1 Submission Instructions

Submit to Brightspace on or before the due date a compressed file (.zip) that includes

1. Header and source files for all classes instructed below.
2. A working Makefile that compiles and links all code into a single executable. The Makefile should be specific to this assignment - do not use a generic Makefile.
3. A README file with your name, student number, a list of all files and a brief description of their purpose (where necessary), compilation and execution instructions. In addition, if you have done anything different from the specification, please detail it here. You may introduce functions, or change functions, as long as the tests still run correctly, and you do not use any libraries other than those permitted (`iostream, string, sstream, iomanip`)

2 Learning Outcomes

You will learn the basics of classes, objects, arrays, and using static memory in C++. You will make a few simple classes, populate the members using constructors and write functions to process the data. You will learn how to provide a working Makefile, and implement a small amount of application logic.

You will also be writing a wrapper library for a GUI in the X11 windowing system.

3 Overview

The X11 windowing system is a fairly low-level library for making and rendering onto a window. Our goal over this assignment (and possibly future assignments, depending on how this goes) is to create a GUI library for the X11 system. That is, we want to handle all the X11 complexity for the user, and present the user with some convenient, easy to use abstractions so that making an X11 GUI is easy.

4 Classes Overview

This application will consist of 4 classes, plus one `Tester` class. It also contains a `defs.h` file. A brief description of each class and the given files follows.

1. `RGB` - The X11 system takes an unsigned long as a colour argument, and reads the first (least significant) 24 bits to get the RGB values (red, green, blue). This class makes defining colours easier.
2. `Button` - a basic, rectangular button with some text written on it.
3. `Panel` - a container for holding GUI elements, so that we can create complex GUI elements that can be reused in different places in our `CuWindow`. For now, it only holds `Buttons`.
4. `CuWindow` - the main class. This class must manage GUI elements (for now, it only handles `Panels`). It also manages creating and destroying an X11 window and graphics context.
5. `Tester` - some functions used to test your code.
6. `defs.h` - this has a preprocessor constant that you may use to initialize your arrays to a consistent size.

   All your classes should all be in separate header and source files that are compiled explicitly in your Makefile!!. In addition you should compile and link the `Tester` class into your executable.

5 Instructions

Download the starting code from Brightspace.

All member variables are `private` unless otherwise noted. All member functions are `public` unless otherwise noted. Some return values are not explicitly given. You should use your best judgment (they will often be `void`, but not always). ALL CLASSES MUST HAVE A PRINT FUNCTION UNLESS OTHERWISE NOTED. This will print to the **standard output**, not on the window itself. This prints metadata about the class like names or sizes but not data contained in a data structure like an array (unless explicitly specified).

You are allowed to pass `string` objects by value *for this assignment only*. Passing by `string&` does not work with string literals (and we will learn why in the lessons on **Encapsulation**). You may use `string` or `const string&`. For the purposes of this assignment they function the same, but the second version is more efficient, as we will learn.

Your finished code should compile into a single executable called `a1` using the command `make a1`, `make`, or `make all`, using a Makefile that you wrote yourself. Your submission should consist of a single zip file with a suitable name (e.g., `assignment1.zip`) that contains a folder containing all your files. This folder should also contain a README with your name, student number, a list of all files that are included. The main thing it should include is **any changes to the specification**. I don't mind when you do things differently - I would encourage it, as long as it is code you write yourself, and not outside libraries - but you *must* document these changes.

5.1 Notes on Automated Marking

Initially `main.cc` will not compile because it references classes and functions that do not exist. Feel free to comment out parts that are not working. Or you can write your classes with empty implementations that you complete as you go (which I would recommend, but it is up to you). You may also `#include` additional header files for classes other than `CuWindow`. You may make your own file with your own `main` function and write your own tests, and compile that separately. But the code provided in `main.cc` using the `Tester` class is what will be run to determine your **application requirements** mark.

Likely most of the bugs you experience will be in your own code. However, you might find bugs or inconsistencies in the provided test code as well. Please bring them to my attention.

When there is a conflict between this specification and the test code, **the test code takes precedence**. I guarantee you there will be discrepancies - most of these can be solved with common sense. A rule of thumb is that running code is far better than exactly following the specification. The specification is just a guide. Your assignment is to produce a running program, whether it follows the specification or not. Make sure it runs correctly, and does the things it is supposed to do!

5.2 The defs.h file

This file defines some handy preprocessor constants so that these values are consistent across our application. In this case it defines `MAX_COMPONENTS` which you should use as the size to initialize your arrays to. It also defines `CuColour`, which you can use to set the colour when drawing on X11 windows, as well as some predefined colours you may use.

5.3 The RGB Class

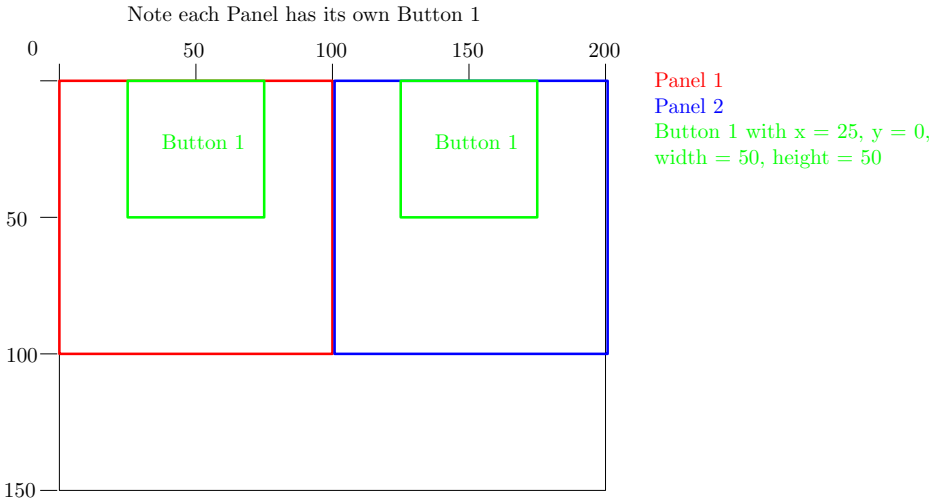
The `RGB` class holds members (`ints`) for the Red, Green, and Blue values for an X11 colour. Its main purpose is to change this into an appropriate `unsigned long` to feed to X11 functions, or to change an unsigned long into appropriate `r, g, b` values. You will need to use bit operations to convert back and forth. Although we are using `ints` to store the r,g,b values, they each only store 1 byte of information (i.e., we could use unsigned chars instead). And although CuColour is 8 bytes, we are only concerned with the first 3 bytes, where the least significant byte corresponds to the b value, the second-most least significant byte is the g value, and the third-most least significant byte is the r value.

1. Member variables:
   (a) `int r, int g, int b`: these must be values in the range 0 to 255.
2. Constructors:
   (a) One constructor that takes `red, green, blue` values as arguments and initializes the member variables appropriately.
   (b) One constructor that takes a `CuColour` value and initializes the member variables appropriately (you will need to use bit operations to do this).
   (c) Make a no-argument constructor that initializes all the members to appropriate default values such as 0.
3. Member functions:
   (a) `getColour`: this should return a `CuColour` value that is generated from the `r, g, b` values.
   (b) `setColour(CuColour)`: This should generate the appropriate `r, g, b` values by reading the third, second, and first bytes (in least-significant order).
   (c) Any other getters and setters you think are necessary.

5.4 The Button Class

The `Button` class has member variables for position, dimensions, and a label of text that is displayed on the `Button`. This label also acts as the unique id of the `Button`. A `Button` is stored inside a `Panel`, rather than in `CuWindow` directly.

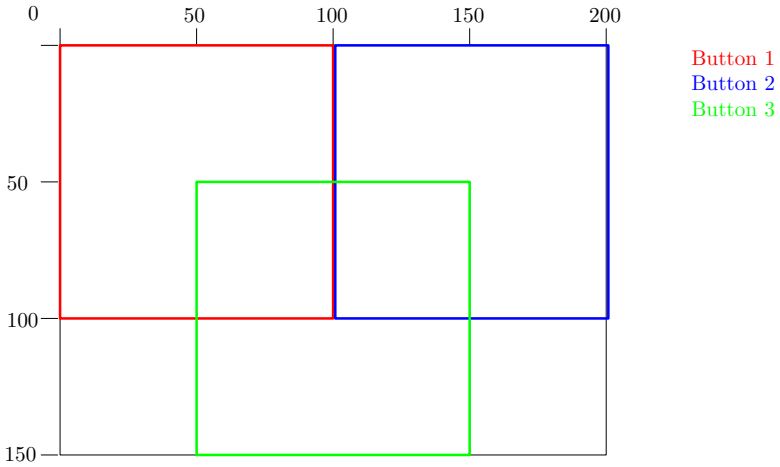
1. Member variables:
   (a) `int x, int y`: the x and y coordinate of the `Button` within its current `Panel`.
   (b) `int width, int height`: the `width` and `height` of the `Button` in pixels.
   (c) `string label`: The text that is displayed on the `Button`, as well as the unique id of the `Button`.
   (d) `RGB fill, border`: the colour of the `Button` and the `Button` border.
2. Constructors:
   (a) Two constructors that takes `x, y, width, height, label, fill, border` as arguments and initializes the member variables appropriately. The first passes in `fill` and `border` as `RGB` objects, and the second should pass in `fill` and `border` as `CuColour` values.
   (b) Make a no-argument constructor that initializes all the members to appropriate default values. Choose your favourite colours for the `fill` and `border`.
3. Member functions:
   (a) You should make getters and setters as needed.
   (b) A `void draw(Display *display, Window win, GC gc, int parentX, int parentY);` function. This should draw the button as a (filled) rectangle (see the X11 documentation). The label should be drawn over top of the button. `parentX` and `parentY` are the x and y-coordinate of the `Panel` that this `Button` is stored in. Thus, when you draw the `Button` onto the window, the x and y-coordinate of the `Button` should be offset by `parentX` and `parentY`. That is, if a Button has location `x=0` and `y=0`, then it should always be drawn in the top left of the `Panel`, wherever the `Panel` is located. If the `Panel` containing the `Button` has coordinates `x=100` and `y=50`, then that is where the `Button` should be drawn.



(c) A `bool overlaps(Button& b)` function. This should return `true` if the `Button` in question overlaps the given Button (it is ok if they are touching at a single pixel). For example,

i. Button 1 we have x=0, y=0, width = 100, height = 100.
ii. Button 2 we have x=100, y=0, width = 100, height = 100.
iii. Button 3 we have x=50, y=50, width = 100, height = 100.

Buttons 1 and 2 do NOT overlap. However, Buttons 1 and 3 do overlap, and Buttons 2 and 3 do overlap. See figure below.



(d) A `print` function. This should print (to the console, not to the Window) all the Button information. You may wish to `#include <iomanip>` for some formatting tools, but you do not have to. For an *acceptable* print output, see below.

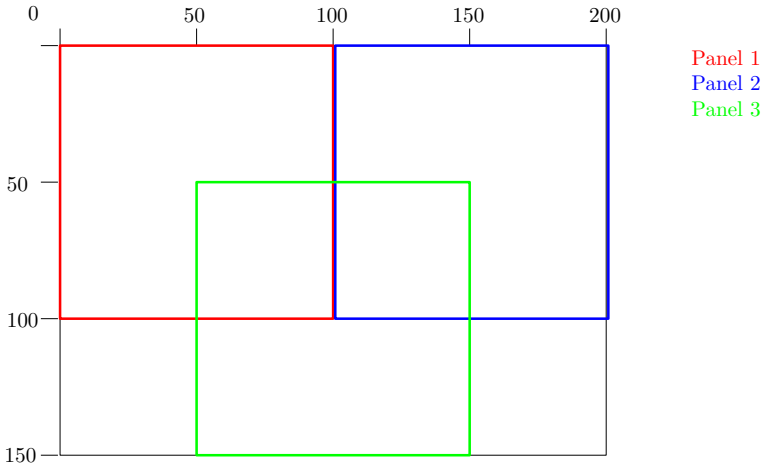
```
Button:   Button 1
Position: 10, 10
Size:     80, 50
```

5.5 The Panel Class

The `Panel` class will duplicate a lot of what the `Button` class does, that is, it has x, y coordinates and width and height. Instead of a label it has a `string id`. It maintains a statically allocated array of `Button` objects.

1. Member variables:
   (a) `int x, int y`: the x and y coordinate of the `Panel` within its current `CuWindow`.
   (b) `int width, int height`: the width and height of the `Panel` in pixels.
   (c) `string id`: The unique id of the `Panel`.
   (d) A statically allocated array of `Button` objects of size `MAX_COMPONENTS`.
   (e) An int that keeps track of the current number of `Buttons` stored (set to 0 initially).
2. Constructors:
   (a) A constructor that takes `x, y, width, height, id` as arguments and initializes the member variables appropriately.
   (b) Make a no-argument constructor that initializes all the members to appropriate default values.
3. Member functions:
   (a) You should make getters and setters as needed.
   (b) `bool addButton` - This function should take a Button as an argument (passed by reference). If this `Button` does not overlap any other `Button` in the `Panel`, AND it does not extend outside of the `Panel`, then add this `Button` to the back of the `Button` array. Return true if the `Button` is added, and false otherwise.
   (c) `bool removeButton (string id)` - This function should find the button with the given id and remove it from the array. It should "close the gap" in the array. That is, consider buttons b1, b2, b3, and b4 in an array in the following indices: 0:b1, 1:b2, 2:b3, 3:b4. If we remove b2, then the resulting array should look like 0:b1, 1:b3, 2:b4.
   (d) A `void draw(Display *display, Window win, GC gc);` function. Technically `Panels` should be invisible. However, we will draw them as a(n unfilled) rectangle for debugging purposes (so we can ensure their placement is correct). In addition, draw all the `Buttons` onto the `Panel`. The `Buttons` x and y coordinates should be offset by the `Panel`'s x and y coordinates. For example, if the `Panel` has x = 300, and the `Button` has x = 10, then the `Button` should be drawn a x-coordinate 310 on the underlying window.
   (e) A `bool overlaps(Panel& p)` function. This should return `true` if the `Panel` in question overlaps the given `Panel` (it is ok if they are touching at a single pixel). For example,
      i. Panel 1 we have x=0, y=0, width = 100, height = 100.
      ii. Panel 2 we have x=100, y=0, width = 100, height = 100.
      iii. Panel 3 we have x=50, y=50, width = 100, height = 100.

      Panels 1 and 2 do NOT overlap. However, Panels 1 and 3 do overlap, and Panels 2 and 3 do overlap. See figure below.



(f) A `print` function. This should print (to the console, not to the Window) all the Button information. You may wish to `#include <iomanip>` for some formatting tools, but you do not have to. For an *acceptable* print output, see below.

```
Panel:    Panel 1
Position: 10, 10
Size:     20, 50
```

5.6 The CuWindow Class

The `CuWindow` class handles the X11 logic for making a display, opening a window and getting a graphics context for drawing. It also maintains a statically allocated array of (non-overlapping) `Panel` objects.

1. Member variables:
   (a) `int width, int height`: the current width and height of the window in pixels. Note, for this assignment we will not worry about what happens if a user resizes the window.
   (b) `string name`: The name of the window (which should be displayed at the top)
   (c) A statically allocated array of `Panel` objects of size MAX_COMPONENTS.
   (d) An int that keeps track of the current number of `Panels` stored (set to 0 initially).
   (e) An `RGB` member for the background colour of the window.
2. In addition, these member variables are necessary to maintain and draw on an X11 window:
   (a) `Display* display`: Connection to the X server.
   (b) `Window window`: To store the XID of the window that we opened.
   (c) `GC gc`: A graphics context (so we can draw on the window).
3. Constructors:
   (a) Two constructors that take `name, width, height, background` as arguments and initializes the member variables appropriately. The first should pass in background as an `RGB` object, the second as a `CuColour` value. The constructor should also open a display, a window, and create a graphics context.
   (b) You should also have a destructor. This should free the graphics context, destroy the window, and close the display.

4. Member functions:

   (a) You should make getters and setters as needed.

   (b) `bool addPanel` - This function should take a `Panel` as an argument (passed by reference). If this `Panel` does not overlap any other `Panel` in the `CuWindow`, AND it does not extend outside of the `CuWindow` boundaries, then add this `Panel` to the back of the `Panel` array. Return true if the `Panel` is added, and false otherwise.

   (c) `bool removePanel (string id)` - This function should find the `Panel` with the given id and remove it from the array. It should "close the gap" in the array. That is, consider panels p1, p2, p3, and p4 in an array in the following indices: 0:p1, 1:p2, 2:p3, 3:p4. If we remove p2, then the resulting array should look like 0:p1, 1:p3, 2:p4.

   (d) `getPanel(string id)`: This function should return a pointer to the `Panel` with the given `id`, or else `nullptr` if no such `Panel` exists.

   (e) A `void draw();` function. You should first fill the window with a rectangle to "blank" everything out and provide a background colour (using the `RGB` member variable for the colour). Then you should draw all the `Panels` and their contents onto the window.

   `NOTE:` X11 does not synchronize by default. Thus, if there are changes being made to `CuWindow` and we attempt to draw, it may not be rendered properly. It is HIGHLY recommended that at the top of this draw function, before doing anything else, you sleep a bit so that any changes to `CuWindow` can be completed. Run the command `usleep(100000)` as the very first line of the `draw` function. You will also need `#include <unistd.h>` at the top of your file.

   (f) A `print` function. This should print (to the console, not to the window) the name of the window and the number of `Panels`.

   (g) A `printPanels` function. This should print out all the `Panels`.

   (h) A `printPanelButtons` function. This should print out all the `Buttons` from all the `Panels`.

6 Grading

The marks are divided into three main categories. The first two categories, **Requirements** and **Constraints** are worth 34 and 16 marks respectively. The third category, **Deductions** is where you are penalized marks.

6.1 Specification Requirements

These are marks for having a working application (even when not implemented according to the specification, within reason). These are the same marks shown in the test suite, repeated here for convenience. Marks are awarded for the application working as requested.

The test script provides a mark out of 34. If you have implemented everything correctly, this will *probably* be your mark for these sections. However, you are still responsible for, and may be penalized for, any errors the test suite does not catch, or any drastic departure from the specification (such as using outside libraries). We reserve the right to modify the mark given by the test script in these cases.

General Requirements

- All marking components must be called and execute successfully to earn marks.
- All data handled must be printed to the screen as specified to earn marks.

Application Requirements: 28 marks

- 1. [3 marks] Test Buttons
- 2. [3 marks] Test Panels
- 3. [5 marks] Test add, remove, and print Buttons in Panel
- 4. [5 marks] Test add, remove, and print Panels in CuWindow
- 5. [2 marks] Test add, remove, and print Buttons and Panels in CuWindow
- 6. [6 marks, manual inspection] Test render window, see examples below.
  - a) [1 mark] Panels are drawn correctly
  - b) [2 marks] Buttons are drawn correctly (1 mark each Panel) - ignore the text for this mark, just look at the rectangles.
  - c) [2 marks] Text is rendered correctly (1 mark each Panel).
  - d) [1 marks] Upper Panel is removed correctly, lower Panel still drawn correctly.
- 7. [4 marks, manual inspection] Student tests. You should test different `Button` colours, different `CuWindow` colours, and different placements of `Panels` and `Buttons`, removing `Panels`, etc. There should be at least 4 different configurations of `Panels` and `Buttons` in your test. Each configuration must have at least 2 `Panels` and 2 `Buttons` (if you are testing the remove function, then after removal you may have fewer `Panels` and `Buttons`). Be sure to output (to `cout`) what each part is testing for.

Requirements Total: 28 marks

Bonus Requirements: 2 marks You may earn up to 2 style points as bonus. In the example shown, the buttons are filled in rectangles, which gets you full marks. However, the lettering is centered in the `Button`, which would earn you 1 bonus mark.

It is not enough to hard code "Button 1" in the center of the button. The text must be centered for any size button and any label. The TAs will check you code for this.

You can earn additional bonus for making the buttons appear protruded, or for rounding the corners, or anything else that improves the appearance. Each improvement earns 1 mark, and you may earn up to 2 marks total.

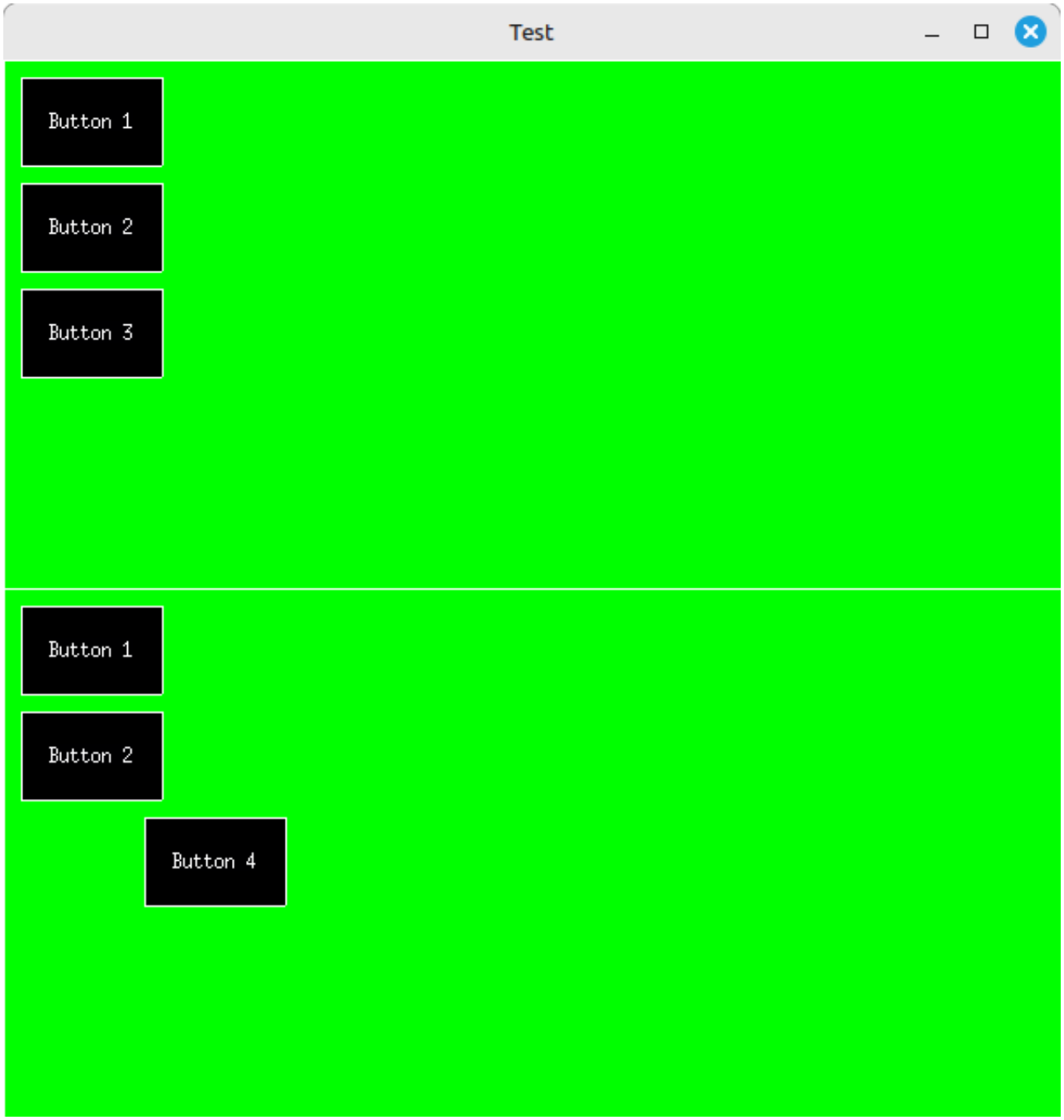


Figure 1: Properly rendered window.

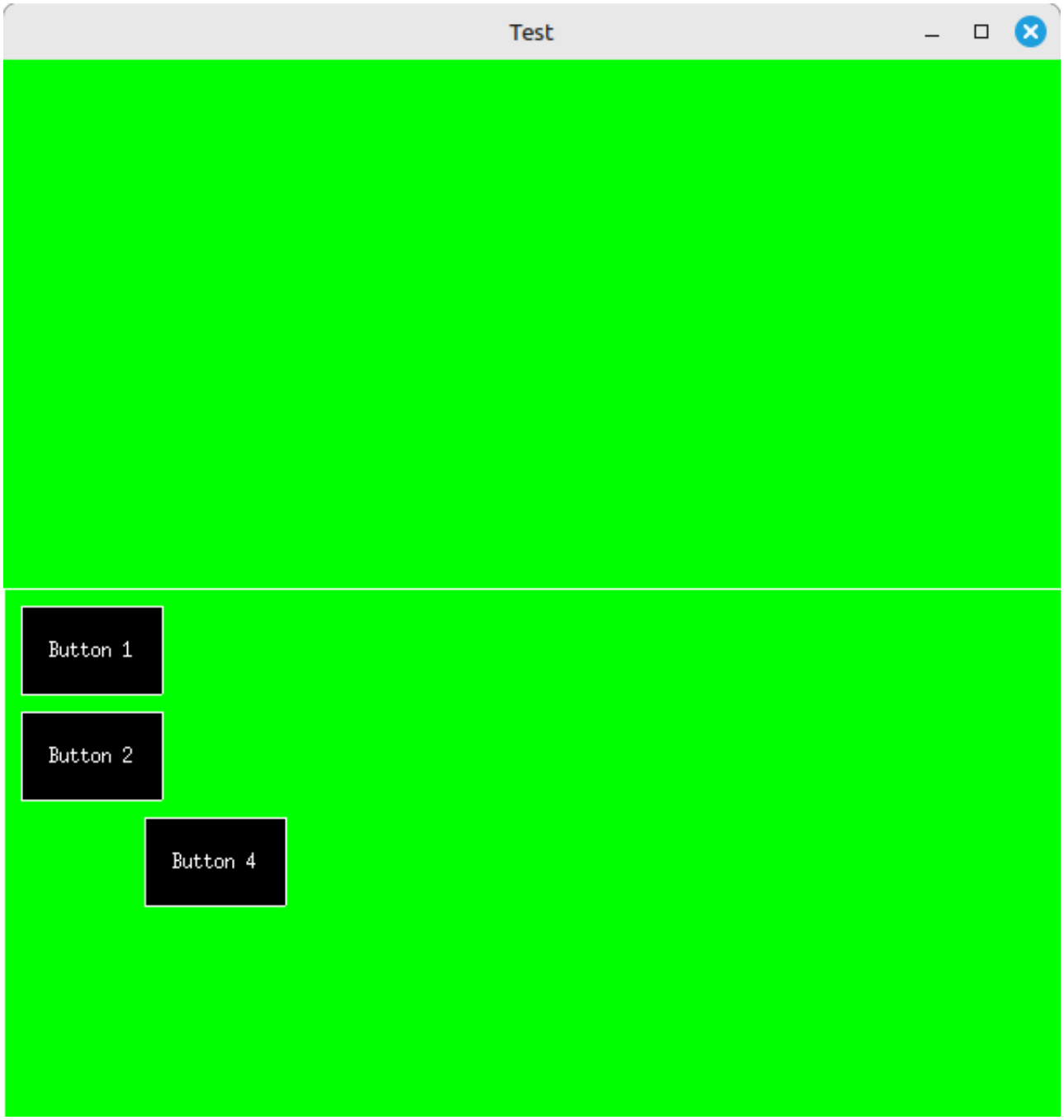


Figure 2: Properly rendered window with upper panel removed.

6.2 Constraints

The previous section awards marks if your program works correctly. In this section marks are awarded if your program is written according to the specification and using proper object oriented programming techniques. This includes but is not limited to:

- Proper declaration of member variables (correct type, naming conventions, etc).
- Proper instantiation and initialization of member variables (statically or dynamically).
- Proper instantiation and initialization of objects (statically or dynamically).
- Proper constructor and function signatures.
- Proper constructor and function implementation.
- Proper use of arrays and data structures.
- Passing objects by *reference* or by *pointer*. Do not pass by value.
  - **Note:** For this assignment only, `strings` may be passed by value.
- Proper error checking - check array bounds, data in the correct range, etc.
- Reasonable documentation (remember the best documentation is expressive variable and function names, and clear purposes for each class).

6.2.1 Constraint marks:

- 2 marks: Proper implementation of the `RGB` class.
- 2 marks: Proper implementation of the `Button` class.
- 2 marks: Proper implementation of the `Panel` class.
- 4 marks: Proper implementation of the `CuWindow` class.

Constraints Total: 10 marks

6.3 Deductions

The requirements listed here represent possible deductions from your assignment total. In addition to the constraints listed in the specification, these are global level constraints that you must observe. For example, you may only use approved libraries, and your programming environment must be properly configured to be compatible with the virtual machine. This is not a comprehensive list. Any requirement specified during class but not listed here must also be observed and may be penalized if done incorrectly.

6.3.1 Packaging and file errors:

1. 5%: Missing README
2. 10%: Missing Makefile (assuming this is a simple fix, otherwise see 4 or 5).
3. up to 10%: Failure to use proper file structure (separate header and source files for example), but your program still compiles and runs
4. up to 50%: Failure to use proper file structure (such as case-sensitive files and/or Makefile instructions) that results in program not compiling, but is fixable by a TA using reasonable effort.

5. up to 100%: Failure to use proper file structure or other problems that severely compromise the ability to compile and run your program.

As an example, submitting Windows C++ code and Makefile that is not compatible with the Linux VM would fall under 4 or 5 depending on whether a reasonable effort could get it running.

6.3.2 Incorrect object-oriented programming techniques:

- Up to 10%: Substituting C functions where C++ functions exist (e.g. don't use `printf`, do use `cout`).
- Up to 25%: Using smart pointers.
- Up to 25%: Using global functions or global variables other than the `main` function and those functions and variables expressly permitted or provided.

6.3.3 Unapproved libraries:

- Up to 100%: The code must compile and execute in the default course VM provided or Openstack. It must NOT require any additional libraries, packages, or software besides what is available in the standard VM or Openstack.
- Up to 100%: Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*

Assignment Total (Requirements and Constraints): 38