Code Explanation: Main.cpp

The main.cpp script serves to outline the application’s core functionality. The header files for the component classes required to carry out the program’s tasks must first be referred to. These include GameConstants and GameResources, which define some of the more customisable aspects of the game such as the font and window icon. Operations concerning the physical appearance of the game are handled with cD3DXManager, cD3DXSpriteMgr, cD3DXTexture, and cSprite. More specific subclasses of cSprite include cStephen and cEnemy, which handle the player character and the obstacle characters, respectively. Lastly, cXAudio and cD3DXFont are incorporated in order to deal with the functions related to sound and score display. The standard namespace will be used for this program.

The next stage in the project’s implementation is to declare all of the global variables. To start, a series of Boolean variables is defined. Game tells the program whether or not the player is in one of the levels, while gameWater, gameLightning, etc. tell the program which level the player is currently in. A Boolean variable for music determines whether or not the background music is playing. Next, a global handle to hold the application when it runs is created, followed by a global variable to hold the window. The DirectX manager and sprite renderer must be explicitly referred to in order to make use of their functionality, otherwise we would not be able to see any of the game’s assets. Before the player character, Stephen, is first initialised, a floating point value of 0 is given for his rotation, ensuring that he will be upright upon spawning. His starting co-ordinates are contained within a vector, as well as the scaling of his sprite. The enemies’ position is also contained in a vector. The size of the window is in a rect container, so that the enemies can be created around the central point of the window without the need for precise calculation on the developer’s part. The enemies themselves are contained in a vector, with iterators George and Index set up for later use. Another Boolean is created to hold information on whether or not Stephen has destroyed an enemy, and his total kill count is held in an integer value for the program’s use, and a character value for display. Stephen himself is created as a list with the iterator Pete, rather than a vector as the changes that affect the enemies are more dynamic than any changes Stephen may possibly go through. Audio objects for the background music and kill sound effect are created as well, with a surface and a background image for the main menu completing the roster of global variables.

The window procedure method checks for available input messages from the queue, in order to determine if a user has pressed a key that corresponds to action in the game. If the player presses the left arrow key, the program checks if Stephen’s position on the x axis does not exceed the left-hand side of the window. If there is sufficient space, Stephen’s sprite will be moved ten units to the left and the loop will return a value of 0 to confirm that there are no errors within the loop. The same process is similar in the opposite direction for moving to the right, although the right-hand bound is increased by 90 so that the front of Stephen’s sprite is stopped against the side of the window, rather than his back. The game offers seven different backgrounds, so if the player presses a number from 1 to 7, the program will set every one of the level-related Booleans to false with the exception of the Boolean that corresponds to the chosen background. For example, if 1 is pressed, gameWater will be true along with the game Boolean that states that gameplay is underway to begin with. Every other scene Boolean (gameFire, gameLightning, etc), will be set to false. Pressing 8 and 9 do similar things, only instead of continuing gameplay, they shut off all of the Booleans except those for the game over and start screens, respectively. Next, if the space bar is pressed and the game Boolean is true, the iterator for the enemy vector will be sent to the beginning of the vector. As long as there is still at least one enemy left for the iterator George to point to, the program will check whether or not that enemy is intersecting with Stephen’s position when the space bar is pressed. If there is an intersection detected, the enemy at George’s position will be eliminated and the total number of killed enemies will be increased by one. This will be reflected in the score counter. If, however, there is no intersection, George will move on to the next enemy in the vector. Regardless of whether or not there was a hit, a sound clip will be played, representing Stephen Stills’ battle cry. Finally, if the window was closed or otherwise destroyed, the application will quit running. No matter what happens, the message will be returned to the default window procedure after it has been executed.

Following this, the initWindow method creates the display window of the game after taking in all the parameters that define the window’s appearance. This is accomplished by using the WNDCLASSEX structure. The parameters that are assigned include the size of the structure, the style of the window class, the callback procedure and any extra bytes needed to allocate for the class and instance. The window icon, background colour and cursor are assigned to either the ones specified in GameResources or the Windows default. Next, the window itself is created, with the title bar text being set to the name of the game: Stephen Stills’ Fabulous Adventure. The size and position of the window are then assigned, and the window is now ready to be displayed on the screen.

WinMain is the main entry point of the program, so this is the method that is used first during run-time. The window is initialised along with the D3DX sprite manager to ensure that they will perform their tasks as required. The timer is then defined to handle the game’s frame rate, along with all related variables such as the time at specific points, frame rate and cumulative time. The EnemyPos variable is created to handle the position of each enemy. A random number generator is seeded in case the number of spawned enemies should be random, but it was decided during development to have a fixed number of enemies spawn, which is set to 5. A for loop is created next, so that for each enemy, their position will be assigned slightly off the screen’s centre. The first three enemies will be on a higher row of the screen than subsequent enemies. Each enemy is drawn with the same sprite: Ridley from the Metroid series. Each enemy’s translation will be given five units on the x axis and the y axis’ direction should be set randomly on each playthrough. Another random number generator is seeded to accomplish this. The surface texture is defined next, along with the backbuffer required to render it. The fonts to display both the kill count and any GUI messages are defined next, both using the same font, again from the Metroid series. These messages’ positions are defined in rects, while Stephen and the enemies’ positions are defined in vectors. Stephen himself is then drawn, using his sprite from the Scott Pilgrim game. His transformation matrix and initial translation are defined along with him. The menu’s background image is called into play along with the main theme of the game: Gabbag by Mark Knight.

A large while loop contains most of the gameplay elements. While the quit message has not been sent and the window is still open, the message queue is checked. If the start menu is open and the music is off – conditions that are both true at the beginning of runtime – the background is drawn and text with instructions on how to play the game is rendered on to the screen. But if the end screen is open, the game is turned off, meaning Stephen and the enemies are removed from that particular screen. More enemies are created in the same way as before, with the loop function, but these enemies are not drawn until game is true again; when the player goes back to one of the levels. This means that players can respawn as many enemies if they want to aid replayability. Text is rendered to thank the player for playing and to encourage them to go back into the level to challenge more enemies. Their score will persistently carry over during this transition. When the gameplay itself is true, Stephen will be drawn on the screen along with the enemies and a counter which keeps track of how many kills Stephen gets. The frame rate is maintained throughout the rest of the process. The background is then drawn to match whichever level’s Boolean is true, for example if gameWater is true then it will be the water level’s background image that is drawn as the background. For each enemy in the enemy vector, the iterator George will get the enemy’s current position and check for collisions with either the ‘walls’ of the window or other enemies. If an enemy collides with the sides of the window, it will be sent in the opposite direction. The second enemy iterator, index, will assist the iterator George in making sure that two colliding enemies will change direction on their collisions. The DirectX manager will then begin a render, using the iterator George for the enemies and Pete for Stephen to update every sprite’s position and ensure that they are drawn in the correct place. An iterator Mike was also initially in play to draw the explosions, although they were later removed from play. Text is then displayed on the screen to match whatever the current background is. Each one of the backgrounds will display its name and number, along with the persistent kill count and a reminder to the player to press the 8 key when they are finished and want to go to the end screen. Once this is finished, the render is ended, all loops are closed, and the Direct X managers are left to clean up the remains.