Project Fresnel Update

Michaël Van Canneyt

May 22, 2024

Abstract

At the end of the year 2022, Project Fresnel was announced: a new graphical interface for Pascal applications, based on CSS. Since then, work has been steadily progressing on this new framework. In this article an overview of what is possible today is presented.

1 Introduction

Project Fresnel was announced in this magazine a little over 1.5 years ago. Work was started immediatly, and work on project Fresnel has not stopped since.

As a reminder, the main goals of project Fresnel were:

- To create a set of controls (or widgets) that are streamable, so descendents of TComponent: the widgets can be manipulated in the IDE.
- Layout is determined completely by CSS.
- Multiple drawing backends must be supported.
- No dependency on the Lazarus LCL.
- Fresnel-Based Forms can coexist with LCL forms in a native application.

The end goal is a UI framework that will allow to create an application UI once, and let it run on any OS and in the browser. Conceptually, the architecture of such an application is depicted in figure 1 on page 2.

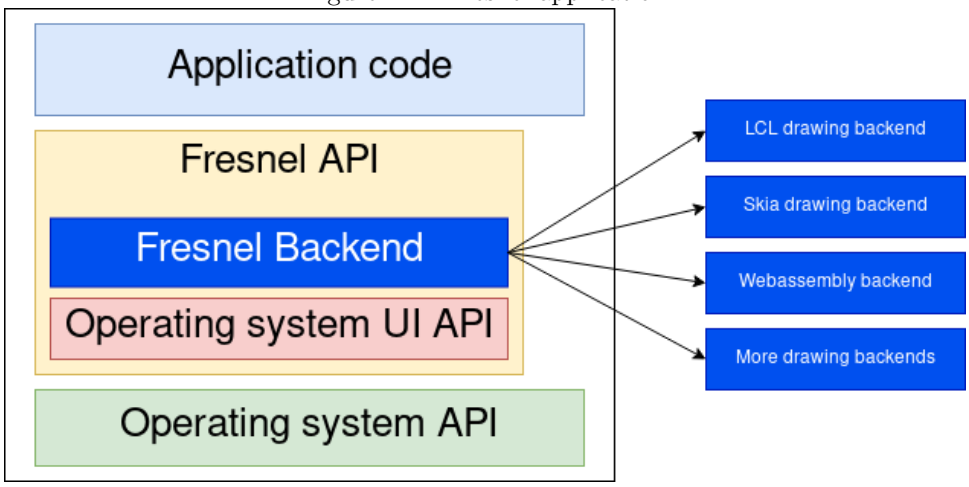
The application code only uses the Fresnel API to display the graphical user interface, other functionality is implemented on top of the operating system API. Fresnel components use the Fresnel backend to do the actual drawing. A Fresnel backend uses the graphical API of the operating system – or a library that makes the drawing easier – to do the actual drawing. Fresnel components do not access the APIs underlying the backend. This ensures that Fresnel components will work with any backend. Several backends can be implemented, and when running your application, you choose the backend in function of the operating system for which you're compiling your application.

In this article, we report on the progress made on each of these goals.

2 Widgets or Controls

A basic set of controls (widgets) has been developed:

Figure 1: A Fresnel application



ViewPort This essentially encapsulates the visible portion of a form. It is the top-level control in a Fresnel graphical window, and has a stylesheet associated with it that determines the style of the elements in a form.

Form is a descendent of a viewport. This is a viewport which can exist by itself.

Div is a basic building block of a graphical UI: a box for which you can specify sizes, borders, background and foreground colors etc.

Span is similar to a `Div` but has different layout flow behaviour: spans will be placed next to each other ('inline' display, in CSS terms).

Label Is exactly what the name implies: it resembles a `Div` but allows you to specify a caption to be shown in the box.

Image A component to show an image.

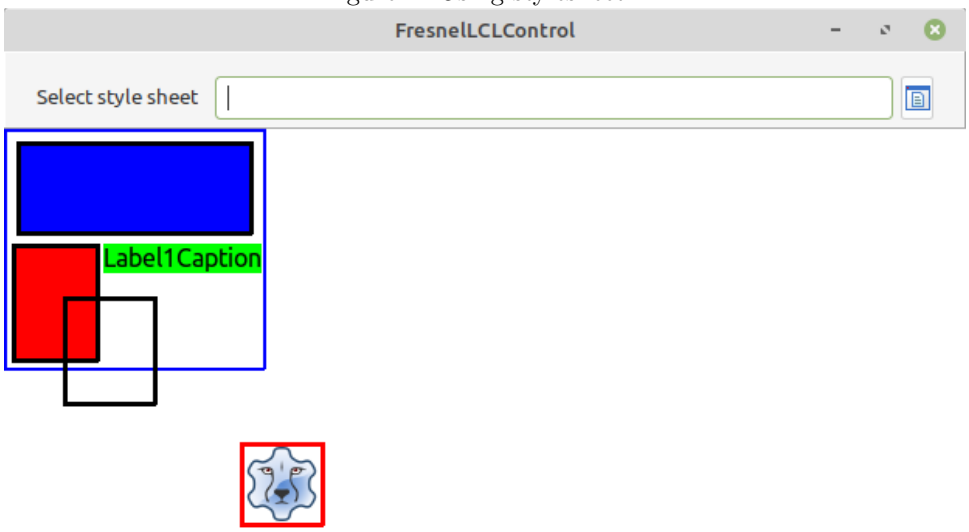
This means that today you can do the following:

```
Div2:=TDiv.Create(Self);
with Div2 do
  begin
  Name:='Div2';
  Parent:=Body1;
  Style:='border-color: black; height:50px; '+
         'position: absolute;  border: 2px; '+
         'left: 30px; top: 100px; width: 50px; '+
         'height: 60px;';
  end;
```

As you can see, the layout of the component is determined by the `Style` property.

Furthermore, these controls can be installed in the IDE, and you can create a Fresnel Form in the designer. This part is still experimental. To do so, you need to recompile the Lazarus IDE with the trunk version of Free Pascal, as project Fresnel requires the use of some units that are not yet present in the released version of Free Pascal..

Figure 2: Using stylesheet 1



3 CSS Layout

The primary goal of project Fresnel is to have the layout determined by CSS. CSS originated in the browser, and became a powerful tool for creating good-looking UIs which is used in all browsers.

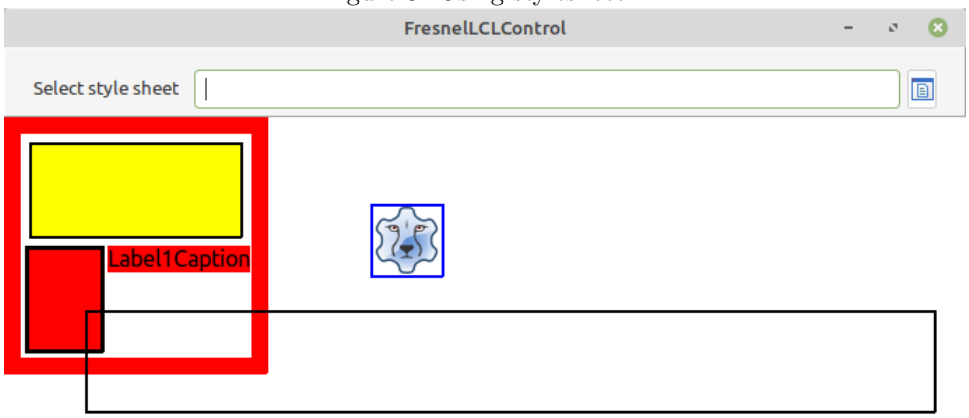
Prior to starting Project Fresnel, Free Pascal already had a CSS parser available. This parser was extended to make it more robust, and an engine was developed to determine the CSS properties that are applicable to a given widget (control). So today, we can specify the CSS of a control using the `Style` property, or using the style sheet of the viewport: the stylesheet can be specified in the `StyleSheet` property.

```
Viewport.Stylesheet.LoadFromFile('style2.css');
```

Figures figure 2 on page 3 and figure 3 on page 4 show the same application, but with a different stylesheet loaded. As expected, the controls adjust their properties (and location) according to what is specified in the CSS.

Needless to say, there is still a lot of work to be done: there are many CSS properties, and currently only the most basic properties are implemented: enough to create simple layouts, without too much of the special effects that make CSS such a powerful mechanism.

Figure 3: Using stylesheet 2



4 Fresnel Event handling

In the LCL, the event handlers such as `OnClick` `OnMouseMove` can be assigned in the Object Inspector. The same is true for Fresnel widgets: Fresnel is designed from the start to be RAD-enabled. While not specifically specified in the goals of Fresnel, the occasion was to used to address some of the shortcomings of the VCL and LCL event mechanisms were addressed, and a complete redesign of the event mechanism was put in place.

The first thing to mention about the new event mechanism is that the signature of the event handlers is different from in the LCL.

The LCL uses the following notification event handler (with slight variations):

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Here, `Sender` is the component instance from which the event originates. You need to typecast the `sender` to access its properties.

The basic event handler in Fresnel looks like this:

```
TEventHandler = Procedure(Event : TAbstractEvent) of object;
```

The `Event` parameter is of type `TAbstractEvent`, which looks like this:

```
TAbstractEvent = Class(TObject)
  // Sender of the event
  Property Sender : TObject Read FSender Write FSender;
  // Event ID used for create
  Property EventID : TEventID Read FEventID;
end;
```

The `Sender` is still available as a member of the event.

Depending on the actual event, a descendent of `TAbstractEvent` is passed on, which contains the necessary information pertaining to the event. For instance, the mouse events all descend from `TFresnelMouseEvent` :

```
TFresnelMouseEvent = Class(TFresnelUIEvent)
```

```
Public
  Property ControlX : TFresnelLength;
  Property ControlY : TFresnelLength;
  Property PageX : TFresnelLength;
  Property PageY : TFresnelLength;
  Property ScreenX : TFresnelLength;
  Property ScreenY : TFresnelLength;
  Property X : TFresnelLength;
  Property Y : TFresnelLength;
  Property Buttons: TMouseButtons;
  Property Button : TMouseButton;
  Property ShiftState : TShiftState;
  Property Altkey : Boolean;
  Property MetaKey : Boolean;
  Property CtrlKey : Boolean;
  Property ShiftKey : Boolean;
end;
```

As you can see, a lot more information is available.

But there is more: for every event, multiple handlers can be registered. The basic Fresnel component exposes a `EventDispatcher` property, which is of type `TEventDispatcher`:

```
TEventSetupHandler = Procedure(Event : TAbstractEvent) of object;
TEventSetupCallBack = Procedure(Event : TAbstractEvent);
TEventSetupHandlerRef = Reference to Procedure(Event : TAbstractEvent);

TEventDispatcher = class(TPersistent)
  // Various forms to register an event handler
  Function RegisterHandler(aHandler : TEventCallback;
                           aEventName : TEventName) : TEventHandlerItem;
  Function RegisterHandler(aHandler : TEventHandler;
                           aEventName : TEventName) : TEventHandlerItem;
  Function RegisterHandler(aHandler : TEventHandlerRef;
                           aEventName : TEventName) : TEventHandlerItem;
  // Dispatch an event.
  // Calls the registered handlers for that event,
  // in the order they were registered.
  // Returns the number of handlers that were called;
  Function DispatchEvent(aEvent : TAbstractEvent) : Integer;
end;
```

This is roughly modeled after many other event dispatching mechanisms in other toolkits (Gtk, Qt) and in the browser. There are 2 things to note about this mechanism:

1. You can register multiple handlers for the same event. Behind the scenes, setting the `OnClick` handler will use the event dispatcher to set one 'click' event handler. Setting the event handler to Nil will remove the handler from the dispatcher.

2. Event handlers no longer need to be object methods. It can also be a anonymous method, or a plain procedure or a local procedure.

At the moment, you stll need to typecast the `Event` to get to the properties, but a mechanism using generics to register a correctly typed event handler will be put in place.

5 Backends

Another important goal for Project Fresnel is that it must be cross platform and must support different drawing backends: the widgets or controls are not aware of the backend in use to draw them. All they get is a canvas on which they can draw themselves if so required.

A backend needs to provide 2 services: The first is to manage the top-level windows (form) and the events sent by the operating system, and the second service is to provide a canvas to draw on. These 2 services are defined independently and can be coded independendently. This means that you could have a backend (for example Gdk3/Gtk3, to manage the windows and events) that uses various drawing backends (Skia or Cairo).

Currently, 3 working backends for the Fresnel widgets exist, and a 4th is in the works:

LCL This was the first backend created for Fresnel. The fresnel controls are drawn on an LCL Canvas. This can be a canvas that is embedded in a LCL form on a `TFresnelControl`: this is a control that embeds the Fresnel viewport; all drawing happens within this control. It can also be a fresnel LCL form: a form that is completely standalone. Events are generated by the LCL and are transformed into Fresnel events. It is this backend that is used when designing a Fresnel form in the IDE.

Gtk3 using Skia The Gtk3 backend is a backend which relies on the Skia library to render the Fresnel controls. Skia is a fast 2D library by Google which runs on various platforms (all major OSes and mobile devices). By creating a Skia drawing backend, the Fresnel framework should run on all platforms that Free Pascal, Skia support. Skia by itself does not offer event handling, so it is paired with Gtk3, which is also cross-platform.

WebAssembly Lastly, a WebAssembly backend is made. Free Pascal supports creating webassembly binaries, and these binaries can be run in the browser. A Fresnel backend was made which uses the browser canvas and the browser events to deliver the needed functionality to Fresnel. It will be presented in the rest of this article.

More backends can of course be made: Using one of the existing backends, it should not be difficult to create a backend that sits directly on top of the OS' native UI mechanisms:

- LCL backend with a Skia renderer backend.
- WinApi backend with a Skia renderer backend.
- WinApi backend with a WinApi renderer backend.
- WinApi backend with a BGRA renderer backend.
- Apple Cocoa backend with a Metal renderer backend.
- Apple Cocoa backend with a Skia renderer backend.

One such backend which is planned by the FPC team is Pas2js: This would allow running a fresnel application as a Javascript application.

6 Compiling Fresnel

To compile Fresnel, the development version of FPC is needed: Fresnel uses some mechanisms which are available only in the development version of FPC (for example, the CSS parser).

Fresnel itself is implemented in a series of lazarus packages. You can compile Fresnel and use without the lazarus packages, if you so desire.

FresnelBase this package contains the basics of Fresnel: the controls, the CSS handling, the event mechanism and the rendering backend specification (it is defined as an interface definition). This package does not depend on the LCL - this was one of the design goals.

FresnelLCL This contains the LCL backend for Fresnel: a renderer that can render a Fresnel form on a form or in a LCL control.

FresnelDsgn Installing this package allows you to to design a Fresnel form in the Lazarus IDE, as you would design a LCL form: You can add Fresnel forms to a standalone Fresnel application or a LCL application and drop Fresnel elements onto the fresnel Forms and use the Object Inspector to set properties like the `Style` property.

Then there are 3 other packages that provide other backends:

fresnel This package automatically chooses a backend depending on some defines. On linux it will choose the Gtk and Skia backend to provide a window and an event mechanism. The drawing itself is done using Skia. This is still a work in progress.

fresnelwasm This package contains the webassembly part of the `WebAssembly` backend. The webassembly backend needs 2 parts: one in webassembly, one in the browser. This package contains the webassembly side of the Fresnel webassembly backend.

p2jsfresnelapi This package contains the javascript part of the ¡`WebAssembly` backend, it must be used in the browser host application that loads the Fresnel Webassembly program.

7 A Fresnel application using the LCL

As an example, we'll show a Fresnel application using an LCL form and a `TFresnelLCLControl` to host the Fresnel controls.

The main form's published section only contains an 'OnCreate' handler, the rest is added manually:

```
TMainForm = class(TForm)
  procedure FormCreate(Sender: TObject);
private
  Body1: TBody;
```

```
  Div1, Div2: TDiv;
  Img1 : TImage;
  Span1: TSpan;
  Fresnel1: TFresnelLCLControl;
  label1 : Fresnel.controls.TLabel;
public
  procedure CreateControls(ViewPort: TFresnelViewport);
end;
```

In the `OnCreate` event, we create the LCL TFresnelLCLControl that will host all Fresnel controls. We set it to take all available space, and load a stylesheet:

```
procedure TMainForm.FormCreate(Sender: TObject);

begin
  Fresnel1:=TFresnelLCLControl.Create(Self);
  with Fresnel1 do
  begin
    Name:='Fresnel1';
    Align:=alClient;
    Viewport.Stylesheet.LoadFromFile('style1.css');
    Parent:=Self;
  end;
  CreateControls(Fresnel1.Viewport);
end;
```

In the `CreateControls` method, we create the Fresnel Controls:

```
 Procedure TMainForm.CreateControls(ViewPort : TFresnelViewport);

  Function CreateControl(aClass : TFresnelElementClass;
                         aName : String;
                         aParent : TFresnelElement = nil) : TFresnelElement;
  begin
    if aParent=Nil then
      aparent:=Body1;
    Result:=aClass.Create(Self);
    Result.Name:=aName;
    Result.parent:=aParent;
  end;

begin
  Body1:=TBody(CreateControl(TBody,'Body1',ViewPort));
  Div1:=TDiv(CreateControl(TDiv,'Div1'));
  Span1:=TSpan(CreateControl(TSpan,'Span1'));
  label1:=TLabel(CreateControl(TLabel,'Label1'));
  Label1.Caption:='Label1Caption';
  Div2:=TDiv(CreateControl(TDiv,'Div2'));
  Img1:=TImage(CreateControl(TImage,'Img1'));
  Img1.Image.LoadFromFile('image.png');
end;
```

Note that we do not need to set any position or color properties. This is all taken care of by the CSS.

The `Image` property of the `TImage` widget deserves some extra attention. This property is of class `TImageData`, which is defined as follows:

```
TImageData = class(TPersistent)
Public
  Constructor Create(aOwner : TComponent); virtual;
  Destructor Destroy; override;
  Procedure LoadFromFile(const aFilename : String);
  Procedure SaveToFile(const aFilename : String);
  Procedure LoadFromStream(const aStream : TStream;
                           Handler:TFPCustomImageReader = Nil);
  Procedure SaveToStream(const aStream : TStream;
                         Handler:TFPCustomImageWriter = Nil);
  Procedure Assign(Source : TPersistent); override;
  Property Data : TFPCustomImage;
  Property ResolvedData : TFPCustomImage;
  Property Width : Word;
  Property Height : Word;
  Property HasData : Boolean;
Published
  Property FileName : String;
  Property ImageName : String;
  Property ImageList : TBaseCustomImageList;
  Property ImageIndex : Integer;
end;
```

The LCL and VCL use 2 approaches to specify an image: directly through a `TGraphic` or indirectly using an `ImageList` and an `ImageIndex` property. Which one is used depends on the actual control. In Fresnel, these 2 approaches have been combined in one single class: `TImageData`. Thus, every Fresnel control that needs to specify an image, has both mechanisms enabled.

The `TImageData` offers also a third mechanism to load images: a `ImageName`, which is used to look up an image by name in a central image store. The central image store can look up image files by name, and can handle multiple sizes and multiple screen resolutions. It caches the images in memory. Thus

```
  Img1.Image.ImageName:='image';
```

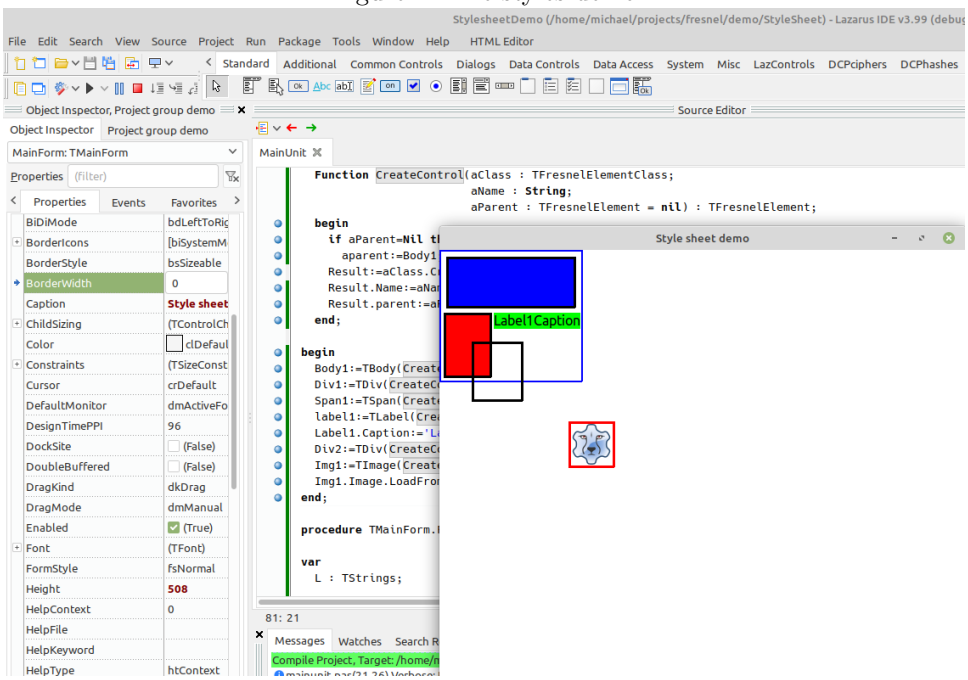
would look for an image file using a standard format ('.png') in a standard set of directories. Both the format and the directory structure are globally configurable. This mechanism makes it easy to configure a set of standard images for an application.

The images are loaded and kept in memory using Free Pascal's `TFPCustomImage` class, which can handle many image formats by default.

The main program file for our program looks like any other program:

```
program StylesheetDemo;

{$mode objfpc}{$H+}

uses
  Interfaces, // this includes the LCL widgetset
  Forms, MainUnit;
```

Figure 4: The styles demo




```
begin
  RequireDerivedFormResource:=True;
  Application.Scaled:=True;
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

It looks like any other Lazarus application.

Running the application will result in an application looking like figure 4 on page 10:

8 Using the webassembly backend

To run a fresnel program in a webassembly backend, you need 2 programs. One is the webassembly program itself, the other is the Javascript program that loads the Webassembly file in the browser. This Javascript program we create of course with pas2js.

We'll start with the webassembly program itself. Due to the nature of webassembly, the fresnel program must be created as a library: As explained in the article on using the browser's API, when the browser runs a webassembly program, it suspends the Javascript execution. As long as the webassembly program runs, no event handlers will run.

So, we need to create a library that initializes the application, and then returns control to the browser, so it can start receiving events. The events are processed in the `__fresnel_tick` callback which is called at regular intervals by the browser.

This method must be exposed by the library:

```
library basic;

uses
  nothreads, fresnel.forms, fresnel.wasm.app, form.main, fresnel.wasm.api;

procedure __fresnel_tick (aCurrent,aPrevious : double);

begin
  fresnel.wasm.api.__fresnel_tick(aCurrent,aPrevious);
end;

exports
  __fresnel_tick;

begin
  Application.HookFresnelLog:=True;
  Application.Initialize;
  Application.CreateFormNew(TMainForm,MainForm);
  Application.Run;
end.
```

The first line in the initialization of the library sets up a hook: the fresnel log will be written to standard output, and will show up in the browser console. The rest of the application startup code looks the same as a standard Lazarus LCL application. The `CreateForm` has been replaced with `CreateFormNew` so the `CreateNew` constructor of the form is called: Currently there are no resources in the webassembly (this is being worked on). Calling `CreateNew` makes sure no resources are loaded.

To demonstrate that the events mechanism works as expected also in the browser, our main form will also hook some events. For this reason, we define an enumeration to select the events that we want to listen to.

```
Type
  THookEvent = (heClick,heMouseMove,heMouseUp,heMouseDown,heMouseEnter,heMouseLeave,heFocus,h
  THookEvents = set of THookEvent;

  { TMainForm }

  TMainForm = Class(TFresnelForm)
  private
    procedure HookAllFresnelComponents;
    procedure LogEventData(Event: TAbstractEvent);
    procedure LogMouseEvent(Event: TFresnelMouseEvent; LogData: Boolean);
  Public
    procedure DoClick(Event: TAbstractEvent);
    procedure DoGeneralEvent(Event: TAbstractEvent);
    procedure DoMouseMove(Event: TFresnelMouseEvent);
    constructor CreateNew(aOwner : TComponent); override;
    procedure HookEvents(aEl: TFresnelELement; Publ: Boolean);
  end;

var
  MainForm : TMainForm;
```

```
const
  AllHookEvents = [Low(THookEvent)..High(THookEvent)];
```

The form is populated in the same way as our LCL version, in our constructor, we call `CreateControls`. This time we pass `Self` as the viewport, since the Fresnel form is the actual viewport:

```
constructor TMainForm.CreateNew(aOwner : TComponent);

const
  GlobalStyle = 'div {padding: 2px; border: 3px; margin: 6px;}';

begin
  Inherited CreateNew(aOwner);
  Width:=640;
  Height:=480;
  Stylesheet.Text:=GlobalStyle;
  CreateControls(Self);
  HookAllFresnelComponents;
end;
```

Note that the `Width` and `Height` of the form are set: The form is the only component which has a width and height property - this is logical, since it is the top-level control. We'll come to the last line shortly, they hook all fresnel events for all controls on the form. Note that we only set the globally applicable CSS styles in the `StyleSheet` property: these styles will be used for all controls, in addition to the CSS styles specified in the 'Style' property of each control.

In `CreateControls`, we demonstrate that the CSS styles can also be directly applied to the controls by setting the `Style` property:

```
Procedure TMainForm.CreateControls(ViewPort : TFresnelViewport);

  Function CreateControl(aClass : TFresnelElementClass;
                         aName : String;
                         aParent : TFresnelElement = nil) : TFresnelElement;
  begin
    if aParent=Nil then
      aparent:=Body1;
    Result:=aClass.Create(Self);
    Result.Name:=aName;
    Result.parent:=aParent;
  end;

begin
  Body1:=TBody(CreateControl(TBody,'Body1',ViewPort));
  Div1:=TDiv(CreateControl(TDiv,'Div1'));
  Span1:=TSpan(CreateControl(TSpan,'Span1'));
  label1:=TLabel(CreateControl(TLabel,'Label1'));
  Label1.Caption:='Label1Caption';
  Div2:=TDiv(CreateControl(TDiv,'Div2'));
  Img1:=TImage(CreateControl(TImage,'Img1'));
  Img1.Image.LoadFromFile('image.png');
  // Apply styles
```

```
  Body1.Style:='border: 2px; border-color: blue;';
  Div1.Style:='background-color: blue; border-color: black; height:50px;';
  Span1.Style:='width: 50px; height:70px; background-color: red; '+
               'border: 3px; border-color: black; margin: 3px;';
  Label1.Style:='background-color: green; ';
  Div2.Style:='border-color: black; position: absolute; border: 2px;'+
              ' left: 30px; top: 100px; width: 50px; height: 60px;';
  Img1.Style:='border-color: red; height:50px; position: absolute; '+
              'border: 2px; left: 150px; top: 200px; width: 48px; height: 48px;';
end;
```

Basically, this is the contents of the style sheet used in our previous example, but applied directly to the controls.

To demonstrate events, we set some event handlers on the events. The `HookAllFresnelComponents` simply loops over all controls and calls `HookEvents`

```
procedure TMainForm.HookAllFresnelComponents;

Var
  C : TComponent;
  I : Integer;
  UsePublished : Boolean;

begin
  UsePublished:=False;
  HookEvents(Self,UsePublished);
  For I:=0 to ComponentCount-1 do
    begin
    C:=Components[I];
    if C is TFresnelElement then
      HookEvents(C as TFresnelElement,UsePublished);
    end;
end;
```

The `UsePublished` parameter allows you to select which mechanism must be used: Specifying `True` will set the traditional property, specifying `False` will use the `AddEventListener` mechanism of the event dispatcher:

```
procedure TMainForm.HookEvents(aEl: TFresnelELement; Publ : Boolean);

begin
 if Publ then
   begin
   aEl.OnClick:=@DoClick;
   aEl.OnMouseMove:=@DoMouseMove;
   aEl.OnMouseEnter:=@DoMouseMove;
   aEl.OnMouseLeave:=@DoMouseMove;
   end
 else
   begin
   aEl.AddEventListener('click',@DoGeneralEvent);
   aEl.AddEventListener('mousemove',@DoGeneralEvent);
   aEl.AddEventListener('mouseenter',@DoGeneralEvent);
   aEl.AddEventListener('mouseleave',@DoGeneralEvent);
```

```
   aEl.AddEventListener('focus',@DoGeneralEvent);
   end;
end;
```

The general event handler `DoGeneralEvent`, which is registered using `AddEventListener`, logs the event and in case of a mouse event logs a little more:

```
procedure TMainForm.DoGeneralEvent(Event: TAbstractEvent);
begin
  LogEventData(Event);
  If Event is TFresnelMouseEvent then
    LogMouseEvent(Event as TFresnelMouseEvent,False);
end;
```

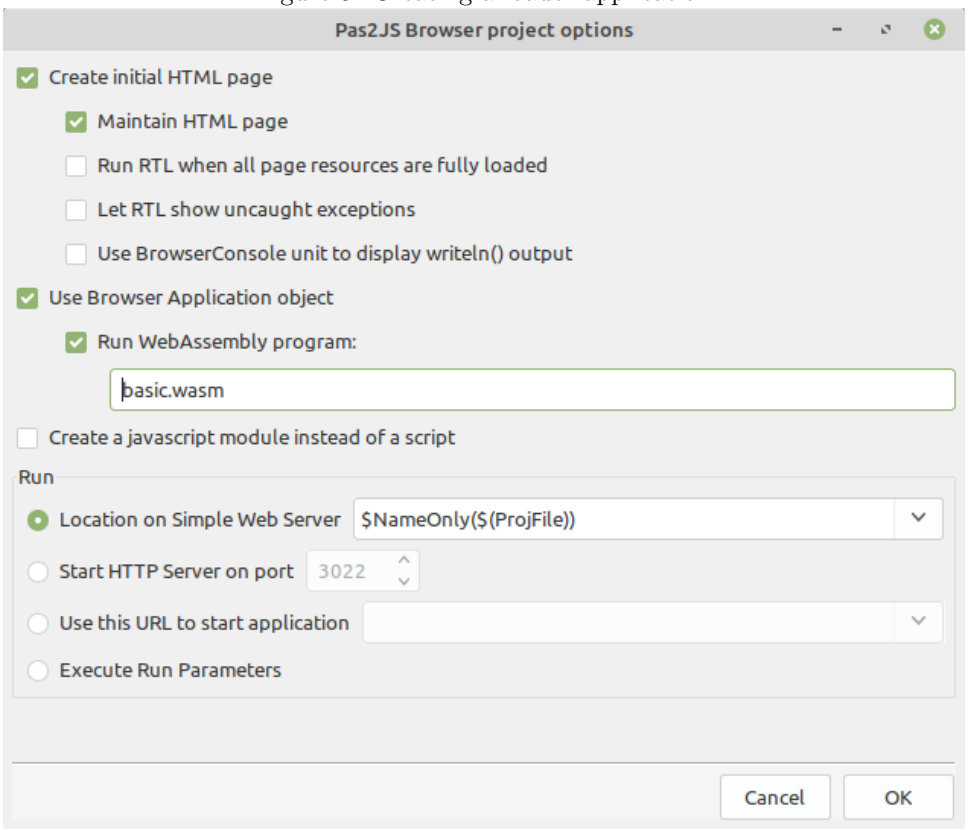
The logging events do little more than writing the event data to standard output:

```
procedure TMainForm.LogEventData(Event: TAbstractEvent);

const
  Fmt = 'Event class %s type: %s, sender : %s';

var
  S : String;

begin
  if Event.Sender=Nil then
    S:='(Nil)'
  else
    begin
    S:=Event.Sender.ClassName;
    if Event.Sender is TComponent then
      S:=TComponent(EVent.Sender).Name+' ('+S+')';
    end;
  Application.Log(etInfo,Fmt,[Event.ClassName, Event.EventName, S]);
end;

procedure TMainForm.LogMouseEvent(Event: TFresnelMouseEvent; LogData : Boolean);

const
  Fmt = 'Mouse Event (X: %f, Y: %f, Button: %s, Buttons: %s) ';

var
  Btn,Btns : String;
begin
  If LogData then
    LogEventData(Event);
  Btn:=GetEnumName(TypeInfo(TMouseButton),Ord(Event.Button)) ;
  Btns:=SetToString(PTypeInfo(TypeInfo(TMouseButtons)),Longint(EVent.Buttons),True);
  Application.Log(etInfo,Fmt, [Event.ControlX,Event.ControlY,Btn,Btns]);
end;
```

Note the use of RTTI to convert button enumerated and sets to actual button names.

Some event handlers have a typed version of the parameter, as can be seen in the `DoMouseMove` event handler, which receives a `TFresnelMouseEvent` parameter:

Figure 5: Creating a loader application




```
procedure TMainForm.DoClick(Event: TAbstractEvent);
begin
  Application.Log(etInfo,'You clicked '+(Event.Sender as TComponent).Name);
end;

procedure TMainForm.DoMouseMove(Event: TFresnelMouseEvent);

begin
  LogMouseEvent(Event,True)
end;
```

With this, our webassembly application is finished. The structure is identical to the LCL application. The differences (events, styles) were simply additions to the code in the regular.

9 The webassembly loader

To run our webassembly program in the browser, we need a Javascript program that loads the webassembly in the browser, provides it with the image file and finally that provides the Fresnel canvas.

To this end, we create a 'Web Browser application' in the IDE. In the dialog that appears, we check the 'Use Browser Application object' and 'Run Webassembly program' options and enter a filename, as in figure 5 on page 15. Setting these

options will create a skeleton project which we can adapt to our needs. We'll rename the application class to `TFresnelHostApplication`. The wizard will have added in the `DoRun` method a call to `StartWebAssembly` with the filename we entered. We'll need to change that.

The first thing to do is to provide the necessary Fresnel API methods to the webassembly. The Pas2js WebAssembly hosting environment has a mechanism to do this: to provide APIs to a webassembly module, a descendant of the `TImportExtension` class must be created and instantiated. Such a descendant has been made for the Fresnel API, a class called `TWasmFresnelAPI`. This class is implemented in the `fresnel.pas2js.wasmapi`, part of the `P2jsfresnelapi` package.

All that we need to do is to create an instance of the `TWasmFresnelAPI` class, passing the WASI environment to the constructor. We do this in the constructor of our application class:

```
constructor TFresnelHostApplication.Create(aOwner: TComponent);
begin
  inherited Create(aOwner);
  FFresnelApi:=TWasmFresnelApi.Create(WasiEnvironment);
  FFresnelAPI.LogAPICalls:=True;
  FFresnelAPI.CanvasParent:=TJSHTMLElement(document.getElementById('desktop'));
  RunEntryFunction:='_initialize';
end;
```

Since our Webassembly module is a library, the function to execute when running it, is not the usual `_start` as for a program, but `_initialize`, which simply executes the initializations sections of the units included in the library and the main library routine.

We set 2 properties on the FresnelAPI instance:

1. We choose to log the API calls (Every API call is logged to the screen)
2. We set the parent element for the canvas: for every Fresnel form, a HTML canvas is allocated. All these canvases are positioned below the `CanvasParent` element.

10 Filesystem support for WebAssembly

The Fresnel application loads an image from file using the usual Object pascal file handling mechanisms. How can we provide this file ? The WASI standard provides all the API calls to open files and read data from files, as well as directory listing mechanisms. It is up to the hosting environment to provide an implementation of these calls. The Pas2JS webassembly hosting environment has implemented these API calls, and uses a plugin mechanism to handle the actual reading from file.

The browser offers a standardized API to access the computer's filesystem in a sandboxed manner:

`https://developer.mozilla.org/en-US/docs/Web/API/FileSystem`

This basically reserves a (hidden) directory for use of your web application. Your application can only access files and directories inside this directory, and these directories are private to each webpage

This API would seem ideal to provide a filesysem to a webassembly. However, there is a catch: the filesystem API is an asynchronous API. The WASI API is synchronous, and this means that currently, the filesystem API is not usable.

So something else must be found. Before the FileSystem API was generally available, a pure Javascript implementation of a FileSystem emulation was created, called `BrowserFS`. It was modeled after the NodeJS filesystem API. This implementation is now known as `ZenFS`:

`https://github.com/zen-fs`

The ZenFS API comes with various backends that are synchronous:

InMemory : Stores files in-memory. This is cleared when the webpage is closed.

WebStorage : Stores files in local or session storage. This means the filesystem can be persisted, even when the webpage is closed.

Pas2jS comes with the necessary units to make use of this API, and here is a plugin for the WebAssembly hosting mechanism to provide a filesystem.

So, how to use the ZenFS filesystem to provide an image file to the webassembly module ? Before starting the webassembly, we load the necessary files from the webserver, and store them in our in-browser filesystem emulation. Since loading the files from the server is asynchronous, this loading needs to be completed before we can start the webassembly.

The ZenFS filesystem needs to be initialized. This initialization is also asynchronous, so we must wait for it to complete before we can start our webassembly program. To make our life a little easier, we will introduce an asynchronous method:

```
procedure RunWasm ; async;
```

This means we can use `await` in the `RunWasm` method to let the filesystem initialization finish before calling `StartWebAssembly`. The code from the `DoRun` method generated by the application wizard in Lazarus is replaced with the following:

```
procedure TFresnelHostApplication.DoRun;

begin
  RunWasm;
end;
```

The actual work now happens in `RunWasm`, which starts by initializing the ZenFS file system. The initialization means that you tell ZenFS where to mount various file systems, similar to the way this happens on a typical unix or linux operating system.

You can use various filesystems at the same time, but for our needs, we'll mount a single filesystem using `WebStorage`:

```
procedure TFresnelHostApplication.RunWasm;

var
  aCount : Integer;

begin
```

```
  Terminate;
  await(tjsobject,  ZenFS.configure(
    new(
      ['mounts', new([
        '/', DomBackends.WebStorage
       ])
      ])
    )
  );
  FS:=TWASIZenFS.Create;
  WasiEnvironment.FS:=FS;
  aCount:=await(LoadFiles);
  Writeln('Loaded ',aCount,' files.');
  StartWebAssembly('basic.wasm',true,@OnBeforeStart,@OnAfterStart);
end;
```

After the ZenFS filesystem is initialized, we create an instance of the `TWASIZenFS` class, and assign it to the WasiEnvironment. Before starting the webassembly, we load the needed files into our virtual filesystem using `LoadFiles`. As mentioned before, this is an asynchronous call, so we wait for it to complete. Lastly, the webassembly is started, specifying 2 callbacks: one to be executed before, one to be executed after the start of the webassembly.

Before diving into these calls, let's see how we can load files into our browser-based filesystem.

The `TWasiHostApplication` class offers 3 calls to preload files from the server into the filesystem emulation:

```
function PreLoadFiles(aFiles : TPreLoadFileDynArray) : TPreLoadFilesResult; async;
function PreLoadFiles(aFiles : Array of string) : TPreLoadFilesResult; async;
function PreLoadFilesIntoDirectory(aDirectory: String;
             aFiles: array of string): TPreLoadFilesResult; async;
```

As you can see, all calls are asynchronous. The first call is the raw download mechanism. You specify the files to preload using an array of records:

```
TPreLoadFile = record
  url : String;
  localname : string;
end;
```

The URL contents will be downloaded and put into the local filesystem as a file with the given path and name. (note that if y ou specify directories, you must create any directories before loading files into them) The second form of the `PreLoadFiles` call accepts an array of strings. This should be an even amount of strings, where each pair is a URl and a local filename: these are simply transformed into an array of `TPreLoadFile` records.

The `PreLoadFilesIntoDirectory` is a utility function that stores all downloaded files in a single directory.

The `LoadFiles` function uses this latter utility function, and is really simple:

```
function TFresnelHostApplication.LoadFiles: Integer;

const
```

```
  Files : TStringDynArray =
        ('image.png','style1.css','style2.css');
var
  Res: TPreLoadFilesResult;
  I : Integer;
begin
  result:=-1;
  Res:=await (PreloadFilesIntoDirectory('/',Files));
  For I:=0 to Length(Res.failedurls)-1 do
    With Res.failedurls[i] do
      Writeln('Failed to preload file: ',url,' : ',error);
  Result:=res.loadcount;
end;
```

The `TPreLoadFilesResult` gives info about the number of loaded files and any errors that may have occurred.

All that remains to be discussed are the 2 callbacks that were passed to the `StartWebassembly` call. The `OnBeforeStart` event is called before the webassembly is started, and we use it to pass the functions that are exported from the webassembly to the Fresnel API:

```
function TFresnelHostApplication.OnBeforeStart(Sender: TObject;
                 aDescriptor: TWebAssemblyStartDescriptor): Boolean;
begin
  FFresnelApi.InstanceExports:=aDescriptor.Exported;
  Result:=true;
end;
```

The fresnel API needs access to the exported functions in order to call the timer function `__fresnel_tick`.

This is exactly why the `OnAfterStart` event handler is needed: once the webassembly module has been initialized, we start the fresnel timer:

```
procedure TFresnelHostApplication.OnAfterStart(Sender: TObject;
                      aDescriptor: TWebAssemblyStartDescriptor);
begin
  Writeln('Starting timer');
  FFresnelApi.StartTimerTick;
end;
```

Last but not least, we need the main program code to set the ball rolling. This looks like any Free Pascal or Lazarus code, with a small addition to set up the console output: the `WriteLn` statements from the webassembly are caught and displayed in the browser console log, but also in a special HTML element (with id "`pasjsconsole`"):

```
var
  Application : TFresnelHostApplication;
begin
  ConsoleStyle:=DefaultCRTConsoleStyle;
  HookConsole;
  Application:=TFresnelHostApplication.Create(nil);
  Application.Initialize;
```

```
  Application.Run;
end.
```

With this, the loader program is finished. All we need now is a HTML page which will execute the code. We need 2 special tags in the HTML: one is the parent for the canvas (with id "desktop") and one is needed to display the output of the `WriteLn` statement. We also need to load ZenFS : 2 Javascript files are needed:

browser.min.js This is the core ZenFS module.

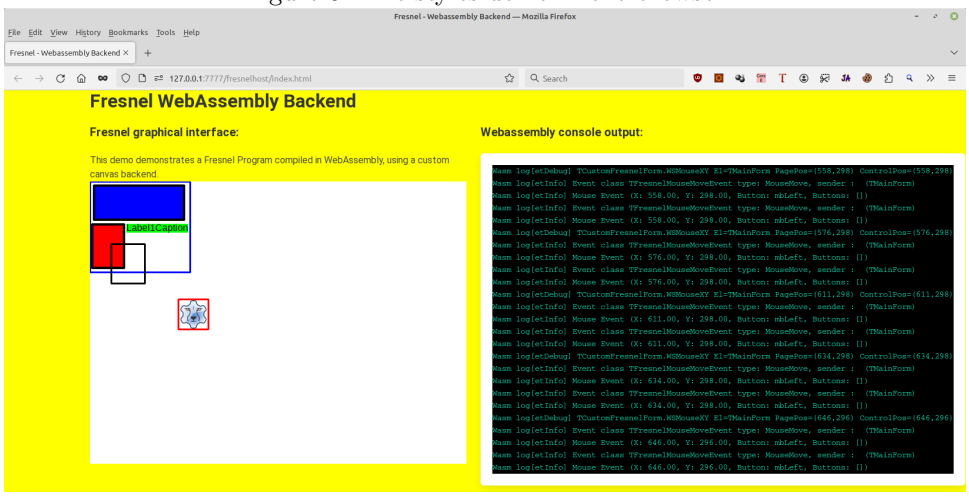
browser.dom.fs This is the ZenFS module that allows to store files in the browser local storage.

Add some CSS styling with Bulma CSS to the mix, and this is our web page:

```
<!doctype html>
<html lang="en">
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Fresnel - Webassembly Backend</title>
  <link href="bulma.min.css" rel="stylesheet">
  <script src="browser.min.js"></script>
  <script src="browser.dom.js"></script>
  <script src="fresnelhost.js"></script>
</head>
<body style="background-color: yellow">
  <div class="container">
    <h1 class="title is-3">Fresnel WebAssembly Backend</h1>
    <div class="columns">
      <div class="column">
        <h1 class="title is-5">Fresnel graphical interface:</h1>
        <p>This demo demonstrates a Fresnel Program compiled in WebAssembly,
           using a custom canvas backend.</p>
        <div id="desktop" style="min-height: 480px;">
        </div>
      </div>
      <div class="column">
        <h1 class="title is-5">Webassembly console output:</h1>
        <div class="box" id="pasjsconsole"></div>
      </div>
    </div>
  </div>
  <script>
    rtl.showUncaughtExceptions=true;
    window.addEventListener("load", rtl.run);
  </script>
</body>
</html>
```

When all this is loaded in the browser, the application will look like figure 6 on page 21. Note the yellow background on the HTML body. This is done to demonstrate clearly that the fresnel background (white) is observed when showing an image with transparency, such as the lazarus icon.

Figure 6: The styles demo in the browser



11 Conclusion

In this article, we've shown that the goals that were outlined for project Fresnel are attainable: we have 2 working backends, a CSS-driven layout, multiple platforms, a powerful event mechanism. With the Skia renderer available, there should be no problem to create a universal graphical application which runs on all native platforms and in the browser. All this using a single codebase, and running at native speed. And obviously, all this using your favourite programming language: Object Pascal.