CS2261 Media Device Architecture

Notes

* Continuing with DMA
* What if we want to set the background?
  + We have our GBA screen, and we would just draw a rectangle the size of the screen, using drawRect and DMA
    - Much faster with DMA than without DMA
  + Alternatively, we could just have one call that starts at videoBuffer and moves down through memory for all the pixels on screen (since video memory is contiguous and the pixels all map to memory starting at videoBuffer)
    - Can avoid the outer loop
    - Code:

void fillScreen(u16 color) {

DMA[3].src = &color;

DMA[3].dst = videoBuffer;

DMA[3].cnt = (240\*160)

}

* DEMO
  + Multiple balls demo:
  + In three different runs of the same code that uses rand() to pick a color for the balls, the ball colors are the same
    - Why?
    - The random function we are using for colors is not truly random, it’s pseudo-random
    - Seed. The rand() function picks a random numbers based off a random number generator that works off a seed through various complicated calculations
      * srand(unsigned int seed)
        + this function gives a special seed to the random number generator
    - the seed defaults to 1 when not manually set, so just using rand() without affecting the seed will produce the same “random” number each time
    - Generally, the seed is changed by setting it equal to the current time
    - But GBA doesn’t have a connection to the time
    - Need a different answer
      * Instead, we can use the player to change a variable we can use for the seed
      * For example, we have a splash screen for the player. Each instance the player spends on the start screen will affect the seed variable, and we use that time spent as the seed
        + Only very rarely will the player spend the exact same time on the splash screen
  + We’ll be using Usenti to convert images into a GBA friendly array
    - Produces a constant array of unsigned shorts
    - Each pixel in the image has a certain color, and that color is converted into a short and placed in the array. Does this for each pixel in the image and you then have one giant array of short colors
  + Usenti Demo: converting a GT Helmet into a bitmapped image
  + Demo with balls and the GT helmet as a splash screen
* Recursive functions:
  + Fundamentally, a function that calls another function
    - Hard to keep track of things
  + Need to keep track of parameter values, return values and local variables and return address
    - Have to make sure things don’t get mixed up
* Factorial example (fact(3)):
  + Certain values have to be stored when recursing through a function, such as
    - Parameter values, return values, local variables, and return address
    - All code that has an address.
    - The way all of these values are kept track of is through an activation/call stack
  + Activation/call stack
    - A stack is a finite Last In First Out (LIFO) data structure
    - Push and pop functions
    - Each function call has a record in the stack, and it gets pushed onto the stack
    - Visualization of a stack for the factorial recursive call

|  |  |
| --- | --- |
| Fac(0) | - base case, return 1 |
| Fac(1) | N = 1  RetVal = 1 \* 1  RetAdd = 1 |
| Fac(2) | N = 2  RetVal = ?  RetAdd = 2 |
| Fac(3) | N = 3  RetVal = ?  RetAdd = 3 |

* + - Finite structure with a limited amount of recursive calls
      * Going over results in a stack overflow
* Volatile constant pointer: From the USENTI output, we know the array generated from the image will be constant
  + Put in the constant area of memory
  + Volatile handles the cases like fillScreen