Homework 9: Software Design Considerations

Team Code Name: \_\_Augmented Reality Simulator\_\_\_\_\_\_\_\_\_\_\_\_\_ Group No. \_5\_

Team Member Completing This Homework: \_\_\_\_\_\_Thor Smith\_\_\_\_\_\_\_\_\_\_\_\_\_

E-mail Address of Team Member: \_\_\_\_smith777\_\_\_ @ purdue.edu

Evaluation:

|  |  |  |  |
| --- | --- | --- | --- |
| SEC | DESCRIPTION | MAX | SCORE |
| 1.0 | Introduction | 5 |  |
| 2.0 | Software Design Considerations | 30 |  |
| 3.0 | Software Design Narrative | 30 |  |
| 4.0 | Summary | 5 |  |
| 5.0 | List of References | 10 |  |
| App A | Flowchart/Pseudo-code for Main Program | 10 |  |
| App B | Hierarchical Block Diagram of Code Organization | 10 |  |
|  | TOTAL | 100 |  |

Comments:

1. Introduction

We propose an Augmented Reality Simulator that allows at least on user to play an electronic game in a mobile, outdoor environment. The device is split into two primary physical parts, but there are three parts for software. The central control unit, the headset, and the GPU will all have software components. The headset routes information to the GPU and the GPU renders images and objects based on that information. The central control unit tells the headset where images and objects are located and provides a user interface. This report will focus on the software on the headset, but will address the software on all components.

The software design must address how data will flow between all hardware and software components. The headset must accurately transform raw inputs from the sensors into values that can be used meaningfully by other software components. The GPS accuracy poses a challenge for reliably detecting simulation collisions. It also poses a problem for accurately drawing the location of objects for the user. The GPU must accurately draw objects and have a shared method of referring to objects specified by the central control unit. The central control unit must provide a user interface and manage the states of a wireless headset.

1. Software Design Considerations

A state machine approach was chosen for the headset versus a polled loop style or purely interrupt driven program. This choice was made because the headset must cycle through states based on wireless data received from the central control unit. The diagram in Figure 1 below shows that states that the headset must go through to successfully demonstrate the required functionality. The headset must begin by broadcasting its presence so that it may be selected for inclusion in the simulation by the central control unit. It must then load static data from the central control unit to the GPU for later rendering. Once this has been completed it may finally enter the simulation phase. Flowcharts describing the operation of the central control unit and the headset main module can be found in Appendix A.

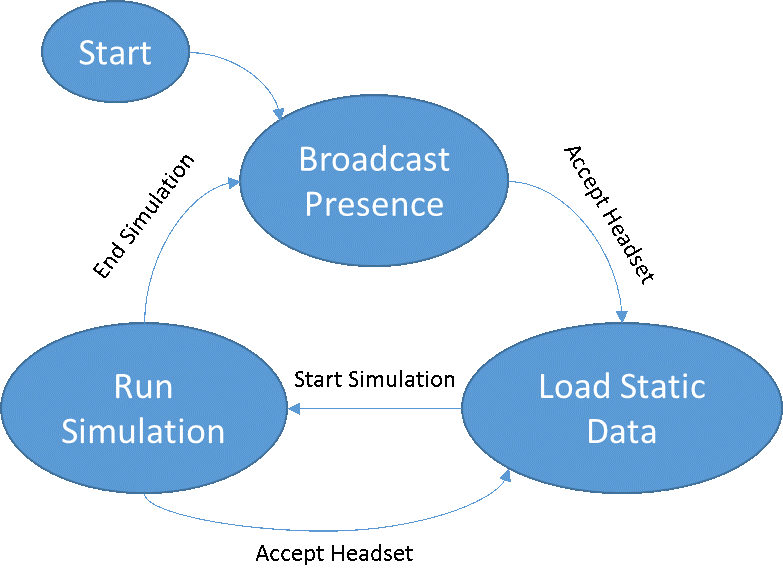


Figure 1: State machine for headset operation.

The communication between the central control unit and the XBee [1] is through UART. In the same manner, the communication between the STM32F4 and the XBee is through UART. The communication between the GPU (raspberry pi) and the STM32F4 is through SPI. The GPU will act as the master and the STM32F4 will act as the slave. The IMU communicates to the STM32F4 through I2C clocked at 400 KHZ (fast mode) [6] and the Venus638FLPx GPS Receiver [2] communicates through UART. The baud rate of the XBee modules is set for 56700 on their corresponding UART ports. The baud rate of the GPS is configured for 38400.

The memory configuration of the headset can be seen in the Figure 2. This image was taken from the programming manual for the STM32F4 [5]. SRAM is mapped from 0x20000000 to 0x20020000 for a total of 128K. The stack will be stored at location 0x20020000 because it is a descending stack. Static and global variables will be stored at location 0x20000000. There are currently no plans to use a heap. All of the memory necessary will be determined at compile time for the headset.

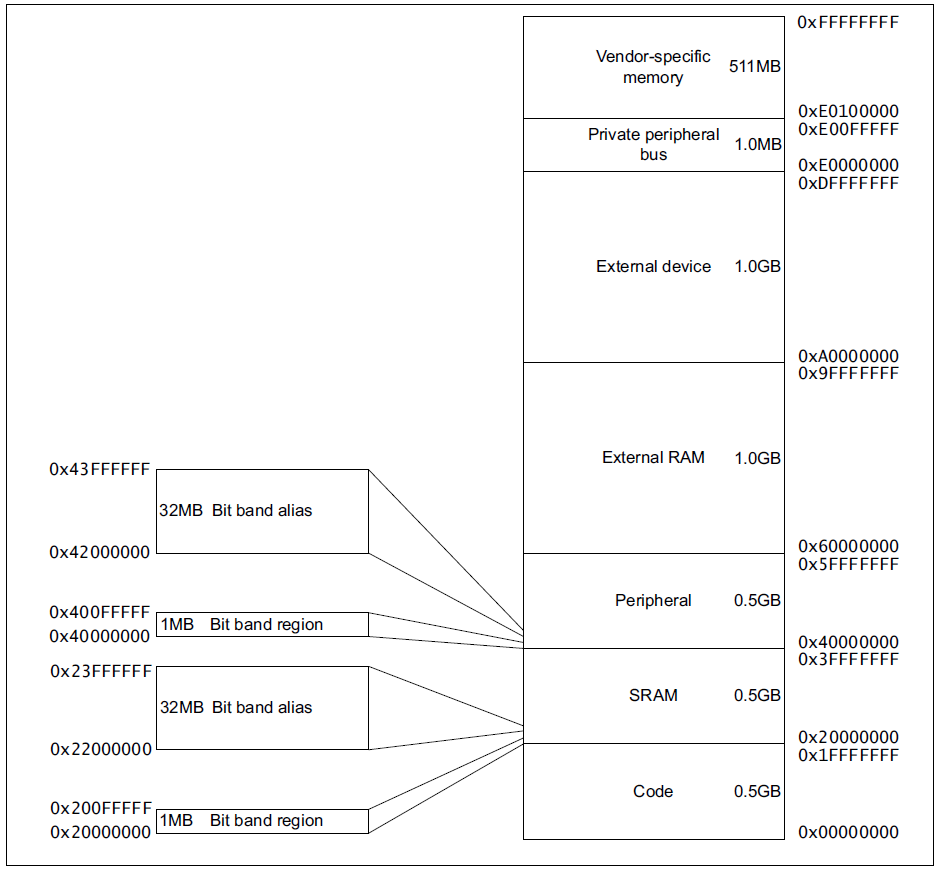


Figure 2: Memory map from STM32F4 Manual [5]

1. Software Design Narrative

Figure 5 in Appendix B shows the hierarchical arrangement of the various code modules included in our design. The code modules in our design are: the central control unit user interface, the central control unit simulation, the headset main module, the headset GPS IRQ, the headset IMU IRQ, the headset battery IRQ, the headset XBee IRQ, and the headset GPU.

The central control unit user interface is written in Python Tkinter [7]. Tkinter was chosen because a complex user interface is not needed, and because the author is familiar with the software. The user interface launches on startup of the central control unit and begins looking for available headsets to join in simulations. The user is able to choose available headsets and add them. The simulation allows a user to select a simulation to run. After selecting a simulation, the rules and hazards are explained to the user. The gui will then launch the simulation and wait for it to end.

The central control unit simulation will first load image and object data to all headsets through wireless communication. It will then proceed to send updates about the position of the image and object data to the user. Updates about the headsets position will be periodically processed and collision detection will be performed to determine simulation events to be triggered. Due to limited accuracy of the GPS, virtual objects will be made large (approximately 2 meters) for the purposes of detecting collisions. Progress has been made in defining the configuration settings for the [XBee](https://github.com/snowpuppy/augreality/tree/master/ccu/test). Most of the work for this module remains.

The headset IMU, XBee, and GPS IRQs pull data from the respective units and pool that data into a fifo buffer. The headset main module periodically sends data collected for the GPS location, IMU orientation, and battery status to the GPU and to the central control unit. When packets are received, the headset main module transitions between the following states: Broadcast Presence, Load Static Data, and Run Simulation. The Broadcast Presence state causes the headset to repeatedly send out a message to the central control unit advertising the headset's availability. The Load Static Data state will transfer data from the CCU to the GPU for storage. The Run Simulation state will wait for object update messages from the central control unit and send the updates to the GPU. The IMU, GPS, and battery IRQs have been completed. Data has been successfully extracted from all of these sensors through the STM32F4. The code can be found [here](https://github.com/snowpuppy/augreality/tree/master/headset/test). STMicroelectronics libraries were used to extract the data [4].

The GPU software was written for the Raspberry Pi using OpenGL ES 2.0 [8]. There are several major software components running on the Pi: a SPI thread to communicate with the microcontroller, a file loader to dynamically load a simulation from a file, input file handling for debug purposes, rendering loop which actually draws the virtual world. For SPI, an LGPL licensed library called Wiring Pi [9] is used to initialize the Pi’s SPI hardware and read and write bytes. All transfers are initiated by the Pi sending a byte indicating the type of data it wants from the microcontroller: i.e. a file, sensor data, or XBee packet. After this the Pi waits for a response, which starts with a byte indicating the number of bytes in the packet. This is checked against the expected size of the data type requested to detect data corruption. Then the specified number of bytes is read into a buffer and stored into the relevant variables with typecasting. The file loader loads a level file in a simple custom text format specifying all objects in the scene with an ID number, initial location, orientation, visibility flag, and model filename. The game objects are stored into an array of a custom object type holding all of this information. To reduce memory usage and speed up load times, certain common models (like the wall) are only loaded once with each object referencing the same model data in memory. Input handling uses the GPL licensed SDL library [10] to check key presses once every frame and execute the appropriate action, like exiting the application or moving the 3D camera. The rendering loop uses a GPL licensed OpenGL ES 2.0 wrapper library called piNGL [11] and is based on a sample application included with the library. Initially an attempt to write a custom OpenGL engine was made and abandoned due to very high difficulty. A 3D camera is created based on GPS and orientation data read in from the SPI thread. Then the array of game objects is looped through, and if the visibility flag is set it is drawn in the appropriate location. After this, a separate camera is created for 2D drawing. Then 2D icons for the battery and RSSI indicators are drawn to the screen as textured squares.

1. Summary

Our proposed project, an augmented reality simulator, aims to display an image to the user that adapts to their position and head orientation. To accomplish these goals, GPS and IMU data is harvested and routed to the GPU and the central control unit. Updated object position information is routed from the central control unit to the headset to allow objects to change their position as well. A user interface wraps all of the functionality.

5.0 List of References

1. Digi International, “XBee-Pro 900HP”, Xbee Pro 900HP datasheet, Aug. 2013 <http://www.digi.com/pdf/ds_xbeepro900hp.pdf>
2. SkyTraq Technology, “Venus638FLPx GPS Receiver”, Venus638FLPx datasheet, Feb. 2010 [Revised Jan. 2011] <http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/GPS/Venus/638/doc/Venus638FLPx_DS_v07.pdf>
3. SGS-Thomson Microelectronics, “L3GD20 and LSM303DLHC 9-axis module for a standard DIL socket”, STEVAL-MKI108V2 datasheet, Nov. 2011 <http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00041389.pdf>
4. STMicroelectronics, “STM32F4 DSP and standard peripherals library, including 82 examples for 26 different peripherals and template project for 5 different IDEs”, STSW-STM32065 peripherals library, Oct. 2013 [http://www.st.com/web/en/catalog/tools/PF257901#](http://www.st.com/web/en/catalog/tools/PF257901)
5. STMicroelectronics, “STM32F3xxx and STM32F4xxx Cortex-M4 programming manual”, PM0214 Programming manual, Sept. 2012 <http://www.st.com/st-web-ui/static/active/en/resource/technical/document/programming_manual/DM00046982.pdf>
6. STMicroelectronics, “STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx advanced ARM-based 32-bit MCUs”, RM0090 Reference manual, Sept. 2012 <http://www.st.com/st-web-ui/static/active/en/resource/technical/document/reference_manual/DM00031020.pdf>
7. Python Software Foundation, “Tkinter – Python interface to Tcl/Tk”, Python Documentation, Oct. 2013 <http://docs.python.org/2/library/tkinter.html>
8. OpenGL, “OpenGL Software Development Kit”, OpenGL Documentation, 2012 <http://www.opengl.org/sdk/docs/>
9. Gordons Projects, “WiringPi”, 2013 <https://projects.drogon.net/raspberry-pi/wiringpi/>
10. SDL, “About SDL”, 2013 <http://www.libsdl.org/>
11. Jon Macey Computer Animation Pages, “piNGL”, 2013 <http://nccastaff.bmth.ac.uk/jmacey/GraphicsLib/piNGL/index.html>

Appendix A: Flowchart/Pseudo-code for Main Program

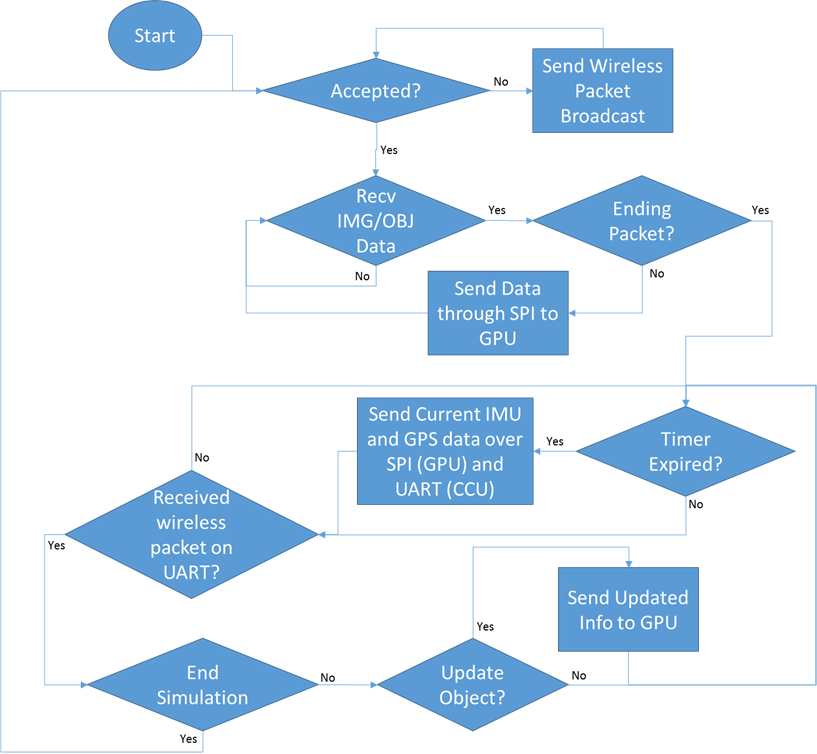


Figure : Flowchart showing headset main module flow.

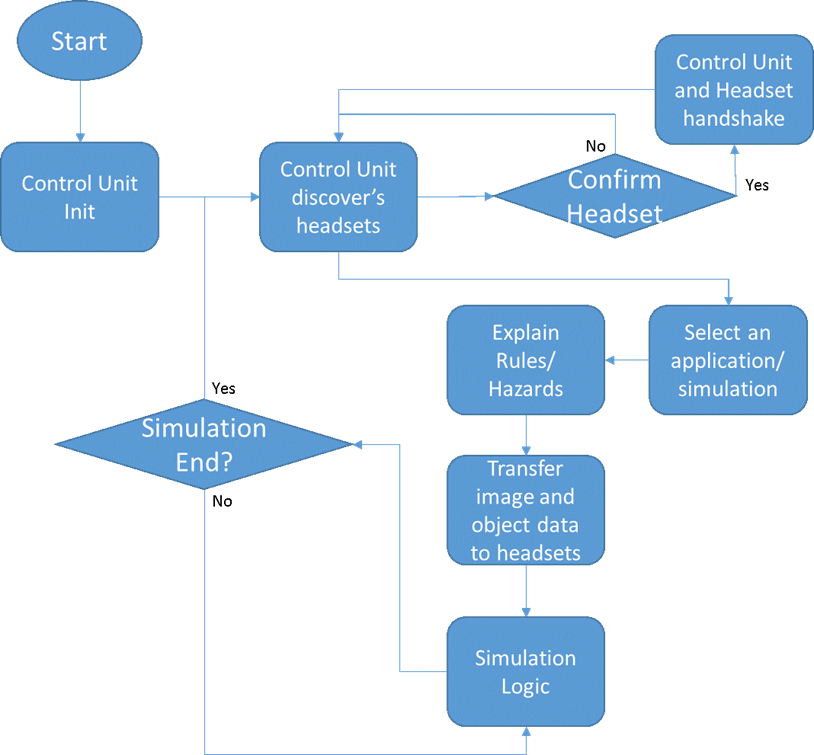


Figure : Flowchart showing CCU user interface flow.

Appendix B: Hierarchical Block Diagram of Code Organization

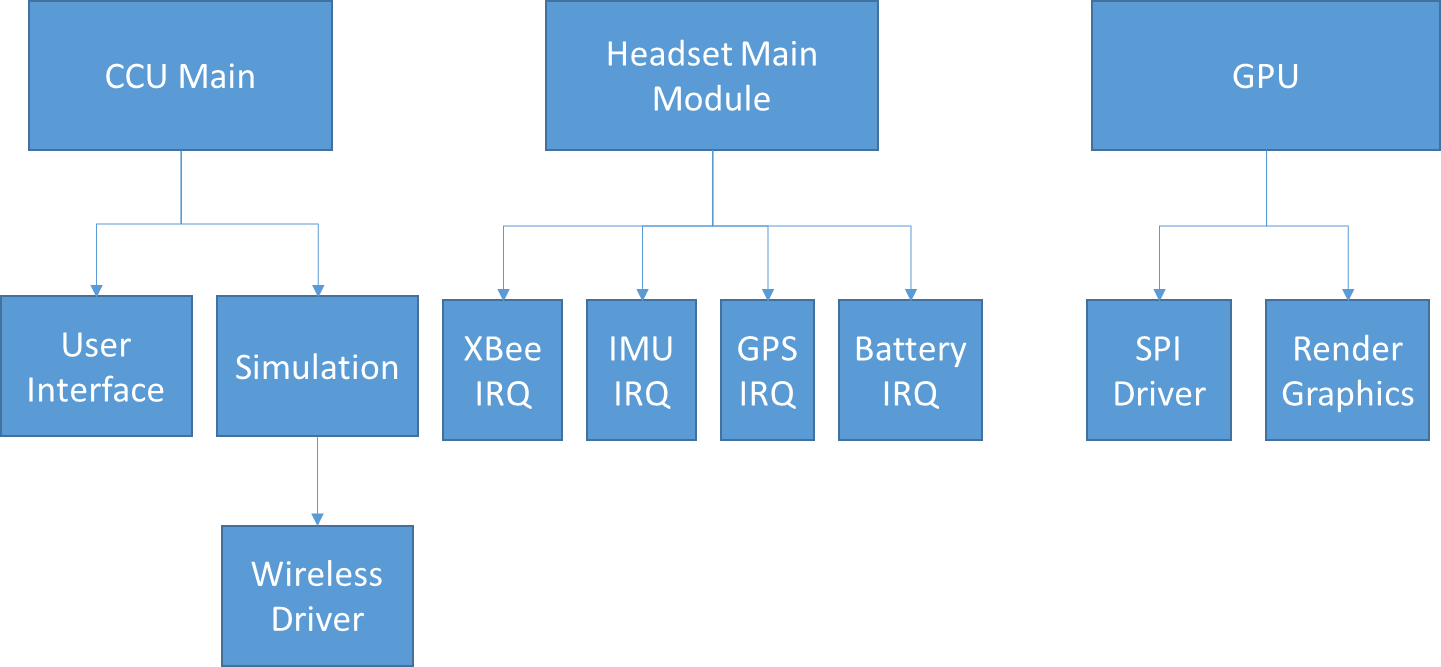


Figure : Hierarchical diagram showing the relationships between software modules.