1. Introduction

In this project, we use Nexys4 DDR FPGA Board with the peripheral (1) QVGA LCD TFT SPI Display; (2) Nunchuck Controller to implement a video game. LCD screen, which follows SPI protocol, is used for display. Nunchuck Controller (gamepad), which follows I2C protocol, is used for user interaction. In the game, the player will use one gamepad to control one ship in the screen. The ship (red circle) is able to (1) move and (2) shoot under the control of the gamepad. Enemies (black triangle) & Friends (blue triangle) randomly occur. Every time the ship successfully shoots the enemy by the bullets (blue rectangle), the score of the ship will increase. Every time the ship is hit by the enemy, the score will decrease. Every time the ship accidently shoots the enemy, the score of the ship will decrease. Every time the ship hits the friends, the score will increase.

2. Preliminary Survey

Before working on this project, we made two experiments to test the basic timing performance of the system.

a. Make elementwise arithmetic operation on the 240*360 uint16_t data array, e.g. add one on every element. In this way, we can get the basic processing frame rate as **4FPS**.
b. Transfer the 240*360 uint16_t data from onboard memory to the SPI display (the RGB color is represented in the16-bit format, i.e. 5r6g5b). In this way, we can get the basic data transfer frame rate.

From the above two experiments, we get the basic timing performance of the system. Therefore, if we would like the increase the fluency of the game, we should:

a. Decrease the processing elements in every loop. For example, if we found an area does not contain any objects (e.g. ship, bullet, enemy) in this loop and in the last loop, then it is unnecessary to process this area (also unnecessary to transfer the data of this area), just let it alone.
b. Decrease the transfer elements in every loop. For example, if we found one pixel has the same color as the previous loop, then it is unnecessary to transfer the color through SPI again. But we should notice that it is not quite economical to make pixel-by-pixel data

transfer. This is because every time we make inconsecutive drawing, extra overhead will be introduced to set the new drawing boundary (call the function setXY). The overhead will become smaller if we draw block by block. But as the block size increases, the possibility that one block contains updated pixel increases. In this way, it should be careful to choose a reasonable block size. At the meantime, we would like the current block to be held in the cache. This also makes the block not too large.

A "**block-based**" processing and rendering strategy is proposed. We choose the block size as 16pixel*16pixel at first. In this way, the 240*320 screen is made up of 15*20 blocks. Currently, one block contains 256 uint16_t data. At the same time, the FIFO depth of SPI is 256 uint8_t. Therefore, the data in one data can feed the FIFO twice. In this way, a **ping-pong** strategy can be introduced (In odd time period, process the first half of the block and transfer the second half of the block; in even time period, process the second half of the block and transfer the first half of the block).

In Section 3, we will introduce the system design, especially (1) the "**block-based**" processing and rendering strategy, (2) the **ping-pong** strategy which overlaps data processing and transferring to improve FPS; in Section 4, we will introduce some **special graphics features** that support polygens and irregular shape in the game.

3. System Design: Block-based processing and rendering strategy

Generally, the whole system runs in a grand loop, which is made up of four steps:

1	Game.input	Get the input from I2C gamepad;
2	Game.run()	CPU updates the state of ship, enemy, bullet, score;
3	Game.draw_grid(…)	CPU calculates which block should be re-rendered
4	General_Display(…)	CPU calculates how to re-render on the screen and transfer data through SPI to screen

Step 1. Get the input from I2C gamepad. I2C does not have as much data transfer compared with SPI. In this way, it is not the bottleneck of the system.

Step 2. CPU updates the state of ship, enemy, bullet, score.

We use a series of inherited class to describe the objects of ship, enemy, bullet, score. The basic class inheritance relationship is shown as below:

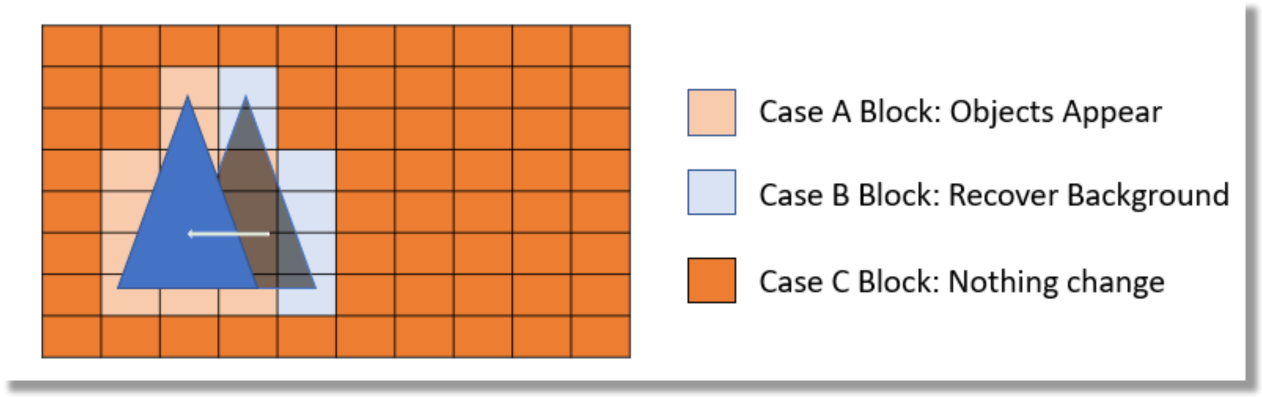
- Class Geometry_Rect: This is the virtual basic class. It has the property of location & size. And it is used to describe the rectangular boundary of the following complex shapes.
- Class drawable: This is inherited from Class Geometry_Rect. This class is used for the basic rectangular objects in the display. It has two functions, i.e.
  - virtual void with_grids(grid_net &grid): This function is used to calculate which blocks are included in displaying this object. The grid net holds a list and a label for every block. The list will record which object appears in this block and the label will record how many objects the block contains in the last frame.
  - virtual void draw_in_grid(…): This function is used to render the grid buffer that shows how the object will be shown on the screen.

- Class Drawable_Script, Class Drawable_Circle, Class Triangle: These classes describe more complex shape and details will be shown in Section 4, which describes the graphics part.

Step 3. CPU calculates which block should be re-rendered. As mentioned above, the whole 240*320 screen is partitioned in 15*20 blocks where each block is sized of 16*16. And every block holds a list and a label. The list will record which object appears in this block in the current frame. The label will record whether any objects have appeared in this block in the last frame. There are three cases:

- Case A. If the list is not empty, it means that there exist objects overlap this block. In this way, we need to re-render this block to make sure the object overlapped in this block is rendered in the right way.
- Case B. If the list is empty but the label is true, it means that although no objects appear in this block in this frame, but some object appear in the last frame. We have to re-render this block by recovering it into the background color. In this way, it shows the objects have leave this block correctly.
- Case C. If the list is empty and the label is false at the same time, it means that no objects appear in this block in the current frame and the last frame. It is unnecessary to re-render this block. We only need to keep this block in the background color (or background figure). In this way, we don't need to calculate how to re-render this block and we also don't need to transfer the data of this block through SPI to screen in the next step.

Take Fig. as an example, a blue triangle moves from right to left. In this example, the light orange blocks are the Case A blocks, which are necessary to re-render since objects appear in the current frame; the light blue blocks are the Case B blocks, which are necessary to re-render since background should be recovered; the dark orange blocks are the Case C blocks, which are unnecessary to re-render.


Case A Block: Objects Appear
Case B Block: Recover Background
Case C Block: Nothing change

This strategy has the following features:

- We only need to re-render case A block and case B block. This will reduce the calculation on how to re-render the block and the data transfer.
- If one block holds one object both in the last frame and the current frame, it is necessary to re-render and re-transfer this block again. It seems to be uneconomical since we waste some data transfer in the case some blocks have nothing to change. For example, if the object in the fig. is not a triangle but a polygon that totally fills all the light orange blocks, there are some overlapped and unchanged blocks when the object moves from light blue blocks to the

light orange blocks. But such "economical" solution has many constraints. First, the object shape should be extremely regular. For example, when the object is changed into the triangle, all the overlapped blocks are not unchanged any more. Second, the color (or the pattern) of the object should be extremely simple. For example, when polygon is not filled with a pure color but filled with a complex pattern instead, the overlapped blocks will also have changes. Third, extra calculation is needed to find out whether one block is changed or not. Such calculation might be not difficult for single object. But the calculation will be quite complex in a multi-object circumstance.

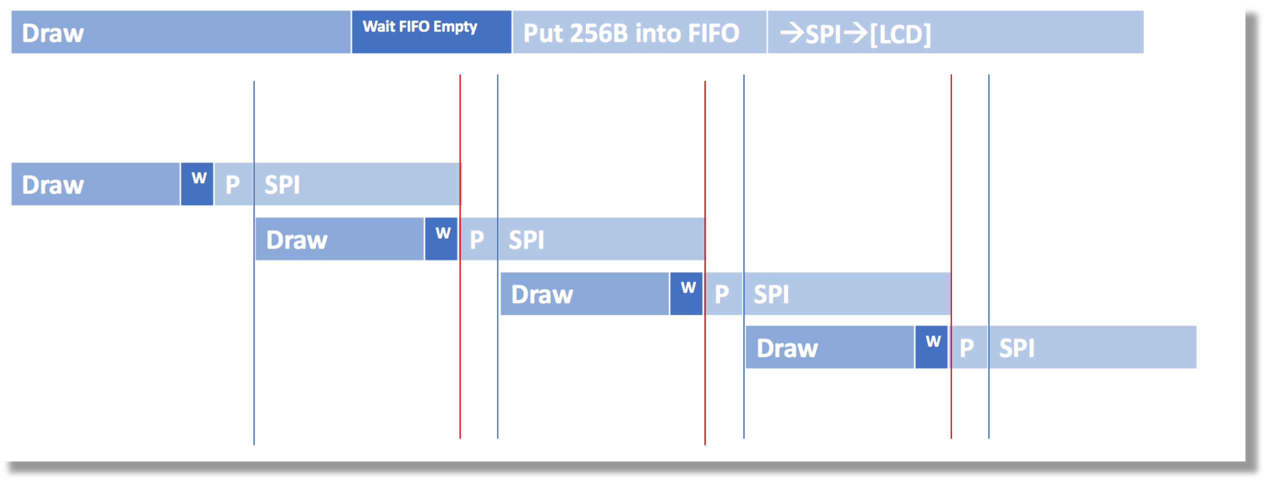
- This is a **cache-friendly** strategy. Compared with basic strategy which holds the whole 240*300 pixels, we only need to hold a small buffer which contains a 16*16*2bytes. And this buffer can be reused for all blocks. In this way, it reduces the memory footprint (from 240*300*2bytes to 16*16*2bytes). It reduces most of the memory acess when it is only necessary to hold a 16*16 block (most of the operations can be done in the cache). Moreover, the FIFO strategy described below will make our system further cache-friendly.

Step 4. CPU calculates how to re-render on the screen and transfer data through SPI to screen.

According to Step 3, if one block is Case A or Case B, then it is necessary to re-render and transfer the data. As mentioned above, the block holds a list which contains all the objects that are overlapped with this block. We will calculate how to render the objects in the block one by one. Basically, to render a rectangle in a block, we only need to calculate the most left-up index and the most right-down index that contains the rectangle in the block. For more complex shape, like circle, triangle, it will be described in Section 5. And in case of multiple objects, the objects are rendered in the order from the objects that have lower priority (which should be shown in the behind, more close to background) to the objects that have higher priority (which should be shown in the front, further to the background). In this way, the earlier rendered objects might be covered by the later rendered objects. For example, the score scripts are rendered in the latest order, so it will show like the score is floating above other objects when ships, bullets, and enemies appear in the blocks that always contain the score.

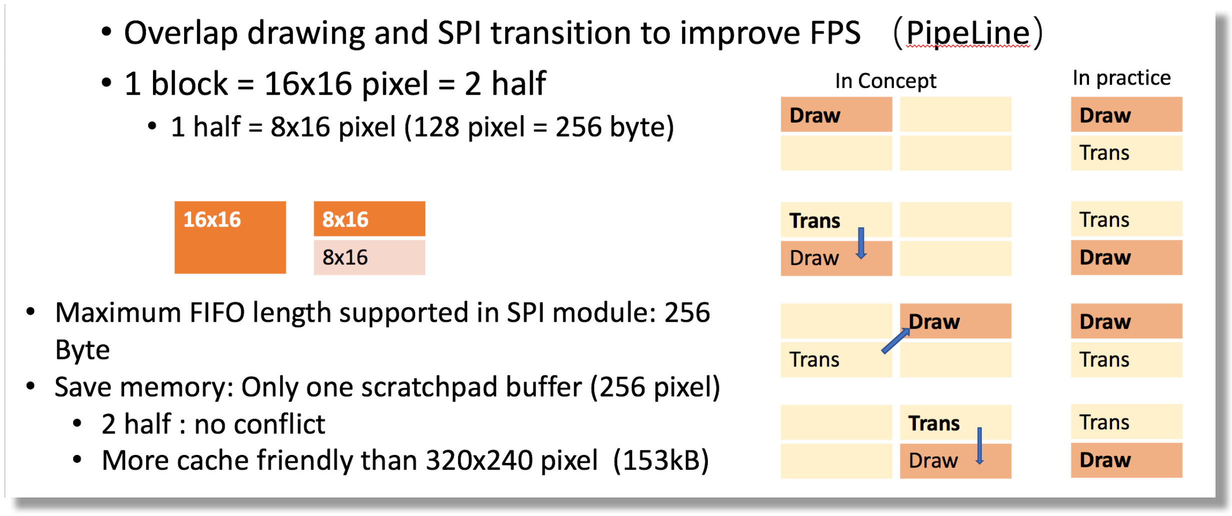
After we figure out how to render the objects, we will send the rendered buffer to the SPI controller. In this project, we use three different versions of data transfer strategy, i.e. the plain strategy, the FIFO strategy, the FIFO+ping-pong strategy.

- The plain strategy: Lab2B offers a simple version code of write 16bit data to the SPI controller, which first write 16bit data to the SPI controller, and then wait for the SPI_IISR to set the transfer complete signal. It means that every time we send a pixel to the SPI controller, we have to wait until the transfer complete signal comes. In this way, this strategy is not time efficient.
- The FIFO strategy: The SPI can be used in a FIFO mode. And we set the FIFO depth as 256bytes. Since every pixel has 2bytes, we can transfer the 16*16 block in two rounds. This means we can send half of the block data once into the FIFO, instead of feeding one pixel once into the FIFO. And we only need to wait twice for the transfer complete signal in one block, instead of waiting the transfer complete signal 256 times.



- The FIFO+Ping-Pong strategy: When we wait for the transfer complete signal, only the SPI is busy sending the data in the FIFO to the LCD, while the CPU works in an idle mode. It would be more time efficient to let the CPU draws (or renders) some other pixels while waiting for the transfer complex signal. As Fig. shows, three events should be done by CPU, i.e. draw (or render), wait FIFO empty, put 256 bytes into the FIFO; only one event should be done by SPI controller, i.e. send data from FIFO to LCD. And we observe that compared with CPU drawing and sending data into the FIFO, SPI sending data from FIFO to LCD spends longer time.

  In this way, we introduce the our ping-pong strategy. In every time period, we draw (or render) the current half block, and wait for the transfer complete signal of the last half block. Once the transfer complete signal comes, we put the data of current half block into the FIFO. Then SPI takes control of the current half block while CPU turns to draw (or render) the next half block of data.
- This strategy also helps to make a **cache-friendly** system. Every time we operate the data of half a block. This means we only need to operate a buffer which has 16*8*2=256 bytes. And the buffer can be reused for all the half blocks. The ping-pong strategy further reduces the memory footprint. It is easier to keep all these 256 bytes in the cache without time-consuming memory access.

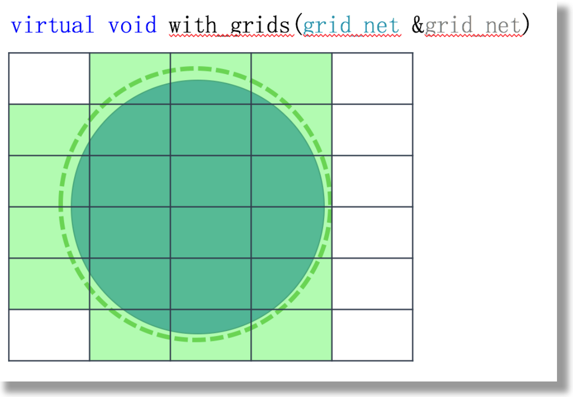


- Overlap drawing and SPI transition to improve FPS （PipeLine）
- 1 block = 16x16 pixel = 2 half
  - 1 half = 8x16 pixel (128 pixel = 256 byte)
- Maximum FIFO length supported in SPI module: 256 Byte
- Save memory: Only one scratchpad buffer (256 pixel)
  - 2 half : no conflict
  - More cache friendly than 320x240 pixel (153kB)

16x16
8x16
8x16
In Concept
In practice
Draw
Trans
Draw
Trans
Draw
Trans
Draw
Trans
Draw
Trans
Draw
Trans
Draw
Trans
Draw

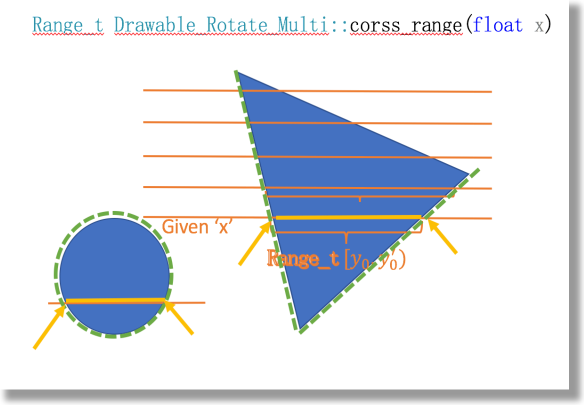
4. Graphics detail

   a. Basic Principle: If we would like to render a shape on the screen, we should know which group of blocks (or grids) to draw, and we should know how to draw a correct fragment within a specific block (or grid). As shown in the right figure, if we would like to draw a green circle, we should know which group of blocks to draw: The object will register to the blocks (or grids, the light green in the right figure) that it is related to. The object list of the block will record which objects is related to this block. Currently, we have sloved which group of blocks (or grids) to draw; and the next problem is to know how to draw a correct fragment within a specific block (or grid).

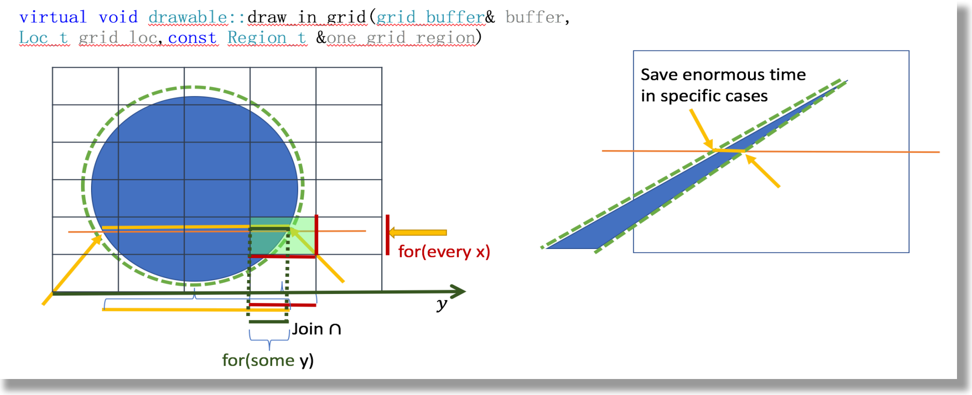


   b. How to render a convex shape: As shonw in the right figure, the sweep line is introduced to render a convex shape. For every line, we need to find the pair of cross points for the boundary of shape and horizon line specified by input x.

      In our implementation, circle and polygon were implemented differently, but the principle is the same. Rectangular without rotation (bullets) have easier method invoking less computation.



   c. How to render part of a convex shape within a grid: to merge the above two together, we get a solution on how to render part of a convex shap within a grid. In our implement, we introduce the sweep line method. In this method, we only need to find out the left and right boundary of every line. Compared with rendering pixel by pixel, this sweep line method saves enormous time in specific cases. For example, a rotated enemy overlaps a small region in one block.


virtual void drawable::draw_in_grid(grid_buffer& buffer, Loc_t grid_loc, const Region_t &one_grid_region)
for(every x)
y
Join ∩
for(some y)
Save enormous time in specific cases