Advanced Embedded – Spring 2023
Team Alpha

Chengyuan Cai, John Chrosniak, Max Dawkins, James O'Connell
{cc4fy,jlc9wr,mld6nh,jgo2nja}@virginia.edu

I. Introduction

Our team set out to create a cube crusher game using the knowledge and tools learned from class. We used the provided Texas Instrument Tiva launch pad [1] and BOOSTXL-EDUMKII [2] booster pack as our gaming device, and Keil [3] as our development platform. The main focus was placed on the usage of multiple threads and experiencing real-life applications of the concepts we have learned. We also explored the use of deadlock prevention and random generators in the field of game development.

The objective of the game is to hit all cubes that appear with the cross-hair before the cubes expire. A random number of cubes will spawn, move randomly around the screen, and eventually disappear if not hit with the cross-hair. If a cube is hit, the player's score increases. If a cube disappears before being hit, the player's life decreases. The game also has levels with increasing difficulty that will automatically trigger as the player's score increases. Lastly, a high score is saved to benchmark the player's performance as they continue playing.

II. System Description

The system's inputs, processes, and outputs are summarized in Figure 1. The game starts out with a screen indicating that the player should press the button SW1 to start gameplay. Upon pressing, Task 7 runs and initiates all other system tasks. The joystick input is continuously read and values are placed into a FIFO by the producer in Task 1. The consumer in Task 2 reads these values and updates the LCD appropriately. The player's score, lives, and level are continuously updated by Task 3. In the event the player wishes to restart the game, they can push button SW2 to initiate Task 4 and restart the game. The random number generator is created in Task 5 for later use with cube generation and movement in Task 6.

III. Design and Implementation

A. Cross-hair & Panel Display

For the cross-hair and joystick section, we simply migrated the work done in Mini Project 1 to the final project GitHub repository and removed the unnecessary relics of the Mini Project. Looking at past examples we have found that many teams had games that were easy to beat because having a fast cross-hair speed allowed them to cover a lot of ground in a short amount of time. Consequently, we made the cross-hair move slower in an attempt to increase the difficulty of the game.

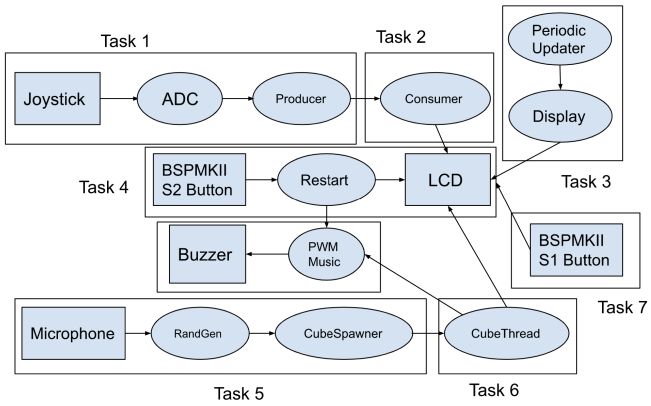

Fig. 1. A block diagram of the game system. Hardware inputs and outputs are denoted with boxes, while software processes are denoted with ovals.

The Panel Display simply consisted of displaying the score and life of the player. Both were declared as global variables. The score variable increases when the cross-hair overlaps with a cube, and the life variable decreases when a cube disappears without being hit. Because multiple threads access these variables, a semaphore was used for updating the life and the score. Lastly, the game over screen appears when a player's life reaches zero.

B. Random Number Generator

The random number generator was written using the C standard library's pseudo-random number generator and the microphone on the BoosterPack. The microphone was used as a source of entropy for setting the seed for the pseudo-random number generator to ensure that the game sequence would not be identical every time. This specific source of entropy was chosen because audio is inherently a varying signal because sound travels as waves. Figure 2 shows a typical recording of an audio signal and the signal of a single tone. This results in a voltage signal that varies significantly over time and thus the seed has a substantial amount of entropy.

To accomplish this, the CPU pin attached to the microphone was configured to be an analog input and the ADC channel used for this pin was set to use Sample Sequencer 3. This Sample Sequencer was chosen because it only records a single sample, which is desired for setting the random seed because there is more entropy associated with a single microphone reading. When the board is initialized, a single microphone

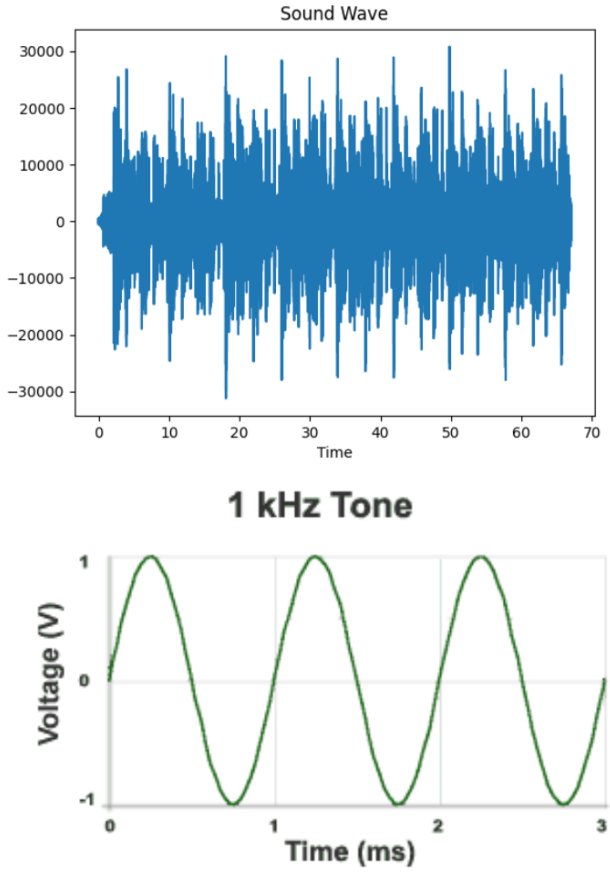

Fig. 2. Typical amplitude of an audio signal over time (top) [4] and the voltage signal for a single tone (bottom). The typical amplitude provides an accurate representation of the randomness the microphone can provide, while the voltage signal for a single tone showcases this method will work in the event the player is sitting in a completely silent room.

sample is recorded and this sample (a value between 0 and 4095) is used as the random seed.

C. Cube Generation & Motion

The cube generation was implemented as a single foreground thread that creates new foreground threads for each cube. The generation thread, called ”CubeSpawner”, waited while cooperatively suspending execution until no cubes existed, and then created a random number of threads using the random number generator. Each of the random number of new threads, called ”CubeThreads”, are identical, and they each handle the interaction and motion of one cube.

A C struct was added to contain information about a cube, including its position, size, direction of motion, whether or not it is active, and a semaphore for resource sharing. When a CubeThread is created, it finds the first available cube in an array of cube structs and sets its position to a random location. There are 36 possible positions a cube can have on a 6x6 grid, and each location has a corresponding semaphore to prevent multiple cubes from moving into the same grid space. The CubeThread handles the motion of the cube by moving in its direction until it hits a wall or another cube and then moves in a new random open direction. The CubeThreads also check if the crosshair is intersecting with its cube, in which case the thread will increase the score and be killed. If the CubeThread has been running for a long enough period of time, it will instead decrease the life and be killed.

Before moving, increasing the score, decreasing lives, or activating a cube, the CubeThreads communicate using the corresponding binary semaphores. The CubeThreads wait on the semaphores until they are available, with the exception of the grid cell semaphores, for which they use the `OS_bTry` function. This function helps prevent deadlocks in certain cases, as explained in the Deadlock Prevention section.

D. Interactive Sound Effects

One of the design requirements for the game system is distinct sound effects that occur due to specific events in the game. To accomplish this, the piezo buzzer hardware on the MKII boosterpack and the PWM generator block on the TM4C123 were used. References to the datasheet revealed the port pin for the buzzer input to be J4.40. This pin is connected through junctions to the GPIO pin PF2 on the TM4C123 microcontroller.

The first step in the audio generation process was to configure the hardware appropriately. Data from the TM4C123 technical reference manual revealed that the port pin PF2 corresponds to the module 1 generator 3 output. Configuration was started by enabling the system clock to control the PWM generator using the run clock configuration (RCC) registers. Following this, the GPIO port F was configured for an alternate function, specifically the encoding (0x5) for the corresponding PWM control module. Finally, the count was disabled for the generator to ensure that the initial value could be loaded properly. The ”down count” mode was also set, along with the appropriate generator A.

With the hardware configured properly, the next step was to create a function that produced a specific note for a time duration, with a given frequency and time count input. This was accomplished by first dividing the system clock frequency by the desired note frequency, to yield the appropriate amount of clock ticks required to pass one period of the note. This value was then assigned to the LOAD register for the PWM generator. It is also worth noting that the duty cycle was automatically set to 50% for each note regardless of this frequency. This was accomplished by loading half the value of the LOAD register into the CMPA register. This value was chosen because signals with duty cycles other than 50% contain greater magnitudes of higher harmonics. These higher harmonics are not completely attenuated by the hardware’s reconstruction filter as the signal is converted from digital to analog form. This causes dissonant sounds which is not in the best interest of creating good game music.

To begin playing the sound, the ENABLE register is toggled, and the counter starts. The duration of the sound is controlled by a defined ”delay” function, that simply increments

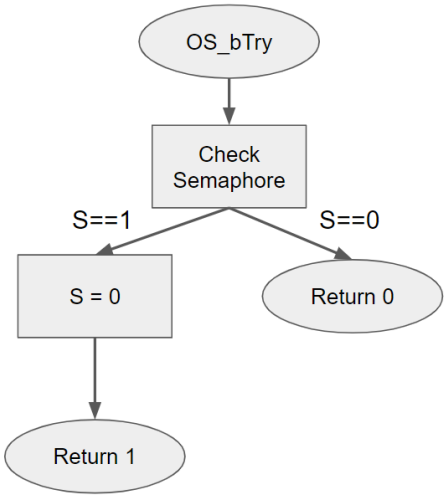

Fig. 3. Flowchart for the `OS_bTry` function. This function is used to prevent deadlock in the event two cubes try to take each other's occupied cells.

a counter with each tick of the CPU. It is set from the system clock to be incremented in milliseconds. Finally, to stop the sound from playing, the counter is once again disabled.

The last step in the interactive sound process is the creation of note patterns, or music. This was accomplished by making a function that takes an array of frequencies and times for input and outputs an array of sounds in that order. This function is called three times with three distinct inputs during gameplay. There is a game over sound that plays when a player's life reaches zero, a game startup sound that plays at the beginning of every game, and a scoring sound that plays when a player hits a cube. Our game start and game over sounds were inspired by the bootup and shutdown tunes provided by the Windows 7 operating system.

E. Deadlock Prevention

A new function `OS_bTry` for acquiring semaphores was written to prevent deadlock in the situation when two blocks collide traveling in opposite directions and get stuck waiting to take each other's space. Rather than busy-waiting or suspending in the event the desired semaphore is already held, the function returns an indication as to whether or not the semaphore was successfully acquired. A flowchart of this function is shown in Figure 3.

The value returned by `OS_bTry` is used to indicate if a cube has successfully found a space to move to. When this call succeeds, the cube can move to the desired position. When this call fails, the cube should pick another random action and try again.

F. Bonus Features

A few additional features were added to the project to improve the game-playing experience. First, levels with increasing difficulties were added to increase the stakes of the game as the player progresses. The difficulty increases from levels 1 through 10, with each level seeing cubes move

Fig. 4. End game display when a high score is (left) and is not (right) set.

5 milliseconds faster and expiring 200 milliseconds faster. This was implemented by adjusting global variables that each `CubeThread` uses when adjusting the position or visibility of cubes.

The game also records high scores over time to benchmark the player's performance. The end game screen will display the player's score and current high score after each game that does not surpass the high score. Whenever a new high score is set, a "New High Score!" message is displayed on the end game screen along with the player's score and the new high score. Both of these views are shown in Figure 4. A global variable was used to keep track of the high score over time. This value does not persist between power cycles of the board, but a non-volatile section of memory could be used to prevent the high score from resetting.

IV. Results & Evaluation

The cube crusher game was successfully implemented by the team using multi-threading. To make sure the operating system and game were functioning as planned, the game was played multiple times. For an accurate assessment of the technology, these games included a variety of death types and scores. These many situations were looked at to make sure the system would function effectively under a variety of edge conditions. The cross-hair caused the cubes to vanish as expected under all tested conditions, and the crosshair always stayed within the bounds of the screen. The final version of the view during gameplay is displayed in Figure 5.

V. Conclusion & Lessons Learned

This project was very insightful and provided a great learning experience in dealing with complex systems that rely on an RTOS. Managing the interactions between system components was more complex than anticipated, emphasizing the value and necessity of semaphores for access to shared resources and variables. Setting thread priorities was also a notable challenge. For example, if the threads responsible for moving the crosshair and cubes were set to a lower priority, the game lagged noticeably. There were also many instances when the crosshair would hit a cube but the cube did not register the hit and would continue moving. Setting appropriate

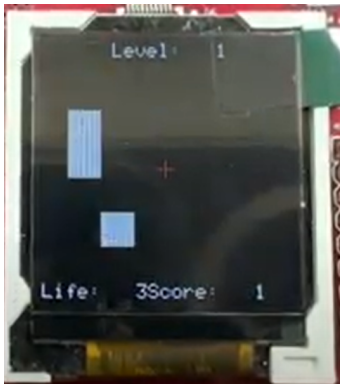

Fig. 5. Final version of the game display during game play. The cross-hair, cubes, life, level, and score are all displayed appropriately.

thread priorities helped address these problems. The team did not have prior experience with game development, so it was also interesting to learn the intricacies associated with video games. Although simple, this game provided an appreciation of what more complex video games must consider during the design process. Lastly, a good insight learned from the project was how to work as an effective software development team. Collaboration through Git and frequent communication helped the team meet deadlines and deliver a successful final product.

TEAM RESPONSIBILITIES

The responsibilities were delegated as follows:

- Chengyuan: Responsible for game life/scoring and cross-hair display
- Max: Responsible for cube generation and movement
- John: Responsible for random number generator, deadlock prevention, bonus features, and start/end game display
- James: Responsible for sound effects

REFERENCES

[1] Texas Instruments, *Tiva tm4c123gh6pm microcontroller*, https://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf, 2014.

[2] Texas Instruments, *Boostxl-edumkii educational boosterpack mkii*, https://www.ti.com/tool/BOOSTXL-EDUMKII, 2021.

[3] ARM, *Arm keil*, https://www.keil.com/, 2019.

[4] Geeks For Geeks, *Sound wave*, https://media.geeksforgeeks.org/wp-content/uploads/20200725122651/image121.png, Jan. 2022.