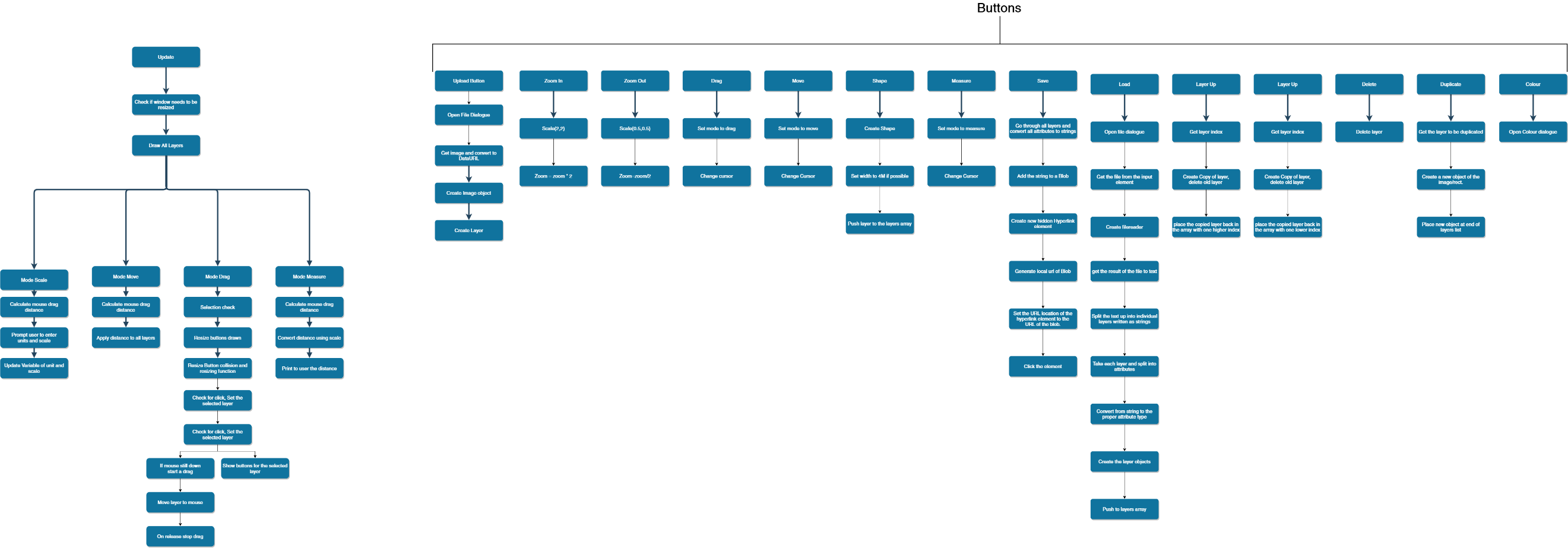
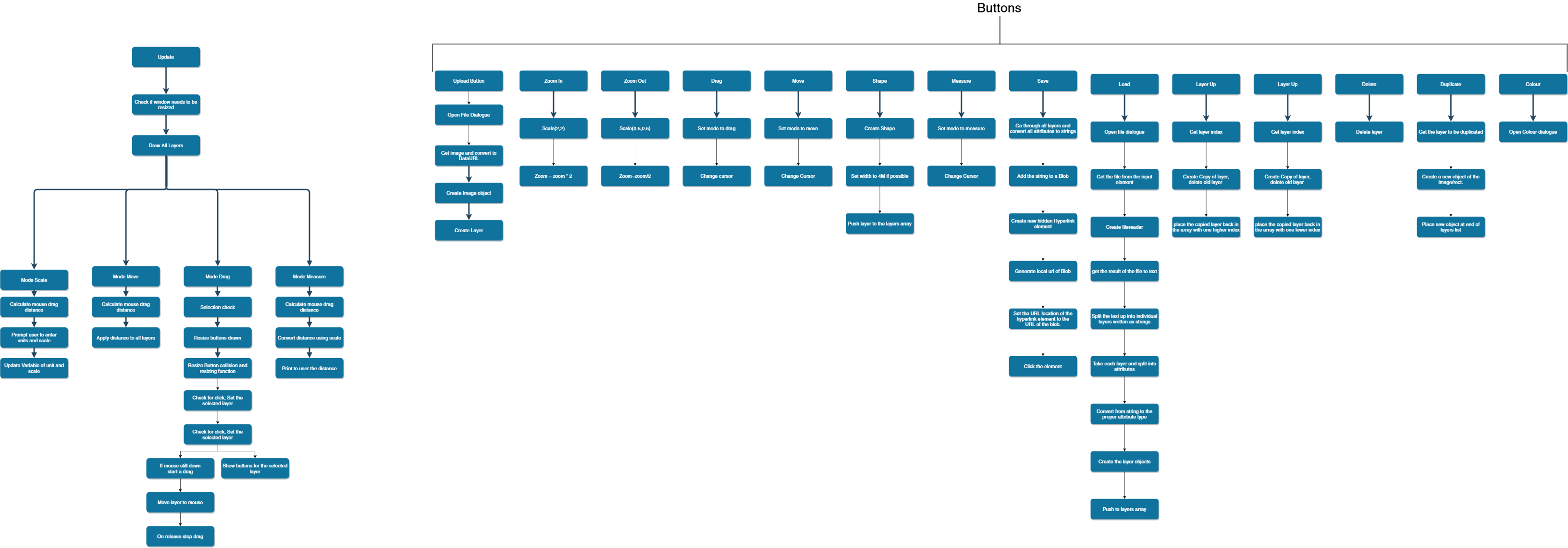
Documented Design

Overall system design

****

****

The overall system design is split up into two sections, since one side is the buttons which are managed by the html page and the other side is my update function which runs the canvas and updates it every frame. These two diagrams are very close to each other where the buttons set attributes for the next update loop. The update diagram has control on when to show some layers.

|  |  |  |
| --- | --- | --- |
| IPSO | Program Section | Item |
| Input | Getting Scale | Scale Factor |
| Scale Units |
|  | Opacity Slider | Opacity |
|  | Colour Getter | Colour |
|  | Background Setter | Background |
|  | Upload | Image |
|  | Load | File |
|  | Price Input Box | Price per m2 |
| Processing | Checking if a layer should be dragged | Is the layer selected already and is the mouse still down? |
|  | Checking if colour should be changed | Check previous colour of layer and check the new colour of element, if different set to new colour. |
|  | Check if mouse is clicking the layer | Is the mouse down and over the area which the is layer already operating? |
|  | Check if canvas is being stretched | If the x/y of the canvas is different to the screen size of the canvas |
|  | Check if dots clicked | Is the mouse closer to the centre of the dot than the radius of the dot and mouse down |
| Storage | Save | All layers and attributes in a text file. |
| Output | Area | Display total of each layer’s area as text. |
|  | Price of job | Display the total area x the price per M2 |
|  | Dimensions | Display the dimensions of the layer on the layer. Get the width and height X by scale |

File Structure

File

X

Y

ShowDimensions

Background

Colour

Opacity

Rotation

Type

Obj

Layer 1

X

Y

ShowDimensions

Background

Colour

Opacity

Rotation

Type

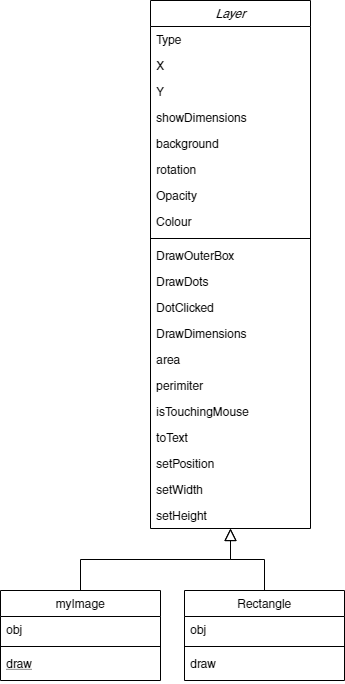
Obj

Layer 2

…

Class Diagram

Class diagram for the Layer class

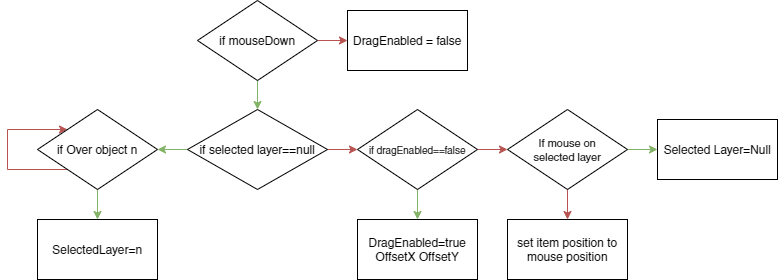


Using a class diagram has helped me to plan for what things I need in my classes and what is useful to inherit. I have one parent class which holds all of the base information (i.e. location) that a layer should have and allow for child classes to take these and have their own draw classes. In theory this approach should allow for other developers to easily add their own type of layer in the future and utilise the methods I have created.

Furthermore, making this a class allows me to split the code up into manageable packages which can be tested individually to check for bugs and errors, this should help me speed up development time significantly as I won’t have to sift through spaghetti code to find errors. This also decreases the amount of lines coded, since reusing classes stops me from retyping code, this is important, especially in JavaScript as the less the browser has to process the higher fps it can run at, improving the user experience and progressing to my goal of usability

Mapping out features on a flow chart gives me a good idea of how to implement them, for instance, dragging, I have very specific requirements for this feature which has lots of different states the user can be in, dragging an object, not dragging an object but selected, just released an object from a drag etc. Visualising problems like this gives me a better idea of the steps and variables needed before I waste time and computing power on a poor solution.

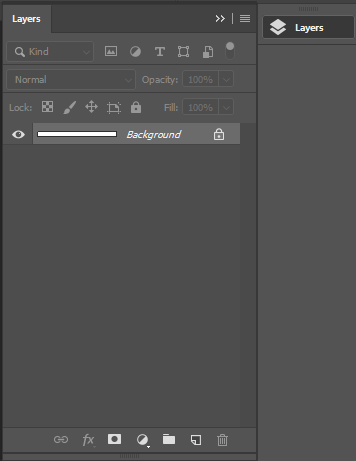
Figure 3 Full flow chart of the dragging function. Green arrows signify true and red arrows signify false.



I iterated over this flowchart multiple times to remove repeated parts of the chart, i.e. two of the same if statements in different locations. This helped me to visualise my code and how it will come together in practice.

Since most of my project relies on the manipulation of shapes I have decided to figure out if there are any similarities to rectangles which I will support and Images which I will support. From this I discovered that they were indeed similar, in fact drawing an image is almost the same process since it has a location, height and width, the only difference is that one contained an image object.

Screenshot from Adobe Photoshop, showcasing the layers system



From this I have discovered that using classes to display rectangles and images would be beneficial as it will provide structure and allow me to organise my code into structured areas where classes could inherit from a main layer class, since both images and rectangles have some base properties and I have decided to name an instance of this class a Layer, fitting with similar systems like Adobe Photoshop, which uses a similar system to keep track of its parts

**Algorithms**

Since I am using the canvas, I have to operate over multiple frames. To detect if a rectangle is being dragged, (Which is an essential to the success of my system), I can store how many consecutive frames the mouse has been over the object and the mouse has been down. If this number is 2 then I know that the user intends to drag the object and can initiate a drag by setting the appropriate variable to save this state. As the mouse is being held down rather than just a click. However, this technique requires the use of a global variable since I must store the number of frames outside the detection function itself which wipes its variables after it returns or finishes.

Global frameQuantity

Function Update

If mouse is down and over rectangle

frameQuantity = frameQuantity + 1

Else

frameQuantity = 0

if frameQuantity >= 2

rectangle Location = mouse Location

Simplified pseudocode of the drag detection algorithm

However the story only gets more complicated from here, since I want to be able to just click the rectangle to access the size editing system or edit properties of the rectangle, I have to account for this and stop detecting clicks when editing size, and show the appropriate sizing features when the rectangle is clicked.

Furthermore, I cannot use a system as literal as the one above, since I have lots of times when I would trigger the system by accident. If I were already dragging another rectangle I do not want to pick up this rectangle and start dragging too if I mouse over it as this would look strange and mean that you would have to drag rectangles carefully around other rectangles to avoid picking them up. From this I can learn that I will need lots of different global variables to describe the state which the interface is in, it should know when an object is being dragged and not try pick up others, which I will do by storing if an object is being dragged in a variable, since you can only drag the layer which is selected. I should also know which layer is selected, since I will want to show resizing features only for this layer and not for other layers which aren’t selected. Another consideration is if I want the rectangle to snap to the location of the mouse. Currently this system would do that as I haven’t allowed for the offset of a mouse being in the middle of a rectangle (Figure 8).

This can be solved with another variable which stores the offset of the mouse from the rectangle location and reapplies it when I am dragging the rectangle, which maintains the position of the mouse with the rectangle. However, this is another variable which will have to store data past one frame, meaning I have to use a global variable.

As you can see, a feature which seems so easy, branches out into many problems which must be solved to make the system function properly.

Visualisation of rectangle snapping to mouse

Frame 1, before mouse goes down

Frame 2, as mouse goes down, selected layer set

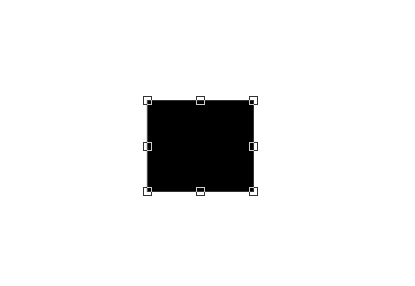
Frame 3, drag detected, rectangle snaps to location of mouse

This problem is flowcharted earlier in the design, which shows what has to be checked each frame to discover what to do on the screen.

Resizing

Being able to resize a layer is very important since a user should be able to change the size of rectangles on the screen. To figure out how I would implement a system like this I looked at photoshop, which I know has a system like this already. Photoshop uses rectangles on each of the corners to signify a zone which the user can click and drag to edit size.

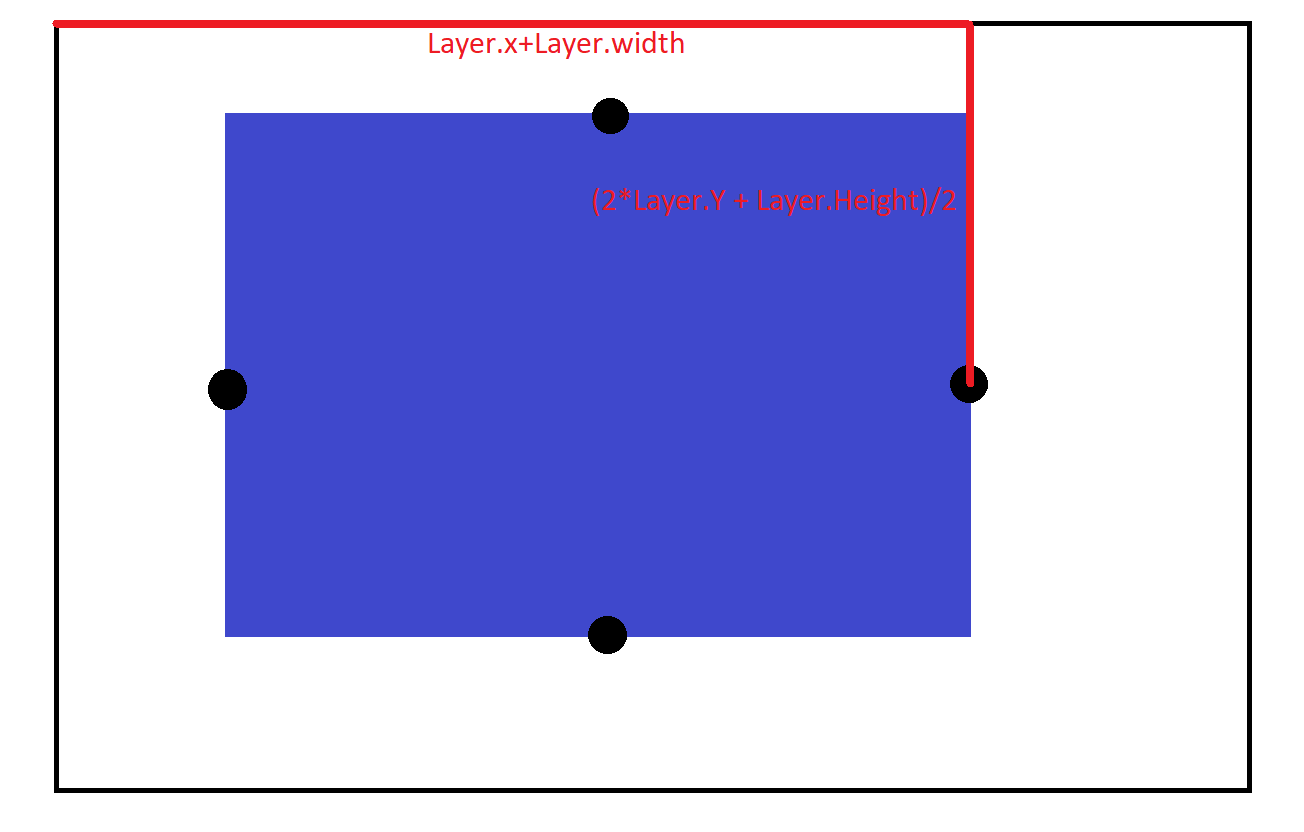
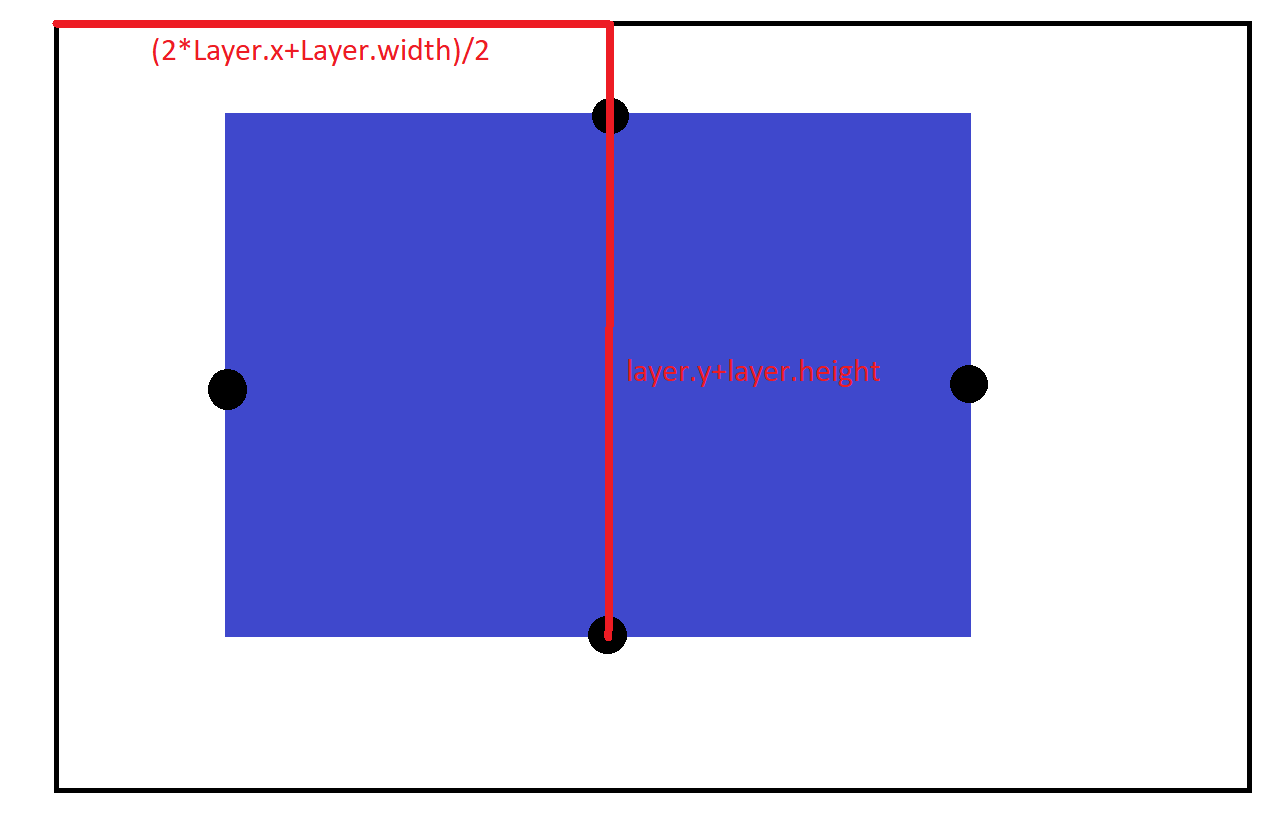
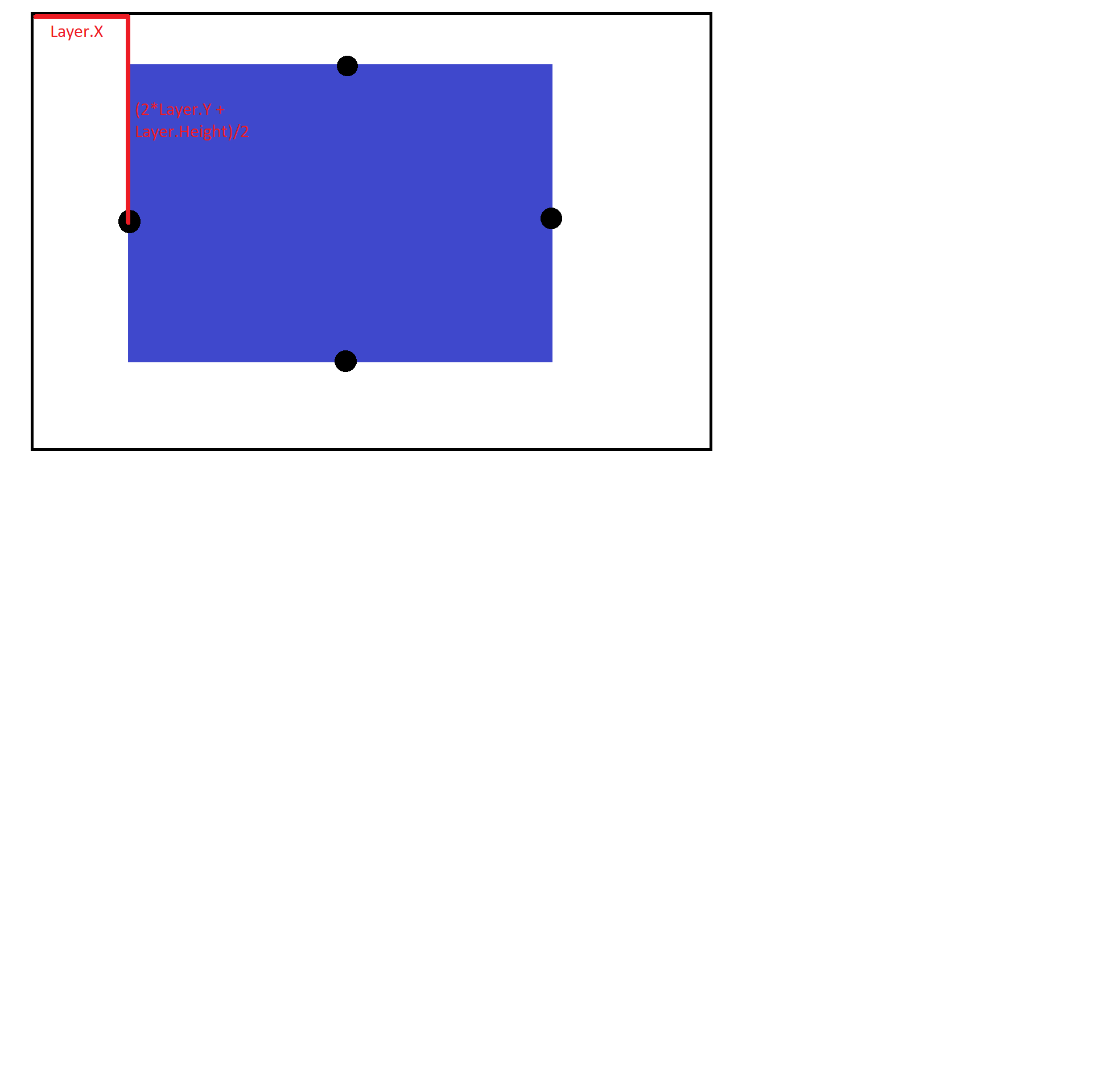
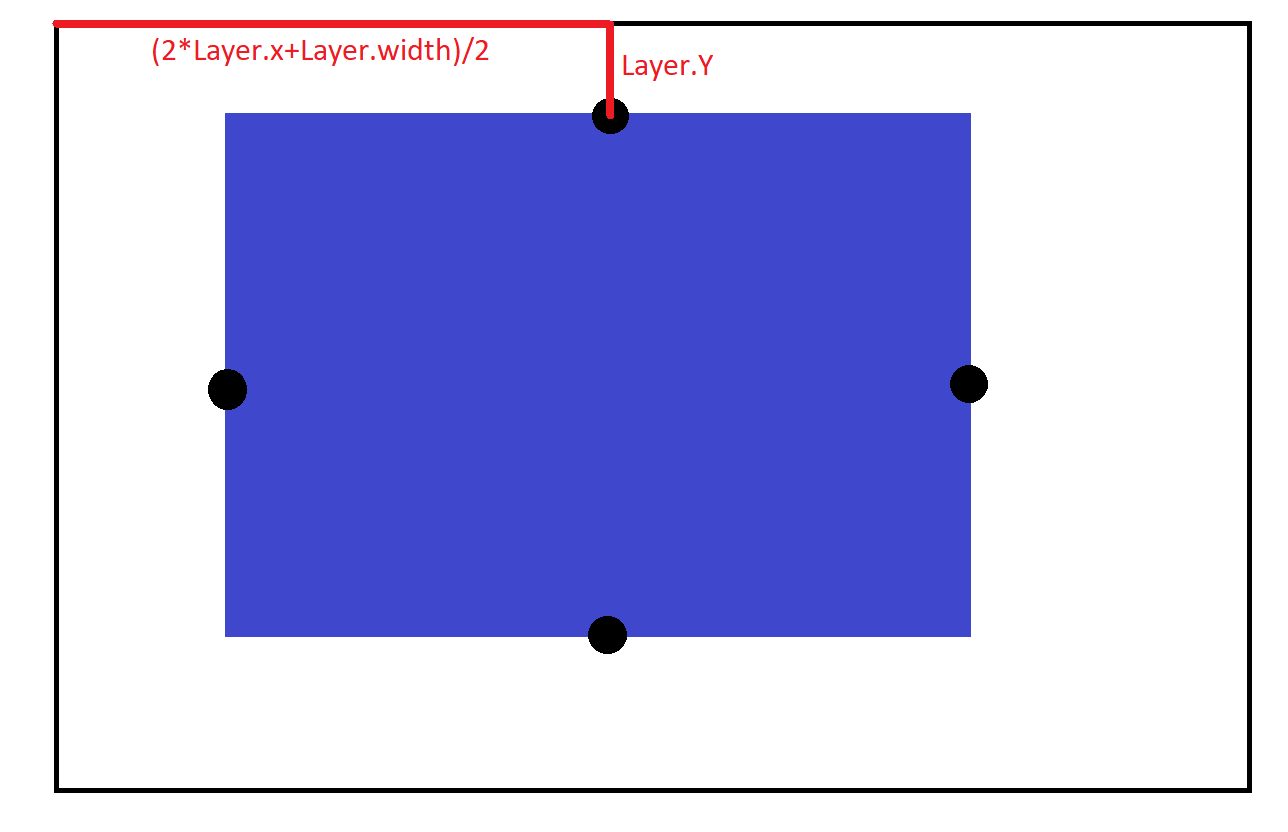
Example of how photoshop visually allow the user to resize shapes



I immediately figured out potential roadblocks or problems I would have to solve to get my desired result.

Calculating where to place the resizing dots, would be a problem since it depended on may factors. Where the resize dots should be placed varies on size of the layer, where the layer is, what type of layer it is and what the zoom of the canvas is.

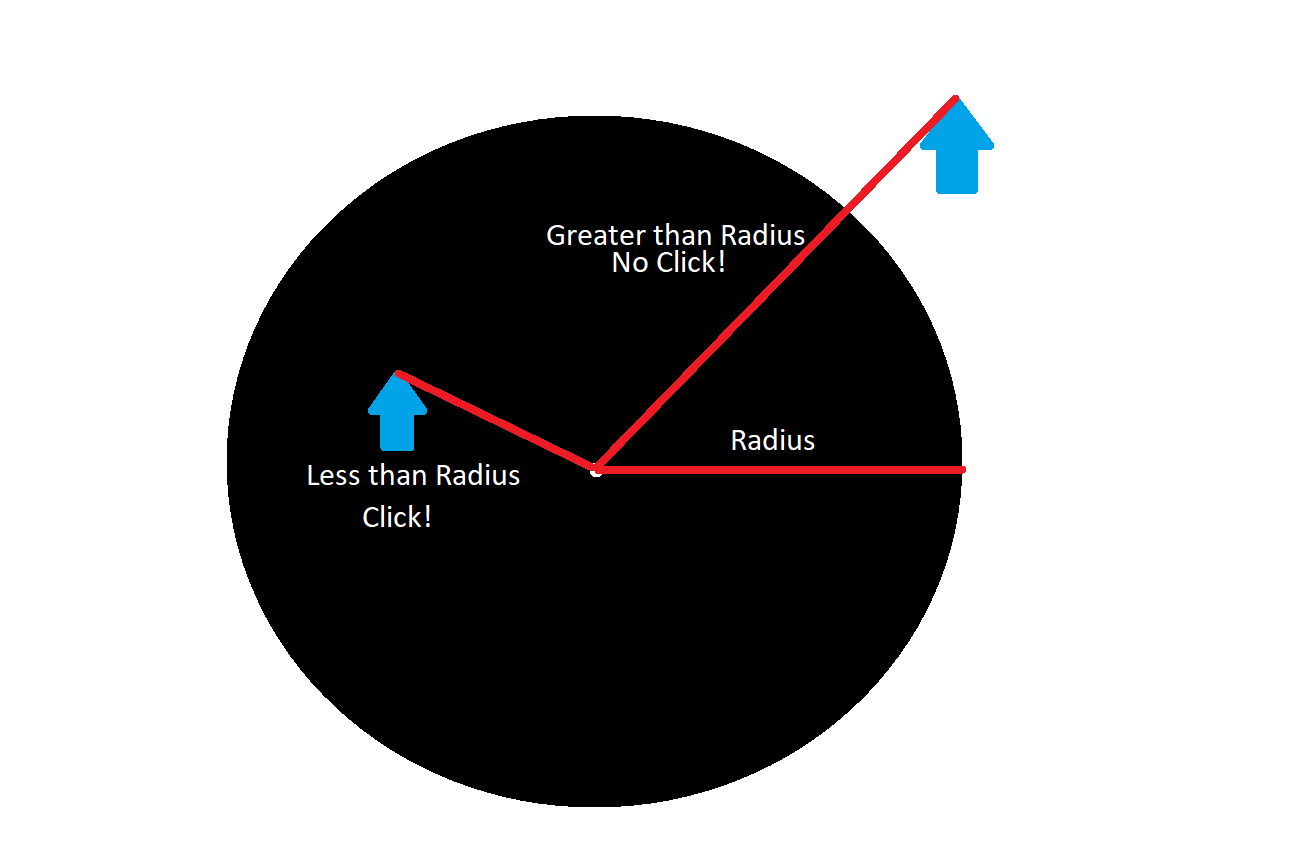
I decided that the resize dots should be controlled directly by my layer class, as it would have internal access to lots of properties about the layer, this would be much easier than an external class or function which would have to access the layer class too often.



Visualisation of where the dots need to be placed.

To get the location of the resize dots I have to find the midpoint between two corners, luckily the canvas handles the zooming, so I do not have to manually change where the resize dots are based upon the zoom. However, I do have to edit the size of the resize dots with zoom as otherwise they become too big when you zoom in or too small when you zoom out.

Detecting when they are clicked can be done by calculating the distance from the centre of the circle and seeing if it is smaller than the radius, if so, it is a click.



Demonstrating how I will detect clicks on a circle

After the program has detected a click then it can adjust the width and height of the layer appropriately, one consideration is that it has to ensure that the width and height of the layer is not negative, as you cannot have a negative width of grass to lay down. This would mess up the calculation of area of the grass, as it would say that the layer has a negative area, which would obviously be wrong.

Pseudocode with equations (only for left resize dot)

Draw Circle at layer X , (2\*layer Y+Layer Width)/2 with the radius 10/zoom

If the square root of ( mouse.x-layerWidth^2 + mouse.y-layerHeight^2 ) < 10/zoom

//Mouse over dot

If MouseDown

If newWidth > 0

Layer X = MouseX

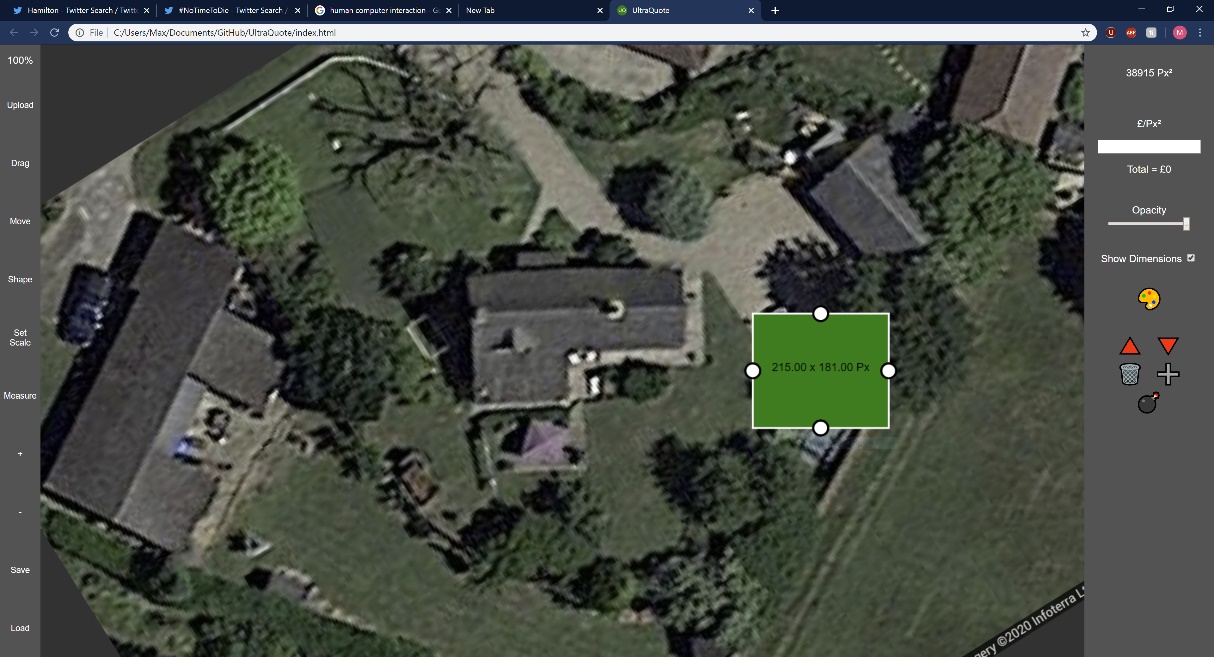
LayerWidth = new Width

//New width is defined by the width of the layer when the resize dot was first clicked + the distance the mouse has gone in x direction (must be done over multiple frames)

**Data Structures**

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Data Type/Structure** | **Validation** | **Sample Data** |
| Layers | Array of Layer |  | [ [Layer], [Layer] ] |
| Layer.x | Integer |  | 10 |
| Layer.showDimensions | Boolean |  | True |
| Layer.Obj | Object of Type Image or Rect |  | {Width:10, Height:4} |
| SelectedLayer | Integer | 0>SelectedLayer<Layers.Length | 3 |
| ScaleUnit | String | Not Null or “” | Metres |
| Scale | Integer |  | 16 |
| DragEnabled | Boolean |  | False |
| Mouse | Object {MouseDown:Boolean  Width: integer  Height: integer} |  | {MouseDown:True  Width:359  Height:2334} |
| Zoom | Integer |  | 64 |
| PreviousSize | Object {Width: integer, Height: integer} |  |  |
| Ui.Value | Integer |  | 43 |

**Human Computer Interaction**



1

4

3

2

6

5

The idea for my Human Computer Interaction is to have everything necessary on the screen at once, while not having multiple pages to scroll through, for this reason I have decided to have a section on the side of the screen which only pops up when a layer is clicked and disappears when nothing is selected, this also goes towards meeting my 9th objective of a “usable” program as a feature like this directly supports usability by guiding them to the correct options.

1. Rectangle- allows the user to drag and drop it around the screen, this means they do not have to use arrow keys or type in a location manually and can instead use a mouse which is intuitive and fits closely to the previous solution allowing for a smooth transition from the previous solution to my new solution. Plus, a user can choose to just select the layer by clicking which brings up the layer specific buttons (3).
2. Price input box – Takes the value, which is input and reactively adjusts the price above, very simple solution, instead of having an on-screen prompt/alert which would have taken time to load and possibly confused the user. This solution gives the user room for error too since the box only allows numbers to be input.
3. Layer specific buttons – Buttons which only show when they are applicable, rectangles show colour, up layer, down layer, delete and duplicate, these buttons disappear when the layer is unselected.
4. Resize dots – allow a user to visually drag these dots with the mouse to resize a selected layer, only show when a layer is selected to avoid screen clutter, only show for the selected layer to avoid user confusion.
5. Background layer – Image which doesn’t allow dragging and does allow rotation, this prevents users from accidentally clicking the background and misaligning their whole project, increases usability.
6. Always shown buttons – Buttons which must always be shown to the user, for instance the importing of an image, saving or loading their work, having these buttons always show gives the user a stable area which doesn’t change, gives user a sense of direction in the UI.
7. Zoom Indicator – Automatically adapts to show the current zoom, this gives the user a figure to stop them from zooming in/out too far to find their work in the canvas.