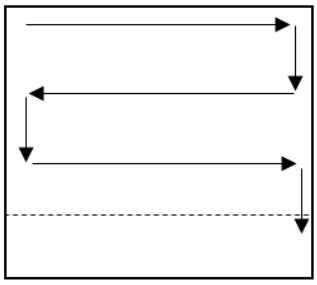


Fig. 4. Moving direction of the alien block (show by arrow signs), broken line represents the shield layer

D. Bullet Thread

Two types bullets need to be handled in this game - one from spaceship and another from alien block. Both bullets are initiated and controlled in the same thread. Alien bullets are generated after a certain interval (1 second). Timer2A has been used to calculate time. In this game, the initial position of the alien bullet has been set in such a way so that it is initiated from the lower middle point of the alien block. So, the initial position of the alien bullet is dependent on the position of the alien block at the time of shooting. Spaceship bullet has been generated each time the player pushes the S2 button. When S2 button is pushed, a flag is asserted and the bullet thread generates a spaceship bullet from the top middle point of the spaceship.

The motion plans for the bullets are simple. Spaceship bullet goes upward and the alien bullets travel in the opposite direction. So, only the vertical axis values are updated (incremented or decremented for alien or spaceship bullets respectively), horizontal axis values remain constant at the initial values. Similar to alien, immediate previous positions of the bullets are also stored to erase the bullets from previous location. The bullet continues to move in the same direction till it hits any object or goes out of screen, then it is destroyed.

E. Display Thread

The display thread handles the rendering of the game's visual elements on the LCD. The decision to move all visual rendering to one thread was made to help ensure the elements were displayed at the same time to prevent them from getting out of sync. The rendering order is depicted below in figure 5.

The display thread was initially very intensive because it redrew each element each time the thread ran. This required each element on the screen to be erased by drawing a background colored block over it and redrawn in its new position using the appropriate bitmap. Given the high number of elements on the screen, this process was far from instantaneous and caused elements to flash. The display thread was optimized by implementing a check before redrawing an element to avoid erasing and drawing it if its position had not changed. This check ended up improving the game's visuals and decreased unnecessary processing done in the thread, improving the game's performance overall.

Another issue the game had related to erasing the previously rendered visual during redraw. In some cases, the previously rendered visual would not erase or only partially erase. This left artifacts on the screen and led to a poor gameplay experience. The cause of the issue was determined to be the separation of the threads handling the positioning and visual rendering of elements. Because the operations were handled in separate threads, it was possible for the position to be updated multiple times between rendering, which meant that the previous position of the element could be different than the previously rendered position of the element. This was resolved by keeping track of the previously rendered position instead of the previous position of the element. A different solution could have utilized semaphores to ensure the previous position of an element never became out of sync with the previously rendered position, but this solution can have a negative impact on the overall game speed, which is important to keep consistent. The chosen solution allows the possibility of multiple position updates before the element is redrawn, but the processor load was low enough that this was not an issue.

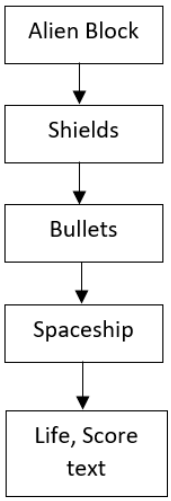


Fig. 5. The rendering order for elements in the display thread

F. Collision Detection Algorithm and Scoring

The collision detection in the final project needed an overhaul from the previous project because game elements no longer moved in large, predefined grid-like movements.

Instead of the play area being mapped to a grid with block occupancy, a system similar to hitbox collision detection needed to be implemented. The collision detection needed to support arbitrarily sized elements that moved in undefined patterns. The algorithm written for this project takes in information regarding two rectangular objects with associated positions as well as widths and lengths. A simple set of conditional statements allows for the easy detection of overlapping objects, which we regard as collisions. These statements are shown in Figure 6 below, captured from [4]. This figure was taken from a interactive GUI which demonstrates the working logic of the proposed algorithm by taking into account the upper left and bottom right corners of the two objects. A surprising feature of this algorithm is its speed with respect to scale. The group believed that because of the large number of items that needed to be checked for collision that there would be a significant increase in delay for the collisionThread to complete one pass. But because this algorithm on the onset just needs a few conditions checked, it keeps the function lightweight.

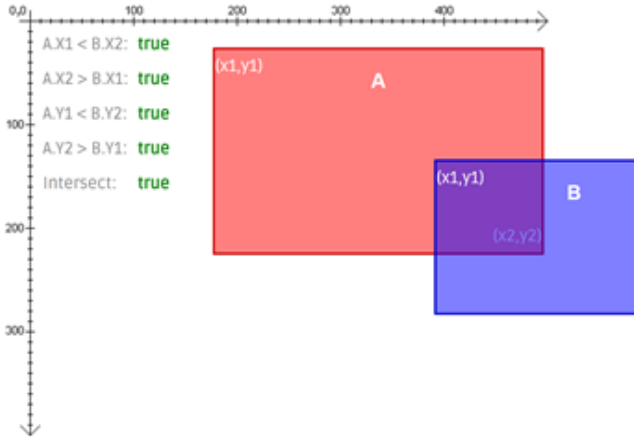


Fig. 6. Graphic depicting collision algorithm

If an object collision was detected, a flag indicating collision status was asserted, and the corresponding collision behavior occurred. If the collision had an impact on life or score, it was handled accordingly.

IV. Evaluation/Results

The final implementation of our game worked well. The aliens had the correct movements and behaviors as well as the shields and player. Collisions were correctly detected and elements had corresponding counters decremented as appropriate. The elements were removed correctly when they were ‘dead’. The bullets did not appear exactly as we intended, but were still visible on the screen and had the correct behavior when fired. While integrating our individual components we had a few parts break, but we were able to restore most of the functionality. One issue that we did not resolve in time was the end game screen. When our program reached the end game condition it entered a hard fault condition which froze the game before it could display the screen.

V. Lessons Learned

One of the most important lessons learned dealt with the importance of well-defined task definitions and a well-structured software architecture.

Implementing efficient thread code was another important lesson. If one thread is inefficient it can degrade the performance of the entire system by doing unnecessary processing. Optimizing a single thread can allow all the other threads to run more often improving software performance.

VI. Team Responsibilities

Team Delta comprises three members - Henry Bishop, Bradley Hallier, and Abu Hasnat Mohammad Rubaiyat. There are lots of scope of individual works in the implementation. We tried to equally divide these tasks among the three members. But the most challenging part was to integrate the individual tasks to get the fully functional game. In this stage, all three members worked together. The individual responsibilities can be summarized as following,

Henry:

- Implementing the accelerometer functionality and control the spaceship using it
- Shield design and functionality
- Development of the collision detection algorithm and functionality of collision thread
- Code integration
- Presentation slide preparation
- Report writing

Bradley:

- Display thread functionality
- Threads optimization
- Scoring, Life calculation, and Game over functionalities
- Collision thread functionality
- Code integration
- Report writing

Hasnat:

- Designing the characters (aliens, spaceship)
- Display thread functionality
- Alien and Bullet thread functionality
- Collision thread functionality
- Code integration
- Report writing

VII. Conclusion

This Space Invaders game utilized a previously developed RTOS and game software framework to develop an entirely new game based on an old classic. By using the accelerometer, the player is able to interact a greater amount with the game itself providing a unique gaming experience. Realistic bitmaps representing the characters and comparable behaviors for all the elements of the game to the original Space Invaders is meant to give the user a fun, blast from the past experience.

Overall, Team Delta was able to show that a well-developed RTOS and basic gaming framework allows for robust implementation of a new style of game only limited by the designers' imaginations.

ACKNOWLEDGMENT

This work has been done under the course project of the course *Advanced Embedded Systems*. We are very grateful to the course instructor Dr. Homa Alemzadeh for her support and guidance.

REFERENCES

[1] http://www.digole.com/tools/PicturetoC_Hex_converter.php

[2] https://www.redbubble.com/people/crampsy/works/25363451-space-invaders-alien-sprite?p=laptop-skin

[3] http://piq.codeus.net/picture/379071/galaga_ship_

[4] https://silentmatt.com/rectangle-intersection/