Top-Level Architecture

Application Loop

At the very top level of our game we have the main **application loop** in which events are handled, game logic is performed and rendering takes place. This loop runs continuously while the program is running, multiple times a second (depending on the frame rate) and is where all of the real-time event driven logic goes.

```
        /* Application Loop */
        Bool running = true;
        while ( running ) {

                /* Handle Events */

                /* Game Logic */

                /* Rendering */
        }
```

Before this loop, all of the game data that should persist between function calls is placed such as the the internal representation of the board and information about the state of input events (like mouse position and whether or not the mouse was clicked).

Game-Data Representation

Internally the game board is represented as a 2D matrix of integers. The value of each integer signifies what type of game piece is located at each position (this is done with enumerated types).

We can either make a 2D array that encapsulates all of the board data, or we can have a 2D array that has a *boundary that is not rendered* onto the screen (the latter is preferable because the boundary can be used for algorithms such as A* pathfinding and others).

```
{
        {1, 1, 1, 1, 1, 1, 1, 1},
        {1, 0, 0, 0, 0, 0, 0, 1},
        {1, 0, 0, 0, 0, 0, 0, 1},
        {1, 0, 0, 0, 0, 0, 0, 1},
        {1, 0, 0, 0, 0, 0, 0, 1},
        {1, 0, 0, 0, 0, 0, 0, 1},
        {1, 0, 0, 0, 0, 0, 0, 1},
        {1, 1, 1, 1, 1, 1, 1, 1}
}
```

Immediate Mode Graphical User Interface

In the **Immediate Mode Graphical User Interface** (**IMGUI**) style, all of the user interface code is handled with procedural function calls as opposed to Retained Mode Graphical User Interfaces which uses an initialization step, and callbacks.

ex:

```
if( button() ) {
      // Action Code
}
```

In the above code, the `button()` function is a full encapsulation of a **universal button** and it's functionally. The `button()` function returns a Boolean value when the button is called. The function returns true if the user has interacted with the menu item. the Action code will be executed (the desired behavior for the button). The function also handles how the button will be rendered (as a side effect of the function being called) depending on the mouse position and UI state.

Since we know that all GUI widgets (like buttons, scroll bars and text fields) use basically the same information (mouse position, whether an item is 'active' or 'hot', which owner was chosen), we will need something to store the universal state for all of the menu items. This is done with a structure called a **UI Context**:

```
ui_id Hot;
ui_id Active;

struct ui_id
{
      int Owner;
      int Item;
      int Index;
};
```

The **UI Context** is basically global data that keeps track of all the information that all of the GUI items could need. Note that the **Hot** and **Active** values keep track of whether of not the GUI item is *about to be interacted with* (eg. the mouse is hovering over a button) and whether or not a GUI item is *actually being interacted with* (eq. A button was just clicked).

Usually there are multiple items that can be interacted with at any given moment. This means that we need to have some **identifier** to distinguish between different items. Using the IMGUI style, this is done by having some arbitrary data that is used to identify the item (eg. 0x01 is button number 1 and 0x02 is button number 2). These **identifier** values are passed into the procedural function calls.

```
bool doButton(ui, ID, Text, …){
    if(Active)
        if(mouseWentUp)
             if(Hot) Result = true;
             setNotActive;
    else
        if(mouseWentDown) setActive;

    if(inside()) setHot();
}
```

Here is the full implementation of the **`doButton()`** function we described earlier.

<u>High-Level Game Logic</u>

<u>State Machines</u>

In games there are a lot of different states that the game can be in.

<u>Translating Mouse Position to Board-Coordinates</u>

To find the corresponding board coordinates based on the position of the mouse, use the following algorithm:

```
for(int row = 0; row < boardSize - 1; row++) {
     for(int col = 0; col < boardSize - 1; col++) {
          if ( mouseX > ( X_MARGIN + col * TILE_SIZE)      &&
               mouseX < (X_MARGIN + (col + 1) * TILE_SIZE) &&
               mouseY > (Y_MARGIN + row * TILE_SIZE)    &&
               mouseY < (Y_MARGIN + (row + 1) * TILE_SIZE) ) {
                    return row and col;
          }
     }
}
```

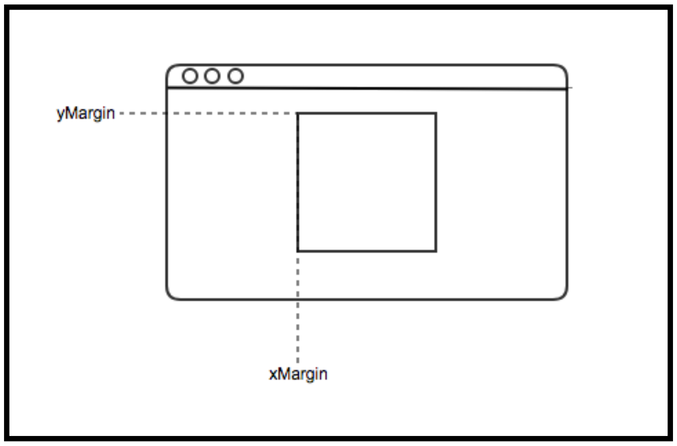
<u>Rendering</u>

`WINDOW_WIDTH`: (This is the width of the window in pixels.)
`WINDOW_HEIGHT`: (This is the height of the window in pixels.)
`BOARD_SIZE`: (This is the size of of the board (all board are assumed to be square))
`TILE_SIZE`: (The tile is a single rectangle that composes the board. This is its size in pixels.)

<u>Centering a Rectangle int a Window</u>

```
xMargin = ( windowWidth - (BOARD_SIZE * TILE_SIZE )) / 2
yMargin = ( windowHeight - (BOARD_SIZE * TILE_SIZE )) / 2
```

These coordinates will give the top-left point of a rectangle that will be centered on the window.



This is useful because we can create a fixed-sized rectangle and display it onto the window regardless of the dimensions of the window (e.x., we can create a square that is automatically centered). This also gives room for putting buttons and text on the screen.