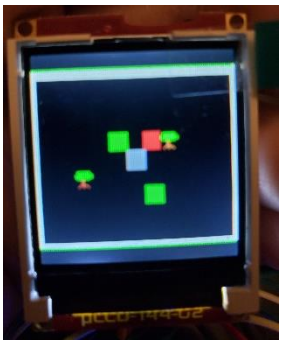
GTEmblem: A Strategy Based Role-Playing Game

The goal of this project is to build a role-playing strategy game on the Mbed platform, therefore fulfilling your lifelong goal to be a video game designer. Since the very first video game platforms, top down role-playing games (RPGs) have held an important place in the video game corpus. This particular game is modeled after the Fire Emblem series. In this project, you will build a handheld top-down RPG game using the gaming circuit constructed during HW3 and your P2-1 hashtable implementation.



In your top down RPG, the protagonist will be controlled by tilting the game board; the accelerometer will be used as the input for character motion. The buttons will be used to trigger actions in the game. Crucially, the map area of the game will be much larger than the area you are able to display on the screen. Objects on the map will be stored in a hash table, and you will look up the correct locations to display them at every game update.

There are several basic features that your game must implement in order to receive baseline credit; advanced features are an opportunity to earn full and possibly extra credit. The list of basic features and examples of some advanced features are given below. (See rubric for grading details.)

Basic Features

These features are for baseline credit and constitute a functional, but minimal, strategy RPG.

Player Motion & The Map

The Player character in this top-down RPG moves around in the direction of the tilt of your breadboard, as measured by the accelerometer.

The Map for your game is the world that the character moves around in. The Map is made up of individual tiles, and a grid of 11 x 9 tiles is displayed on the screen at any time. The Map should be at least 100 x 50 tiles and have walls around the edges to prevent the Player from leaving the map. The Player should not be able to walk through the walls.

You will need to populate the Map with Items relevant to the game - these Items can include scenery, walls, Non-Player Characters (NPCs), objects, stairs, etc. Implementation details for the Map are discussed in detail later in this document.

World Interaction

There are several buttons available in the hardware setup created in HW3. Two of them have required functions:

- **Action button:** This is the primary way the player interacts with the world. Pressing this button with the cursor over the character will select the character. Using the accelerometer, you can move the cursor up to the character's range. Once you've decided on where to move the character, you can press the action button again to confirm and the character will be moved to the cursor location.
- **Back button:** This serves two functions:
  First, after you've selected a character, but before you've confirmed where to move the character, you can use the back button to deselect the character, returning it to its position before it was selected.
  Second, if you don't have a character selected, you use this button to end your turn.

You are free to use the other buttons for any features of your choice. These might be useful to open a menu, for example.

The Game

The point of this game is to defeat all of the enemy characters. In the provided baseline program, your three characters are viewable in the starting view as red squares. In order to win, the Player must move their characters towards the enemy's characters (green squares in the baseline program) and defeat them in combat. Each character has attributes of attack, defense, movement range, and hit points (health). These attributes are point values that are used to determine character movement and attack capability as described in the basic feature summary below.

Basic Feature Summary:

- Accelerometer moves the cursor
  - The cursor can move anywhere within the bounds of the map; it is not constrained by obstacles
- Selecting a character:
  - Pushing the action button when the cursor is over a character should select the character for movement
  - Pushing the back button after a character is selected should deselect the character and allow the cursor to freely roam the map
  - Each player should only be able to select their own characters
- Moving a character:
  - Selecting a new tile after a character has been selected should move the character
  - Movement of the cursor after selecting a character should only be allowed up to the character's movement range
  - Each character can only be moved once per turn
  - Characters cannot move through obstacles such as trees, rocks, and walls

        - To be clear: the map boundaries present as an external wall impassable to both the characters and the cursor, which are distinct from internal wall, tree, and rock obstacles that are impassable to just the characters.

- Attack
    - Each character has an attack range of one (characters can only attack each other when they are next to each other)
    - When a character is placed next to an enemy, the character should attack the enemy, and the enemy should attack back provided the enemy has not been defeated
    - Each attack should lower the enemy's hit points, with the damage dealt = Attacking character's attack – defending character's defense. The damage dealt cannot be negative.
        - For example, assume character A has an attack of 10, a defense of 3, and 15 hit points, while character B has an attack of 8, a defense of 4, and 10 hit points. Character A is moved next to character B. Character A hits character B for 10-4 = 6 damage, leaving character B with 4 hit points left. Character B his character A for 8 – 3 = 5 damage, leaving A with 10 hit points left.
- Taking turns
    - Pushing the back button while a character is not selected should end that player's turn
    - After a player ends their turn, a speech bubble should state that their turn is over, and control should be handed to the other player
        - You as the player should be able to play as both players (when it is their turn)
- The map must be bigger than the screen (at least 100*50 tiles)
    - It should include at least 10 obstacles such as walls, trees, and rocks that a character cannot pass through.
- Art includes at least one sprite
- Display game-over screen when either player is defeated (player loses all their characters)

Advanced Feature Examples

Advanced features in this project are open ended and allow you to make the game your own. The features listed below are all acceptable, but this is not an exhaustive list. Each extra feature is worth +5 points, and you must tell your grader before the demo begins which extra features you used. Other features that are at or above the difficulty level of those listed here are acceptable *but must be approved by an instructor or graduate TA*: there will be a pinned discussion topic on Piazza to confirm extra features. If you intend to use features that aren't listed here, start a follow-up on this discussion to clear it with the GTAs or instructors before grading begins.

- **Visual representation of range**: When a character is selected, provide a visual representation of their movement range. You must have some obstacles in your map, and your visual representation should reflect the loss of movement to obstacles
  - For example, if a character has a movement range of seven, and there is a wall three spaces in front of the character, the movement visualization should reflect the fact that the character would use all of their movement to move in front of the wall (because they're moving around the wall)
- **Enemy AI:** Instead of allowing the user to control both players' characters in turn, implement an enemy AI and only allow the user to control one player's characters
  - For 5 pts, at least one enemy character should remain in their starting location until a player's character steps into their attackable range (attackable range being their movement range + their attack range). When this occurs, the enemy should move and attack the player's character
  - For 5 pts, at least one enemy character should always move towards the nearest player's characters to attack
- **Weapons**: Implement weapons for the characters. Each character can carry one or more weapons (up to a limit of your choice). Each weapon should have a strength value, which is added to a character's attack when calculating the damage dealt. Each weapon should also have an attack range, and at least one weapon must have an attack range of more than 1. This will enable characters to attack while not being directly next to their target. For example, if character A has a bow with an attack range of 2, and is moved two tiles away from character B that has a sword with an attack range of 1, character A should damage character B, but character B should not damage character A.
- **Implement "avoid":** Each character should be given an avoid stat between 0 and 100, and a skill stat between 0 and 200. When characters attack each other, hit percentages should be calculated as skill – avoid (with anything over 100 being forced to 100, and anything below 0 being forced to 0). Random numbers should be generated that implement the likelihood of the attack actually hitting the other character. If the attack misses, then no damage should be dealt.
  - For example, suppose character A has an avoid of 45 and a skill of 95, while character B has an avoid of 30 and a skill of 130. Character A moves to attack character B. Character A has a 95 – 30 = 65% chance of hitting character B, and character B has a 130 – 45 = 95% chance of hitting character A.
    Two random numbers between 0 and 100 are generated; the first random number is 50, which is between 0 and 65, meaning character A hits character B, dealing damage. The second random number is 99, which is not between 0 and 95, so character B misses character A and deals no damage.
- **Implement healing items:** Characters should be allowed to hold healing items which should restore their hitpoints. You can choose to either allow the player to determine when the healing items should be used, or have the item be used automatically when the damage to a character's health exceeds the healing capability of the item.
  - For example, suppose you implement potions which heal 10 hitpoints, and decide to have the item used automatically when the damage to a character's health exceeds the

healing capability of the item. Character A holds a potion and starts the game at 25 hitpoints. They're attacked by character B and is left with 18 hitpoints. Because the potion heals 10 hitpoints, but they've only lost 7 hitpoints, the potion isn't used yet. Then character B attacks character A again, leaving character A with 11 hitpoints. The potion is now used, because character A has lost 14 hitpoints in total. After the use of the potion, character A has 21 hitpoints.

- **Character info menu:** *<This is worth 2.5 points unless you implement items/weapons>* Implement a menu to display information about a character. When you push the third button with the cursor over the character, it should display the character's stats – health, attack, and defense, along with any other new stats you implemented
  - If you implement healing items and or weapons, show them as part of a character's inventory for another 2.5 pts
- **Implement different terrain types:** Terrain types could boost defense, attack, or even implement healing at the end of each turn.
- **Difficulty modes:** Give the player the option to change the difficulty of the map. For example, one player could start with more health, better weapons, stronger attack, etc.
- **A starting page** that explains the game (and any additional features added)
- **Sound effects** for different actions (attack, movement, etc)
- **Animation** for attacks or movement. Animation could be implemented as cycling through sprites or some other reasonable method
- **Saving the game state** and being able to read back in the saved game state
- **Multiple distinct levels** (you must make substantial changes to the level, such as map structure, number of characters, character stats, etc)

P2-2 Technical Reference

In this project, you'll be combining hardware interface libraries for an LCD screen, pushbuttons, speakers, and an SD card reader into a cohesive game. The shell code has several different modules. This document is intended to be a reference for various technical considerations you'll need when implementing your game.

Hash Table

The game will make use of your HashTable library, implemented in P2-1. In order to use this library within the Mbed environment, the easiest strategy is to simply copy and paste the code into the correct files. The shell project has two files already for this purpose: hash_table.cpp and hash_table.h. Copy your completed code from P2-1 into these files before starting anything else.

USB Serial Debug

Debugging is an important part of any software project, and dealing with embedded systems can make debugging difficult. Fortunately, there is a built-in serial monitor on the Mbed that allows you to see printf-style output from the Mbed on your computer. The tutorial to set that up can be found here:

https://os.mbed.com/handbook/SerialPC

The Serial pc object described in the tutorial is already set up for you in globals.h, so you won't need to declare it again. Any file that includes this header can print to the USB like so:

pc.printf("Hello, world!\r\n");

Game Loop Overview

The basic structure for organizing a video game is called this *game loop*. Each iteration of this loop is known as a *frame*. At each frame, the following operations are performed, typically in this order:

- Read inputs
- Update game state based on the inputs
- Draw the game
- Frame delay

You'll be implementing parts of each of these steps, along with setting up the game loop to call them in the correct order. The game loop shell code with timing already implemented is in main.cpp.

Read Inputs. Reading user input for a frame happens only once during the frame. This serves two purposes. For one, it isolates the part of the code that has to deal with the input hardware to just the read_inputs function, allowing the rest of the code to deal with only the results of the input operation. Secondly, it ensures a constant value of the "true" input for a particular frame. If there are multiple parts of your game update logic that have to interact with the inputs, it is convenient to know that these inputs are guaranteed to be the same.

Update game. This is where the magic happens. Based on the current state of the game -- where the Player is standing, whether the player is holding the key, what the NPCs are doing -- you compute what the next state should be, based on the inputs you've already measured. For example, if the accelerometer is tilted toward the top of the screen, the Player should move up in the map. This is where most your development will be focused.

Draw game. With the state updated, you now need to show the user what changed by drawing it to the screen. This step is discussed in much more detail below, but for now you'll want to know that the entry point to this portion of the code is called draw_game.

Frame delay. By default, loops in C run as fast as the instructions can possibly execute. This is great when you're trying to sort a list, but it's really bad for games! If the game updates as fast as possible, the user might not be able to understand what's happening or control the character appropriately. So, we introduce a delay that aims to make each frame take 100ms. The time for all the proceeding 3 steps is measured, and the remaining time is wasted before starting again. If more than 100ms has already passed, no additional delay is added. As you're developing your game, be careful that your frames don't get too long, or the feel of your game will degrade.

Map Module

In order to think about updating the game state (moving the player) and drawing the screen, we first need some way to represent the world. This module accomplishes that task.

The map is a two-dimensional grid whose origin is at the top-left corner of the world. The X coordinate increases toward the right, and the Y coordinate increases toward the bottom. This left-handed coordinate system is chosen for consistency with the graphics; see that section for more details. The finest granularity of the map is a single grid cell; the player moves from cell to cell, and each cell contains at most one MapItem. If the cell is empty, then that cell is free space on the map.

The map in this game is represented by a collection of MapItems held in a HashTable. The keys in the HashTable are (x,y) pairs. The data in the HashTable are all of type MapItem, defined in map.h. So, for example, if you access the key "(10, 23)" in the HashTable and the data is a MapItem whose type is WALL, then the player should not be able to walk into that cell.

The shell code is written so that the use of the HashTable is hidden inside the map module; that is, the hash_table.h is only included from map.cpp, and the HashTable functions are only used internally to that module. The public API of the map module does not expose the HashTable, since this is an implementation detail of the map and does not affect the rest of the game. This hides the complexity of the HashTable (questions like "what is the best hash function?" and "how do I map (x,y) pairs into integer keys for use in my hash table?") within the map module itself, and simplifies the rest of the game logic.

The public API for the map module is given in map.h. All functions and structures are documented there. There are functions for accessing items in the map (e.g. get_here, get_north), modifying the map (e.g. add_wall), and selecting the active map. *You are encouraged to add more functionality to this API as you deem necessary for your game.* The point of an API is to be useful to the programmer; if these functions are insufficient, add more!

Map Items. This is the basic unit of the map, and is the underlying type of all the void* data in the map HashTable. Each MapItem has an integer field, type, which tells you what kind of item it is. This allows you to store different information in the map, such as the location of walls and the location of trees, using the same data structure. Each MapItem also has a function pointer of type DrawFunc, that will draw that MapItem. Its inputs are a pixel location (u,v) of the tile. Finally, each MapItem has two additional parameters: an integer flag, walkable, that describes if the player is allowed to walk on that cell; and a void* data for storage of any extra data required during the game update. Walls probably don't need extra data; NPCs or stairs or the door might.

Two-dimensional keys. As you implemented in P2-1, the HashTable accepts only unsigned integers as keys. However, for this application you need to use two integers (the X & Y coordinates) as the key. In order to do this, you need to have a function to map these coordinates unambiguously into a single integer. This function is called XY_KEY, and is private to map.cpp. You then also need a hash function that will take this key as normal and produce a hash value for bucket selection in the HashTable.

The Active Map. All operations in the Map API use the "active map." In the shell code, there is only one active map. The function get_active_map returns this map, and the function set_active_map does nothing. You may modify these to allow selection between multiple maps. Once an active map is selected using set_active_map, all other functions (accessors and modifiers) will use the currently active

map. Only one map can be active at a time; setting a new active map implicitly deactivates the previous active map.

Graphics

The graphics module houses most of the drawing code for the game. This includes all the drawing functions for the various MapItems. The entry point for drawing the screen under normal operations (not in a speech bubble) is the draw_game function. This section describes how that function accomplishes drawing the tiles, and various ideas to consider as you extend this function for your own game.

Coordinate Reference Frames. There are several relevant coordinate frames for this game. The first we have already covered: the map frame. This frame's coordinates are labelled (x,y) and its origin is the top corner of the map. X increases right, and Y increases down. All frames in the game are this left-handed orientation.

The next frame is the local drawing frame. This frame is centered on the Player, and ranges from (-5,-4) to (5,4), i.e. it is an 11x9 grid of cells. This frame is iterated in draw_game and each cell in drawn in turn using the DrawFunc from the map, or a draw_nothing function if there is no MapItem. The coordinates of this frame are labelled (i, j).

Finally, there are the pixels on the screen. This frame has its origin at the top-left corner of the screen. The screen is at 128x128 array of pixels. The coordinates of this frame are labelled (u,v). Each cell in the map is 11x11 pixels.

Drawing functions. As noted above, each MapItem has an associated DrawFunc that knows how to draw that item. These functions take as input a (u,v) coordinate for the top-left pixel of the tile, and draw an 11x11 image that represents the MapItem. An example is the draw_wall function, in graphics.h.

You will need to add more draw functions as you add more types of MapItem. You are free to implement these in whatever way you like, using the full power of the uLCD library. However, a simple way to do this has been given to you. The draw_img function takes a string of 121 (= 11 * 11) characters, each representing a pixel color, and translates that into a BLIT command to draw those colors to the screen. You can use this function to make nice graphics very simply by defining a new string that represents the image you want to draw. This is the recommended method for generating art for your game.

Drawing Performance. The screens are notoriously slow to draw, and the length of time it takes to complete a drawing command is proportional to the number of pixels that it changes on the screen. So, the drawing code goes through some hoops to make sure that things keep moving quickly. In particular, the drawing code requires not only the current player position (Player.x and Player.y) but also the previous position (Player.px and Player.py), in order to determine what has changed on the screen. If an element on the screen has not changed, it is not redrawn. This saves time and make the game update more quickly. You'll need to be careful with this as you decide how many items to put in your map and how to draw the new items you add.

You must design, implement, and test your own code. There are many, many ways to code this project, and many different possibilities for timing, difficulty, responsiveness and general feel of the game. Your project should represent your interpretation of how the game should feel and play. Any submitted project containing code (other than the provided framework code and mbed libraries) not fully created and debugged by the student constitutes academic misconduct.

Mbed reference materials: http://ece2035.ece.gatech.edu/readings/embedded/index.html