LAB 04:

States, Structs, Pooling, and Text

Provided Files

- `main.c`
- `gba.c`
- `gba.h`
- `game.c`
- `game.h`
- `font.c`
- `font.h`
- `text.c`
- `text.h`

Files to Edit/Add

- `main.c`
- `text.c`
- `game.c`
- `game.h`
- `Makefile`
- `.vscode`
    - `tasks.json`

Instructions

In this lab, you will be completing several different TODOs, which will piece by piece, make a simple Space-Invaders-like 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 just a *blank black screen*. Complete the TODOs in order, paying close attention to the instructions.

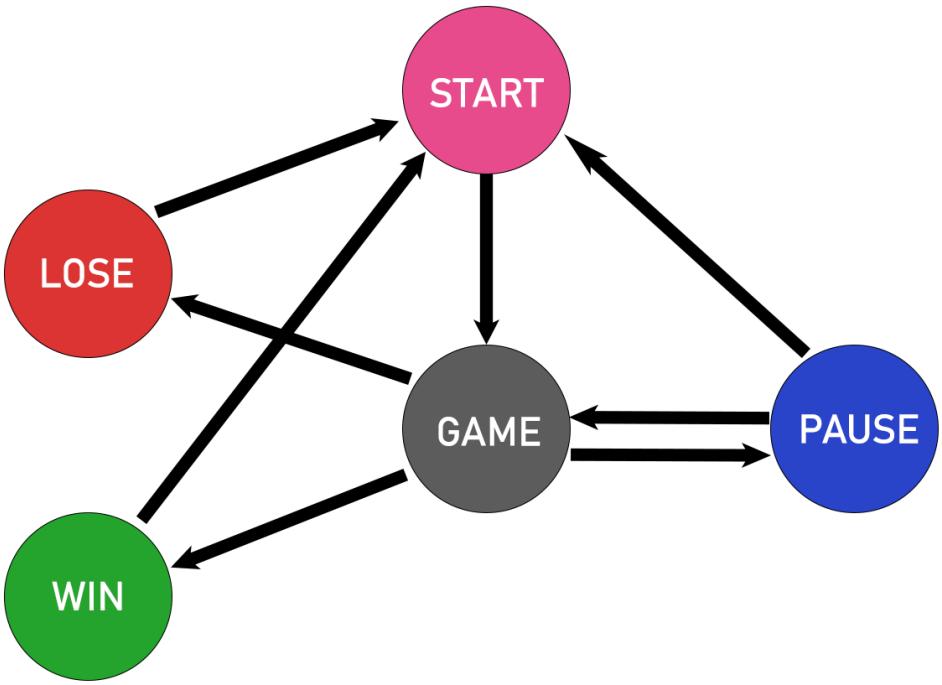
TODO 1: State Machine

For our game to be user-friendly, we need to implement a state machine first. The state

machine for this assignment will have the following states:

- ➔ START
  - ◆ Consists of a blank PINK screen (given macro in gba.h)
  - ◆ The seed for the random number generator increases each frame spent in this state
  - ◆ Pressing SELECT takes you to the GAME state, seeds the random number generator, and calls `initGame()`
- ➔ GAME
  - ◆ Consists of a blank BLACK background with a game running on top of it (we will add in the game in later TODOs)
  - ◆ Calls `updateGame()`, `waitForVBlank()`, and `drawGame()` in that order
  - ◆ Pressing SELECT takes you to the PAUSE state
  - ◆ Pressing B takes you to the WIN state (for now)
  - ◆ Pressing A takes you to the LOSE state
- ➔ PAUSE
  - ◆ Consists of a blank BLUE screen (given macro in gba.h)
  - ◆ Pressing SELECT takes you to the GAME state without reinitializing the game
  - ◆ Pressing START takes you back to the START state
- ➔ WIN
  - ◆ Consists of a blank GREEN screen
  - ◆ Pressing SELECT takes you back to the START state
- ➔ LOSE
  - ◆ Consists of a blank RED screen
  - ◆ Pressing SELECT takes you back to the START state

To help you visualize it, the state machine looks like this:



TODO 1.0
- In the `main` while loop, create the switch-case for the state machine.

TODO 1.1
- Below the already-written `initialize` function, make the functions for all of your states with their respective goTo functions.
    - ***Hint:*** Make sure that the states work like the ones described above, and that you don't do things every frame that only need to be done once (like fill the screen).

TODO 1.2
- Now that you have made these functions, put the prototypes for them at the top of `main.c`.

TODO 1.3
- Call your `goToStart` function in the `initialize` function so that when the game starts, the `START` state is first. If you haven't already done so, call your state functions in the state machine switch-case.

Compile and run. You should be able to travel through all the states by pressing the respective buttons. If not, fix this before going further.

TODO 2: Text
Now that we have the states made, it's time to add text to each of the screens to label them.

TODO 2.0
- In `text.c`, complete the `drawChar` and `drawString` functions.
- `drawString` should make use of `drawChar`, calling it inside of a loop

TODO 2.1
- In `main.c`, update your *state transition* functions to label each state somewhere on the screen (make sure it doesn't clip past boundaries, and that the color is visible!)
    - For `START`, draw "press SELECT to play" in a color of your choice.
    - For `PAUSE`, draw "paused" in a color of your choice.
    - For `WIN`, draw "you win!" in a color of your choice.
    - For `LOSE`, draw "you lose!" in a color of your choice.
- You aren't required to center the text on the screen, but if you're like me you will really want to center it horizontally! This is a formula to do this:

```
col = (240 - (numberOfCharacters * 6)) / 2)
```

Compile and run. You should be able to see your labels in each state now.

TODO 3: Cannon
Now that we have our state machine set up, we can begin coding the game. We will do

this in `game.c` and `game.h`, and call those functions in `main.c` (like we already are with `initGame()`, `updateGame()`, and `drawGame()`). Let's start by getting the cannon working.

TODO 3.0

- In `game.h`, create the `struct` for the player, typedef-ed `CANNON`. `CANNON` should have a `row`, `col`, `oldRow`, `oldCol`, `hSpeed`, `height`, `width`, `color`, and a `bulletTimer` (we will see the use for this later). All of the `CANNON` members can be of type `int` except `color`, which should be a `u16`.

UNCOMMENT 3.0

- Below this, uncomment the line that declares the player.

UNCOMMENT 3.1

- In `game.c`, uncomment the line that declares the player here.

UNCOMMENT 3.2

- Below the `drawBar` function, uncomment `initCannon()`, `updateCannon()`, and `drawCannon()`.

UNCOMMENT 3.3

- Now that we have these cannon functions, uncomment the prototypes in `game.h`.

UNCOMMENT 3.4

- Uncomment `initCannon()` in `initGame()`.

UNCOMMENT 3.5

- Uncomment `updateCannon()` in `updateGame()`.

UNCOMMENT 3.6

- Uncomment `drawCannon()` and `drawBar()` in `drawGame()`.

Compile and run. When you enter the game state, you should see the cannon under the blue/green bar, and move the cannon left and right. If not, fix this before going further.

TODO 4: Bullets

Now that the player is moving, let's give it a pool of bullets to shoot.

UNCOMMENT 4.0

- In `game.c`, uncomment the line that declares the pool of bullets here.

UNCOMMENT 4.1

- Below the `drawCannon` function, uncomment `initBullet()`, `fireBullet()`, `updateBullet()`, and `drawBullet()`.

TODO 4.0

- Complete `initBullets()` by initializing all the bullets with the following values:
  - `height`: 2
  - `width`: 1
  - `row`: the player's row
  - `col`: the player's col
  - `oldRow`: the player's row
  - `oldCol`: the player's col
  - `vSpeed`: -4
    - vSpeed is the bullet's vertical velocity
  - `color`: `WHITE`
  - `active`: 0

TODO 4.1

- Complete the `updateBullet()` function. This takes in a pointer to a bullet. If the bullet is *inactive*, the function does nothing. If it is *active*, it moves the bullet up the screen. If the bullet goes off the top of the screen, make it inactive.
  - ***Hint:*** the bullet's vertical velocity is `vSpeed`!

UNCOMMENT 4.2

- Now that we have these bullet functions, uncomment the prototypes in `game.h`.

UNCOMMENT 4.3

- Uncomment `initBullets()` in `initGame()`.

TODO 4.2

- In `updateGame()`, call `updateBullet()` for each of your bullets.

TODO 4.3

- Complete `fireBullet()`. This should iterate through the bullets to find the first inactive bullet in the pool and initialize it. When you are finished initializing, break out of the loop. Initialize the bullet like so:
  - `row`: the player's row
  - `col`: player col + (player width / 2) + (bullet width / 2)
    - This is the center of the cannon's top
  - `active`: 1
  - `erased`: 0

UNCOMMENT 4.4

- In `updateCannon()`, uncomment `fireBullet()` so that they actually fire.

TODO 4.4

- Since pressing B fires a bullet, it should not also win the game every time you press it. So, back in `main.c`, comment out the fact that pressing B wins the game.

TODO 4.5

- In `drawGame()`, call `drawBullet()` for all of the bullets.

Compile and run. When you enter the game state, you should be able to fire bullets by pressing B. You should be able to see multiple at once if you fire quickly enough. If not, fix this before going further.

TODO 5: Balls

Now that the player is moving and shooting, let's give it something to shoot at.

TODO 5.0

- Most of the code to make the balls work has already been written for you, so in `updateGame()`, call `updateBall()` for each of your balls.

TODO 5.1

- In `drawGame()`, call `drawBall()` for all of the balls.

Compile and run. When you enter the game state, you should be able to see several balls bouncing around. When you pause and unpause, they shouldn't have moved. If not, fix this before going further. ***Every time you re-run the game, the starting positions of the balls should be different.*** The positions may change only slightly, or may look to be the same if you idle in the START state for the same duration of time each run. Ensure `srand()` is working correctly by entering the GAME state immediately upon starting your game (i.e. running your `Project.gba` file), and by entering the GAME state 30 seconds after starting your game. If the starting position of the balls differ, you're fine. If not, you aren't using `srand()` correctly.

TODO 6: Collision

Having all those balls bounce around isn't very fun yet. Make it so that shooting all of them allows you to win the game.

TODO 6.0

- In `updateBall()`, loop through all of the bullets. If an active bullet is colliding with this ball, make both the bullet and the ball inactive, then decrement `ballsLeft`.

TODO 6.1

- To allow the player to win the game, you need to go back into `main.c`, and, in the game state function, transition to the `WIN` state if `ballsLeft` is 0.

Compile and run. When you enter the game state, you should be able to shoot the balls. If a bullet hits a ball, they both disappear. If you shoot all the balls, the win state should begin. If not, fix this.

At this point, you should be able to travel to all of the states, play the game, and win the game. If so, then you are done.

Tips

- Review lecture and recitation material for how to implement a state machine, pooling, `drawChar`, and `drawString`.
- Follow each TODO in order, and **only move forward if everything is correct!**

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 Lab04_LastnameFirstname, for example:

> "Lab04_TroopaKoopa.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.