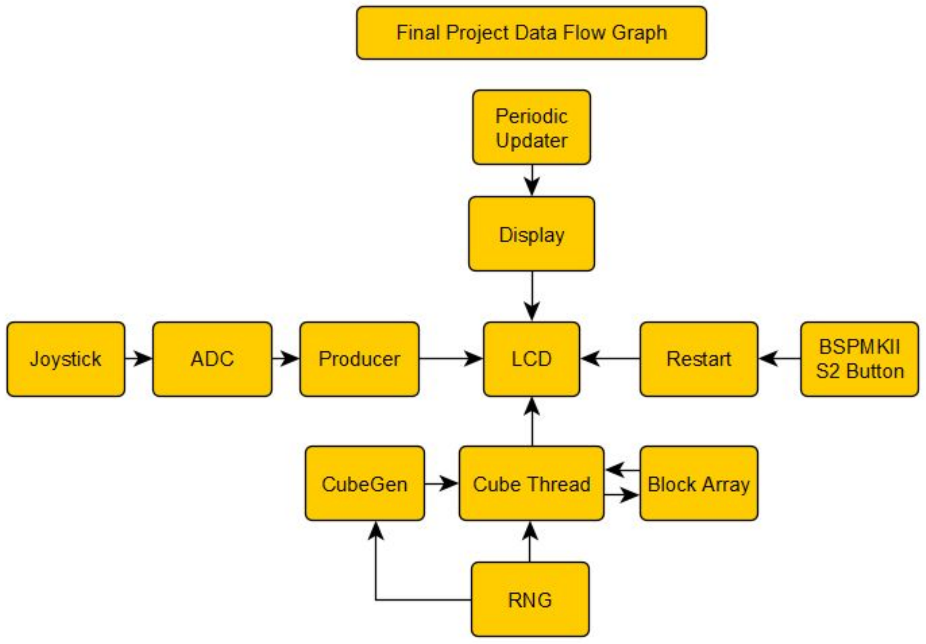
Final Project Part 1 Report

Jon Gustafson - jag9cj
Josh Neal - jdn3yb
Willliam Trace LaCour-wtl6c
Justin Vaughn Javier-jtj5gs

Team Responsibilities

Team members got together to work collectively during the entire design and implementation. Josh was responsible for transcribing the code as the team worked and focused on cube generation. Trace handled the editing and construction of the report as well as debugging. Jon was responsible for understanding and summarizing how the random number generator is used and implemented semaphore usage. Justin led the team in debugging techniques and focused on the LCD_Update function.

Flow Diagram



Random Number Generator

The code used to generate random numbers in this application involved the use of a linear feedback shift register(LFSR) as outlined by the provided Maxim Integrated link. A LFSR is used to pseudo randomly generate a sequence of numbers. Initially, the register is provided a seed value. This seed value is the first output generated by the register. On every clock cycle, the system will use a mask to select specific output bits. These bits are then exclusively ored with each other and the result is fed back into the register. The register then shifts all bits over by 1 position. During this shifting procedure, the most significant bit is replaced with the previously calculated xor result and the least significant bit is dropped. By following this procedure, the LFSR is able to generate a pseudo random sequence of numbers. To further randomize the number generated, this project makes use of two different LFSRs. The output of these two LFSRs are xored together and a subset of the bits were used as the result.

Methods for Preventing Deadlock

To prevent deadlock and race conditions, semaphores were used. Semaphores were used to prevent multiple threads from accessing a shared resource at the same time. This process was used to protect both physical and software based resources. For example, an LCD semaphore was used to restrict access to the LCD. By restricting access in this way, only one thread was able to write to the LCD at a time. In addition to this semaphore, semaphores were also used to protect the “score” and “life” variables. Whenever these variables were incremented or decremented, they were first blocked so that other threads could not read or modify the data while the variables were being changed. To control thread access and prevent deadlocking, blocking semaphores were used in a dynamic priority scheduled system.

Hit Detection

21,20	42,20	63,20	84,20	105,20	126,20
21,40	42,40	63,40	84,40	105,40	126,40
21,60	42,60	63,60	84,60	105,60	126,60
21,80	42,80	63,80	84,80	105,80	126,80
21,100	42,100	63,100	84,100	105,100	126,100
21,120	42,120	63,120	84,120	105,120	126,120

LCD Block Array Positions
X,Y

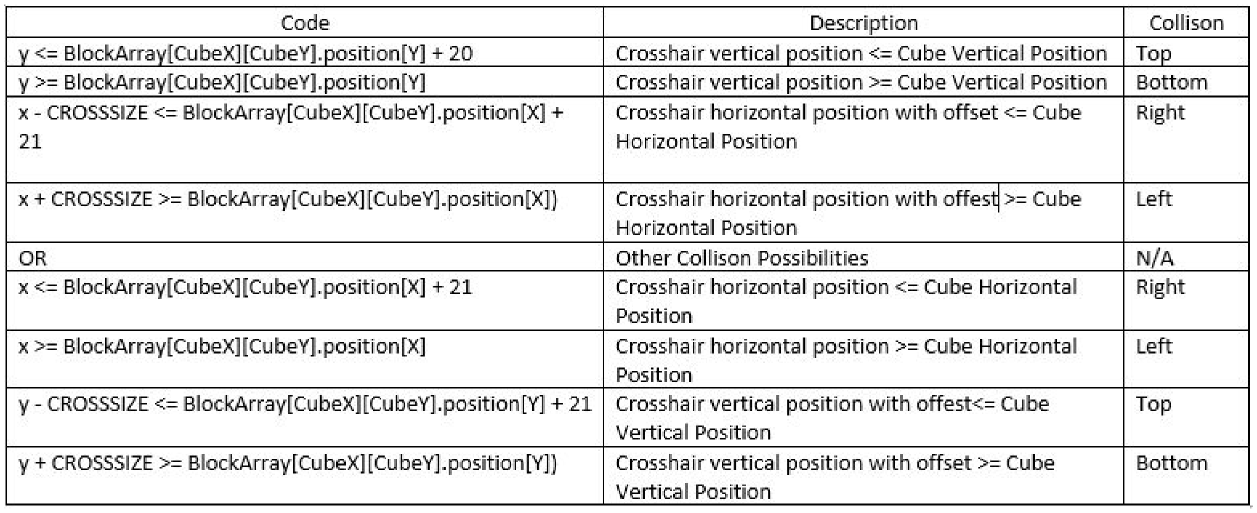
The table above illustrates how the LCD screen was split. It was divided into a series of blocks. Each block on the screen could be occupied by one of the cubes or it could remain empty.

CROSSSIZE = 5
Crosshair draw:

```
BSP_LCD_DrawFastVLine(x, y - 4, 9, color);
BSP_LCD_DrawFastHLine(x-4, y, 9, color);
```

The code above shows the code used to draw the LCD cross hair.

Hit detection explanation



Code	Description	Collison
y <= BlockArray[CubeX][CubeY].position[Y] + 20	Crosshair vertical position <= Cube Vertical Position	Top
y >= BlockArray[CubeX][CubeY].position[Y]	Crosshair vertical position >= Cube Vertical Position	Bottom
x - CROSSSIZE <= BlockArray[CubeX][CubeY].position[X] + 21	Crosshair horizontal position with offset <= Cube Horizontal Position	Right
x + CROSSSIZE >= BlockArray[CubeX][CubeY].position[X])	Crosshair horizontal position with offest >= Cube Horizontal Position	Left
OR	Other Collison Possibilities	N/A
x <= BlockArray[CubeX][CubeY].position[X] + 21	Crosshair horizontal position <= Cube Horizontal Position	Right
x >= BlockArray[CubeX][CubeY].position[X]	Crosshair horizontal position >= Cube Horizontal Position	Left
y - CROSSSIZE <= BlockArray[CubeX][CubeY].position[Y] + 21	Crosshair vertical position with offest<= Cube Vertical Position	Top
y + CROSSSIZE >= BlockArray[CubeX][CubeY].position[Y])	Crosshair vertical position with offset >= Cube Vertical Position	Bottom

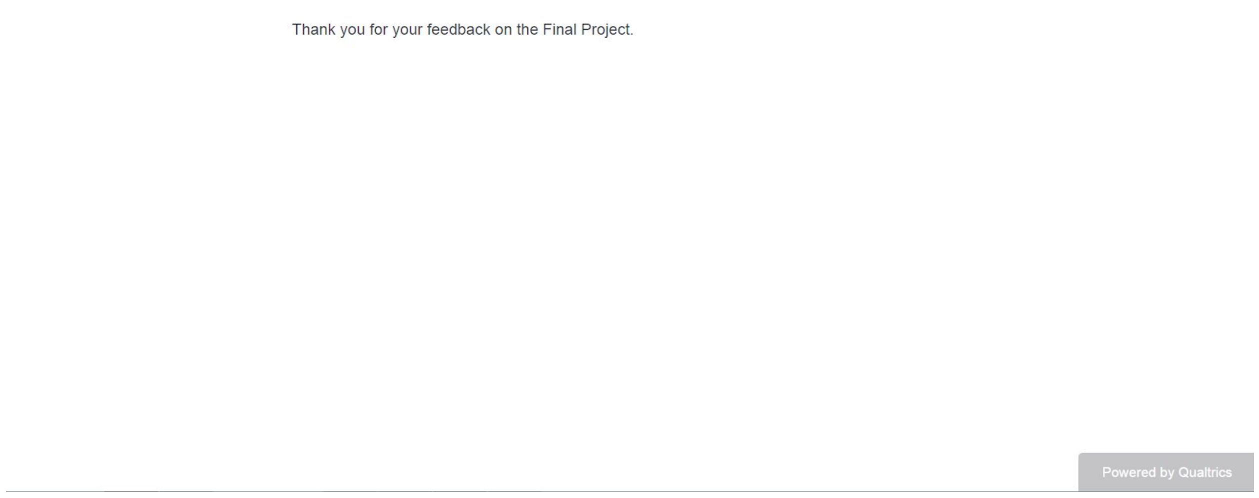
The table above shows the different checks that were performed to avoid cube collisions.

Video

https://drive.google.com/file/d/1_MfCN8_hLagBId3wF2-Y1AOYlkXG6eZN/view?usp=sharing

Survey Responses

Jon's Response:



Thank you for your feedback on the Final Project.

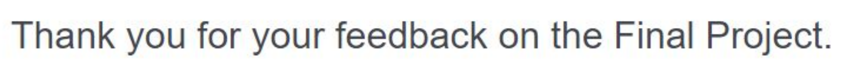
Justin Vaughn Javier:

Thank you for your feedback on the Final Project.

Josh Neal:

Thank you for your feedback on the Final Project.

William Trace LaCour:


Thank you for your feedback on the Final Project.