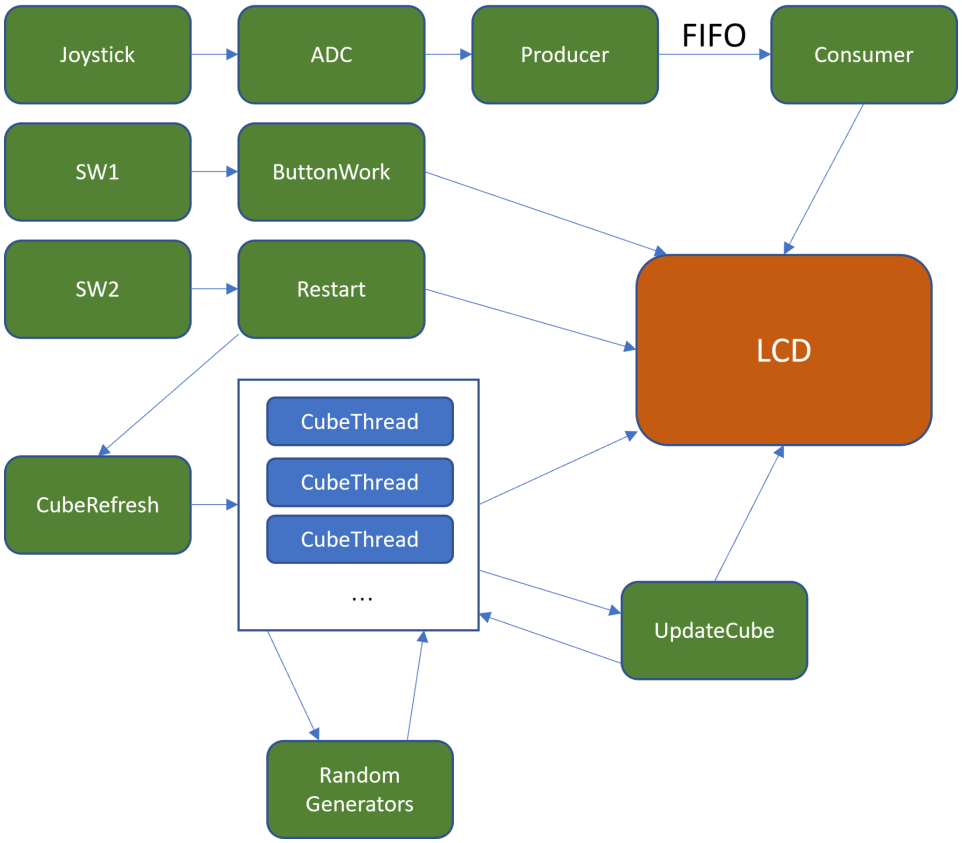
Final Project Part1 Report

Yimin Gao (yg9bq)
Zhenghong Chen (qus9bh)
Chenyang Zhong (cz7rd)
Bassam Khamis (bm7tu)

Team Responsibilities:

Our team members worked together during the whole design and implementation. Yimin Gao was responsible for the basic structure of the design (Random Number Generator, ThreadCube, UpdateCube). Zhenghong Chen was responsible for deadlock prevention implementation and the CubeRefresh function. Chenyang Zhong was responsible for the report and debugging and helped with the implementation of hit logic in part 4, optimization of Restart function. Bassam Khamis was responsible for formulating the report.

Design Diagram:



Random Number Generator:

The code used the linear feedback shift register to provide a pseudo randomly generated sequence of numbers. At first, the register provided a seed value as the first output. On every cycle, the system selects specific output bits using a mask. The bits are exclusively ORed with each other and the result is put back to the most significant bit of the LFSR. The register then shifts all bits by one. With this process, the LFSR is able to generate a pseudo random sequence of numbers. To further randomize the number, the project uses two different LFSRs. THe output of these two are Xored to get the final randomized result.

Deadlock Prevention:

Mechanisms for prevention or detection/resolution of deadlock:

We use a global variable array containing 5 integers to store the address of the current cube. We assume that the coordinates of the current cube are (x, y), and the ranges of x and y are both [0, 5]. We store the value of 10*x+y into the array. When the element of the array is not set to an address, we set it to 99. We didn't use a semaphore here to simplify the deadlock prevention logic. When updating the location, one cube could only scan from the array to see if the location is occupied. (Similar to a binary semaphore)

```
40  uint32_t location[5]={99,99,99,99,99};
```

With such a global array implemented, when **adding a new cube**, we firstly check whether the random initial position of the cube is already occupied. If it is occupied, we need to re-randomize until the targeting cube is not occupied. After we have determined the starting position, we store the new position in an available element of the array (first one that equals 99) and update the location. Before **killing the cube thread**, we need to set the current possible invalid (set the element of the array to 99).

When updating the cube position, we need to **firstly make sure that the cube will not exceed the boundary**. Then we need to **check if the targeting address is already occupied** (if the location is already stored in the location global array). We need to re-randomize the direction of the cube until it meets the above two conditions.

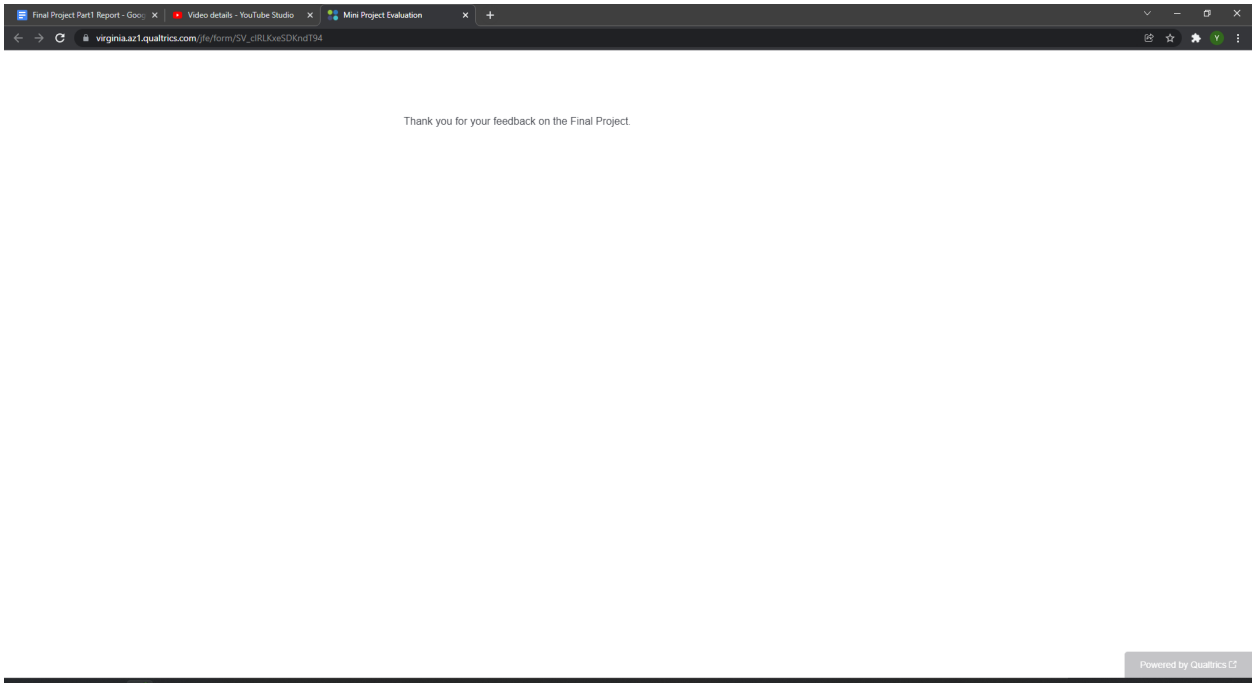
Hit Logic:

Since we implemented an array storing positions of current existing cubes, the hit condition is checked in each ThreadCube execution. (If the position of the crossing is inside the range of the cube) To make the cube updates less frequently, we used an integer value called update to limit the cube update frequency to update the position of cube once for each 100 executions of CubeThread.

Video: https://youtu.be/G3kn_UWcSNw

Survey:

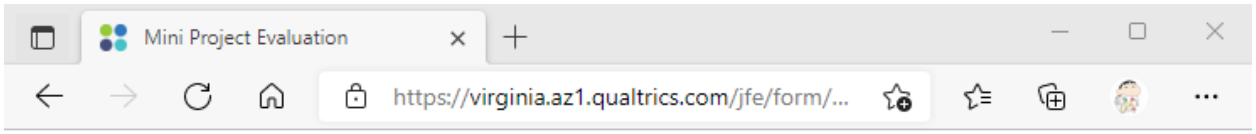
yg9bq:



cz7rd:

Thank you for your feedback on the Final Project.

qus9bh:



Thank you for your feedback on the Final Project.

Powered by Qualtrics

bm7tu: