The goal of this project is to **build a Quest 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. Such notable titles as *The Legend of Zelda*, *Pokémon Diamond, Final Fantasy* are exemplars of this style. In this project, you will build a handheld top-down RPG game using the gaming circuit constructed during HW3 and your hash-table implementation from P2-1.

In your top-down RPG, the protagonist will be controlled using the Nav Switch as the input for character motion control. 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 **baseline features** that your game must implement to receive baseline credit; **advanced features** are an opportunity to earn full and possibly extra credit on the project. The list of basic features and examples of some advanced features are given below. (Also, see rubric for grading details).

Basic Features

These features are for baseline credit and constitute a functional, but minimal, adventure game. Please note that the story, art, and actions available in the game are intentionally vague, and the specific design of the game is up to your creativity. RPG games are extremely varied, and this project is an opportunity for you to make something unique, creative, and more importantly, fun! **Feel free to change the suggested storyline to your own – using your own characters, features, battle, etc.., keeping in mind to meet the basic requirements**. So, have fun with it! 😊

Player Motion & The Map

The ***Player*** character in this top-down RPG moves around in the direction as pressed on the Nav Switch. 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 ***Player*** is always displayed in the center of the screen. The ***Map*** should be **at least 50 x 50 tiles** and should 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 your game - these ***Items*** can include scenery, walls, non-player characters (NPCs), objects, stairs, etc. Implementation details for the ***Map*** are discussed in more detail in the accompanying technical document.

You will most likely need to use all four buttons for this project. You should at the very least have an action button, an attack button, and a menu button.

World Interaction

The essence of this game, as with all good RPG games, is a quest! To win, the ***Player*** must complete a quest by interacting with characters and objects in the world to obtain a ***key*** to pass through a locked ***door***.
A Boss is an enemy that can only be defeated by using non-regular attacks (special attacks and spells). You must defeat a Boss to complete a quest. In the demo, Purdue Pete is a Boss.
A regular enemy can be defeated using regular attacks. In the demo, the ice enemy is an example of a regular enemy.

Plot:
There is none. Make up a story about a fallen hero trying to regain his lost kingdom or something. This is an open ended project so go wild.

YOUR GAME MUST FOLLOW THIS BASIC STRUCTURE (At least for the Basic Features):

1. Talk to an ***NPC*** (*Non-Playable Character)* to start the quest. The ***NPC*** gives the ***Player*** instructions to fight ***Purdue Pete*** (or any boss of your liking) located on a different ***Map*** (in a house, in a cave, in a lake, etc.).
2. The NPC must ask the Player to defeat an enemy and bring them an item that drops once that enemy is killed.
3. ***The Boss*** can only be defeated using a special attack which the NPC will teach him in exchange for the item.
4. The ***Player*** must find its way to the different ***Map Entrance*** (a cave entrance, magic portal, staircase, etc.) avoiding *Map Obstacles* (such as trees, maze, enemies, spikes).
5. In the new ***Map***, the ***Player*** must defeat ***the Boss***, by using special attacks against the wizard. The spells can be activated via a menu or by standing on floor tiles, or any other method you wish to implement. The **only** stipulation is that there **must be more than one choice** for which attack to use.
6. Upon using the special attack, a **dialogue box must appear describing what attack you have used**. Again, there must be more than one choice for what spell/attack to cast and the dialogue box must correspond to the attack used and the scenario.
7. Once ***the Boss*** is defeated, walk to the same NPC again to **complete the quest.** The NPC will give the player a ***key*** as a reward.
8. Find the locked door and use the ***key*** to open it.
9. Walk through the ***door*** and interact with something: a *chest*, a *throne*, etc. Display a " You win!" screen.

All interactions in the quest should be triggered by the action button when standing near the target. If the interaction involves a conversation that supports the storyline, the action button should trigger a speech bubble. For example, to begin the quest the ***Player*** should stand adjacent to the ***NPC*** and press the action button, triggering *Step 1* and displaying a speech bubble with instructions for *Step 2*. Speech bubbles should cover the bottom part of the ***map*** and should scroll when the ***Player*** presses the action button. More detail on their implementation is given in the technical document.

The NPC must say different things in these states:

1. Starting the quest (*Step 1*: First time talking to the NPC)
2. Quest Started but Item not provided to NPC
3. Item Provided and special attack learned but Boss not defeated
4. Boss defeated and Key received by Player

The ***door*** must block ***Player*** motion when *closed* and should be visually different once it is *opened*. (This difference can be as simple as the ***door*** disappearing from the ***map***, or it can be different art showing an *open* ***door***). The ***door*** should give some indication that you're trying to open it, such as a ***speech bubble*** or visual animation, and it should respond differently when the character has/has not obtained the ***key***. Your ***door*** must block the ***Player*** if the ***Player*** doesn't have a ***key***, and it should only open when the player presses the action button while holding the ***key***.

Graphics

You should put some effort into making your game look good! At least **one** sprite is required as a basic feature. A sprite in this context is a tile that's more than just a rectangle. You should be using the `uLCD.BLIT()` or `draw_img()` functions to accomplish this.

A status area is set aside at both the top and bottom of the screen. The top status area should display at a minimum, the current ***Player*** coordinates within the ***map***. These areas can also be used to show information such as quest progress, character inventory, what the action button will do at this location, etc. Be creative!

For clarity, while the ***Player*** is moving, it's a good idea to have some background items on the ***map*** that scroll by as the player moves. The trees in the demo fulfill this purpose. This is not strictly required, but you'll probably want to add it.

Basic Feature Summary

1. Nav Switch moves the ***Player***. Status Bar is displayed at top showing player coordinates
2. Walls block character motion.
3. The first ***map*** must be bigger than the screen (at least 50*50 tiles). The top status area should display at a minimum, the current ***Player*** coordinates within the ***map.***
4. ***Stairs***/***ladders***/***portals***/***doors*** go between the first and the second ***map***. The second map must be visually different from the first map.
5. More than *one* attack to choose from to fight ***Enemies and Bosses***. Bosses can only be defeated using special attacks. Regular enemies can be defeated by using a combination of both types of attacks.
    - To defeat an enemy (Boss or regular enemy), you must go up to the enemy and press the attack button/special attack button.
6. Quest works (***key*** & ***door*** work).
    - The door must not be able to open before the quest is complete

- When you first talk to the NPC, he must give you the quest
- The NPC must ask you to bring them a certain item in order to learn the special attack to kill **the Boss** and unlock the 2nd Map
- **The Boss** must be in the second map
- Use speech bubbles appropriately

7. Display a game over screen when the quest is complete.
8. Use ***Speech bubbles*** in the quest.
9. Art includes *at least one* sprite.
10. Enemies must drop items when defeated.
    - There must be at least two distinct enemy types
    - Each unique enemy drops a different item. For example, in the demo, the ice enemy drops an "ice helm" and the fire enemy drops "rage horn"
    - The Boss must also drop an item
    - Speech bubbles must show when you defeat an enemy indicating what item you just collected.

Running the Demo

To successfully complete the demo, first move the character to the right to find the NPC and press the action button when next to the NPC. The NPC' sprite is defaulted as a yellow and red diamond. The NPC will give a speech bubble that hints about how to defeat Purdue Pete. It will ask you to bring it an "ice helm" from an ice soldier. Go right and defeat the ice enemy and then go talk to the NPC to progress the plot. Walk left to the grayish square that represents a cave entrance. Stand in this region and press the action button. This will transport the player into Pete's evil lair. To defeat Pete, the player must stand on the correct attack tile and press the action button. In this case as hinted by the NPC the correct attack would be a stinging strike denoted by a lightning sprite. Pete will give some dialogue of its defeat and turn into mud. To exit the room, walk to the staircase on the far left and return to the NPC. The NPC will award you with a key and to complete the game walk to the far right until you see a golden gate door. Walk up to the door, press the action key to unlock the chest, and "You Win!" screen will display.

The demo is just an example. You can implement the features any way you like.

Advanced Features

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. Other features that are at or above the difficulty level of those listed here are acceptable. **Each extra feature is worth +5 points**, and you must tell your grader before the demo begins which extra features you used.

There will be a pinned discussion topic on Ed Discussions to confirm extra features. If you intend to use features that aren't listed here, start a follow-up on this discussion in the thread to clear it with the GTAs or instructors before the posted deadline on the thread.

Example Features

- Add a start page.
    - Must include options to view controls, start game, and the name of your game
- Sound effects for interactions / background music.
- Different modes of locomotion (e.g., running, hopping, etc.). They should be visually distinctive.
- Animation for interactions with things in the map (e.g., exclamation mark above NPC when you are talking, apple trees look different when apples are picked off etc.).
- In-game menu (each counts as a separate feature. Pick any number of features from the following)
    - Save the game. (You need to actually save the game and not just have a button that says "Save Game").
    - Show status information. Must at least include play time, and a running count of each type of enemy defeated
- In game inventory with useable items
- Multiple lives and the possibility to lose:
    - Health & stuff that hurts you.
- Mobile (walking) NPCs or wizards (monsters).
- Save the game (persistent over a power-off).
- Bigger objects in the map that block the character. Examples:
    - A very tall tree that hides the character.
    - A feature you can walk behind/under such as a bridge.
- Multiple Bosses to defeat. Each Boss must require a unique and different special attack to defeat.
- Throwable combat items.
    - Animated sword that launches across the screen
- Side quests: Example: NPC has the player solve a puzzle/riddle or perform a task in exchange for a reward.
    - Must have multiple NPCs
- Enemies have health bars and take multiple hits to defeat
- Player levels up / Gains XP which unlock additional abilities
- Have a cutscene in the game
    - The map must disappear and you must have sprites on the screen for any and all items/characters in the scene. The cutscene must take place and complete without any user input.
- Objects other than enemies also drop items when attacked

P2-2 Technical Reference

In this project, you'll be combining hardware interface libraries for an LCD screen, pushbuttons, and possibly, 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 Hash-Table library, implemented in P2-1. 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

For Windows users, please refer to https://os.mbed.com/handbook/Windows-serial-configuration on more information on configuring Serial PC, and install TeraTerm. For MAC/Linux users, this process is a bit more baked in. Please refer here for necessary information: https://os.mbed.com/handbook/Mac-or-Linux-terminals.

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");
```

Please, note that in ***globals.h*** a `F_DEBUG` flag is defined and set. You can use this flag to run your game in *debug mode* if it is *set* (1), or normal mode if it is *reset* (0). You can also use it to guard your `printf(…)` statements and toggle them off or on.

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:

1. Read inputs
2. Update game state based on the inputs
3. Draw the game
4. 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 is 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 must 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 must 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 of 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

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 the graphics section for more detail). 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 Hash-Table. The keys in the Hash-Table are `(x,y)` pairs. The data in the Hash-Table are all of type ***MapItem***, defined in *map.h*. So, for example, if you access the key "(10, 23)" in the Hash-Table 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 Hash-Table is hidden inside the ***map*** module; that is, the *hash_table.h* is only included from *map.cpp*, and the Hash-Table functions are only used internally to that module. The public API of the ***map*** module does not expose the Hash-Table, since this is an implementation detail of the ***map*** and does not affect the rest of the game. This hides the complexity of the Hash-Table (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*** Hash-Table. 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.

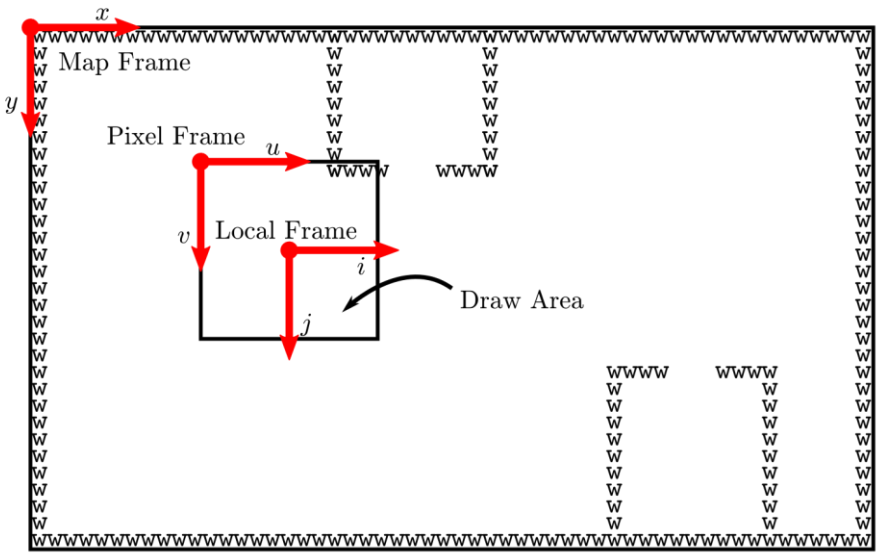
Two-dimensional keys: As you implemented in P2-1, the Hash-Table accepts only unsigned integers as keys. However, for this application you need to use two integers (the X & Y coordinates) as the key. In order to do this, you need to have a function to map these coordinates unambiguously into a single 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 Hash-Table.

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. As an advanced feature, 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 however 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`), 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 on your map and how to draw the new items you add.

Project Submission

For your solution to be properly received and graded, there are a few requirements.

1. Follow the instructions to export your project **(.c and .h files)** files as described in the Project Setup on Keil Studio Online IDE documentation.
2. Add your exported archive file into a zip folder titled P2-2.zip.
3. Upload **P2-2.zip** and **P2-checklist.txt** to Canvas before the scheduled due date. Do not put the checklist inside the zip folder. It must be a separate upload. If you have implemented advanced features from the Ed post, your checklist must include screenshots of your request and the GTA approval from Ed.
4. Upload your compiled bin file and name it P2-2.bin
5. Make a video demo-ing your features and paste a youtube link in the submission comments. Please do not upload raw video files.

You should design, implement, and test your own code. Any submitted project containing code (other than the initial setup code) not fully created and debugged by the student constitutes academic misconduct. We are doing plagiarism checks on assignments across sections of ECE2035, past and present.

In addition, the materials provided by the instructor in this course are for the use of the students currently enrolled in the course. Copyrighted course materials may not be further disseminated. Project code files must NOT be made publicly available anywhere.