Implementing a program of path planning using D&C convex hull algorithm

To create a program, in C++, that outputs a GUI requires additional libraries in order to get video support. The program was built using SDL, which is a simple direct layer media library that allows C++ to compile code to produce a video window. A text library was imported to display text information on the GUI. Finally, optimizing the textures and surfaces created when dealing with C++. In the application window, the data is displayed in a square called the working-screen, other information regarding the data is found next to the working-screen and a display of coordinates is written below with an indicator for the index.

To begin with creating the D&C convex hull algorithm, A integer that defines the number of random points there are on the working-screen. Then, two variables, type vector int, that holds a value for the global x and y are defined. Next, populate the two variables with either set values or random numbers within a seed based on time within a range of the working-screen. Then, point a, b, and c are defined and inserted. In order, to make things easier, a bubble sort is used on the set of x vectors and the index of y is also changed. This results in a set of two vector points that holds x and y where x is sorted from lowest to highest.

The Convex Hull begins with the upper half of the convex hull going through all points to see if the three points being checked, the anchor point, after-anchor point, and checking point, have a positive or negative slope with the condition function:

((y.at(p2) - y.at(p1)) * (x.at(p3) - x.at(p2)) > ((y.at(p3) - y.at(p2)) * ((x.at(p2) - x.at(p1)))))

If the points meet the condition, the after-anchor point is removed, causing the while-loop to break and a new point is generated to be checked. Else, the next condition:

(x.at(p3) <= x.at(p2) && (y.at(p3) <= y.at(p2)))

Checks the coordinates that may have the same value that messes up the program. This fail-safe makes sure that an incorrect point is not considered. If there is an incorrect point, it is removed, and the while-loop is broken. Finally, the condition:

(x.at(p2) > x.at(p3) && (y.at(p2) <= y.at(p3)) && ((x.at(p2) < x.at(p1)) && y.at(p2) >= y.at(p1)))

Considers that if the point has not been a failure, then to compare if it belongs in the convex hull. If it meets the condition, it means that the point does not meet the conditions of the convex hull and is removed.

Finally, after all the conditions are made and checked, if none pass, the point will be added to the convex hull and it will keep adding until it has searched through all the coordinates.

Next, the lower convex hull is defined similarly to how the upper convex hull is created, where the conditions' y comparisons are reversed, as it is the lower part of the hull and searching for a different slope change. When both convex hulls are created, they are inputted into a separate, vector int type, convex hull which contains all the convex hulls, called TotalConvexHull. Finally, the program draws from line using the indexes defined in the TotalConvexHull.

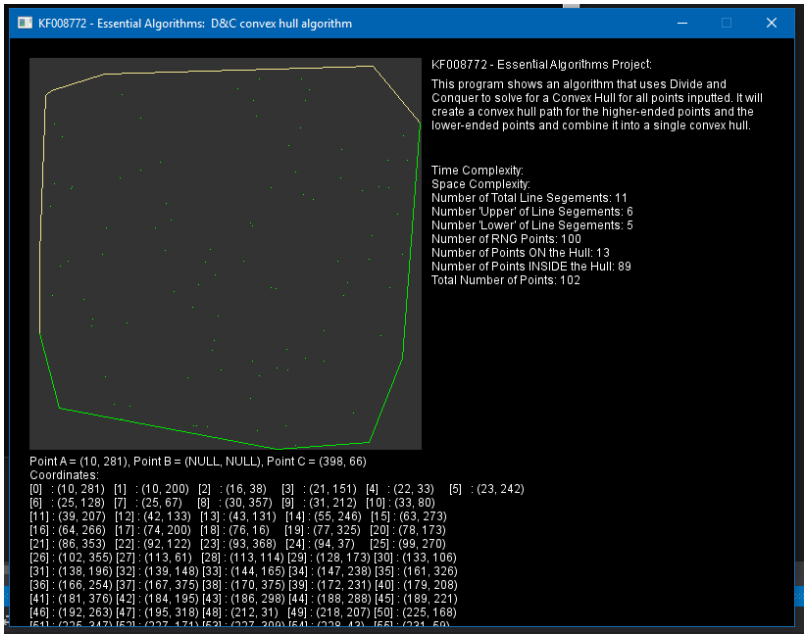

Figure 1: 100 Random Points (A to C)

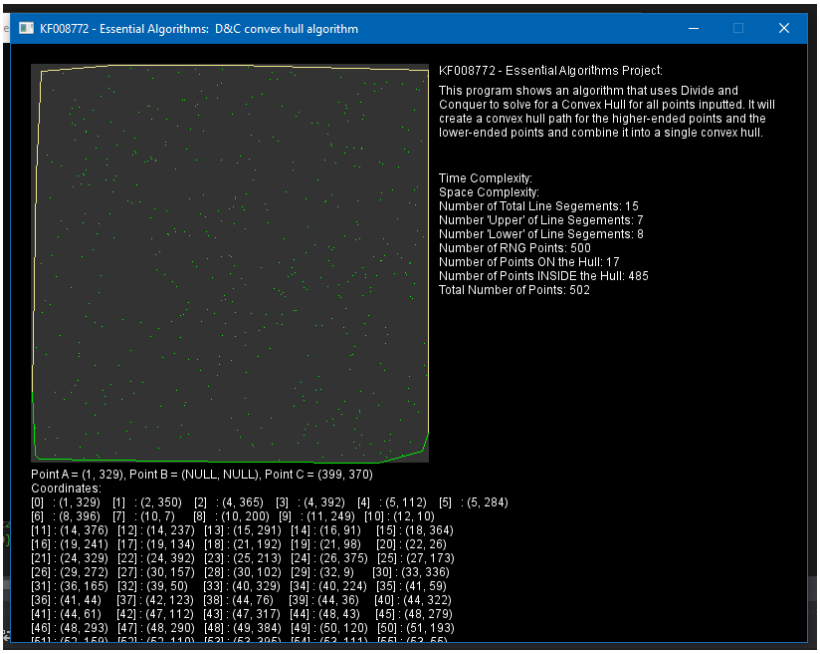

Figure 2 500 Random Points (A to C)

When implementing two different convex hulls in order to check from a to b then from b to c, two extra convex hulls are defined as a vector. The first set of convex hulls will then check up to point B. The first extra convex hull will check from point B and check similarly to the upper convex hull, however, is called RightUpperConvexHull. The second convex hull will check from point B and checks similarly to the lower convex hull, called RightLowerConvexHull. These extra convex hulls will be inserted into the total convex hull.

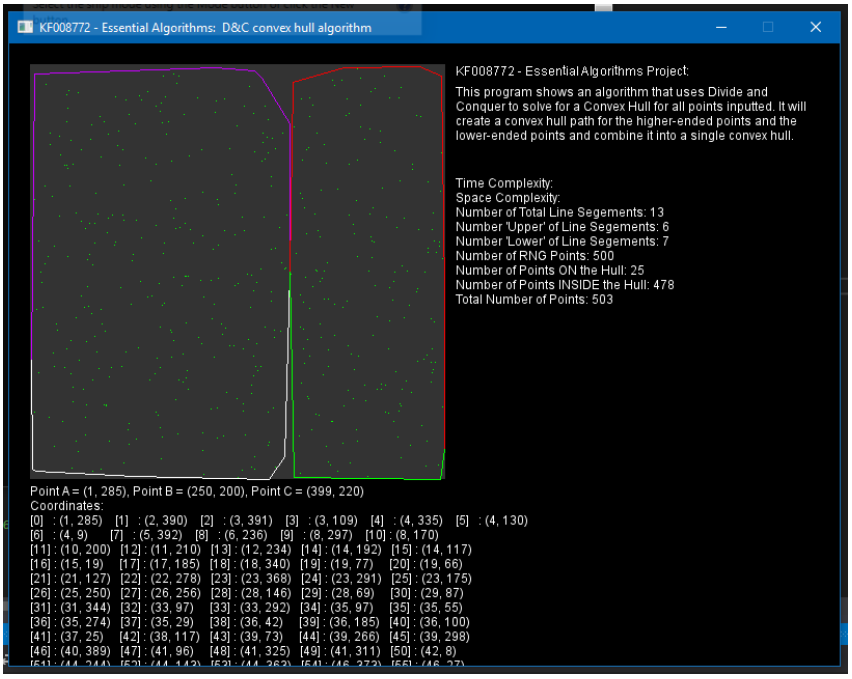

Figure 3 500 Random Points (A to B to C)

Figure 4, below, represents the task 1 where two paths are defined from A to C around an obstacle highlighted in light blue. The two paths represent the convex hull where the yellow is the upper convex hull and the green is the lower convex hull.

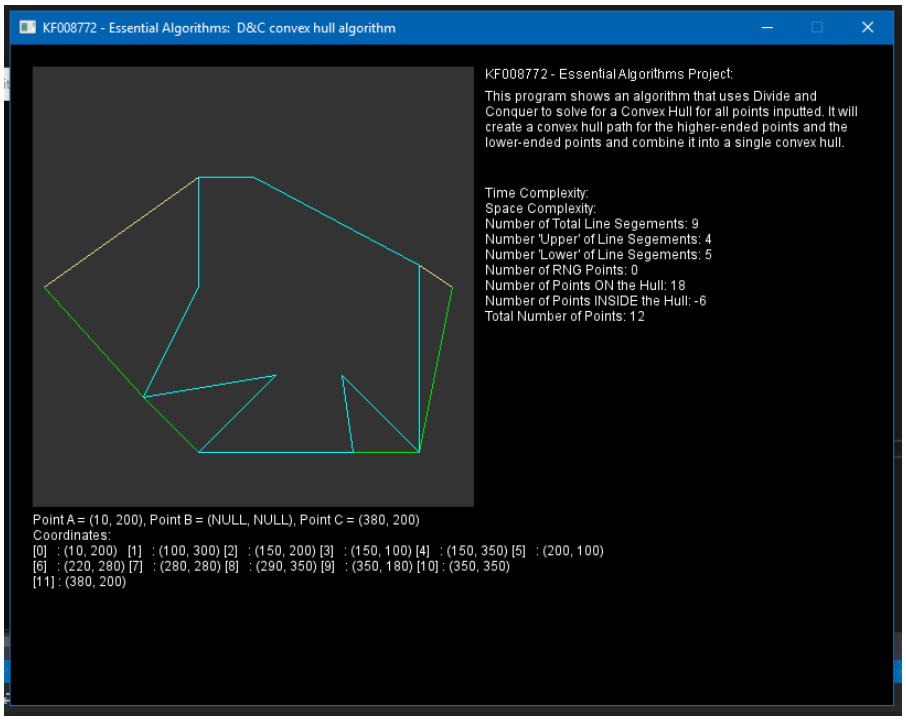

Figure 4 Task 1 (A to C)

Figure 5 represents the task 2 where there are two paths from A to B, where the two paths represent one convex hull. Purple being the Left upper convex hull, white being the left lower convex hull. Next, two paths from B to C, where the two other paths represent another convex hull, red being the right upper convex hull and the green being the right lower convex hull.

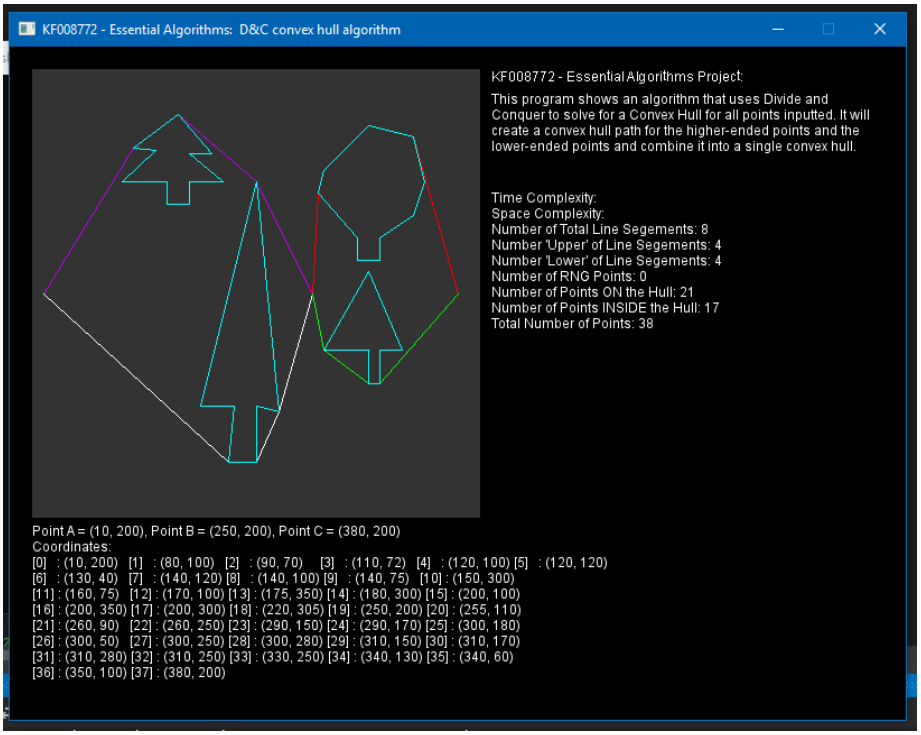

Figure 5 Task 2 (A to B to C)

In conclusion, this program created was a huge inspiration. It is very easily under stable that it can then be used to create image recognition algorithms, path finding algorithms, and much more. Seeing how fast the computer was able to calculate it and seeing the iterations that algorithm went through before the final product was beautiful and neat.

Appendix and Code:

1. Lecture notes
2. MIT OpenCourseWare. (2016). 2. Divide & Conquer: Convex Hull, Median Finding. [Online Video]. 4 March 2016. Available from: https://www.youtube.com/watch?v=EzeYI7p9MjU&t=1464s. [Accessed: 20 January 2019].

My code:

```
/***************************************

KF008772 - Essential Algorithms Project

***************************************/

//C Libaries Included:
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <string>
#include <vector>
#include <algorithm>
#include <utility>

//SDL Libraries Included:
#include <SDL.h>
#include <SDL_ttf.h>


SDL_Window* window = NULL;
SDL_Renderer* renderer = NULL;


/*
* Function to clear the screen and to draw on.
*/
void clear()
{
	SDL_SetRenderDrawColor(renderer, 0x0, 0x0, 0x0, 0xFF);
	SDL_RenderClear(renderer);
```

```
}

/*
* Function to draw the space that the algorithm is working on
*/
void drawWorkingSpace()
{
	SDL_Rect workingspace = { 20, 20, 400, 400 };
	SDL_SetRenderDrawColor(renderer, 0x33, 0x33, 0x33, 0xFF);
	SDL_RenderFillRect(renderer, &workingspace);

}

//CONSTANT integer defining the size of the workspace for the HEIGHT and WIDTH
const int workspaceSize = 400;

//Boolean value used to check changes made in a convex hull to change the colour
bool convexChecker = false;

//LIST OF VECTORS THAT ARE USED WITHIN THE PROGRAM
//Set of x and y that is sorted and searched
std::vector<int> x;
std::vector<int> y;


//vectors holding the X and Y coordinates of the huge obstacle defined in the coursework
std::vector<int> obstacleX;
std::vector<int> obstacleY;


//vectors holding the X and Y coordinates of the unique 3 tree obstacles
```

```
//tree 1
std::vector<int> tree1X;
std::vector<int> tree1Y;
//tree 2
std::vector<int> tree2X;
std::vector<int> tree2Y;
// tree 3
std::vector<int> tree3X;
std::vector<int> tree3Y;

// tree 4
std::vector<int> tree4X;
std::vector<int> tree4Y;



//Convex Hull Vectors:

//VECTOR of the Entire ConvexHull that contains firstly, the Upper Convex Hull and then the Lower Convex Hull
std::vector <int> TotalConvexHull;

//upper convex hull
std::vector<int> UpperConvexHull;

//lower convex hull
std::vector<int> LowerConvexHull;



std::vector <int> RightUpperConvexHull;
std::vector <int> RightLowerConvexHull;
```

```
// 0 - 1000
//!! NUMBER OF POINTS ENTRY. Defining the number of random points that will be inputted into the workspace
int NumberOfPoints = 0;

//Point B index
int pointbIndex = NULL;



//   ____________________________________________________________
//        |                                                    |
//        | MAIN FUNCTION: holding arguments for the video and io |
//        |____________________________________________________________|
//
int main(int argc, char* args[])
{

        //Initializing everything from SDL (video, text)
        int SDL_Init(SDL_INIT_EVERYTHING);

        //Initialize SDL_TTF (text library)
        TTF_Init();

        //Initialize and create the WINDOW with its DETAILS with WINDOW == NULL failsafe
        window = SDL_CreateWindow("KF008772 - Essential Algorithms:  D&C convex hull algorithm", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800, 600, 4);
        if (window == NULL)
        {
                //Return error when window failed to initialize
                std::cout << "Failed to initialize the window!" << std::endl << SDL_GetError() << std::endl;
        }
```

```
//Create a pointer for the SURFACE that will be used on the window
SDL_Surface *screen = SDL_GetWindowSurface(window);

//Create a RENDERER which will draw to the window
renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);

//Create a colour variable for white which will be used for the screen.
Uint32 white = SDL_MapRGB(screen->format, 255, 255, 255);

//Fill the screen with white for default
SDL_FillRect(screen, NULL, white);

//Function to update Window surface
SDL_UpdateWindowSurface(window);

//Set and initialize the boolean variable that will quit the from the MAIN-LOOP
bool quit = false;

//SDL Event intialized which will wait to be called
SDL_Event e;

//TTF_Font initialized with type size
TTF_Font *arial = TTF_OpenFont("res/arial.ttf", 12);

//Font colour
SDL_Color color = { 255, 255, 255, 255 };



//Random gen. based on the computer time at NULLs
srand((unsigned int)time(NULL));
```

```
//Random X and Y input generator
for (int counter = 0; counter < NumberOfPoints; counter++)
{

	//Random x and y variables
	x.push_back(rand() % workspaceSize + 1);
	y.push_back(rand() % workspaceSize + 1);

}

//Set of custom x and y vector inputs:

//set 1
//x = { 100, 150,	200, 205, 250, 300, 350, 380 };
//y = { 110, 300, 200, 260, 90,  240, 300, 105 };

//set 2
//x = { 60, 90,  110, 160, 180, 200, 210, 380 };
//y = { 200, 100, 50, 320, 100,  300, 200, 105 };

//set 3
//x = { 200, 50,  77, 270, 120, 300, 56, 66 };
//y = { 100, 50, 320, 100, 60,  300, 100, 76 };

//rock set of coordinates that uses 10 points
//		1,  2,  3,  4,  5,  6,  7,  8,  9,  10,  11
//x = { 100, 150, 150, 200, 350, 350, 280, 290, 150, 220 };
//y = { 300, 200, 100, 100, 180, 350, 280, 350, 350, 350 };



//Point A
```

```
x.push_back(10);
y.push_back(200);


//Point B - > Hide in comments when not using two seperate convex hulls
int pointBX = 250;
int pointBY = 200;
//Comment the 'push_back' below out for point B
x.push_back(pointBX);
y.push_back(pointBY);

//Point C
x.push_back(380);
y.push_back(200);


//uncomment or comment
//op obstacle that messes with the program
/*obstacleX = { 100, 150, 150, 200, 350, 350, 280, 290, 150, 220};
obstacleY = { 300, 200, 100, 100, 180, 350, 280, 350, 350, 280};*/


//uncomment or comment
//3 little trees
tree1X = { 130, 90, 110, 80, 120, 120, 140, 140, 170, 140, 160};
tree1Y = { 40, 70, 72, 100, 100, 120, 120, 100, 100, 75, 75};

tree2X = { 300, 260, 255, 290, 290, 310, 310, 340, 350, 340 };
tree2Y = { 50, 90, 110, 150, 170, 170, 150, 130, 100, 60 };

tree3X = { 200, 150, 180, 175, 200, 200, 220 };
tree3Y = { 100, 300, 300, 350, 350, 300, 305 };
```

```
tree4X = { 300, 260, 300, 300, 310, 310, 330 };
tree4Y = { 180, 250, 250, 280, 280, 250, 250 };


//Input obstacle into global X and Y
for (int o = 0; o < obstacleX.size(); o++)
{
	x.insert(x.end(), obstacleX.at(o));
	y.insert(y.end(), obstacleY.at(o));
}

//Input tree1 into global X and Y
for (int o = 0; o < tree1X.size(); o++)
{
	x.insert(x.end(), tree1X.at(o));
	y.insert(y.end(), tree1Y.at(o));
}

//Input tree2 into global X and Y
for (int o = 0; o < tree2X.size(); o++)
{
	x.insert(x.end(), tree2X.at(o));
	y.insert(y.end(), tree2Y.at(o));
}
//Input tree3 into global X and Y
for (int o = 0; o < tree3X.size(); o++)
{
	x.insert(x.end(), tree3X.at(o));
	y.insert(y.end(), tree3Y.at(o));
}
```

```
	//Input tree4 into global X and Y
	for (int o = 0; o < tree4X.size(); o++)
	{
		x.insert(x.end(), tree4X.at(o));
		y.insert(y.end(), tree4Y.at(o));
	}

	//Bubble sort
	//Sorting by X coordinate
	for (int i = 0; i < x.size(); i++)
	{
		//inner loop
		for (int j = i + 1; j < x.size(); j++)
		{
			//check
			if (x.at(i) > x.at(j))
			{
				//swap
				std::swap(x[i], x[j]);
				std::swap(y[i], y[j]);
			}
		}
	}



	//
	// LEFT Upper Convex Hull searching
	//
	for (int newcount = 0; newcount < x.size(); newcount++)
	{
		//Marking the index of point B with the last coordinate comparing with the
point at B
```

```
		if(x.at(newcount) == pointBX)
		{
			pointbIndex = newcount;
		}

		//Checking all points up to point B and allowing passage when there is not point B
		if((x.at(newcount) <= pointBX) | pointbIndex == NULL)
		{
			UpperConvexHull.push_back(newcount);
			int size = UpperConvexHull.size();
			bool rightturn = false;

			while ((size > 2) && rightturn == false)
			{

				int p3 = UpperConvexHull.at(size - 1);
				int p2 = UpperConvexHull.at(size - 2);
				int p1 = UpperConvexHull.at(size - 3);

				//Checking the change in slope and removing elements if it fails
				if ((y.at(p2) - y.at(p1)) * (x.at(p3) - x.at(p2)) > ((y.at(p3) - y.at(p2)) * ((x.at(p2) - x.at(p1)))))
				{
					UpperConvexHull.erase(UpperConvexHull.end() - 2);
					size--;
					rightturn = false;
				}

				//Checking and solving for the same coordinates problem
				else if (x.at(p3) <= x.at(p2) && (y.at(p3) <= y.at(p2)))
				{
```

```
                                        UpperConvexHull.erase(UpperConvexHull.end() - 1);
                                        size--;
                                        rightturn = false;
                                }

                                //Checking and solving for the irregular problem
                                else if (x.at(p2) > x.at(p3) && (y.at(p2) <= y.at(p3)) &&
((x.at(p2) < x.at(p1)) && y.at(p2) >= y.at(p1)))
                                {
                                        UpperConvexHull.erase(UpperConvexHull.end() - 1);
                                        size--;
                                        rightturn = false;

                                }

                                //Else continue with adding towards the left upper convex hull
                                else
                                {
                                        rightturn = true;
                                }

                        }
                }



        }


        //
        // LEFT Lower Convex Hull searching
        //
        for (int newcount = 0; newcount < x.size(); newcount++)
```

```
        {
                //Marking the index of point B with the last coordinate comparing with the
point at B
                if (x.at(newcount) == pointBX)
                {
                        pointbIndex = newcount;
                }

                //Checking all points up to point B and allowing passage when there is not
point B
                if((x.at(newcount) <= pointBX) | pointbIndex == NULL)
                {
                        LowerConvexHull.push_back(newcount);
                        int size = LowerConvexHull.size();
                        bool leftturn = false;

                        while ((size > 2) && leftturn == false)
                        {

                                int p3 = LowerConvexHull.at(size - 1);
                                int p2 = LowerConvexHull.at(size - 2);
                                int p1 = LowerConvexHull.at(size - 3);

                                //Checking the change in slope and removing elements if it
fails
                                if ((y.at(p2) - y.at(p1)) * (x.at(p3) - x.at(p2)) < ((y.at(p3) -
y.at(p2)) * ((x.at(p2) - x.at(p1)))))
                                {
                                        LowerConvexHull.erase(LowerConvexHull.end() - 2);
                                        size--;
                                        leftturn = false;
                                }

                                //Checking and solving for the same coordinates problem
```

```
                                else if (x.at(p3) <= x.at(p2) && (y.at(p3) <= y.at(p1)))
                                {
                                        LowerConvexHull.erase(LowerConvexHull.end() - 1);
                                        size--;
                                        leftturn = false;
                                }

                                //Checking and solving for the irregular problem
                                else if(x.at(p2) > x.at(p3) && (y.at(p2) >= y.at(p3)) && ((x.at(p2)
< x.at(p1)) && y.at(p2) <= y.at(p1)))
                                {
                                        LowerConvexHull.erase(LowerConvexHull.end() - 1);
                                        size--;
                                        leftturn = false;
                                }

                                //Else continue with adding towards the left lower convex hull
                                else
                                {
                                        leftturn = true;
                                }

                        }
                }

        }



        //
        // RIGHT Upper Convex Hull searching
        //
        for (int newcount = pointbIndex; newcount < x.size(); newcount++)
        {
```

```
		if (pointbIndex != NULL)
		{
			if (x.at(newcount) >= pointBX)
			{
				RightUpperConvexHull.push_back(newcount);
				int size = RightUpperConvexHull.size();
				bool rightturn = false;

				while ((size > 2) && rightturn == false)
				{
					int p3 = RightUpperConvexHull.at(size - 1);
					int p2 = RightUpperConvexHull.at(size - 2);
					int p1 = RightUpperConvexHull.at(size - 3);

					//Checking the change in slope and removing elements
if it fails
					if ((y.at(p2) - y.at(p1)) * (x.at(p3) - x.at(p2)) > ((y.at(p3) -
y.at(p2)) * ((x.at(p2) - x.at(p1)))))
					{
	RightUpperConvexHull.erase(RightUpperConvexHull.end() - 2);
						size--;
						rightturn = false;
					}

					//Checking and solving for the same coordinates
problem
					else if (x.at(p3) <= x.at(p2) && (y.at(p3) <= y.at(p2)))
					{
	RightUpperConvexHull.erase(RightUpperConvexHull.end() - 1);
						size--;
						rightturn = false;
					}
```

```
                                        //Checking and solving for the irregular problem
                                        else if (x.at(p2) > x.at(p3) && (y.at(p2) <= y.at(p3)) &&
((x.at(p2) < x.at(p1)) && y.at(p2) >= y.at(p1)))
                                        {

        RightUpperConvexHull.erase(RightUpperConvexHull.end() - 1);
                                                size--;
                                                rightturn = false;


                                        }

                                        //Else continue with adding towards the right upper
convex hull
                                        else
                                        {
                                                rightturn = true;
                                        }

                                }
                        }
                }


        }



        //
        // RIGHT Lower Convex Hull searching
        //
        for (int newcount = pointbIndex; newcount < x.size(); newcount++)
        {
                if (pointbIndex != NULL)
```

```
		{
			if (x.at(newcount) >= pointBX)
			{
				RightLowerConvexHull.push_back(newcount);
				int size = RightLowerConvexHull.size();
				bool leftturn = false;

				while ((size > 2) && leftturn == false)
				{

					int p3 = RightLowerConvexHull.at(size - 1);
					int p2 = RightLowerConvexHull.at(size - 2);
					int p1 = RightLowerConvexHull.at(size - 3);

					//Checking the change in slope and removing elements
if it fails
					if ((y.at(p2) - y.at(p1)) * (x.at(p3) - x.at(p2)) < ((y.at(p3) -
y.at(p2)) * ((x.at(p2) - x.at(p1)))))
					{
	RightLowerConvexHull.erase(RightLowerConvexHull.end() - 2);
						size--;
						leftturn = false;
					}

					//Checking and solving for the same coordinates
problem
					else if (x.at(p3) <= x.at(p2) && (y.at(p3) <= y.at(p1)))
					{
	RightLowerConvexHull.erase(RightLowerConvexHull.end() - 1);
						size--;
						leftturn = false;
					}
```

```
                                        //Checking and solving for the irregular problem
                                        else if (x.at(p2) > x.at(p3) && (y.at(p2) >= y.at(p3)) &&
((x.at(p2) < x.at(p1)) && y.at(p2) <= y.at(p1)))
                                        {

        RightLowerConvexHull.erase(RightLowerConvexHull.end() - 1);
                                                size--;
                                                leftturn = false;
                                        }

                                        //Else continue with adding towards the right lower
convex hull
                                        else
                                        {
                                                leftturn = true;
                                        }

                                }
                        }
                }
        }

        //Important Vector function to reverse the Right Lower Convex Hull
        std::reverse(RightLowerConvexHull.begin(), RightLowerConvexHull.end());



        //Fill in the Total Convex Hull with the UPPER convex hull
        for (int i = 0; i < UpperConvexHull.size(); i++)
        {
                TotalConvexHull.push_back(UpperConvexHull.at(i));

        }
```

```
	//Fill in the Total Convex Hull with the LOWER convex hull
	for (int i = 0; i < LowerConvexHull.size(); i++)
	{
		TotalConvexHull.push_back(LowerConvexHull.at(i));
	}


	//Fill in the Total Convex hull with the RIGHT Upper convex jull
	for (int i = 0; i < RightUpperConvexHull.size(); i++)
	{
		TotalConvexHull.push_back(RightUpperConvexHull.at(i));
	}

	//Fill in the Total Convex hull with the RIGHT Lower convex jull
	for (int i = 0; i < RightLowerConvexHull.size(); i++)
	{
		TotalConvexHull.push_back(RightLowerConvexHull.at(i));
	}


	//List of Text displays (Title, About, Data, Coordinates)


	//Text on the right panel with the title of the project
	std::string TitleString = "KF008772 - Essential Algorithms Project: ";

		const char * c0 = TitleString.c_str();

		SDL_Surface *titleMessage = TTF_RenderText_Solid(arial, c0, color);
		SDL_Texture *titleText = SDL_CreateTextureFromSurface(renderer,
titleMessage);
		SDL_Rect titleRect;
		titleRect.x = 430;
```

```
		titleRect.y = 20;
		SDL_QueryTexture(titleText, NULL, NULL, &titleRect.w, &titleRect.h);
		SDL_FreeSurface(titleMessage);
		titleMessage = NULL;


	//Text on the right panel below the title with the About string of the program
	std::string AboutString =
			"This program shows an algorithm that uses Divide and Conquer to solve for a Convex Hull for all points inputted. It will create a convex hull path for the higher-ended points and the lower-ended points and combine it into a single convex hull.";

		const char * c1 = AboutString.c_str();

		SDL_Surface *informationMessage = TTF_RenderText_Blended_Wrapped(arial, c1, color, 350);
		SDL_Texture *informationText = SDL_CreateTextureFromSurface(renderer, informationMessage);
		SDL_Rect informationRect;
		informationRect.x = 430;
		informationRect.y = 40;
		SDL_QueryTexture(informationText, NULL, NULL, &informationRect.w, &informationRect.h);
		SDL_FreeSurface(informationMessage);
		informationMessage = NULL;


	//Text Coordinates string
	std::string coordinatesString;
		std::string coordinatesString2;
		std::string indexS;
		std::string xS;
		std::string yS;
		//For loop to go through all the points
```

```
		for (int i = 0; i < x.size(); i++)
		{
			//Applying the string and parsing the corresponding integers
			indexS = std::to_string(i);
			xS = std::to_string(x.at(i));
			yS = std::to_string(y.at(i));

			//Put index into a nice box parentheses
			coordinatesString2.append("[" + indexS + "]");

			//If index is less than 2, input another space for formatting
			if (indexS.size() == 1)
			{
				coordinatesString2.append("  ");
			}

			//Put coordinates into a nice parentheses
			coordinatesString2.append(" : (" + xS + ", " + yS + ") ");

			//if x coordinate size is less than 3, add spaces for number less than 3 for formatting
			if (xS.size() < 3)
			{
				for (int k = xS.size(); k <= 3; k++)
				{
					coordinatesString2.append(" ");
				}

			}
			//if y coordinate size is less than 3, add spaces for number less than 3 for formatting
			if (yS.size() < 3)
			{
```

```
                for (int k = yS.size(); k <= 3; k++)
                {
                        coordinatesString2.append(" ");

                }

        }

        //Skip every 5 coordinates
        if ((i % 5) == 0 && (i != 0))
        {
                coordinatesString2.append("\n");
        }

}

//Point A 'x and y' String
std::string pointAXs = std::to_string(x.at(0));
std::string pointAYs = std::to_string(y.at(0));
std::string pointAS = "(" + pointAXs + ", " + pointAYs + ")";

//Point B 'x and y' String
std::string pointBXs = std::to_string(x.at(pointbIndex));
std::string pointBYs = std::to_string(y.at(pointbIndex));

//Output null when B is null
if (pointbIndex == NULL)
{
        pointBXs = pointBYs = "NULL";
}

std::string pointBS = "(" + pointBXs + ", " + pointBYs + ")";
```

```
		//Point C 'x and y' String
		std::string pointCXs = std::to_string(x.at(x.size() - 1));
		std::string pointCYs = std::to_string(y.at(y.size() - 1));
		std::string pointCS = "(" + pointCXs + ", " + pointCYs + ")";

		//Points A and B and C
		std::string coordinatesString3 = "Point A = " + pointAS + ", Point B = " + pointBS + ", Point C = " + pointCS;

		//All the coordinates strings together
		coordinatesString = coordinatesString3 + "\nCoordinates: \n" + coordinatesString2;

		const char * c2 = coordinatesString.c_str();

		SDL_Surface *coordinatesMessage = TTF_RenderText_Blended_Wrapped(arial, c2, color, 700);
		SDL_Texture *coordinatesText = SDL_CreateTextureFromSurface(renderer, coordinatesMessage);
		SDL_Rect coordinatesRect;
		coordinatesRect.x = 20;
		coordinatesRect.y = 425;
		SDL_QueryTexture(coordinatesText, NULL, NULL, &coordinatesRect.w, &coordinatesRect.h);
		SDL_FreeSurface(coordinatesMessage);
		coordinatesMessage = NULL;


	//Text DATA string
	std::string dataString;

		//Time complexity string
```

```
		std::string tmString = "\nTime Complexity: ";


		//Space complexity string
		std::string smString = "\nSpace Complexity: ";


		//Number of Line Segements (Total, Upper, Lower)
		int NumLineSegements = UpperConvexHull.size() + LowerConvexHull.size() -
2;
		int UpperConvexLineSeg = UpperConvexHull.size() - 1;
		int LowerConvexLineSeg = LowerConvexHull.size() - 1;
		std::string lineSegementsString = "\nNumber of Total Line Segements: " +
std::to_string(NumLineSegements);
		std::string indiviConvexLineSegString = "\nNumber 'Upper' of Line
Segements: " + std::to_string(UpperConvexLineSeg) + "\nNumber 'Lower' of Line
Segements: " + std::to_string(LowerConvexLineSeg);


		//Number of points (RNG, Total, ON, INSIDE)
		int outsideNum = TotalConvexHull.size();
		int insideNum = x.size() - TotalConvexHull.size();
		std::string numPointsString = "\nNumber of RNG Points: " +
std::to_string(NumberOfPoints);
		std::string outsideNumString = "\nNumber of Points ON the Hull: " +
std::to_string(outsideNum);
		std::string insideNumString = "\nNumber of Points INSIDE the Hull: " +
std::to_string(insideNum);
		std::string totalNumString = "\nTotal Number of Points: " +
std::to_string(x.size());


		//Data string with all the strings (TM, SM, NumLines, UpperLines, LowerLines,
RNG, ON, inside, Total)
		dataString = "\n" + tmString + smString + lineSegementsString +
indiviConvexLineSegString + numPointsString + outsideNumString + insideNumString +
totalNumString;


		const char * c3 = dataString.c_str();
```

```
		SDL_Surface *lineSegmentsMessage = TTF_RenderText_Blended_Wrapped(arial, c3, color, 350);
		SDL_Texture *lineSegementsText = SDL_CreateTextureFromSurface(renderer, lineSegmentsMessage);
		SDL_Rect lineSegementsRect;
		lineSegementsRect.x = 430;
		lineSegementsRect.y = 100;
		SDL_QueryTexture(lineSegementsText, NULL, NULL, &lineSegementsRect.w, &lineSegementsRect.h);
		SDL_FreeSurface(lineSegmentsMessage);
		lineSegmentsMessage = NULL;



	/****************************************
	*                                       *
	*	!MAIN LOOP WITHIN THE MAIN FUNCTION! *
	*                                       *
	****************************************/
	while (!quit)
	{

		//Clear Function to defaults.
		clear();

		//While loop that will poll for events while the loop runs
		while (SDL_PollEvent(&e) != 0)
		{
			//Function to quit the window and quit
			if (e.type == SDL_QUIT)
			{
				quit = true;
				break;
```

```
		}
	}

	//Function to draw the working space
	drawWorkingSpace();



	//Text rendering:

	// TEXT - Title being shown
	SDL_RenderCopy(renderer, titleText, NULL, &titleRect);

	//TEXT - data being shown
	SDL_RenderCopy(renderer, informationText, NULL, &informationRect);

	//TEXT - coordinates shown
	SDL_RenderCopy(renderer, coordinatesText, NULL, &coordinatesRect);

	//TEXT - line segements shown
	SDL_RenderCopy(renderer, lineSegementsText, NULL,
&lineSegementsRect);

	//Loop to draw all the points
	for (int j = 0; j < x.size(); j++)
	{
		SDL_SetRenderDrawColor(renderer, 0, 255, 0, 255);



		SDL_RenderDrawPoint(renderer, x.at(j) + 20, y.at(j) + 20);
	}
```

```
		//draw the upper convex hull
		SDL_SetRenderDrawColor(renderer, 240, 230, 140, 255);
		for (int d = 0; d < UpperConvexHull.size() - 1; d++)
		{
			if (convexChecker == true)
			{

				SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
			}

			SDL_RenderDrawLine(renderer, x.at(UpperConvexHull.at(d)) + 20,
y.at(UpperConvexHull.at(d)) + 20, x.at(UpperConvexHull.at(d + 1)) + 20,
y.at(UpperConvexHull.at(d + 1)) + 20);
		}



		//draw the lower convex hull
		SDL_SetRenderDrawColor(renderer, 0, 230, 0, 255);
		for (int d = 0; d < LowerConvexHull.size() - 1; d++)
		{
			if (convexChecker == true)
			{

				SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
			}


			SDL_RenderDrawLine(renderer, x.at(LowerConvexHull.at(d)) + 20,
y.at(LowerConvexHull.at(d)) + 20, x.at(LowerConvexHull.at(d + 1)) + 20,
y.at(LowerConvexHull.at(d + 1)) + 20);
		}

		int upperconvexsizemin1 = UpperConvexHull.size() - 1;
```

```
		int lowerconvexcheck = upperconvexsizemin1 + LowerConvexHull.size();
		int newconvexcheck = lowerconvexcheck + RightUpperConvexHull.size();

		//draw all the convex hull
		for (int convexCounter = 0; convexCounter < TotalConvexHull.size() - 1; convexCounter++)
		{
			if (pointbIndex != NULL)
			{
				SDL_SetRenderDrawColor(renderer, 200, 0, 255, 255);

				//used to skip last point of upper convex hull being removed to prevent the start of the lower convex to connect to the last of the upper convex
				if (convexCounter == UpperConvexHull.size() - 1)
				{
					convexCounter++;
				}

				if (convexCounter >= upperconvexsizemin1)
				{
					SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
				}

				if (convexCounter > lowerconvexcheck)
				{
					SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
				}

				if (convexCounter > newconvexcheck)
				{
					SDL_SetRenderDrawColor(renderer, 0, 255, 0, 255);
				}
```

```
                int convexCounterTwo = convexCounter + 1;


                SDL_RenderDrawLine(renderer,
x.at(TotalConvexHull.at(convexCounter)) + 20, y.at(TotalConvexHull.at(convexCounter)) +
20, x.at(TotalConvexHull.at(convexCounterTwo)) + 20,
y.at(TotalConvexHull.at(convexCounterTwo)) + 20);
            }

        }

        //draw obstacle
        for (int k = 1; k < obstacleX.size(); k++)
        {
            SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
            SDL_RenderDrawPoint(renderer, obstacleX.at(k) + 20, obstacleY.at(k)
+ 20);

            SDL_SetRenderDrawColor(renderer, 0, 255, 255, 255);
            SDL_RenderDrawLine(renderer, obstacleX.at(k) + 20, obstacleY.at(k)
+ 20, obstacleX.at(k - 1) + 20, obstacleY.at(k - 1) + 20);

            if (k + 1 == obstacleX.size())
            {

                SDL_RenderDrawLine(renderer, obstacleX.at(0) + 20,
obstacleY.at(0) + 20, obstacleX.at(k) + 20, obstacleY.at(k) + 20);

            }

        }

        //draw tree 1
```

```
		for (int k = 1; k < tree1X.size(); k++)
		{
			SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
			SDL_RenderDrawPoint(renderer, tree1X.at(k) + 20, tree1Y.at(k) + 20);

			SDL_SetRenderDrawColor(renderer, 0, 255, 255, 255);
			SDL_RenderDrawLine(renderer, tree1X.at(k) + 20, tree1Y.at(k) + 20,
tree1X.at(k - 1) + 20, tree1Y.at(k - 1) + 20);

			if (k + 1 == tree1X.size())
			{

				SDL_RenderDrawLine(renderer, tree1X.at(0) + 20,
tree1Y.at(0) + 20, tree1X.at(k) + 20, tree1Y.at(k) + 20);

			}

		}

		//draw tree 2
		for (int k = 1; k < tree2X.size(); k++)
		{
			SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
			SDL_RenderDrawPoint(renderer, tree2X.at(k) + 20, tree2Y.at(k) + 20);

			SDL_SetRenderDrawColor(renderer, 0, 255, 255, 255);
			SDL_RenderDrawLine(renderer, tree2X.at(k) + 20, tree2Y.at(k) + 20,
tree2X.at(k - 1) + 20, tree2Y.at(k - 1) + 20);

			if (k + 1 == tree2X.size())
			{

				SDL_RenderDrawLine(renderer, tree2X.at(0) + 20,
tree2Y.at(0) + 20, tree2X.at(k) + 20, tree2Y.at(k) + 20);
```

```
		}

	}

	//draw tree 3
	for (int k = 1; k < tree3X.size(); k++)
	{
		SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
		SDL_RenderDrawPoint(renderer, tree3X.at(k) + 20, tree3Y.at(k) + 20);

		SDL_SetRenderDrawColor(renderer, 0, 255, 255, 255);
		SDL_RenderDrawLine(renderer, tree3X.at(k) + 20, tree3Y.at(k) + 20,
tree3X.at(k - 1) + 20, tree3Y.at(k - 1) + 20);

		if (k + 1 == tree3X.size())
		{

			SDL_RenderDrawLine(renderer, tree3X.at(0) + 20,
tree3Y.at(0) + 20, tree3X.at(k) + 20, tree3Y.at(k) + 20);

		}

	}

	//draw tree 4
	for (int k = 1; k < tree4X.size(); k++)
	{
		SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
		SDL_RenderDrawPoint(renderer, tree4X.at(k) + 20, tree4Y.at(k) + 20);

		SDL_SetRenderDrawColor(renderer, 0, 255, 255, 255);
```

```
                        SDL_RenderDrawLine(renderer, tree4X.at(k) + 20, tree4Y.at(k) + 20,
tree4X.at(k - 1) + 20, tree4Y.at(k - 1) + 20);

                        if (k + 1 == tree4X.size())
                        {

                                SDL_RenderDrawLine(renderer, tree4X.at(0) + 20,
tree4Y.at(0) + 20, tree4X.at(k) + 20, tree4Y.at(k) + 20);

                        }

                }

                //Present what to draw
                SDL_RenderPresent(renderer);

        }

        //Destroy memory in renderer
        SDL_DestroyRenderer(renderer);
        renderer = NULL;

        //Destroy memory in window
        SDL_DestroyWindow(window);
        window = NULL;

        //Close and quit ttf
        TTF_CloseFont;
        TTF_Quit();

        //Quit sdl
        SDL_Quit();
```

```
        //return 0
        return 0;

}
```