CAB202 Tutorial 3

Arrays and timing

At the core of almost every program, there is the manipulation and operation on underlying data. The representation of this data is one of the most crucial elements in the design of a program. This tutorial introduces constructs for representing collections of information (arrays). For example if a character was assigned for every square in a typical graphical game screen (80×24), that would mean creating 1920 differently named variables manually. Instead, these 1920 characters can be held in a single array variable. By the end of this tutorial, you will be able to make use of arrays and array indexing. This tutorial is worth 3% of your final mark for this subject.

Timing of operations

Throughout the semester we have been using delay calls (either `sleep()` or `timer_pause()` in the ZDK) to control timing in programs. This is not only wasteful (having your computer rest for an amount of time is not making the most of the resources available), but it also is an extremely unreliable way to run timed events. The displaying of graphics is a prime example of this. The time between refreshes of the screen (i.e. frame rate, frames per second, FPS, Hz) should be constant and not fluctuate depending on the time it takes to execute sections of your code. Take the following code snippet:

```
while(1) {
     do_work();
     draw_and_show_screen();
     timer_pause(100);
}
```

We have operated under the assumption that a loop like this would run 10 times a second (or at 10Hz). This is true when the `do_work()` and `draw_and_show_screen()` functions take a negligible amount of time to complete. But what would happen if the `do_work()` function took 1s to complete? Then your time between refreshes would drop to 1.1 seconds (or a frame rate of ~0.9Hz). Even worse, what would happen if the time taken for `do_work()` was completely unpredictable and ranged from negligible to a number of seconds? Not only would the game be slow, but the display would be jittery, jumpy, and irresponsive.

```
#include “cab202_timers.h”
timer_id timer = create_timer(1000);
while(1) {
     while(!timer_expired(timer)) {
          do_small_pieces_of_work();
     }
     draw_and_show_screen();
}
```

The above code makes use of time based operation. The code uses a timer to ensure the screen refreshes at approximately 1Hz (you will need to include `cab202_timers.h`). This is only a simple

software based timer, and consequently there are conditions where this structure still won't guarantee time based execution. Later in the semester we will cover hardware timers and interrupts which provide an even more robust solution.

Non-Assessable Exercises

NOTE: These exercises are not assessable, and the tutor can help you with them. Assessable exercises are available on the AMS.

1. Make use of timed operation to provide a consistent in game frame rate

For this question, consider you are trying to create a game and it is imperative that the game screen has a refresh rate of 1Hz (1 frame per second). The template file provides a mock game that reports statistics about the frame rate. It displays 1) desired frame rate, 2) current frame rate, 3) the average frame rate over the last 50 frames, and 4) the total work done by the worker function. The current implementation uses a delay based main loop (see the implementation of **`work_for_hopefully_1s()`**). As the statistics show, this implementation provides a terrible frame rate with jumpy and unpredictable behaviour (because it assumes `do_around_50ms_work()` takes exactly 50ms rather than enforcing a frame rate of 1Hz). Swap the call in main from **`work_for_hopefully_1s()`** to **`work_for_actually_1s()`** and complete the implementation of this function so that:

- The frame rate is always as close to 1Hz as possible.
- The average is stable (i.e. should not be changing much with each iteration).
- The amount of 'work' being done increases as quickly as possible (i.e. if you don't call the function **`do_around_50ms_work()`** the work completed will not increase).

2. Make 40 ball sprites move down the screen

Consider a basic game where 40 balls (represented by an 'o' character) are falling down the screen. For example, maybe the aim of the game is to grab all the balls before they hit the bottom.

The demo code provided initialises 40 balls at the top of the screen. They are placed evenly along the top row of the screen. The code is set up so that the **`move_and_draw_balls()`** function is called 4 times a second (at 4Hz or every 250ms). Complete the `move_and_draw_balls()` implementation such that:

- The y value corresponding to each ball in the **`y_array`** is modified in this function.
- Every call moves all 40 balls down. The distance that each ball should move down is provided by the array in parameter **`dy_array[]`**. For example if `dy_array[9]` equals 3, then the 10th ball should move down 3 rows.
- The balls stop in the bottom row (i.e. a call to the function when a ball is already at the bottom should not move it). The game exits when all balls are at the bottom.
- The function must call `show_screen()` **once** on every function call.
- You **must not modify** the values in parameter **`dy_array[]`**! You should only access them.

3. Flip an arrow in different directions

Consider a game where the player's icon must show the direction they are facing. The player can either face top right, top left, bottom right, or bottom left. One approach would be to create 4

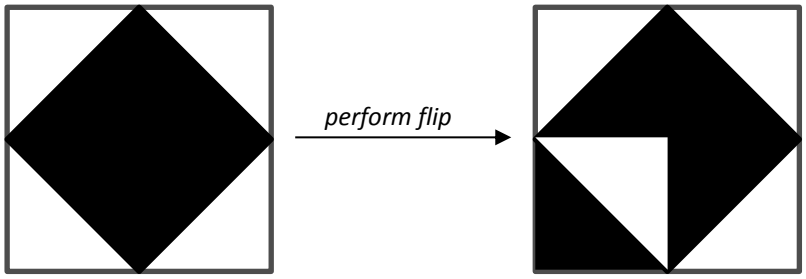
different two dimensional character arrays with the sprite for each direction. This is wasteful considering you can simply flip one sprite to face the desired direction.

The demo code provided explores this problem in a bare bones case. The code creates a 10x10 character array representing an arrow facing the top left corner. The 'f' key is pressed, to toggle between drawing the original arrow, and the arrow created by calling the **`draw_flipped_arrow()`** function. The rest of the screen is blocked out (with a solid colour) so that you can easily see that you aren't accidentally moving the arrow when you flip it. Complete the following 3 different versions of the `draw_flipped_arrow()` function:

1. Arrow faces the top right corner (i.e. flipped along the y axis)
2. Arrow faces the bottom left corner (i.e. flipped along the x axis)
3. Arrow faces the bottom right corner (i.e. flipped along both axes)

Challenge exercises:

- *Modify the template file so that rather than drawing a 10x10 arrow, a 20x20 arrow is drawn instead (it should still be centred in the middle of the screen, with the blocked out area around it).*
- *Change the arrow to a diamond that can be thought of as 4 equal pieces (by drawing a horizontal and vertical line down the middle). Update your flipping function to only flip the bottom left corner of the diamond like the image below (**hard**)*



- *Create a function with the signature* **`draw_flipped_diamond(int piece, int direction)`** *where piece is numbered 1-4 from left to right, top to bottom, and direction is 0-2 (where 0 equals flip on x axis, 1 flip on y axis, 2 flip on both axes). If invalid arguments are supplied, nothing should be done (**extra hard**).*

4. User controlled array indexing

Write a basic program which contains two arrays: one that holds your full name (with spaces) and another with your student number (with the 'n' at the front). The top left of the screen should display what array is currently selected, and the current index in the array. The user's input should do the following:

- Pressing the 't' key should toggle between the two arrays
- When the name array is selected, the left and right keys should control what letter of your name is currently displayed in the middle of the screen. When reaching either end, the selection should wrap to the other side.

- When the student number array is selected, the entire array should be displayed and an asterisk ('*') printed above the current index of the array (like above, use the arrow keys and wrap at the edges). When a number key is pressed, the current index (where the asterisk is) and the character corresponding to the pressed number should swap positions.

Note: when trying to check if an arrow key is pressed you should use the declared ncurses constants available here (e.g. `key_code == KEY_LEFT` ***would check for a left arrow press):***
https://www.gnu.org/software/guile-ncurses/manual/html_node/Getting-characters-from-the-keyboard.html

Challenge exercises:

- *Extend the functionality for the name array, so that space characters are skipped when they are selected (and the array index in the top left corner should reflect this).*
- *For the student number array, make the up and down arrow presses increase and decrease the value of the current digit. If the 'n' character is currently selected, the keys should do nothing. The number should wrap at 9 and 0 (like counting but without 'carrying the one').*
- *Extend the previous, but this time 'carry the one'. The functionality should exactly mimic counting up and down with your student number.*

Assessed Exercises (AMS)

Complete the assessed exercises via the AMS (available at http://bio.mquter.qut.edu.au/CAB202).