**CT5PROGC Reflective Report**

**States**

|  |  |  |  |
| --- | --- | --- | --- |
| **Game state** | **Screenshot** | **Description** | **Links** |
| Title screen |  | Title screen for the game. Displays title of game, displays the player sprites, offers some options for gameplay | Game screen |
| Game screen | F:\CT5PROGC\REPORT\COURSEWORK_04 (2).png | Main game screen. Displays updating timer and score, player sprites and enemy sprites will update on screen depending on player input for gameplay | Game Over  Pause menu |
| Pause menu |  | Pause screen. Displays options to continue game or return to the main menu. | Title screen  Game screen |
| Game Over |  | Game over screen. | Title screen |

**Gameflow**

Title screen

Game screen

Pause menu

Game Over

**Input**

Referring to the input header file included with the devkitpro creation, the initial build of the game used controls that solely used the “keysHeld” function, with commands that only needed to be carried out once per press being controlled by a list of “Boolean” variables that would be set true and false based on the current status of keysHeld and checked against to stop the command being carried out more than once if the GBA key was held. Later the “boolean” states were moved into an enumerated array before being removed to use the keysDown function in the input header, which will only run the command once on key press as standard. This simplified the input commands and lowered the amount of variables in memory.

**States/Logic**

The game is split into four game states as shown above, using previous knowledge of game creation all of the states initially were being run through the same script. All of the game’s inputs would be checked for once, then run commands based on what the current game state was, and certain sections of code would only run when checking for each game state, using multiple if statements for each state.

The code was made clearer by creating separate input checks for each game state, as well as consolidating all running code into a single if statement for each state, only the sprites were being redrawn, the VBlankWait (Required for each frame to be drawn sequentially) and the mmFrame (updates each sound frame) is run regardless of the game state. The game structure was later consolidated further by moving build functions to separate files, moving defines and structures to header files and moving as much update code into separate functions as possible, making the main code easier to read and understand, allowing for less issues at runtime.

Object variables were moved into a data structure which contained values for all variables that any of the game objects required, making the code easier to understand. The structure was then split into 3 structures that only contained the values each type of object required to lessen the amount of unused variables in memory.

Collision code was added using a “bounding box” method – All objects check if they are within the same space as each other based on their x and y values, in addition to the specified width and height of the object. This a simple method requiring minimum mathematical procedures, which should be more effective on the GBA’s processor.

The enemies will spawn at a random rate, with either two or four maximum on screen at a time. Rather than removing the OBJ of the enemy entirely when it is “removed” from the game the GBA’s off-screen space is utilised – moving the OBJ to a location where it isn’t visible at runtime or affect gameplay. The two kinds of enemy will move across the screen (One ‘jumping’ whenever it reaches the ground), either being removed from the screen by reaching the other side, causing collision with the “shield” object, or the game being reset. Enemies could have been better dealt with by working with an array of enemy objects, allowing for more than 4 set instances of enemies and making code clearer (Running code for each instance in the array via loop).

The artefact seemed rather difficult with the possibility of all four enemies on screen at once, so functionality was set for the player to be able to choose an ‘Easy’ or ‘Hard’ mode on the title screen, if ‘Easy’ is selected only half of the enemies will be able to be initialized on screen.

The shield in the game originally merely followed the player, but conversion to the X movement in .asm made the shield “spin” around the player as it only stopped to change direction around the boundaries of following. This was kept as a feature and extra controls were implemented to give more control of the shield – Holding the B button will make it follow as originally intended, and the shoulder buttons force it to stay in their respective direction.

**Graphics**

The in-game font was created in a bitmap file using Photoshop, converted using a specialised exporter to an external data file and loaded in via a loop. While other graphics are created manually, creating the font in a separate image file was a far simpler process as it allowed for drawing exactly as displayed.

All text is displayed on one tile layer – the front most tile layer - using C. This allows for easier updating of the text as only one layer needs to be cleared per frame via a loop.

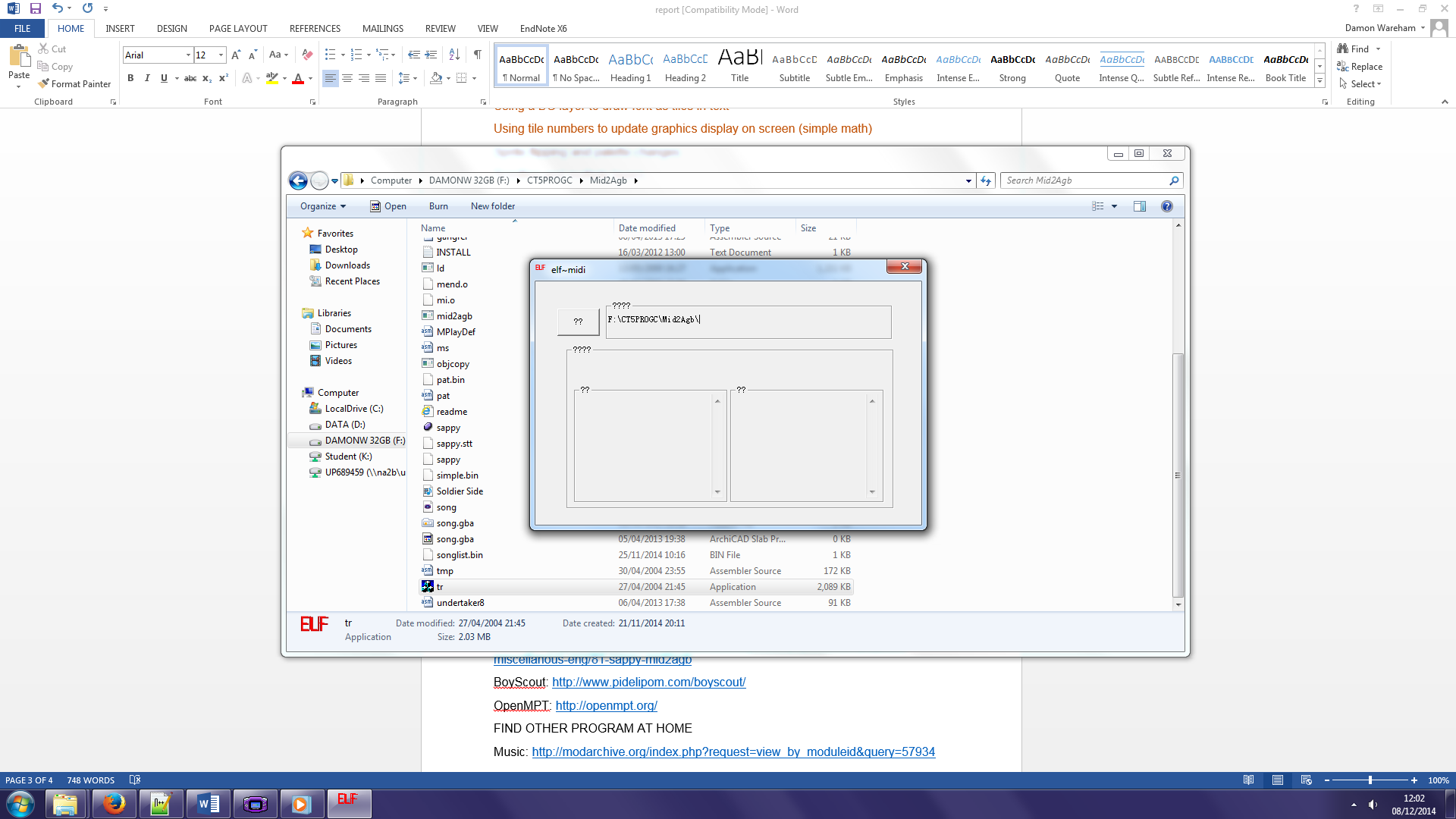
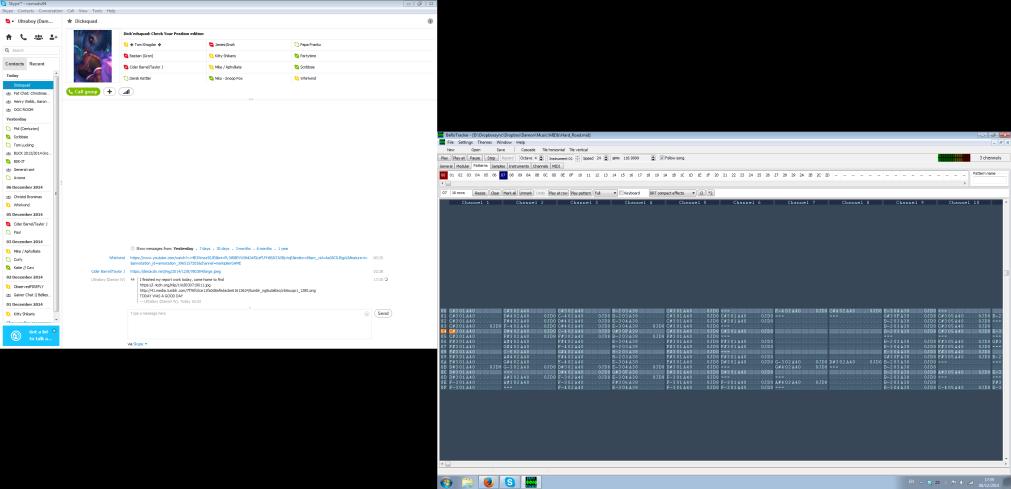
The HUD – Timer and Score, is displayed by reading each variable for the timer and score and displaying the tile based on the variable – The number font is stored sequentially in the tile data. To stop the timer or score going over or under these tile numbers if the timer reaches 0 or the score reaches 999 they are hard-coded to display ‘”WIN” or “LOSE” respectively in this case. The same effect is used for updating the tiles used to display animation in the game’s OBJs, only requiring one line of code for each character update.

All four instances of the enemy objects are being created using the same sprite set – Using horizontal flipping and palette swaps for each instance, lowering the amount of sprite data required. Text is also redrawn depending on the game state by changing palettes. All sprites objects are drawn with two tiles each, 1x2 for the player and shield, 2x1 for the enemies.

Later on in development the game’s layers were better utilised – tiles being drawn in the background layers with scrolling being introduced on each respective layer – The “platform” will scroll along with the player if they move to the far right or left of the screen to give the illusion of an infinite map. The tiles are drawn via loops – Alternating between drawing and not drawing depending on the location. Scrolling was done by finding the right memory addresses inside of the I/O viewer tool included in the VisualBoyAdvance emulator.

**Sound**

The initial sound implementation in the artefact was the example file provided after modifying the makefile to implement sound files.

Originally the game’s music would be sourced from conversion of MIDI files, but finding software which could perform conversion was difficult. Mid2aGB was a program that was advertised as being able to do so but seemed to not run as intended.

BeRoTracker was found, a program that converted .MIDI files into .MOD files (Which the maxmod libraries are able to support), but as the GBA is only able to support 4 channels of sound for playback the files found were unusable.

BoyScout was later found and tried, as a program which specifically worked as a tracker for GBA hardware, but maxmod was unable to support the file format exported.

A music piece was found pre-made online that only used 4 channels, which has been implemented in the final piece of work. OpenMPT was then used to create original sound for the pause menu and jumping sound effects, which supported 4-channel .MOD sound creation.

In the main code the sound effect is implemented using mmEffect, as using mmStart while a music sample is playing replaces the currently playing sample, rather than playing both. mmJingle would also be a maxmod command which could be used to similar effect. Music can be turned off by player choice with functionality on the title screen.

At first, if sound was stopped or paused, the previous frame of sound would play constantly, making the game seem like it had ‘frozen’. This was fixed by replacing any pause or stop functions by playing an (almost) silent sound file, giving the same effect without the ‘glitch’ effect.

**Low-level code**

The timer, score and movement were re-written from C into ARM assembly in order to demonstrate functionality, using the C code as reference on how to create the code, using code branching. Loops were attempted based on C code but were not working by the final artefact.

Initially all of the assembly functions were writing using the LDR function, which loads a 32-bit value into the register for processing, but the function would only work with integer variables. Finding a reference guide online however the LDRB, which is able to load in an 8-bit value, filling the remainder of the register with 0 (null) values. This allows it to work with 8-bit char variables, allowing for four times less memory usage for each value which doesn’t require more than 8 bits.

**References**

Maxmod: <http://devkitpro.org/maxmod.org/ref/>

Mid2agb: <http://www.hackromtools.altervista.org/index.php/en/main-page/20-miscellanous-eng/81-sappy-mid2agb>

BoyScout: <http://www.pidelipom.com/boyscout/>

OpenMPT: <http://openmpt.org/>

BeRoTracker Classic: <http://www.hitsquad.com/smm/programs/BeRoTracker/>

Music used: <http://modarchive.org/index.php?request=view_by_moduleid&query=57934>

Assembly reference: <http://problemkaputt.de/gbatek.htm>