ECGR 4101/5101, Fall 2013: Lab 4

Learning by Example: Labyrinth Game

Learning Objectives:

This lab will test your ability to apply the knowledge you have obtained from the last two labs to and expand that knowledge further by introducing you to some numerical methods and there embedded implementation.

Note: be warned, this lab expects that you are very comfortable with the Lab2 and Lab3 procedures for creating a HEW project from scratch and implementing the custom LCD interface. Details on these steps will be skipped in this lab as it is expected that you already know how to do this.

Additionally, this lab will not provide as much step by step detail as Lab 2 and 3 did so be prepared to spend additional time researching datasheets, and using the age old process of trial and error. I also recommend that you get acquainted with the HEW debugger as being able to add breakpoints and variable watches is very useful when trying to figuring out why your code isn't working correctly.

General Information:

1. Review the steps provided by the lab 3 supplemental information document
2. Review the steps provided by the lab 4 Prelab Activity
3. Copy the necessary files into your new workspace and import them
4. Write your code to create the labyrinth game
5. Demonstrate your working project to the TA, and turn in a lab report.

Prelab Activity:

You may use the PCs in EPIC 2130/32 or your own PC to do this lab experiment. If you want to work on lab assignments on your own PC, then you will need to load the necessary tools on your PC in order to perform this exercise.

In this lab you will need to create a project workspace that is very similar to the lab3 workspace since you will be using custom LCD graphics. It is recommended that you either follow the prelab exercise provided in lab 3 or backup your lab 3 workspace and use it as the base for lab 4.

Tip: you will not need any of the custom LCD characters from lab3, so don't feel obligated to include the Top Gun logo in this project.

In this lab you will also be using the 3-axis accelerometer that is included on the RX63 development board. Since part of this lab assignment is to figure out how to get data from the 3-axis accelerometer detail steps about how to do this will not be provided, however, as a helpful hint you might want to use the YRDKRX63N project wizard to create a "***sample code***" project of the "**IIC Master**" and look at how that sample project communicates with the accelerometer. Use this sample code as your base code for

this project. It is advised that you understand this code as you will be updating this code to design this project.

A major part of this project is the understanding of the basics of an LCD and it's working.

To expand upon your Glyph knowledge further, two commands you might want to consider adding to your lcd.c lcd_initialise() function are

```
GlyphWrite(G_lcd, GLYPH_FRAME_RATE, 137);
GlyphWrite(G_lcd, GLYPH_CONTRAST, 255);
```

GlyphWrite is a method that writes a LCD command to the LCD driver while GLYPH_FRAME_RATE will configure the LCD to operate at a particular refresh rate and GLYPH_CONTRAST will configure the LCD pixel darkness.

Note: the max refresh rate is 137Hz for the LCD provided by the RX development kit.

Note: GLYPH_CONTRAST has a value between 0 to 255, where 255 is the darkest setting and 0 is the lightest. If you have issues with this command (the screen appears completely dark, try reducing this amount by half)

Note: please check out both the LCD datasheet and the Glyph API manual for more information about all the cool and useful things you can do to your LCD configuration.

One issue that you might have discovered in lab 3 is the LCD screen inability to correctly set the y value to a particular pixel. After further investigation, you might have discovered that the y offset was rounded to the nearest 8th pixel. So for example trying to set the LCD to a y offset between pixel 1 to pixel 7 would result in the next character being rendered at either pixel 0 or at pixel 8.

This rounding y offset property is a function of the LCD screen used, and it makes rendering graphics that required pixel by pixel movement choppy looking as you might have noticed in lab 3 when the jets moved up and down on the screen.

Because this lab will require pixel by pixel movement of an object, and at this point you now have some experience working with custom LCD characters it is time to discuss some interesting ways around this y offset limitation that the LCD driver has.

At this point you should know from experience that a custom LCD character the size of the LCD screen can successfully be rendered on the LCD screen if the LCD x,y offset is located at 0,0. This fact implies that we can dynamically generate a LCD character the size of the screen that contains all of our correctly located pixel information and then render this dynamically generated LCD character to the LCD screen in order to resolve the y offset issue caused by the LCD driver.

This concept could be classified as a screen buffering, in which we keep a virtual copy of the LCD screen in memory and update the LCD only when changes are detected.

Before we can create a LCD character that will function as our screen buffer we must first examine how LCD characters are defined in a font file.

From bitmap_font.c it becomes apparent that custom characters are defined as
`const uint8_t <character>[]`
or more verbally stated as constant unsigned character array, which implies that typically characters are not changeable at runtime.

However, if the const term is removed, the character can be changed at runtime and it will be uploaded to the LCD via the glyph driver when the character is written to the LCD the exact same way as a const character which thus allows us to easily create a screen buffer from a non-constant character definition.

Note: To make your life easier, a screen buffer character definition has been provided on the embedded website in a file called lab4.c that you should add to your bitmap_font.c file in a manner described by lab3.

Additionally you should also add
`extern uint8_t Screen[];`

to lcd.c in order to quickly access the screen buffer that needs to be modified.

Note: be sure that you don't forget to add the new character to the Bitmaps_table otherwise you will not be able to use the Set_LCD_Char command to render the character.

Note: be sure that you select the appropriate font before trying to render the character, Set_Font_Bitmap is required.

Once you have a screen buffer character, the next challenge is mapping the information in the screen buffer to pixels on the LCD screen. In lab 3 we briefly explained how characters are stored in font memory and to quickly summarize, the first two bytes are the width and height of the LCD character, while the 3 to end of array byte represent LCD character data.

Each byte in the LCD character data represents 8 vertical pixels while each byte index represents a horizontal movement to the right. Once the byte index reached the width of the LCD character the index is reset back to zero and the vertical offset is increment by 8.
For example a 3 by 16 LCD character would have the following data structure.

Byte 1 would be 3 which is the width of the character while byte 2 would be 16 which is the height of the character after this the LCD character data would follow (Byte 3 to Byte 8) and have the LCD mapping as shown on the next page.

	Column 1	Column 2	Column 3
Line 1	Byte 3	Byte 4	Byte 5
Line 2			
Line 3			
Line 4			
Line 5			
Line 6			
Line 7			
Line 8			
Line 9	Byte 6	Byte 7	Byte 8
Line 10			
Line 11			
Line 12			
Line 13			
Line 14			
Line 15			
Line 16			

These 6 bytes of data represent 3*16 or 48 pixels on the LCD screen.

Note that each byte represents 8 vertical pixels in which the LSB is the top pixel and the MSB is bottom pixel in the vertical pixel array.

To correctly map the screen buffer coordinates to LCD coordinates a C routine similar to this pseudo code is required. (Note if you want to really understand how LCD characters work I recommend you try to create your own pixel algorithm and not look at the one below)

```
Function LCD_set_pixel(x, y)
        Define xindex=0
        Define yindex=0
        Define shiftby=0
        if x>=96 or y>=64 or x<0 or y<0 then
                Exit function since x y is outside of the lcd screen
        end if
        yindex=y divided by 8  // since each byte holds 8 pixels
        shiftby=y minus (yindex times 8 ) // find the bit offset in the byte
        xindex=2 plus (yindex times 96) plus x // find the current byte based on the vertical offset
        Screen[xindex]=Screen[xindex] bitwise or (1 shifted left by shiftby) // set the pixel to on state
End Function
```

You will need to implement this pseudo function in your lcd.c and if you implemented it correctly calling

```
Set_LCD_Pos(0,0);
LCD_set_pixel(0, 0)
Set_LCD_Char(<screen_tag>);
while(1){}
```

Should render the top left pixel on.

Once you have completed all of these steps, compile your code to ensure everything is in working order.

At this point you should be ready to begin the lab, good luck and have fun!

Some Helpful Advice:

Note is recommended that you look over the datasheet for the ST7579 found at http://www.tianma.com/web/uploads/controller/20080316012510_ST7579_V0.9a.pdf

you should also look over the documentation from Renesas about Glyph found at http://www.renesasrulz.com/servlet/JiveServlet/previewBody/1685-102-1-1614/Generic%20API%20for%20Graphics%20LCD%20v1.00.pdf

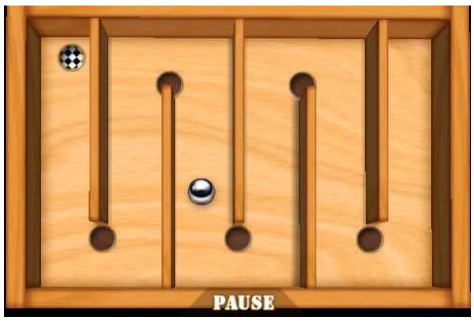
you should also look over the documentation from Renesas API found at http://www.renesas.eu/media/products/software_and_tools/introductory_and_evaluation_tools/european_pdfs/RPDL_RX62N_API_UsersManual.pdf

and you should also look over the accelerometer datasheet found at http://www.sparkfun.com/datasheets/Sensors/Accelerometer/ADXL345.pdf

Prelab Questions:

1. Where should your “working directory” be located when using the lab computers?
2. Where is the first place to look for help with labs in this class?
3. What is the true width and length in pixels of the LCD on the Renesas RX dev board (you will need to find this with your LCD_set_pixel command) ?
4. What is the problem with using the Glyph y offset command?

Laboratory Assignment Overview:



Labyrinth Game

Rolling a ball on a flat surface

It is the purpose of this lab to introduce you to a new piece of peripheral called an accelerometer (which is a device that measure acceleration) that came with your RX development kit.

Before beginning the lab a few fundamental concepts should be reviewed

You might recall from physics class the classical gravitational acceleration equation is

$$g = -\frac{GM}{r^2}$$

Where g is the barycentric gravitational acceleration at a point in space, G is the gravitational constant of the universe, M is the mass of the attracting object, and r is the distance between the two objects.

You might also remember that the gravitational acceleration for objects on earth is around

$$a(t) = 9.81\frac{m}{s^2} = 1g$$

Hopefully also remember, that a relationship between acceleration a(t), velocity v(t), and position u(t) exist

$$a(t) = \frac{dv(t)}{dt} = \frac{du^2(t)}{dt^2(t)}$$

$$v(t) = \frac{du(t)}{dt}$$

But to make things easier to read and write let's take the differential equations that were in Leibniz's notation (that's the d/dt notation) and rewrite them in Lagrange's notation (prime notation)

$$a(t) = v'(t) = u''(t)$$

$$v(t) = u'(t)$$

At this point you might be wondering why we are covering the acceleration due to gravity and the relationship between acceleration, velocity, and position. The conceptual answer is, we are trying to use the 3-axis accelerometer on the RX development board to determine the gravitational acceleration in a particular direction (X,Y,Z) and use this acceleration information to move a virtual ball around a flat surface.

In order to accomplish this task, we will need to get gravitational acceleration information from the 3-axis accelerometer and solve 6 ordinary differential equations (ODE) - initial value problems (IVP) in order to find the velocity of the ball and the position of the ball on the surface of the cone.

While this might sound complicated, the truth of the matter is, you have most likely preformed this task before using a numerical method that simplified the ODE-IVP into an algebraic equation and dint even realize it.

Without further ado let's introduce the Euler's Method of solving an ODE-IVP

$$f(i+1) = f(i) + \Delta_t f'(i)$$

To understand how this equation works, you must first understand that when we work with an embedded system that we are no longer utilizing a function that is continuous in time which is typically represented by the variable t, but rather a discreetly sampled version of the same function that consist of n number of samples that is evaluated for at particular sample called i.

The Euler's Method fundamentally states that the future value of the function f is equal to the current value of the function f plus the amount of time that passes between each sample times the current value of the functions derivate.

Application of this method for our given problem results in the formulation of 6 equations that describe how velocity and position change as a function of acceleration.

The velocity equations for the X,Y,Z axis are

$$v_x(i+1) = v_x(i) + \nabla_t a_x(i)$$

$$v_y(i+1) = v_y(i) + \nabla_t a_y(i)$$

$$v_z(i+1) = v_z(i) + \nabla_t a_z(i)$$

While the X,Y,Z position equations are

$$x(i+1) = x(i) + \nabla_t v_x(i)$$

$$y(i+1) = y(i) + \nabla_t v_y(i)$$

$$z(i+1) = z(i) + \nabla_t v_z(i)$$

It is important to not overwrite your current function value with your future function value until you have solved all of the equations. Also it is important to size your step size delta t correctly. A time step that is too small will make the simulation respond very slowly to changes in the accelerometer, while a large time step will make the simulation overact to accelerometer movements.

Note: it is also important to check your (X,Y,Z) position and ensure it is within the boundary of your lcd. A simple if than statement can be used to accomplish this task, however, forcing a position in either the (X,Y,Z) direction to be a particular value should only be done if you recalculate the current velocity and future velocity for that direction.

For example if the future value of position x is less than the LCD screen we can force the x position to be zero to prevent the ball from rolling beyond the screen but we should do the following to prevent the velocity from increasing which after a long period of time could result in unresponsive behavior.

First we set the future value to the boundary

$$x(i+1) = 0$$

Next we recalculate the current velocity since it's a function of the future position

$$v_x(i) = \frac{x(i+1) - x(i)}{\nabla_t}$$

Then we recalculate the future velocity since the current velocity changed

$$v_x(i+1) = v_x(i) + \nabla_t a_x(i)$$

All of this would have to be done inside the if past boundary then statement

It should be pointed out that the Euler's Method is not the most accurate method for solving an ODE-PVT problem and other methods do exist, which implies that if you wish to implement another method you are free to-do so (trapezoidal, Rk2,Rk3,Rk4,Rk5) are all valid ways to solve the numerical ODE. Euler is the simplest and is recommended.

In order to make our ball roll around the surface of a cone, we need to formulate the equation of a cone and implement it as our z boundary condition.

At this point you should have enough information to begin the lab.

Laboratory Assignment:

1. Do the prelab activity described above.
2. Read over the mathematical equations provided
3. You will now modify the existing prelab code to create the labyrinth game.
4. Next you will need to figure out how to communicate with the accelerometer. Hints have been provided in the prelab activity

   Note that accelerometer_demo.c describes the "accelerometer_demo_update" function and the riic_master_main.c calls this function every 100ms using

   ```
   cmt_periodic_callback_set(&cmt_callback);
   cmt_start(TIMER_COUNT);
   ```

   You should strongly consider using the volatile keyword in your accelerometer global declaration and use the float storage type to avoid register errors later. For example I defined the following global variables in accelerometer.c
   `volatile float adjusted_X, adjusted_Y;`
   to hold my accelerometer data.

   NOTE: adjusted_z is called only if ACCELEROMETER_DEBUG is defined. Currently it is defined in the accelerometer_demo.c file. You can use it to debug your code. However, its definition has to be commented when running the Labyrinth game as it is not going to be used in the game.

   Also note that the cmt_callback function in riic_master_main.c file calls for display of the temperature sensor. This part has to be commented when running the labyrinth game as we are not going to use the temperature sensor data in the game.

   Also to prevent HEW related casting bugs you might want to consider modifying accelerometer_demo_update (which you will have to gut some of the functionality you do not need) to have the following

   (Code snippet)

   ```
   adjusted_x = (float)(g_accel_x - g_accel_x_zero);
   adjusted_y = (float)(g_accel_y - g_accel_y_zero);
   ```

   Note that the casting term (float) was used to avoid nasty short to float data errors.

5. At this point you need to add the screen buffer character provided on the embedded website to font_bitmap.c and add a function to lcd.c called LCD_set_pixel that will toggle a pixel on in the empty screen character as being on. Pseudo code was provided in the prelab.

Note you will need to extern in the character in lcd.c by

```
extern uint8_t Screen[];
```

6. Next you will need to write a function that will clear the screen buffer as an pseudo example

```
Function Clear_Screen()
      define lp=0
      for  lp from 2 to 2+96*8  increment by 1
            Screen[lp]=0;
      End for
End Function
```

7. Next you will need to write a function in lcd.c that will draw a circle on the screen buffer whose center is at (x,y) with a radius r. I recommend that you look into the Midpoint circle algorithm; in particular the raster circle algorithm works well. Note be sure you cite any sources used to accomplish this, (NOTE: other students outside of your group cannot be used as a reference). Write the function and pass parameters x, y and radius. For the ball, parameters x and y will vary to denote movement of the ball while the radius will be fixed. This function will be used later to draw the sink holes.
8. Add a code to the accelerometer_demo_update function that will toggle a "draw" flag. You have to make the "draw" flag accessable to the main function (in riic_master_main.c) as well as the accelerometer_demo_update function (in accelerometer_demo.c).
9. In the main loop you should write a function that will draw the contents of the screen buffer to the LCD similar to the following pseudo code

```
While Loop // the main loop
      If draw==1
            Set_Font_Bitmap()
            Lcd_clear()
            Clear_Screen()

            // Code that draws on the screen goes here

            Set_LCD_Pos(0,0)
            Set_LCD_Char( custom character number)
            draw=0
      End if
End While
```

10. At this point test your circle algorithm, pixel drawing algorithm and ensure that it is correctly displaying on the LCD screen.

11. At this point you are ready to implement the Euler methods discussed above in the main while(1) loop, first you will need to define variables to hold your ball's current position and velocity and future position and velocity.

Note because there are a sizeable number of floating point variables required, it is recommend that you make these variables global, as stack corruption can occur if too many floats are defined locally. There is nothing worse than having your code correct but the debugger reports the wrong answer because of a complier quark. For your sanity, I recommend that you define all of your floats globally!

I recommend naming the current position x y z , the current velocity as vx vy vz , the future position x_n y_n z_n , and the future velocity as vx_n vy_n vz_n

12. Next you will need to define your simulation step size as a variable (delta_t)

    I recommend a step size of 150E-3 to start with.

13. Next you need to define 3 floating point variables that will translate the value from the accelerometer into a useable acceleration.

    The best way to figure this out is to use the debugger to observe the accelerometer 3 axis values at their maximum exposer to gravity (IE flip the accelerometer so its axis is facing the ground in 6 different directions for the X,Y,Z axis and record those values)

    You can then take the distance between each axis values and divide 9.81 by this number to find your acceleration per accelerometer unit.

    Typically the accelerometer will report that 1g is between -40 to 40 on an top and bottom of the axis

    The conversion coefficients I used were
    float gx=.3111;
    float gy=.297;
    float gz=.297;
    Yours will be different, but these will give you a baseline.

    You could also take an average of these numbers and use a single number for all conversions.

14. Create variables that will hold your current acceleration
    Note: I recommend ax,ay,az

15. Inside the draw function, since we only want to update our position at a rate of 100ms to avoid having a fast moving ball calculate ax ay az by multiplying your measured accelerometer value by your conversion coefficient(s).

Note you will need to multiply ax and az by a negative 1.0 to make the simulation forces respond as expected since the board's accelerometer is mounted differently than our simulation equation expect.

Note always use 1.0 not 1 to let the complier know that the number is not an integer to avoid casting errors.

16. Calculate the future velocity (vx_n vy_n vz_n) and position (x_n y_n z_n) using Euler's method, note we are assuming all initial conditions are zero.

    To provide one Euler example vx_n is found by
    vx_n=vx+ delta_t *ax;

17. Check x_n, y_n, and ensure it is inside the lcd boundary, if it's not then change the value and update velocity and future velocity.

    note you will need to recalculate velocity and future velocity if you force the position to a fixed value as discussed earlier.

18. You can now safely set the future position equal to the current position.

    or in other words

```
vx=vx_n;
vy=vy_n;
vz=vz_n;
x=x_n;
y=y_n;
z=z_n;
```

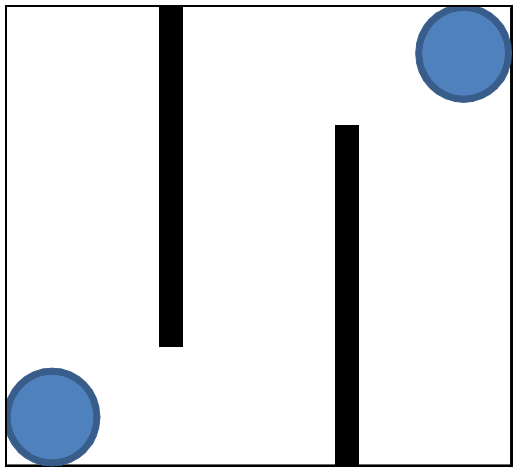
19. Use your screen draw function you created earlier to draw a circle at the calculate x,y value with the radius.

20. Test your program and watch the ball roll around the screen. Now that you have the ball rolling on the screen we are ready to place obstacles and sink hole and insert various screens.

21. At startup, the LCD should show 'Labyrinth', 'Level: 1' and the name of team members. Press SW1 to proceed.

22. The maze should have 2 walls and 2 sink holes as shown. You will have to design this by updating your "Screen" variable initialized in the bitmap table. The walls are about 3 pixels thick. The sink holes should touch 2 edges of the screen. I used radius = 8 for the sink holes. You will

have to figure out the appropriate center points for the two sink holes. You can use the function you created earlier to draw the sink holes. The length of the walls should be enough to allow the ball to easily slip between the sink hole and the wall.



23. The initial position of the ball should be at the top left corner. You need to make the ball traverse through the maze to the final position viz the bottom right corner. Once the ball reaches the finish line, you need to erase this screen and display the next level. Press SW1 to start this new level. You will have to define a variable to store the value of the current level. For the next level you can increase delta_t by a small value to make it more the level challenging (or you can leave it to be the same).

24. Next, you need to write the code to ensure that the ball does not traverse through the walls. You can use 'if' conditions for this. You will have to redefine the values of x_n and y_n if the position of the ball overlaps that of the walls.

25. Now, you need to write the code to determine if the ball has fallen into a sink hole. To show the ball falling into the sink hole, you can decrease the radius of the ball for a few iterations before freezing the screen. SW3 should then reset the level and take you back to the screen showing the level number.

26. Next add some code that will cause the green LEDs to flash in a pattern when you complete a level and red LEDs when the ball falls into the sink hole.

27. Bring the new board to the lab TA and demonstrate the new code (without the HEW application running). When the TA checks your board, he will also take your lab report. You will not need to include a printout or soft copy all of the code – just “snippets”.

Lab Report:

1	First screen shows Labyrinth, Level 1 and names of team members.	
1	LCD is rendered using a screen buffer character	
2	LCD is rendered at a proper refresh rate	
3	Maze is drawn on the LCD	
4	Ball does not traverse through the walls.	
5	If ball falls into a sink hole, SW3 resets the level and takes you to a screen showing the correct level number.	
7	LEDS blink correctly when the ball hits the wall/ falls into the sink hole	
8	Group added their own unique spin to the project	

Include in your lab report observations and procedure like the following:

The general learning objectives of this lab were . . .

The general steps needed to complete this lab were . . .

Some detailed steps to complete this lab were . . . .

1. Step one

2. Step two

3. . . . .

Code generated or modified to complete this lab...

No need to include all the files for the lab. Just include the modified code.

Some important observations while completing/testing this lab were . . .

In this lab we learned . . . .