LAB 05:
Structs, Object Pooling, and DMA

Provided Files

- `main.c`
- `gba.c, gba.h`
- `game.c, game.h`
- `font.c, font.h`
- `sound.h`

Files to Edit/Add from Lab00

- `Makefile`
- `.vscode`
    - `tasks.json`
- `main.c`
- `gba.c`
- `game.c, game.h`

Instructions

In this lab you will be completing several different TODOs, which will complete a simple Brick Breaker game. Each TODO represents a component of the game, and is broken down into sub-TODOs. Your code will likely not compile until you complete an **entire TODO block**, at which point the game should compile with the new component added and working.

After you download and unzip the files, add your Makefile and your tasks.json, and then **compile and run** it. At this point, you should see the start screen. Complete the TODOs in order, paying close attention to the instructions.

TODO 1: Score

We have provided `drawChar` and `drawString` for you. We want to be able to see our current progress to victory during the game state.

TODO 1.0

- In `goToGame()`, draw "score:" at (180, 1) using the color `HOTPINK`.

TODO 1.1

- In `game()`, use `sprintf()` to save the current score (`score`) in the text form to a character array.
    - The char array has already been created for you, called `buffer`. `sprintf` is a function in `sdio.h`, which we have `#include`'d for you. It returns `null`, and works like this:
        - `sprintf(ptrName, formatterString, variables, …)`
        - This is exactly like `printf`, but with an extra argument at the beginning (the `ptrName` to save to).

TODO 1.2

- We want to erase the previous score before we draw the new one! Draw an `OFFWHITE` rectangle at (220, 1) that is the size of a character.
    - ***Hint:*** Look back at `font.c` – how wide and how tall is each character?
- Then, draw the score at this location (using `drawString` with `buffer` as the character array parameter input) in `PORTAGE`.

TODO 1.3

- Inside of `main.c`, update the `start()` function to increment the seed variable (`rSeed`) every frame.
- Then, when the player presses `START` to transition to the `GAME` state, use `srand()` to seed the random number generator using `rSeed`.

Build and run. You should be able to travel to the game state and see the current score. Right now, it will be 0. In the next TODO, we will add blocks that can be broken to increase the score.

Note: Why did we draw "score:" in `goToGame()`, but the actual number in `game()`? That's because the "score:" text does not change, so we only need to draw it once. The

actual score, however, can change at any time, so we have to account for that possibility every frame.

TODO 2: Blocks

Now we will add the bricks to be broken! This will take place in `game.h` and `game.c`.

TODO 2.0

- In `game.h`, create the `struct` for the blocks, typedef-ed `BLOCK`. `BLOCK` should have the following members: `x`, `y`, `oldx`, `oldy`, `xVelocity`, `width`, `height`, `color`, `active`, and `erased`. All of the `BLOCK` members should be of type `int` except `color`, which should be an `unsigned short`.

TODO 2.1

- In `game.c`, uncomment `initBlocks()`.
- Inside `initBlocks()`, change `colorPicker` to choose a random number between 0 and 4 (inclusive) using `rand()`.

TODO 2.2

- Also in `game.c`, uncomment `drawBlocks()`.

TODO 2.3

- Uncomment the `updateBlocks()` function. This takes in a pointer to a block. If the block is *inactive*, the function does nothing. If it is *active*, it performs the collision checks we wrote for you. Inside of the main collision check, set the block to inactive and not erased (active = 0, erased = 0), and increment `score` by 1.

TODO 2.4

- In `updateGame()`, call `updateBlock()` for each of your blocks.
    - Hint: the number of blocks is a macro named `BLOCKCOUNT` and the blocks are stored in an array named `blocks`.

TODO 2.5

- Complete `newBlock()`. This should iterate through the blocks to find the first inactive block in the pool and initialize it. When you are finished initializing, break out of the loop. Initialize the block like so:
    - `active`: 1
    - `erased`: 0

Compile and run. When you enter the game state, you should see layers of blocks of different colors. Each time you restart the game, you should see them being different colors (since we randomized it and seeded our random number generator). **Note: When you hit a block, sometimes it will seem to not disappear. This is because**

we are calling newBlock to immediately set a block to be active again which is sometimes the block we just destroyed. This is intended, don't worry!

TODO 3: DMA

For our game to be fast, we need to use DMA. The simplest way to do this is to write a function that sets up the registers of the specified channel.

TODO 3.0

- In `gba.c`, complete the `DMANow` function.
    - This function should set up all the registers of the given DMA channel and turn it on for us. This allows us to use DMA whenever we need it with a single line, without having to set all the registers line-by-line in every location we want to use it. There are additional comments in the `DMANow` function that will help you write it.

TODO 3.1

- Rewrite the `fillScreen` function to use DMA, using your new `DMANow` function.
- Make sure to use **DMA channel 3**. You may not use any loops.
    - ***Hint:*** If you are copying one thing (one single source) to every pixel in the `videoBuffer`, what do you need to tell the DMA control to do (or not do) to the source? Look at `gba.h` for helper macros!
    - ***Hint:*** Make sure the `src` and `dst` parameters are *addresses* to the locations you are copying from and to.

TODO 3.2

- Rewrite the `drawRect` function to use DMA, using your new `DMANow` function.
- Make sure to use **DMA channel 3**. You may use only **one** loop.
    - ***Hint:*** You must use one loop here because, unlike `fillScreen`, the area you are copying to (`dest`) is not contiguous. Each row of the rectangle is, though. Use DMA to draw one row at a time.
    - ***Hint:*** Make sure the `src` and `dst` parameters are *addresses* to the locations you are copying from and to.

Build and run. The game should look the same, but a lot snappier during transitions. If not, fix it before moving forward.

TODO 4: Analog/DMG Sound

Finally, let's add a fun little beep whenever there is a collision between a block and the ball to make our game slightly more interesting.

TODO 4.0

- Inside `initialize()` in `main.c`, enable sounds in the sound on/off register (`REG_SOUNDCNT_X`).
    - ***Hint:*** We might have made a useful macro that sets bit 7 for you! Check out `sound.h` to see all the macros for sound.

TODO 4.1

- In `updateBlocks()` in `game.c`, use sound channel 2 to play a sound with the following characteristics:
    - The volume of the envelope (`DMG_ENV_VOL`) should be 4.
    - The step time (`DMG_STEP_TIME`) should be 2/64.
        - ***Hint:*** Whatever value you put in those bits in memory is automatically divided by 64!
    - The note played should be G6.
        - ***Hint:*** Use the `NOTES` enum!
    - Set the reset bit (`SND_RESET`) so that the sound is reset to initial volume settings.
- Remember that there are two main registers we need to set when we use channel 2, `REG_SND2CNT` and `REG_SND2FREQ`. Look over the slides (or TONC, or GBATek) to see which register controls what for the above settings!

Build and run. **Compare your lab to the provided Example.gba**. If everything works identically, submit your lab.

Submission Instructions

Ensure that **cleaning** and building/running your project still gives the expected results. **Please reference previous assignments for instructions on how to perform a "clean" command.**

Zip up your entire project folder, including all source files, the Makefile, and everything produced during compilation **(including the .gba file)**. Submit this on Canvas. Name your submission Lab05_LastnameFirstname, for example:

"Lab05_BandersonBarson.zip"

It is your responsibility to ensure that all the appropriate files have been submitted, and that your submitted zip can be opened and everything cleans, builds, and runs as expected.