Design and Implementation of the Gobang Game

by Sif

20 December 2023

Contents

1 Abstract

This paper describes the process of creating an MFC (Microsoft Foundation Class) project. The paper outlines the key components and features of the MFC framework used in the project. It aims to provide a comprehensive understanding of the MFC project development process and serve as a guide for developers interested in creating their own MFC applications.

2 Introduction

The MFC application will cover the key features of an average application, this will include the following:

- The main game window.
- A menu.
- A Toolbar
- A Statusbar

Along with those will include functionality for modifying these key features. Below is an image of the final look for the application.

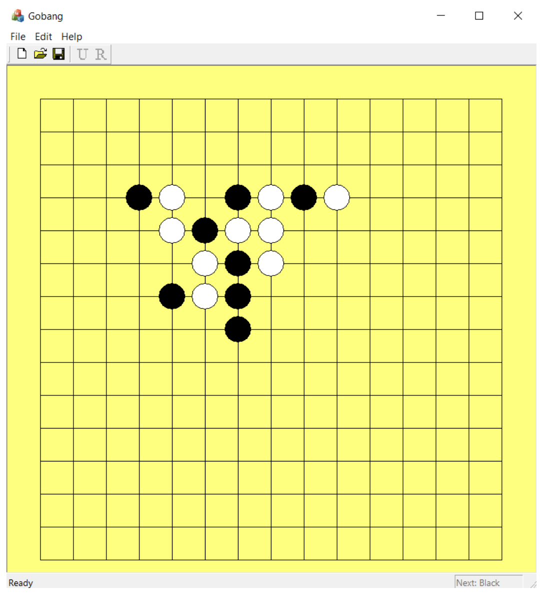


Figure 1: Gobang game

This paper will go into more depth as to what and how the functionality is implemented.

The project structure is Single-Document Interface(SDI) oriented which is an application architecture where only one document can be open at a time. It provides a simple and focused building experience, making it suitable for applications that don't require multiple document handling such as this project.

3 Board Grid

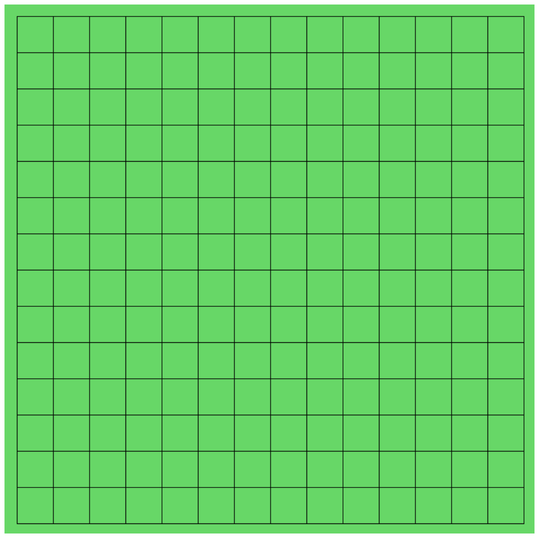


Figure 2: Gobang Grid

The board was created by drawing a simple grid with a for loop that would repeatedly draw a square of a given size N times(in our case 14 times) in the OnPaint message handler. The code is as follows:

```
for (int row = 1; row <= 14; row++) {
    for (int col = 1; col <= 14; col++) {
        int x = col * squareSize;
        int y = row * squareSize;

        dc.MoveTo(x, y);
        dc.LineTo(x + squareSize, y);
        dc.LineTo(x + squareSize, y + squareSize);
        dc.LineTo(x, y + squareSize);
        dc.LineTo(x, y);

    }
}
```

3.1 Hit Box Check

The Hit box is the most crucial feature of this project and does not actually follow the initial grid built previously. The visible board is only made for visual purposes it does not serve any actual functionality.

The hit box is an invisible grid that checks whether the player has clicked within it and if so it will place a piece based on whose turn it currently is. Here is a visual representation of the hit box grid:

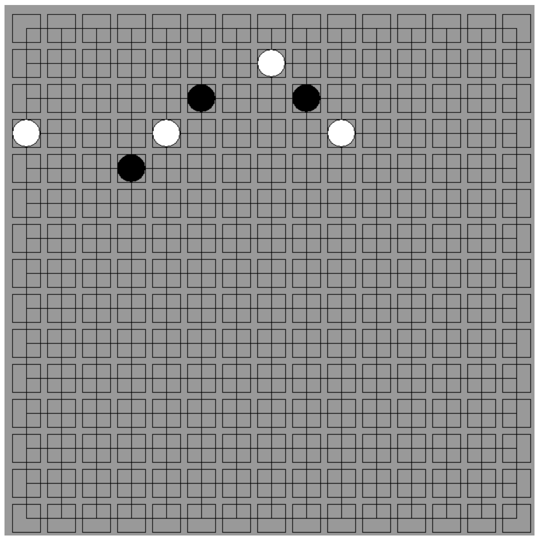


Figure 3: Hitbox Grid

The hit boxes are made to be smaller than the board squares to prevent the player from accidentally placing a piece in an undesired spot when clicking near the middle of a board square.

Here is the code written for the hit box grid(This code does not include the code for the visual representation):

```
for (int row = 1; row <= 15; row++) {
    for (int col = 1; col <= 15; col++) {
        int x = col * squareSize;
        int y = row * squareSize;

        if (point.x >= x - collider &&
            point.x <= x + collider &&
            point.y >= y - collider &&
            point.y <= y + collider &&
            board[row - 1][col - 1] == 0) {

            pDC->Ellipse(CRect(x - collider, y - collider,
                               x + collider, y + collider));
            if (m_white == 1) board[row - 1][col - 1] = 1;
            else board[row - 1][col - 1] = -1;

            if (board[row - 1][col - 1] != 0)
                if (winCheck(row, col))
                        (m_white == 1) ? MessageBox(_T("white wins")) :
                                         MessageBox(_T("black wins"));

            m_white *= -1;

            CPoint point = { row - 1, col - 1 };
            m_moveList.push(point);
        }
    }
}
```

The hit box check is being done during the LButtonDown() function meaning that everytime the player left clicks on the client window it will check whether the click occurred within the hit box area, if it did then it will check whether that area is empty with board[row - 1][col - 1] == 0; the actual values are stored within a 2D array called 'board' which has values of 1, 0, or -1 (1 = white, 0 = empty, -1 = black).

After all that has been checked and confirmed to be an viable move it will draw a circular piece using the Ellipse() function of MFC, the colour of the piece will be decided with this piece of code:

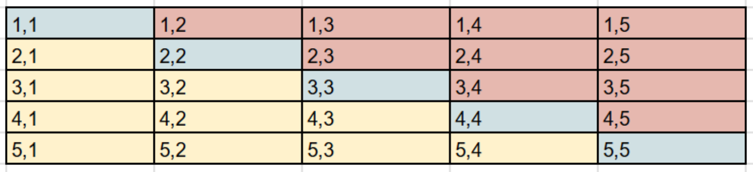
```
CBrush brush;
brush.CreateSolidBrush((m_white == 1) ? RGB(255, 255, 255) : RGB(0, 0, 0));
pDC->SelectObject(brush);
```

The turn is decided with the variable m_white which is either 1 for white or -1 for black, after each turn it is multiplied by -1 to switch between both values. The values 1 and -1 are also stored within the board array after being placed.

3.2 Win Check

After a piece has been placed we check if the last move resulted in a win. This is done by checking all the pieces in the row, column, and the diagonals based on the last piece that was placed. This is done in a rather long piece of the code, the row and columns are quite straightforward as we simply do a loop over one column or row to check if there are 5 pieces with the same colour in continuity. The diagonal checks are more complicated because of how 2D arrays are structured, the code required two separate parts for the top left to bottom right and top right to bottom left.

3.2.1 Top left to Bottom Right



1,1	1,2	1,3	1,4	1,5
2,1	2,2	2,3	2,4	2,5
3,1	3,2	3,3	3,4	3,5
4,1	4,2	4,3	4,4	4,5
5,1	5,2	5,3	5,4	5,5

Figure 4: Example array

Above is shown an example of how the array is handled for top left to bottom right check; the array is divided into three different checks. Using a variable called 'divider' we divide the loop into three parts, below shows the first:

```
int divider = row - col;
if (divider > 0)
    for (int i = 0;
        (row - 1) - (col - 1) + i < sizeof(board) / sizeof(*board); i++)
        if (board[(row - 1) - (col - 1) + i][i] == control)
```

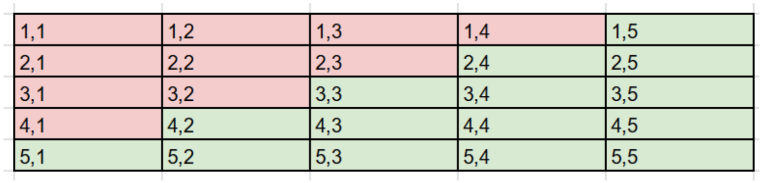
This code checks for yellow section of the array shown in Fig.4, it takes the position of the last placed piece in the array and subtracts the value for the row with the column, if it returns a value below zero that would mean that the last piece was placed in the lower left section of the board. Afterwards we loop through from the top left most element to the right bottom most checking for a win.

```
else if (divider < 0)
    for (int i = 0;
        (col - 1) - (row - 1) + i < sizeof(board) / sizeof(*board); i++)
        if (board[i][(col - 1) - (row - 1) + i] == control)
```

Above is the code that checks for the red section of the array(refer back to Fig.4); the divider returns a value of over 0. Lastly below is the code for the middle section of the board:

```
else
    for (int i = 0; i < sizeof(board) / sizeof(*board); i++)
        if (board[i][i] == control)
```

3.2.2 Top Right to Bottom Left



1,1	1,2	1,3	1,4	1,5
2,1	2,2	2,3	2,4	2,5
3,1	3,2	3,3	3,4	3,5
4,1	4,2	4,3	4,4	4,5
5,1	5,2	5,3	5,4	5,5

Figure 5: Example array

Above shows the structure used to separate the win check method for looping from top right to bottom left. Unlike the previous method we can separate the board into two parts instead of three.

```
if ((row)+(col) < (sizeof(board) / sizeof(*board)) + 1)
    for (int i = 0; i < col + row - 1; i++)
        if (board[i][(col - 1 + row - 1) - i] == control)
```

The code above shows the loop for the red section in Fig.5 which is checked by calculating the sum of the coordinates of the last placed piece; if the sum is less than the size of the board (if the board is NxN then the size is N) then the last piece was placed in the red section. The code then loops through from the top right to the bottom left to check for a win condition. Below is the code that will handle the else case meaning the green section:

```
else
    for (int i = 0; i <= deltaX; i++)
        if (board[row - 1 - deltaY + i][col - 1 + deltaY - i] == control)
```

4 Menu

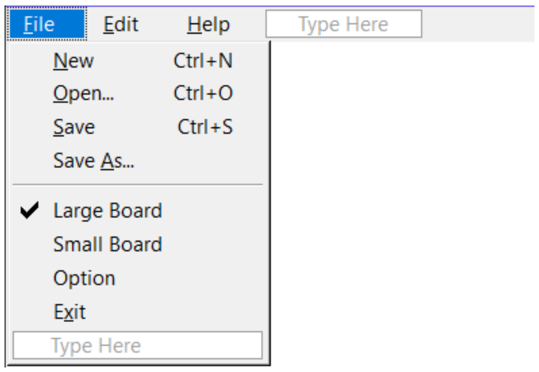


Figure 6: Menu

The menu was designed using the Visual Studio edit interface, each menu item has it's own ID and accelerator in the accelerator table as well if needed. The click events are all handled with the ON_COMMAND() message handler and custom functions for each item.

Here is the code to load the accelerators for the menu items:

```
m_hAccelTable = ::LoadAccelerators(AfxGetInstanceHandle(),
        MAKEINTRESOURCE(IDR_MENU));
```

Clicking on New will reset the board with the InitialiseBoard() function and certain other variables. The Open, Save, and Save as will be discussed in the Save states section of the paper. Large board and Small board simply change the value of the square size and collision size to a smaller value and redraw the board afterwards, the check mark is handled with the ON_UPDATE_COMMAND_UI() message handler. The Options opens a dialog box that handles certain settings. The Exit will send a confirmation message if the game is unsaved, otherwise it will close the application.

The Edit menu includes the items Undo and Redo which make use of the stack library in C++ in order to take advantage of the top, pop, and push function that allows us to get previously done actions easily, below is the code for both functions:

```
void CMainFrame::OnEditUndo()
{
    if (!m_moveList.empty())
    {
        board[m_moveList.top().x][m_moveList.top().y] = 0;
        m_redoList.push(m_moveList.top());
        m_moveList.pop();

        m_white *= -1;

        UpdateTurnStatus();
        Invalidate();
    }
}


void CMainFrame::OnEditRedo()
{
    if (!m_redoList.empty())
    {
        board[m_redoList.top().x][m_redoList.top().y] = m_white;
        m_moveList.push(m_redoList.top());
        m_redoList.pop();

        m_white *= -1;

        UpdateTurnStatus();
        Invalidate();
    }
}
```

The Help menu includes the About item which leads to a dialogue box with information about the application.

4.1 Options Dialog Box

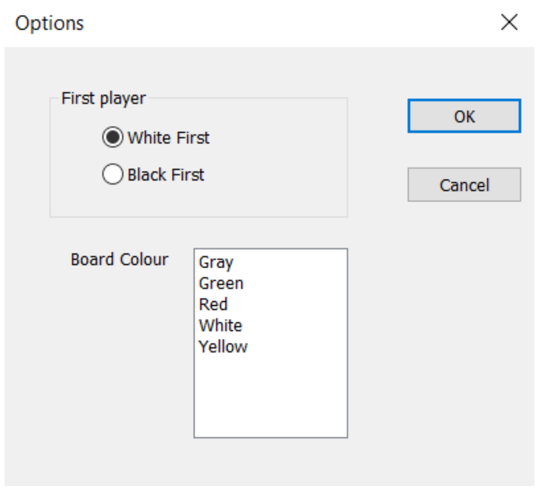


Figure 7: Options dialog

The options window consists of two key components which are the radio buttons for changing which side plays first and a list box for changing the background colour. The variables are transferred via DoDataExchange(). The code below shows how the dialog box is called:

```
Options dlg(m_firstPlayer, m_boardColour);
if (dlg.DoModal() == IDOK)
{
    m_firstPlayer = dlg.GetTurn();
    m_boardColour = dlg.GetBoardColour();
    isModified = true;
}
```

The window is called with the DoModal() function which after the user presses on OK will lead to transferring the values of the chosen variables from the Options window to the actual view. The window is Invalidated afterwards to update the view.

5 Toolbar and Statusbar

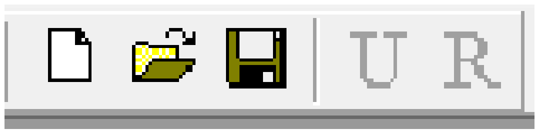


Figure 8: Toolbar

The toolbar as shown above consists of five features: New, Open, Save, Undo, and Redo. The design was done in the Visual Studio editor interface and the ID given were the same as those from the menu which means that these buttons use the same code as their menu counterparts. The toolbar is also dockable which means that it can be moved and placed elsewhere, here is the code that allows it to be dockable:

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockPane(&m_wndToolBar);
```

Figure 9: Toolbar

Above is an image of the statusbar. It was made with the CMFCStatusBar class and included only two indicators; one for the showing the hovered item and status and a second for showing whose turn it is. The status was done automatically by the wizard, all that was needed was to add the prompts to each menu and toolbar item. The turn display was change using a function named UpdateTurnStatus() which switched the text with this code:

```
CString turn = (m_white == 1) ? _T("Next: White") : _T("Next: Black");
m_wndStatusBar.SetPaneText(1, turn);
```

6 Save State

The save function used the CFileDialog for the Save As function and the CArchive in order to serialise ther data and store it into the created file(using the extension .sav). Below is the code for the filter and dialog box:

```
LPCTSTR pszFilter = _T("Game State (*.sav)|*.sav|");

CFileDialog dlg(FALSE, _T("gobang"), NULL,
    OFN_OVERWRITEPROMPT, pszFilter);
```

After setting the filter and creating the dialog box we call it with DoModal() and then get the file path with GetPathName(). With a newly create it file we open it in Create and Write mode and create a CArchive variable as shown below:

```
if (file.Open(filePath, CFile::modeCreate | CFile::modeWrite))
    CArchive ar(&file, CArchive::store);
```

The data that needs to be store is then stored with the CArchive variable eg. ar <<m_boardColour.

The Save function is similar to the Save As, the only difference is that it does not call a dialog box and directly saves to the file path saved in the m_filepath variable.

The Open function works the same way as the Save As function but backwards, instead of using this: ar <<m_boardColour. It will use this ar >>m_boardColour. After opening a file the window will also Invalidate() to update it.

7 Conclusion

The MFC library is a very useful tool that still allows for a very customisable and controlled workflow. It is easier to use than Win32 but gives as much freedom for the developer. Though there were many complicacies during the development it is overall a very versatile and straightforward tool to use. With the help of the Visual Studio editor interface and wizard it has become a lot easier to create complex application in shorter amounts of time.

References

[1] TylerMSFT. “MFC Desktop Applications.” Microsoft Learn, learn.microsoft.com/en-us/cpp/mfc/mfc-desktop-applications?view=msvc-170. Accessed 6 Jan. 2024.

[2] TylerMSFT. “Creating an MFC Application.” Microsoft Learn, learn.microsoft.com/en-us/cpp/mfc/reference/creating-an-mfc-application?view=msvc-170. Accessed 6 Jan. 2024.

[3] “MFC - Getting Started.” Tutorialspoint, www.tutorialspoint.com/mfc/mfc_getting_started.htm. Accessed 6 Jan. 2024.

[4] Archiveddocs. “Learning to Program with MFC.” Microsoft Learn, learn.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-6.0/aa279433(v=vs.60)?redirectedfrom=MSDN. Accessed 6 Jan. 2024.

[5] “SDI and MDI.” Microsoft Learn, learn.microsoft.com/en-us/previous-versions/b2kye6c4(v=vs.140). Accessed 6 Jan. 2024.