Computation I 5EIA0
Exercise 2: Tetris (v1.3, August 27, 2020)
Deadline Wednesday 16 September 23:55

Figure 1: Tetris Logo `https://www.youtube.com/watch?v=VNbo1AGqKrI`

In this exercise you will write a simple version of the Tetris game (`https://en.m.wikipedia.org/wiki/Tetris`) that uses the buttons and graphics on the PYNQ board.

Note that to display the graphics generated on the PYNQ in the Virtual Machine you need to login to the PYNQ with `ssh -XC pynq`.

Task 1. Create a new file `1-flag.c`. Let's first get used to the graphics functions that are provided by the library with the PYNQ board in the `libpynq.h}` file:

```
void init_display(int height, int width, int scale, pixel displaybuffer[height][width]);
void draw_display();
void clear_display();
void set_pixel(int y, int x, unsigned char r, unsigned char g, unsigned char b);
```

Your program must indicate that it wants to use the standard C I/O library and the buttons and graphics library, using the usual include directrives. The first thing to do is to create a display window of, say 15 pixels wide and 9 pixels high. The number 20 is the scale factor of the window: if you set it to 1 then you get small pixels and you can have more detailed graphics. For now, 20 is good, but feel free to play with it.

```
#include <stdio.h>
#include "libpynq.h"
#define HEIGHT 12
#define WIDTH 24
int main (void) {
  pixel display[HEIGHT][WIDTH];
  init_display (HEIGHT, WIDTH, 20, display);
  // yellow pixel in the centre of the screen
  set_pixel (HEIGHT/2,WIDTH/2,255,255,0);
  /* your code here */
}
```

Now add some code to draw a Dutch flag that consists of three equal bands of red, white, and blue. Draw the flag line by line; this means that the pixel coordinate changes quicker in the horizontal (X or WIDTH) direction than in the vertical (Y or HEIGHT) direction. Use two loops to first draw the red band of the flag, then two loops for the white part, and then two loops for the blue part.
Inside the loops use the `set_pixel` function. To get a good idea of the order your draw the pixels, update the display with `draw_display` after every `set_pixel` and add a small delay (e.g. 100 msec). You'll notice that the graphics window closes as soon as the program finishes. To keep the window open add a `sleep_msec` function call of e.g. 2 seconds. The output should look like this:

Figure 2: Drawing line by line (width loop inside height loop).

Next, modify your program to reduce the number of `for` loops from 6 to only 2. You will probably need an `if` statement inside to check if you need to draw a red, white, or blue pixel. If you want, you can even reduce the two loops to only one loop (hint: use the modulo and divide operators to compute height and width).

Note that you need to login to the PYNQ with `ssh -XC pynq`. If you get the terminal error "Error: Could not open display" then either you forgot the `-X` or the X forwarding has been lost, then exit from PYNQ and login again.

Task 2. Extend your program to also draw the flag column by column. (You'll need to call `clear_display` after drawing the flag line by line.)

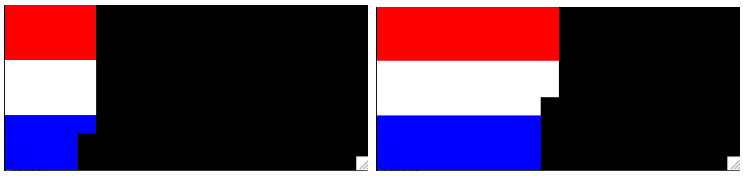
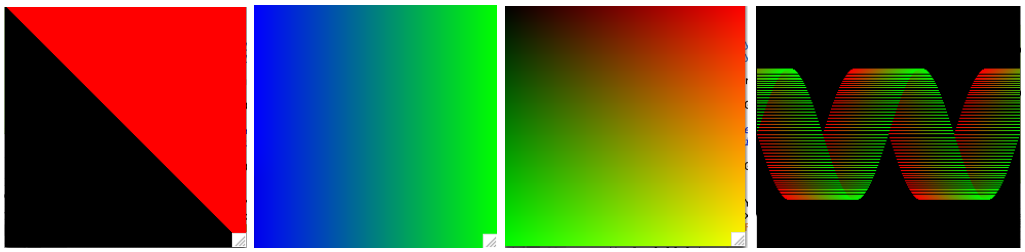


Figure 3: Drawing column by column (height loop inside width loop).

Try different display sizes (e.g. 20x10, 20x9, 11x31) to ensure that your program works for all display ratios (and not for e.g. only WIDTH=HEIGHT).

Task 3. This is an optional task. Create a new file `2-sine.c`.



If you want, you can try some more sophisticated shapes or colour schemes. Notice that the pixels are much smaller now. Use a display of e.g. 200 by 300 and scale of 1. For example, draw the black & red triangle shown above. Try different colour patterns, e.g. changing the colour from left to right (hint: `set_pixel(y,x,0,x,255-x)`).
Or changing the colours from left to right as well as top to bottom (Hint: you need to mix two colours, starting at 100% of one colour and gradually change it to 100% of the other colour, by using something like `set_pixel(y,x,y,x,0)`.)
Plotting the sine curves is more sophisticated and requires the use of the `sin` function from the `#include <math.h>` library. (Hint: use a horizontal/x for loop to draw a single sine curve (something like: `set_pixel(HEIGHT/2+HEIGHT/4*sin(x*4*3.14/WIDTH),x,255,0,0);`). Then add another loop that shifts the sine to the right and modifies the colour.)

Task 4. Copy `1-flag.c` to a new file `3-drop-block.c`. Remove the code for the flag, but keep the rest.
Define a two-dimensional array called `occupied` of the same size as the screen:

```
int occupied[HEIGHT][WIDTH] = {};
```

Note that the array is initialised with zeroes by using the `= {};`. We keep track of where blocks are on the screen in the `occupied` array: when `occupied[y][x]` is zero then there is no block, and if it is one then there is a block.
Write a function `void draw_grid(int occupied[HEIGHT][WIDTH])` that has two nested loops (e.g. y for height, x for width) in which you call:

- `set_pixel(y,x,r,g,b)` with RGB equal to black if `occupied[y][x]` is zero
- `set_pixel(y,x,r,g,b)` with RGB equal to red if `occupied[y][x]` is one

After the two loops you should update the display with `draw_display`. When you call `drawgrid` on the empty `occupied` array then it should display the black screen.
Now drop a block (consisting of a single pixel) from the (0,WIDTH/2) position to the (HEIGHT-1,WIDTH/2) position on the screen. Every 200 milliseconds move the block one row lower. You do this by writing a while loop in which you

- set `occupied[y][x]` to one (the falling block)
- call `drawgrid`
- wait 200ms
- set `occupied[y][x]` to zero (remove the falling block)
- increment y
- exit the loop if y is equal to HEIGHT (i.e. the pixel has fallen off the screen).

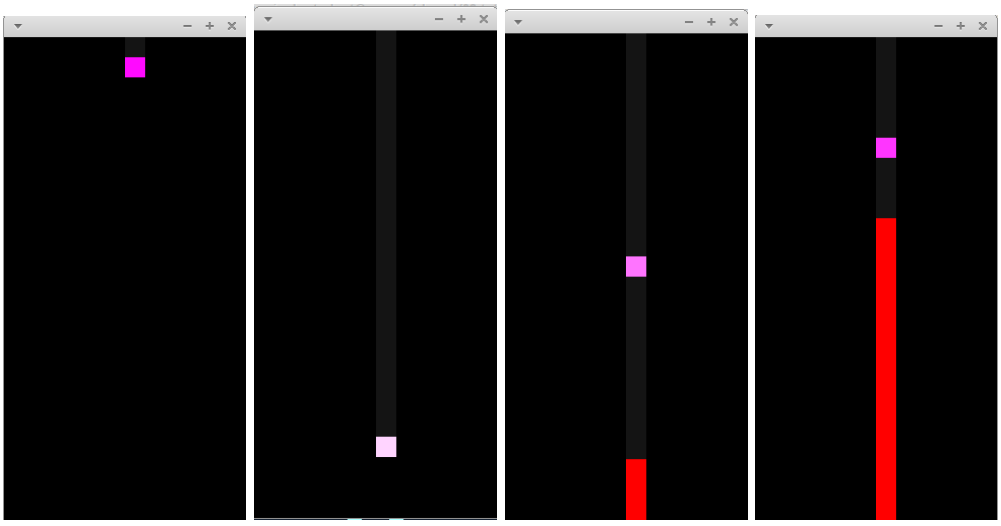
Before the loop initialise y to 0 (top of the screen) and x to WIDTH/2 (the middle of a line).
Add an infinite loop around the such that you repeatedly see a single block dropping down down the middle of the screen.

Task 5. Copy the program to a new file `4-stack-blocks.c`. Now you have to extend the program to not let the block drop out of the window at the bottom, but to keep it there. When the next block arrives it should stop at above the last block. In other words, blocks stack until they reach the top of the screen, like in real Tetris.

This is not too complicated because you already keep track of where blocks are. When you want to draw the falling block that's at position y,x you now need to check that the next position (i.e. `occupied[y+1][x]`) is not occupied. If the position is occupied, then you leave the block there (i.e. leave `occupied[y][x]` at one, i.e. current block) and drop the next block. Repeat until the stack of block reaches the top.

Your program will look something like:

```
#define HEIGHT 12
#define WIDTH 24
int main (void) {
  pixel display[HEIGHT][WIDTH];
  init_display (HEIGHT, WIDTH, 20, display);
  int y, x = WIDTH/2;
  do {
    // start with a new block from the top
    y = 0;
    // check that it's not run into the bottom or another block
    while (y < HEIGHT && !occupied[y][x]) {
      occupied[y][x] = 0;
      increment y;
      occupied[y][x] = 1;
      drawgrid(occupied);
      small delay;
    }
  } while (y > 1); // blocks reached the top
  // finished
  long delay;
}
```



Task 6. Copy the program to a new file `5-buttons.c`. Next, we add user input to our Tetris. When we push button 0 the block should move to the right, and pushing button 3 should move it to the left. The following help function in the `lib_pynq` library is very useful:

```
extern int button_states[nrpushbuttons];
extern void sleep_msec_buttons_pushed (delay);
```

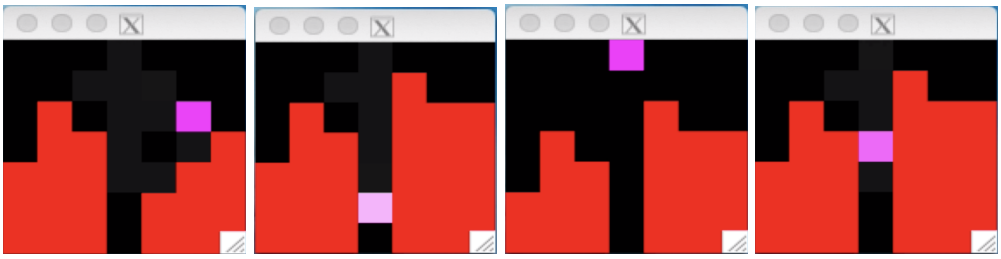
After calling `sleep_msec_buttons_pushed` with a small delay, e.g. 30 msec, the array `button_states` contains the number of times the buttons have been pushed during the time interval. Watch out! This may be a very large number, so you probably want to do the following:

```
sleep_msec_buttons_pushed (delay);
if (button_states[0] > 0) { /* button 0 has been pushed, move the block to the right */ }
if (button_states[3] > 0) { /* button 3 has been pushed, move the block to the left */ }
```

every time you move the block to the next row.

Task 7. Copy the program to a new file `6-tetris.c`. If the sampling delay for the buttons is small (e.g. 100 msec) then blocks moves down very quickly, and a larger delay (e.g. 300 msec) make them move slower. Ask the player if (s)he wants an easy, moderate, or hard game, and then wait one second before starting the game. Keep track of the score, i.e. the number of blocks that the player managed to position before hitting the top. This is a sample output of the final program:

```
Would you like an easy (3), moderate (2), or hard (1) game? 2
Game starts in 1 second
Your score is 10 blocks!
Would you like to play again [yn]? n
Bye!
```



Task 8. Copy the program to a new file `7-tetris-row.c`. The final feature to add is to remove the final row when it is full, i.e. it has blocks from left to right. First, you need to detect this (use a for loop). Then you need to shift contents of the `occupied` array one row down (the top row becomes empty). Finally, redraw the screen with the new contents using the `occupied` array.

Submission: Submit your file `7-tetris-row.c` that implements task 8 on Oncourse (`Exercise 2: Tetris (due 16 Sept 23:55)`). You can resubmit as often as you want until the deadline.

- **30/8 v1.1** Added ssh -XC.
- **2/9, v1.2** Added screen shots,changed pi to 3.14.
- **20/9, v1.3** Restructured a bit to make Task 4-5 step smaller.