EE 356
Notes on Minesweeper

There are two ways (at least) to write the Minesweeper game in C# WPF. The first method uses buttons as cells and the second uses only graphics on a bitmap. The discussion below gives an overview with some sample code of how to do each of these.

Method 1 – Buttons
Minesweeper presents a grid of cells. These cells can be buttons. Since a reasonable sized grid is 9 x 9 or 16 x 16 you will need to create the buttons dynamically in code. Each button will also need a click event but fortunately, we can use a single event that is triggered when any button is clicked. The click event receives an argument specifying the source which can tell us which button was clicked.

For the xaml code you only need a Window and a Grid. These need to be big enough to accommodate your array of buttons. As an example, for a button size of 20 x 20 and an array that is 16 x 16 we will need a window that is at least 320 x 320 just for the array of buttons. It will need to be larger than this to accommodate other items such as text boxes, menus, etc. The code in the .cs file will generate the buttons. Here is some sample code which generates an array of 16 x 16 buttons where each button is 20 x 20.

```
1   for(r=0;r<16;r++)
2       {for(c=0;c<16;c++)
3          {Button btn = new Button();
4           btn.VerticalAlignment = VerticalAlignment.Top;
5           btn.HorizontalAlignment = HorizontalAlignment.Left;
6           btn.Content = "";
7           btn.Width = 20;
8           btn.Height = 20;
9           btn.Margin = new Thickness(10+20*c,20*r,0,0);
10          btnName = "btn" + (c+16*r).ToString();
11          btn.Name = btnName;
12          btn.Background = Brushes.LightGray;
13          btn.Click += new RoutedEventHandler(button_Click);
14          btnList.Add(btn);
15          grd1.Children.Add(btn);
16         }
17      }
```

Figure 1
Code fragment for generating 256 buttons.

For every button we want to create we need to declare a new button as on line 3. To set the button position we need to specify the alignment (lines 4 and 5) and the width and height (lines 7 and 8). To make the buttons touch each other in an array we set the margin ad 20 times the row and column number plus and offset as in line 9. The margin is set from the top and left side because of the alignment statements.

Each button is given a unique name which is derived from its row and column number. This is done in lines 10 and 11. The name assigned is "btnxxx" where xxx is a number which is 16

times the row plus the column. We can later pull this name apart and find the row an column number.

Line 13 adds an event handler for the button. "button_Click" is the name of this handler. All buttons use the same event handler.

Line 15 makes the button a child of the grid which must be named "grd1" in the xaml code.

Line 14 adds each button to a List which was created as a private member of the class:

```
private List <Button> btnList = new List<Button>();
```

For this arrangement the index into the list corresponds to the number that was assigned as part of the button name. For example we will find btn87 at `btnList[87]`. We can get the row and column location but deleting btn from the string button name and converting the remaining number to an int. The row will be the index/16 and the column will be the index mod 16. If you run this code you should get something like that shown in Figure 2.

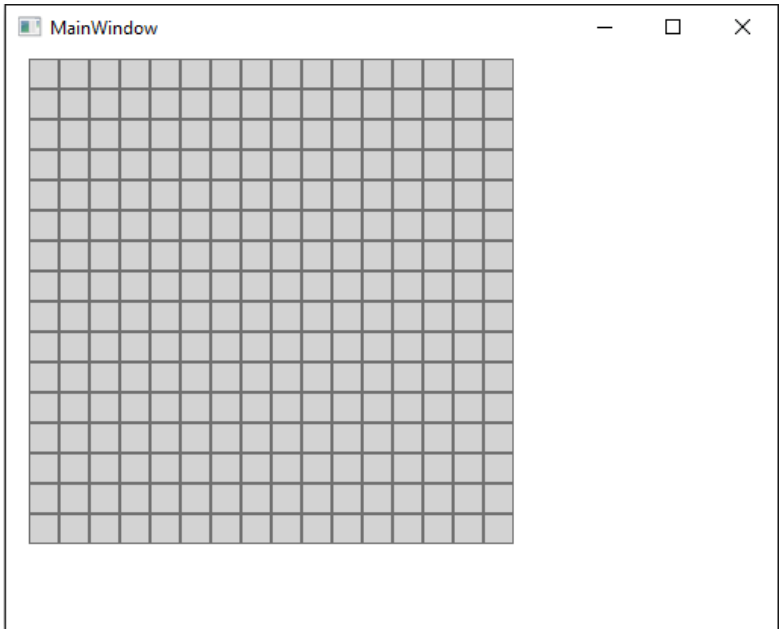

Figure 2
A grid of 16 x 16 buttons where each button is 20 x 20 pixels.

Before you click on a button you will have to write a click event. Figure 3 shows the skeleton of the code.

```
void button_Click(object sender, RoutedEventArgs e)
   {object a = e.Source;
    Button btn = (Button)a;
    ...
   }
```

Figure 3
A skeleton outline of the button click event.

Each cell in Minesweeper needs two parameters: the number of mines in its neighborhood and a Boolean variable which tells whether or not that number should be displayed. You may be able to encode this information into the button properties. You can also create a separate *cell class*. You can then make a two dimensional array of these cells where each cell has a corresponding button. (Its row and column numbers can be used to find the index in the List and vice-versa). The cell class can have an int which will give the number of mines in its neighborhood plus a Boolean variable which indicates that it can be displayed.

After you have created the Cell class you can create an array like this:

```
private Cell [,] mineField = new Cell[16, 16];
```

Note that declaring a class does not run a constructor so you also need to write a loop to instantiate all of the Cells:

```
for(c=0;c<MAXC;c++)
   {for(r=0;r<MAXR;r++)
       mineField[c, r] = new Cell();
   }
```

Figure 4

This code fragment instantiates each cell in the array by running its constructor.

Method 2 – Graphics and Bitmaps

For this method we use no buttons – instead we draw the minesweeper game on a bitmap and make the bitmap a source for an image.

For the xaml we need only to place an image element onto our default grid. The window and image element need to be large enough to accommodate all of the cells in your minesweeper game. For a 16 x 16 where each cell is 20 x 20 this comes to 320 x 320 minimum. If you are putting buttons and other items on the screen your window and grid will have to be larger.

The image element will also need a click event for the mouse. You can use the `mouseLeftButtonDown` event from the events menu. Note that if you create this event and set a breakpoint inside the event code the image *will not* respond to the click event until something is placed on the image element.

Figure 6 shows some sample code which draws a grid of 16 x 16 cells on a bitmap where each cell is 20 pixels x 20 pixels. Once the bitmap is complete it is set as the source of the image element called imgPlot in this method.

Line 3 creates a black pen one pixel wide which is used to draw lines. Line 4 creates a drawing visual – essentially a memory block that we can set pixel data for the screen. Line 5 creates a drawing context called dc. The drawing context is what we use to draw with and it is assigned to the drawing visual we created in Line 4.

Lines 7 to 10 draw 17 column lines for the 16 columns using a black pen. Lines 11 to 14 do the same for the horizontal lines.

Line 15 closes the drawing context. This step is essential. If you fail to close the context before rendering you will get nothing.

Line 16 and 17 create the bitmap. In this case it is called a RenderTargetBitmap. We must specify the bitmap size – in this case 360 x 360 which is the same as the image size. The next two numbers (96, 96) give a resolution that is about 96 pixels/inch. The last argument is the pixel format. (I have not tried other numbers for the resolution and pixel format but these work well for this assignment.)

Line 18 renders the Drawing Visual to the bitmap and Line 19 makes the bitmap the source for the image element. Figure 7 shows the result.

```
 1   private void DrawGrid()
 2     {int r, c;
 3      Pen blkPen = new Pen(Brushes.Black, 1);
 4      DrawingVisual vis = new DrawingVisual();
 5      DrawingContext dc;
 6      dc = vis.RenderOpen();
 7      for(c=0;c<=MAXC;c++)
 8         {dc.DrawLine(blkPen, new Point(10+c*20, 10),
 9                                 new Point(10+c*20, 10+16*20));
10         }
11      for(r=0;r<=MAXR;r++)
12        {dc.DrawLine(blkPen, new Point(10, 10+r*20),
13                                 new Point(10+16*20, 10+r*20));
14        }
15      dc.Close();
16      RenderTargetBitmap bmp = new
17                  RenderTargetBitmap(360, 360, 96, 96, PixelFormats.Pbgra32);
18      bmp.Render(vis);
19      imgPlot.Source = bmp;
20     }
```

Figure 6

This method creates a 16 x 16 cell grid where each cell is 20 x 20. The grid is written to a bitmap and the bitmap is made the source if the image (imgPlot).

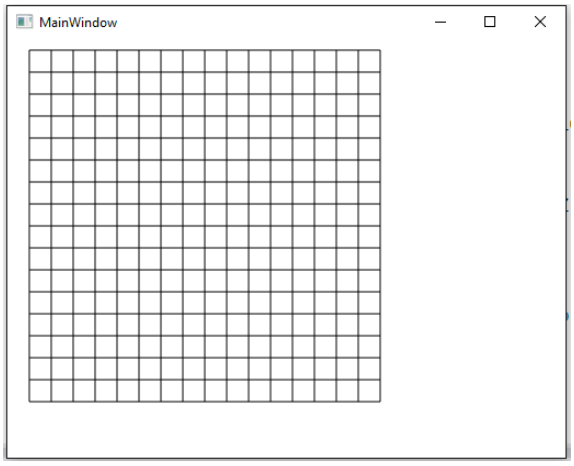

Figure 7

The 16 x 16 grid created by the method in Figure 6

To complete the project using graphics you will also need to be able draw and color a rectangle, draw text on the screen, and create a Cell class as was done in the button method.

For the Cell class you will want to have four private variables: Number of mines in neighborhood, a *show* Boolean variable to indicate that the cell should be displayed, and a row and a column number. (In the Button Method the row and column numbers could be extracted from the button name.)

The syntax for drawing a rectangle is:

```
dc.DrawRectangle(brush, pen, new Rect(x1, y1, width, height);
```

The brush is used to paint the rectangle. It must be a *SolidColorBrush* as in:

```
SolidColorBrush redBrush = new SolidColorBrush(Color.FromRgb(255, 0 , 0));
```

You can make the brush parameter *null* in which case the rectangle is not colored. The pen is used to draw the lines around the outside of the rectangle and it is the same as was used for the DrawLine in Figure 6.

To put text on the screen you need to use the FormattedText class as in the example below.

```
FormattedText fmtxt = new FormattedText("Hello Mom",
                        System.Globalization.CultureInfo.CurrentCulture,
                        FlowDirection.LeftToRight, new Typeface("Times New Roman"),
                        typeSize, brush);
dc.DrawText(fmtxt, new Point(x1, y1);
```

In the FormattedText declaration the typeSize is the font size – typically an integer between 8 and 24. In this example "Hello Mom" is the text that is placed on the screen with its top left corner at x_1, y_1.

Algorithm for Click on a Cell

The mines that are in the game are randomly placed except that the first cell that is clicked on may not have a mine under it and nether should it have a mine in its neighborhood. So you cannot place the mines on the board until after the first click. When the user clicks on a cell, if it is a mine the player loses and the game is over. If it is in the neighborhood of a mine that cell is exposed along with the number of mines in its neighborhood. If there are no mines in the neighborhood, then all empty cells open up to the edge of the minefield. Figure 8 shows an example of the opening click that exposes empty cells to the edge of the minefield. All of the cells around the edge have numbers indicating the number of mines in their neighborhoods. You will need to write an algorithm that will find all of the empty cells out to the edge of the minefield. There are two ways that this can be done: recursively, and non-recursively.

Recursive Algorithm.

For this algorithm you need to write a method which is recursive. Suppose this method is called *RecursiveOpen* and it has arguments specifying the cell that has been clicked so that you know the row and column number. If this cell is empty, you mark it to be exposed. You can set the *Show* variable to true in the Cell class. Next you write a loop which goes to all eight of its neighbors. If the cell in the neighborhood is empty, it calls itself and passes itself the new row and column number. Otherwise, it marks the cell as *Show* and goes on. This process will go on until the algorithm runs can find no more empty cells and is able to complete its search of all of the neighboring cells.

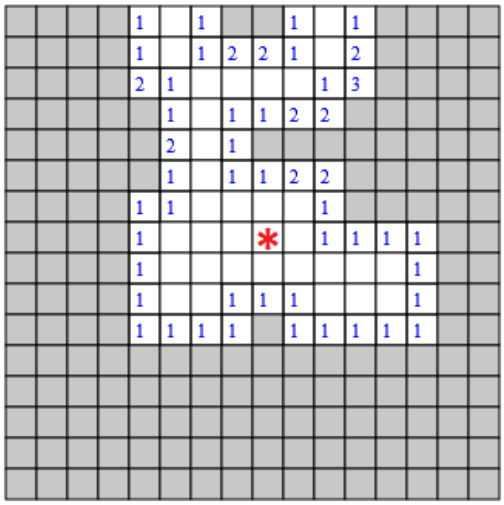

Figure 8
The first cell clicked on is made empty as well as all of the cells in its neighborhood. The initial cell is marked with an asterisk.

Non-Recursive Algorithm
There are lots of way to write the non-recursive algorithm. One way to do it is to use a Queue and place empty cells in the queue along with all of their empty neighbors. C# supports a queue structure:

```
private Queue<Cell> cellQ = new Queue<Cell>();
```

In this declaration the queue is named cellQ and it holds members of the Cell class. You could also hold Points which give a row column location. The relevant operators are:

`cellQ.Enqueue` – Place a cell in the queue
`cellQ.Dequeue` – Remove a cell from the queue
`cellQ.Count` – Get the number of items in the queue

For the non-recursive algorithm, when an empty cell is clicked on, you place it in the queue and enter a while loop which runs until the queue is empty. Inside the while loop you mark the empty cell as *Show* and move to each of its eight neighbors. If these neighbors are empty you place them in the queue and end the loop. This loop continues placing neighbors of neighbors in the queue until all of the empty cells are found and marked.