# Project 1 – Noughts and Crosses

Contents

[Project 1 – Noughts and Crosses 1](#_Toc448589451)

[Contents 1](#_Toc448589452)

[Project Design 3](#_Toc448589453)

[Introduction 3](#_Toc448589454)

[Program Reviews 3](#_Toc448589455)

[Program requirements 4](#_Toc448589456)

[Informal program specification 5](#_Toc448589457)

[Compulsory 5](#_Toc448589458)

[Optional 5](#_Toc448589459)

[Program comparison 6](#_Toc448589460)

[Problem Decomposition 6](#_Toc448589461)

[Example High Level Flowcharts 7](#_Toc448589462)

[Development 9](#_Toc448589463)

[Setting up the project 9](#_Toc448589464)

[Setting up the folder system 10](#_Toc448589465)

[Main cpp file 13](#_Toc448589466)

[Creating the class (Globalvars.h header file) 13](#_Toc448589467)

[Members 14](#_Toc448589468)

[Other cpp files 16](#_Toc448589469)

[Initialising 17](#_Toc448589470)

[Menu Systems 18](#_Toc448589471)

[Main Menu 18](#_Toc448589472)

[Option Menu 20](#_Toc448589473)

[Game Loop 21](#_Toc448589474)

[Key Inputs 23](#_Toc448589475)

[Extra Functions 23](#_Toc448589476)

[Initialising SDL and components 23](#_Toc448589477)

[Image loading (loadImg) 23](#_Toc448589478)

[Updating image on the window (windowUpdate) 24](#_Toc448589479)

[Checking if the grid position is empty (checkGridEmpty) 24](#_Toc448589480)

[Checking whose turn it is (whosTurn) 25](#_Toc448589481)

[Checking which player has won or if it’s a draw (checkWin) 25](#_Toc448589482)

[Resetting the game grid (resetGrid) 26](#_Toc448589483)

[Error printing to a file 27](#_Toc448589484)

[AI logic 27](#_Toc448589485)

[Attack Pattern 1 28](#_Toc448589486)

[Attack Pattern 2 30](#_Toc448589487)

[Attack Pattern 3 31](#_Toc448589488)

[31](#_Toc448589489)

[AI Attack and defend 31](#_Toc448589490)

[AI failsafe 32](#_Toc448589491)

[Meeting requirements 33](#_Toc448589492)

[Testing 36](#_Toc448589493)

[Main Menu testing 36](#_Toc448589494)

[Option Menu testing 36](#_Toc448589495)

[Play testing 37](#_Toc448589496)

[Play vs human 37](#_Toc448589497)

[Play vs AI 38](#_Toc448589498)

[Conclusions and Review 39](#_Toc448589499)

[Review 39](#_Toc448589500)

[Conclusion 39](#_Toc448589501)

[Bibliography 40](#_Toc448589502)

# Project Design

## Introduction

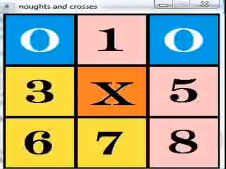
For the final project, the program that will be created is a Noughts and Crosses game. The game will be created using the C++ programming language and will use SDL libraries in order to show a GUI window instead of a console window. The game has to be constricted to a square grid, which noughts and crosses is. Instead of using a 1 dimensional array, a 2 dimensional array will be used instead.

To add more of a complexity to the project, a game AI will also be created for the game. The AI must at least know how to block the player, or win. This AI must also play fair i.e. will take the correct turn and not do double turns or overwrite any part of the grid which has already been played on. The AI will most likely be the weak point of the project i.e. will be the most likely part of the game which may cause the program to break, so great care must be taken whilst creating it.

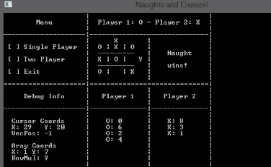
Another part to be added to the project is the ability for users to customise the playing grid i.e. change the graphics. Pre made graphics will be created and the users will have the ability to choose between these.

## Program Reviews

<https://www.youtube.com/watch?v=OgkEbdo21vA>

This is the first program to be reviewed. This program was created using C++ and SDL 1.2 (older version). It features a GUI and uses key inputs to place noughts or crosses on. The graphics could be improved, as all the graphics are of inconsistent colours which makes the program look too “messy”. The program also does not show which player’s turn it is. This can be improved by adding space at the bottom or at the sides which show which player’s turn it is. The program opens the game straight away and does not feature a menu. For the program to be created, there must be space for the menu and this can also be space which will show which player’s turn it is.

One final criticism for this program is that the “Win” screen looks really clumsy and that all its graphics look rushed.

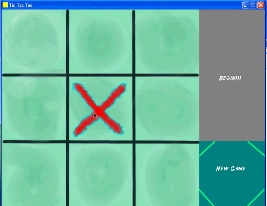
<https://www.youtube.com/watch?v=_s_4Gu7Fzec> (warning: loud music)

This program uses a C++ programming language however does not use SDL, and the program is console based. The design of the items on screen look very intimidating to people with low computer skills because of the high number of items on the screen which are not related to the Noughts and Crosses game. This program includes “Cursor Coords” which I assume could be the mouse cursor coordinates which means that the game may have a mouse input.

Unlike the previous program, this program does not have any graphics at all, instead it uses ASCII characters to create the game board and lines which separate different components of the game. This program also features a menu system and a game AI, as shown on the menu which contains:

* Single Player
* Two Player (<< the creator of this program has made a typo)
* Exit

<https://www.youtube.com/watch?v=rMJqMV5Lab0>

This program is built using the C++ programming language and uses SDL. The program has a good graphics layout and even has an initial menu system before going into the game and also provides an option to see the credits, which shows the people that were involved with creating this program.

This program however, does not have the ability to allow for more than one player, since the AI is always run. The program to be created should allow for there to both be 1 or 2 players.

One final criticism for this program is the fact that the creator added an unnecessary delay on the AI’s turn, this may annoy the user and slow down the game.

## Program requirements

The items that will be required for the Noughts and Crosses game to be created:

* Has graphics
* The user should be able to choose which style of graphics they want via the options menu
* User can choose between different items on the main menu i.e. play vs human, play vs AI or options menu
* Play vs AI will set the game to let the AI take the other turn instead of another human player
* A graphic which tells which player has won or if it’s a draw
* The correct graphic must be shown i.e. when its nought’s turn, it must show a nought on the chosen part of the grid
* Players cannot write over a square that is not empty
* Player can quit whenever they want
* Consistent style throughout the program
* Records and saves any errors to a file
* At least 3 different graphics styles for the user to choose from

Programming requirements:

* Using C++ programming language
* Built for Windows OS and is a windows application
* Uses SDL libraries
* Project settings for Visual Studio must be all set to make sure that the people marking the project won’t have to go through the struggle of setting the project up

Optional:

* Mouse input
* Game Sounds
* Sounds Option in the option menu

## Informal program specification

### Compulsory

The program to be created should be built specifically for Windows OS and must be using C++ programming language. In order for the program to show graphics on screen, an external library called SDL will need to be used, however, for this project, SDL2 will be used instead of SDL1.2 which is provided by the university. To ensure that the marking process of the project is easy for the markers, and to make sure the project works on their computer, the project folder will have to be set up so that all the image resources and the library and include files are included in the folder. The project (solution) file will also have to be set up so that the project will always refer to the correct files in the project folder and would not matter where it is on the project marker’s computer. This must be done via the use of $(SourceDir) in the project settings.

The program needs to have graphics and utilises these graphics for the gameplay, and in the menu system. There should at least be 3 different graphics for the user to choose from, this is in order to give a variety of choices for the user and to accommodate for the fact that they may be colour blind.

The menu system must have multiple items i.e. these can be what the user wishes to do on the program i.e. access the option menu or play the game with another player or versus an AI. A graphic must show this clearly, this is to ensure that the user knows what options they have and which input they need in order to initiate that option.

Play vs AI must initiate an AI so that the second player is the game AI instead of another player. This AI must also be smart enough to make sure that it can block the player’s moves so they don’t win and also not cheat by putting a cross on a part of the grid which is not empty. The AI must also not have a big delay which may annoy the player.

During the game multiple different graphics will be used, one to notify the player(s) whose turn it is. This is to make sure that the players know who has to do the next move. The correct graphic must also be applied to where the current player places their move i.e. if it’s nought’s turn, a nought must be placed on where they chose to do their move.

Once the game ends via a win or a draw, there must be a graphic which shows which player has won or when the game has drawn. This is so that the players know who has won or if the game has ended in a draw. There must a delay in this case, otherwise the players will not be fast enough to read and know who has won.

### Optional

Optional components to be added to the noughts and crosses game will be to allow the users to use mouse inputs. This is because with a GUI, users are most likely to expect a mouse input. Adding game sounds to the game may also make the project more complex, but since the project is complex enough, this has been decided to be an optional requirement. Adding sounds would be helpful to players to know that the other player has done a turn. The sound can also prompt users if anyone has won, using various sounds.

The last optional component to be added (if the previous are added) is to give the user(s) the ability to turn off the sounds via the option menu. This is because the players may not want to have the sounds on, especially if the sounds become too repetitive if the players do their turns quickly.

## Program comparison

The program to be created for this project will be different to the reviewed programs, because the program to be created gives the users the ability to customise the game according to the user’s preference. The users will have the ability to change the graphics of the game according to what they prefer. The program to be created also has an option menu, which none of the reviewed programs have. The option menu is what the user(s) can use to change the graphics of the game.

The program to be created will also use SDL2 instead of SDL1.2 which is what the reviewed programs used (except the last reviewed item – it was created in 1.2 and was updated by the creator to SDL 2). SDL2 has different processes to SDL1.2 which means that the processes in the reviewed program will be slightly different to the noughts and crosses game which will be created.

Unlike the reviewed programs, this project will have the ability to let users choose between playing against another player and playing against the game AI and also have graphics, making it more “user-friendly”. The second reviewed game does have the ability to play between another player or an AI however, does not seem user-friendly as it is console based and could be intimidating to computer illiterate people.

## Problem Decomposition

Since C++ programming language is used, a class can be used. The class will hold some variables as private members, so that they cannot be accessed without permission. The other members will be the different functions for the game. This means that only one class will be required, which is the game itself.

These will be the different game states i.e.:

* Main Menu
* Option Menu
* In game vs human
* In game vs AI

There are 4 game states i.e. when the user is in the main menu, option menu, or in game vs human or AI. These game states can be accessed via a loop, with the main menu loop being the loop where all the other loops can be accessed i.e. the main menu will always run and the menu loop ends when the player wants to quit the game. This means that the main menu is always in the background and everything in the game will revolve around the main menu.

There are also 4 game states, however only 3 loops are required for the game. The game loop will contain both of the loop for a game vs a human and a game vs AI. When the user sets the game to play vs an AI, a Boolean variable will be set to true. The variable will be checked if the variable is true, if so, the AI is run.

The other 2 loops will be the Main Menu Loop and the Option Menu Loop. The option menu loop and the game loops can be run from the main menu according to the key input that is done by the users. Reading the key input will be done via SDL.

Switch-case statements are also required in the project. These will be mostly used to handle the key inputs. The switch case will be used to check a key input event and will execute code according to what button the user presses – i.e. when the user presses the Esc key, the menu should be exited or the game should end.

Switch-case statements however cannot be used for anything, in which case, nested if statements will have to be used, this is for things such as checking variables with non-constant values i.e. the array values. The if statements will be used to check the value of the array to see whether or not they contain 0, 1 or 2. A value of 0 means that the part of the grid is empty, 1 or 2 means that the player 1 or 2 (respectively) has filled in the spot. If statements will have the most use in the project since many parts have to be compared according to the values the variables contain.

Other functions required:

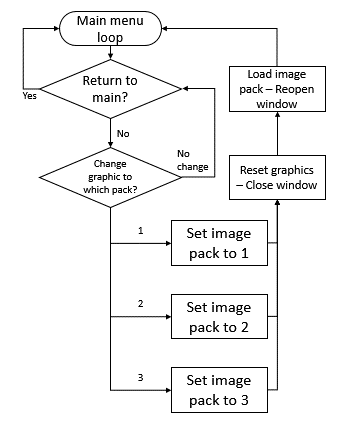
* Initialising SDL and components
* Image loading
* Updating image on the window
* Checking if the grid position is empty
* AI logic
* Checking which player has won or if it’s a draw
* Resetting the game grid
* Error printing to a file

These functions will be explained more during the development.

## Example High Level Flowcharts

This is the flowchart which shows the operations which will be handled by the main menu loop. The other loops are run when chosen via the main menu loop. The menu loop is still in the background but does not do anything when the game loop or option loop is run. The entire program is ended when the user(s) choose to end the game.

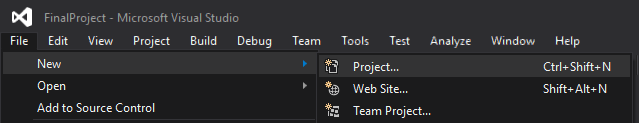
Once the users exit the game loop or the option menu loop, the users are brought back to the main menu loop. Should the user close the window manually, the program must be set so that all the loops including the main menu are closed to end the program

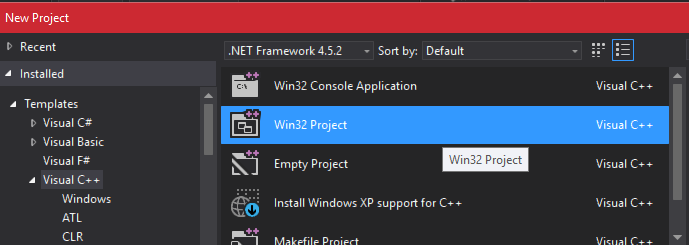
This is a flowchart that shows the actions that the code must do in the option menu loop. The only part that loops in the option menu is where the user chooses to do nothing, therefore it will loop until the user either chooses to change the image pack or chooses to return to the main menu.

If the user chooses to change the image pack, then a variable image pack will be set to the correct value relative to the user’s choice. The window will be closed and a function that loads the images will be called. The function will load the image pack related to the value of the image pack variable. The window will then be reopened with the updated image pack but will reopen the window in the main menu.

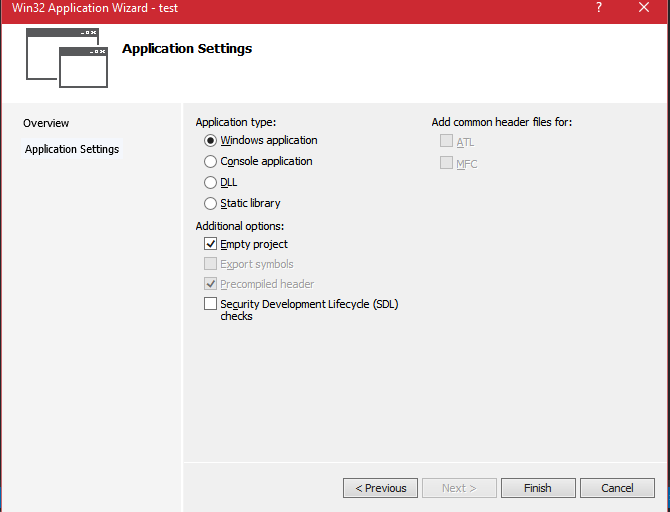
# Development

## Setting up the project

The project begins with the creation of a project in Visual Studio.



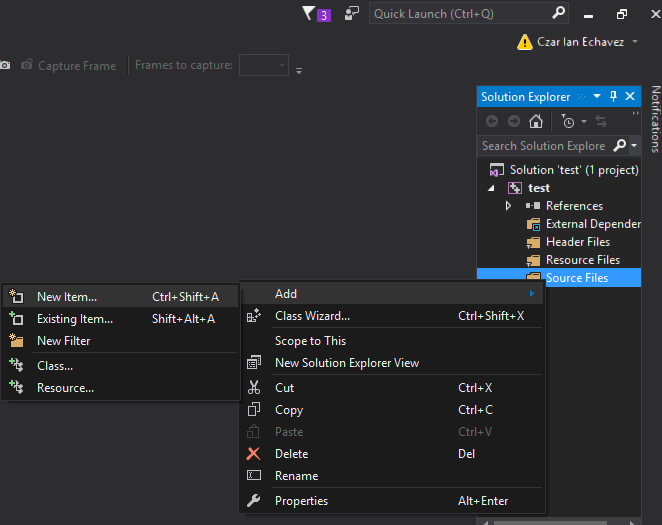
The project can be a console application, however since graphics are used, the windows application (Win32) are much more preferred for this project.



A window appears which asks for the application settings for the project. Pressing next shows the window on the left. Set Software Development Lifecycle to false and set empty project to true.

The next step is to get the SDL libraries from the SDL website: <https://www.libsdl.org/>

For this project, SDL 2 will be used. The specific version used for this project however is from: <https://buildbot.libsdl.org/sdl-builds/sdl-visualstudio/?C=N;O=A>

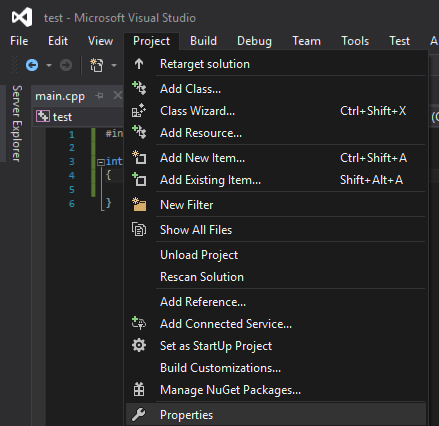


The websites were taken from the tutorial video used to help with this project: <https://www.youtube.com/watch?v=FxCC9Ces1Yg>

The steps taken in the video are the same as the steps taken in setting up SDL2 for this project as well.

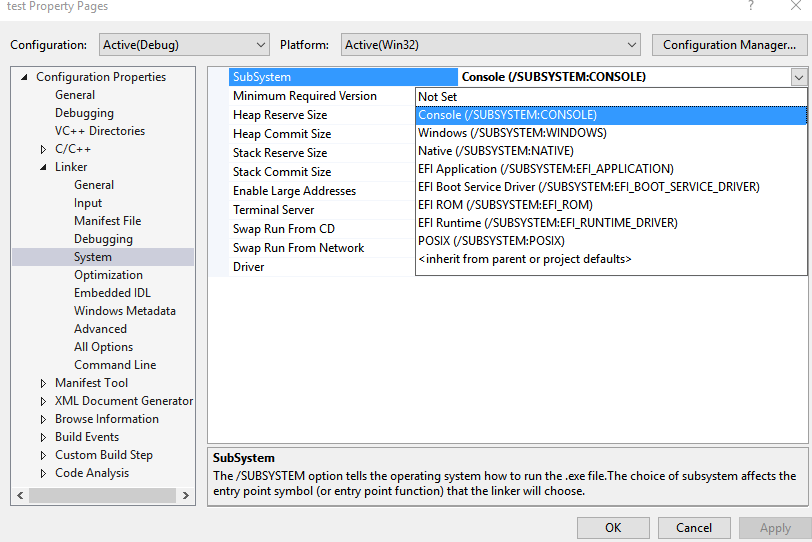
For the following example a new project is created to show the process done on the actual project.

First, the project must be compiled. This can be done by creating a main cpp file with a function with runs and then quits.

For now the project must be set as a console application, this is so that the code compiles. The code (in the main cpp) should just open, then close so that Visual Studio sets up all of the files for the project. The code should be something as simple as:

**int main()**

**{ return 0; }**

Opening Project >> Properties opens another window.

Going to Configuration Properties >> Linker >> System then choosing subsystem as console will allow the code to compile. Compile the code by going to Build >> Build Solution.

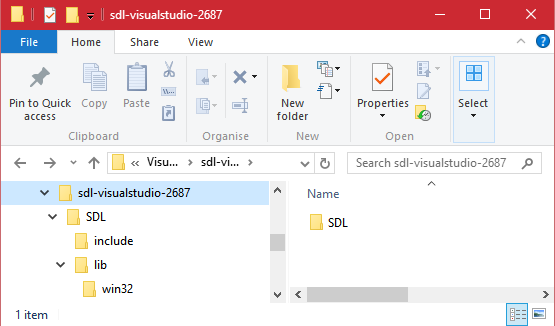
After this is done, revert the project to Windows by going back to the project properties.

## Setting up the folder system

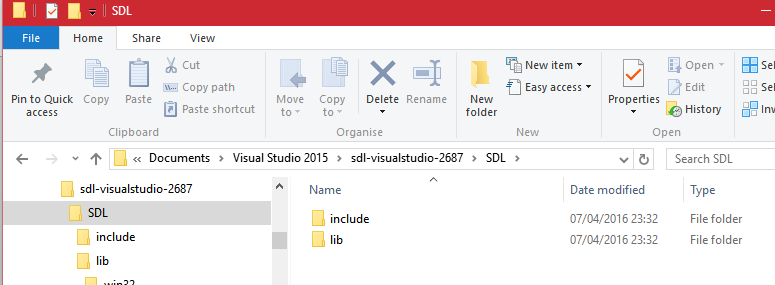
The folder for the project should look something like the image on the left afterwards (after compiling the previous).

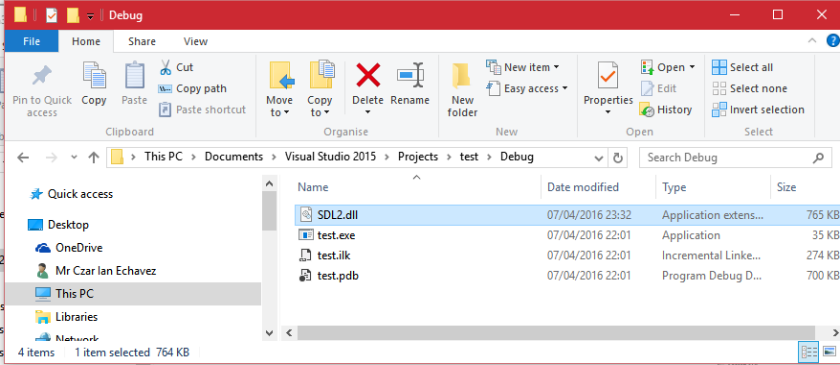
A new folder called Debug and a folder with the same name as the project will be created.

The Debug folder is required since it will contain the exe file of the project and this will also be the place where the dll files will be placed with.

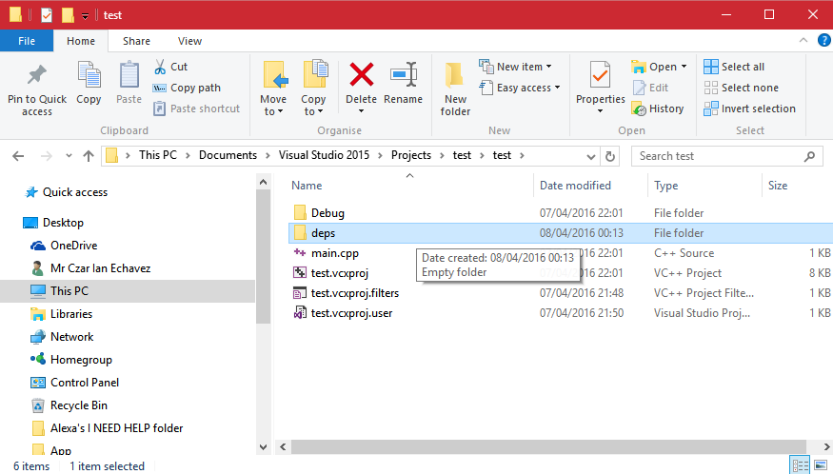
The test folder (the same name as the project) is what is needed to be set up next. Extract the SDL files from: <https://buildbot.libsdl.org/sdl-builds/sdl-visualstudio/?C=N;O=A>

These are the contents of the extracted file:

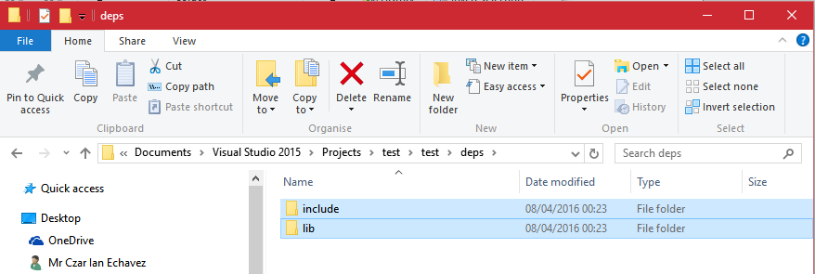
The “include” and “lib” folders are what is required by the project. These need to be copied to the folder system of the project, however, not all the folders should be placed on the same place.



The lib file needs to be placed in the same folder as the executable file, which is in the Debug folder (The outer Debug folder, not the inner Debug folder that is with the solution file).

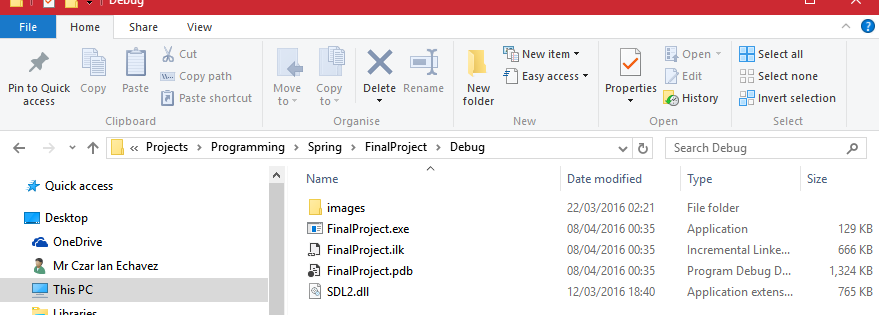


In the “test” folder (which is in the same folder as the solution file – which would have the same name as the project name) a new folder is created which will hold SDL’s include and lib file.

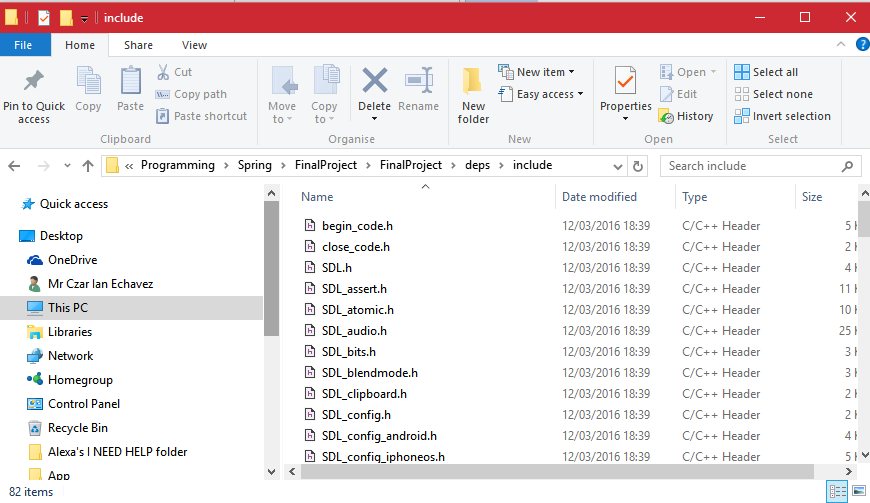
In the deps file, 2 more folders need to be created, one to hold all the include files and the other to hold the library files.

To finish this part of the project, the contents of the include folder from the extracted files need to be placed into the newly created include folder and the lib files from the extracted file need to be placed into the lib folder.

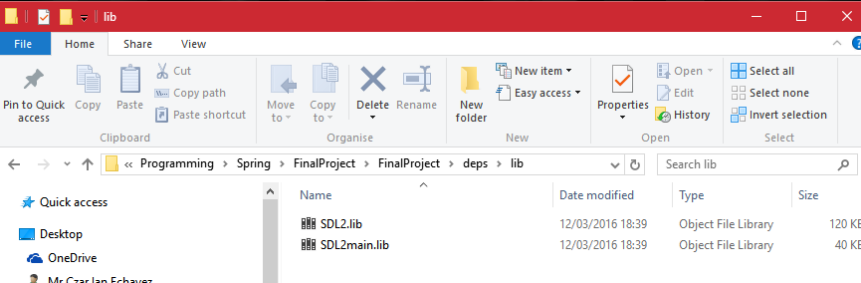
The contents of the Debug folder which contains the executable file:

The folder contains an “images” folder. This folder is an exact copy of the images folder in the solution folder. This is so that the images can be used when the user activates the executable file or uses the solution file to run the program.

The SDL2.dll is also added here. It is required by SDL to function.

The contents of the include folder:

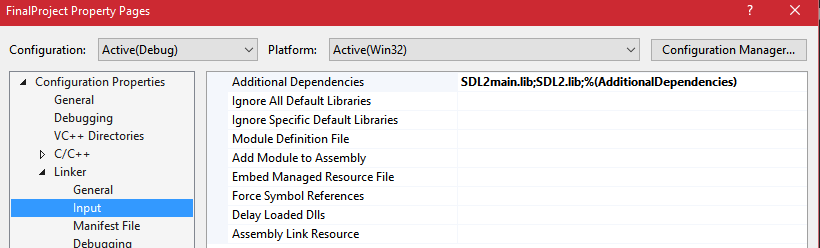
The contents are all header files which are needed for SDL. The files are the include files that will be used in the project.

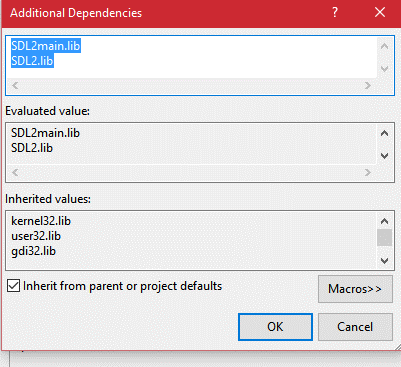


The contents of the lib folder:

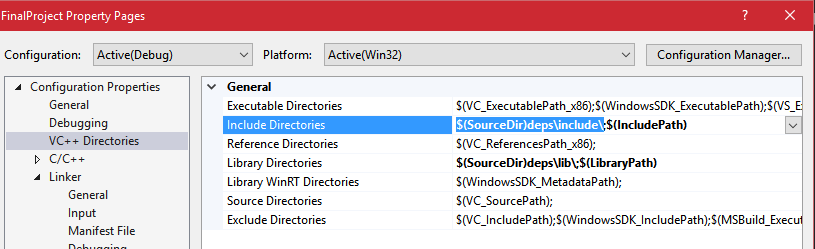
There are only 2 items in the lib folder. These are the only libraries essential for the project.

The final 2 parts of setting up the project is to open the solution file and go to the project properties.

The window on the left should open and then the setting in Configuration Properties >> Linker >> Input >> Additional Dependencies will need to be changed to include the SDL libraries.

Pressing on the Additional dependencies opens a window. In this window, the name of the SDL libraries need to be added.

Adding in “SDL2main.lib”, pressing enter and then adding “SDL2.lib” will finish this part of the setup.



The final step is telling the project where to look for the include directories and library directories i.e. the file paths for the include and lib folders.

The image above shows both the library and include directories to reference for this project, but instead of putting the raw filepath down, “$(SourceDir)” is used. This is so that the project automatically looks for where the solution folder is located and then adds the extra file paths for where within the folder the include and library files are. Source Dir is essential for this project since, it will help the person marking the project easily access a working program, this is because the project marker will not have to manually edit the file paths.

## Main cpp file

The contents of the main.cpp file are what is shown on the left. The main.cpp will only create an XO class called game. The entire game is handled by the XO class.

#include "GlobalVars.h"

int main(int argc, char \*\*argv)

{

XO game;

return 0;

}

Globalvars.h is the header file that holds all the variables needed for the game and also will be where the libraries are referenced because all the other files are going to reference Globalvars.h

Argc and argv stands for argument count and argument vector respectively.

## Creating the class (Globalvars.h header file)

On the left is all the libraries used within the header file. It is placed here because all the other files refer to the Globalvars.h header file.

#pragma once

#include "SDL.h"

#include <iostream>

#include <fstream>

#include <string>

#include <cstdlib>

using namespace std;

#pragma once allows the program to reference #includes without having to be copied over to other parts of the program. #pragma once is used in the project because it will make sure that the compiler does not get confused and reference #includes more than once.

There are 2 types of libraries referenced in this project – the SDL include files and the standard libraries.

SDL.h contains everything the project needs for bmp file processing and also key inputs.

<iostream> handles inputs and outputs of the program.

<fstream> handles inputs and outputs to external files i.e. when the program prints out errors to a txt file.

<string> and using using namespace std handles string that is used for the project, which is mostly for error handling

<cstdlib> references the C standard library

On the left is the structure of the class that is used for the project. The class is called XO and will have numerous members split between being public or private.

class XO

{

public:

//CONSTRUCTOR//

XO();

//FUNCTIONS//

//SDL SURFACES//

private:

//GLOBAL VARIABLES//

};

There is also a constructor, which is essential for classes.

Members

The public members will mostly consist of the functions required by the game and also will hold the SDL Surfaces i.e. the image that show up on screen.

The private members will only consist of variables, this is because nothing else but the class itself should access its data.

//VARIABLE WHICH DETERMINES WHICH IMAGE PACK TO LOAD//

int imgPack;

//GRID GLOBAL VARIABLE//

char grid[3][3];

//PLAYER VALUE VARIABLES//

int player1;

int player2;

//MULTIPLE FUNCTIONS NEED TO REFER TO THIS THEREFORE NEEDS TO BE GLOBAL//

//DON'T GIVE ME SASS ABOUT THIS//

bool quitPlay = false;

bool quitGame;

//AI RELATED//

bool hasAImoved = false;

bool playAI = false;

bool AIsetup = false;

int AIattackPattern = NULL; //DETERMINES HOW THE AI WILL SET ITS MOVES ON THE GRID//

int attStep = 0;

Above is all the private members of the class. As shown above, all the private members are variables. The members can be split into different types:

* Image handling
* Handles the game grid
* Handles player turns
* Menu handling
* AI handling

The variables are what is used to either store data or is checked for what state the game is currently in.

On the left are the public members of the class. Most of the members are functions, with the exception of the window size.

**The public members**

//LOADER FUNCTIONS//

bool initialise();

bool loadImg();

void windowUpdate();

//LOOP FUNCTIONS//

void mainLoop();

void optionLoop();

void gameLoop();

//CHECK FUNCTIONS//

bool checkGridEmpty(int, int);

int checkWin();

int whosTurn();

//RESET FUNCTIONS//

void resetGrid();

void close();

//ERROR FUNCTION//

void errorLog(string error);

//THE GAME'S AI//

void GameAI();

void attPattern1();

void attPattern2();

void AIattack();

void AIdefend();

void AIfailsafe();

//WINDOW AND SCREEN//

SDL\_Window\* window = NULL;

SDL\_Surface\* screen = NULL;

//GAME IMAGES//

SDL\_Surface\* gameBackground = NULL;

SDL\_Surface\* Nought = NULL;

SDL\_Surface\* Cross = NULL;

SDL\_Surface\* MainMenu = NULL;

SDL\_Surface\* OptionMenu = NULL;

SDL\_Surface\* XTurn = NULL;

SDL\_Surface\* OTurn = NULL;

SDL\_Surface\* Draw = NULL;

SDL\_Surface\* Win = NULL;

//WINDOW SIZE//

int winWidth = 500;

int winHeight = 500;

The functions can be placed into different groups according to what the function does.

The LOADER FUNCTIONS are the functions which handle all the image/variable loading. These are used to load the image files for use by the program. The functions are also used to change the graphics of the window.

The LOOP FUNCTIONS are the functions which handle the menu loops for the project. There are 3 loops, the main menu loop, the option menu loop and the game loop. These loops are activated when the user wants to access the part of the game i.e. the option menu or the game itself.

The CHECK FUNCTIONS are functions which checks against specific variables and returns values related to what the function does.

The RESET FUNCTIONS are functions which reset certain parts of the game. Reset grid sets all the grid values to 0. The close function is one part of a window reset. Close will close all surfaces and closes SDL and the other part of resetting is the initialise loader function.

The ERROR FUNCTION handles error printing i.e. will print a specific error to a problem which occurs, to an external txt file.

THE GAME'S AI are functions which will handle all the decisions that the AI makes in order to make sure it does not lose against a player/

The rest of the functions deal with the SDL Surfaces which are the graphics of the game, and the window size are variables which state the window sizes.

## Other cpp files

The highlighted cpp files are the extra cpp files that have been created and will help divide the functions into different parts. All these cpp files also use #include “Globalvars.h” to make sure the files are also able to reference other libraries and that all the functions are under a related cpp file to make sure that the project is as organised as possible.

These are the contents of “ExtraFunctions.cpp”. The cpp only contains the class’ constructor and the errorLog function, which prints errors to an external file

#include "GlobalVars.h"

XO::XO(){…}

void XO::errorLog(string error){…}

#include "GlobalVars.h"

void XO::GameAI(){…}

void XO::attPattern1(){…}

void XO::attPattern2(){…}

void XO::AIattack(){…}

void XO::AIdefend(){…}

void XO::AIfailsafe(){…}

These are the contents of “GameAI.cpp”. The cpp file contains all the functions that handle the AI of the game. The GameAI is the main function for this cpp file. The function is what will access the other AI functions. 2 of the functions are the attack patterns which are used for specific actions the AI will be doing, whereas the last 3 are the core decision making part of the AI. AI attack, defend and failsafe are functions which look at a number of possibilities so that the AI makes proper decisions to prevent the player from winning.

#include "GlobalVars.h"

bool XO::initialise(){…}

void XO::gameLoop(){…}

int XO::whosTurn(){…}

bool XO::checkGridEmpty(int x, int y){…}

int XO::checkWin(){…}

void XO::resetGrid(){…}

These are the contents of “GameLogic.cpp”. The cpp file contains the functions that handle the game’s logic including the game loop. GameLogic essentially are the core functions that make the game work.

As shown, the game loop is one of the many functions in the cpp file. The rest of the functions are the CHECK FUNCTIONS and the RESET FUNCTIONS

#include "GlobalVars.h"

bool XO::loadImg(){…}

void XO::windowUpdate(){…}

void XO::close(){…}

These are the contents of “ImgHandling.cpp”. The cpp file contains the functions that handle the loading of images. loagImg is a function that loads different images according to the image pack that needs to be loaded. windowUpdate is the function which will update the graphics of the window and close is a function that empties the SDL surfaces and closes the GUI window.

These are the contents of “MenuLoops.cpp”. There are only 2 functions and they are both loops for the menu and the option menu. The game loop is much more complex than these 2 functions and is also not a menu, therefore is placed on a separate cpp file.

#include "GlobalVars.h"

void XO::mainLoop(){…}

void XO::optionLoop(){…}

## Initialising

XO::XO()

{

//DEFAULT IMAGE PACK//

imgPack = 1;

//CALLS INITIALISE FUNCTION//

bool init = initialise();

if (init == false)

{

errorLog("Something went wrong while initialising");

SDL\_GetError();

}

//VARIABLE USED TO CHECK IF THE PLAYER WANTS TO END GAME//

quitGame = false;

mainLoop();

//QUITS PROGRAM//

close();

}

The entire program begins with the main.cpp creating an XO class. The code on the left then runs. The code sets the imgPack to 1, which is the default image pack. The code then runs the function initialise(). Once initialise is run, the constructor checks if the program was properly initialised and then runs the main menu loop. Once the main menu loop is closed, the close() function is called.

This is the first step that the initialise function does. The function creates a local variable which is preset to true. The function then does an if function to check if everything is initialised, which also initialises all SDL components.

bool init = true;

if (SDL\_Init(SDL\_INIT\_EVERYTHING) < 0)

{

//PRINTS TO FILE - SDL CANNOT INIT//

errorLog("SDL Could not be initialised, check if all SDL files are in correct folders");

//GET SDL BUILT IN ERROR//

SDL\_GetError();

init = false;

}

If SDL could not be initialised, the error log prints an error to the file and sets the local variable to false.

else

{

//CREATE A NEW WINDOW//

window = SDL\_CreateWindow(

"XO", //GAME WINDOW NAME//

SDL\_WINDOWPOS\_CENTERED, //WINDOW IS CENTERED IN X DIRECTION//

SDL\_WINDOWPOS\_CENTERED, //WINDOW IS CENTERED IN Y DIRECTION//

winWidth, //WINDOW WIDTH//

winHeight, //WINDOW HEIGHT//

SDL\_WINDOW\_SHOWN //THE WINDOW IS VISIBLE//

);

If SDL loads properly, a window is created called window, which has the properties above. The window will be called XO and is centered on the screen and has the width and height stated in the Globalvar.h header file.

The function continues in the same else statement. The window is checked to see if it has been properly created or not. If it is NULL, then the window was not created and the error is logged and the variable init is set to false.

if (window == NULL)

{

//PRINTS TO FILE - WINDOW CANNOT INIT//

errorLog("Window could not be created");

//GET SDL BUILT IN ERROR//

SDL\_GetError();

init = false;

}

else

{

//GET THE WINDOW SURFACE//

screen = SDL\_GetWindowSurface(window);

//CALLS LOAD IMG FUNCTION//

bool imgInit = loadImg();

if (imgInit == false)

{

errorLog("Something went wrong while loading images");

}

//WRITES THE GAME BACKGROUND INTO THE SCREEN//

SDL\_BlitSurface(gameBackground, NULL, screen, NULL);

//RECTANGLE USED TO DRAW MENU TO SCREEN//

SDL\_Rect Menu;

Menu.x = 25;

Menu.y = 350;

Menu.w = 420;

Menu.h = 140;

//DRAWS THE MENU TO SCREEN//

SDL\_BlitSurface(MainMenu, NULL, screen, &Menu);

//UPDATES THE WINDOW WITH THE BLITTED IMAGE//

SDL\_UpdateWindowSurface(window);

}

}

return init;

The else statement of the above if statement will continue by setting up the screen surface to connect to the window’s surface.

The loadImg function is then run and an if function checks to see if the images have been loaded properly.

The game background is then blitted to the screen surface.

An SDL Rect is then created which give the coordinate and size for the main menu to show up.

The main menu is then blitted onto the coordinates of the created rectangle and then the window’s surface is then updated.

The function then returns the value of init.

## Menu Systems

//INITIALISE AN EVENT//

SDL\_Event mainMenuInput;

//THIS RECTANGLE IS THE LOCATION OF THE MENU//

SDL\_Rect MenuRect;

MenuRect.x = 25;

MenuRect.y = 350;

MenuRect.w = 420;

MenuRect.h = 140;

### Main Menu

The first thing the main menu loop does is to set up am SDL Event called mainMenuInput, which is what will be used to see which keys are pressed i.e. for key inputs.

Another thing created is the rectangle where the option menu can be placed if the option menu is chosen by the user.

2 while loops are then created. The outer while loop will make the menu loop until the user wants to quit the game. The inner while loop will keep looping to detect any user inputs.

//THE MAIN LOOP//

while (!quitGame)

{

//WHILE AN EVENT IS BEING POLLED//

while (SDL\_PollEvent(&mainMenuInput))

{

//IF A JEY IS BEING PRESSED//

if (mainMenuInput.type == SDL\_KEYDOWN)

{

switch (mainMenuInput.key.keysym.sym)

{

case SDLK\_ESCAPE:

screen = NULL;

quitGame = true;

break;

Within the inner loop is an if statement which will run a switch case statement when a user presses down a key.

The first case statement for the main menu checks if the user wants to close the game. If the user presses the escape key, the screen is set to NULL and quitGame is set to true and therefore exits the main menu loop and closes the game.

The next case statements deal with the user inputs which accesses the other loops.

case SDLK\_1:

player1 = 1;

player2 = 0;

playAI = false;

SDL\_BlitSurface(OTurn, NULL, screen, &MenuRect);

resetGrid();

gameLoop();

break;

case SDLK\_2:

player1 = 1;

player2 = 0;

playAI = true;

SDL\_BlitSurface(OTurn, NULL, screen, &MenuRect);

resetGrid();

gameLoop();

break;

case SDLK\_3:

optionLoop();

break;

}

The top case looks at whether or not the user has pressed button “1”, if the player has pressed button “1”, the player1 and player2 variables are set up. Since this option is for playing the game without AI, the playAI variable is set to false.

The blitted surface is always O’s turn, since player O will always go first. The resetgrid and gameloop functions are then called.

Resetgrid sets all the grid arrays to 0 to make sure that the board is cleared from any previous games.

The second case looks at whether the user has pressed button “2”. As before, everything is set the same as button “1”, except, playAI is pressed as true, since this option allows the user to play against the game AI.

The final case statement for the main menu is the option for the user to access the option menu. Once the user presses button “3” the optionLoop function is run.

//ELSE IF THE WINDOW IS CLOSED//

else if (mainMenuInput.type == SDL\_QUIT)

{

quitGame = true;

}

Outside the switch statement, but still within the same while loop, an else if statement is used to check whether the user has closed the window i.e. pressing the “X” in the top right of the window.

If the user closes the window, the variable quitGame is set to true to exit the main menu loop and the program is closed.

### Option Menu

SDL\_Rect OptBox;

OptBox.x = 25;

OptBox.y = 350;

OptBox.h = 140;

OptBox.w = 420;

SDL\_BlitSurface(OptionMenu, NULL, screen, &OptBox);

windowUpdate();

SDL\_Event optionMenuInput;

bool returntoMain = false;

The first thing the option menu loop does is set up the graphic for the option menu.

The SDL Rect sets up a rectangle which the blitted image will be placed on.

windowUpdate will make the window update so the user can see the option menu.

The function then makes an event for looking at the option menu inputs and also a Boolean called returntoMain which is a variable will be checked to see whether the user would like to the return menu or not.

while (!returntoMain)

{

while (SDL\_PollEvent(&optionMenuInput))

{

if (optionMenuInput.type == SDL\_KEYDOWN)

{

switch (optionMenuInput.key.keysym.sym)

{

case SDLK\_ESCAPE:

SDL\_BlitSurface(MainMenu, NULL, screen, &OptBox);

windowUpdate();

returntoMain = true;

break;

The same as the main menu, 2 loops are made, one to keep looping until the user want to return t main menu and another to keep looping to check for user inputs.

The if statement checks for when the user has pressed down a button.

The first case statement blits the main menu graphic to the screen and calls windowUpdate function so the user can see the updated graphic. The variable returntoMain is then set to true so that the option menu loop can be closed and the main menu loop is resumed.

case SDLK\_1:

imgPack = 1;

close();

initialise();

returntoMain = true;

break;

case SDLK\_2:

imgPack = 2;

close();

initialise();

returntoMain = true;

break;

case SDLK\_3:

imgPack = 3;

close();

initialise();

returntoMain = true;

break;

The next case statements look at the user’s inputs if they want to change the image pack. The cases changes the image pack according to the input i.e. the key that they pressed.

All these cases then do the same thing i.e. closes the window and reinitialises by calling the close and initialise functions.

returntoMain is then set to true so that the option menu is closed and the program can return to the main menu.

else if (optionMenuInput.type == SDL\_QUIT)

{

returntoMain = true;

quitGame = true;

}

An else if statement is added for when the user closes the window, the variables returntoMain and quitGame is set to true so the program is quit.

## Game Loop

SDL\_Event gameInput;

while (!quitPlay)

{

while (SDL\_PollEvent(&gameInput))

{

int turn;

if (gameInput.type == SDL\_KEYDOWN)

{

switch (gameInput.key.keysym.sym)

{

As per usual, in the loops, an SDL Event is created as well as 2 while loops, with one within another.

The outer while loop looks at when the user wants to quit playing, while the other looks at any of the user’s input. Within the inner while loop, a local variable called turn is created. This variable is used to determine which player’s turn it is.

The other components within the inner while loop is an if function which checks whether a user has pressed a button down, which leads to a switch case statement to check which button the user has pressed.

case SDLK\_2:

if (checkGridEmpty(1, 0) == true)

{

turn = whosTurn();

grid[1][0] = turn;

if (playAI == true)

{

SDL\_Delay(100);

whosTurn();

GameAI();

}

}

break;

On the left is the same structure as all the options while the game is active. The case will only do something while the grid the button is assigned to is empty, this is to avoid the users from overwriting each other’s moves i.e. if player 1 chooses button “1” and player 2 also chooses button “1” then the value of button “1” will remain as player 2’s move even though player 1 chose it first.

To check if the grid is empty, the function checkGridEmpty is called, with the array position of the related option (button key).

If the option is available i.e. the related grid is empty, then the function whosTurn is called and is given to the local variable turn and that value is passed to the grid array of the related option.

The next part checks to see whether or not the game is currently vs AI or vs human players. If the current game is vs AI, the small delay is done and whosTurn is called to make sure that the current turn is the AI’s turn. The GameAI function is then called.

The last case statement in the game loop input is the Escape button. If the player presses escape, then the game loop is closed by setting quitPlay to true.

case SDLK\_ESCAPE:

quitPlay = true;

break;

}

}

else if (gameInput.type == SDL\_QUIT)

{

quitPlay = true;

quitGame = true;

}

Outside the switch statement but in the same while loop, an else if statement is used to check whether the user has closed the window or not.

The last part the loops do is to check whether or not there is a win or if the game is a draw. If there is a win or a draw, the if statements check which player has won or if there is a draw by checking the value of the variable winner.

windowUpdate();

int winner = checkWin();

if (winner != 0)

{

SDL\_Rect winMsg;

winMsg.x = 130;

winMsg.y = 255;

winMsg.w = 75;

winMsg.h = 75;

//IF NOUGHTS WINS//

if (winner == 1)

{

SDL\_BlitSurface(Nought, NULL, screen, &winMsg);

winMsg.x += 75;

winMsg.w = 150;

SDL\_BlitSurface(Win, NULL, screen, &winMsg);

}

//IF CROSS WINS//

if (winner == 2)

{

SDL\_BlitSurface(Cross, NULL, screen, &winMsg);

winMsg.x += 75;

winMsg.w = 150;

SDL\_BlitSurface(Win, NULL, screen, &winMsg);

}

//IF DRAW//

if (winner == 3)

{

winMsg.x = 175;

winMsg.w = 150;

SDL\_BlitSurface(Draw, NULL, screen, &winMsg);

}

quitPlay = true;

}

windowUpdate();

If winner is equal to 1, then the nought player has won.

If winner is equal to 2, then the cross player has won.

And finally if winner is equal to 3, the game is a draw.

The appropriate graphic is then blitted to the surface by using the SDL Rect that was defined above.

The value of quitPlay is then changed to true and windowUpdate is called to update the window graphics.

resetGrid();

playAI = false;

AIsetup = false;

AIattackPattern = 0;

hasAImoved = false;

attStep = 0;

SDL\_Delay(3000);

close();

initialise();

Outside both loops is the resetting of variables. resetGrid is a function which sets all the array elements to 0 so the grid is set for the next game. The next is to set all the global variables used in the game to 0 or false so that they can be used for the next game.

An SDL Delay of 3 seconds (3000 milliseconds) is added so that the player(s) are able to see who has won before the next step is done i.e. the window is closed and then reinitialised in the main menu.

## Key Inputs

//THE MAIN LOOP//

while (!quitGame)

{

//WHILE AN EVENT IS BEING POLLED//

while (SDL\_PollEvent(&mainMenuInput))

{

//IF A JEY IS BEING PRESSED//

if (mainMenuInput.type == SDL\_KEYDOWN)

{

switch (mainMenuInput.key.keysym.sym)

{

case SDLK\_ESCAPE:

screen = NULL;

quitGame = true;

break;

The reason why the input checking is within 2 loops and an if function is so that the input is check while the related loop is active i.e. the input is within the main menu, option menu or during the game loops.

The inner loop will keep looping while the program is polling an event i.e. while the program is still in an input state, the loop will keep iterating until the input is no longer needed to be checked. The if statement inside both loops is used to check if a button has been pressed down. If a button is being pressed down, then the program will so a switch case statement to see which of the buttons have been pressed down and will do a specific action according to the button being pressed and also which menu system the user is on. Escape will remain consistent because it will be the button which will let the user leave the current menu or game and also closes the game if in the main menu.

## Extra Functions

### Initialising SDL and components

SDL\_Init(SDL\_INIT\_EVERYTHING)

On the left is the line of code which is used to initiate all of SDL’s subsystems. This code is used in the initialise function as an if condition to check if the SDL subsystems have been properly initialised. Even if the code is within an if condition, it is still run just like any other function.

SDL\_Init is a built in function made by SDL and is compulsory in order to run all its components.

### Image loading (loadImg)

bool XO::loadImg()

{

bool success = true;

if (imgPack == 1) {…}

if (imgPack == 2) {…}

if (imgPack == 3) {…}

return success;

}

This is the basis of the loadImg function. A local variable called success is also created, which will hold the value which the function will return, as shown at the bottom by “return success”.

There are also 3 if statements that are done to check which image pack was chosen by the user i.e. when the user was on the option menu.

Each image pack contains the same number of images but have different graphics.

On the left is the process of loading an image, which will be the same for all the other images.

gameBackground = SDL\_LoadBMP("images/pack1/XOBoard.bmp");

if (gameBackground == NULL)

{

errorLog("Image could not be loaded: XOBoard.bmp");

SDL\_GetError();

success = false;

}

The SDL\_LoadBMP line will load the bmp image to the SDL Surface related to the image. An if function is then used to check whether or not the image was successfully loaded.

An if function is then used to check whether or not the image has been loaded properly. If the image has not been loaded, the errorLog function is called and logs the error to an external file. The SDL\_GetError is a built in function for SDL to get errors. The variable success is then changed to false, since it will be the value needed to be returned by the function.

### Updating image on the window (windowUpdate)

void XO::windowUpdate()

{

if (quitPlay == false) {…}

SDL\_UpdateWindowSurface(window);

}

These are the contents of windowUpdate. There are 2 main components – an if statement to check if the program is in game and the other component is an SDL function to update the surface that is going to show up on the window.

The contents of the if function are the processes needed for the game to properly display the noughts and crosses while the program is in game.

The first thing done within the if statement is to create a new rectangle where the noughts and crosses will be placed in.

SDL\_Rect rect;

rect.x = 132;

rect.y = 10;

rect.w = 75;

rect.h = 75;

if (grid[0][0] == 2)

{

SDL\_BlitSurface(Cross, NULL, screen, &rect);

}

else if (grid[0][0] == 1)

{

SDL\_BlitSurface(Nought, NULL, screen, &rect);

}

Each specific position of the nought and crosses game grid has its own rectangle coordinate but has the same sizes.

If statements are then used to check the value of the array that the position of the grid is related to. According to the value of the grid position, the appropriate image is blitted onto the position of the rectangle, which is the position that the user has chosen to put a move on.

### Checking if the grid position is empty (checkGridEmpty)

bool XO::checkGridEmpty(int x, int y)

{

if (grid[x][y] == 0)

{

return true;

}

else

{

return false;

}

}

checkGridEmpty is the function used to check if the specific part of the grid is empty or not. The function takes in 2 “int” inputs which are used to check the x and y i.e. the 2d array coordinate which will be checked. The function will then return a value of true or false according to the value of the part of the grid.

### Checking whose turn it is (whosTurn)

int XO::whosTurn()

{

SDL\_Rect playerTurn;

playerTurn.x = 25;

playerTurn.y = 350;

playerTurn.w = 420;

playerTurn.h = 140;

if (player1 > player2) {…}

else if (player2 >= player1) {…}

}

On the left is the function used to check which player’s turn the current game is on. There are 3 parts to the function, first a rectangle is created to be the place where the graphic to notify the players whose turn it is. The other 2 deal with finding out whose turn it is and shifting the variables to swap the turn to the other player and to blit the image to the surface so that the players can see the graphic.

if (player1 > player2)

{

player2++;

SDL\_BlitSurface(XTurn, NULL, screen, &playerTurn);

return 1;

}

else if (player2 >= player1)

{

player1++;

SDL\_BlitSurface(OTurn, NULL, screen, &playerTurn);

return 2;

}

These are the contents of the if statements. The first thing both statements do is to find out which player has a higher value, therefore find whose turn it is. Then the if statement increments the variable for the player that is to have their turn next. The graphic which says whose turn it is gets blitted to the surface and then the function returns which player’s turn it currently is.

### Checking which player has won or if it’s a draw (checkWin)

int XO::checkWin()

{

int win = 0;

The checkWin function is what is used to determine whether the game has reached a win or a draw. The first thing the function does is to create a local variable which is set to 0.

Then an if statement is used to check whether or not the game has reached a draw. This is done first because it allows the value of win to be overwritten should the winning move be the final move done. The if statement also looks at all the values of the grid arrays. If none of the arrays are 0, this means that the players have placed a move there and once the grid is filled but there are no winners, then the game is a draw i.e. win = 3.

if (grid[0][0] != 0 && grid[0][1] != 0 && grid[0][2] != 0 &&

grid[1][0] != 0 && grid[1][1] != 0 && grid[1][2] != 0 &&

grid[2][0] != 0 && grid[2][1] != 0 && grid[2][2] != 0)

{

win = 3;

}

On the left is the process of checking the win states. The function will have 3 different checks: horizontal, vertical and diagonal. The function looks at all the possible combinations of grid values that would form the above i.e.

//HORIZONTAL WIN STATES//

if ((grid[0][0] == 1 && grid[1][0] == 1 && grid[2][0] == 1) || (grid[0][1] == 1 && grid[1][1] == 1 && grid[2][1] == 1) ||

(grid[0][2] == 1 && grid[1][2] == 1 && grid[2][2] == 1))

{

win = 1;

}

//VERTICAL WIN STATES//

if ((grid[0][0] == 1 && grid[0][1] == 1 && grid[0][2] == 1) ||

(grid[1][0] == 1 && grid[1][1] == 1 && grid[1][2] == 1) ||

(grid[2][0] == 1 && grid[2][1] == 1 && grid[2][2] == 1))

{

win = 1;

}

//DIAGONAL WIN STATES//

if ((grid[0][0] == 1 && grid[1][1] == 1 && grid[2][2] == 1) ||

(grid[2][0] == 1 && grid[1][1] == 1 && grid[0][2] == 1))

{

win = 1;

}

|  |  |  |
| --- | --- | --- |
| A | B | C |
| A | B | C |
| A | B | C |

|  |  |  |
| --- | --- | --- |
| A | A | A |
| B | B | B |
| C | C | C |

|  |  |  |
| --- | --- | --- |
| A |  | B |
|  | AB |  |
| B |  | A |

The function also checks for player 2’s win states. The player 2 check is the same as above however, the if statements compare the grid[x][y] to 2 instead of 1 and also runs win = 2; instead of win = 1;

### Resetting the game grid (resetGrid)

void XO::resetGrid()

{

//THIS IS TO RESET THE GRID TO ZERO VALUES SHOULD THE PLAYERS WANT TO PLAY ANOTHER GAME//

grid[0][0] = 0;

grid[0][1] = 0;

grid[0][2] = 0;

grid[1][0] = 0;

grid[1][1] = 0;

grid[1][2] = 0;

grid[2][0] = 0;

grid[2][1] = 0;

grid[2][2] = 0;

quitPlay = false;

windowUpdate();

}

This is the function which is used to clear the grid so that it can be reused for another game. The function sets all the possible array values to 0 and then sets quitPlay to false and then calls the windowUpdate function in order to update the graphics so that the noughts and the crosses are removed from the screen.

The function allows the game to be reused and allows the game to be played more than once before the user wants to close the program.

### Error printing to a file

void XO::errorLog(string error)

{

ofstream errorlog;

//OPEN (OR CREATE) errorlog.txt//

errorlog.open("errorlog.txt", ios::app);

//APPEND error INTO FILE//

errorlog << error << endl;

//CLOSE FILE//

errorlog.close();

}

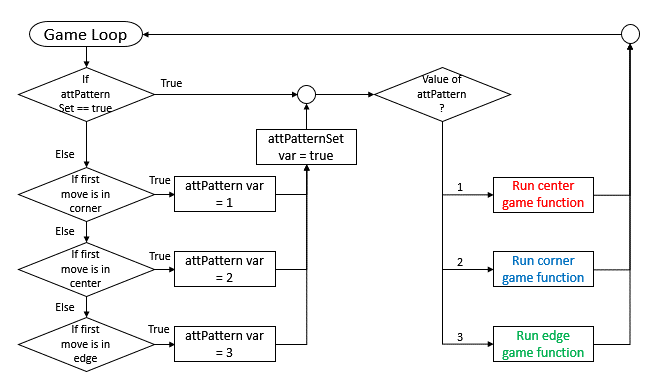
This is the function that handles the error printing to a file. The function takes a string input called error. The string is what the function prints to a text file.

The function first looks for a file called errorlog then opens one in append mode. The error is then put into the file and the errorlog file is then closed.

The errorlog file can be accessed via the folder system of the project.

## AI logic

The AI is the most complex part of this project. The AI requires multiple functions that all intertwine and not all the functions may be used in one run of the game loop, however with multiple runs and if the user uses multiple different initial moves, these functions will be required.

To easily explain how the AI works, multiple different flowcharts will be used to explain the functions and the actions required in order for a working AI.

The flowchart above shows the AI setup. The left side of the flowchart is only executed once per run, this is to set up the attack pattern for the current game.

The attack patterns make the AI harder to beat since in noughts and crosses, the opponent automatically loses if they do a move on specific areas. The only weakness of this AI would be when the player begins on the edge, since the algorithm for that is ridiculously complex. Instead the edge game will end up just the AI using a combination of attacking, defending and random moves.

The code on the left follows the flowchart on the previous page, however, one component is added and it is to check if there is already a win, if there is a win or a draw, the AI would not be able to do anything since the hasAImoved variable is set to true. This is because the AI may still perform a move even after the player has won or if it is a draw.

void XO::GameAI()

{

//CHECKS IF THE PLAYER HAS ALREADY WON IF THE PLAYER HAS//

//ALREADY WON, THE AI IS SHUT//

if (checkWin() != 0)

{

hasAImoved = true;

}

//SETS UP THE AI'S ATTACK PATTERN//

if (AIsetup == false) {…}

if (AIattackPattern == 1) {…}

if (AIattackPattern == 2) {…}

if (AIattackPattern == 3) {…}

}

Below that is the AI setup component and the AI attack patterns.

The AI setup will initially set the variable AIsetup to true. This is so that the AIsetup is only called once per game loop iteration.

if (AIsetup == false)

{

AIsetup = true;

if (grid[0][0] == 1 ||

grid[0][2] == 1 ||

grid[2][0] == 1 ||

grid[2][2] == 1)

{

AIattackPattern = 1;

}

if (grid[1][1] == 1)

{

AIattackPattern = 2;

}

if (grid[1][0] == 1 ||

grid[0][1] == 1 ||

grid[2][1] == 1 ||

grid[1][2] == 1)

{

AIattackPattern = 3;

}

}

The AI setup’s purpose is to set the AIattackPattern which is what the AI will be following until the end of the game iteration (loop). The AIsetup looks at what the player’s first move is to determine the attack pattern. The AI will use attack pattern 1 if the player started in the corner. The AI will use attack pattern 2 if the player starts in the middle and finally attack pattern 3 if the player started on an edge that is not a corner.

### Attack Pattern 1

The flowchart on the left shows the process in attack pattern 1. The attack pattern always begins by having the AI do a move on the center. Then the AI will do a move on any of the edges. After this, the AI will just use a combination of attack, defending and random moves to make sure the player does not win.

In the main GameAI function, this is what calls the attack pattern 1 function.

if (AIattackPattern == 1)

{

attPattern1();

}

void XO::attPattern1()

{

if (attStep == 0)

{

grid[1][1] = 2;

hasAImoved = true;

}

On the left is the attPattern1 function i.e. the attack pattern 1. The first thing that the function does is check if the value of attStep is 0. attStep is a variable that is used to check which step the AI is currently in so that it can perform the correct moves in the correct order.

attStep == 0 means that it is the AI’s first move. Therefore in attack pattern 1, the first thing the AI does is to do a move in the center.

The next step for the AI to do in attack pattern 1 is to first check if the human player has any moves that could form 3 in a row. If the player does not have any moves that form 3 in a row in their next turn, then the AI will choose a move which is in an edge.

if (attStep == 1)

{

if (hasAImoved == false)

{

AIdefend();

}

while (!hasAImoved)

{

int choice = rand() % 4 + 1;

if (choice == 1 && grid[1][0] == 0)

{

grid[1][0] = 2;

hasAImoved = true;

break;

}

if (choice == 2 && grid[0][1] == 0)

{

grid[0][1] = 2;

hasAImoved = true;

break;

}

if (choice == 3 && grid[2][1] == 0)

{

grid[2][1] = 2;

hasAImoved = true;

break;

}

if (choice == 4 && grid[1][2] == 0)

{

grid[1][2] = 2;

hasAImoved = true;

break;

}

}

}

The choosing of a random edge is done by a loop which will keep looping until the AI has done a move.

A randomised variable always gets a random value between 1 and 4 per iteration in the loop and then the AI does a move according to what the value of choice was.

The AI will only do a move if the grid choice was empty, if not, then it will loop again, with a new choice value and will loop until a choice is made.

On the right is the last part of the attack pattern. The AI no longer needs to do specific moves and therefore will just need to attack, then defend, then use random moves until the end of the game.

else

{

AIattack();

AIdefend();

AIfailsafe();

}

attStep++;

The attack pattern then finishes by incrementing the value of attStep.

### Attack Pattern 2

On the left is the flowchart which shows the process done in attack pattern 2. In attack pattern 2, the first thing the AI does is to choose a random corner.

The next step for the AI to do is to attack or defend, if not the AI will need to choose another random corner. Finally the AI will need to attack, defend or choose randomly to make sure that the player does not win.

if (AIattackPattern == 2)

{

attPattern2();

}

This is the code that runs the attack pattern 2 function. The code is in the main GameAI function.

void XO::attPattern2()

{

if (attStep == 0)

{

while (!hasAImoved)

{

int choice = rand() % 4 + 1;

if (choice == 1 && grid[0][0] == 0)

{

grid[0][0] = 2;

hasAImoved = true;

break;

}

if (choice == 2 && grid[2][0] == 0)

{

grid[2][0] = 2;

hasAImoved = true;

break;

}

if (choice == 3 && grid[0][2] == 0)

{

grid[0][2] = 2;

hasAImoved = true;

break;

}

if (choice == 4 && grid[2][2] == 0)

{

grid[2][2] = 2;

hasAImoved = true;

break;

}

}

}

This is the attack pattern 2 function. The first thing the attack pattern does is to do a move on a random corner.

The randomisation is the same as in attack pattern 1. The while loop will keep looping until a choice is made.

Choice is a local variable that will keep getting new randomised values from 1-4 in each loop iteration.

The second thing the attack pattern does is the same as the first step, however the AI needs to do an AI attack and defend before doing a move on a corner.

The function then does AI attack, defend or does a random move until the end of the game to make sure that the player does not win.

### Attack Pattern 3

if (AIattackPattern == 3)

{

if (hasAImoved == false)

{

AIattack();

}

if (hasAImoved == false)

{

AIdefend();

}

if (hasAImoved == false)

{

AIfailsafe();

}

}

Attack Pattern 3 only requires a small amount of code and does not have its own function, instead, attack pattern 3 is inside the main GameAI function.

The attack pattern only consists of an AI attack, defend or a random move, since the actual algorithm needed to ensure the player does not lose is very complex and would take too long to code.

### 

for (int i = 0; i <= 2; i++)

{

//LOOKS AT THE LEFT OF ROW I//

if (grid[i][0] == 0 &&

grid[i][1] == 2 &&

grid[i][2] == 2 &&

hasAImoved == false)

{

grid[i][0] = 2;

hasAImoved = true;

break;

}

//LOOKS AT THE MIDDLE OF ROW I//

if (grid[i][1] == 0 &&

grid[i][0] == 2 &&

grid[i][2] == 2 &&

hasAImoved == false)

{

grid[i][1] = 2;

hasAImoved = true;

break;

}

//LOOKS AT THE RIGHT OF ROW I//

if (grid[i][2] == 0 &&

grid[i][0] == 2 &&

grid[i][1] == 2 &&

hasAImoved == false)

{

grid[i][2] = 2;

hasAImoved = true;

break;

}

}

### AI Attack and defend

Both AI attack and defend have the same process of checking. On the left is the AI attack, AI defend uses the same concept as AI attack, but instead of checking if grid element is equal to 2, the check is instead done to see if the element is equal to 1.

Both AI attack and AI defend will have an action which will set the chosen grid element to 2 i.e. the AI’s move.

The code currently on the left is the loop which is used to check the horizontal (rows) of the grid to see if there are any 2 in a row so that the AI can form a 3 in a row and therefore win. There is also a separate loop for the vertical checking where instead of the left element is incremented in the loop the right element is incremented (i.e. grid[i][0] -> grid[0][i])

The diagonal checks do not use a loop and therefore instead use if statements.

On the left is the process of checking the diagonals for any 2 in a row. This cannot be done in a loop and had to be hardcoded into the program.

//LOOK AT TOP LEFT//

if (grid[0][0] == 0 &&

grid[1][1] == 2 &&

grid[2][2] == 2 &&

hasAImoved == false)

{

grid[0][0] = 2;

hasAImoved = true;

}

//LOOK AT TOP RIGHT//

if (grid[0][2] == 0 &&

grid[1][1] == 2 &&

grid[2][0] == 2 &&

hasAImoved == false)

{

grid[0][2] = 2;

hasAImoved = true;

}

//LOOK AT BOTTOM LEFT//

if (grid[2][0] == 0 &&

grid[1][1] == 2 &&

grid[0][2] == 2 &&

hasAImoved == false)

{

grid[2][0] = 2;

hasAImoved = true;

}

//LOOK AT BOTTOM RIGHT//

if (grid[2][2] == 0 &&

grid[1][1] == 2 &&

grid[0][0] == 2 &&

hasAImoved == false)

{

grid[2][2] = 2;

hasAImoved = true;

}

//LOOK AT CENTER//

if (grid[1][1] == 0 &&

((grid[0][0] == 2 && grid[2][2] == 2) ||

(grid[2][0] == 2 && grid[0][2] == 2)) &&

hasAImoved == false)

{

grid[1][1] = 2;

hasAImoved = true;

}

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

The comments tell which part of the grid the if statement is currently looking at. By looking at the top left, the if statement looks at what diagonal can be created with that position i.e.

Where the red shading shows what the if statement will look at. The if statement will check if the top left is empty, while the if statement also check if the other red square are either filled by 2’s, so the AI can win, or filled by 1’s so the AI prevents the player from winning.

void XO::AIfailsafe()

{

while (!hasAImoved)

{

int choice = rand() % 9 + 1;

### AI failsafe

On the left is the AIfailsafe function also known as the random choice function. The function has a loop which will keep looping until the AI has made a choice. In each iteration of the loop, the variable choice is given a random value between 1 – 9.

if (choice == 1 && grid[0][0] == 0)

{

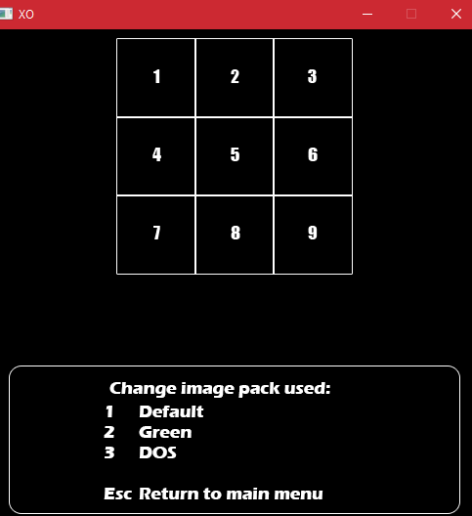
grid[0][0] = 2;

hasAImoved = true;

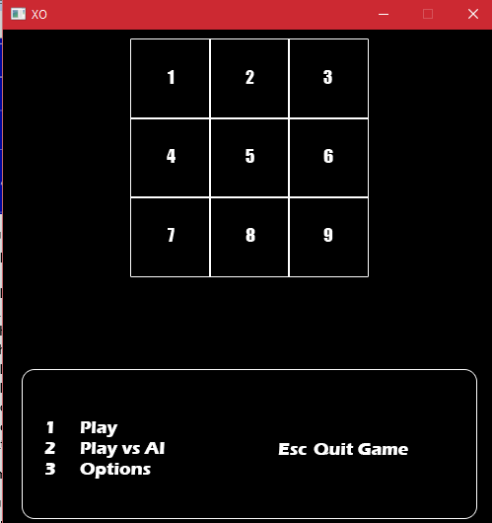
}

An if statement then checks the value of the choice variable and also checks if the grid element linked to the choice value is empty. If the grid element is empty then the AI will do a move on that position and set the hasAImoved variable to true.

## Meeting requirements

* **Has graphics**
* **The user should be able to choose which style of graphics they want via the options menu**

The program has met the above requirements since the program makes use of graphics and also the user(s) are able to choose different graphics from an option menu, as shown by the images of the left.



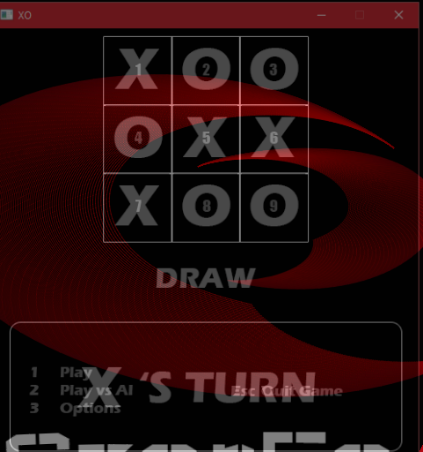
* **User can choose between different items on the main menu i.e. play vs human, play vs AI or options menu**

On the left is the main menu of the game. The player(s) are able to choose between the different items, such as Play, Play vs AI, Options and Quitting the game. The user(s) are able to access these options by pressing the buttons related to the item that they would like to access.

* **Play vs AI will set the game to let the AI take the other turn instead of another human player**

The option Play vs AI allows the user(s) to play vs an AI which takes its turn after the user, instead of needing another player to take the next turn. (Cannot be shown by an image)

Therefore the requirement has been met.

* **A graphic which tells which player has won or if it’s a draw**
* **The correct graphic must be shown i.e. when its nought’s turn, it must show a nought on the chosen part of the grid**

(The screenshots are partially transparent for an unknown purpose, the background is my computer’s screensaver)

As shown on the left, there is a graphic between the menu and the grid which shows which player has won. The graphic can also show “Draw” if the game grid is filled but no one has won.

* **Players cannot write over a square that is not empty**

This requirement is met by ensuring that the button does not do anything if the part of the grid the player wants to do a move on is already filled, i.e. the if statement inside the button’s case statement which checks the grid element’s value before doing anything

* **Player can quit whenever they want**

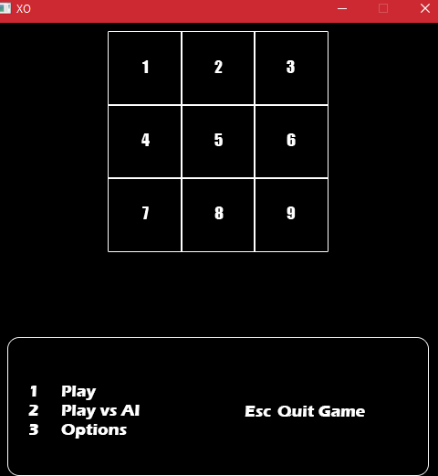
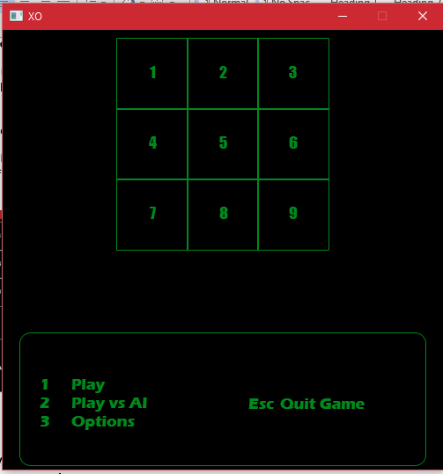
The requirement is met because there are 2 ways the player(s) can quit the game, and they can quit the game whenever they want. The player(s) can close the window, which in turn turns off the game, or the player(s) can quit the game by pressing the Escape button.

* **Consistent style throughout the program**

The style remains consistent throughout the entire program, since the text font remains the same through the entire graphics style and the colours also remain consistent throughout the graphics style.

* **Records and saves any errors to a file**

The function errorLog is able to create an external file which will save the related error to the external file. The errorLog function saves the error in a way which adds the latest error to the previous, i.e. appends the data.

* **At least 3 different graphics styles for the user to choose from**

On the left are the 3 different graphics styles that the player(s) can choose from.

These settings can be accessed via the Options Menu. The 3 graphics styles are as shown:

* Default, which looks like the console
* Green, which looks… green?
* DOS, which is the blue version of the Disk Operating System

Programming requirements:

* **Using C++ programming language**
* **Built for Windows OS and is a windows application**
* **Uses SDL libraries**
* **Project settings for Visual Studio must be all set to make sure that the people marking the project won’t have to go through the struggle of setting the project up**

The requirements above are met. The programming language used to create the program was C++ and works in the windows operating system. The program can be run by an executable file i.e. is a windows application.

The program also uses an external library i.e. SDL. SDL is used to allow the program to use graphics.

The project settings have also been set so that the project is ready to use by the project marker.

Optional:

* **Mouse input**
* **Game Sounds**
* **Sounds Option in the option menu**

The above optional requirements were not met due to time constraints. While the knowledge is available to apply the mouse input into the project, there is not enough time to apply the knowledge to the project and also test if the mouse input would work.

Game sounds and the ability to choose sounds from the option menu could also not be applied to the project due to time constraints and the fact that researching how to do this for the project would take too long and not allow for enough time for testing.

The project also requires the student to create a game, so long that it is limited to a square grid. This project is a game that meets that requirement because the game is limited to a square grid that is 3 X 3 and that is done by the use of an array game board.

# Testing

## Main Menu testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Input Type** | **Expected output** | **Actual output** | **Pass** | **Comments** |
| 1 | Valid | Starts the game without the AI | The game is started, but without the AI | Yes |  |
| 2 | Valid | Starts the game with the AI | The game is started, with the AI | Yes |  |
| 3 | Valid | Opens the option menu | The options menu is opened in the same area as the main menu | Yes |  |
| 4 | Invalid | Nothing | Nothing | Yes |  |
| W | Invalid | Nothing | Nothing | Yes |  |
| Esc | Valid | Closes the game | Closes the game | Yes | Window is closed and the program is closed |

Above are the tests done on the main menu part of the game. The tests done checks all the valid key inputs for the game as well as some key inputs which should not have done nothing. There were no problems encountered as all valid keys performed their set tasks and invalid key inputs did not affect anything in the game. The main menu has passed testing.

## Option Menu testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Input Type** | **Expected output** | **Actual output** | **Pass** | **Comments** |
| 1 | Valid | Opens the main menu with no change | Opened the main menu with no change | Yes | No change because graphics are currently set image pack. Window is closed and reopened to main menu |
| 2 | Valid | Opens the main menu, changes graphics to green image pack | Opened the main menu, changed the graphics to green image pack | Yes | Window is closed and reopened to main menu |
| 3 | Valid | Opens the main menu, changes graphics to DOS image pack | Opens the main menu, changed graphics to DOS | Yes | Window is closed and reopened to main menu |
| 4 | Invalid | Nothing | Nothing | Yes | Key is not set to any input |
| W | Invalid | Nothing | Nothing | Yes | Key is not set to any input |
| Esc | Valid | Closes the option menu, does not change any graphics | Closes the option menu, did not change any graphics | Yes | Window is closed and the program is closed |

Above is the testing performed on the option menu part of the game. The tests done not only checks that the valid key inputs work, but also that the graphics are changed to the correct image pack. The invalid keys did not affect the game and the Esc key performs what it should do. The option menu has passed testing.

## Play testing

### Play vs human

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Input Type** | **Expected output** | **Actual output** | **Pass** | **Comments** |
| 1 | Valid | ONLY puts a nought on the top left of the grid | Only puts a nought on the top left of the grid | Yes | To test that the AI is not activated, since the option is only for playing vs another player |
| 2 | Valid | Puts a cross on the middle top of the grid | Puts a cross on the middle top of the grid | Yes | The test is continued from the above test |
| 2 again | invalid | Nothing happens, is still nought’s turn | Nothing happens, is still nought’s turn | Yes | Tests the fact that users cannot overwrite each other’s turn |
| 3, 4, 5, 6, 7 | valid | Forms this pattern:  O X O  X O X  O - -  And O wins | Forms the pattern | Yes | Tests the win state for nought and that the inputs correspond to what is shown on screen, test is linked to previous |
| 1, 2, 3, 4, 5, 9, 8, 7, 6 | valid | Forms this pattern:  O X O  X O O  X O X  And is a draw | Forms the pattern | Yes | Tests the draw state, the test is not linked to previous |
| 9, 1, 2, 3, 4, 5, 6, 7 | valid | Forms this pattern:  X O X  O X O  X – O  And X wins | Forms the pattern | Yes | Tests the X win state, the test is not linked to previous |
| 0 | invalid | Nothing happens | Nothing happens | Yes | Checks invalid input keys |
| W | invalid | Nothing happens | Nothing happens | yes | Checks invalid input keys |
| Esc | valid | 3 second delay then the window closes and reopens in the main menu | Does the expected output | Yes | Tests the escape button that allows the players to quit the game whenever they want |

### Play vs AI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Input Type** | **Expected output** | **Actual output** | **Pass** | **Comments** |
| 1 | Valid | Nought on the top left and AI makes a cross in the center | Forms the related nought in the position of the button but AI will always do a move in the center | Yes | Tests attack pattern 1 i.e. the corner game algorithm.  All the inputs were initial moves. |
| 3 | Nought on the top right and AI makes a cross in the center |
| 7 | Nought on the bottom left and AI makes a cross in the center |
| 9 | Nought on the bottom right and AI makes a cross in the center |
| 5 | Valid | Nought in the center and the AI makes a cross on a corner | The nought is formed in center and the AI does a move in a corner | Yes | Tested multiple times, tests attack pattern 2 |
| 2 | Valid | The AI should random moves | The AI does random moves | Yes | The attack pattern 3 is being tested. The AI should initially do a random move |
| 4 |
| 6 |
| 8 |
| Playing the game | - | The AI should prevent the player from winning | The AI blocks the player’s 2 in a row while trying to form its own 3 in a row.  Starting in a corner or center will always end in a draw. Starting on an edge can allow the AI to win or lose | Yes | Tests the overall AI |
| 0 | Invalid | Nothing happens | Nothing happens | Yes | The invalid keys should not do anything |
| Esc | Valid | 3 second delay and the window is closed and reopened on the main menu | 3 second delay and the window is closed and reopened on the main menu | Yes | The key allows the player to quit whenever they want |
| Closing the window | Valid | 3 second delay and the program is closed | 3 second delay and the program is closed | Yes | Allows the player to quit whenever they want and also close the program when they want |

# Conclusions and Review

### Review

During this project, I was able to learn how to use external libraries for programs, such as the use of the dll files that are used by the executable file or the library and include files used by the solution file that is used by Visual Studio. In my personal statement for my university application, this was one of the things I was interested in finding out and during this project, I was able to find out why the dll files are essential for use of the program. In this project, I was also able to learn how to properly design an interface that people could easily understand, and how a graphical user interface helps achieve this. Another thing I have learnt during this project is how to effectively use algorithms in order to do actions, such ask the AI in this project. The AI was challenging but I was able to manage to do this by also using a work breakdown structure kind of process, as shown in the AI section where the explanation is split into multiple flowcharts to ensure that the implementation of the AI is as simple as possible.

### Conclusion

This project is quite successful since the program created, the noughts and cross game meets both the requirements of the coursework and also the compulsory requirements, the project however did not meet the optional requirements due to the time constraints.

Most of the difficulty in this project was trying to research the SDL components and how to use SDL, this was overcome by the use of careful research and the use of tutorial videos. Other difficulties in the project was the creation of the AI, I overcame this problem by researching the algorithms which are related to noughts and crosses. The use of minimax was initially researched, but instead the video which is linked in the GameAI part of the project was used and I created my own algorithm instead. Creating my own algorithm took quite a long time and was the large portion of the entire project. There is still a weakness in the AI since it is not capable of preventing no win scenarios when a user starts on a non corner edge. However, I am very happy with how the AI turned out since it is certainly capable of defending against a player win and also by being capable of beating the project marker during the demonstrator, even in the AI’s early version (though they would not admit to it, and instead say that they were checking the win state).

# Bibliography

SDL library main webpage <https://www.libsdl.org/>

SDL2 file <https://buildbot.libsdl.org/sdl-builds/sdl-visualstudio/?C=N;O=A>

The basis for the AI algorithm <https://www.youtube.com/watch?v=5DebznNDuxU>

<https://www.youtube.com/watch?v=EiShxMXwogU>

**Tutorials used:**

Installing SDL <https://www.youtube.com/watch?v=FxCC9Ces1Yg>

Opening a window <https://www.youtube.com/watch?v=7hF5BiPhP8Q>

Input processing <https://www.youtube.com/watch?v=Bk74hW9KJmc>