Code explanation

**SpaceAdventuresSDL.cpp**

The program starts by executing the main method located in the SpaceAdventuresSDL class. This class contains two variables: pgmWNDMgr and theGame. Its worth noting that these variables are pointers to a static instance of their class. The game variable’s type is cGame and the pgmWNDMgr variable’s type is cSDL2WNDManager. These classes contain a method called getInstance which we call to get an instance of the class. The variable plsInstance holds the instance of the class. The method getInstance uses the variable to check if plsInstance is empty, in which case the instance is created and returned.

The main method with return type int, initializes the window by calling the method initWnd. This happens inside an if statement which checks if the window has initiated. If the method returns true, then the window has been initialized. Else if the method returns false the main method returns -1 and so the program is terminated. Briefly the method has three parameters: WINDOW\_TITLE, WINDOW\_WIDTH, WINDOW\_HEIGHT. WINDOW\_TITLE is of type string while WINDOW\_WIDTH and WINDOW\_HEIGHT are of type int. The method Initializes SDL's Video subsystem, creates the main window and the renderer. If it fails to create these conponenets or initialize SDL’s subsystems the method returns false.

The main method then calls three functions from the object theGame. These functions are initialized, run, and clean up. The initialize function has two parameters: a SDL window and a SDL renderer. This method is responsible for setting up the assets the game uses. The run function is responsible for running the game after initialization. Meaning that after everything has been initialized, the elements of the game must be updated and rendered on the window. The final function called is clean-up in which the memory used in the program is cleared. The main method then returns 0 and so the program is terminated.

**cGame class**

cGame\* cGame::getInstance()

Returns a pointer to an instance of cGame. If the variable plsInstance is empty the method creates a new object, otherwise the method returns the instance. The reason static is used, is because the program uses one instance of cGame, since the purpose is to build one game. The same logic applies to other classes (manager classes).

**void cGame::initialise(SDL\_Window\* theSDLWND, SDL\_Renderer\* theRenderer)**

The purpose of the class is to initialize elements the game will later update or render. First, the method uses the SDL API to get the width and height of the window. In addition to this the method sets the m\_lastTime variable which we will use to calculate delta time.

The function also sets the renderer colour which in this case is black, with RGBA(Red, Green, Blue, Alpha): 0,0,0,255.

SDL\_RenderPresent updates the renderer. Since there is nothing displayed up to this point the window is black. At this point the managers can be initialized. The function setRenderer is called to pass the renderer to the theTextureMgr object. The game will be using audio and fonts, so it is necessary to initialize SDL’s ttf and mixer. After that the program checks and initializes the xbox controller. If the xbox controller is connected, then it is stored and ready to use. Otherwise the controller cannot be used.

theTexture manager provides functionality that the program will be using. But first the textures must be stored to this object. For this to be done, two parallel arrays of type LPCSTR(one containing the names and one containing the relative location of each texture) are used. A for loop goes through every element of the parallel array and stores each element to the texture manager.

textureName = { "startSceneBackground", "endSceneBackground", "theBackground", "theRocket", "theAlienSpaceShip", "bullet", "AlienShipBullet" };

texturesToUse = { "Images\\startSceneBackground.png", "Images\\endSceneBackground.png", "Images\\theBackground.png", "Images\\rocketSprite.png", "Images\\alienSpaceShip.png", "Images\\Bullet.png", "Images\\AlienShipLazer.png" };

//startSceneBackground, "endSceneBackground"

//Add textures to texture manager

for (int tCount = 0; tCount < textureName.size(); tCount++)

{

theTextureMgr->addTexture(textureName[tCount], texturesToUse[tCount]);

}

The base of the algorithm is used for adding elements to every manager. In the sound manager requires three parallel arrays. The third array stores the sound type (music used for background music, SFX used for sound effects). In the button manager the third array stores the position of the button on the screen. The text displayed in the game is also a texture and thus the texture manager is used. Two parallel arrays (one with the name of the texture and the other representing the actual text) are used to add textures to the texture manager. These textures represent the UI elements of the game. These UI elements also include the player’s lives and the player’s score. This means that we need to convert an integer into a LPCSTR:

string scoreString = to\_string(scoreValue);

LPCSTR scoreLPCSTR = scoreString.c\_str();

The code from the project shows that the program first turns the value into a string and then, into a LPCSTR.

On initialization the game also calls the Play function from the sound manager which plays the background music in every scene:

theSoundMgr->getSnd("theme")->play(-1);

The function sets the initial/default background the game will be using. As we can see in the code the function sets the sprites position, texture, and dimensions.

//Set the background

spriteBkgd.setSpritePos({ 0, 0 });

spriteBkgd.setTexture(theTextureMgr->getTexture("theBackground"));

spriteBkgd.setSpriteDimensions(theTextureMgr->getTexture("theBackground")->getTWidth(), theTextureMgr->getTexture("theBackground")->getTHeight());

spriteBkgd.setSpriteDimensions(theTextureMgr->getTexture("theRocket")->getTWidth(), theTextureMgr->getTexture("theRocket")->getTHeight());

The method sets the initial/default state of the game. The variable gameState is an enum. The program uses this variable to call the right functions on each scene.

theGameState = INSTRUCTIONS;

The method also initializes the rocket by setting its position, texture, and scale. I used render width/2 theRocket.getSpriteCentre().x because I wanted the rocket to spawn in the middle of the scene. The top right corner of the sprite represents the position of the sprite. So, by adding the centre of the sprite the centre of the sprite is at the centre of the screen(in the x-Axis).

theRocket.setSpritePos({ renderWidth / 2 + theRocket.getSpriteCentre().x , 600 });

theRocket.setTexture(theTextureMgr->getTexture("theRocket"));

theRocket.setSpriteScale({ -0.6f, -0.6f });

theRocket.scaleSprite();

The scene manager “takes” all the initialized elements and uses them to update and render every scene.

sceneMng = new cSceneMgr(theRenderer, theButtonMgr, theTextureMgr, theFontMgr, theSoundMgr, spriteBkgd, renderWidth, renderHeight);

sceneMng->setGameState(theGameState);

sceneMng->setRocket(theRocket);

sceneMng->setGameTextList(gameTextList);

sceneMng->setLoop(true);

sceneMng->setFiles("Data/rocket\_data.dat", "Data/alien\_ships\_data.dat", "Data/bullets.dat", "Data/UI\_Data.dat");

In this code fragment the constructor is called with parameters: theRenderer, theButtonMgr, theTextureMgr, theFontMgr, theSoundMgr, spriteBkgd, renderWidth and renderHeight. The constructor has these parameters because these variables are the most important, for the game to work. For example, the game needs to have a renderer before having a rocket. As we can also see the class has setters and getters which we can use to set the initialized elements.

Files are a very important feature of the game. The program checks if the files have data used to generate the PLAYING scene. If there are no data, the scene is generated with its default state (3 lives, 0 score and 27 alien ships). if the files are not empty then the playing scene is generated with the data provided from the files.

if (sceneMng->filesEmpty())

{

sceneMng->spawnAlienShips(26, 60); //26 60

sceneMng->setScoreValue(0);

sceneMng->setLivesValue(3);

}

else

{

sceneMng->loadPlayingSceneData();

}

**void cGame::run(SDL\_Window\* theSDLWND, SDL\_Renderer\* theRenderer)**

This method is responsible for running the game. Inside this method is a while loop that stops the main method from terminating the game. Essentially if the loop’s variable is equal to true, the function will call the update, render, and get input method which updates the loop’s state. The loop state is updated in the getInput method because this method handles events. So, if the player for example closes the window, the event is handled, and the loops value is set to false.

**void cGame::render(SDL\_Renderer\* theRenderer)**

This method is responsible for rendering the game. SDL\_RenderClear(theRenderer); clears the renderer. sceneMng->renderScene(); calls a function that renders the game and SDL\_RenderPresent(theRenderer) updates the renderer so that changes can be shown on the screen.

**void cGame::update(double deltaTime)**

This method calls a function of the scene manager that updates the state of every game object.

**bool cGame::getInput(bool theLoop)**

This method calls a function from the scene manager which handles events and returns a Boolean. This Boolean is then returned by this method and determines if the game is terminated or not.

**double cGame::getElapsedSeconds()**

returns the delta time. Delta time represents the time passed from the last call to the current.

**void cGame::cleanUp(SDL\_Window\* theSDLWND)**

As the game initializes the libraries used, it should also close them before the program is terminated.

**cAlienShip class**

This class is used to create every alien ship object of the game. Therefore, the class defines the possible state and behaviour of every alien ship. In addition to this the class inherits from the cSprite class. (An alien ship is a sprite)

**void cAlienShip::update(float deltaTime)**

The update method updates the state of the alien ship object. The float rateOfFire stores the time remaining for the alien ship to shoot. After an alien ship shoots the variable is generated randomly. The SDL\_Rect currentSpritePos stores the sprites position. The method getSpritePos() returns the position of the sprite. This method is inherited from the cSprite class. The rest of the code on this method moves the object left or right. The variable’s initial position represents the initial position of the object. If the object is more than 35 pixels after the initial position, then the object moves left. Accordingly, if the object is more than 35 pixels before the initial position, the object moves right. The algorithm uses the bool right to define the movement of the object. If the bool is true, the alien ship moves right. Else if the bools value is false, the object moves left. When the object moves to the right the x coordinate of the current sprite position rectangle is equal to its last value multiplied by the object’s velocity and finally multiplied by deltaTime. This is because the movement should be related to the time passed. If deltaTime was not considered, then the movement would rely on function call. The method ends by positioning the object to its new location: setSpritePos({ currentSpritePos.x , currentSpritePos.y }); and by setting the bounding rectangles used in collisions this->setBoundingRect(this->getSpritePos());

**void cAlienShip::initialize()**

This method sets the initial/default state of the object. The velocity of the object is equal to 3 and the bool move right is true.

**void cAlienShip::setRateOfFire()**

The method sets the rate of fire by generating a random number. The random number’s value is between 2 and 20.

The other methods are setters and getters used to set the state of the object and provide access to other objects.

**cSceneMgr Class**

**cSceneMgr \* cSceneMgr::getInstance()**

returns a static instance of the object.

**cSceneMgr::cSceneMgr(SDL\_Renderer \* theRenderer, cButtonMgr \* theButtonMgr, cTextureMgr \* theTextureMgr, cFontMgr \* theFontMgr, cSoundMgr \* theSoundMgr, cSprite spriteBkgd, int renderWidth, int renderHeight)**

This method runs when the instance of the object is made. It sets the variables required for a scene to function. These variables include a pointer to the renderer, the texture manager, the font manager, the sound manager, and other variables like the background sprite, the render width and height. The method also sets the default scale of the alien ship sprite which will be used to generate a vector of cAlienShips.

**bool cSceneMgr::filesEmpty()**

opens files, checks if files are empty, closes files and returns true if the files are empty and false otherwise.

**void cSceneMgr::renderTexture(cTexture \* tempTextTexture, SDL\_Rect pos, FPoint scale)**

This method renders a texture using the renderTexture function. This function’s parameters are: the renderer, a cTexture, an SDL\_Rect representing the position and an FPoint representing the scale. The renderer has been set on the contractor, so the render Texture method’s parameters are: the texture to use, the position and the scale.

**void cSceneMgr::renderButton(int posX, int posY ,LPCSTR buttonName)**

The method’s purpose is to render buttons. It has 3 parameters two integers representing the position of the button on the window and the name of the button to render. The buttons have been added in the initialization method of the game object, so by using the scene manager, the buttons can be rendered.

**void cSceneMgr::renderBackground(LPCSTR textueName, cSprite backgroundSprite)**

The method renders the background sprite. It sets the texture of the sprite according to the scene that the game is currently running and renders the background itself.

**void cSceneMgr::renderBullets(vector<cBullet\*> theBullets)**

A for loop is used to renders every element of the cBullets list. The render method is provided by the cSprite class.

**void cSceneMgr::renderAlienSpaceShips(vector<cAlienShip>** **theAlienShips)**

A for loop is used to renders every element of the theAlienShips list. The render method is provided by the cSprite class.

**void cSceneMgr::renderUI(LPCSTR textureName, vector<LPCSTR> gameTextList, bool updateBool, int UIValue)**

The method uses the index used to identify if the lives or the score have been updated on the gameTextList, which holds the text displayed on the screen. The if statement checks that a collision has been made. The bool: updateBool is true when a collision occurs. The score and lives are not updated on method call as this is more efficient. When the updated elements have been rendered the updataBool variable is set to false. When the bool is true, the UI element given by the texture name is deleted. Proceeding the UI value is converted to an int. if the texture name is equal to livesValue then the index becomes 3, representing the 4th element (Lives)of the gameTextList. Otherwise the index is equal to 1(Score). After this process the gameTextList is updated and the texture is added to the texture manager.

**void cSceneMgr::renderInstructions()**

This method renders the instructions scene (opening/1st scene) calling functions in the class. Each function represents a different element (text, button, background).

**void cSceneMgr::renderEnd()**

The method renders the End scene which is the scene presented after the player wins or loses. The scene is being rendered with function calls provided by the class. Each function represents a different element (text, button, background).

**void cSceneMgr::renderPlaying()**

Method for rendering the playing scene. Everything said about the two previous methods is applied here. The only difference is that the method checks if updateLives or updateScore are true. If the variables are true, the renderUI function is called and the bool is being set to false.

**void cSceneMgr::renderScene()**

This method renders the scene using the gameState enum. According to the value of this variable the method calls different functions to render each scene.

**void cSceneMgr::updateGameState(LPCSTR buttonName, gameState newGameState, SDL\_Point theAreaClicked)**

Updates the game state when a button is pressed. If a button has not been pressed, then the game state remains the same.

**void cSceneMgr::updateGameState(gameState gameState)**

The overload of the previous method analysed. The game state is set to be equal to the parameter of the method.

**void cSceneMgr::updateBulletsVisibility(float deltaTime)**

The method iterates through every element of theBullets vector. The method then checks if the bullet is in the window. If it is not, then the method deactivates(removes) the bullet to free unused memory. Then it checks if each bullet is active. If the bullet is active, then it is updated, and the iterator is increased by one. Otherwise the bullet is erased (removed from the vector). It is worth noting that the gameDataBullets string adds to itself the position of every bullet. This variable is used to add data to the file storing the position of each bullet.

**void cSceneMgr::updateAlienShipsVisibility(float deltaTime)**

Everything mentioned in the last method is applies here the only difference is that the iterator is using the cAlienShips vector and that we check if the alien ships have been destroyed. If they have the game is over (the player wins) and the end scene is loaded.

**void cSceneMgr::updateAlienShipsFireRate(float deltaTime)**

The method goes through every alien ship using an iterator and determines if their rate of fire variable is less than 0. If it is, the alien ships are ready to shoot and so a bullet is spawned. Using the sound manager, a sound effect is also played to give feedback to the player. In addition to this the rate of fire must be reset so that the alien ship can shoot again.

**void cSceneMgr::updateCollisions(float deltaTime)**

Checks if a collision occurs. A for loop uses an iterator and examines every bullet. An if statement checks if a bullet collides with the rocket, if the bullet is generated by an alien ship and if the player has more than 0 lives. If the statement returns true then, lives are decreased by one, the Boolean updateLives is set to true, so that the UI is updated, and the bullet is deactivated, so that it can be removed from the vector. An if statement checks the player’s lives. If the player has more than 1 live, then the explosion sound is played. Otherwise the sound PlayerLost is played and the gameState is being set to END. This indicates that the player lost. At the same scope of the for loop, a second for loop, checks collisions between each alien ships. The for loop uses an iterator. Inside the scope of the second for loop, an if statement checks if a bullet collided with an alien ship and if the bullet is originated by the theRocket object. If the statement returns true, then the score is increased by 300, the updateScore Boolean is being set to true, the explosion sound is played and the two objects (the bullet and the alien ship) are deactivated.

**void cSceneMgr::updateRocket(float deltaTime)**

calls the objects update function to update the rocket

**void cSceneMgr::autoSave(float deltaTime)**

The method writes the data to files using the strings which hold the UI and position of every game object. The files are then opened and the data(strings) are “written” to them. The files are then closed.

**void cSceneMgr::spawnAlienShips(int remainingShips, float height)**

A recursive method used to spawn the alien ships. The int spawnedShips keeps track of the alien ships added to the alienShips vector. A for loop repeats a process depending on the value of remaining ships. This variable represents the alien ships that are yet to be added to the vector. An if statement checks if the position of the alien ship to be added to the vector does not exceed the boundaries of the screen. If it does not, then the sprite position, texture, scale, and rate of fire functions are called. The alien ship is also added to the vector and the variable spawned ships is increased by one. Else if this condition is not true, the function is called with the parameters for the remaining ships being the subtraction of the remaining ships and spawned ships. The height is increased by 100 so that the next column can be filled.

**void cSceneMgr::spawnRocket()**

This method spawns the rocket. Using the functions provided by the cSpriteclass which the rocket inherits, the method sets the position, texture, and scale.

**void cSceneMgr::spawnRocket(string rocketData)**

Uses the objects functions to spawn a rocket. The method sets the position, scale, and texture of the cSprite theRocket

**void cSceneMgr::updateXY(string alienShipsData)**

This method updates the two vectors holding the coordinates stored in the files. 4 variables are set: tempX and tempY store the current x and Y coordinate. While the bool updateX and updateY represent which coordinate the method should store in, tempX or tempY. A for loop reads every character of the string and adds it to the tempX until it finds a comma (‘,’). When this happens, the characters are stored to tempY(the bool’s value is reversed: from true to false and from false to true ). The next time a comma is found the vectors store the tempY and temX. These variables are then cleared, and the process is repeated until every character has been used.

**void cSceneMgr::spawnAlienShips(vector<int> xAxis, vector<int> yAxis)**

Uses the parameters to spawn every alien ship (add alien ships to vector). A for loop iterates a number of functions until all coordinates have been used. The functions are: setting the position, texture, scale sprite, set rate of fire, set initial position, and initialize the alienShip. The alien ship is then added to the vector. The process iterates for the next coordinates

**void cSceneMgr::spawnBullets(vector<int> xAxis, vector<int> yAxis)**

The two parameters are used to spawn the bullets (add bullets to the vector). A for spawns a bullet depending on the size of the xAxis and yAxis. The two parameters represent the set of positions that the bullets should have when spawned. The first bullet for example should be spawned horizontally on the value of the first element of the xAxis vector and vertically on the first element of the yAxis vector. The bullet is spawned with the same code as in the shoot mechanic of alien ships and the rocket.

**void cSceneMgr::generateUI(string uiData)**

This method generates the ui by reading the data from a file. Since the ui is generated bools updateScore and UpdateLives is set to true. The string tmpScore holds the temporary value of the score. As a for loop goes through every character of the string, the variable tmpScore adds every character, until the end of the string. The for loop begins from 1, representing the second character of the string. The first character is the lives value, so it can be comverted to an int and being set to the variable livesValue.x

**void cSceneMgr::updateInstructionsScene()**

updates the instruction scene by checking if the buttons are pressed which affect the game state. If the play button is pressed the game state changes to PLAYING. If the exit button is pressed the game state is set to QUIT. The method also resets the area clicked so that buttons in other scenes are not affected.

**void cSceneMgr::updateEndScene()**

updates the state of the game on button press. Resets the area clicked as in the other update functions, clears the alien space ships since the game has ended. Respawns the alien space ships if the game is replayed, clears the bullets (there should not be any bullets spawned as the game restarts), resets the (ui scoreValue = 0; livesValue = 3) and sets the update Booleans to true as the gameText array should be updated.

**void cSceneMgr::updatePlayingScene(float deltaTime)**

This method calls functions to update the PLAYING scene. In this scene the rocket is updated, the gameState on button press, the visibility of bullets, and alien space ships, the rate of fire of every alien ship which defines the time required for the enemy to shoot, the collisions and the autosave feature which adds data to files.

**void cSceneMgr::loadPlayingSceneData()**

This method reads data from the 4 files and generates the PLAYING scene. The method starts by opening the files in an if statement. If the files could not be open opened, a message in the console shows the SDL error and reports the issue. Otherwise the message on the consoles confirms that the files opened. The function spawnData uses the data from the file and spawns the rocket. The file containing the rocket’s position is then closed. The function updateYX is then called which reads the files and returns a vector with the positions stored in the file. Using that function the Bullets and alien ships can be spawned. The two vectors: x, y must be cleared after generating the objects by calling the spawn bullets and spawn alien ships functions. The files are then closed. The last operation is generating the UI. This is achieved by reading the numbers representing the score and lives from the file. As always, the file has to be closed.

**bool cSceneMgr::updateEvents()**

This method uses the API to handle different events. The purpose then is to respond to player’s input. This impute includes mouse events, key events and xbox 360 controller events.

**void cSceneMgr::updateScenes(float deltaTime)**

Uses gameState to call function to update each scene. It also deletes the strings used to store data to files so that they can be updated depending on the game state of the playing scene.

**void cSceneMgr::clearFiles()**

The method opens, empties, and closes the files.

**Setters**

Setters are used after the initialization process. After this process elements of the game are passed to the object cSceneMgr.

**cRocket Class**

**cRocket::cRocket() : cSprite()**

The constructor runs when an instance of the object is instantiated. The rate of fire is set in the scope of the method.

**void cRocket::update(double deltaTime)**

Updates the state of the object. Rate of fire is updated by subtracting delta time. This way the total time passed can be found. A SDL\_rect currentSpritePos stores the position of the sprite. This equation then moves the object: currentSpritePos.x += rocketVelocity.x \* 50 \* deltaTime;

delta Time adjust the movement so that it depends on time and not on function call. rocketVelocity is the velocity of the object.

It is worth noting that the velocity decreases over time which makes the movement of the rocket more realistic. The rocket does not stop immediately when one of the arrow keys is released. Finally, the position of the sprite is set on each function call.

**void cRocket::shoot(vector<cBullet\*> theBullets, cTextureMgr \* theTextureMgr, int numBullets)**

Represents the shoot mechanic of the game. The algorithm is similar to spawnBullets method in cGame class. The difference is that since the rocket shoots the rate of fire must be reset.

**void cRocket::moveLeft(float velocity), void cRocket::moveRight(float velocity, int renderWidth)**

These methods move the object left and right by setting the rockets velocity, if the rocket does not exceed the window boundaries.

**cSpaceAdventureGame.h**

It includes the header files the program uses. So instead of having to include the required classes every time, the header includes all the classes that I will be using as objects. For instance, the game class includes this header file which includes all the required classes. This way I can create objects without having to include them first in the class.

**GameConstants.h**

Defines the constants and includes the header files the game uses. The purpose is that since the constants and header files are stored I will not need to rewrite code on multiple classes. Instead, this header file allows me to include it and gain access to it. The constants it includes, concern the button types, game state, sound type and textType. The variables mentioned are therefore enumerators. The header also defines the title, height, and the width of the window.

**cTextureMgr Class**

The class responsible for creating textures for the game. I used this manager to add, delete and get textures as the state of the game changed. The class also uses the renderer which textures are using to be displayed. Another feature of the manager is that it provides error checking. At the loadTexture method: If the texture’s id is not equal to zero then the program: displays a message on the console (that the texture is loaded), determines the dimensions of the texture and returns true. If the condition is not met, the output in the console shows that the texture has not been loaded along with the SDL error occurred. The method then returns false. The method loadTexture has one overload. The parameter on the second method is a pointer to an SDL texture. The original method sets the texture using the relative location.

**cSoundMgr Class**

This manager class allows sounds to be added and removed. The class also contains a method which initializes the SDL mixer library and handles error detection. The class also uses its distractor function to delete the sounds and close the mixer.

**cSound class**

The class loads sound that the manager can then store, provide access, and remove. Sounds are loaded and played. A switch statement is used to curry out these functions by determining the sound type. The game uses two sound types: music (background music) and SFX (sound effects).

**cSDL2WNDManager class**

The class provides functions to initialize the window, create the renderer, provide error checking, and use getters to allow other classes to access its state. The method initWND takes three parameters: a string representing the window’s title, an int representing the window’s width and a second int representing the window’s height. The method initializes the SDL’s video library and using the parameters creates the SDL window. The method Get error is called when an error occurs (the library could not be initialized, or the window could not be created). The error is displayed in the command line.

**cSDL2CTRManager class**

The class initializes the SDL’s game controller library, adds a SDL\_controller and handles errors. If SDL\_Init(SDL\_INIT\_GAMECONTROLLER) is less than 0 then the library could not be initialized and the method initCTR returns false. A for loop with end condition, the number of the joysticks connected to the system goes through every joystick. An if statement checks if a joystick is a game controller. If it is then the theXBOX360Controller variable is defined(“opened”). A message also informs that the xbox360 controller has been created. When the controller is found the for loop breaks since the game is playable by one player. The last step of the method is to check if the game controller is empty. If it is, the message: Unable to create XBOX360Controller is displayed and the function CheckSDLError is called. The class also includes getters so that other classes can use the controller.

**cFontMgr, cFont**

The cFont creates/loads fonts while the cFontMgr adds, removes, returns fonts. Essentially it is the same as the sound class and sound manager class. The difference is that the classes handle fonts.

**cFileHandler**

The file handler makes the use of files possible by providing functions like openFile, closeFile, readDataFromFile and wrightDataToFile. Open file: opens the file while handling errors in case the file could not open. The reading data from file function is using an fstream (output) and a string (lineFromFile) to read data from a line in the file. Each line is read while the getLine(returns a line) function returns true. The wright data to file function has a string parameter which’s data is stored to the fstream theFile (which represents the file). The files should be closed after reading or righting data to them.

**cButton, cButtonMgr**

The button manager adds(stores), removes, returns buttons. It follows the same logic as the other managers. The buttons (which are sprites) are responsible for changing the game state of the game. The update method checks if the button is pressed and returns the new game state defined by the method’s parameters. If the button is not pressed, then the current game state is returned. The area clicked is used to determine if the button is pressed or not. The Boolean clicked determines if the button is being set when the update method runs. If the button is pressed, then the variable is equal to true, otherwise it is being set to false. getClicked and setClicked(bool state) provide access to other classes, so that they can set the Boolean or “get” it.

**cBkGround**

This class represents the background image of the game, which is a texture. Using the inherited functions of cSprite the background can be set (position, texture…) and rendered.