Updates in project Fresnel: The Edit control

Michaël Van Canneyt

May 17, 2025

Abstract

Work on project Fresnel, the CSS-driven widget set for Object Pascal, continues. Lots of work in the background, and the start of an edit control.

1 Introduction

Project Fresnel is a new set of native controls for Object Pascal. The goal of Fresnel is to have a new, cross-platform, set of controls, where the styling of the control (the look and feel) is completely handled by CSS. No dependency on the VCL or LCL exists. Currently, Fresnel supports MacOS, windows, Linux and the browser as platforms.

Lately, work has been focused on supporting scrollbars and the edit control.

2 Scrollbars and mouse capture

Often, a control - or widget - has more content than can fit in the surface it has available. In such case you can smply crop the contents, or you can show a scrollbar in order to let the user manipulate the visible portion of the contents.

In order to create a scrollbar, you need some drawing routines to actualy draw the scrollbar, but you also need mouse handling: the user will pull the scrollbar up or down (or left and right) with the mouse. The user's handling of the mouse pointer is not always very precise: it can happen that when dragging the scrollbar's thimb handle, the mouse pointer will move outside the space allocated for the scrollbar. If so, the scrollbar needs to continue to receive events, till the user lets go of the mouse button. This process is called 'mouse capture': although the mouse is moved off an actual control, the control continues to receive the mouse events.

So, in order to be able to create a scrollbar, support for mouse capture has been implemented.

3 Edit control - focus

When you type on the keyboard, the keystrokes are communicated to the control that has focus. If an edit control is to work, it needs to receive the keystrokes when the window is active. The concept of focus did not yet exist in fresnel, so it was implemented. Only the button and edit control are focusable at this moment.

Naturally, controls and other components can be notified when an element gets (or loses) focus. In fact, there are 4 events that can be listened to, all are transmitted

using the `TFresnelFocusEvent` descendent of `TFresnelUIEvent`:

evtBlur (or 'blur') will be triggered on the element that loses focus.

evtFocusOut (or 'focusout') will be triggered on the element that loses focus and all its parents up to the common parent with the newly focused control.

evtFocus (or 'focus') will be triggered on the element that gets focus.

evtFocusIn (or 'focusin') will be triggered on the element that gets focus and all its parents.

Currently, no support exists for tab order: Tab order determines how the focus shifts to the next focusable control when the user presses the TAB key. Support for tab order is planned.

4 Edit control - key strokes

For the user to be able to type in an edit, keyboard strokes must be recorded. The `fresnel.keys` unit contains all keyboard key definitions. For each key, there is a numerical key code, and a key name. Special keys have a negative code, enumerated in the `TKeyCodes` record.

```
TKeyCodes = Record
    Alt                         = SpecialKeysOffset - 0000;
    AltGraph                    = SpecialKeysOffset - 0001;
    CapsLock                    = SpecialKeysOffset - 0002;
    Control                     = SpecialKeysOffset - 0003;
    // etc
  end;
```

Normal key codes which produce a printable (or visible) character, have a positive code: the unicode identifier of the character.

A keystroke is also associated with a name for the key: this will be a human-readable name such as 'Up','Down', 'Left', 'Home', 'end', all are defined in a second record:

```
  TKeyNames = Record
  Const
    Alt = 'Alt';
    AltGraph = 'AltGraph';
    CapsLock = 'CapsLock';
    Control = 'Control';
    // etc.
```

The key-up and key-down keyboard events are handled by the `TFresnelKeyEvent`:

```
TFresnelKeyEvent = class(TFresnelEvent)
  function ShiftState : TShiftState;
  Property Code: ShortString;
  Property Key : String;
  Property NumKey : Integer;
  Property Altkey : Boolean;
  Property MetaKey : Boolean;
```

```
  Property CtrlKey : Boolean;
  Property ShiftKey : Boolean;
end;
```

The (unique) `EventID` property will be either `evtKeyUp` or `evtKeyDown`. The `ShiftState` property is known from the VCL/LCL, the `AltKey`, the `MetaKey`, `CtrlKey` and `ShiftKey` exist for easy access. The `Numkey` contains the numerical key code, and `Key` is the printable value of the key (if it is printable). `Code` contains the key name.

When a key is pressed that produces a printable character or that changes a text, a `evtInput` event is triggered. This event will be triggered for any kind of input: This can be a key from the keyboard, a paste operation from the clipboard, or a drop of a text on an edit control. VCL/LCL users can think of it as the more general equivalent of the keypress event or `WM_CHAR` message.

5 Edit control - clipboard support

Naturally, an edit needs to support copy and paste operations, across applications. Copy and Paste happens through the OS (or the UI layer of the OS) clipboard support. So clipboard support has been implemented for Fresnel, independent of the clipboard support of the LCL. The clipboard support uses MIME types to identify the contents of the clipboard, meaning that for example text on the clipboard is identified with `text/plain`. While the implementation can support any kind of contents, the MIME support is currently in place only for text and image data.

6 Edit control - pseudo elements

The selection color and background in an edit control is controlled by CSS just as any other visual aspect of the edit box. CSS uses a "pseudo element" to allow you to set the color of selected text: this is done because things like selection span over multiple elements, or span only part of an element. Using pseudo elements, the input selection color can be specified as follows in CSS:

```
input::selection {
  background-color: darkblue;
  color: white
}
```

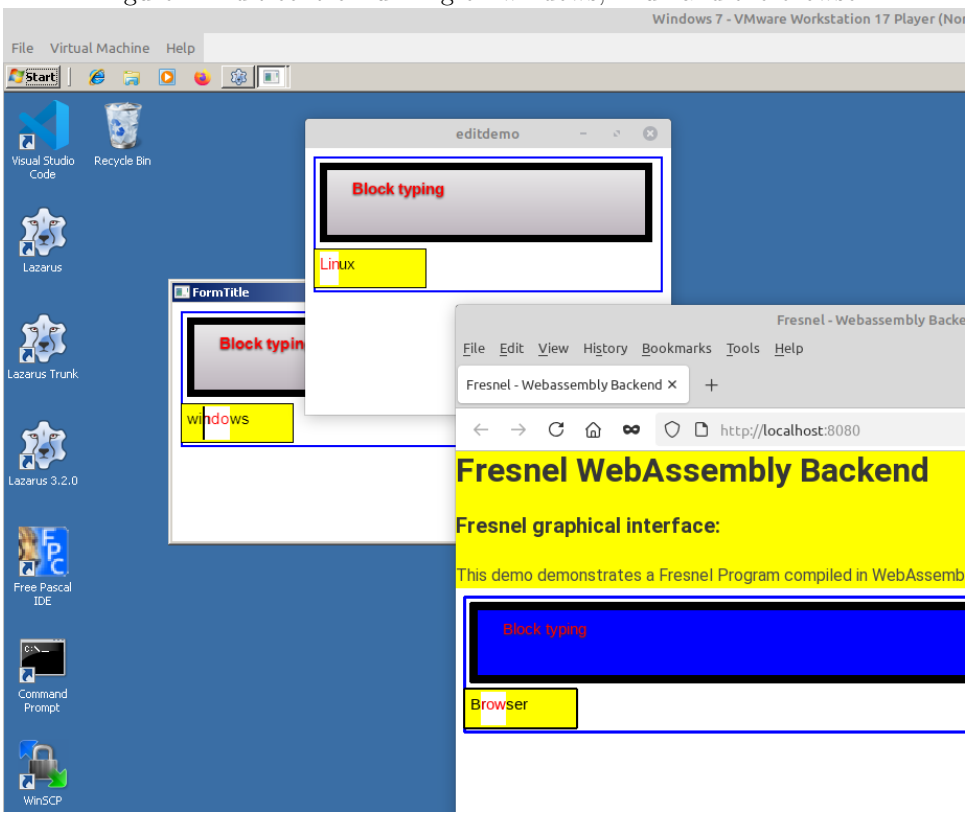
When you create a control and need to get the values of these properties, then you need to actually create a pseudo element:

```
FSelectionElement:=TPseudoElSelection.Create(self);
FSelectionElement.Parent:=Self;
```

The CSS engine of fresnel will populate the CSS values of this pseudo element with the specified values of the style sheet. To retrieve these CSS values, the code is identical to code which retrieves CSS values for your own element:

```
lSelBackColor:=FSelectionElement.GetComputedColor(fcaBackgroundColor,fpimage.colDkBlue);
lSelColor:=FSelectionElement.GetComputedColor(fcaColor,fpimage.colWhite);
```

Figure 1: Edit control running on windows, linux and the browser



With all these improvements in place, the edit control has been implemented, and can be seen in action for 3 platforms in figure 1 on page 4.

The main form in this demo sets the following CSS when starting:

```
Stylesheet.Text:=
  'div {'+
  '  padding: 3px; '+
  '  border: 2px solid black; '+
  '  margin: 6px;'+
  '}'+sLineBreak+
  'input::selection { color: red; background-color: white } ' ;
```

The edit selection color is clearly visible in the figure.

7 Fresnel - webassembly

Lots of improvements have been implemented in the Fresnel webassembly backend:

- Instead of drawing the contents of a window immediately upon invalidation, drawing now happens in a `RequestAnimationFrame` callback: the requestanimationframe is an event triggered by the browser when it is ready to update the display.
- Double buffering: instead of drawing directly on a HTML canvas element,

first everything is drawn on an offscreencanvas, which is then drawn on the actual canvas.

- Support for threaded applications: Thanks to the double buffering mechanism, the Javascript side of the Webassembly Fresnel widgetset can run in a web worker: The drawing on an offscreen canvas can be done in a web worker. The offscreen canvas is then transferred to the HTML page main thread, and drawn in the `RequestAnimationFrame` callback.
- Font support and unicode character support has been improved.
- Window sizing/Positioning and support for multiple windows has been completed.

To enable threaded applications, you need to create the fresnel API in the Web Worker Main and Thread Runner using the `TWasmFresnelWorkerApi` class from the `fresnel.worker.pas2js.wasmapi` unit

```
FFresnelAPI:=TWasmFresnelWorkerApi.Create(WasiEnvironment);
```

In the main program you must use the `TWasmFresnelWebApi` class from the `fresnel.web.pas2js.wasmapi` unit:

```
FFresnelAPI:=TWasmFresnelWebApi.Create(WasiEnvironment);
```

What will happen is that the Worker version of the Fresnel API will forward many requests to the main thread version of the Fresnel API.

8 FMX using Fresnel Webassembly backend

As reported in a previous contribution, FPC can compile FMX and FMX programs for webassembly, where the webassembly backend of FMX uses Fresnel to do the drawing.

The result of all the improvements mentioned above, is that the FMX controls are now fully functional. The 'controls' demo of FMX has been recompiled, and is available at

`https://idefix.freepascal.org/~michael/fmx/controls/`

showcases this, as shown in figure 2 on page 6

The patches that must be applied to FMX in order to be able to compile FMX with FPC have been published at:

`https://gitlab.com/freepascal.org/fmx-using-fpc`

The host environment has been updated to cater for the changes in the webassembly Fresnel backend.

9 Conclusion

Work on Project Fresnel is continuing on several fronts: on the one hand, the backends are improved, on the other hand a set of basic controls is being implemented. The whole of Fresnel is not yet ready for actual use, but one by one, the controls needed to create a real-life application are made. The next step are some dialogs such as the open/save file dialogs.

Figure 2: FMX controls demo in the browser