Threading in the browser

Michaël Van Canneyt

February 5, 2025

Abstract

In Delphi and Free Pascal, threading can be taken for granted: the `TThread` class makes it easy to launch a task in a thread. The browser knows no real threading, but has some alternatives. We investigate them here.

1 Introduction

When a task takes a long time, then it can be advantageous to execute it in a thread, so it is running in the background. This leaves the user free to do other work in the UI of the program.

This is no different in the browser: When executing Javascript code, the browser is not doing anything else: no events will be executed, no callbacks will be called. Only when the Javascript has finished executing will event callbacks be executed: you can easily test this by creating an infinite loop and a button with a 'click' event handler: while the loop is executing, the button click event will not be triggered.

You can easily test this with the following simple program:

```
program loop;

{$mode objfpc}

uses
  SysUtils, JS, Web;

var
  Count : Integer;
  Btn : TJSHTMLButtonElement;

procedure DoCLick(event : TJSEvent);

begin
  inc(Count);
  Btn.InnerText:='click '+IntToStr(count)+', click again?';
end;

begin
  Btn:=TJSHTMLButtonElement(Document.GetElementByID('btn'));
  Btn.addEventListener('click',@DoClick);
  Window.setTimeout(
    procedure
```

```
    var
       n : NativeInt;
    begin
      While true do
      N:=TJSDate.Now;
    end,3000
  );
end.
```

The script attaches a 'click' handler to a button. In this handler, the caption of the button is changed. After 3 seconds, the program launches an infinite loop. As soon as that loop is started, the button will no longer react to the click event. The browser will even display a warning 'this page is slowing down the browser' and offers you to stop the javascript program.

You can load this script with the following HTML page:

```
<html lang="en">
<head>
  <title>Project1</title>
  <script src="loop.js"></script>
</head>
<body>
  <button id="btn">Click me</button>
  <script>
    window.addEventListener("load", rtl.run);
  </script>
</body>
</html>
```

Javascript is single-thread in the browser. Since the browser knows no real threading, how can we execute long-running tasks in the browser ? The browser knows web workers: javascript programs that you can start and will run in the background while your web page is showing. They are used instead of actual threads, and for this reason you'll often see them referred to as threads, and the main page javascript is referred to as the main thread. In this article we'll show how to use web workers with pas2js. Later on we'll show how they can be used to simulate threads in a FPC webassembly program.

2 The web worker

To allow you to execute computatonally expensive tasks, the browser offers the web worker. The web worker is a separate process that you can start using Javascript in your main page. This new process will execute a javascript program and runs it separately from the main page javascript. The Javascript PDF viewer presented in other articles, uses this technique: it parses the PDF in a web worker, and the result of the parsing is sent back to the main HTML pages' Javascript.

In terms of natively executable programs, it is much like launching a separate executable, and not wait for it to finish.

To do so, you simply create a `TJSWebWorker` class and pass it the name of the javascript file to execute.

```
var
```

```
  Worker : TJSWebWorker;

begin
  Worker:=TJSWebWorker.New('myworker.js');
end;
```

This is not so different from starting a thread.

The web worker will execute as long as the web page is shown: when the page is close, the worker is stopped as well. There is a special kind of web worker that keeps on executing, even after you closed the page: the service worker. For the benefit of audio processing, there is also an "audio worklet" which is a web worker dedicated to the processing of audio data.

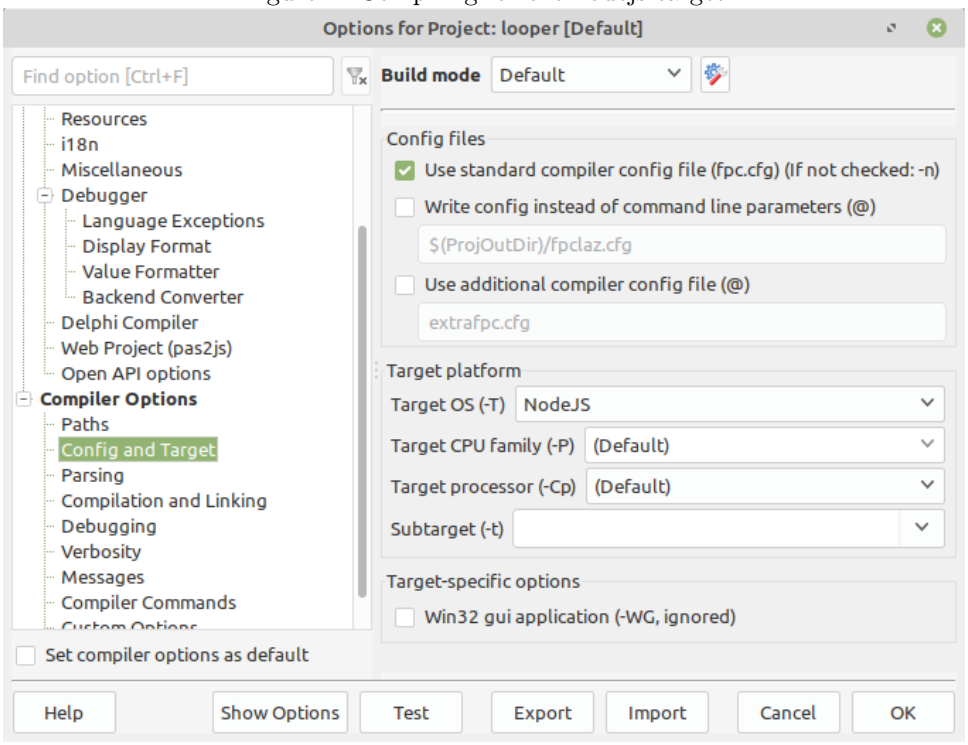
Since the web worker is executed in the background, it cannot interact with the DOM of your main page: only non-UI Javascript classes can be used in the web worker. This also means it cannot be used to directly execute event handlers for buttons and other UI elements.

Web workers are actually executed in a separate process. This means that they have different memory spaces, different Javascript exceution contexts: a javascript object created in the HTML pages' main Javascript code is not accessible in a worker, and vice versa. So they are not really threads: an essential feature of threads is that threads started in the same process can access each others' memory.

To show how this works, we'll rewrite the infinite loop demo so it runs the loop in a web worker after 3 seconds have passed:

```
program threadloop;

{$mode objfpc}

uses
  SysUtils, JS, Web;

var
  Count : Integer;
  Btn : TJSHTMLButtonElement;

procedure DoCLick(event : TJSEvent);

begin
  inc(Count);
  Btn.InnerText:='click '+IntToStr(count)+', click again?';
end;

begin
  Btn:=TJSHTMLButtonElement(Document.GetElementByID('btn'));
  Btn.addEventListener('click',@DoClick);
  Window.setTimeout(
    procedure
    var
      worker : TJSWorker;
    begin
      Worker:=TJSWorker.New('looper.js');
    end,3000
  );
```

Figure 1: Compiling for the nodejs target




```
end.
```

The 'looper.js' program is generated from the following pas2js code:

```
program looper;

uses js;

var
    n : NativeInt;
begin
  While true do
    N:=TJSDate.Now;
end.
```

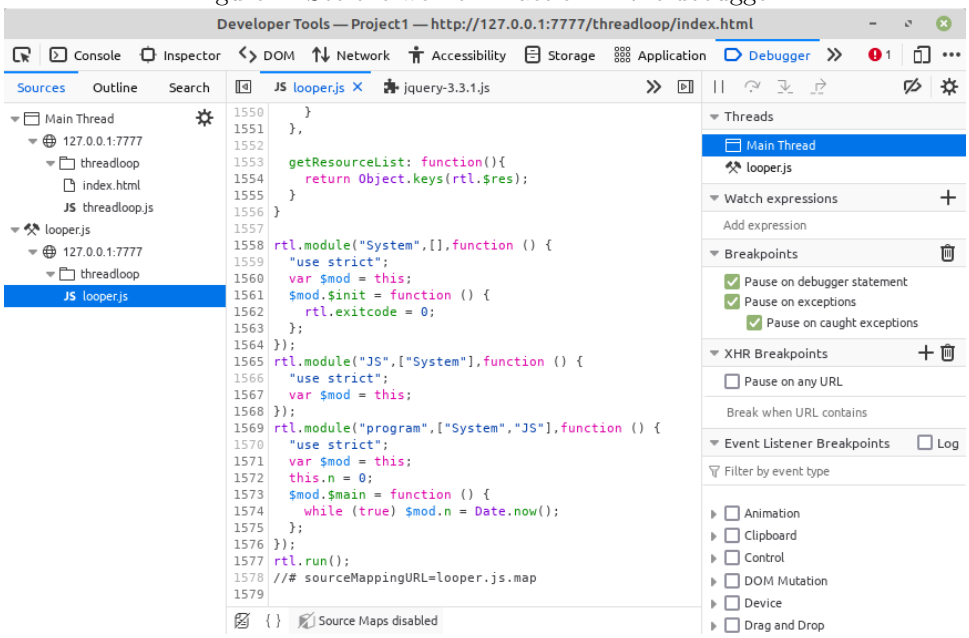
The worker program (`looper`) should be compiled for the `nodejs` target (set this in the 'Config and target' page of the compiler options, figure 1 on page 4). This will include the necessary code to actually start the program. Remember that in the HTML page, an explicit call to `rtl.run()` is included, and this call must be included at the end of the worker program. Setting the pas2js compile target to `nodejs` does that for you.

To load the new program, you need to change the script tag in index.html to

```
<script src="threadloop.js"></script>
```

If you now load the html page, the button's 'click' handler will remain functional after 3 seconds; the infinite loop is running in a background worker. You can see

Figure 2: See the worker in action in the debugger



this in the Javascript debugger, where each worker associated to the main page is also shown (see figure 2 on page 5) in the left side of the debugger. You can set breakpoints and debug a worker just like you would a regular pas2js program.

3 Communicating with the web worker

The example above simply runs an infinite loop in the worker. This is of course not very interesting. The idea of the web worker is to offload some tasks from the main html pages' Javascript to a separate process. This implies some form of communication between the main html page thread and the worker: the page tells the worker what to do, and the worker reports back the results to the page.

Since the worker and the main page cannot simply share objects as you would do in a native application, some other way to communicate is needed. The browser offers 2 ways:

1. Messages. You can send a message from the main page thread to the web worker, and vice versa. It's very easy to send a message to another thread: create a data object and send it. The browser will automatically serialize and deserialize the object (create a JSON representation and interpret it at the other end).

2. Shared memory and atomic memory operations. Shared memory and atomic memory operations have the advantage that they can be used for communicating between various workers in a blocking way. They require careful designing.

The shared memory technique will be discussed together with threads in webassembly in a next contribution. In this article, we'll concentrate on using messages.

Messages are sent over a channel. There are 2 kinds of channels:

- A message channel. This is a communication channel with 2 communication ends (called ports), one of which you can pass to a worker using a message. The ports can then be used to send messages to the other end in either direction.
- A broadcast channel: this is like a message channel, except that all workers can receive messages or send them on the channel.

When creating a worker, the browser automatically creates a channel between the thread that created the worker and the new worker. This channel is somewhat special: Using this channel, you can transfer some Javascript objects: the object will no longer be available to the sender, but is owned by the receiver once received.

The `TJSWorker` class that we used to start a worker, has a `PostMessage` method which the creator of the worker can use to send messages to the worker. It also has a 'message' event which you can hook into to listen to messages sent by the worker.

The worker global scope has a 'PostMessage' method which it can use to send messages to the thread that started it.

We'll demonstrate this messaging system with a small example: a simple calculator. We create a page with 2 edits, some labels and one button. When the button is clicked, the values in the edits are sent to a worker, which will add the numbers and sends back the result. The HTML looks like this:

```
<html lang="en">
  <head>
    <title>Calculator</title>
    <script src="main.js"></script>
  </head>
  <body>
    a: <input id="a" type="text"><p>
    b: <input id="b" type="text"><p>
    Result: <span id="result">?</span><p>
    <button id="btn">Add</button>
    <script>
      rtl.run();
    </script>
  </body>
</html>
```

As you can see, every item has an ID. The main program uses these to get references to the HTML elements:

```
program main;

{$mode objfpc}

uses
  JS, Classes, SysUtils, Web;

var
  a,b : TJSHTMLInputElement;
  Res : TJSHTMLElement;
  btn : TJSHTMLButtonElement;
  worker : TJSWorker;
```

```
Procedure BindElements;

begin
  a:=TJSHTMLInputElement(Document.getElementById('a'));
  b:=TJSHTMLInputElement(Document.getElementById('b'));
  res:=TJSHTMLInputElement(Document.getElementById('result'));
  btn:=TJSHTMLButtonElement(Document.getElementById('btn'));
  btn.addEventListener('click',@HandleCalc);
end;
```

The `HandleCalc` event handler will send a message to the worker (a reference to it is kept in the `Worker` variable):

```
procedure HandleCalc(Event : TJSEvent);

var
  lA,lB : Double;

begin
  lA:=StrToFloat(a.value);
  lB:=StrToFloat(b.value);
  Worker.PostMessage(New(['a', lA, 'b', lB]));
end;
```

The `New` function in the `JS` unit creates a small Javascript object, equivalent to the following javascript object literal:

```
{
  a: lA,
  b: lB
}
```

The object will be serialized, and then is sent to the worker where it is again deserialized (reconstructed).

The main part of the program initializes the worker variable and calls `BindElements`:

```
procedure HandleResult(Event : TJSEvent);

var
  lMessage : TJSMessageEvent absolute event;

begin
  res.InnerText:=String(lMessage.data);
end;

begin
  BindElements;
  worker:=TJSWorker.new('calc.js');
  worker.addEventListener('message',@handleresult);
end.
```

The `HandleResult` function in our `main` program will be called when the `calc` program sends back the result of the calculation.

The result of all this is that we send an object with the values of the 2 edits to the worker, and wait for a reply of the worker.

The worker listens for messages, does the calculation and sends back the result using `PostMessage`:

```
program calc;

uses
  JS, WeborWorker, WebWorker;

var
  Self_: TJSDedicatedWorkerGlobalScope; external name 'self';

procedure HandleMessage(Event : TJSEvent);
var
  lMessage : TJSMessageEvent absolute Event;
  lA,lB,lRes : Double;

begin
  lA := Double(TJSObject(lMessage.Data)['a']);
  lB := Double(TJSObject(lMessage.Data)['b']);
  lRes := lA + lB;
  Self_.postMessage(lRes);
end;

begin
  Self_.addEventListener('message',@HandleMessage);
end.
```

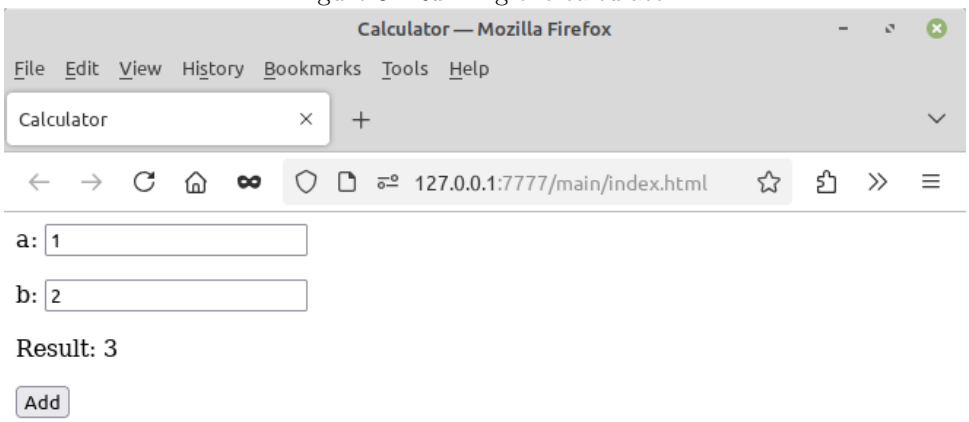
The `Self_` variable is the global scope of the web worker program: in the main HTML page, the 'window' variable plays the role of the global scope: all global variables and messages are actually members of the window variable. In a worker, there is no 'window' variable. There is only the 'self' variable. Since the `Self` identifier has special meaning in Object Pascal classes, we define 'Self_' instead, which is of class `TJSDedicatedWorkerGlobalScope` which contains all fields and methods of the global worker scope: this includes the `postMessage` method. This global `postMessage` method sends messages to the thread that created the worker (in our case, the `main` program), and only to that thread.

With all this code, we can now test the calculator program, and perform a simple calculation. The result can be seen in figure 3 on page 9. Recuperating the values of `a` and `b` from the message is a bit cumbersome. This can be made more readable by defining

```
TCommand = class external name 'Object' (TJSObject)
  a,b : double;
end;
```

The astute reader may have noticed that we use an external class for this, with as actual object the basic Javascript object. This has the effect that all the overhead (constructor code, initializing alll fields etc.) of a pascal class is not present. It's a nice technique to send literal objects while keeping full typesafety: the compiler checks that you only assign defined fields with the correct type, and additionally only the fields that are assigned will actually be present in the instance. You can think of it as a lightweight record.

Figure 3: Running the calculator



Using this definition, we can write code that is more readable and foolproof:

```
procedure HandleMessage(Event : TJSEvent);
var
  lMessage : TJSMessageEvent absolute Event;
  lCmd : TCommand;

begin
  lCmd := TCommand(lMessage.Data);
  lRes := lCmd.A + lCmd.B;
  Self_.postMessage(lRes);
end;
```

The same class can be used to send the message: by putting all message commands in a unit you can create a protocol between threads.

4 Transferring objects

The above example uses a very simple message object to communicate. Serializing and deserializing it will not take a lot of time. But what if the result of the calculation (or indeed the data on which to perform the calculation) is really big ? Well, the browser allows you to actually send an object using postmessage, but only for certain objects, such as memory buffers or image data.

To demonstrate this, we'll make something that is a little more computationally expensive, and which will resemble more a realistic use case than the examples we've presented till now.

Drawing complex scenes on a canvas or treating video images is something computationally intensive.

So, to make things easy to explain, we'll draw a rotating torus in 2D, using a gradient color shift between red and green along the circumference of the torus. To make things more interesting, we'll draw many such tori, each with a different value for blue in its RGB components (`aBlue` in the below function).

The torus will be drawn so it is smaller than the canvas on which it is drawn. This allows us to determine the torus radius (`lRadius` below), and we'll use a torus tube radius of 20% of the torus radius (`lTubeRadius`).

To draw the torus, we'll draw it in circles around the central axis, consisting of coloured segments. A constant `SegmentCount` determines the number of segments used to draw the torus: The angle `lSegmentAngle` varies from 0 to 2π in `SegmentCount` steps.

The torus will rotate around 2 axis `(X,Y)`, the rotation around these axes represented by `aAngleX`, `aAngleY`.

The torus is then projected on the screen with a "focal length" 500 - meaning that the Z axis is perpendicular to the screen and you're looking along the Z axis.

The beginning and end points of each segment are calculated by the local `CalcTorusPoint` function:

Due to the grid layout we used, the various canvases will change in size when the page resizes. For this reason, at the beginning of the routine, we set the size of the drawing canvas to the actual size of the canvas element: the `offsetWidth` and `OffsetHeight` properties of a HTML element can be used for this.

Putting all this together results in a function `drawTorus`, to which we pass the HTML Canvas and the 2D drawing context for that canvas:

```
procedure drawTorus(aCanvas : TJSHTMLCanvasElement;
                    aContext : TJSCanvasRenderingContext2D;
                    aAngleX, aAngleY : Double;
                    aBlue: Integer);
Type
  T2DPoint = Record
    x,y : Double;
  end;


var
  lCenterX,lCenterY : Double;
  lRadius,lTubeRadius : Double;

  Function CalcTorusPoint(aRingAngle, aSegmentAngle :Double) : T2DPoint;
  var
    lx,ly,lz,
    lRotatedX,lRotatedZ,
    lRotatedX2,lRotatedY2,
    lScale: Double;
  begin
    lX:=(lRadius + ltubeRadius * cos(aRingAngle)) * cos(aSegmentAngle);
    lY:=(lRadius + lTubeRadius * cos(aRingAngle)) * sin(aSegmentAngle);
    lZ:=(lTubeRadius * sin(aRingAngle));

    lrotatedX :=lX * cos(aAngleY) - lZ * sin(aAngleY);
    lrotatedZ :=lX * sin(aAngleY) + lZ * cos(aAngleY);

    lRotatedX2 := lRotatedX * cos(aAngleX) - lY * sin(aAngleX);
    lRotatedY2 := lRotatedX * sin(aAngleX) + lY * cos(aAngleX);

    lScale :=500 / (500 + lRotatedZ);
```

```
    Result.x := lCenterX + lRotatedX2 * lScale;
    Result.y := lCenterY + lRotatedY2 * lScale;
  end;

var
  lRingAngle,lSegmentAngle : Double;
  I,J : Integer;
  p1,p2 : T2DPoint;
  lGradient : TJSCanvasGradient;
  lSegmentColor : Double;

begin
  With aCanvas do
    begin
    // in case the window is resized.
    Width:=Round(offsetWidth);
    Height:=Round(offsetHeight);
    lCenterX:=width/2;
    lCenterY:=height/2;
    end;
  lRadius:=min(lCenterX,lCenterY) * 0.6; // Adjust size
  lTubeRadius:=lRadius*0.2;
  aContext.ClearRect(0,0,aCanvas.Width,aCanvas.Height); // Clear before drawing
  For I:=0 to RingCount do
    begin
    lRingAngle:=i/RingCount * Pi * 2;
    for j:=0 to SegmentCount do
      begin
      lSegmentAngle:=J/SegmentCount * Pi * 2;
      p1:=CalcTorusPoint(lRingAngle,lSegmentAngle);

      if (j<SegmentCount) and (i<RingCount) then
        begin
        lSegmentAngle:=(J+1)/SegmentCount * PI * 2;
        p2:=CalcTorusPoint(lRingAngle,lSegmentAngle);

        aContext.beginPath();
        lSegmentColor := j / SegmentCount;
        lgradient:=aContext.createLinearGradient(p1.X, p1.Y, p2.X, p2.Y);
        lgradient.addColorStop(0, Format('rgb(%d,%d,%d)',[
            Round(255 * (1 - lSegmentColor)),
            Round(255 * lSegmentColor),
            aBlue]));
        lgradient.addColorStop(1, Format('rgb(%d,%d,%d)',[
            Round(255 * (1 - (j + 1) / SegmentCount)),
            Round(255 * (j + 1) / SegmentCount),
            aBlue]));
        aContext.strokeStyle:=lgradient;
        aContext.moveTo(p1.X, p1.Y);
        aContext.lineTo(p2.X, p2.Y);
        aContext.stroke();
        end;
      end;
    end;
```

```
end;
```

Note the creation of the linear gradients: they calculate the begin and end value of the gradient based on the segment position.

In a first version, we'll use this routine to draw the toruses on the canvas in the main thread. We'll assume a 5x5 grid of toruses, and we'll let the browser position the canvases using some CSS, the grid display is just what we need for this:

```
<!doctype html>
<html lang="en">
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8">
  <title>Torus demo</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="toruses.js"></script>
  <style>
    body {
      margin: 0;
      overflow: hidden;
      display: grid;
      grid-template-columns: repeat(5, 1fr);
      grid-template-rows: repeat(5, 1fr);
      height: 100vh;
    }

    canvas {
      display: block;
      width: 100%;
      height: 100%;
    }
  </style>
</head>
<body>
  <script>
    rtl.run();
  </script>
</body>
</html>
```

In our program code, we can start by creating the canvases and positioning them in the HTML page. We also calculate the RGB blue component for each of the tori:

```
const
  RowCount = 5;
  ColCount = 5;
  SegmentCount = 30;
  RingCount = 20;
  CanvasCount = RowCount * ColCount;

var
  canvases : Array[0..CanvasCount-1] of TJSHTMLCanvasElement;
  contexts : Array[0..CanvasCount-1] of TJSCanvasRenderingContext2D;
  Blues : Array[0..CanvasCount-1] of Integer;
```

```
Procedure InitCanvases;

var
  i : integer;

begin
  for I:=0 to CanvasCount-1 do
    begin
    Canvases[i]:=TJSHTMLCanvasElement(document.createElement('canvas'));
    contexts[i]:=TJSCanvasRenderingContext2D(Canvases[i].getContext('2d'));
    document.body.appendChild(canvases[i]);
    Blues[I]:=(255 div CanvasCount)*I;
    end;
end;
```

The torus rotation animation is achieved with the `RequestAnimationFrame`. This browser routine allows you to tell the browser you wish to create an animation, and register a callback that will be called when the browser next redraws the screen. Whenever you need to do animation, this is the optimal method to ensure the drawing happens at the appropriate time. The exact timing used by the browser depends on the screen refresh rate. The `RequestAnimationFrame` callback is also not triggered if the page is not visible.

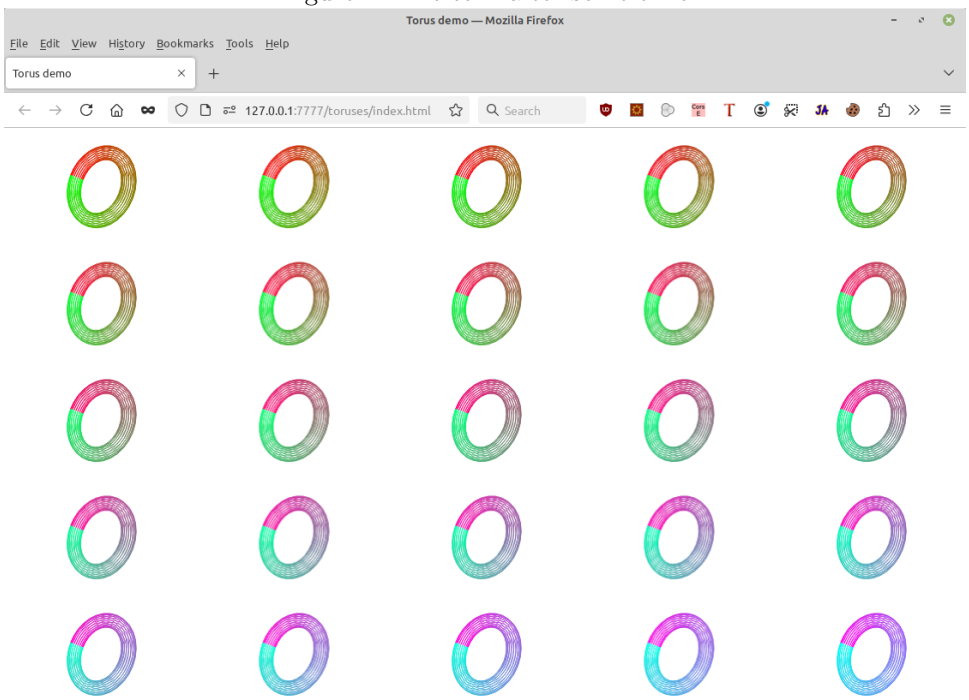
It would be a bad idea to use a timer for this, as timer ticks tend to accumulate if the drawing takes too long, and timer ticks are also triggered when the page is not visible, thus draining your battery. `RequestAnimationFrame` is no.

In the animation callback, we increase the rotation angles (`angleX` and `angleY`), and we actually draw all the tori on their respective canvases:

```
var
  angleX : Double = 0;
  angleY : Double = 0;

procedure DoDraw;

var
  I : Integer;

begin
  for I:=0 to CanvasCount-1 do
    drawTorus(Canvases[i],Contexts[i],AngleX,AngleY,Blues[i]);
end;

procedure animate(aTime: TJSDOMHighResTimeStamp);

begin
  angleY := angleY+0.01;
  angleX := angleX+0.005;
  DoDraw;
  window.requestAnimationFrame(@animate);
end;
```

Since the animate callback is called only once for each call to `requestAnimationFrame`, we must call it again when everything was drawn.

Figure 4: The torii after some time



When the browser window is resized, the size of the canvases will change (due to the grid css). So we must redraw:

```
procedure DoResize(aEvent : TJSEvent);

begin
  DoDraw;
end;
```

With all this in place, we can now set the ball rolling in the program's main block: we initialize the canvases, call `Animate` a first time, and register a callback for the window `resize` event:

```
begin
  InitCanvases;
  Animate(0);
  window.addEventListener('resize', @DoResize);
end.
```

The result will look like figure 4 on page 14. You'll notice that the page is not very responsive, and that the torii are rotating slowly: this is because the routine to draw the torii is taking a lot of CPU. Chromium will actually complain that the `RequestAnimationFrame` callback is too slow.

To speed things up, we will now move the drawing of the torii to a web worker (a 'thread'). For simplicity, we'll create a worker per canvas.

We need to be able to send 2 messages to the worker: One to initialize the drawing: the index of the canvas for which to draw, the value of the `blue` component, and the initial size of the canvas.

When the page size changes, the new dimensions of the canvases must be communicated to the workers, so we will also need a message for that.

The worker will use this information to create a so-called offscreen canvas: this is a canvas which is not displayed - in essence, a bitmap on which you can draw as you would on an actual canvas element. Such a canvas can be created in a web worker.

Once the worker has finished drawing a torus, it must send the generated image back to the main thread. The content of an offscreen canvas can be converted to a bitmap (a `TJSImageBitmap` instance), and this bitmap can be transferred to the main thread: the `PostMessage` can transfer such a bitmap in an efficient way.

To ease the sending all these messages, we will define 2 classes. Each class has a 'cmd' field to indicate the command. and a 'idx' field to indicate for which torus the command is used. We'll put these classes in a separate unit:

```
const
  cmdFrame = 'frame';
  cmdInit = 'init';
  cmdResize = 'resize';

type
  TCommand = class external name 'Object' (TJSEvent)
    cmd : string;
    idx : integer;
    fps : Integer;
    width : integer;
    height : integer;
    blue : integer;
  end;

  TFrameCommand = class external name 'Object' (TJSEvent)
    cmd : string;
    bitmap : TJSImageBitmap;
    idx : Integer;
  end;
```

For the worker thread, we define a small class to keep all the parameters needed to draw the torus:

```
TTorusRenderer = class(TObject)
  FBuffers : Array[0..1] of TJSHTMLOffscreenCanvas;
  FContexts : Array[0..1] of TJSOffscreenCanvasRenderingContext2D;
  FBufIndex,
  FIndex,
  FFPS,
  FWidth,
  FBlue,
  FHeight : Integer;
  FLast : NativeInt;
  FAngleX,FAngleY : Double;
  Constructor create(aIndex,aFPS,aBlue,aWidth,aHeight : Integer);
  procedure allocateBuffers;
  procedure SendFrame(aIdx: Integer);
  procedure DoRender;
  Procedure DoTick;
```

```
  Procedure NewSize(aWidth,aHeight : Integer);
end;
```

You see that the object uses 2 buffers: one which is being drawn on in the DoTick event, the other is sent to the main thread. When done, the buffers are swapped.

In the constructor, we initialize all fields with the parameters sent by the main thread, and we allocate the buffers:

```
constructor TTorusRenderer.create(aIndex, aFPS, aBlue, aWidth, aHeight: Integer);
begin
  FBufIndex:=0;
  FIndex:=aIndex;
  FFPS:=aFPS;
  FWidth:=aWidth;
  FHeight:=aHeight;
  FBlue:=aBlue;
  AllocateBuffers;
  DoRender;
  Self_.setInterval(@DoTick,Round(1000/FFPs));
end;

procedure TTorusRenderer.allocateBuffers;
var
  I : Integer;
begin
  For I:=0 to 1 do
    begin
    FBuffers[I]:=TJSHTMLOffscreenCanvas.New(FWidth,FHeight);
    FContexts[I]:=FBuffers[I].getContextAs2DContext('2d');
    end;
end;
```

Here you see how to create an offscreen canvas: you can simply use the `TJSHTMLOffscreenCanvas` class, the constructor needs only the width and height of the canvas to create. After that, the drawing context can be retrieved just as for a regular canvas.

Once the canvas and drawing contexts are allocated, the initial image is drawn and sent to the main thread. To draw the torus, we call the function we created earlier, passing it the context and the required other parameters:

```
procedure TTorusRenderer.DoRender;

begin
  drawTorus(FContexts[FBufIndex],FWidth,FHeight,FAngleX,FAngleY,FBlue);
  SendFrame(FBufIndex);
end;
```

Note that we're not passing the actual canvas element but just the width and height. The reason will be explained shortly.

To send the frame, we transfer the content of the offscreen canvas to a bitmap using `transferToImageBitmap`. To actually send it we make use of the `TFrameCommand` class and the `PostMessage` command:

```
procedure TTorusRenderer.SendFrame(aIdx : Integer);
```

```
var
  Cmd : TFrameCommand;

begin
  cmd:=TFrameCommand.New;
  cmd.cmd:=cmdFrame;
  cmd.bitmap:=FBuffers[aIdx].transferToImageBitmap();
  cmd.idx:=FIndex;
  self_.postMessage(Cmd,[Cmd.bitmap]);
end;
```

Note that the second parameter to the `PostMessage` is an array with a single element; the second parameter of `PostMessage` is the list of objects that are part of the first argument (the data) and which you wish to transfer to the message receiving thread instead of simply serializing them. After it was transferred, the object is no longer usable in the sending thread.

In our case, we pass the bitmap object created with `transferToImageBitmap` to the main thread.

When the size of the page changes, the canvases in the main page change size, and this new size is communicated through a message.

Upon receipt of the message, we set the new size on our renderer class:

```
procedure TTorusRenderer.NewSize(aWidth, aHeight: Integer);
begin
  FWidth:=aWidth;
  FHeight:=aHeight;
  AllocateBuffers;
  DoRender;
end;
```

We reinitialize the canvas and context, after which we redraw and send the new torus image. This is also why we no longer pass the canvas to the `DrawTorus` function as in our first version of the `DrawTorus` function. Instead we can simply pass the canvas width and height: the resizing of the canvas happens here.

The last instruction in the constructor of the `TTorusRenderer` class was to start a timer: in the timer, we animate our torus, we rotate the torus, swap buffers and redraw it:

```
procedure TTorusRenderer.DoTick;

var
  lNow : NativeInt;

begin
  lNow:=TJSDate.now;
  if lNow=FLast then exit;
  FLast:=lNow;
  FAngleY := FAngleY+0.01;
  FAngleX := FAngleX+0.005;
  FBufIndex:=1-FBufIndex;
  DoRender;
end;
```

Note that we check the timestamp of the tick: if the timestamp didn't change, we do not redraw.

With this class, we can now create our worker. Its main routine is not so different from the previous worker, it simply registers a message event listener `HandleMessage`. In the message handler, depending on the kind of message, we create the torus renderer object or resize it:

```
var
  Renderer : TTorusRenderer;

Procedure HandleMessage(E : TJSEvent);

var
  lMessage : TJSMessageEvent absolute e;
  Cmd : TCommand;

begin
  Cmd:=TCommand(lMessage.Data);
  Case Cmd.cmd of
    cmdInit:
      Renderer:=TTorusRenderer.Create(cmd.idx,cmd.fps,Cmd.blue,cmd.Width,cmd.height);
    cmdResize:
      if assigned(Renderer) then
        Renderer.NewSize(cmd.width,cmd.height);
  end;
end;

begin
  Self_.addEventListener('message',@handlemessage);
end.
```

This concludes the worker part of our program. In the main program, we now no longer need the torus drawing routine. Instead, we simply draw the images that we received on the canvases. Since we no longer need to transform the image, this is a fast operation.

Since for each torus we need to keep several pieces of information: the canvas, the context, the worker which will draw the torus and the actual image, we'll create a record to hold that information, and define an array of these records:

```
Type
  TTorusData = record
    worker : TJSWorker;
    canvas : TJSHTMLCanvasElement;
    context : TJSCanvasRenderingContext2D;
    blue : Integer;
    Bitmap : TJSImageBitmap;
  end;

var
  Torus : Array[0..CanvasCount-1] of TTorusData;
```

The `bitmap` field will be filled with the bitmap sent by the worker thread.

The array is initialized at the start of the program, as in the previous version:

```
Procedure InitCanvases;

var
  i : integer;
  lCanvas : TJSHTMLCanvasElement;

begin
  for I:=0 to CanvasCount-1 do
    begin
    Torus[i].Blue:=(255 div CanvasCount)*I;
    lCanvas:=TJSHTMLCanvasElement(document.createElement('canvas'));
    // Set initial size
    lCanvas.Width:=100;
    lCanvas.Height:=100;
    Torus[i].Canvas:=lCanvas;
    Torus[i].context:=TJSCanvasRenderingContext2D(lCanvas.getContext('2d'));
    Torus[i].Blue:=round(255*I/CanvasCount);
    document.body.appendChild(lCanvas);
    end;
end;
```

After initializing the array, we can initialize the workers that will draw the tori. We create the worker, and send it the 'init' command.

```
Procedure InitWorkers;

var
  i : integer;
  Cmd : TCommand;
  lWorker : TJSWorker;

begin
  for I:=0 to CanvasCount-1 do
    begin
    lWorker:=TJSWorker.New('torusworker.js');
    Torus[i].Worker:=lWorker;
    cmd:=TCommand.New;
    cmd.cmd:=cmdInit;
    cmd.idx:=i;
    cmd.width:=Round(Torus[i].canvas.OffsetWidth);
    cmd.height:=Round(Torus[i].canvas.OffsetHeight);
    cmd.fps:=30;
    cmd.blue:=Torus[i].blue;
    lWorker.postMessage(cmd);
    lWorker.addEventListener('message',@HandleWorkerMessage);
    end;
end;
```

After sending the 'init' command to the worker we register a message handler `HandleWorkerMessage` for the worker.

In this message handler we recuperate the bitmap and store it in the appropriate record:

```
procedure HandleWorkerMessage(aEvent : TJSEvent);
```

```
var
  lMessage : TJSMessageEvent absolute aEvent;
  lCmd : TFrameCommand;
  Idx : Integer;

begin
  lCmd:=TFrameCommand(lMessage.Data);
  if lCmd.cmd=cmdFrame then
    begin
    Idx:=lCmd.idx;
    if (Idx>=0) and Assigned(lCmd.bitmap) then
      begin
      if assigned(Torus[Idx].Bitmap) then
        Torus[Idx].Bitmap.Close;
      Torus[Idx].Bitmap:=lcmd.Bitmap;
      end;
    end;
end;
```

The line

```
Torus[Idx].Bitmap.Close;
```

Tells the browser that it can discard the bitmap data: Javascript is a garbage collected language. If you generate lots of images, then memory will increase till the garbage collector kicks in. By calling 'close' on the bitmap, the actual bitmap data is already freed, thus reducing memory usage.

The drawing of the bitmaps is done in the `DoDraw` routine:

```
procedure DoDraw;

var
  I : Integer;
  lData : TTorusData;

begin
  for I:=0 to CanvasCount-1 do
    begin
    lData:=Torus[i];
    lData.Context.ClearRect(0,0,lData.Canvas.Width,lData.Canvas.Height); // Clear before draw
    if Assigned(lData.Bitmap) then
      lData.context.drawImage(lData.Bitmap,0,0);
    end;
end;
```

The `DoDraw` routine is called as part of the `requestAnimationFrame` callback (`animate`):

```
procedure animate(aTime: TJSDOMHighResTimeStamp);

begin
  DoDraw;
  window.requestAnimationFrame(@animate);
end;
```

When the browser window is resized, the canvases will be resized, and we need to send the new size to all workers.

This happens in the `DoResize` callback:

```
procedure DoResize(aEvent : TJSEvent);

var
  i,lWidth,lHeight : integer;
  Cmd : TCommand;
  lData : TTorusData;

begin
  for I:=0 to CanvasCount-1 do
    begin
    lData:=Torus[i];
    cmd:=TCommand.New;
    cmd.cmd:=cmdResize;
    lWidth:=Round(lData.canvas.OffsetWidth);
    lHeight:=Round(lData.canvas.OffsetHeight);
    cmd.width:=lWidth;
    cmd.height:=lHeight;
    lData.canvas.width:=lWidth;
    lData.canvas.height:=lHeight;
    lData.Worker.postMessage(cmd);
    end;
end;
```

This explains why a reference to the worker is kept instead of simply a torus index: we need the actual worker instance to be able to send the resize message.

All that is needed is to set the ball rolling: initialize the canvas and workers, set the correct size, and then request the first animation frame:

```
begin
  InitCanvases;
  InitWorkers;
  window.addEventListener('resize', @DoResize);
  DoResize(nil);
  window.requestAnimationFrame(@animate);
end.
```

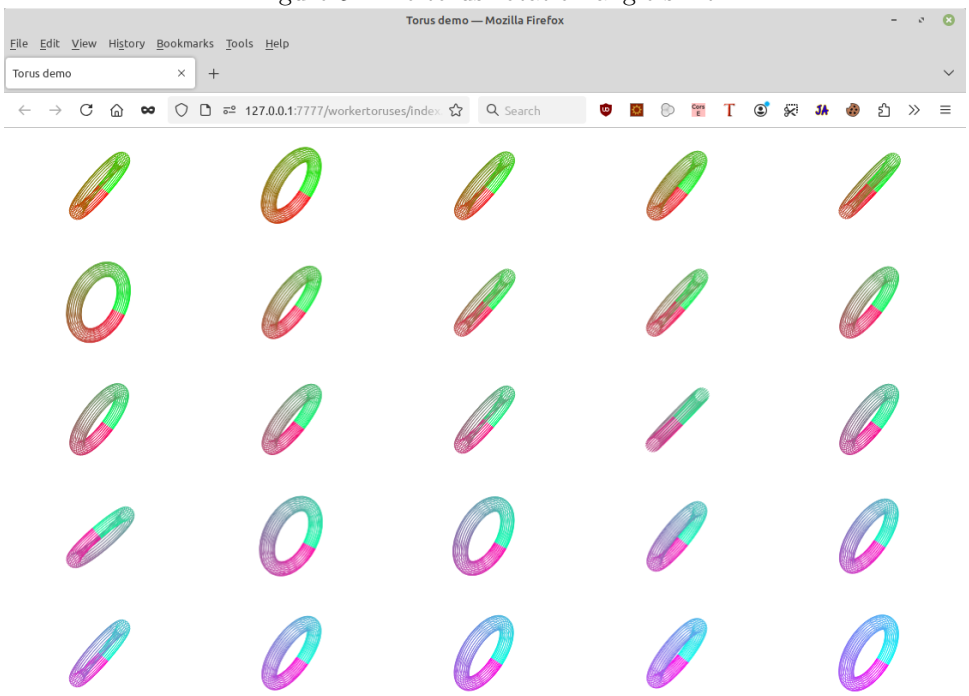
With this change the worker-based version of our project is now done.

One way can we spot the difference is by looking at the positions of the top-left and bottom-right torus: because the workers are started one after the other, the timer ticks start with some offset: the rotation of the last torus starts therefore at a later time and the rotations are shifted. You can see this in figure 5 on page 22.

If you try this example, it is worth to note that it works well in firefox. It does not work so well in Chrome or Chromium-based browsers: The reason is the use of the gradient: due to what we presume is an inefficient way to handle gradients, the drawing in Chrome does not work well: removing the gradient and just drawing the segments in a solid color solves the issue, and the result will resolve smoothly.

This change can be done quite simply using an IFDEF conditional define in the `TorusDraw` function:

Figure 5: The torus rotation angle shift




```
  lRGB:=Format('rgb(%d,%d,%d)',[
      Round(255 * (1 - lSegmentColor)),
      Round(255 * lSegmentColor),
      aBlue]);
{$IFDEF USEGRADIENT}
  lgradient:=aContext.createLinearGradient(p1.X, p1.Y, p2.X, p2.Y);
  lgradient.addColorStop(0, lRGB);
  lRGB:=Format('rgb(%d,%d,%d)',[
    Round(255 * (1 - (j + 1) / SegmentCount)),
    Round(255 * (j + 1) / SegmentCount),
    aBlue]);
  lgradient.addColorStop(1, lRGB);
  aContext.strokeStyle:=lGradient;
{$ELSE}
  aContext.strokeStyle:=lRGB;
{$ENDIF}
```

The result is a slightly less smooth color transition along the torus' surface, but the rotation is now smooth in Chromium as well.

Chrome also does not handle the grid layout very well: the canvases are not correctly layed out.

Luckily, none of this is relevant to the technique we're trying to demonstrate: the point of this article is not so much to draw a torus, the point is to demonstrate how to offload tasks to a web worker.

5 Conclusion

In this article, we showed how to create simple and more complicated workers, a technique used in many modern web pages. The PDF viewer uses similar techniques, and many websites for example for online word processors use similar techniques.

In a next article, we'll show how to use these workers to implement `TThread` for a webassembly program runnning in the browser.