Compiling an FMX application for the browser

Michaël Van Canneyt

October 7, 2024

Abstract

Firemonkey (FMX) is the cross-platform GUI framework of Delphi. It allows you to compile your application for all major native platforms: Windows, MacOS, Android, iOS. And now, thanks to Free Pascal, also for the browser.

1 Introduction

Since a long time, Delphi uses Firemonkey (FMX) to offer a GUI for its supported platforms other than Windows. However, a major platform is missing: the browser.

The creator of Firemonkey, Eugene Kryukov, was aware that the Free Pascal team - sponsored by Tixeo.com - was undertaking an effort to port Firemonkey to the browser. His untimely death - barely a week before the first successful execution of the 'all controls' FMX demo in the browser - means he was unable to see his brainchild executing in the browser.

After more than a year of work, you can now compile an FMX application to webassembly using Free Pascal, and execute it in the browser. For this, many things had to be done in Free Pascal:

- Extended RTTI had to be integrated in the main branch.
- The compiler had to be extended to enable all Object Pascal features for webassembly (notably, threads and Invoke())
- The RTL distributed with FPC had to be made more compatible to the Delphi RTL: namespaces were introduced, and many missing units were added.
- Some patches were needed to adapt FMX so it is compilable by FPC.
- To ease working in the browser, DWARF support for the browser was added (currently only Chromium allows to debug your pascal code in the browser).
- Enable filesystem support for webassembly running in the browser.
- Give the webassembly access to the browser DOM and, in particular to the HTML canvas.

After all that work, now you can compile a basic FMX application and execute it in the browser. See figure 1 on page 2. Many things are of course not yet possible. Some due to the nature of the browser, some due to missing features in FPC.

What is not (yet) possible ?

Figure 1: Hello, World! FMX style, in the browser

- Database access. `TDataset` is available, but no asynchronous data layer exists (yet). You can load data from file, if you have a `TDataset` capable of doing so.
- Live bindings - Free Pascal does not yet have the basic objects used in Live bindings.
- System Dialogs such as "file open", "file save", "print" etc. This is because the browser does not expose the needed underlying APIs, and the APIs that are available are asynchronous, making them incompatible with the basic APIs defined by the FMX framework
- 3D rendering. This is on the roadmap, the browser offers WebGL and therefore theoretically it is possible to add this.

If these limitations are acceptable (some may be lifted by the time you read this article), and you have a Delphi license which allows you access to the FMX sources, then you can compile a FMX program for webassembly.

How to compile your own FMX application to webassembly?

To be able to compile a Delphi application to FMX, several steps are needed.

2 Create a webassembly cross-compiler.

The first thing to do is to create development environment where you can compile to webassembly: You need a webassembly cross-compiler. If you already have made one, you can skip this step - or repeat it to make sure you have the latest features.

Instructions to create a cross-compiler can be found at:

`https://wiki.freepascal.org/WebAssembly/Compiler`

They have been published in a previous article as well ("Getting started with FPC and WebAssembly" januari 2022).

For starters, you need to have the released version of FPC installed on your system.

Creating a cross-compiler for webassembly starts by downloading the development version of the compiler using git. So you need a git client, any standard git client should be OK.

In a directory of your choice, execute the following command:

```
git clone https://gitlab.com/freepascal.org/fpc/source.git fpc
```

Then, you need to make sure that the binaries installed by FPC are in your `PATH`. Once that is OK, go to the toplevel directory of the FPC sources you cloned with git, and execute the following command. The backslash means the command is continued on the following line, you can omit it if you type the 2 displayed lines on one line:

```
make clean crossall OPT='-dSKIP_INTERNAL20231102' \
     PP=ppcx64 OS_TARGET=wasi CPU_TARGET=wasm32
```

This will recompile the compiler and all packages. The `-dSKIP_INTERNAL20231102` is an option to disable a specific compiler error, which can occur when compiling FMX. It is expected to disappear later.

If the compilation went successful, you can install this new version with the following command:

```
make crossinstall PP=ppcx64 OS_TARGET=wasi CPU_TARGET=wasm32
```

With this, you will have a working cross-compiler.

3 Create a Delphi-compatible RTL.

However, having a cross-compiler is not sufficient. With the cross-compiler, you can compile FPC sources, and the result will work in the browser. But to compile FMX and a FMX program, we need to have a Delphi-compatible RTL as well.

The sources of the Free Pascal RTL and packages can be compiled in one of 2 modes:

- The FPC backwards-compatible RTL: a string is a 1-byte string. No namespaces are used for the units.
- The Delphi-compatible RTL: a string is a 2-byte string. Namespaces are used.

To create the Delphi-compatible RTL, the instructions van be found on:

```
https://wiki.freepascal.org/FPC_Unicode_RTL
```

The first thing to do is to create 3 files, needed for sub-target support. Sub-target support allows you to compile and install an RTL with different settings, and use them in your builds without too much effort. It was introduced to accomodate easy building and installing of the RTL and packages using different compiling options.

You can think of it as a build configuration as offered in Delphi or Lazarus, but at the level of the compiler.

We need 3 subtargets, and for that we need to create 3 named configurations. For unix-like operating system (including the mac) these 3 files are located in your home directory and are called:

```
.fpc-unicodertl.cfg
.fpc-browser.cfg
.fpc-unicodertl-browser.cfg
```

For windows-like operating systems, the files are located next to the compiler binary (`ppcrosswasm32`) and are called:

```
fpc-unicodertl.cfg
fpc-browser.cfg
fpc-unicodertl-browser.cfg
```

The first of these files (.fpc-unicodertl.cfg) contains the following 2 lines:

```
-dUNICODERTL
-Municodestrings
```

This is needed for the creation of a delphi-compatible RTL: 2-byte strings, and namespaced unit files.

The second of these files (.fpc-browser.cfg) contains the following line:

```
-CTwasmexceptions
```

This enables the use of native Webassembly exception support. In time, this switch will disappear, since all webassembly engines today support webassembly exceptions.

The third file (.fpc-unicodertl-browser.cfg) combines the previous 2 files:

```
#INCLUDE /home/michael/.fpc-unicodertl.cfg
#INCLUDE /home/michael/.fpc-browser.cfg
```

Of course, you must change the path `/home/michael/` to the correct path for your setup.

Once these files are made, the delphi-compatible RTL and packages can be constructed. From the top-level directory of the FPC source tree, type the following commands:

```
cd rtl
make clean all SUB_TARGET=unicodertl-browser FPC_DOTTEDUNITS=1 \
     OPT="-g" PP=ppcrosswasm32
make install SUB_TARGET=unicodertl-browser FPC_DOTTEDUNITS=1 \
     PP=ppcrosswasm32
cd ..
```

again, the backslash at the end of the line means the command is continued on the next line.

If all went without error, then the basic RTL units have been compiled with namespaces, and were installed in a dedicated directory (separate from the default directory).

The rest of the units (and there are many) can be compiled and installed similarly

```
cd packages
make clean all SUB_TARGET=unicodertl-browser FPC_DOTTEDUNITS=1 \
     OPT="-g" PP=ppcrosswasm32
make install SUB_TARGET=unicodertl-browser FPC_DOTTEDUNITS=1 \
     PP=ppcrosswasm32
cd ..
```

Note that we specified `OPT=-g`, this means the `-g` option is added to the compiler command-line, and instructs the compiler to generate debug info.

With this, you should have a compiler ready to generate webassembly binaries from FPC or Delphi sources. If you wish to compile Delphi-compatible sources for your default operating system, you must select the `unicodertl` subtarget, this means adding the appropriate command-line option to the compiler command-line, e.g.:

```
ppcx64 -tunicodertl hello.dpr
```

if you wish to compile FPC sources for webassembly, select the `browser` subtarget:

```
ppcrosswasm32 -tbrowser hello.pp
```

if you wish to compile Delphi sources for webassembly, select the `unicodertl-browser` subtarget:

```
ppcrosswasm32 -tunicodertl-browser hello.dpr
```

4 Preparing the FMX sources

In order to compile a FMX program, you need of course the FMX sources. FMX is a copyrighted product, with the copyright owned by Embarcadero. This means that of course the Free Pascal team cannot distribute FMX. You can only proceed if you have a valid Delphi license that allows you the use of FMX.

Unfortunately, the FMX sources do not compile with Free Pascal out-of-the box. The FMX sources uses some pascal constructs that are not supported by Free Pascal:

- a non-generic class and generic class cannot have the same name in Free Pascal. Delphi allows this, and it is used in their messaging system.
- Inline variables.

The use of these features is not widespread, and the number of changes is not too great, so it is possible to create a patch (a diff file). This patch has been created. The patch is small enough so it is completely impossible to reconstruct the complete FMX sources from it.

We contacted Embarcadero to ask wether it would be considered a copyright infringement if we distributed a patch to their sources. Thankfully, the answer was negative, so we can make the patch available.

To use this patch, the best approach is to make a copy of the FMX sources, and apply the patch at:

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

This repository contains all the files necessary to compile an FMX application for WebAssembly.

The patch can be applied to the FMX version shipped with version 12.1 of Delphi. It may succeed on other versions. For example it works mostly on 11.3, only a small part needs to be checked manually.

To apply the patch, download the patch file and put it in the directory where you copied the FMX sources. Then, on the command-line, go to the copy of the FMX sources and execute the following command:

```
patch < fmx-fpc-1210.diff
```

Free Pascal on Windows distributes a version of the patch tool, this can be used to execute the above command.

If everything went correctly, then you should have a version of FMX that compiles with FPC. However, due to the structure of FMX, we cannot quite test this: if you try compiling it at this point, compilation will fail with a message of a missing unit.

5 The Webassembly Canvas

Firemonkey provides an abstraction of the Operating System's GUI APIs. Basically, it defines some abstract interfaces which must be implemented for each platform. The most important ones are the Canvas API and the Window API. The Window API is responsible for creating a window, and handling the messages that the system sends to this window: mouse moves, mouse clicks, keystrokes and so on.

Such a Window API and Canvas API have been created for use in webassembly.

Unfortunately, this API must be added in the FMX sources themselves. So, the patch above will include a reference to some files which we have not yet discussed, and will fail to compile until the next step has been completed: adding the Canvas and Windows APIs.

The window and Canvas API for webassembly have been built upon the Window and Canvas API that have been developed for Project Fresnel, the project to provide FPC and Lazarus developers with a UI framework that is based on using CSS.

An application created with Fresnel runs in the browser using this API (see article: "Project Fresnel Update", may 2024), and since it would be silly to recreate what already exists, the Fresnel canvas is reused for the FMX canvas.

This API contains 2 parts:

- The Fresnel webassembly backend API: both the webassembly API definitions and the Pas2JS browser-side implementation of this definition.
- The FMX canvas and window API that builds on top of the Fresnel backend.

In the repository, the webassembly side of both parts are in the `Src/Wasm` directory, in subdirectories `Fresnel` and `FMX`, respectively. The browser hosting part is in Src/Pas2JS, it is needed to actually implement the API for the browser. The Fresnel part is a copy of what can be found in the Fresnel repository, it is included for convenience.

So, to proceed, you need to check out this repository:

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

And add the needed directories to your FPC configuration. If you're using Lazarus, then a package is provided that does everything for you: `fpcforfmx.lpk`, it is located in the `Src/Wasm` directory. You can add this as a dependency to a lazarus project in order to compile your FMX project.

The `Src/Wasm/Package` directory contains a Lazarus package (`fmx.lpk`) in which all FMX files that are known to compile are included, so you can drop this package file in the directory with the FMX sources, or put the patched version of FMX in the directory where the `fmx.lpk` file is.. This package can also be added to your Lazarus FMX project, so the cmpiler will find all files it needs to compile your project.

6 Hosting a FMX program in the browser

A webassembly program (commonly called a 'module') is bytecode which is loaded into a hosting program, which then executes the bytecode instructions. By default, the webassembly bytecode has no access to the environment in which the hosting program is running. The hosting program can (and must) provide APIs to the webassembly module to interact with the environment. The webassembly module has a list of API calls that it expects to receive, and if one of these API calls is missing, it will not be able to start : similar to having dependencies on dynamically loadable libraries in a native environment.

A typical program requires file system access, access to the clock - and in the case of an FMX application, a canvas and some UI events. The file system access is defined in the WASI standard, and this is provided by Free Pascal's pas2js compiler. Pas2js also provides direct access to HTTP and websocket transport as well as the regular

expressions engine of the browser. Moreover, through JOB - Javascript Object Bindings - you have access to every API in the browser (See the article "Using the browser APIs from WebAssembly", may 2024).

The pas2js units that make a canvas available to a webassembly have been developed for project Fresnel, and they are reused to allow FMX to access the Browser canvas. There are 2 units:

fresnel.wasm.shared Some constants and basic types used in the API.

fresnel.pas2js.wasmapi The actual implementation of the API. Here you will find calls for a timer, allocating a canvas and the usual drawing operations on the canvas. In essence, it allows the webassembly module to do everything what can bedone with a HTML canvas - which is quite a lot.

These 2 units are available in the abovementioned repository in the `Src/Pas2JS` folder.

Pas2JS offers a complete environment for loading webassembly modules and making available APIs to the loaded modules: the `TWasiHost` and `TWasiEnvironment` classes. The API has been discussed in other articles.

Using this API, it is possible to make a generic hosting program for a webassembly module that runs a FMX program. Such a hosting program has been added to the repository above. It is a standard pas2js program, using the application class developed for hosting webassembly programs. We'll explain the structure of this program, so you can adapt it to your needs.

As usual, we have to define an application class, and since we want to have support for loading webassembly modules, it be a `TBrowserWASIHostApplication` descendant:

```
TFMXHostApplication = class(TBrowserWASIHostApplication)
Private
  FZenFS: TWASIZenFS;
  FFresnelApi : TWasmFresnelApi;
  FRegexp : TWasmRegExpAPI;
Public
  constructor Create(aOwner : TComponent); override;
  procedure RunWasm; async;
  procedure DoRun; override;
end;
```

In the definition, you can see 3 fields: each field will hold a class that provides a specific functionality to the program:

TWASIZenFS creates a virtual filesystem in the browser, using ZenFS. This filesystem will be accessible to the webassembly program.

TWasmFresnelApi The Fresnel API - which is what FMX needs to generate a UI.

TWasmRegExpAPI Access to the regular expression engine of the browser. FMX uses regular expressions for example in the edit component, to be able to validate input.

These classes are instantiated and configured - where else - in the constructor of the application class, which starts by creating the filesystem support and registerering it with the WASI environment in the first lines of the constructor's code:

```
constructor TFMXHostApplication.Create(aOwner: TComponent);
begin
  inherited Create(aOwner);
  // Create and register the filesustem
  FZenFS:=TWASIZenFS.Create;
  WasiEnvironment.FS:=FZenFS;
  // The fresnel API.
  FFresnelApi:=TWasmFresnelApi.Create(WasiEnvironment);
  FFresnelApi.CanvasParent:=GetHTMLElement('desktop');
  FFresnelApi.CreateDefaultCanvas:=True;
  FFresnelApi.MenuSupport:=True;
  // The regular expression engine.
  FRegexp:=TWasmRegExpAPI.Create(WasiEnvironment);
  // An FMX program is a library.
  RunEntryFunction:='_initialize';
  if Assigned(hostConfig) then
    begin
    WasiEnvironment.LogAPI:=HostConfig.logWasiAPi;
    FFresnelApi.LogAPICalls:=HostConfig.logFresnelAPI;
    WasiEnvironment.Environment.Add('FMX_LOGLEVEL='+HostConfig.FMXLogLevel);
    end;
end;
```

Similarly the Fresnel API is created and configured: the fresnel API needs a HTML tag under which it will create and position the windows that FMX creates. We enable menu support: the menu of an FMX application is rendered in HTML. This way, menu environments like on the Mac or mobile operating systems can be supported.

The last API that is needed is the regular expression engine. When regular expression support is created, the `RunEntryFunction` is set. When the webassembly module is started, the `RunEntryFunction` function is the function that is called first. An FMX program is actually a library, and the library is initialized with the `_initialize` function.

Lastly, the logging level is set for the various APIs, this is useful for debugging. The settings are read from a JSON object that is defined in a file `hostconfig.js` which is included in the HTML file. This approach allows to configure the host application and define which webassembly needs to be loaded without having to recompile the host application.

As usual, the `DoRun` method of the application must be overridden to implement the functionality of the application.

```
procedure TFMXHostApplication.DoRun;

begin
  RunWasm;
end;
```

The `RunWasm` function is declared `async`, telling the browser that it need not wait for the return. This allows us to configure the zenfs file system: the zenfs file system is initialized with as backend the web storage, meaning that the files are stored in the local storage of the browser:

```
procedure TFMXHostApplication.RunWasm;
```

```
var
  wasm : String;

begin
  Terminate;
  aWait(TJSObject,ZenFS.configure(
     New([
     'mounts', New([
        '/',DomBackends.WebStorage
      ])
  ])));
  if Assigned(HostConfig) and isString(HostConfig.wasmFilename) then
    Wasm:=HostConfig.wasmFilename
  else
    begin
    // Allow to load file specified in hash: index.html#mywasmfile.wasm
    Wasm:=ParamStr(1);
    if Wasm='' then
      Wasm:='HelloWorldFMX.wasm';
    end;
  StartWebAssembly(Wasm, true);
end;
```

After that the configuration object and the URL are examined to see what file must be loaded. You can specify the filename in the configuration object, or append it as the hash in the URL (`#wasmfilename`)

Lastly, the application is created and started:

```
var
  Application : TFMXHostApplication;
begin
  Application:=TFMXHostApplication.Create(nil);
  Application.Initialize;
  Application.Run;
end.
```

That's it. With the above program, you can load any FMX program and run it without needing to recompile the host program. By adding some API classes (http, websocket and others will follow) you can provide more functionality to the webassembly program.

With all this work, we are now ready to compile the first FMX application for Webassembly.

7 The first try: 'Hello, World'

To check whether all this work payed off, we will create a small 'hello world!' program. A simple form with a label and a button: when the button is clicked, the background color of the form is changed. Keeping in mind that one should try walking before running, we will not use a form file (.fmx) but we'll create the form and the elements on it in code.

Before starting, an important detail needs to be mentioned: Any webassembly program which needs to run a kind of message loop needs to be created as a library. This is because webassembly is architecturally more a library (the used term is module) than a program. It is a library which exports functions. The host environment (in our case, the browser) calls a function and suspends all other activity till the webassembly function returns.

That would mean that in the case of a program, the whole program is executed before the browser resumes normal processing: rendering the HTML, reacting to user events etc. To the user, this would seem as if the browser freezes till the program exits.

One could argue that the program can install some UI hooks and then exit the program, but this will not work: when the program exits, the Pascal runtime will finalize all units including the system unit: when one of the installed hooks is called, the program is in an unusable state.

Instead, we use a library project: As mentioned before, the library exposes an '_initialize' routine, which executes simply the initialization code of all units and the libary `begin..end` block. We must therefore install the necessary hooks and create the first form in the `begin..end` block, and after that the `_initialize` function exits, and the hooks (mouse clicks etc.) will do their work: since the library was not finalized, everything will continue to function.

With this knowledge, we can create our first FMX application:

```
library HelloWorld;

uses
  System.FPWideString
  , System.Unicode.Unicodeducet
  , System.CodePages.unicodedata
  , System.MonitorSupport
  , FpImage.Reader.PNG
  , FpImage.Reader.JPEG
  , FpImage.Reader.Bitmap
  , FMX.Wasm.WindowHandle
  , FMX.Controls.Wasm
  , FMX.Canvas.Wasm
  , FMX.Platform.Wasm
  , FMX.Forms
  , form.main in 'form.main.pas' {Form1};

{$R *.res}

begin
    Application.Title:='FMX Application running in the browser';
    Application.Initialize;
    Form1:=TForm1.CreateNew(Application,0);
    Form1.Name:='Form1';
    Application.MainForm:=Form1;
    Form1.Show;
    Application.Run;
end.
```

The `begin..end` block looks almost like a normal UI application project. The only difference is that - because we're not using a form file, we need to create the

form manually and use the `CreateNew` constructor: this constructor creates the form, but skips the streaming from the form's associated fmx file. When using the `Application.CreateForm` function, the streaming cannot be skipped.

The uses clause deserves some special attention. Free Pascal works differently from Delphi in many aspects; one aspect is the treatment of unicode: The Free Pascal RTL delegates this functionality to plugins (so-called managers), similarly for monitor support: support for the system unit's `TMonitor` record is implemented elsewhere.

The units that implement these plugins must be initialized before the rest of the program's units are initialized, so they must be first in the uses clause of the main program, and not somewhere hidden in some other units. That is why the first 4 units are present in the uses clause:

- The `System.FPWideString` `System.Unicode.Unicodeducet` and `System.CodePages.unicodedata` implement Unicode support using 100% native object pascal code.
- `System.MonitorSupport` implements the `TMonitor` support.

FMX relies on the back-end services to provide support for images. For WebAssembly, the `FPImage` framework is used to provide support for images. Again, this is a framework implemented in 100% object pascal code. The `FpImage.Reader.PNG` `FpImage.Reader.JPEG` and `FpImage.Reader.Bitmap` units register support for reading PNG, JPEG and Bitmap files, respectively.

Lastly, the `FMX.Wasm.WindowHandle`, `FMX.Controls.Wasm`, `FMX.Canvas.Wasm` and `FMX.Platform.Wasm` units activate the WebAssembly backend for FMX: as the unit names suggest, they provide Window handle, controls styling and `Canvas` support. The last unit registers a lot of other platform services, although many of them are still empty at this moment.

The `FMX.Forms` and `forms.main` units are present in any FMX program : the former contains the `TApplication` and `TForm` implementation, the latter contains the implementation of the program's main form.

The main form looks like any other FMX form, with the exception that all controls are created in code instead of relying on the streaming system reading a `.fmx` form:

```
TForm1 = class(TForm)
  lblHello: TLabel;
  btnClickMe : TButton;
  procedure DoClick(Sender: TObject);
  procedure DoMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Sing
  procedure DoMouseUp(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Single
  procedure HandlePaint(Sender: TObject; Canvas: TCanvas; const ARect: TRectF);
public
  constructor CreateNew(aOwner : TComponent; Dummy: NativeInt = 0); override;
end;
```

This looks not so different from a normal form, Since we're not using the streaming system, the `OnCreate` event cannot be used to create the controls, because it is not called. Instead the `CreateNew` constructor must be used. The constructor starts by setting the form properties, the position and size, asw ell as the caption:

```
constructor TForm1.CreateNew(aOwner: TComponent; Dummy: NativeInt = 0);
begin
  inherited;
```

```
  Left := 0;
  Top := 0;
  Caption := 'FMX Hello world';
  Name:='form1';
  Fill.Color:=TAlphaColors.AquaMarine;
  Fill.Kind:=TBrushKind.Solid;
  ClientHeight := 640;
  ClientWidth := 480;
  FormFactor.Width := 640;
  FormFactor.Height := 480;
  FormFactor.Devices := [TDeviceKind.Desktop];
  OnPaint:=HandlePaint;
```

This is what can be found in any `.fmx` form, but it is translated to code. The last line implements the `OnPaint` handler of the form, which will be used to paint the background.

After configurig the form, we create a label control; Again, we start by set position, size and caption, making the label visible;

```
  lblHello:=TLabel.Create(Self);
  With lblHello do
    begin
    Font.Size:=12;
    Font.Style:=[];
    Name:='lblHello';
    Parent:=Self;
    Position.X := 80.0;
    Position.Y := 56.0;
    Size.Width := 241.0;
    Size.Height := 41.0;
    Size.PlatformDefault := False;
    Text := '"Hello, world!" from FMX';
    TabOrder := 0;
    HitTest:=True;
    Visible:=True;
    OnClick:=DoClick;
    OnMouseUp:=DoMouseUp;
    OnMouseDown:=DoMouseDown;
    end;
```

The last lines set some event handlers, simply to demonstrate that our application reacts on user events.

The last control to be created is a button.

```
  btnClickMe:=TButton.Create(Self);
  With btnClickMe do
    begin
    btnClickMe.Font.Size:=12;
    btnClickMe.Font.Style:=[];
    btnClickMe.Name:='btnClickMe';
    btnClickMe.Parent:=Self;
    btnClickMe.Visible:=True;
    btnClickMe.Position.X := 80.0;
```

```
    btnClickMe.Position.Y := 150.0;
    btnClickMe.Size.Width := 120.0;
    btnClickMe.Size.Height := 32.0;
    btnClickMe.Size.PlatformDefault := False;
    btnClickMe.Text := 'Click me...';
    btnClickMe.OnClick:=DoClick;
    end;
end;
```

The same pattern as for the previous controls is followed: setting position, size and caption, and then the `OnClick` event handler.

The paint handler sets the fill brush and draws a rectangle:

```
procedure TForm1.HandlePaint(Sender: TObject; Canvas: TCanvas; const ARect: TRectF);

begin
  Canvas.Stroke.Color:=TAlphaColors.Yellow;
  Canvas.Stroke.Kind:=TBrushKind.Solid;
  Canvas.Fill.Kind:=TBrushKind.None;
  Canvas.DrawRect(TRectF.create(79,55,80+242,56+42),1);
end;
```

The background color of the form is set in the `OnClick` handlers of the label and the button:

```
Procedure TForm1.DoClick(Sender : TObject);

begin
  Writeln('Got click :)');
  if Fill.Color=TAlphaColors.AquaMarine then
    Fill.Color:=TAlphaColors.Lavender
  else
    Fill.Color:=TAlphaColors.AquaMarine
end;
```

Finally, we display some messages when the user clicks the mouse on the label:

```
Procedure TForm1.DoMouseUp(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: S
begin
  Writeln('Got mouse button "',Button,'" up at (',X:6:2,',',Y:6:2,')');
end;

Procedure TForm1.DoMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y:
begin
  Writeln('Got mouse button "',Button,'" down at (',X:6:2,',',Y:6:2,')');
end;
```

With all this code, the application is ready to compile and run. The result can be seen in figure 1 on page 2, and after a click on the button, the result can be seen in figure 2 on page 15. The button does not look like a normal Windows button. This is because the style file is the Linux `ubuntu` style: all development happens on Linux, and so the ubuntu style was chosen.

Figure 2: The onclick handler at work

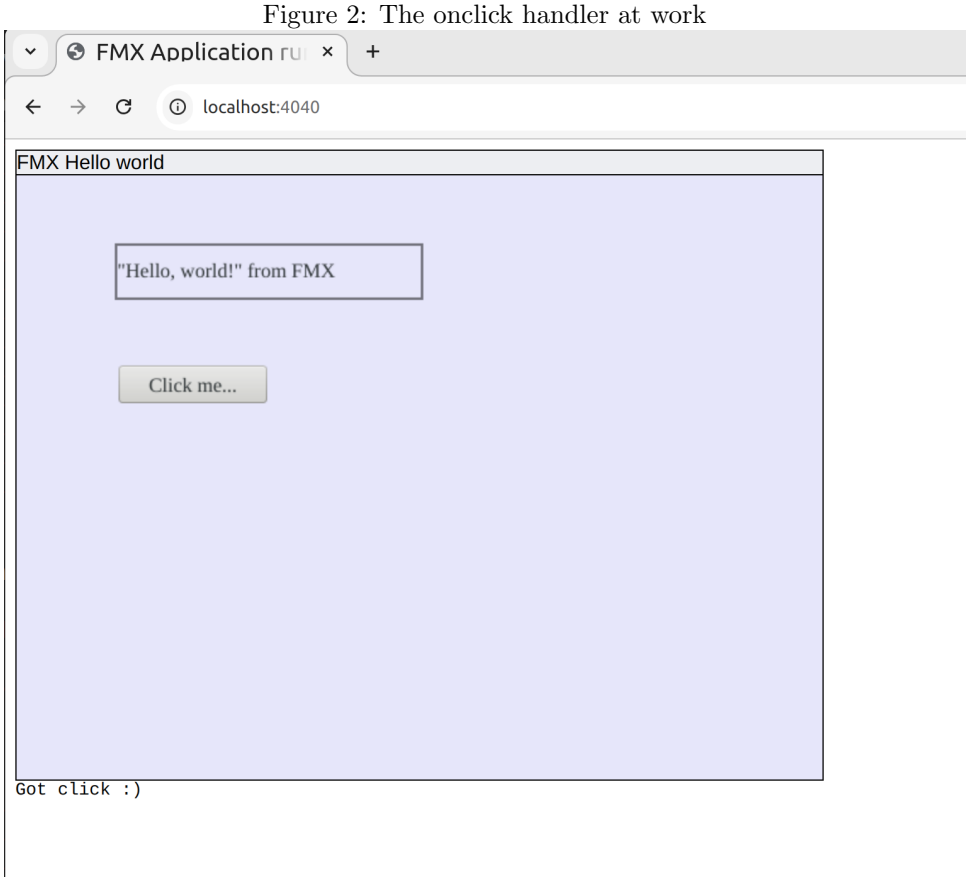
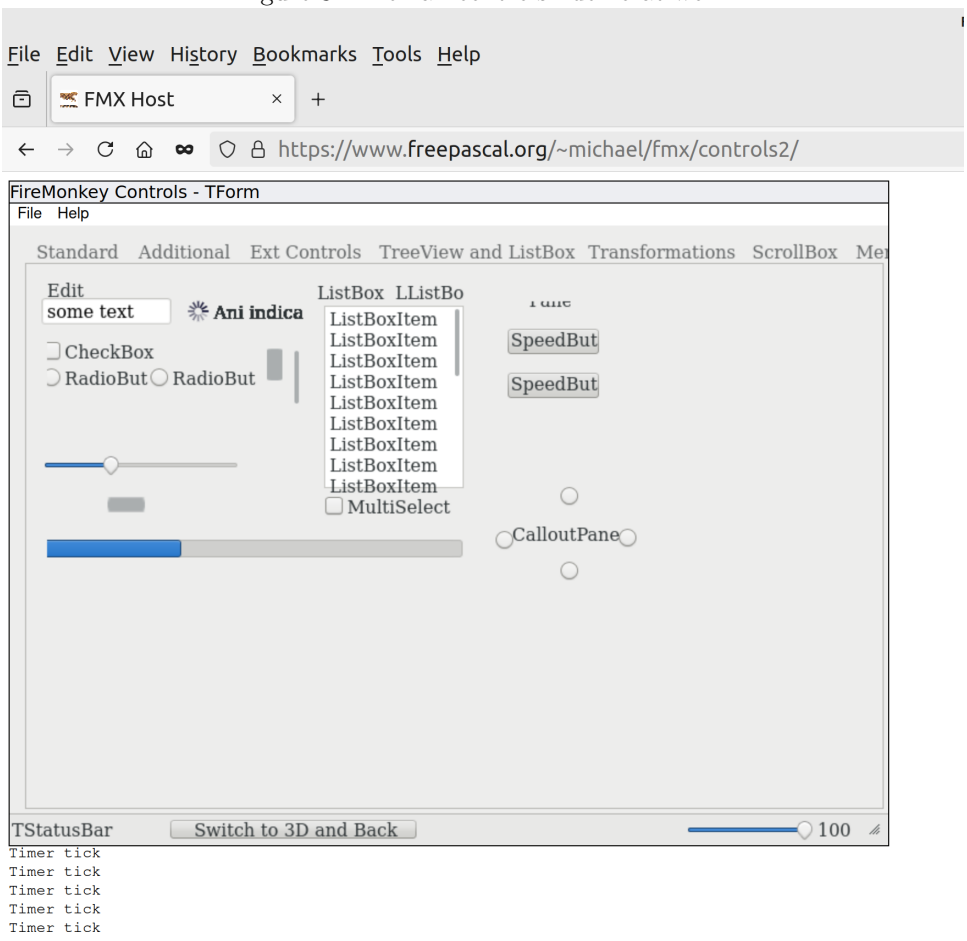


Figure 3: The "all controls" demo at work



8 Going further

The 'All controls' demo of firemonkey showcases all possible controls of firemonkey. It is a good test to see whether all the controls behave as expected. It will also show that the form loading functions correctly.

At the current time, some modifications are needed.

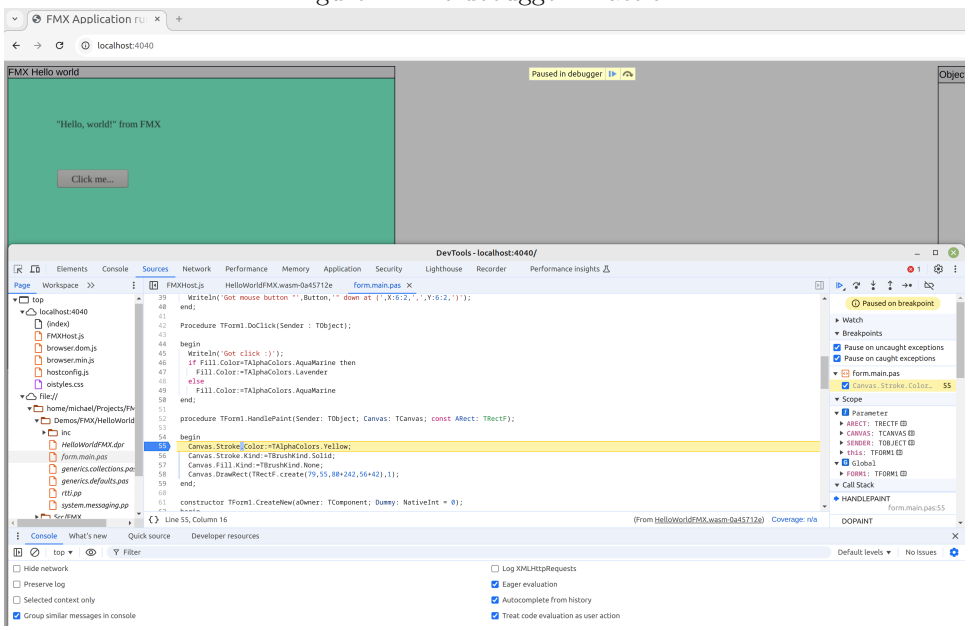
One of the possibilities of Firemonkey is to use 3D rendering. This obviously requires support of the backend, and at this point in time, this support has not yet been added to the backend - although it is planned to add this. So the button to switch to 3D must be removed (or at least disabled and the code behind it removed), as well as the units that refer to the 3D rendering engine.

Once that is done, the demo runs as intended with the exception of the listview and the treeview components. Currently, the failure of these components is still under investigation. The result can be seen in figure 3 on page 16.

9 Debugging

The astute reader will have noticed that when building the RTL and packages, the `-g` option was specified to the compiler. This means that the RTL and packages

Figure 4: The debugger in action



will be built with debug info. Also a 'names' section is generated which is usable by all browsers to display the names of functions.

To be able to use the debug information, you can install a plugin in the Chromium browser which will allow you to debug your FMX application in the browser. The plugin can be found here:

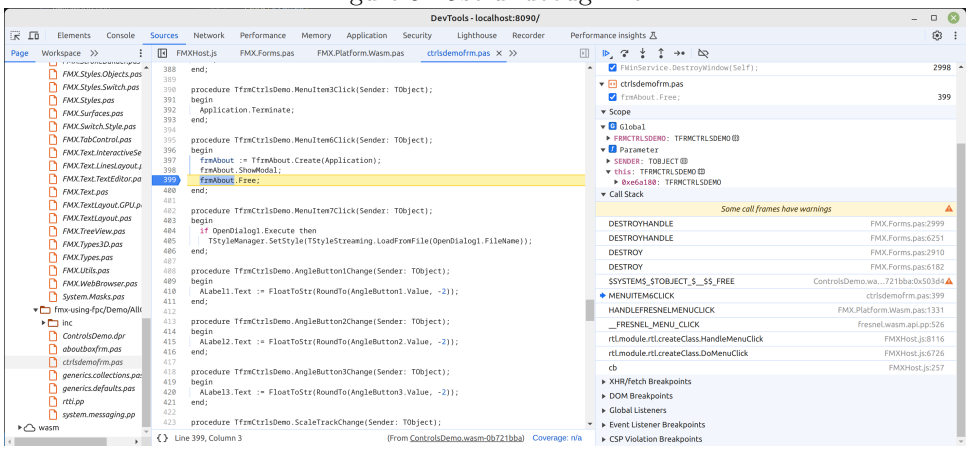
`https://chromewebstore.google.com/detail/cc++-devtools-support-dwa/pdcpmagijalfljmkmjngeonclg`

Once installed, you have lots of tools available to analyse what is happening in your program: you can put breakpoints, and step through your code as can be seen in figure 4 on page 17 and figure 5 on page 18. Note that the plugin is geared towards C/C++, so some types of information are not yet available (notably strings and properties). Nevertheless it makes the debugging experience a lot better than having to write messages to the browser console.

To enable a more comfortable way to debug, a module was developed that allows you to inspect objects using a more object-inspector kind of interface. This module allows you show a tree of live objects and to inspect them. To use this instrument, you need to add the `wasm.debuginspector.rtti` unit to your uses clause. This unit is included in the Free Pascal packages for the webassembly compiler.

```
TWasmDebugInspector = Class(TComponent)
Public
  constructor Create(aOwner: TComponent); override;
  destructor destroy; override;
  function ClearObjectTree: Boolean;
  function ClearObjectInspector: Boolean;
  function SendObjectProperties(aObject: TObject; aVisibilities: TMemberVisibilities): Boolea
  function SendObjectTree(aObject: TObject; const aCaption : string): Boolean; virtual;
  function SendObjectTree(aObject: TObject): Boolean; virtual;
  Property OnGetObjectChildren : TObjectChildrenEvent;
end;
```

Figure 5: Useful debug info




```
Function WasmDebugInspector : TWasmDebugInspector;
```

The `WasmDebugInspector` function returns a ready-to-use instance of this class, but you can create a descendant and use that if you wish to change its behaviour. The 2 important calls are

SendObjectTree which will show a tree structure with `aObject` at the top. Optionally you can display a caption. By default it will follow the owner-owned tree of objects to determine the children of an object, but you can change this behaviour using the `OnGetObjectChildren` event handler (more about this below).

SendObjectProperties will display the properties of object `aObject`, and you can specify the visibilities that must be shown. (an extension is planned where the fields can also be shown)

The `ClearObjectTree` and `ClearObjectInspector` calls will clear the object tree and object inspector displays.

In order to be able to show the tree or object inspector, you must of course also add something to the Javascript host program. The host side of the above object is implemented in the `TWasmDebugInspectorApi` class, implemented in the `debug.objectinspector.wasm.pas` unit, which you can add to the project uses clause, together with the `debug.objectinspector.html` and `wasm.debuginspector.shared` units;

The `TWasmDebugInspectorApi` class is just a bridge between the webassembly module and the Javascript environment. It uses 2 classes to manage the actual display: `THTMLObjectTree` and `THTMLObjectInspector`. These 2 objects are responsible for displaying the actual object tree and object inspector grid. They can be used with Pas2JS to show Pas2JS objects, if you want.

The tree and property grid must be shown somewhere in the HTML page, so we add some HTML tags to the body of our hosting page:

```
<div style="display: flex">
  <div id="desktop">
  </div>
```

```
  <div id="debug" style="display: flex">
    <div id="objectTree">
    </div>
    <div id="objectInspector">
    </div>
  </div>
</div>
```

You can see that we added an ID to each of the relevant elements, this is to be able to access the elements in code.

The object inspector and object tree make use of CSS, so that must be included as well in the `index.html` page.

```
<link href="oistyles.css" rel="stylesheet">
```

The constructor of our application is changed somewhat: To make the display of the object tree and object inspector optional, we add a boolean field `ShowOI` to the host config object, and we use that to show or hide the HTML:

```
if Assigned(hostConfig) then
  begin
  WasiEnvironment.LogAPI:=HostConfig.logWasiAPi;
  FFresnelApi.LogAPICalls:=HostConfig.logFresnelAPI;
  WasiEnvironment.Environment.Add('FMX_LOGLEVEL='+HostConfig.FMXLogLevel);
  DoShowOI:=HostConfig.ShowOI;
  end
else
  DoShowOI:=True;
if DoShowOI then
  ShowObjectInspector
else
  HideObjectInspector;
```

Where the `ShowObjectInspector` and `HideObjectInspector` calls do the actual work. The `HideObjectInspector` is simple: it changes the `display` style of the div in which we placed the object tree and object inspector:

```
procedure TFMXHostApplication.HideObjectInspector;
begin
  GetHTMLElement('debug').style.setProperty('display','none');
end;
```

The `ShowObjectInspector` is somewhat more complicated. It creates the 2 html renderers (`THTMLObjectTree` and `THTMLObjectInspector`) and sets the ID of the element where they should render the tree and grid:

```
procedure TFMXHostApplication.ShowObjectInspector;

begin
  FObjectTree:=THTMLObjectTree.Create(Self);
  FObjectTree.ParentElementID:='objectTree';
  FObjectInspector:=THTMLObjectInspector.Create(Self);
  FObjectInspector.ParentElementID:='objectInspector';
  FObjectInspector.Border:=True;
```

```
  FObjectInspector.VisibleColumns:=[{ocKind,ocVisibility,}ocName,ocValue];
  FObjectInspector.PropertyVisibilities:=AllMemberVisibilities;
  FObjectInspectorAPI:=TWasmObjectInspectorApi.Create(WasiEnvironment);
  FObjectInspectorAPI.DefaultInspector:=FObjectInspector;
  FObjectInspectorAPI.DefaultObjectTree:=FObjectTree;
  FObjectInspectorAPI.HandleInspectorEvents:=[Low(THandleInspectorEvent)..High(THandleInspect
  GetHTMLElement('debug').style.setProperty('display','flex');
end;
```

Lastly the API is created and connected to the HTML rendering objects.

Note that with this mechanism, the API will not be available if you set `ShowOI` to `false`. As a consequence, if your webassembly expects to have the debug inspector API available, it will fail to load. The code can be changed easily enough so that the API is always available, but the tree and property inspector grid are simply not shown...

On top of the owner-owned object tree defined by `TComponent`, FMX has 2 parent-child object trees when you create a form and populate it with controls. All graphical objects descend from `TFmxObject`, which are in a parent-child tree. But not all graphical objects are controls. The style elements are graphical objects, but are not controls. Therefore, controls form a separate tree from the graphical objects. The most interesting tree is the graphical objects, since they allow you to examine the style elements as well.

So we must have a callback for our `TWasmDebugInspector` class, which we'll call `DebugHelper`. It has a `DoChildren` method which is used as event handler for the `TWasmDebugInspector.OnGetObjectChildren` event.

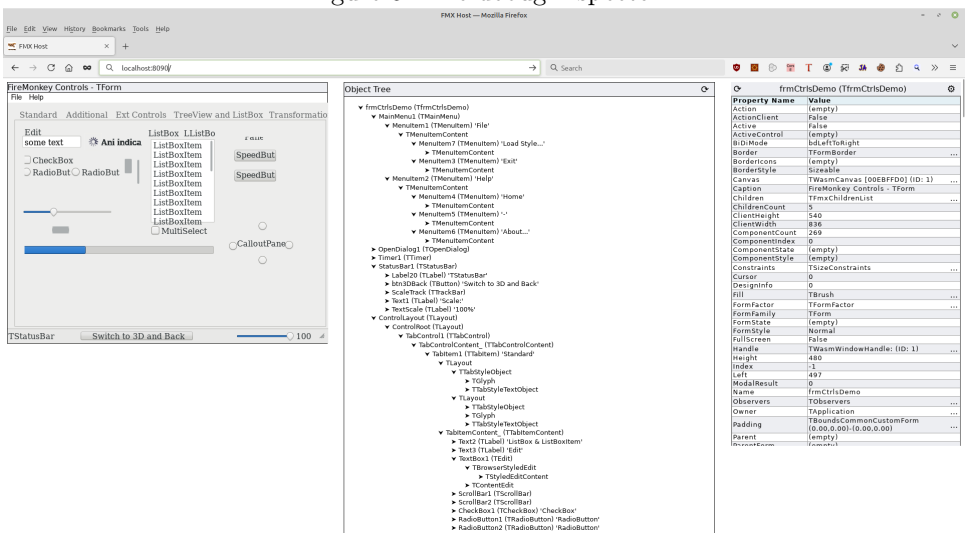
```
TDebugHelper = class
private
  FShowLog: Boolean;
  procedure DoChildren(aSender: TObject; aObject: TObject; var aChildren: TObjectDynArray; va
  procedure DoLog(Level: TWasmOILogLevel; const Msg: string);
  procedure DoObjectTree(ATree: TStrings; aObject: TfmxObject; const aPrefix: string);
Public
  constructor Create;
  destructor destroy; override;
  Procedure GetObjectTree(aTree : TStrings; aObject : TFmxObject);
  Procedure ShowObjectTree(aObject : TFmxObject; const aPrefix : String = '');
  Procedure SendObjectTree(aObject : TFmxObject);
  Procedure SendObjectTree(aObject : TFmxObject; const aCaption : string);
  Procedure SendObjectProperties(aObject : TFmxObject);
  property ShowLog : Boolean Read FShowLog Write FShowLog;
end;

var
  DebugHelper : TDebugHelper;

Procedure ShowObjectTree(aObject : TFmxObject; const aPrefix : String = '');
Procedure SendObjectTree(aObject : TFmxObject);
Procedure SendObjectTree(aObject : TFmxObject; aCaption : String);
Procedure SendObjectProperties(aObject : TFmxObject);
```

The plain procedures are for convenience, they just call the method of the same name on the `DebugHelper` instance of `TDebugHelper`. The `ShowObjectTree` methods

Figure 6: The debug inspector



show the object tree on the console. The `SendObjectTree` methods send the object tree to the object inspector.

With this class, the last thing to do is to show the object tree and the current object. In the `TfrmCtrlsDemo` form class, we add the following lines to the constructor:

```
SendObjectTree(Self);
SendObjectProperties(Self);
```

The result is visible in figure 6 on page 21. Clicking on an object in the object tree will show that object in the object inspector. You can use the ellipsis (...) symbol to inspect properties that are class values: the object inspector will show the object the property refers to. (there is a back arrow to come back). This allows you to navigate between objects, and see string typed values.

10 Conclusion

While not yet complete, thanks to the Fresnel backend, the port of FMX to the browser is already functional. Some controls still need some debugging, some other need still to be implemented. Enabling 3D rendering using WebGL is also on the drawing board. Work on FMX for the web progresses, and progress will be reported here, but for those that already wish to play with it (and possibly contribute) can already do so.