wxErlang - Speeding Up

This is a sequel to my earlier, *Getting Started*. There, we went through what wxErlang is, ie the `wx` library in the Erlang distribution.

It is, yes, an Erlang binding to the core wxWidgets GUI library, but it uses a different approach to event handling, making good use of Erlang's message passing paradigm. In particular, it provides the `wx_object` abstraction, which allows us to treat a widget, simple or composed, as an OTP `gen_server`. We saw how to use the `wx_object`, how to layout widgets by means of sizers and how to shape the provided widgets to our liking.

In this piece we will go a step further. We will learn how to draw and we will use a part of what we'll learn to build a chess board.

We will then construct a system where two players, each sitting on a computer of his own, will use the chess board and a modified version of the chess clock we developed in *Getting Started* to play a complete game.

In so doing, we will learn more about some of the peculiarities of wxErlang and also something about how distributed Erlang works and how it can be a powerful complement to our GUI endeavours.

In *Device Contexts*, we will learn what device contexts are and how to draw line and various shapes. We will learn how these functions are affected by clipping and logical functions. We'll also see the peculiarity of paint and size events and how to handle them. We'll see some utility fucntions like the use of the clipboard, images and bitmaps.

In *Chessboard* we will learn more about images and bitmaps and mouse events. We will see what flickering is and how we can minimize it.

In *Chessboard revisited*, we will talk about cursor shapes and we will construct the chess board out of 64 processes, one for each square of the chess board.

In *Chess Player*, we will put the board we developed into a frame together with the two clocks and the button to say the player has moved the piece. In doing this, we will go through some quirks with static texts and learn how to handle keyboard events.

In *Playing the game*, we will learn about menus. We will get to play a game between two players. This requires distributed Erlang which we will see very briefly as this is about wxErlang and not distributed Erlang. Nevertheless, if you are not very familiar with distributed systems, I encourage you to go through the code to see how message exchanges between processes on remote nodes can be employed in simple applications like this, and by extension in complex ones.

I'm really curious to know if people who have experience with technologies other than Erlang find this prototype too complex or simpler than if it had been done in the technologies familiar to them. So please share your considerations. Thanks.

Unlike *Getting Started* I have not put all of the code in the text as it would have been too bulky. Instead, you can download it at github. It is divided up into several directories. Each directory should be self-sufficient, so many files are simply replicated between one directory and the other.

Table of Contents

Device Contexts

Let's say we want to draw something, for example a chart, a graph, or to compose some picture. Can we do that with wxErlang?

As we saw in *Getting Started*, one easy way to find it out would be to go through the examples in `wx:demo/0`. Another would be to go through the wxWidgets documentation. There we will find a rough guide to GUI application development. I think it is instructive to go through it. It answers our question on if and how we can do drawings, and we can also spot where wxErlang differs from wxWidgets:

Writing a wxWidgets application: a rough guide

To set a wxWidgets application going, you will need to derive a wxApp class and override wxApp::OnInit.

We use wx:new() to initialize wxWidgets. We could actually make an Erlang application and have wx_objects run as supervised processes.

An application must have a top-level wxFrame or wxDialog window. Each frame may contain one or more instances of classes such as wxPanel, wxSplitterWindow or other windows and controls.

A frame can have a wxMenuBar, a wxToolBar, a status line, and a wxIcon for when the frame is iconized.

A wxPanel is used to place controls (classes derived from wxControl) which are used for user interaction. Examples of controls are wxButton, wxCheckBox, wxChoice, wxListBox, wxRadioBox, wxSlider.

Instances of wxDialog can also be used for controls and they have the advantage of not requiring a separate frame.

Instead of creating a dialog box and populating it with items, it is possible to choose one of the convenient common dialog classes, such as wxMessageDialog and wxFileDialog.

You never draw directly onto a window - you use a *device context* (DC). wxDC is the base for wxClientDC, wxPaintDC, wxMemoryDC, wxPostScriptDC, wxMemoryDC, wxMetafileDC and wxPrinterDC. If your drawing functions have **wxDC** as a parameter, you can pass any of these DCs to the function, and thus use the same code to draw to several different devices. You can draw using the member functions of **wxDC**, such as wxDC::DrawLine and wxDC::DrawText. Control colour on a window (wxColour) with brushes (wxBrush) and pens (wxPen).

There. That's the answer. We can draw, and the means to draw is a device context.

To intercept events, you add a DECLARE_EVENT_TABLE macro to the window class declaration, and put a BEGIN_EVENT_TABLE ... END_EVENT_TABLE block in the implementation file. Between these macros, you add event macros which map the event (such as a mouse click) to a member function. These might override predefined event handlers such as for wxKeyEvent and wxMouseEvent.

No, we don't do that in wx. Events get sent to Erlang processes as Erlang messages. Unless you decide to use callback functions.

Most modern applications will have an on-line, hypertext help system; for this, you need wxHelp and the wxHelpController class to control wxHelp.

There is no help controller as such, though we can intercept wxHelp events just as any other event and deal with them. The launchDefaultBrowser/1,2 functions in wx_misc module can be used to open a URL. See wx:demo/0 for an actual example.

GUI applications aren't all graphical wizardry. List and hash table needs are catered for by wxList and wxHashMap. You will undoubtedly need some platform-independent file functions, and you may find it handy to maintain and search a list of paths using wxPathList. There's a miscellany of operating system and other functions.

Yes, that's true, but we don't need any of these from wxWidgets. Erlang/OTP already has, IMHO, a richer set of functions. As we will see, Erlang's constructs are extremely powerful.

Ok, so we need a *device context*, but what is it?

The wxWidgets documentation tells us:

A wxDC is a *device context* onto which graphics and text can be drawn. It is intended to represent a number of output devices in a generic way, so a window can have a device context associated with it, and a printer also has a device context. In this way, the same piece of code may write to a number of different devices, if the device context is used as a parameter.

Notice that wxDC is an abstract base class and can't be created directly, please use wxPaintDC, wxClientDC, wxWindowDC, wxScreenDC, wxMemoryDC or wxPrinterDC.

To be honest, I don't understand much from this. Some place in the wxWidgets documentation I have also seen device contexts explained as *surfaces* on which you can draw.

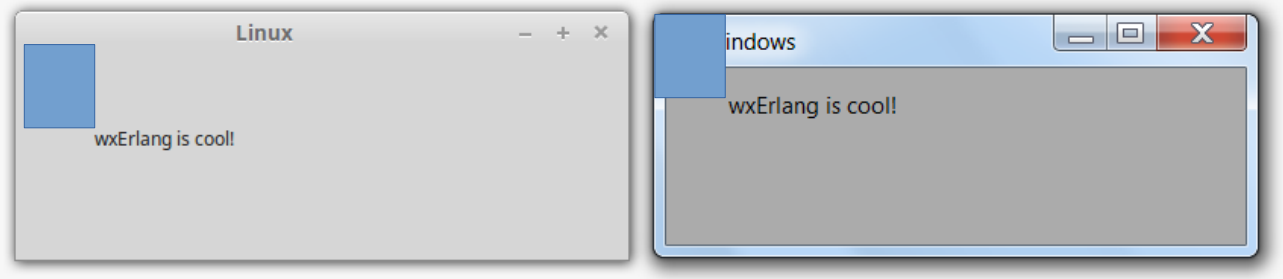
I try to look at it this way: **`wxDC`** is an API that abstracts out the ways to draw on objects - *devices* - such as frames, screens, printers and bitmaps. Instead of extending the APIs of *those* devices, wxWidgets associates other objects - *device contexts* - to them, to take care of the API implementation. In other words, the objects in `wxDC` class hierarchy are being delegated the job of drawing on the relative objets. So for the *Window device*, we have a `wxWindowDC`. For a *Printer device*, we have `wxPrinterDC`, and so on.

One important *device* is a **`wxBitmap`**, which represents a pixmap in memory. The associated *device context* is the **`wxMemoryDC`**. Actually, any DC can be *imagined* as a pixmap, even though the associated *device* is difficult to imagine as such. (eg a postscript printer).

Enough talk. We need to get some hands on experience. Off to the shell!

```
1> wx:new().
{wx_ref,0,wx,[]}
2> Frame = wxFrame:new(wx:null(), -1, "Linux").
{wx_ref,35,wxFrame,[]}
3> wxFrame:show(Frame).
true
4> DC = wxWindowDC:new(Frame).
{wx_ref,36,wxWindowDC,[]}
5> wxDC:drawText(DC, "wxErlang is cool!", {50,50}).
ok
```

And yes, the message got printed, both in Linux (Mint), as well as in Windows (7).



With a difference, though. The position of the text is not the same as is evident by the blue rectangle I have placed on the corner to figure out what the difference is.

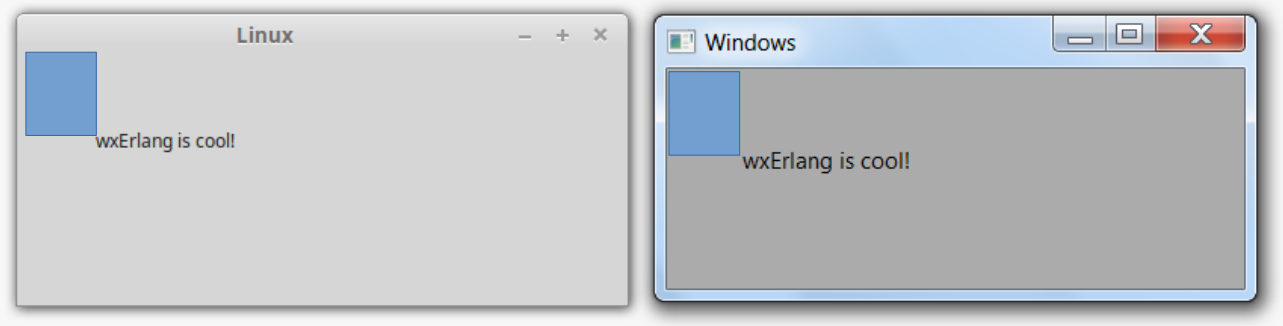
Let's have a look at the wxWidgets documentation:

> A wxWindowDC must be constructed if an application wishes to paint on the whole area of a window (client and decorations). This should normally be constructed as a temporary stack object; don't store a wxWindowDC object.
>
> To draw on a window from inside **OnPaint**, construct a wxPaintDC object.
>
> To draw on the client area of a window from outside **OnPaint**, construct a wxClientDC object.
>
> To draw on the whole window including decorations, construct a wxWindowDC object (Windows only).

A bit ambiguous, but practically, when you use **`wxWindowDC`** in Windows, the coordinate system origin is at the top-left of the entire window, including any decorations or borders. However, at least with the distribution I have, I cannot get to actually *draw* on the decorations. In Linux, on the other hand (again in my distribution), it is at the top-left of the *client* area.

Notice, though, that you can construct a wxPaintDC object from inside OnPaint, or in wx, from ***inside a callback***.

If we had used a **`wxClientDC`**, instead of a `wxWindowsDC`, we would have gotten:



And this time the position within the client area is the same in both platforms. In other words, if you want consistent behaviour across both platforms, better use `wxClientDC`.

The text got drawn in the default font and with the default text foreground colours. This can be changed. The font with **`wxDC:setFont/2`**, specifying a **`wxFont`**. We already saw in *Getting Started* how to create `wxFont` objects. The colour is a **`wx_colour`** object, an RGB tuple, specified in **`wxDC:setTextForeground/2`**.

If you are curious, you can query the defaults for your system with the functions in the **wxSystemSettings** module. The wxWidgets documentation for the class has more, and understandable information.

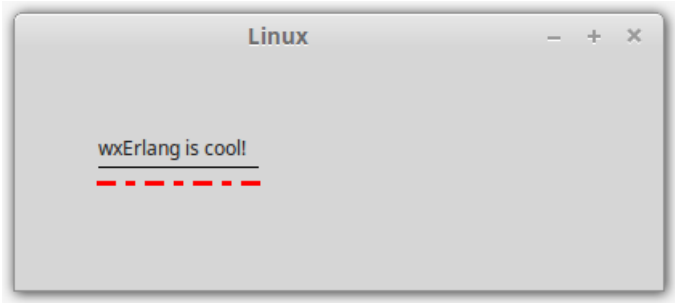
Drawing Lines

We can obviously draw lines. Let's give it a try:

```
11> wxDC:drawLine(DC, {50,70}, {150,70}).
ok
```

Lines are drawn using pens, instances of **wxPen**. The line we just drew will have been drawn with the default pen. We can use a pen to our liking by setting it as the pen in the DC:

```
7> Pen = wxPen:new({255,0,0}, [{width, 3}, {style, 104}]).
{wx_ref,37,wxPen,[]}
8> wxDC:setPen(DC, Pen).
ok
9> wxDC:drawLine(DC, {50,80}, {150,80}).
ok
```



The wx documentation tells us how to construct one:

```
new(Colour, Options::[Option]) -> wxPen()
        Colour = wx:wx_colour()
        Option = {width, integer()} | {style, integer()}
```

The possible values for style are indicated in the wxWidgets documentation:

wxSOLID	Solid style.
wxTRANSPARENT	No pen is used.
wxDOT	Dotted style.
wxLONG_DASH	Long dashed style.
wxSHORT_DASH	Short dashed style.
wxDOT_DASH	Dot and dash style.
wxSTIPPLE	Use the stipple bitmap.
wxUSER_DASH	Use the user dashes: see wxPen::SetDashes.
wxBDIAGONAL_HATCH	Backward diagonal hatch.
wxCROSSDIAG_HATCH	Cross-diagonal hatch.
wxFDIAGONAL_HATCH	Forward diagonal hatch.
wxCROSS_HATCH	Cross hatch.
wxHORIZONTAL_HATCH	Horizontal hatch.
wxVERTICAL_HATCH	Vertical hatch.

As we saw in *Getting Started*, these constants are defined in macros in the `wx.hrl` file of your Erlang distribution, and 104 corresponds to `wxDOT_DASH`.

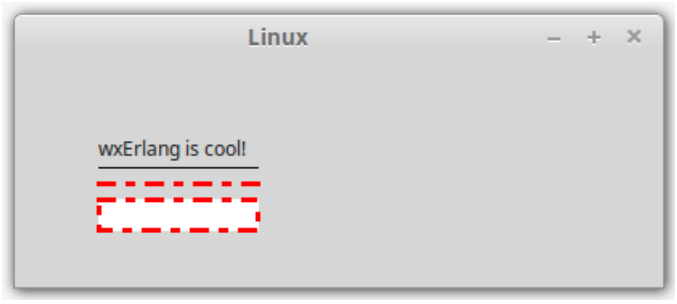
Do heed the warning that accompanies these definitions:

> Different versions of Windows and different versions of other platforms support *very* different subsets of the styles above - there is no similarity even between Windows95 and Windows98 - so handle with care.

Drawing Shapes

We can draw shapes, such as rectangles. Here's one:

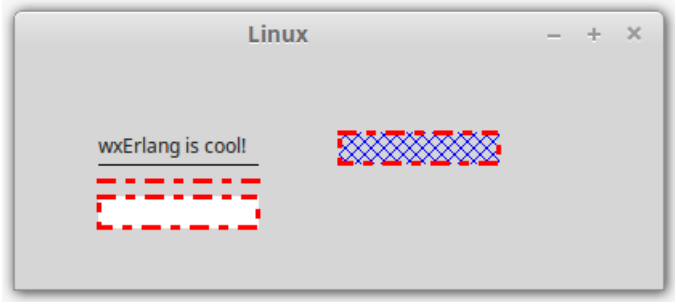
```
10> wxDC:drawRectangle(DC, {50,90,100,20}).
ok
```



The pen is still the one we had set.

Whereas lines are drawn with a *pen*, shapes are filled with a *brush*, an instance of **`wxBrush`**. The rectangle we just drew got filled with the default brush. We can construct brush to our liking, and use that instead:

```
11> Brush = wxBrush:new({0,0,255}, [{style, 112}]),
11> wxDC:setBrush(DC, Brush),
11> wxDC:drawRectangle(DC, {200,50,100,20}).
ok
```



From the wx documentation, brushes can be constructed thus:

```
new(Colour) -> wxBrush()
        Colour = wx:wx_colour()
new(StippleBitmap) -> wxBrush() when
        StippleBitmap::wxBitmap:wxBitmap().
new(Colour, Options::[Option]) -> wxBrush()
        Colour = wx:wx_colour()
        Option = {style, integer()}
```

The style, according to the wxWidgets documentation, can be one of the constants:

wxTRANSPARENT	Transparent (no fill).
wxSOLID	Solid.
wxSTIPPLE	Uses a bitmap as a stipple.
wxBDIAGONAL_HATCH	Backward diagonal hatch.
wxCROSSDIAG_HATCH	Cross-diagonal hatch.
wxFDIAGONAL_HATCH	Forward diagonal hatch.
wxCROSS_HATCH	Cross hatch.
wxHORIZONTAL_HATCH	Horizontal hatch.
wxVERTICAL_HATCH	Vertical hatch.

We used `wxCROSSDIAG_HATCH` (112).

Note the style **`wxTRANSPARENT`** (106). It means no fill. Not to be confused with the alpha channel of colours. We haven't referred to the Mac platform so far, but it does have support for alpha channel, unlike Windows, or Linux. We'll see later that the alpha channel, aka transparency, *is* supported for bitmaps, but not for pens and brushes.

Clipping and Logical Functions

When drawing, it's important to be aware of two things: *clipping* and *logical functions*.

Clipping means confining any drawing action to a region, a subset of the entire drawing surface. The simplest region is a rectangle, but it can be a union of any of number of shapes. If we set the clipping region of a device context, any drawing will be limited to that region.

A logical function, on the other hand, defines how the pixels we draw, do actually get drawn, taking into consideration what is already drawn. Assuming *src* to be the pixel we want to draw and *dest* the pixel already drawn in the device context, the available logical functions and what they do is documented in the wxWidgets documentation:

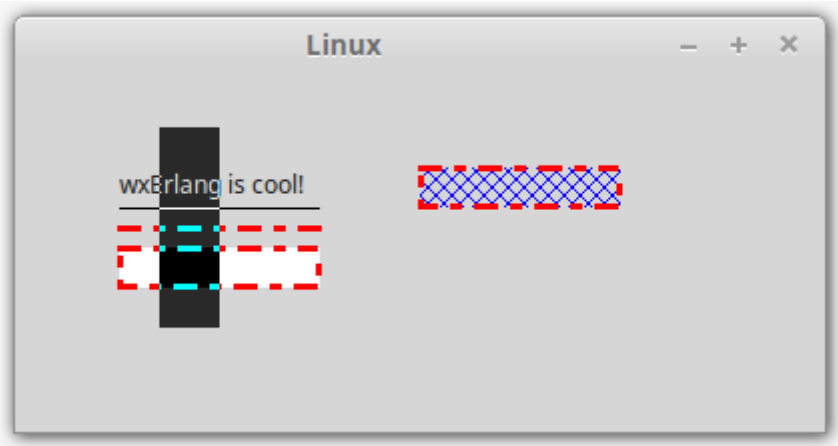
```
wxAND                   src AND dst
wxAND_INVERT            (NOT src) AND dst
wxAND_REVERSE           src AND (NOT dst)
wxCLEAR                 0
wxCOPY                  src
wxEQUIV                 (NOT src) XOR dst
wxINVERT                NOT dst
wxNAND                  (NOT src) OR (NOT dst)
wxNOR                   (NOT src) AND (NOT dst)
wxNO_OP                 dst
wxOR                    src OR dst
wxOR_INVERT             (NOT src) OR dst
wxOR_REVERSE            src OR (NOT dst)
wxSET                   1
wxSRC_INVERT            NOT src
wxXOR                   src XOR dst
```

The default is `wxCOPY` (5) and what it does is to paint the source pixel, without taking into consideration what was already drawn. `wxINVERT` (2) doesn't take into consideration the source and just toggles what was present: black to white and white to black, etc. `wxXOR` (1) is generally used to

combine the source and destination pixels in such a way that drawing the source once again gives you back what was originally there.

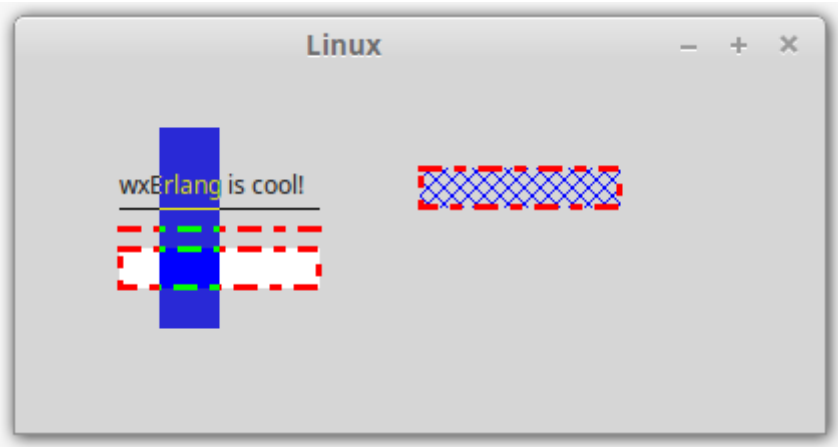
Let's try these concepts. We'll set a clipping rectangle, set the pen to a solid black line, 1 pixel wide, and set the brush to a solid yellow. Then we'll draw a big square using the logical function `wxINVERT` (2). We will see just the clipping reagion painted with all colours inverted.

```
12> wxDC:setClippingRegion(DC, {70, 30, 30, 100}),
12> wxDC:setLogicalFunction(DC, 2),                          %%   2 = wxINVERT
12> YellowBrush = wxBrush:new({255,255,0}, [{style,100}]), %% 100 = wxSOLID
12> wxDC:setBrush(DC, YellowBrush),
12> wxDC:drawRectangle(DC, {0,0,200,200}).
ok
```



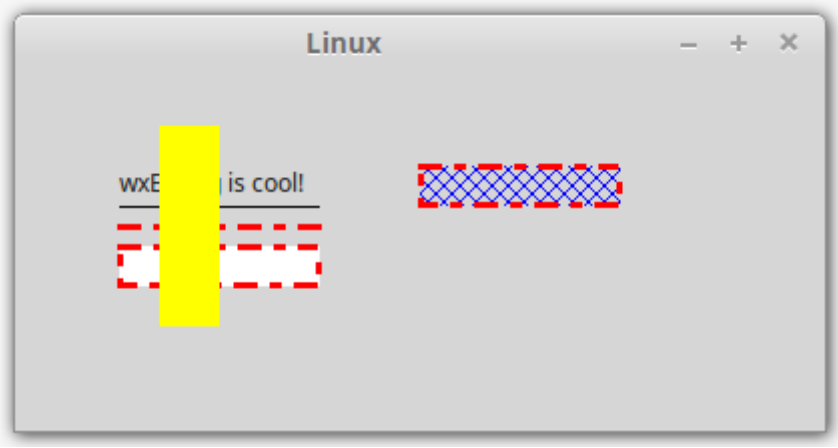
The black rectangle is the clipping region. The things already drawn there have inverted colours. If we draw the square again, the original drawing will be restored.

If, on the other hand, we try with the logical function `wxXOR` (1), we should get:



Remember, our brush is yellow! In this case too, drawing the square again should restore the original drawing.

Finally, just to demonstrate that we are not cheating, we set the logical function back to the default, i.e. `wxCOPY` (5) and draw the square again:



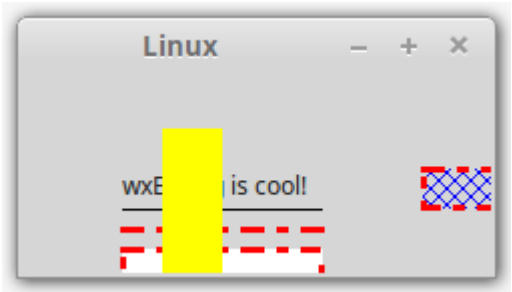
And this time around, we get the clipping region painted yellow, as expected.

Paint Event

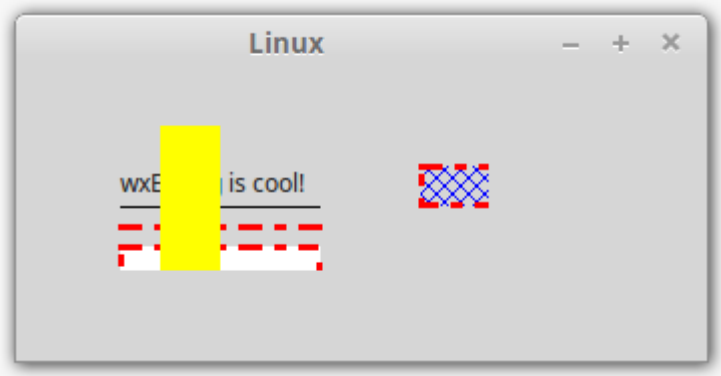
One word of caution. If the window is hidden or minimized or is not "shown", ***nothing will actually get drawn***! It is ok if the window is not visible because it is *behind* other windows, but not otherwise. wxWidgets takes the load off you to determine when things need to be redrawn. e.g. after you un-minimize a window; it triggers a **`wxPaint`** event. The event is handled by widgets *internally* to get their stuff done, but you *can* subscribe to it and wxWidgets will then leave the handling to you so you can redraw your stuff on your own.

The lesson here is that drawing things in wxWidgets is not a draw-and-forget affair. ***You must be prepared to redo what has already been done!***

To demonstrate this, try enlarging the frame. Our drawing should not be effected. Now try reducing the size so that the frame is smaller than the drawing.



When you enlarge it again, the part that had got covered by the frame will no longer be there:



Even worse, if you minimize the frame and then un-minimize it, the drawing will be lost.

So, how do we deal with this? Well, we must keep track of what we draw and redraw it whenever the paint event is triggered. To see when *that* happens, let's subscribe to the event and in the handler just print out the fact that the event had been delivered. We'll need to do this in a `wx_object`.

```
init([]) ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), ?wxID_ANY, "paint event capture"),
    wxFrame:connect(Frame, paint),  %% subscribing to the paint event
    wxFrame:show(Frame),
    {Frame, #{frame => Frame}}.

handle_event(#wx{} = Event, State) ->
    io:format("got ~p~n", [Event]),
    {noreply, State}.
```

If we compile[1] and run it, we immediately get a paint event:

```
got {wx,-2007,{wx_ref,41,wxFrame,[]},[],{wxPaint,paint}}
```

This comes from `wxFrame:show(Frame)`.

Now move the frame around. As long as the entire frame is visible, we should get no events.

Now move the frame out of the screen so that a part of it is not visible. We still shouldn't get any paint event.

Now move the frame back to the screen. We should get some paint events.

Now minimize the frame. Again, no events.

Un-minimize it. Yes, we get paint events alright.

Resize the frame. We should get paint events when we enlarge the window, but not when we make it smaller.

Most of the times you don't really need to repaint if the window is made smaller or a part of it gets hidden becuse you don't really care. However, if we want to keep the entire window content visible no matter what size the window is, this strategy is not applicable. Our chessboard will be one such example. If we are playing the game, we need to see the entire board all the time.

1 drawing/paint_event_capture.erl

There are at least two ways to deal with the resizing issue. One[2] is to set the frame style to ***full repaint on resize***:

```
Frame = wxFrame:new(wx:null(), ?wxID_ANY, "paint event capture",
     [{style, ?wxDEFAULT_FRAME_STYLE bor ?wxFULL_REPAINT_ON_RESIZE}]),
```

This instructs the frame to do a full repaint of the window whenever it is resized.

Size Event

The other way[3] is to intercept the size event, **`wxSize`**, which gets generated every time the window is resized, and force the generation of a paint event by means of **`wxWindow:refresh/1`**:

```
handle_event(#wx{event = #wxSize{}}, State = #{frame := Frame}) ->
    wxWindow:refresh(Frame),
    {noreply, State};
```

Event Chaining

A word of caution, though. You have to be extra careful when handling size events because they are one of a class of events that are *chained*. When a window handles the size event, it passes it on to all of its contained windows after doing whatever it needs to do. If we handle the event ourselves, we break that chaining. This could be exactly what we want, but if it is not, we have to pass the event on by using the **`skip`** option when subscribing to the event.

```
wxFrame:connect(Frame, size, [{skip, true}])
```

Here's a small experiment to demonstrate event chaining. We create[4] a `wx_object` with a frame and three buttons placed in a vertical box sizer with instructions to expand them to the size of the frame.

```
    MkButton = fun(TheFrame, TheSizer, TheLabel) ->
                   B = wxButton:new(TheFrame, ?wxID_ANY, [{label, TheLabel}]),
                   wxSizer:add(TheSizer, B, [{flag, ?wxEXPAND}])
               end,
    [MkButton(Frame, Sizer, L) || L <- ["One", "Two", "Three"]],
```

An aside. When we add a widget to a sizer, it returns a **`wxSizerItem`** object which contains information about the widget in the sizer. According to wxWidget documentation:

> The wxSizerItem class is used to track the position, size and other attributes of each item managed by a wxSizer.

Back to our expirement, we subscribe to the size event without the skip option and do nothing in the size event handler.

Next, we make our widget handle a *resize* message, increasing its width by a 100 pixels:

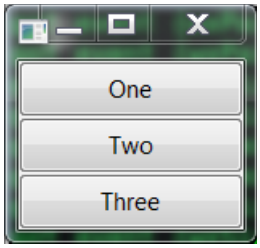
```
handle_info(resize, State = #{frame := Frame}) ->
    {W, _} = wxFrame:getSize(Frame),
    wxFrame:setSize(Frame, W + 100, -1),
    {noreply, State};
```

---

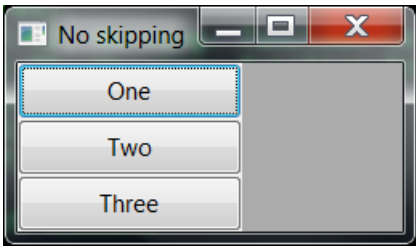
2 drawing/paint_event_capture_full_repaint.erl
3 drawing/force_paint_by_size_capture.erl
4 drawing/resize_no_skip.erl

Tip: in wxWidgets, if you don't want to change the value of a parameter, such as the height in this case, you just specify it as -1. In the example above, the height of the frame will remain unchanged.

Now we are set. If we start the `wx_object`, we'll get a frame similar to:



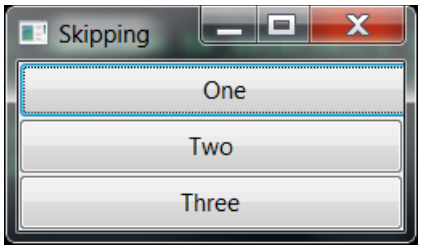
If we now send the window the `resize` message, the width of the frame is increased by a 100 pixels, but the buttons remain where they were *even though we had instructed the sizer to expand the buttons*.



The reason is that we blocked the size message from propagating to the frame's children. If, instead, we specify the skip option in the subscription to the event[5]:

```
wxFrame:connect(Frame, size, [{skip, true}]),
```

the resize message will have the desired effect: all the buttons will resize to fit the frame:



Synchronous Event Handling

Since we'll be dealing with the paint event, let's have a look at the wxWidgets documentation for it. We notice there is a remark:

> Note that in a paint event handler, the application must *always* create a **wxPaintDC** object, even if you do not use it. Otherwise, under MS Windows, refreshing for this and other windows will go wrong.

Oh well! This is quite a warning. It actually puts a strong constraint on how to handle paint events. Since, as we saw earlier, a `wxPaintDC` can only be created from *within a callback*, this rules out

5 drawing/resize_and_skip.erl

handling the paint event as an event, with the added concern, as we saw in *Getting Started*, that we then do not have access to the `wx_object` state.

Before despairing, let's go back to the `wx` documentation and have another look at the **`wxEvtHandler:connect/3`** specification:

```
connect(This::wxEvtHandler(), EventType::wxEventType(), Options::[Option]) -> ok
        Option = {id, integer()} | {lastId, integer()} | {skip, boolean()} | callback | {callback,
function()} | {userData, term()}
```

Here we see that it also accepts a `callback` option, *without* any associated function. Now what could that mean? Turns out it is actually an **undocumented feature**. It converts wxWidget's event callback to an event that must be handled *synchronously* within `wx_object`. With **`handle_sync_event/3`,** to be precise.

We can run a small experiment to verify this. We subscribe to an event in a `wx_object` specifying the `callback` option, without the callback function. When we start the `wx_object` and generate the event we subscribed to, there will be no handler for it and the `wx_object` will crash with a printout of what it expected.

So, for example, if we subscribe the frame of our `wx_object` to the paint event in this way[6], we will see the the following crash report:

```
2>
=ERROR REPORT==== 11-Jan-2018::19:17:27 ===
wxe_server:288: Callback fun crashed with {'EXIT, undef, [{unhandled_callback,
                                                           handle_sync_event,
                                                           [{wx,-2006,
                                                             {wx_ref,35,
                                                              wxFrame,[]},
                                                             [],
                                                             {wxPaint,paint}},
                                                            {wx_ref,39,
                                                             wxPaintEvent,[]},
                                                            #{frame =>
                                                               {wx_ref,35,
                                                                wxFrame,[]}}],
                                                          ...
```

In other words, the `wxe_server` tried to invoke the callback `handle_sync_event` with three arguments: the `wx` record, a reference to the paint event and the `wx_object` state.

We see no mention of this in the wx documentation, but if we inspect the code for *`canvas`* in `wx:demo/0`, we will see that this function *is* indeed employed.

In the next section we will use `full_repaint_on_resize` style for the frame and handle the paint event with a synchronous event handler in which we will be able to construct a `wxPaintDC` to do the job.

Images and Bitmaps

We have learnt how to draw shapes and lines, but we still need to see how to deal with images.

6 drawing/unhandled_callback.erl

The DCs in wxWidgets don't deal with images directly, but through bitmaps. Bitmaps are created in memory by specifying their size:

```
wxBitmap:new(Width, Height).
```

Or they can be loaded from images in files, as long as a handler for the image format is installed in wxWidgets. The ususal suspects are probably installed in your distribution.

```
wxBitmap:new(ImageFilename).
```

There are other ways too, like creating the bitmap from a bitstring or a `wxImage` object, but we'll just stick to these variants.

Once loaded, or ready, the bitmap can be drawn on to a DC with the **`wxDC:drawBitmap/3,4`** or the **`wxDC:blit/5,6`** functions.

When we create the bitmap, it is just some area in memory with random pixels, unless we created it out of an image on file. As with any other drawing, drawing on bitmaps requires a DC, the **`wxMemoryDC`**. You either create the DC using the bitmap you want to work on:

```
wxMemoryDC:new(wxBitmap()) -> wxMemoryDC()
```

Or you select the bitmap into it:

```
wxMemoryDC:selectObject(wxMemoryDC(), wxBitmap()) -> ok
```

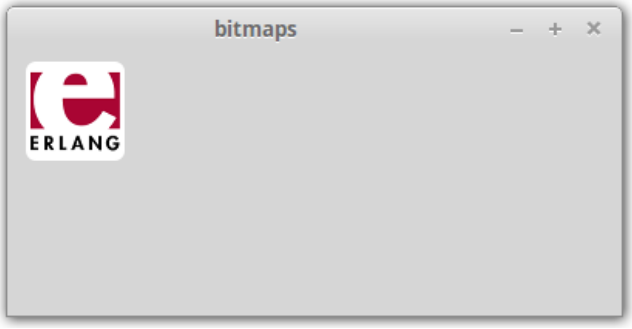
You then use the DC as any other DC and when you are done, ***you destroy it*** to get back your painted bitmap.

Here are some extracts from the wxWidgets documentation on `wxMemoryDC`:

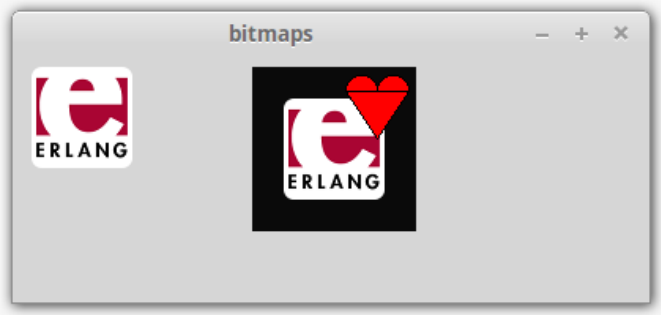
> A memory device context provides a means to draw graphics onto a bitmap.
>
> A bitmap must be selected into the new memory DC before it may be used for anything.

And here's an experiment in which we use the Erlang logo from our Erlang distribution:

```
1> wx:new(),
1> Frame = wxFrame:new(wx:null(), -1, "bitmaps"),
1> wxFrame:show(Frame).
true
2> Logo = wxBitmap:new(
           filename:join(code:lib_dir(wx,priv), "erlang-logo64.png")).
{wx_ref,36,wxBitmap,[]}
3> CDC = wxClientDC:new(Frame).
{wx_ref,37,wxClientDC,[]}
4> wxDC:drawBitmap(CDC, Logo, {10,10}).
ok
```



```
5> Bitmap = wxBitmap:new(wxBitmap:getWidth(Logo)+40, wxBitmap:getHeight(Logo)
+40).
{wx_ref,38,wxBitmap,[]}
6> MDC = wxMemoryDC:new(Bitmap).
{wx_ref,39,wxMemoryDC,[]}
7> wxDC:setBackground(MDC, wxBrush:new({10,10,10})),
7> wxDC:clear(MDC).
ok
8> wxDC:drawBitmap(MDC, Logo, {20,20}).
ok
9> Heart = fun(DC, X,Y) ->
9>   wxDC:drawLines(DC, [{X,Y},{X-20,Y+30}, {X-40,Y}]),
9>   wxDC:setBrush(DC, wxBrush:new({255,0,0})),
9>   wxDC:drawArc(DC, {X,Y}, {X-20,Y}, {X-10,Y}),
9>   wxDC:drawArc(DC, {X-20,Y}, {X-40,Y}, {X-30,Y}),
9>   wxDC:floodFill(DC, {X-20, Y+10}, {0,0,0}, [{style, 2}])
9> end.
#Fun<erl_eval.18.99386804>
10> Heart(MDC, wxBitmap:getWidth(Bitmap) - 5, 15).
true
11> wxMemoryDC:destroy(MDC).
ok
12> wxDC:drawBitmap(CDC, Bitmap, {150,10}).
ok
```

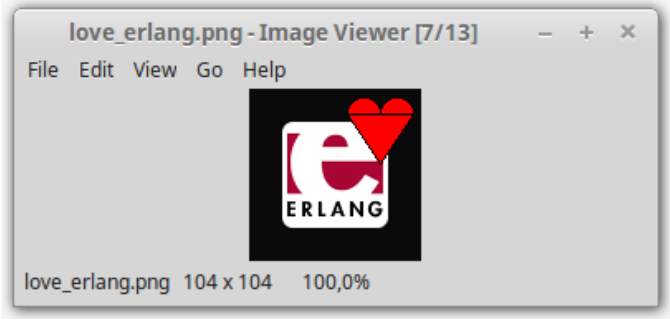


We drew a heart onto the bitmap using the **`wxDC:drawLines`**, **`wxDC:drawArc`** and **`wxDC:floodFill`** functions.

We can save the bitmap to file:

```
14> wxBitmap:saveFile(Bitmap,"love_erlang.png", 15).
true
```

The 15 there is the value for the constant `wxBITMAP_TYPE_PNG`. If we go look in our directory, we should see the file `love_erlang.png`:



As an exercise, think of a way to draw the heart without getting that horizontal black line.

Clipboard

We could also copy the image to the clipboard. The wxWidgets documentation gives us some hints on how to do that using the **`wxClipboard`**:

> A class for manipulating the clipboard. [snip]
>
> To use the clipboard, you call member functions of the global wxTheClipboard object.
>
> See also the wxDataObject overview for further information.
>
> Call wxClipboard::Open to get ownership of the clipboard. If this operation returns true, you now own the clipboard. Call wxClipboard::SetData to put data on the clipboard, or wxClipboard::GetData to retrieve data from the clipboard. Call wxClipboard::Close to close the clipboard and relinquish ownership. You should keep the clipboard open only momentarily.

In wxErlang, the way to access the global `wxTheClipboard` is **`wxClipboard:get/0`**. Here's how we could copy our bitmap to the clipboard:

```
TheClipboard = wxClipboard:get(),
ClipboardBitmap = wxBitmapDataObject:new({bitmap, Bitmap})
true = wxClipboard:open(TheClipboard),
true = wxClipboard:setData(TheClipboard,ClipboardBitmap),
ok = wxClipboard:close(TheClipboard).
```

Providing Art

Yet another interesting capability is retrieving custom or system-provided art, such as icons or bitmaps. The **`wxArtProvider`** class is used for this.

According to the wxWidgets documentation:

> wxArtProvider class is used to customize the look of wxWidgets application. When wxWidgets needs to display an icon or a bitmap (e.g. in the standard file dialog), it does not use a hard-coded resource but asks wxArtProvider for it instead. This way users can plug in their own wxArtProvider class and easily replace standard art with their own version. All that is needed is to derive a class from wxArtProvider, override its CreateBitmap method and register the provider with wxArtProvider::Push:
>
> There's another way of taking advantage of this class: you can use it in your code and use platform native icons as provided by wxArtProvider::GetBitmap or wxArtProvider::GetIcon
>
> Every bitmap is known to wxArtProvider under an unique ID that is used when requesting a resource from it. The ID is represented by wxArtID type and can have one of these predefined values (you can see bitmaps represented by these constants in the artprov sample):
>
> wxART_ADD_BOOKMARK
> wxART_DEL_BOOKMARK
> wxART_HELP_SIDE_PANEL
> wxART_HELP_SETTINGS
> wxART_HELP_BOOK
> wxART_HELP_FOLDER
> wxART_HELP_PAGE
> wxART_GO_BACK
> wxART_GO_FORWARD
> wxART_GO_UP

wxART_GO_DOWN
wxART_GO_TO_PARENT
wxART_GO_HOME
wxART_FILE_OPEN
wxART_PRINT
wxART_HELP
wxART_TIP
wxART_REPORT_VIEW
wxART_LIST_VIEW
wxART_NEW_DIR
wxART_FOLDER
wxART_GO_DIR_UP
wxART_EXECUTABLE_FILE
wxART_NORMAL_FILE
wxART_TICK_MARK
wxART_CROSS_MARK
wxART_ERROR
wxART_QUESTION
wxART_WARNING
wxART_INFORMATION
wxART_MISSING_IMAGE

Additionally, any string recognized by custom art providers registered using Push may be used.

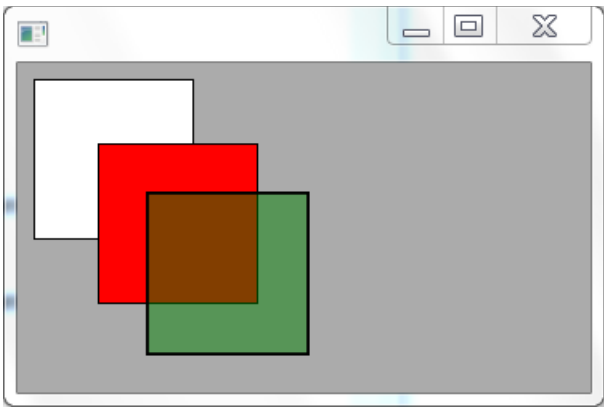
Here's how we could draw the native bitmap for an error icon, scaled to 100x100 pixels, onto our `Frame`:

```
DC = wxClientDC:new(Frame),
ErrorBmp = wxArtProvider:getBitmap("wxART_ERROR", [{size, {100,100}}]),
wxDC:drawBitmap(DC, ErrorBmp, {0,0}),
wxBitmap:destroy(ErrorBmp),
wxClientDC:destroy(DC).
```

Transparency

Talking of bitmaps, it's important to note that transparency is supported, even though not for pens or brushes. We can demonstrate this with a little experiment in which we draw a white rectangle (default brush), then a red rectangle using a brush with an alpha channel, and finally a green, semi-transparent rectangle, generated by an external program such as Inkscape, and stored in a file:

```
29> DC = wxClientDC:new(Frame),
29> wxDC:drawRectangle(DC, {10,10,100,100}),
29> wxDC:setBrush(DC, wxBrush:new({255,0,0,125})), %% red with alpha
29> wxDC:drawRectangle(DC, {50,50,100,100}),
29> BM = wxBitmap:new("semitransparent.png"),
29> wxDC:drawBitmap(DC, BM, {80,80}).
ok
```



As we can see, the red rectangle, despite the 50% transparency of the brush, is painted opaque, whereas the green rectangle is painted 50% transparent because that's how it was generated.

We'll take another look at transparency in a later chapter.

Let's now move on to using some of the concepts we've learnt here to construct a chessboard for our chess players.

Chessboard

In this chapter we will develop a chessboard for our chess players, so they can put the chess clock to good use. A chessboard is made up of sixty four alternating black and white squares laid out in eight files (columns) and eight ranks (rows). The bottom-left square is black. There are thirty two pieces; sixteen black and sixteen white. For each colour, there are eight pawns, two rooks, two knights, two bishops, one queen and one king.

The game starts with the white pieces all lined up in ranks one and two and the black pieces in ranks seven and eight. Each player takes a turn at moving a piece. White moves first. There are several rules that determine which moves are legitimate and which are not. The goal is to capture the oponent's king. The one who is able to do so, wins. If it is impossible for both players to capture the other's king, no one wins and a draw is declared.

The games can take a very long time, with each player trying to figure out the best move. A variant of the game limits the allowed amount of time. That's where the chess clock is used. The type we constructed in *Geting Started*.

We'll start with drawing the chessboard with all the pieces placed on it in the starting position.

We need a square drawing surface, which we'll divide into 64 squares, 8 rows by 8 columns. We'll allow the user to resize it, but we'll put a constraint that the size of the board be such that the size of the 64 squares is always an even number of pixels. So for a panel `W` pixels by `H` pixels, the size of the 64 squares will be `((min(W, H) div 8) div 2) * 2` pixels.

```
square_size(W,H) -> ((min(W,H) div 8) div 2) * 2 end.
```

For a panel 500x430 that gives us `((min(500,430) div 8) div 2) * 2` = 52 pixels.

In the chess game, the ranks are numbered 1 to 8 starting from the bottom most, if drawn on a vertical surface, and the files A to H starting from the left most. An algebric convention is employed to name the squares by their file and rank. For computing, it is easier to deal with zero-based indexes, so we'll name the squares with a tuple of integers, {0,0} for the top-left square (A8), and {7,7} for the bottom-right square (H1).

We can define two functions to do the mappings, if needed:

```
to_internal([File, Rank]) -> {File - $A, $8 - Rank} end.
from_internal({Column, Row}) -> [Column + $A, $8 - Row] end.
```

So, `to_internal("A2") = {0,6}` and `from_internal({0,6}) = "A2"`.

We'll need to determine the rectangle to draw for each square:

```
rectangle({Column,Row}, SquareSize) ->
     {Column * SquareSize, Row * SquareSize, SquareSize, SquareSize} end.
```

We'll need to know whether the square is black or white:

```
square_colour({Column,Row}) ->
     case (Column+Row) div 2 of 0 -> white; _ -> black end end.
```

We'll need a brush to paint the white squares:

```
White = {140,220,120},
WhiteBrush = wxBrush:new(White),
```

And of course one to paint the black squares:

```
Black = {80,160,60},
BlackBrush = wxBrush:new(Black),
```

Note that the squares don't have to be white and black, just a light shade and a dark shade.

We'll need to either draw the pieces with lines and shapes or use bitmaps, which, as we saw in the previous chapter, we can generate from images stored in files. I have used *Inkscape* to generate PNG images of all the white and black pieces, and if you don't want to make your own pieces, you can downlaod[7] them.

No support for user-provide Art

We could `push` these images to the `wxArtProvider` and then use them in drawing the chessboard, but unfortunately **wxErlang does not support that**. So we have to deal with the images oursleves.

wxImage

However, there is one difficulty with bitmaps. Since we will be allowing the user to resize the chessboard, the bitmaps will have to adapt to the board's dimensions. There is no function in the `wxBitmap` module that will allow us to do that. Nevertheless, wxWidgets does provide the **`wxImage`** abstraction. It allows us to read images off files, and also has functions to scale and manipulate those image.

So, we will load all the piece images into `wxImage` objects, scale them to the appropriate size and cache them so that we can create the bitmaps when needed. We'll use a map for the cache and index it with the tuple `{piece_colour, piece_role}`:

```
load_images() ->
    ImageFileNames = #{
      {black, rook}    => "black_rook.png",
      {black, knight}  => "black_knight.png",
      {black, bishop}  => "black_bishop.png",
      {black, queen}   => "black_queen.png",
      {black, king}    => "black_king.png",
      {black, pawn}    => "black_pawn.png",
      {white, rook}    => "white_rook.png",
      {white, knight}  => "white_knight.png",
      {white, bishop}  => "white_bishop.png",
      {white, queen}   => "white_queen.png",
      {white, king}    => "white_king.png",
      {white, pawn}    => "white_pawn.png"},
    maps:map(fun(_K,V) -> wxImage:new(
                      filename:join("../images", V),
                      [{type, ?wxBITMAP_TYPE_PNG}]) end,
            ImageFileNames).
```

7 images/

Since `wxImage` and `wxBitmap`, just like `wxDCs`, are C++ objects that are not garbage collected, we need to make sure we remove them with the appropriate **`destroy`** functions, once we are done, to avoid memory leaks.

```
[wxImage:destroy(I) || I <- maps:values(Images)],
```

Finally, we need to know the **layout**: what piece is placed in which square, when drawing the chess board. This, too, we'll keep in a map, keyed on the square name. Before starting the game, we'll initialize this map so that it reflects the starting position of all the pieces:

```
init_board() ->
    Columns = lists:seq(0,7),
    BlackPieces = [{black,rook}, {black,knight}, {black,bishop}, {black,queen},
                   {black,king}, {black,bishop}, {black,knight}, {black,rook}],
    WhitePieces = [{white,rook}, {white,knight}, {white,bishop}, {white,queen},
                   {white,king}, {white,bishop}, {white,knight}, {white,rook}],
    Row1 = [{{C,1}, {white,pawn}} || C <- Columns],
    Row6 = [{{C,6}, {black,pawn}} || C <- Columns],
    Row0 = [{{C,0}, lists:nth(C+1, WhitePieces)} || C <- Columns],
    Row7 = [{{C,7}, lists:nth(C+1, BlackPieces)} || C <- Columns],

    maps:from_list(Row0 ++ Row1 ++ Row6 ++ Row7).
```

We'll subscribe to the `paint` event and draw the board in the paint event handler. As we said in the previous section, we will handle the paint event in a callback because Windows requires it. We'll draw a piece only if we see from the layout that there's one on the square:

```
paint_board(#{panel := Panel,
              layout := Layout,
              image_map := ImageMap,
              white_brush := WhiteBrush,
              black_brush := BlackBrush}) ->
    {W,H} = wxPanel:getSize(Panel),
    SquareSize = square_size(W,H),

    PaintSquare =
      fun(DC,C,R) ->
            Brush = case square_colour(C,R) of
                      black -> BlackBrush;
                      white -> WhiteBrush
            end,
            Rectangle = rectangle(C,R,SquareSize),
            wxDC:setBrush(DC,Brush),
            wxDC:drawRectangle(DC, Rectangle),

            case maps:get({C,R}, Layout, none) of
                none -> ok;
                Piece ->
                  {X,Y,SW,SH} = Rectangle,
                  Image = wxImage:scale(maps:get(Piece, ImageMap),SW,SH),
                  PieceBitmap = wxBitmap:new(Image),
                  wxDC:drawBitmap(DC, PieceBitmap, {X,Y}),
                  wxImage:destroy(Image),
                  wxBitmap:destroy(PieceBitmap)
            end
      end,

    DC = wxPaintDC:new(Panel),
    wxDC:setPen(DC, ?wxTRANSPARENT_PEN),
    Seq0to7 = lists:seq(0,7),
```

```
    [PaintSquare(DC,C,R) || R <- Seq0to7, C <- Seq0to7],
    wxPaintDC:destroy(DC).
```

We created a *scaled copy* of the piece image, and then created a bitmap from it.

Let's put all these bits together into a module[8], compile it and run it (making sure the image files are in the directory `../images`) :

```
8> c(drawn_board).
{ok,drawn_board}
9> drawn_board:start_link().
{wx_ref,37,wxPanel,<0.123.0>}
```

Huh? We just get an empty frame, no board.

If we try to resize it, though, the board, with all the pieces set in their starting position does get drawn:

You may want to take a peek at the code, to notice that we have:

- employed the style **`FULL_REPAINT_ON_RESIZE`** for the frame,
- subscribed to the paint event with option `callback,`
- handled the paint event in the *synchronous* event handler,
- painted the board with a `wxPaintDC`.

Not too bad for starters.

So why did we have to resize the board? To be honest, I don't know, and have not investigated much. I traced the paint event triggering and saw it wasn't triggered by the `wxFrame:show`.

8 chessboard/drawn_board.erl

However, we can force its generation if we add a `wxWindow:refresh/1` right after `wxFrame:show`. It's commented out in the module we just compiled, so you can verify this by removing the comment sign and recompiling.

Do you notice any flicker or flashing when resizing the board? We'll have more to say on that soon.

Mouse Events

Next, let's see how to move the pieces. We'll do it in two steps. We'll click on a piece to select it. Then we'll clik on the square where we want to drop the piece to drop it there.

To capture the click, we need to subscribe to some event that gets generated when we the mouse is clicked. We see in the `wx` documentation that there is an event called **`wxMouseEvent`**:

Use wxEvtHandler:connect/3 with EventType:
left_down, left_up, middle_down, middle_up, right_down, right_up, motion, enter_window, leave_window, left_dclick, middle_dclick, right_dclick, mousewheel

So we subscribe to `left_down`.

```
wxPanel:connect(Panel, left_down),
```

In the event handler, we will get a `wxMouse` event which contains the coordinates of where the mouse was clicked. With those coordinates, we should be able to determine which square it was and select it if it contains a piece that can be moved.

```
where(X,Y,Panel) ->
      {W,H} = wxPanel:getSize(Panel),
      SquareSize = square_size(W,H),
      {X div SquareSize, Y div SquareSize}.
```

`where/3` will tell us the coordinates as `{Column, Row}`.

Now this is just a click, we don't really know whether it is a click to *select* a piece to move it, or it is a click to *drop* a selected piece. This requires some intelligence, which we may want to delegate to some other process.

To provide a visual feedback that a piece *has* been selected, we could set the background of the square to a different colour, eg `{238,232,170}`. When we drop the piece in some other square, we will remove the piece from the selected square and resotre its colour.

We will also substitute any piece there may have been at the dropped square. This means that we need to remember which square *is* selected in the board's state, and the paint handler has to take this into account as well. Instead of painting the single squares, we genereate a paint event with `wxWindow:refresh/1` and let the paint handler handle it.

So, the state is changed to:

```
          #{frame => Frame,
            panel => Panel,
            layout => init_board(),
            image_map => load_images(),
            white_brush => wxBrush:new(White),
            black_brush => wxBrush:new(Black),
```

```
            selected_brush => wxBrush:new({238,232,170}),
            selected => none}
```

We handle the mouse event with the assumption that when there is a click with nothing selected, we are selecting a piece, and that when something *is* already selected, we are dropping it.

```
handle_event(#wx{event=#wxMouse{leftDown=true,x=X,y=Y}},
           State = #{panel := Panel,
                     layout := Layout,
                     selected := none}) ->   %% selecting a piece
    {C,R} = where(X,Y,Panel),
    case maps:get({C,R}, Layout, none) of
      none ->
            {noreply, State};
      _ ->
            wxPanel:refresh(Panel),
            {noreply, State#{selected => {C,R}}}
    end;

handle_event(#wx{event=#wxMouse{leftDown=true,x=X,y=Y}},
           State = #{panel := Panel,
                     layout := Layout,
                     selected := Selected}) ->   %% dropping a selected piece
    {C,R} = where(X,Y,Panel),
    Piece = maps:get(Selected, Layout),
    NewLayout = maps:put({C,R}, Piece, maps:remove(Selected, Layout)),
    wxPanel:refresh(Panel),
    {noreply, State#{layout => NewLayout, selected => none}};
```

We will need to modify the `paint_board` function to use the `selectedBrush` if the square to be drawn is the one selected:

```
            Brush = case Selected of
                      {C,R} ->
                              SelectedBrush;
                      _ ->
                              case square_colour(C,R) of
                                    black -> BlackBrush;
                                    white -> WhiteBrush
                              end
                    end,
```

Recompile[9] and run it. You should be able to select the pieces and drop them somewhere. There is no enforcement of allowed moves, but from a GUI point of view, it works!

Flicker

Moving the pieces around causes some annoying flicker, more so under Windows than under Linux. Hunting for information on flickering, you may bump into a page in the wiki of wxWidgets on Flicker-free drawing. Among other things, it says:

> **Flicker free drawing** can be achieved by a two step process.
>
> 1. Disable erase background event. When wxWidgets wants to update the display it emits two events: an erase background event and a paint event. You must implement an empty method for the erase background event (in other words: intercept the EVT_ERASE_BACKGROUND event and dont call event.Skip()).

9 chessboard/move_piece.erl

2. Use a double buffer. This means drawing to a bitmap instead of to the display. When drawing is complete, you copy the bitmap to the display. Note that the bitmap must be the same size as the window.

These two steps work together. If you only disabled the erase event, the display would contain leftovers from the last paint event. Since you also use a bitmap that covers the entire window you will automatically overwrite everything. If you only used a double buffer, you would still see a flash because the window was updated twice: 1st by the erase event, 2nd by the paint event.

Looking into the wx documentation we see there is a **`wxErase`** event with event type **`erase_background`**. So we'll subscribe to this event to intercept it and handle it as a synchronous callback:

```
    wxPanel:connect(Panel, erase_background, [callback]),
```

Tha handling is simple. We just ignore the event!

```
handle_sync_event(#wx{event=#wxErase{}}, _, _) -> ok.
```

If we try[10] it, we'll see that things are already a lot better. The rather annoying flash is gone in Windows, but a certain flicker still remains.

Let's also try the second of the measures, that of painting onto a bitmap and then painting that onto the frame. Actually, this is easily achieved by using a buffered DC:

This class provides a simple way to avoid flicker: when drawing on it, everything is in fact first drawn on an in-memory buffer (a wxBitmap) and then copied to the screen, using the associated wxDC, only once, when this object is destroyed. wxBufferedDC itself is typically associated with wxClientDC, if you want to use it in your `EVT_PAINT` handler, you should look at wxBufferedPaintDC instead.

And the documentation for **`wxBufferedPaintDC`** says:

This is a subclass of wxBufferedDC which can be used inside of an `OnPaint()` event handler. Just create an object of this class instead of wxPaintDC and make sure wxWindow::SetBackgroundStyle is called with wxBG_STYLE_CUSTOM somewhere in the class initialization code, and that's all you have to do to (mostly) avoid flicker. The only thing to watch out for is that if you are using this class together with wxScrolledWindow, you probably do **not** want to call PrepareDC on it as it already does this internally for the real underlying wxPaintDC.

So, let's follow this advice by setting the background style of the panel to custom style:

```
    Panel = wxPanel:new(Frame,[{size, {320,320}},
                               {style, ?wxFULL_REPAINT_ON_RESIZE}]),
    wxPanel:setBackgroundStyle(Panel, ?wxBG_STYLE_CUSTOM),
```

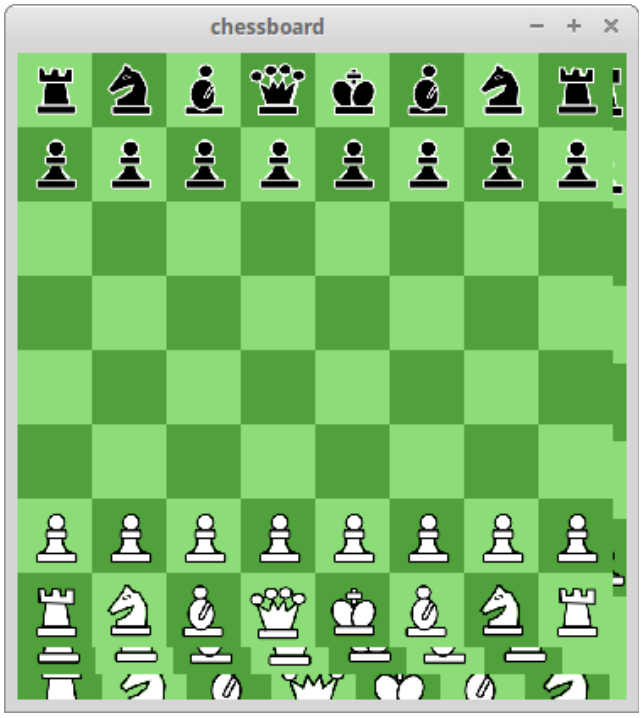
and then using `wxBufferedPaintDC`, instead of `wxPaintDC`:

```
    DC = wxBufferedPaintDC:new(Panel),
    wxDC:setPen(DC, ?wxTRANSPARENT_PEN),
    Seq0to7 = lists:seq(0,7),
    [PaintSquare(DC,C,R) || R <- Seq0to7, C <- Seq0to7],
    wxBufferedPaintDC:destroy(DC).
```

If we try[11] this use of a buffered DC, we'll see the flicker is practically gone.

10 chessboard/no_erase.erl
11 chessboard/buffered_dc.erl

However, if we resized the window several times, we could end up with something like:



This is not what we bargained for. The reason is that we are only painting a portion of the entire panel. That extra space has "garbage" we need to get rid of.

One way is to paint it with the background colour of the board panel, which we keep in the state:

```
background_brush => wxBrush:new(wxPanel:getBackgroundColour(Panel)),
```

Then, within the `paint_board` function, where we already have the panel's size and the square size:

```
{W,H} = wxPanel:getSize(Panel),
SquareSize = square_size(W,H),
```

we can paint the extra space clean:

```
  wxDC:setBrush(DC, BackgroundBrush),
  BoardSize = 8 * SquareSize,
  wxDC:drawRectangle(DC,{BoardSize,0,W-BoardSize,H}),
  wxDC:drawRectangle(DC, {0,BoardSize,BoardSize,H-BoardSize}),
```

Try it[12]. You will see it works.

The second, simpler way, is to erase the background before drawing.

```
    wxDC:setBackground(DC, BackgroundBrush),
    wxDC:clear(DC),
```

Try[13] again. The result should be the same.

12 chessboard/flicker_free1.erl
13 chessboard/flicker_free2.erl

Chessboard revisited

Let's see if we can somehow enforce that pieces can be moved only according to the rules of chess.

We need to identify which pieces have allowed moves - according to the rules - and, once selected, which squares the selected piece can be moved to.

The player playing black is allowed to move only the black pieces, while the player playing white is allowed to move only the white pieces. Obvious, but we need to keep track of whose turn it is to move.

Cursors

From the GUI point of view, we'd like to be able to give a visual cue of which pieces can be moved and which squares they can be moved to, once selected. My preference would be to change the cursor into a hand shape on the eligible squares. So let's see how we can do that.

wxWidgets handles cursors by means of **`wxCurosr`**.

> A cursor is a small bitmap usually used for denoting where the mouse pointer is, with a picture that might indicate the interpretation of a mouse click. [snip]
>
> A single cursor object may be used in many windows (any subwindow type). The wxWidgets convention is to set the cursor for a window, as in X, rather than to set it globally as in MS Windows, although a global ::wxSetCursor is also available for MS Windows use.

We already have a problem: cursors cannot change within parts of the window, our board's squares. They can only be changed from one window to the other. Pity, our squares are not windows.

Grid Sizer

Hey, wait! What stops us from making them into windows, `wx_objects` in their own right? There'll be too many of them? 64, to be precise, for just one board. Well, since when did an Erlanger become afraid of too many processes? Let's get on with it and transform our board into a grid of 64 square widgets! Did we say grid? ok, a Grid Sizer be it, then.

> [wxGridSizer] is a sizer which lays out its children in a two-dimensional table with all table fields having the same size, i.e. the width of each field is the width of the widest child, the height of each field is the height of the tallest child.

We'll use a grid sizer with 8 rows and 8 columns and no space between them:

```
Sizer = wxGridSizer:new(8,8,0,0),
```

In each of those 64 squares we'll add a wx_object: `chess_square`. The square doesn't have to know everything about the board, but it does need the following information:

- a reference to the panel, or frame, to use as the parent
- the board's pid, to be able to send it messages
- a brush to paint the square
- a brush to paint the square when it is selected

- a flag to say if it *is* selected
- the image of any piece that is lying on it
- and though not needed now, we will also provide it with its location on the board

```
%% in chess_square.erl

start_link(Location, BoardPid, Parent, Brush, SelectedBrush) ->
    wx_object:start_link(
      ?MODULE, [Location, BoardPid, Parent, Brush, SelectedBrush], []).

init([Location, BoardPid, Parent, Brush, SelectedBrush]) ->
    Panel = wxPanel:new(Parent, [{style, ?wxFULL_REPAINT_ON_RESIZE}]),
    wxPanel:setBackgroundStyle(Panel, ?wxBG_STYLE_CUSTOM),
    wxPanel:connect(Panel, paint, [callback]),
    wxPanel:connect(Panel, erase_background, [callback]),
    State = #{
      location => Location,
      board_pid => BoardPid,
      square_panel => Panel,
      image => none,
      brush => Brush,
      selected_brush => SelectedBrush,
      selected => false},
    {Panel, State}.
```

When the square gets a paint event, we need to clear it with the background colour which is suitable for the state of the square, and paint the piece, if there is one, onto it. We block the erase event to suppress any flashing due to it.

```
handle_sync_event(#wx{event=#wxPaint{}}, State) ->
    paint_square(State);
handle_sync_event(#wx{event=#wxErase{}}, _, _) ->
    ok.

paint_square(#{square_panel := Panel,
               image := PieceImage,
               brush := Brush,
               selected_brush := SelectedBrush,
               selected := Selected}) ->
    Paint =  fun  (_DC, none) -> ok;
                  (DC, Image) ->
                    {W,H} = wxPanel:getSize(Panel),
                    ScaledImage = wxImage:scale(Image,W,H),
                    PieceBitmap = wxBitmap:new(ScaledImage),
                    wxDC:drawBitmap(DC, PieceBitmap, {0,0}),
                    wxImage:destroy(ScaledImage),
                    wxBitmap:destroy(PieceBitmap)
             end,

    DC = wxBufferedPaintDC:new(Panel),
    wxDC:setPen(DC, ?wxTRANSPARENT_PEN),
    wxDC:setBackground(DC, case Selected of
                             true  -> SelectedBrush;
                             false -> Brush
                           end),
    wxDC:clear(DC),
    Paint(DC, PieceImage),
    wxBufferedPaintDC:destroy(DC).
```

Now that we have the square code[14] done, we turn our attention to the construction of a board using these squares. We'll do this in the module `chess_board`.

As we said, we are going to use an 8 x 8 Grid Sizer. We will create 64 square widgets and add them to the sizer. We will keep a map of the square widgets and a map of their pids so we can easily access any square we desire and communicate with it. The maps will be keyed on the square's location on the board: `{Column, Row}`:

```
%% in chess_board.erl

SquaresList = [MkSquare(C,R) || R <- lists:seq(0,7), C <- lists:seq(0,7)],
SquareMap = maps:from_list(SquaresList),
SquarePidMap = maps:map(fun(_,V) -> wx_object:get_pid(V) end, SquareMap),
```

`MkSquare/2` is a function, a closure, that we define in the board's `init/1` function:

```
-define(SQUARE, chess_square).
-define(UTILS, chess_utils).
-define(WHITE, {140,220,120}).
-define(BLACK, {80,160,60}).
-define(SELECTED_COLOUR, {238,232,170}).

start_link() ->
    wx_object:start_link(?MODULE, [], []).

init([]) ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), ?wxID_ANY, "chess_board"),
    MainSizer = wxBoxSizer:new(?wxVERTICAL),

    Board = wxPanel:new(Frame, [{style, ?wxFULL_REPAINT_ON_RESIZE}]),
    wxPanel:setBackgroundStyle(Board, ?wxBG_STYLE_CUSTOM),

    Grid = wxGridSizer:new(8,8,0,0),
    wxPanel:setSizer(Board, Grid),

    WhiteBrush = wxBrush:new(?WHITE),
    BlackBrush = wxBrush:new(?BLACK),
    SelectedBrush = wxBrush:new(?SELECTED_COLOUR),
    BackgroundBrush = wxBrush:new(wxPanel:getBackgroundColour(Board)),

    Layout = ?UTILS:init_board(),
    ImageMap = ?UTILS:load_images(),

    MkSquare =
      fun(C,R) ->
            SquareColour = ?UTILS:square_colour(C,R),
            Brush = case SquareColour of
                      white -> WhiteBrush;
                      black -> BlackBrush
                  end,
            Square = ?SQUARE:start_link(
                      {C,R},
                      self(),
                      Board,
                      Brush,
                      SelectedBrush),
            {{C,R}, Square}
      end,
```

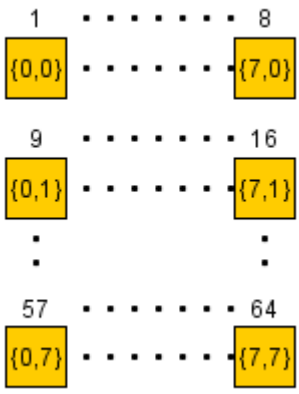
14 game1/chess_square.erl

```
    SquaresList = [MkSquare(C,R) || R <- lists:seq(0,7), C <- lists:seq(0,7)],
    [wxSizer:add(Grid, Square, [{flag, ?wxEXPAND}])
     || {_,Square} <- SquaresList],

    SquareMap = maps:from_list(SquaresList),
    SquarePidMap = maps:map(fun(_,V) -> wx_object:get_pid(V) end, SquareMap),
    layout_pieces(Layout, ImageMap, SquarePidMap),

    wxSizer:add(MainSizer, Board, [{flag, ?wxEXPAND}, {proportion,1}]),
    wxFrame:setSizer(Frame, MainSizer),

    wxFrame:show(Frame),
    {W,H} = wxFrame:getClientSize(Frame),
    wxPanel:setSize(Board, W, H),
    wxWindow:refresh(Frame),

    State = #{frame => Frame,
            board => Board,
            layout => Layout,
            image_map => ImageMap,
            {white, brush} => WhiteBrush,
            {black, brush} => BlackBrush,
            selected_brush => SelectedBrush,
            background_brush => BackgroundBrush,
            square_map => SquareMap,
            square_pid__map => SquarePidMap,
            selected => none},

    {Frame, State}.
```

We've put the functions `square_colour/2`, `init_board/0`, `load_images/0` in a separate module `chess_utils`.

`square_colour/2` returns the colour of the square (black or white), based on its location (Column, Row).

Please note the sequence of column-row tuples and the sequence with which the widgets are added to the grid. We are going to need to know this later.



We need to provide those squares that have a piece on them, the image of the placed piece. We do this by sending the square the message `{image, PieceImage}`.

However, since we will also need to tell the squares whether they are selected or not, whether they *can be* selected or not, and so on, we provide a utility function in the `chess_square` module to set the value of such state variables:

```
handle_info({Property, Value}, State = #{square_panel := Panel}) ->
    wxWindow:refresh(Panel),
    {noreply, maps:put(Property, Value, State)};
```

If in setting such properties, the square appearance changes (for example when we change the image of the piece on the square), we will also request the re-painting of the square by means of `wxWindow:refresh/1`.

Using this utility function, here's how we provide the images of all the pieces:

```
layout_pieces(Layout, ImageMap, SquarePidMap) ->
    maps:fold(
      fun(Location,Piece,_) ->
            Image = maps:get(Piece, ImageMap),
            Pid = maps:get(Location, SquarePidMap),
            Pid ! {image, Image}
      end,
      [],
      Layout).
```

We go through the layout of the board and for each location that has a piece on it, we send the square the relative piece image.

Note that we don't bother about painting the board itself as the single squares cover the entire board surface.

It'd be a good idea to see if the board will get drawn as expected before going any further by compiling[15] all the files: `chess_square`, `chess_board` and `chess_utils` and running `chess_board:start_link/0`.

It does! Just a slight flicker when we resize it. However, the board doesn't stay square. And we don't have any constraints in the sizers that can enforce the board's squareness. We'll tackle this problem later. For now let's move on.

Changing the Cursor

Let's see if we can now tell a square it can be selected (is selectable) and have the cursor change to a hand when moved over it.

Here's a snippet from the wxWidgets documentation:

wxWindow::SetCursor

virtual void SetCursor(**const wxCursor&***cursor*)

Sets the window's cursor. Notice that the window cursor also sets it for the children of the window implicitly.

15 `game1/`

The *cursor* may be `wxNullCursor` in which case the window cursor will be reset back to default.

Parameters

cursor

Specifies the cursor that the window should normally display.

We'll use the boolean property `selectable` to indicate to the square if it can be selected or not. Similarly, we'll use the property `landable` to indicate to the square if a piece is allowed to land on it or not. We modify the property setting handler to modify the square's cursor when the property being set is one of these two:

```
%% in chess_square.erl

handle_info({Property, Value}, State = #{square_panel := Panel})
  when Property =:= selectable; Property =:= landable ->
    case Value of
      true ->
          wxWindow:setCursor(Panel, wxCursor:new(?wxCURSOR_HAND));
      false ->
          wxWindow:setCursor(Panel, ?wxNullCursor)
    end,
    {noreply, maps:put(Property, Value, State)};
```

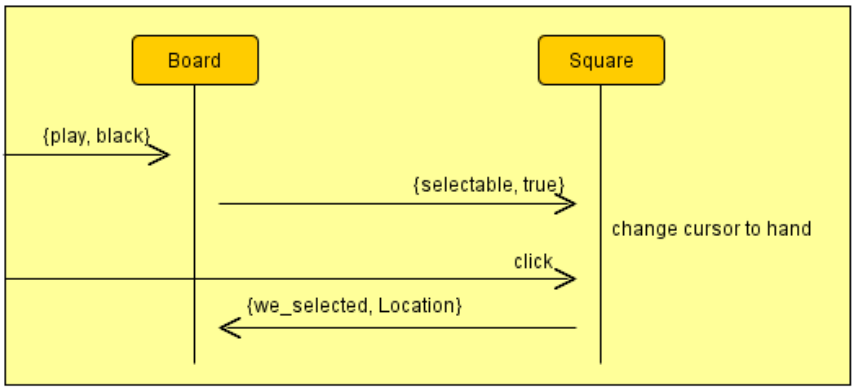
We are not going to implement the chess rules yet, but just for the sake of verifying that this works, we'll allow the player to move the pieces of his colour. We'll send the board a message `{play, Colour}` to indicate it should allow us to select the pieces of the said `Colour`.

```
%% in chess_board.erl

handle_info({play, Colour}, State = #{layout := Layout,
                                      square_pid_map := SquarePidMap}) ->
    maps:fold(
      fun(Location, {C, _}, _AccIn) when C =:= Colour ->
            maps:get(Location, SquarePidMap) ! {selectable, true}
      end,
      [], Layout),
    {noreply, State};
```

As in the previous section, we will subscribe to the mouse left click to actually select the piece on a square and to then move it to another location. This time the click will be detected by the square and not the board, so we will have the square inform the board that it has been clicked by means of the message `we_selected`.

The board, in turn, will mark the selectable pieces as no longer selectable, and will mark those squares where the selected piece can land as landable. Let's assume we allow the selected piece to land on any empty square or on an opponent's piece. In order to do this, the board will need to know the location, on the board, of the selected piece. So we let the square give this information inside the `we_slected` message: `{we_selected, Location}`.

The following sequence diagram illustrates this:




```
%% in chess_square.erl

handle_event(#wx{event=#wxMouse{type = left_down}},
           State = #{square_panel := Panel,
                     board_pid := BoardPid,
                     selectable := true,
                     selected := false,
                     location := Location}) ->
    BoardPid ! {we_selected, Location},
    wxPanel:refresh(Panel),
    {noreply, State#{selected => true}};

handle_event(#wx{event=#wxMouse{type = left_down}}, State) ->
    {noreply, State};
```

The board, upon receiving this message will prepare the squares where the selected piece can land:

```
%% in chess_board.erl

handle_info({we_selected, SquareLocation},
          State = #{square_pid_map := SquarePidMap,
                    layout := Layout}) ->
    {Colour, _} = maps:get(SquareLocation, Layout),
    maps:fold(
      fun(Location, {C, _}, _AccIn) when C =:= Colour ->
            maps:get(Location, SquarePidMap) ! {selectable, false};
       (_,_,_) ->
            maps:get(SquareLocation, SquarePidMap) ! {selectable, true}
      end,
      [], Layout),
    {noreply, State#{selected =>  SquareLocation}};
```

We should now have a piece selected and ready to move to some other location. That location is where the player will click again, provided the square is `landable`. This time the square will tell the board there was a `move` with the message `{we_moved, Location}`

```
%% in chess_square.erl

handle_event(#wx{event=#wxMouse{type = left_down}},
```

```
          State = #{board_pid := BoardPid,
                    landable := true,
                    location := Location}) ->
    BoardPid ! {we_moved, Location},
    {noreply, State};
```

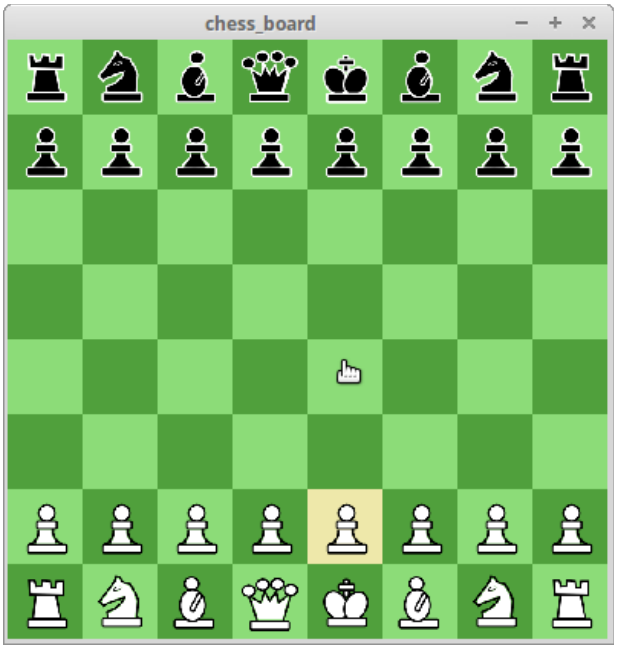
The moving of a piece is a complicated affair. The board will need to:

- mark all the landable squares not landable any more
- mark the opponent's pieces as selectable for the next move
- mark the selected square as not selected any more
- record that no square is selected any more
- remove the image of the chess piece from the selected square
- set the image of the square the piece was moved to, to the image removed from the selected square

```
%% in chess_board.erl
handle_info({we_moved, MovedToLocation},
          State = #{selected := FromLocation,
                    square_pid_map := SquarePidMap,
                    image_map := ImageMap,
                    layout := Layout}) ->

    [Sq ! {landable, false} || Sq <- maps:values(SquarePidMap)],

    Piece = {Colour, _} = maps:get(FromLocation, Layout),
    Opponent = ?UTILS:opponent(Colour),

    maps:fold(
      fun(Location, {C, _}, _AccIn) when C =:= Opponent ->
            maps:get(Location, SquarePidMap) ! {selectable, true};
       (_,_,_) ->
            ok
      end,
      [], Layout),

    FromPid = maps:get(FromLocation, SquarePidMap),
    FromPid ! {selected, false},
    FromPid ! {image, none},
    PieceImage = maps:get(Piece, ImageMap),
    TargetPid = maps:get(MovedToLocation, SquarePidMap),
    TargetPid ! {image, PieceImage},

    {noreply, State#{selected => none,
                layout => (maps:remove(FromLocation, Layout))#{
                        MovedToLocation => Piece}}};
```

It's time to try[16] it all to make sure it works.

```
1> B = chess_board:start_link().
{wx_ref,35,wxFrame,<0.63.0>}
2> wx_object:get_pid(B) ! {play, white}.
{play,white}
3>
```



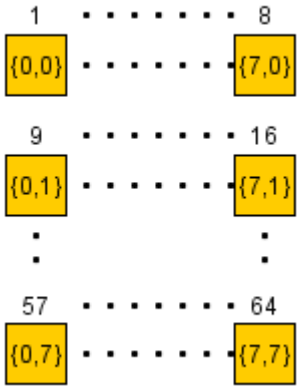
16 game2/

Chess Player

Playing Black

The chess board we have crafted will be used by one of the chess players. We have already seen that the player may be playing the white pieces or the black ones. So far we have seen the board from the point of view of a player playing white. What about the other player? The board wouldn't look right to him as the black pieces are on top.

We can employ a very simple change to achieve this. Remember how we layed out the pieces in the board grid?



To get a board from the point of view of a player playing black, all we need do is invert the order of the square locations. So, instead of going from {0,0} to {7,0} and so on, up to {7,7}, we'll go from {7,7} to {0,7} and so on, up to {0,0}, as if we had turned the board upside down.

And we will re-construct the board the right way, for the white or the black player, when we are told whether we are playing black or white. So, when constructing the default white board at the board creation, we'll create a number of functions, closures, to keep in the board's state so that we can then apply them to reconstruct the board when needed:

```
%% in chess_board.erl
    MkBoard =
      fun(SquareMap, Seq) ->
            wxSizer:clear(Grid),
            [wx_object:stop(Sq) || Sq <- maps:values(SquareMap)],
            SquareList = [MkSquare(C,R) || R <- Seq, C <- Seq],
            [wxSizer:add(Grid, Square, [{flag, ?wxEXPAND}])
             || {_, Square} <- SquareList],
            SquareList
      end,

    MkWhiteBoard = fun(ChessBoard, SquareMap) ->
                     Squares = MkBoard(SquareMap, lists:seq(0,7)),
                     wxPanel:layout(ChessBoard),
                     Squares
                end,
    MkBlackBoard = fun(ChessBoard, SquareMap) ->
                     Squares = MkBoard(SquareMap, lists:seq(7,0,-1)),
                     wxPanel:layout(ChessBoard),
                     Squares
                end,
```

```
    SquareMap = maps:from_list(MkWhiteBoard(Board, #{})), %% default board

...

    State = #{frame => Frame,
            board => Board,
            layout => Layout,
            make_white_board => MkWhiteBoard,
            make_black_board => MkBlackBoard,
            image_map => ImageMap,
            {white, brush} => WhiteBrush,
            {black, brush} => BlackBrush,
            selected_brush => SelectedBrush,
            background_brush => BackgroundBrush,
            square_pid_map => SquarePidMap,
            selected => none},
```

Note that we use the function `wxSizer:clear/1`. This will remove all widgets added to the sizer. A flag can be provided to destroy the removed widgets. However, we don't have "real" widgets in our implementation, but wx_objects. So, we make sure we terminate all of them by explicitly calling `wx_object:stop/1`.

Note also that we use `wxWindow:layout/1` after having added all the new square widgets to the grid sizer. This is necessary because the sizer has not had the opportunity to adjust the added widgets to their proper places and sizes according to its constraints. (In fact, if you omit this step, all the new widgets will get drawn one on top of the other in the upper left corner of the board).

To layout the board correctly, someone must inform the board which colour it will play. We do this by sending it the message `{role, Colour}`.

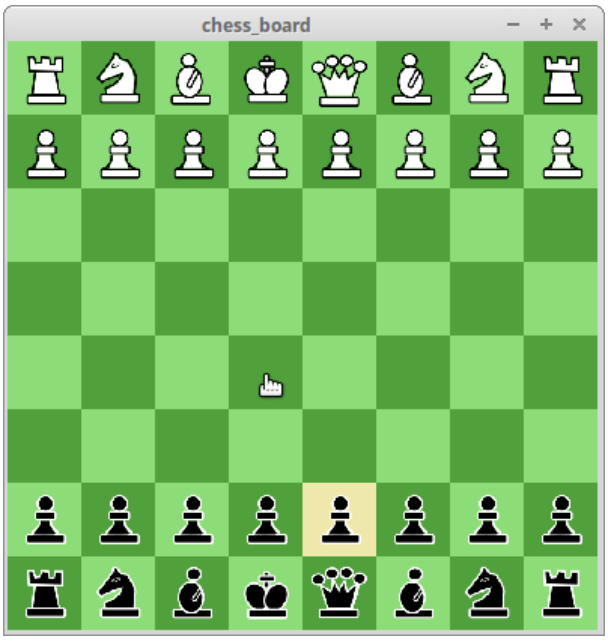
```
%% in chess_board.erl

handle_info({role, Role},
             State = #{board := Board,
                       image_map := ImageMap,
                       make_white_board := MkWhiteBoard,
                       make_black_board := MkBlackBoard,
                       square_map := PreviousSquareMap}) ->
    SquareMap = maps:from_list(
              case Role of
                  white -> MkWhiteBoard(Board, PreviousSquareMap);
                  black -> MkBlackBoard(Board, PreviousSquareMap)
              end),
    Layout = ?UTILS:init_board(),
    SquarePidMap = maps:map(fun(_,V) -> wx_object:get_pid(V) end, SquareMap),
    layout_pieces(Layout, ImageMap, SquarePidMap),

    {noreply, State#{layout => Layout,
                square_map => SquareMap,
                square_pid_map => SquarePidMap}};
```

Let's verify the board does place the pieces correctly for the black player by compiling everything[17] again:

```
9> B = chess_board:start_link().
{wx_ref,35,wxFrame,<0.2461.0>}
10> wx_object:get_pid(B) ! {role, black}.
```

17 game3/

```
{role,black}
11> wx_object:get_pid(B) ! {play, black}.
{play,black}
12>
```

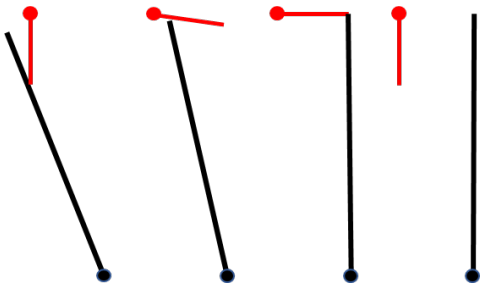


Chess Clock revisited

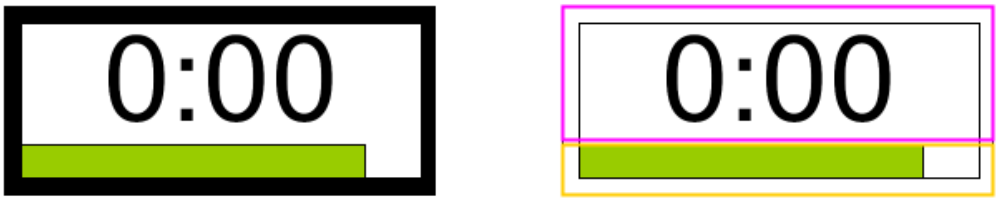
We're getting closer to having a fully working chess board players can use. However, the clock we crafted in *Getting Started* is not really that useful as a chess clock. For one, chess game durations, even those of *fast* chess games, are hardly ever in the order of seconds, but more like hours and minutes. For another, chess players require strong concentration. They can't be distracted with a countdown of seconds left. To remedy this, we will have our clock display hours and minutes and not seconds. We will use a colon to separate the two, and have it flash on and off every second just to show when the clock is actually running.

Nevertheless, during the last few minutes, chess players need to have some idea of what fraction of a minute is still available. This is more of a qualitative indication than a quantitative one. So chess clocks have a small red hand-like indicator around minutes 57. It gets lifted up gradually by the minutes hand, until at the hour, this indicator falls down vertical. The angle of this hand gives a clearer indication to the chess player on how much time he still has.

The sequence of images below, with just the minutes hand and the time-up indicator hand illustrates this:



We will try something simpler for the last minute. We will use a **guage** that will appear in the last minute only. And since there will be two clocks, one for each player, we will want to provide some indication on which clock is which. We will do this by having the clock surrounded with a white or black border:



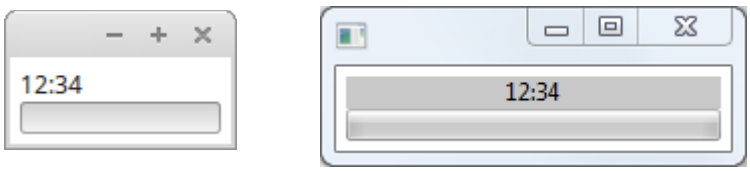
We'll want a clock like the one on the left. And this can be built with just two widgets, as can be seen on the right: a static text and a gauge. We will put the two widgets in a vertical box sizer, each with a border on just three of the four sides. We'll make the underlying panel the colour of the player's pieces so the border turns out the right colour.

Let's try it in a small help module:

```
init([]) ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), -1, ""),
    wxFrame:setBackgroundColour(Frame, {255,255,255}),
    Sz = wxBoxSizer:new(?wxVERTICAL),

    T = wxStaticText:new(Frame, -1, "12:34", [{style, ?wxALIGN_CENTRE}]),
    wxStaticText:setBackgroundColour(T, {200,200,200}),
    wxSizer:add(Sz, T, [{flag, ?wxLEFT bor ?wxTOP bor ?wxRIGHT bor ?wxEXPAND},
{border, 5}]),

    Gauge = wxGauge:new(Frame, -1, 60, [{style, ?wxGA_HORIZONTAL bor ?
wxGA_SMOOTH}, {size, {-1, 16}}]),
    wxSizer:add(Sz, Gauge, [{flag, ?wxLEFT bor ?wxBOTTOM bor ?wxRIGHT bor ?
wxEXPAND}, {border, 5}]),

    wxFrame:setSizer(Frame, Sz),
    wxSizer:setSizeHints(Sz, Frame),

    wxFrame:show(Frame),
    {Frame, #state{}}.
```

Static Text quirks

If we compile and run this, we'll be in for a surprise under Linux (the one on the left).

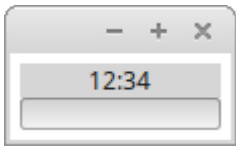


The static text in the Linux version is neither correctly aligned, nor has the correct backgorund colour. The underlying GTK in Linux does not support text widgets with their own background panel. GTK automatically resizes the text widget to the size of the text, so even though the text *is*

correctly aligned *internally*, it does not show. To make it work, we have to provide our own background panel.

```
%% in chess_clock.erl

init([]) ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), -1, ""),
    wxFrame:setBackgroundColour(Frame, {255,255,255}),
    Sz = wxBoxSizer:new(?wxVERTICAL),

    BGPanel = wxPanel:new(Frame),
    T = wxStaticText:new(BGPanel, -1, "12:34", []),

    TSz = wxBoxSizer:new(?wxVERTICAL),
    wxSizer:add(TSz, T, [{flag, ?wxALIGN_CENTRE}]),
    wxPanel:setSizer(BGPanel, TSz),

    wxSizer:add(Sz, BGPanel, [{flag, ?wxLEFT bor ?wxTOP bor ?wxRIGHT bor ?
wxEXPAND}, {border, 5}]),

    Gauge = wxGauge:new(Frame, -1, 60, [{style, ?wxGA_HORIZONTAL bor ?
wxGA_SMOOTH}, {size, {-1, 16}}]),
    wxSizer:add(Sz, Gauge, [{flag, ?wxLEFT bor ?wxBOTTOM bor ?wxRIGHT bor ?
wxEXPAND}, {border, 5}]),

    wxFrame:setSizer(Frame, Sz),
    wxSizer:setSizeHints(Sz, Frame),

    wxFrame:show(Frame),
    {Frame, #state{}}.
```

And this time around, we get what we desired:



Keeping the board square

So now we have the board, the two clocks and of course a push button to say the player has moved, let's see how to put them all together into a frame. This is how we suppose it will look like:

We could achieve this with a vertical box sizer whose first element is a horizontal box sizer with the two clocks. We'll let the frame represent the chess player and put the code in the module `chess_player.erl`. In starting it, we'll pass it the player's name to use as the frame caption. Here's what `chess_player:init/1` would look like:

```
%% in chess_player.erl

start_link(PlayerName) ->
    wx_object:start_link(?MODULE, [PlayerName], []).

init([PlayerName]) ->
    wx:new(),
    Frame = wxFrame:new(wx:null(), ?wxID_ANY, PlayerName),
    Sz = wxBoxSizer:new(?wxVERTICAL),

    Ck1 = chess_clock:start_link(Frame, self()),
    Ck2 = chess_clock:start_link(Frame),

    CkSz = wxBoxSizer:new(?wxHORIZONTAL),
    wxSizer:add(CkSz, Ck1, [{proportion, 1}, {flag, ?wxEXPAND}]),
    wxSizer:add(CkSz, Ck2, [{proportion, 1}, {flag, ?wxEXPAND}]),
    wxSizer:add(Sz, CkSz, [{flag, ?wxEXPAND}]),

    Board = chess_board:start_link(Frame, self()),
    wxSizer:add(Sz, Board, [{proportion, 1}, {flag, ?wxEXPAND}]),

    Button = wxButton:new(Frame, -1, [{label, "Moved"}]),
    wxButton:disable(Button),
    wxSizer:add(Sz, Button, [{flag, ?wxEXPAND}]),

    wxFrame:setSizer(Frame,Sz),
    wxFrame:layout(Frame),

    wxFrame:show(Frame),

    {Frame, #{name => PlayerName,
            frame => Frame,
            my_clock_pid => wx_object:get_pid(Ck1),
            other_clock_pid => wx_object:get_pid(Ck2),
            board_pid => wx_object:get_pid(Board),
            board => Board,
            button => Button}}.
```

We won't try it just yet. Let's first try to make the board square. At least when we start chess_player.

We do this by taking the bigger of the board's width and height and readjusting the size of the frame by that amount so the board comes out square.

```
%% in chess_player.erl

    {FW, FH} = wxFrame:getSize(Frame),
    {BW,BH} = wxPanel:getSize(Board),
    Dw = max(BW,BH) - BW,
    Dh = max(BW,BH) - BH,
    wxFrame:setSize(Frame, FW + Dw, FH + Dh),
```

Beware. We need to set the size of the frame *after* "showing" the frame.

Handling key events

While we are at it, let's also provide a means to make the board bigger or smaller while keeping it square. Since the players won't often be changing the size, let's resize the board with some key combination: Ctrl+ to increase the size and Ctrl- to make it smaller.

All we need do is subscribe to the keystroke event and when the player presses Ctrl+, we increase the size of the frame. First, let's look at the wx documentation for the key event:

```
Use wxEvtHandler:connect/3 with EventType:
        char, char_hook, key_down, key_up
See also the message variant #wxKey{} event record type.
```

And #wxKey{} is defined as:

```
wxKey() =
        #wxKey{type=wxKeyEventType(), x=integer(), y=integer(),
                keyCode=integer(), controlDown=boolean(),
                shiftDown=boolean(), altDown=boolean(), metaDown=boolean(),
                scanCode=boolean(), uniChar=integer(), rawCode=integer(), rawFlags=integer()}
```

So we subscribe to the **`key_up`** event in `chess_player`'s `init/1`:

```
    wxFrame:connect(Frame, key_up),
```

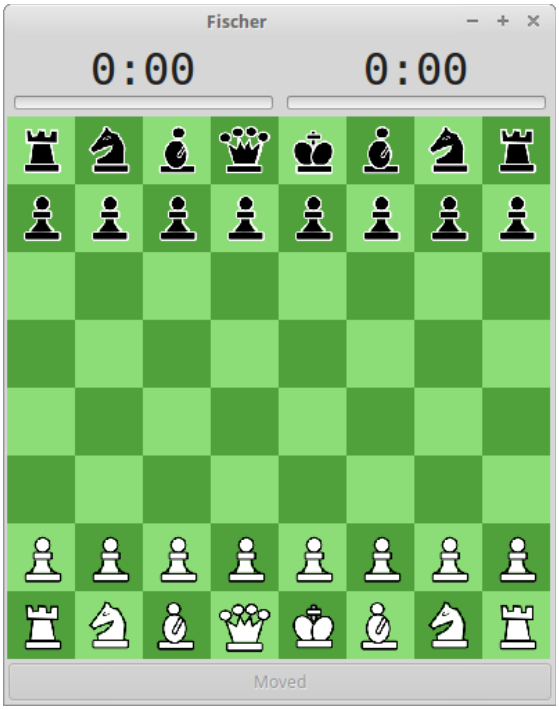
and then handle that event in `handle_event/2`:

```
handle_event(#wx{event=#wxKey{keyCode = KeyCode, controlDown = true}},
             State = #{frame := Frame}) ->
    {W,H} = wxFrame:getSize(Frame),
    case KeyCode of
      $+ -> wxFrame:setSize(Frame,W+8,H+8);
      $- -> wxFrame:setSize(Frame,W-8,H-8);
      _ -> ok
    end,
    {noreply, State};

handle_event(#wx{} = _Wx, State) ->
    {noreply, State}.
```

It's about time to see if our design[18] worked.

```
4> chess_player:start_link("").
{wx_ref,35,wxFrame,<0.137.0>}
5>
```

18 game4/



The frame's there alright, but the clocks don't have white or black borders. Come to think of it, we didn't inform the clocks which colour to paint their borders with, did we? We'll fix that soon enough.

The board does start square. Let's try Ctrl+ or Ctrl- to resize it. Huh? No joy!

Keyboard focus

Without making it too mysterious, wxWidgets sends key events to the widget that has the **focus**. We didn't say which widget should get the focus, did we?

To set the focus on a widget we need to apply `wxWindow:setFocus/1`. However, make sure you set the focus only on widgets that *can* handle key events. wxFrames, for example, *do not* handle key events. Even then, when we set the focus, it is actually given to one of the child widgets, if any. The last one to get painted, as the matters go.

In our case with a white chessboard, that would be the square at the location {0,0} or the top-left corner. So, ok, we will set the keyboard focus on the board itself. And we have to remember another thing as well. Key events are not ***command events***, so they don't get propagated automatically to the parent widgets.

It may not make too much sense because it is written for the C++ binding, but the wxWidgets documentation does give us some useful information on this:

> ... events set to propagate (See: wxEvent::ShouldPropagate) (most likely derived either directly or indirectly from wxCommandEvent) will travel up the containment hierarchy from child to parent until the maximal propagation level is reached or an event handler is found that doesn't call event.Skip().

So there's our problem solved. We need to set the focus on the *board*, handle the key event in `chess_square` and propagate it to `chess_player` through `chess_board`.

Still, there's a problem. We won't be using `wxEvent:resumePropagation/2` as it applies to `wxEvent` objects and we are not using callbacks to handle the event. If we stop to think a little bit, we'll realize that with wx_object's way of capturing the event, we are in the Erlang world, and can easily "propagate" the event to the parent widget by means of Erlang message passing.

```
%% in chess_player:init/1
    wxPanel:setFocus(Board)

%% in chess_square:init/1
    wxPanel:connect(Panel, key_up),

%% in chess_square.erl
handle_event(#wx{event = #wxKey{} = Event}, State = #{board_pid := BoardPid})
->
    BoardPid ! {propagate_event, Event},
    {noreply, State};

%% in chess_board.erl
handle_info({propagate_event, Event}, State = #{player_pid := PlayerPid}) ->
    PlayerPid ! {propagate_event, Event},
    {noreply, State};

%% in chess_player.erl
handle_info({propagate_event, #wxKey{keyCode = $+, controlDown = true}},
          State = #{frame := Frame}) ->
    {W,H} = wxFrame:getSize(Frame),
    wxFrame:setSize(Frame,W+8,H+8),
    {noreply, State};
handle_info({propagate_event, #wxKey{keyCode = $-, controlDown = true}},
          State = #{frame := Frame}) ->
    {W,H} = wxFrame:getSize(Frame),
    wxFrame:setSize(Frame,W-8,H-8),
    {noreply, State};
handle_info({propagate_event, _Event}, State) ->
    {noreply, State};
```

Let's try[19] again. This time we should be able to resize the square using the key combinations Ctrl+ and Ctrl-.

In the next section, we will finally design the players' interactions and start a real game between two players.

19 game5/

Playing the game

Now let's figure out how the game will get played. We have two players, each on a different computer. So first thing, they must "see" each other. We will do that using distributed Erlang. However, we can't have everyone in the world know each other, so we'll set up the communication through a third party, an arbiter, which could run on a "known" Erlang node. We'll develop the arbiter in the module `chess_arbiter` and will register it as such in the global registry.

When `chess_player` is started, no game is being played and we still do not know if the player will play black or white. So, we will paint the board a different colour to indicate it is not yet active. We will indicate this in the square's state and so will pass the square colours when not active in the square's constructor:

```
%% in chess_square.erl

start_link(Location, BoardPid, Parent, ActiveBrush, InactiveBrush,
SelectedBrush) ->
    wx_object:start_link(
      ?MODULE, [Location, BoardPid, Parent, ActiveBrush, InactiveBrush,
SelectedBrush], []).

%% in init/1
    State = #{
      location => Location,
      board_pid => BoardPid,
      square_panel => Panel,
      image => none,
      active_brush => ActiveBrush,
      inactive_brush => InactiveBrush,
      selected_brush => SelectedBrush,
      selectable => false,
      selected => false,
      active => false},
```

When painting the square, we will use the active or inactive brush depending on if `active` is true or false. Initially, as we can see above, the squares will be in the inactive state because there is no game being played yet.

We will construct the active and inactive brushes for white and black in `chess_board` and pass the appropriate brushes in the `square` constructor:

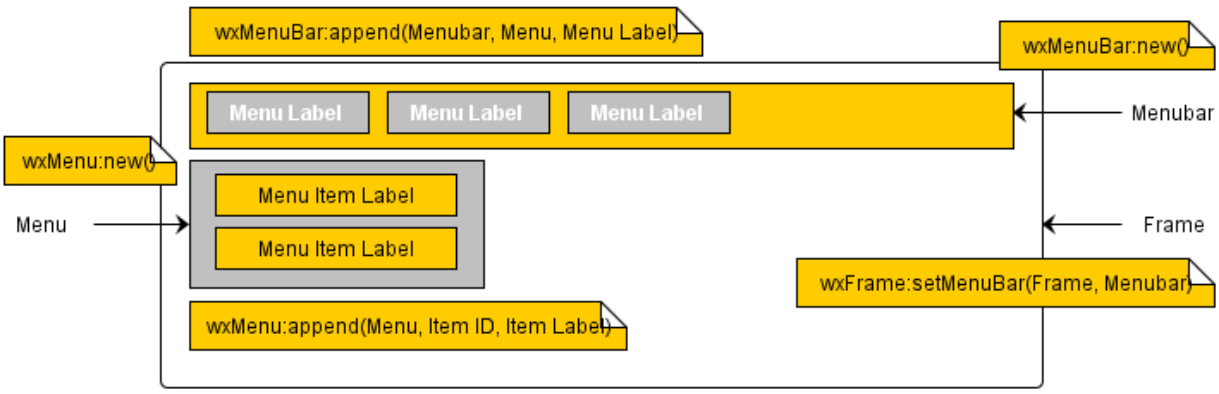
```
%% in chess_board.erl

    ActiveWhiteBrush = wxBrush:new(?WHITE),
    ActiveBlackBrush = wxBrush:new(?BLACK),
    InactiveWhiteBrush = wxBrush:new({150,150,150}),
    InactiveBlackBrush = wxBrush:new({100,100,100}),

MkSquare =
      fun(C,R) ->
            SquareColour = ?UTILS:square_colour(C,R),
            ActiveBrush = case SquareColour of
                            white -> ActiveWhiteBrush;
                            black -> ActiveBlackBrush
                          end,
            InactiveBrush = case SquareColour of
                              white -> InactiveWhiteBrush;
```

```
                            black -> InactiveBlackBrush
                        end,
            Square = ?SQUARE:start_link(
                        {C,R},
                        self(),
                        Board,
                        ActiveBrush,
                        InactiveBrush,
                        SelectedBrush),
            {{C,R}, Square}
    end,
```

Once the game starts, the player will be either black or white. This, we will call the player's `role` and will keep in the `chess_player` state. Initially, it will be `undefined`.

Menus

We will let the player request to play a game through a `Game` menu in a menu bar we will add to the frame.




```
%% in chess_player:init/1

    MB = wxMenuBar:new(),

    GameMenu = wxMenu:new([]),
    wxMenu:append(GameMenu, ?wxID_NEW, "&New Game"),
    wxMenu:appendSeparator(GameMenu),
    wxMenu:append(GameMenu, ?wxID_EXIT, "&Quit"),

    BoardMenu = wxMenu:new([]),
    wxMenu:append(BoardMenu, ?wxID_ZOOM_IN, "&Bigger\tCtrl++"),
    wxMenu:append(BoardMenu, ?wxID_ZOOM_OUT, "&Smaller\tCtrl+-"),

    HelpMenu    = wxMenu:new([]),
    wxMenu:append(HelpMenu, ?wxID_ABOUT, "&About"),
    wxMenu:append(HelpMenu, ?wxID_HELP, "&Help"),

    wxMenuBar:append(MB, GameMenu, "&Game"),
    wxMenuBar:append(MB, BoardMenu, "&Board"),
    wxMenuBar:append(MB, HelpMenu, "&Help"),

    wxFrame:setMenuBar(Frame,MB),
```

Here we create three menus in the menu bar: Game, Board and Help. The Game menu has items *New Game* and *Quit*. The Board menu has items *Bigger* and *Smaller* to resize the board. The Help menu has items *About* and *Help*.

Menu items have identities, some of which are pre-defined and a helper macro is available (eg ? `wxID_EXIT`). In Linux, when using these identities, icons and keyboard shortcuts are automatically added.

An ampersand sign in the label is used to provide a keyboard mnemonic. It lets the letter following the ampersand to be underlined (in Windows), and the key corresponding to that letter to be used in combination with the *alt* key to invoke the item. The menu label can include the keyboard shortcut using the tab character (\t). Here we have added those for the items *Bigger* amd *Smaller*.

The *menu bar* is attached to the *frame* with `wxFrame:setMenuBar/2`.

Please note that *menu items* are appended to *menus*, while *menus* are appended to the *menu bar*. A *separator* can be appended to a menu and will simply draw a separating line in the menu.

One thing to keep in mind is that we do not need to destroy the *menu* or *menu items*. The *menu bar* destructor, and thus the *frame*'s destructor, will take care of that. In fact, according to the wxWidgets documentation:

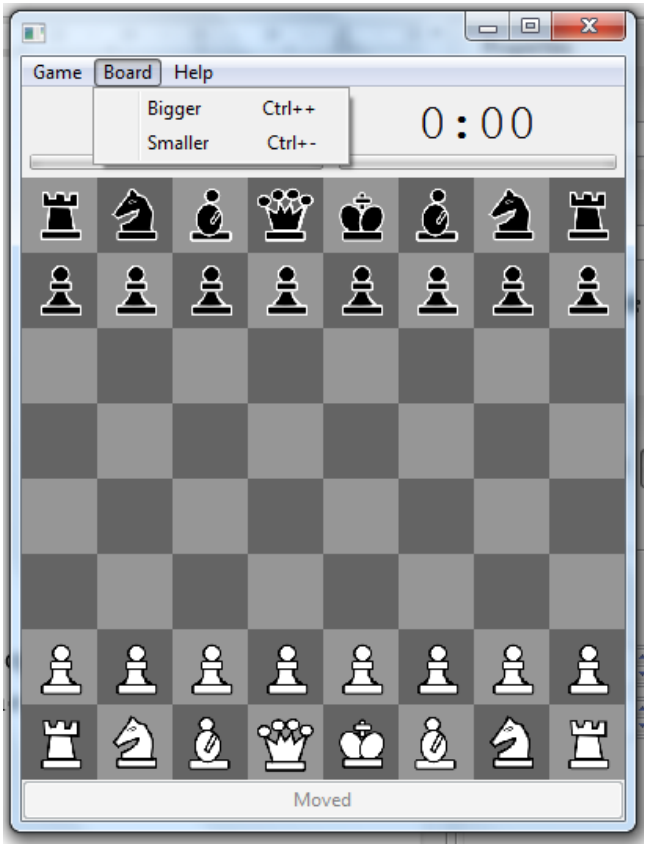
> All menus attached to a menubar or to another menu will be deleted by their parent when it is deleted. As the frame menubar is deleted by the frame itself, it means that normally all menus used are deleted automatically.

If you are using MacOS X or PalmOS, you may want to keep in mind that:

> *wxID_ABOUT* and *wxID_EXIT* are predefined by wxWidgets and have a special meaning since entries using these IDs will be taken out of the normal menus under MacOS X and will be inserted into the system menu (following the appropriate MacOS X interface guideline). On PalmOS *wxID_EXIT* is disabled according to Palm OS Companion guidelines.

So let's see[20] how that turns out to be before going any further.

20 game6/



It seems like the menus are there alright. The board does display all greyed out because the game has not yet started. Cool. Let's move on to handling the menu.

When the user selects a menu item, an event is generated. We need to subscribe to that event in order to handle it. Let's check in the `wxMenuEvent` documentation what we need to subscribe to:

> Use wxEvtHandler:connect/3 with EventType:
>
> menu_open, menu_close, menu_highlight
>
> See also the message variant #wxMenu{} event record type.

The type #wxMenu{} is defined as:

> wxMenu() = #wxMenu{type=wxMenuEventType(), menuId=integer(), menu=wxMenu:wxMenu()}

`menu_open`, `menu_close` or `menu_highlight` do not seem like what we were expecting, do they? So let's look at the wxWidgets documentation for some inspiration:

> ## wxMenuEvent
>
> This class is used for a variety of menu-related events. Note that these do not include menu command events, which are handled using wxCommandEvent objects.

Ok, that explains it. Back to the wxCommandEvent documentation, we see that:

> Use wxEvtHandler:connect/3 with EventType:
>
> command_button_clicked, command_checkbox_clicked, command_choice_selected, command_listbox_selected, command_listbox_doubleclicked, command_text_updated,

command_text_enter, **command_menu_selected**, command_slider_updated, command_radiobox_selected, command_radiobutton_selected, command_scrollbar_updated, command_vlbox_selected, command_combobox_selected, command_tool_rclicked, command_tool_enter, command_checklistbox_toggled, command_togglebutton_clicked, command_left_click, command_left_dclick, command_right_click, command_set_focus, command_kill_focus, command_enter

See also the message variant #wxCommand{} event record type.

and #wxCommand{} is defined as:

wxCommand() = #wxCommand{type=wxCommandEventType(), cmdString=unicode:chardata(), commandInt=integer(), extraLong=integer()}

So let's subscribe to the *command_menu_selected* event and handle the menu item *Bigger*. Let's also add a printout in the catchall event handler to see what we are not explicitly handling:

```
%% in chess_player:init/1

    wxFrame:connect(Frame, command_menu_selected),

%% in chess_player.erl

handle_event(#wx{id=?wxID_ZOOM_IN,
             event=#wxCommand{type = command_menu_selected}},
           State = #{frame := Frame}) ->
    {W,H} = wxFrame:getSize(Frame),
    wxFrame:setSize(Frame,W+8,H+8),
    {noreply, State};

handle_event(#wx{} = _Wx, State) ->
    io:format("chess_player got event ~p~n", [_Wx]),
    {noreply, State}.
```

If we recompile[21] and run `chess_player:start_link("")`, we can test the menu items. Selecting the menu item *Bigger* under menu *Board* will indeed make the board bigger because we are handling that menu. If, instead, we select the item *Smaller*, we get the printout:

```
got event {wx,5138,
              {wx_ref,35,wxFrame,[]},
              [],
              {wxCommand,command_menu_selected,[],-1,0}}
```

which is practically what we expected.

Let's now retry our key bindings. Ctrl+ makes the board bigger as expected and there is no printout in the shell. Ctrl- makes the board smaller, and we get the printout as if we had selected the menu item *Smaller*. However, to bring the board size back to what it was, we have to make it smaller twice more. In other words, each Ctrl+ triggers the menu Bigger, as well as the keystroke handling, so the board increases in size by 16 pixels.

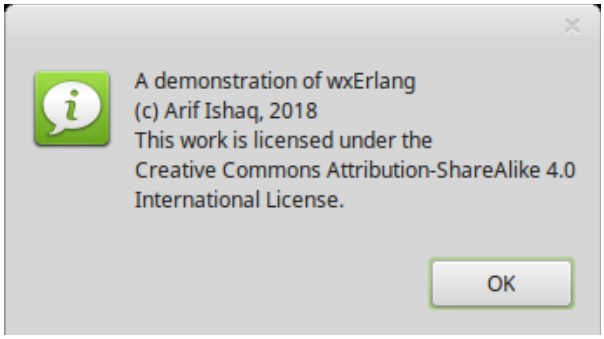
That said, we don't need to handle the key event in chess_square and propagate it up to chess_player through chess_board. Binding the key combination to the menu items serves the same purpose.

21 game7/

About menu

Handling[22] of the About menu is straightforward. We just display some message in a message dialog.

```
handle_event(#wx{id=?wxID_ABOUT,
                 event=#wxCommand{type = command_menu_selected}},
             State = #{frame := Frame}) ->
    M = wxMessageDialog:new(Frame, "A demonstration of wxErlang\n(c) Arif
Ishaq, 2018\nThis work is licensed under the\nCreative Commons Attribution-
ShareAlike 4.0\nInternational License."),
    wxMessageDialog:showModal(M),
    {noreply, State};
```

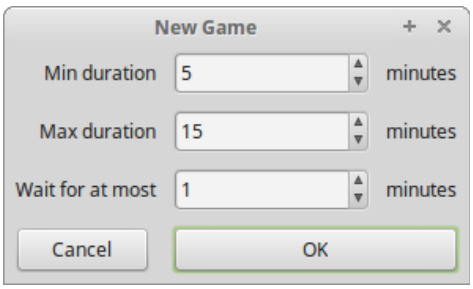


New Game menu

Let's now think about starting the game. We've already said the two players will be on differnt Erlang nodes. And we also know from *Getting Started* that we need an arbiter to coordinate the game. The arbiter will be on some known node to which the players can connect. How should we start the game? At first I had thought it was a simple matter of a player starting the game, but if we let the two players be on different machines, perhaps half a world apart, it is not that simple after all. So, here is what I have implemented. You may want to think of a different scheme.

I let a player connect to the node where the arbiter is running. Then the player lets the arbiter know it is willing to play a game with some other player and indicates the minimum and maximum duration of the game he is willing to play. The player also indicates how long he is willing to wait for some other player to join.

To do this, the player must select *New Game* from the *Game* menu. This will pop up a dialog in which the player can select the minimum and maximum durations and the maximum amount of time he is willing to wait for the other player to join. We will let the player select these values using `wxSpinCtrl` widgets. Here is what our dialog will look like:



22 game8/

We implement it in the module `chess_new_game_dialog`. We employ a `wxGridBagSizer` this time, just to demonstrate how that works.

```
    D = wxDialog:new(Frame, ?wxID_ANY, "New Game"),
    Sz = wxGridBagSizer:new(),
```

In creating the spin controls, we specify the minimum, the maximum and the default value and that the arrow keys be shown:

```
    MinDurationCtrl = wxSpinCtrl:new(D, [{style, ?wxSP_ARROW_KEYS}, {min, 1},
{max, 180}, {initial, 5}]),
```

The text labels are all wxStaticText widgets. We request the sizer to align them right.

Now here's the novelty. To add the widgets to the sizer, we imagine it as a grid and specify the zero-based coordinates of the cell where we want the widget placed:

```
    wxGridBagSizer:add(Sz,
      wxStaticText:new(D, ?wxID_ANY, "Min duration"),
      {0,0},
      [{flag, ?wxALIGN_CENTER_VERTICAL bor ?wxALIGN_RIGHT bor ?wxALL},
       {border, 5}]),

    wxGridBagSizer:add(Sz,
      MinDurationCtrl,
      {0,1},
      [{flag, ?wxALL}, {border, 5}]),
```

And what's more, we can also specify how many columns and rows the widget will occupy. So, for example, we will let the Ok button occupy just one row, but two columns:

```
    OK = wxButton:new(D, ?wxID_OK, [{label, "OK"}]),
    wxButton:setDefault(OK),
    wxGridBagSizer:add(Sz, OK, {3,1},
      [{span, {1,2}}, {flag, ?wxALL bor ?wxEXPAND}, {border, 5}]),
```

Finally, we show the dialog with `wxDialog:showModal/1`.

In the *New Game* menu item handler in `chess_player`, we simply start `chess_new_game_dialog`, which returns a reference to the dialog.

```
%% in chess_player.erl
handle_event(#wx{id = ?wxID_NEW, event = #wxCommand{type =
command_menu_selected}}, State = #{name := Name, frame := Frame}) ->
    D = chess_new_game_dialog:start_link(Frame),
```

However, there's a problem. When we "end" the dialog by pressing Ok or Cancel, the dialog will close, but `chess_player` won't know anything. Even if `chess_player` could somehow block, to capture what `dialog:showModal/1` returned, that would at most say which of the two buttons was pressed and not what the values of the various spin controls were.

We solve both problems by having the `chess_new_game_dialog` module implement the function `get_choice/1` that `chess_player` can call. It will return the user's choice, ok or cancel, and a map of the spin control values in case the user had actually chosen the Ok button.

```
%% in chess_new_game_dialog

get_choice(Dialog) ->
```

```
    gen_server:call(Dialog, get_choice, infinity).

handle_call(get_choice, _From,
          State = #{dialog := D,
                    min_ctrl := MinDurationCtrl,
                    max_ctrl := MaxDurationCtrl,
                    wait := WaitCtrl}) ->
    Choice = case wxDialog:getReturnCode(D) of
                 ?wxID_OK -> ok;
                 ?wxID_CANCEL -> cancel
             end,
    MinDuration = wxSpinCtrl:getValue(MinDurationCtrl),
    MaxDuration = wxSpinCtrl:getValue(MaxDurationCtrl),
    Wait = wxSpinCtrl:getValue(WaitCtrl),

    Reply = {Choice, #{min_duration => MinDuration,
                       max_duration => MaxDuration,
                       max_wait => Wait}},
    {stop, normal, Reply, State}.
```

So when `chess_player` calls

```
chess_new_game_dialog:get_choice(wx_object:get_pid(D))
```

it will block until the `chess_new_game_dialog` returns the user's selection and the spin control values.

wxErlang limitation

Note that the user could select a minimum duration which is greater than the maximum duration. We would like to avoid this by signalling the inconsistency, by say, changing the colour of the durations to red, and disabling the OK button, and so on.

The wx documentation on wxSpinEvent does suggest some events we can subscribe to for this purpose:

> Use wxEvtHandler:connect/3 with EventType:
>
> command_spinctrl_updated, spin_up, spin_down, spin
>
> See also the message variant #wxSpin{} event record type.

However, any such attempt will fail, because there is a limitation in the wxErlang implementation of dialogs. When you use wxDialog:showModal, wx blocks altogether until you close the modal dialog. It clearly says so in the wx documentation:

> Currently the dialogs' *show modal* function freezes wxWidgets until the dialog is closed. That is intended but in Erlang, where you can have several GUI applications running at the same time, it causes trouble. This will hopefully be fixed in future **wxWidgets** releases.

Synchronizing new game requests

Now let's go back to figuring out how to synchronize the game between the two remote players.

`chess_arbiter` will get a request from a `chess_player`, which the `chess_player` will somehow generate after obtaining the player's choices, as we have just seen, and will wait for the maximum time, indicated in the request, for a request to play from a second player.

When a similar request is received from a second player within the maximum time, and this second player is willing to play for the durations compatible with the first player's wishes, the arbiter will inform both players they should prepare to start the game after some time. This, because the first player may be doing something while waiting for a second player. Once that time has passed, the arbiter will choose which colour each player will play, inform them of this so they can set their boards and clocks accordingly, and then tell them to start the game. White will make the first move, so the white clock will be requested to start ticking.

This all seems ok, but there is a problem. How will the arbiter know how to contact the players?

If `chess_player` were to send the request in an Erlang message, it could include its pid in that message. However, the player is on a different node, and his pid, as he can find out with `self/0`, is a pid local to the node he is running on and does not make much sense in another node. The solution I have found is provided by the `gen_server:handle_call` function. According to the wx documentation:

```
Module:handle_call(Request, From, State) -> Result
  Types
  Request = term()
  From = {pid(),Tag}
  State = term()
  Result = {reply,Reply,NewState} | {reply,Reply,NewState,Timeout}
    | {reply,Reply,NewState,hibernate}
    | {noreply,NewState} | {noreply,NewState,Timeout}
    | {noreply,NewState,hibernate}
    | {stop,Reason,Reply,NewState} | {stop,Reason,NewState}
      Reply = term()
      NewState = term()
      Timeout = int()>=0 | infinity
      Reason = term()

From is a tuple {Pid,Tag}, where Pid is the pid of the client
```

There! If `chess_player` were to make the request with a `gen_server:call`, the arbiter could pick up the pid from that call invocation. Let's call that invocation `want_to_play`:

```
%% in chess_player.erl

handle_event(#wx{id = ?wxID_NEW, event = #wxCommand{type =
command_menu_selected}}, State = #{name := Name, frame := Frame}) ->

    D = chess_new_game_dialog:start_link(Frame),

    case chess_new_game_dialog:get_choice(wx_object:get_pid(D)) of
      {ok, #{min_duration := MinDuration,
             max_duration := MaxDuration,
             max_wait := MaxWait}}  ->
          chess_arbiter:want_to_play(Name, MinDuration, MaxDuration, MaxWait);
      _ -> ok
    end,
    {noreply, State};
```

and in `chess_arbiter`:

```
want_to_play(Name, MinDuration, MaxDuration, MaxWait) ->
    gen_server:call({global, chess_arbiter},
                    {want_to_play, Name, MinDuration, MaxDuration, MaxWait}).
```

We won't go into the details of how `chess_arbiter` handles all this, as it has little to do with wxErlang. Besides, you can read the code[23] if you are curious.

If no other player makes a request to play within the maximum time a player is ready to wait, `chess_arbiter` will send it the message **`nomatch`**.

If, on the other hand, a request to play is matched with one from another player, `chess_arbiter` will send the two players a sequence of messages. First the message **`{prepare_to_play, Name2, Duration, StartDelay}`**, where Name2 is the name of the other player, Duration is the number of minutes the game is allowed to last and StartDelay is the number of seconds after which the game will start. Then, after StartDelay seconds, the messages **`{role, Colour, Layout}`**, where Colour is the colour the player will play and Layout is the board layout to start with; and **`{play, Colour}`**, where Colour is the colour whose turn it is to play.

Next we'll see what chess_player will do upon receiving these messages.

Counting down to start

Upon receiving the message to *prepare to play*, we will let `chess_player` do a count down to game start, so that the players will see the game is about to start and stop doing other stuff. `chess_player` will also reset the two clocks to the duration of the game.

```
handle_info({prepare_to_play, _Opponent, Duration, When},
          State = #{frame := Frame,
                    my_clock_pid := MyCk,
                    other_clock_pid := OtherCk}) ->
    chess_countdown:start_link(Frame, When),
    MyCk ! {reset, Duration*60},
    OtherCk ! {reset, Duration*60},
    {noreply, State};
```

Although we did develop a countdown widget in *Getting Started*, let's develop another one using a simple static text, instead of the text control, as we don't need to edit that text.

The reason I choose to develop one with static text is to demonstrate how to centre the text in a frame. If you recall, we said static text in GTK is not associated with a panel, so alignment through styles doesn't work, and we have to align it in a containing widget by menas of a sizer.

In paricular, we use "stretchable" spaces. In a vertical box sizer, we would add them above and below the static text:

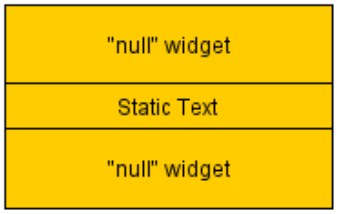
```
    D = wxDialog:new(Frame, ?wxID_ANY, "Count down to start", [{style, ?
wxSTAY_ON_TOP bor ?wxCAPTION bor ?wxCLOSE_BOX}, {size, {200,200}}]),
    Label = integer_to_list(Seconds),
    Sz = wxBoxSizer:new(?wxVERTICAL),
    Count = wxStaticText:new(D, ?wxID_ANY, Label, [{style, ?wxALIGN_CENTRE}]),
```

23 game9/

```
    Font = wxFont:new(64, ?wxFONTFAMILY_TELETYPE, ?wxFONTSTYLE_NORMAL, ?
wxFONTWEIGHT_NORMAL),
    wxStaticText:setFont(Count, Font),
    wxSizer:addStretchSpacer(Sz),
    wxSizer:add(Sz,Count,[{flag, ?wxALIGN_CENTRE}]),
    wxSizer:addStretchSpacer(Sz),
    wxDialog:setSizer(D, Sz),
```

`addStretchSpacer` is actually just a shorthand notation for adding a "null" widget with instructions to the sizer to expand the widget in proprortion.

As we can see from the figure below:



the two "null" widgets will expand equally if the frame is resized and the static text will remain centred.

Setting the player colour

The second message, `{role, Colour, Layout}`, informs the player what his colour will be and what the initial layout of the pieces looks like. Even though the layout is always the same, this allows us to "reset" the layout if we were to start afresh. It also allows us to play variants, like Chess960.

The player forwards the message to the board and sets the colours of the two clocks.

```
%% in chess_player.erl

handle_info({role, MyColour, Layout},
          State = #{board_pid := Board,
                    my_clock_pid := MyCk,
                    other_clock_pid := OtherCk}) ->
    Board ! {role, MyColour, Layout},
    MyCk ! {set_colour, MyColour},
    OtherCk ! {set_colour, chess_utils:opponent(MyColour)},
    {noreply, State#{role => MyColour}};
```

It also sets the state variable `role` to the assigned colour.

The board, in turn, will reconstruct all the squares according to the colour it will play. And since we are now starting a game, it is not longer inactive, and all the squares will be set to the active state, thus changing their colour from the inactive grey to the active colour:

```
%% in chess_board.erl

handle_info({role, Role, Layout},
            State = #{board := Board,
                      make_white_board := MkWhiteBoard,
                      make_black_board := MkBlackBoard,
                      image_map := ImageMap}) ->
```

```
    SquaresMap = maps:from_list(case Role of
                          white -> MkWhiteBoard(Board, #{});
                          black -> MkBlackBoard(Board, #{})
                        end),
    SquarePidMap = maps:map(fun(_,V) -> wx_object:get_pid(V) end, SquaresMap),
    layout_pieces(Layout, ImageMap, SquarePidMap),
    chess_utils:mark(active, true, maps:keys(SquaresMap), SquaresMap),
    {noreply, State#{role => Role,
                layout => Layout,
                square_map => SquaresMap,
                square_pid_map => SquarePidMap}};
```

Handling the turn

The final message, `{play, Colour}`, gets the game going. The players will start the clocks of the colour which is playing, and the player whose turn it is to play, will tell the board to determine which of its pieces can be moved and change the cursor to a hand in the relative squares.

```
%% in chess_player.erl

%% our move
handle_info({play, Colour}, State = #{board_pid := Board,
                              role := Colour,
                              my_clock_pid := MyCk,
                              other_clock_pid := OtherCk}) ->
    Board ! prepare_to_select,
    MyCk ! {ticking, true},
    OtherCk ! {ticking, false},
    {noreply, State};

%% opponent's move
handle_info({play, _Colour}, State = #{my_clock_pid := MyCk,
                               other_clock_pid := OtherCk}) ->
    MyCk ! {ticking, false},
    OtherCk ! {ticking, true},
    {noreply, State};
```

The board has to determine which of its pieces can be moved, but in order to do this it either has to know the rules of the game, or has to ask some process that knows those rules. We implement the rules in the module `chess_rules`.

Chess rules

Apart from some special rules, whether a given piece can be moved and to which positions is determined mainly by the layout of all the pieces. The special rules apply to the pawn, the king and the rook. Bear with me a moment to lay out these rules so we can understand why we have some strange looking state variables in `chess_board`.

If the opponent had moved his pawn two squares trying to avoid being captured by a pawn of the player whose turn it is to play, that pawn can be taken as if it had been moved only one square and not two. This is called taking the pawn *en passant*. So, other than the layout, we must know if the opponent did move his pawn two squares. We keep this information in the board's state variable `en_passant`.

If the king has never moved then the king is allowd to *castle* on the side, queen's or king's, of the rook that has never moved. We keep this information in the board's state variables

`can_castle_queen_side`, `can_castle_king_side`. Actually, this particular information is needed only to determine where the king can be moved to, rather than if it can be moved at all.

If the player moved his king to castle, he must move the relative rook over. So we keep a state variable `castling` and when that is true, the only allowed move is moving the rook over to complete the castling.

If the player had already started castling by moving his king two squares in the direction of a rook, the only possible move for the player is that rook. We keep this information in the state variable `castling`.

Finally, the opponent may have moved a piece in such a way as to put the player under check. We keep this info in the state variable `under_check`.

We won't go through how we implement the chess rules, but if you are curious, you can through the module `chess_rules`. In particular, we implement the function `get_movable/3`, which takes a list of square locations, the board layout and the en passant information to determine which of the pieces in those square locations can be moved.

```
%% a taste of chess rules application

handle_info(prepare_to_select,
          State = #{role := Colour,
                    player_pid := PlayerPid,
                    layout := Layout,
                    square_map := SquareMap,
                    square_pid_map := SquarePidMap,
                    under_check := UnderCheck,
                    castling := Castling,
                    en_passant := EnPassant}) ->
    SelectableSquares =
      case Castling of
          {true, RookLocation} -> [RookLocation];
          false ->
            SameColoured = chess_utils:get_squares(Colour, Layout),
            chess_rules:get_movable(SameColoured, Layout, EnPassant)
      end,

    %% if we can't select any square, it's either a checkmate or a draw
    case SelectableSquares of
      [] ->
          case UnderCheck of
            false ->
                PlayerPid ! {we_end, draw};
            {true, KingLocation} ->
                %% not intuitive, but we need to remove the "under check"
highlighting
                maps:get(KingLocation, SquarePidMap) ! {under_check, false},
                PlayerPid ! {we_end, checkmate}
          end;
      _ ->
          chess_utils:mark(selectable, true, SelectableSquares, SquareMap)
    end,
    {noreply, State};
```

There is a lot of messages going back and forth from one player's board to the other player's board, mediated by the arbiter. This you can see in the code[24], but from wxErlang point of view, there are no further new concepts.

So you may want to go ahead and compile the code and play it with someone. Just to see what it does, whether the rules are obeyed.

You will first need to start a node in which to run the chess_arbiter:

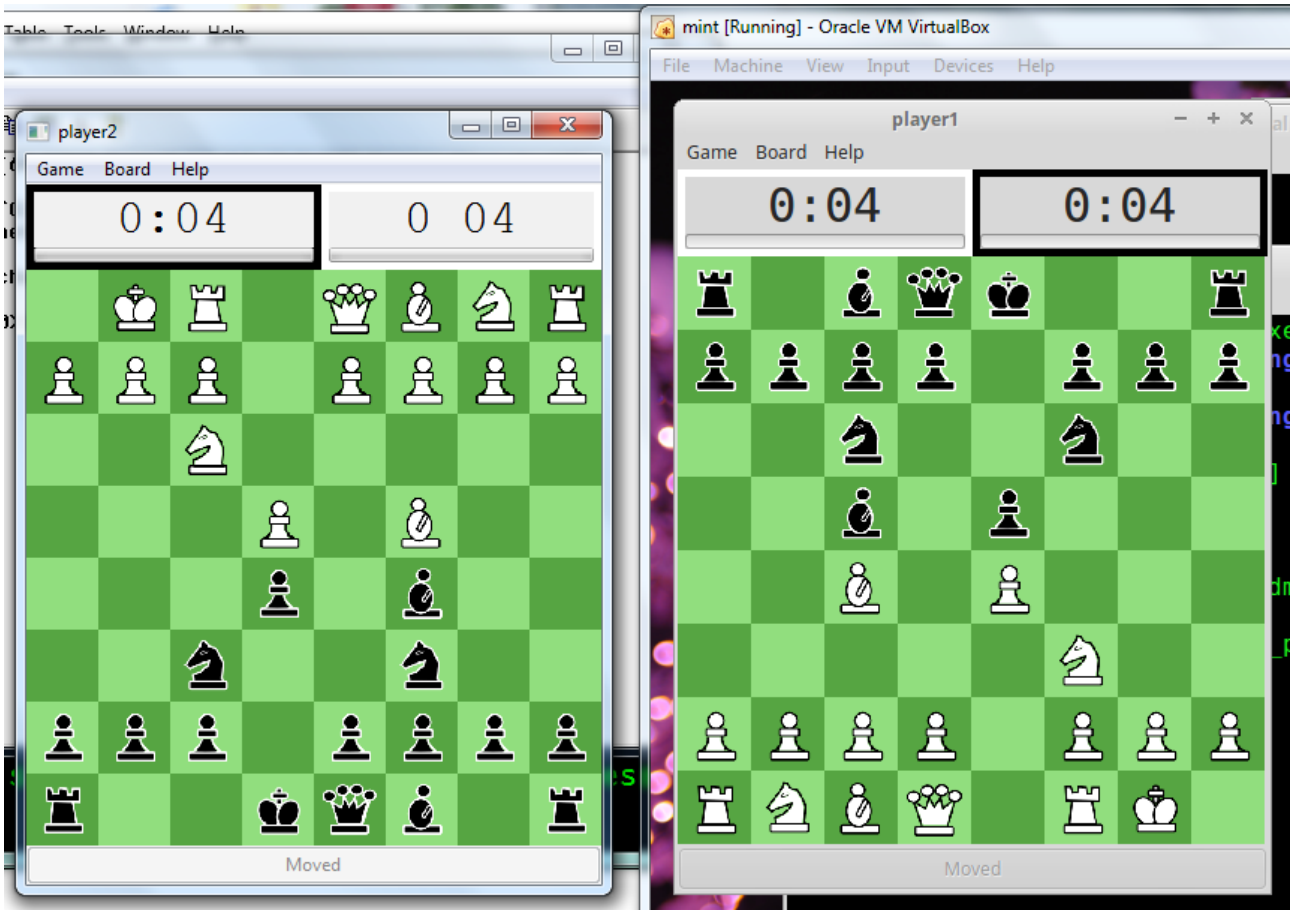
```
$ erl -name arbiter@192.168.56.101 -setcookie chess

Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:2:2] [ds:2:2:10] [async-
threads:10] [hipe] [kernel-poll:false]

Eshell V9.2  (abort with ^G)
(arbiter@192.168.56.101)1> chess_arbiter:start_link().
{ok,<0.69.0>}
(arbiter@192.168.56.101)2>
```

Next, you start two chess_players from two other nodes after connecting the nodes to the arbiter node. Here I start one player in Windows and one in the hosted Linux machine:

```
$ erl -name player1@192.168.56.101 -setcookie chess

Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:2:2] [ds:2:2:10] [async-
threads:10] [hipe] [kernel-poll:false]

Eshell V9.2  (abort with ^G)
(player1@192.168.56.101)1> net_adm:ping('arbiter@192.168.56.101').
pong
(player1@192.168.56.101)2> chess_player:start_link("player1").
{wx_ref,35,wxFrame,<0.75.0>}
(player1@192.168.56.101)3>
```

```
> start werl -name player2@192.168.56.1 -setcookie chess

Erlang/OTP 20 [erts-9.0] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]

Eshell V9.0  (abort with ^G)
(player2@192.168.56.1)1> net_adm:ping('arbiter@192.168.56.101').
pong
(player2@192.168.56.1)2> chess_player:start_link("player2").
making a board
{wx_ref,35,wxFrame,<0.78.0>}
making a board
(player2@192.168.56.1)3>
```

24 game9/



You may want to see what happens when the time is up (in the last minute the gauge should start indicating the end is near, for example).

You may also want to see what happens when a pawn marches all the way to the other side of the board (it should get promoted to a queen, a bishop, a rook or a knight) and how the player is given the choice to select what piece he wants the pawn promoted to.

Try to find out what happens when a user decides to quit from the menu, or when one player checkmates the other.

Enjoy!