# Converting terrain generation functions to Web Assembly (v2 - revised)

This is a report on my first attempt to convert the most heavily used functions in my browser-based terrain generator from Javascript (JS) to Web Assembly (WASM). The aim was to make these functions, and therefore the whole terrain generation process, faster. The height and terrain calculators have been converted successfully with a 3-fold speed gain in Firefox but it seems other browsers are not so capable yet. Nevertheless I have implemented my new versions in a published program, with a proviso that will be explained here later.

The text format for WASM is poorly documented online and so the conversion process involved considerable trial and error. The source files for a full working example will be presented here, with the aim of helping other developers.

My terrain generator is described in detail in a PDF on github. It may be helpful to read at least the early sections of that first.

Heights are calculated by `function calcHeight (x, y)` where `x` is measured eastwards and `y` northwards on the ground. As used in The Forest (HTML/JS) this produces contour maps of the kind shown here. This particular example has x running from 23396 to 24195 and y from 5318 to 5917. That is an area covering 800 x 600 metres. So `calcHeight()` was called 480,000 times to generate this map fragment.
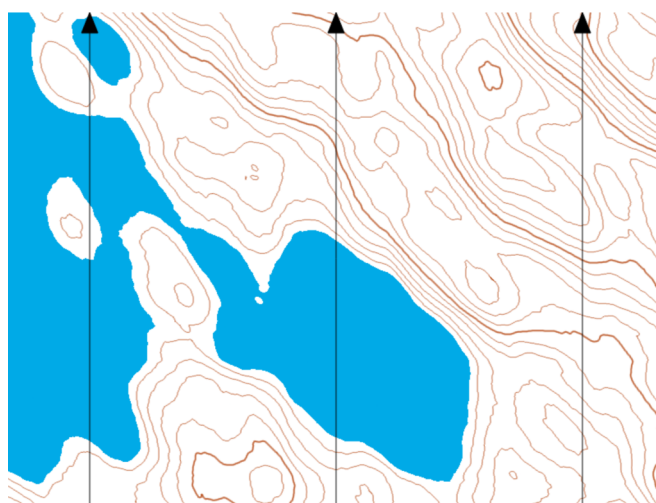
I started to learn the source text for WASM on Mozilla Developer Network (MDN). That is a reasonable start but it only gives a range of example snippets rather than being a thorough tutorial. All relevant documents I can find online at present are like that. It seems that there is no complete manual available yet. It also seems that most people are using cross-assemblers from other languages at present, rather than writing WASM directly as source text (in .wat files).

It is worth noting that MDN has a useful WebAssembly Reference page but it was not linked from the main guide. I only found it by accident but I used it a lot.

My first development hurdle was that the free IDE I usually use, Apache Netbeans (with HTML/JS plug-in for my browser-based work), is not yet able to supply the correct MIME type of "application/wasm" when serving .wasm files from its localhost server. Nor would it let me set such a MIME type.

So I installed Visual Studio Code (VSC) because the main Web Assembly web site showed that there is a VSC extension for handling .wat files (the WASM text format) and very simply also saving them assembled as .wasm. You can get the extension here. Another extension of VSC (here) enables me to test in localhost mode on my preferred browser, Firefox. Apart from having to learn how to use VSC, that all works very well. I can recommend this set-up (VSC plus 2 extensions). It seems to be the

simplest way to make .wasm files without having to install a large amount of complicated software.

First I show here the output from my test program that compares the performance of the original JS version of `calcHeight()` with the converted WASM version. It demonstrates that the WASM version produces the same output and that it is significantly faster.

The first table uses some random values of x and y to see whether the calculated heights are the same from both functions. There is a tiny difference, at most 0.3 mm. That is slightly puzzling because both JS and WASM use the standard representation of floating point numbers. Such a tiny difference is insignificant for the height map but it may be important in a later stage when I convert the function for generating point features scattered around. I have an idea about the cause of the difference and I will shortly investigate that.

Some random (x, y):

x	y	old ht	wa ht	Δ
2966.8242	-4069.1841	229.6382	229.6382	-0.0000
2828.7965	1180.7077	270.2930	270.2930	-0.0001
-1781.1949	-4587.8112	304.2154	304.2152	0.0001
-3314.7576	-678.4057	380.0891	380.0891	-0.0000
247.1096	-3870.8726	300.2132	300.2131	0.0001
4350.8734	1145.9241	209.7752	209.7752	-0.0000
3204.7328	-4840.1291	282.6710	282.6708	0.0003
-231.0913	4479.8578	264.9243	264.9242	0.0001
1516.7637	3207.6690	466.0985	466.0985	-0.0001

4 million calls each time:

	n	mean (ms)	std.dev.
calcHeight(x, y)	100	278.53	17.79
waCalcHeight(x, y)	100	95.79	8.66

Update: The cause was found as I went further. It was because I had used `f32` for my data whereas numbers in JS are always 64-bit floating. So the present document has been revised to show `f64`.

The second table above shows the time taken by 4 million calls of each version, in milliseconds. This was in Firefox running on Windows 11 in a Samsung Galaxy 360 laptop with an Intel i7 processor. The test was run 100 times because a multitasking system such as Windows will not produce the same measurement every time. I reported the average and standard deviation of the 100 values.

The original Javascript

Here is an extract from my JS version of the terrain generator, just showing the function `calcHeight()` and the data it needs.

```
// The length of this array MUST be a power of 2:
const PROFILE = [
 77, 80, 84, 88, 92, 96,101,104,108,112,115,118,120,123,126,129,
131,133,134,134,133,133,131,130,129,126,123,122,122,122,123,125,
126,130,134,137,137,138,138,137,135,133,129,123,118,111,105,101,
 97, 93, 90, 86, 82, 78, 74, 71, 69, 67, 67, 67, 66, 67, 69, 71,
 73, 74, 73, 73, 71, 69, 66, 62, 58, 54, 52, 52, 54, 55, 58, 59,
 62, 63, 63, 65, 65, 65, 66, 66, 67, 69, 70, 73, 77, 80, 82, 85,
 88, 90, 93, 95, 96, 96, 96, 96, 93, 92, 90, 85, 80, 75, 71, 67,
 63, 60, 58, 55, 52, 50, 47, 44, 43, 41, 40, 39, 36, 35, 33, 32,
 30, 28, 24, 20, 15, 11,  7,  3,  2,  2,  2,  2,  2,  2,  3,  6,
  7, 10, 11, 15, 18, 22, 24, 25, 25, 26, 26, 25, 25, 25, 25, 25,
 26, 28, 29, 30, 33, 36, 37, 39, 39, 40, 40, 40, 39, 39, 39, 37,
 37, 37, 36, 36, 36, 35, 35, 33, 33, 32, 30, 28, 25, 20, 15, 11,
 10,  9,  9,  9,  9, 11, 14, 15, 17, 17, 18, 18, 18, 18, 18, 18,
 17, 17, 17, 15, 14, 13, 11, 11, 10, 10, 10, 11, 13, 14, 17, 20,
 22, 25, 28, 30, 35, 39, 41, 45, 50, 58, 63, 69, 73, 77, 80, 82,
 84, 84, 85, 85, 84, 84, 82, 81, 80, 75, 73, 71, 71, 73, 74, 75];
```

```
const AH = [0,13,21,22,29];
const BH = [27,26,21,11,1];

const R128 = 1 / 128;
const BITMASK = PROFILE.length - 1; // All bits set

function calcHeight(x, y)
{ var ht = 0;
  for (var i = 0; i < 5; i++)
  { var j = (AH[i] * x + BH[i] * y) * R128;
    var jint = Math.floor (j);
    var jfrac = j - jint;
    var prof0 = PROFILE[jint & BITMASK];
    var prof1 = PROFILE[(jint + 1) & BITMASK];
    ht += prof0 * (1 - jfrac) + prof1 * jfrac; // interpolate
  }
  return ht;
}
```

Operators in WASM must always have operands of the same type. Notice particularly that the variable `jint` is used in two ways above: in a subtraction to form `jfrac` and as part of the indexing into `PROFILE`. In the first of those it will need to be a floating point value, type `f64` in WASM, but for indexing it needs to be a true integer of type `i32`. So two versions of `jint` are needed and I called them `jint_f` and `jint_i`.

The source files for the tests

terrain.html

I used a very basic HTML page as shown alongside here. It brings in two JS files, of which terrain.js is the one which will invoke a .wasm file (assembled WASM). The other script, report.js, runs the tests and puts the results in the two paragraph elements seen here.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Terrain wasm</title>
    <meta charset="UTF-8">
    <meta name="viewport"
content="width=device-width, initial-
scale=1.0">
  </head>
  <body onload="run()">
    <p id="table"></p>
    <p id="times"></p>
    <script src="terrain.js"></script>
    <script src="report.js"></script>
  </body>
</html>
```

terrain.js - Javascript

This file contains all of the original version shown previously because that will be run as well as the converted version, to compare the two. The following code is in addition to that.

```
function waCalcHeight(){} // Will be used if successful

function run() // entry from HTML onload
{
  if (!WebAssembly || !WebAssembly.instantiateStreaming)
    return;

  try
  {
    const memory = new WebAssembly.Memory({initial: 1});
                                      // 1 page of 64kB
    const view = new DataView(memory.buffer);
```

```
    const NB = 8; // No of bytes per number, f64
    for (var i = 0; i < PROFILE.length; i++)
    {
       view.setFloat64(i * NB, PROFILE[i], true); // little endian
    }
    const AH0 = PROFILE.length * NB;  // f64 => 2048
    const BH0 = AH0 + AH.length * NB; // f64 => 2088
    for (i = 0; i < AH.length; i++)
    {
       view.setFloat64(AH0 + i * NB, AH[i], true);
       view.setFloat64(BH0 + i * NB, BH[i], true);
    }
    const importObject = {js: {mem: memory}};
    WebAssembly.instantiateStreaming(
       fetch("terrain.wasm"), importObject)
       .then((obj) =>
    { waCalcHeight = obj.instance.exports.getHeight;// sets func
       report();
    })
       .catch ((e) => { alert  (e.toString()); });
  }
  catch (e) { alert  (e.toString()); }
}
```

Unlike JS, WASM is very strict about number types. They cannot be mixed. After some experimenting I decided that `f64` (64-bit floating) would be appropriate for all of the data and `i32` (32-bit integer) for memory addressing/indexing.

Memory in pages of 64kB can be shared between JS and WASM so the first thing the code above does is to create such a `WebAssembly.Memory` object and load it with the `PROFILE` and the parameter arrays `AH` & `BH` that `calcHeight()` needs. They are all loaded as 64-bit floats so that WASM can read them as `f64`. This memory is just one block so we need to note where `AH` and `BH` start, after `PROFILE`. Another point to be aware of is that the memory is addressed as bytes and 64-bit values occupy 8 bytes. WASM is always little-endian, which means the least significant byte of such numbers comes first (not all machines do it that way round).

Type `DataView` and its methods are standard JS you can read about on MDN.

Having put into the memory object the data WASM will need we create a literal object of the form `{js: {mem:memory}}` (a literal object within a literal object) and pass that as one parameter of the method `instantiateStreaming` of `WebAssembly`. The other parameter fetches our assembled .wasm file. A JS `Promise` is returned and the `.then` part of that makes any functions exported from the WASM available for use as Javascript functions. The significant line is

```
    waCalcHeight = obj.instance.exports.calcHeight;
```

in which `calcHeight` is the name of a function as defined within the WASM file, to be shown next. After that assignment the function `waCalcHeight()` is available for use, expecting 2 numbers `x` and `y` as parameters. In my example the function `report()` then goes on to do the tests and report its findings.

terrain.wat - the Web Assembly source text

This file will be assembled to make terrain.wasm, invoked as shown above. The assembling is very easy in VSC with the extension described earlier. It is only necessary to right click on this file and select "Save as Web Assembly binary file". You then get the normal file saving dialogue and the file name should end in .wasm.

This saving operation can produce error messages if the syntax of the .wat file is wrong. Such messages are not very explicit. They refer to line numbers in the assembled file which is not very helpful. Hence the need for a great deal of experimentation to get it right.

It is essential to know that WASM operates as a stack machine. Any operator works on the values most recently put on the stack, which pushes down. A unary operator simply changes whatever is on the top of the stack. A binary operator, such as add or multiply combines the top two values to leave only the result on the top of the stack.

Back in the 1970s, when microprocessors were new, Hewlett Packard made the best hand-held calculators and they were designed on this stack principle. We used to program them in Reverse Polish Notation (RPN). It is interesting to see that this is still a very useful technique for making really efficient programs.

The following listing is my complete version that works. Note that end of line comments in WASM begin with double semicolons, `;;`, and block comments are between `(;` and `;)`.

```
(module
  (import "js" "mem" (memory 1))
  ;;(global $PROF0 (mut i32) (i32.const 0))
  ;; byte address of start of PROFILE - but decided not needed
  (global $BITMASK (mut i32) (i32.const 255))
  ;; PROFILE.length - 1
  (global $AH0 (mut i32) (i32.const 2048))
  ;; byte address of start of AH
  (global $BH0 (mut i32) (i32.const 2088))
  ;; byte address of start of BH
  (global $R128 (mut f32) (f32.const 0.0078125));; 1 / 128
  (func $calcHeight (param $x f64)(param $y f64)  (result f64)
    (local $ht f64) ;; ht = 0
    (local $i8 i32) ;; i8 = 0 Loop counter +8 for addressing
    (local $j f64)(local $jint_i i32)(local $jint_f f64)
    (local $jfrac f64)
    (loop $loop
      ;; j = (AH[i] * x + BH[i] * y) * R128
      local.get $i8
      global.get $AH0
      i32.add
      f64.load ;; AH[i]
      local.get $x
      f64.mul ;; AH[i] * x
      local.get $i8
      global.get $BH0
      i32.add
      f64.load ;; BH[i]
      local.get $y
      f64.mul ;; BH[i] * y
      f64.add
      global.get $R128
      f64.mul ;; = j
      local.tee $j ;; set and keep on stack

      ;; jint = floor(j)
      f64.floor ;; jint = floor(j)
```

```
        local.tee $jint_f ;; set and keep on stack
        i64.trunc_f64_s ;; to signed i32
        i32.wrap_i64
        local.set $jint_i ;; set and pop from stack

        ;; jfrac = j - jint
        local.get $j
        local.get $jint_f
        f64.sub ;; jfrac = j - jint
        local.set $jfrac ;; set and pop from stack

        ;; prof0 = PROFILE[jint & BITMASK];
        local.get $jint_i
        global.get $BITMASK
        i32.and
        i32.const 3
        i32.shl ;; x 8 for byte address
;;        global.get $PROF0
;;        i32.add ;; Assume PROF0 == 0
        f64.load
        f64.const 1
        local.get $jfrac
        f64.sub
        f64.mul ;; prof0 * (1 - jfrac)

        ;; prof1 = PROFILE[(jint + 1) & BITMASK];
        local.get $jint_i
        i32.const 1
        i32.add ;; jint + 1
        global.get $BITMASK
        i32.and
        i32.const 3
        i32.shl ;; x 8 for byte address
;;        global.get $PROF0
;;        i32.add ;; Assume PROF0 == 0
        f64.load
        local.get $jfrac
        f64.mul ;; prof1 * jfrac

        ;; ht += prof0 * (1 - jfrac) + prof1 * jfrac interpolate
        f64.add
        local.get $ht
        f64.add
        local.set $ht

        ;; loop: add 8 to $i8
        local.get $i8
        i32.const 8
        i32.add
        local.tee $i8 ;; set and keep on stack

        ;; if $i8 is less than 40 branch to loop
        i32.const 40
        i32.lt_s ;; less than, signed
        br_if $loop
    )
```

```
      local.get $ht ;; result, f64
   )
   (export "calcHeight" (func $calcHeight))
)
(; original JS reminder:
function calcHeight(x, y)
{ var ht = 0;
   for (var i = 0; i < 5; i++)
   { var j = (AH[i] * x + BH[i] * y) * R128;
      var jint = Math.floor (j);
      var jfrac = j - jint;
      var prof0 = PROFILE[jint & BITMASK];
      var prof1 = PROFILE[(jint + 1) & BITMASK];
      ht += prof0 * (1 - jfrac) + prof1 * jfrac; // interpolate
   }
   return ht;
};)
```

Some points worth noting in that:

- I made all of the global declarations mutable (keyword **mut**) because I got syntax errors otherwise. I suspect that may be a problem with the VSC extension.

- In building index values for the memory I use **i32.shl** which shifts the top stack value left by the number of bits specified by the previous value. In this program the shift is 3 and that is a fast way of multiplying integers by 8. Remember that the values in the memory block are 64-bit (8-byte) floats.

- I noted earlier that the variable **jint** is needed as an integer for indexing **PROFILE** but also as a float for multiplying in the interpolation. As can be seen on the previous page this means using two different ways of converting to integer, **floor** and **trunc**. Unfortunately these two operations behave differently when the operand is negative; something to keep in mind but the code above works.

Next steps - calculating terrain kind and existence of point features

I proceeded to convert another part of my terrain generator which determines vegetation types and whether there is one of the scattered point features (boulder, pond, etc) at each (x, y) position, as on the full map shown here. This is more complicated and it makes more use of bitwise operators. I also had to find out how to write if-then-else in WASM.

At the end I will show performance data and how I deal with browsers that do not perform.
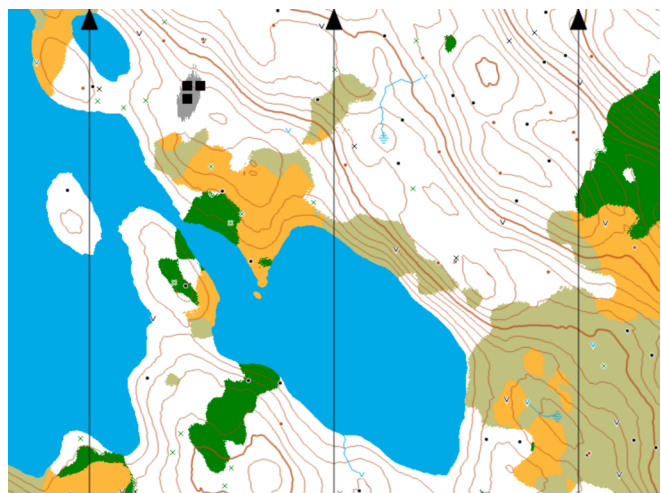
My function **calcTerra(x, y)** returns two integers which are indexes into lists of terrain kinds and feature kinds. To make the function return a single value (rather than getting into deeper WASM water) I multiplied one by 256 and added the other. This effectively makes a 2-byte value that can easily be separated. I changed my

original JS to do the same thing, to make comparisons fair.

Another function is called several times by `calcTerra()`. It is called `calcProf()`. It is similar to `calcHeight()` but simpler because it assumes integer x and y parameters and so does not do the interpolation. It uses other parameter arrays, not the `AH` and `BH` that were used by `calcHeight()`.

Following is the original JS (cut down a bit from my full terrain generator which can do other things such as allow for moving objects). Note that several new constants are needed, all to be stored in the shared memory block with addresses calculated as before.

```
const FEATURES = { NONE:0, MINE:1, BOULDER:3, ROOT:4,
                   WATERHOLE:5, KNOLL:6, X:7, CONE:8};

const TERRAINS = { TOWN:21, GRASS:22, MOOR:23, WOOD:24,
                   THICKET:25};

const A1 = [-43, -43, -56, 31, 4];
const B1 = [-3, -12, 22, 2, 32];
const A2 = [-24, -25, 60, 10, - 30];
const B2 = [15, -54, -34, -51, -43];
const A3 = [-51, -62, -58, -64, 33];
const B3 = [-44, 20, 27, -64, -44];
const PI1000 = Math.PI * 1000;
const PI10000 = Math.PI * 10000;

function calcProf(A, x, B, y)// A = Ax or Bx arrays
{
  var p = 0;
  for (var i = 0; i < 5; i++)
  {
    p += PROFILE[
         Math.floor(((A[i] * x + B[i] * y) >> 7)) & BITMASK];
  }
  return p;
}

function calcTerra(x, y)
{ var feature = FEATURES.NONE;
  var xr = Math.round (x), yr = Math.round (y);
  var xryr = xr * yr;
  var a = calcProf (B2, xr, A3, yr);
  var f = Math.round (a * xryr * R128) & 0xfff;

  if (4 === f)
  {
    var xyff = xryr & 0xff;
    if (xyff < 32) feature = FEATURES.MINE;
    else if (xyff < 128) feature = FEATURES.BOULDER;
    else if (xyff < 160) feature = FEATURES.WATERHOLE;
    else if (xyff < 200) feature = FEATURES.KNOLL;
    else feature = FEATURES.ROOT;
  }
  else if (8 === f
            && (Math.round (PI1000 * xryr) & 0xff) < 12)
  { feature = FEATURES.X; }
```

```
    else if (16 === f
              && (Math.round (PI10000 * xryr) & 0xff) < 8)
    { feature = FEATURES.CONE; }

    var terrain = TERRAINS.WOOD;
    a = calcProf (A1, x, B1, y);

    if (120 > a)
    {
      terrain = TERRAINS.TOWN;
      feature = FEATURES.NONE
    }
    else
    {
      a = calcProf (A2, x, B2, y);
      var b = calcProf (A3, x, B3, y);

      if (255 > a)
      { if (255 > b) terrain = TERRAINS.GRASS;
        else terrain = TERRAINS.MOOR;
      }
      else if (200 > b) terrain = TERRAINS.THICKET;
    }

    return feature + terrain * 256; // bytes 0 and 1
}
```

And here is the additional content for the ,wat file

```
;; Byte addresses of starts of data values:
(global $A10 (mut i32) (i32.const 2128))
(global $B10 (mut i32) (i32.const 2168))
(global $A20 (mut i32) (i32.const 2208))
(global $B20 (mut i32) (i32.const 2248))
(global $A30 (mut i32) (i32.const 2288))
(global $B30 (mut i32) (i32.const 2328))
(global $PI1000 (mut i32) (i32.const 2368))
(global $PI10000 (mut i32) (i32.const 2376))

(func $calcProf (param $a0 i32)(param $x f64)
  (param $b0 i32)(param $y f64) (result f64)
  ;; $a0 and $b0 are array start addresses in global memory
  (local $p f64) ;; p = 0
  (local $i8 i32) ;; i8 = 0 Loop counter +8 for addressing
  (loop $loop
    local.get $i8
    local.get $a0
    i32.add
    f64.load ;; A[i]
    local.get $x
    f64.mul ;; A[i] * x
    local.get $i8
    local.get $b0
    i32.add
    f64.load ;; B[i]
    local.get $y
    f64.mul ;; B[i] * y
```

```
        f64.add ;; (A[i] * x + B[i] * y)
        i64.trunc_f64_s
        i64.const 7
        i64.shr_s ;; (A[i] * x + B[i] * y) >> 7
        i32.wrap_i64
        global.get $BITMASK
        i32.and
        i32.const 3
        i32.shl ;; x 8 for byte address
;;        global.get $PROF0
;;        i32.add ;; Assume PROF0 == 0
        f64.load
        local.get $p
        f64.add
        local.set $p

        ;; loop: add 8 to $i8
        local.get $i8
        i32.const 8
        i32.add
        local.tee $i8 ;; set and keep on stack

        ;; if $i8 is less than 40 branch to loop
        i32.const 40
        i32.lt_s ;; less than, signed
        br_if $loop
      )

    local.get $p ;; result, f64
  )

  (func $calcTerra (param $x f64)(param $y f64) (result i32)
    (local $xr f64)(local $yr f64)(local $xryr f64)
    (local $f i32)(local $xyff i32)
    (local $feature i32)(local $terrain i32)
    local.get $x
    f64.nearest ;; round to nearest integer
    local.tee $xr ;; save and keep on stack
    local.get $y
    f64.nearest
    local.tee $yr
    f64.mul
    local.tee $xryr
    (call $calcProf(global.get $B20)(local.get $xr)
                   (global.get $A30)(local.get $yr))
    f64.mul
    global.get $R128
    f64.mul
    f64.nearest
    i64.trunc_f64_s
    i64.const 0xfff
    i64.and ;; f = Math.round (a * xryr * R128) & 0xfff;
    i32.wrap_i64
    local.tee $f
    i32.const 4
    i32.eq ;; pops 2 & leaves boolean: 1 for then, 0 for else
    (if
```

```wasm
(then ;; f === 4
  local.get $xryr
  f64.nearest
  i64.trunc_f64_s
  i64.const 0xff
  i64.and
  i32.wrap_i64
  local.tee $xyff
  i32.const 32
  i32.lt_s
  (if
    (then ;; xyff < 32
      i32.const 1 ;; MINE
      local.set $feature
    )
    (else
      local.get $xyff
      i32.const 128
      i32.lt_s
      (if
        (then ;; xyff < 128
          i32.const 3 ;; BOULDER
          local.set $feature
        )
        (else
          local.get $xyff
          i32.const 160
          i32.lt_s
          (if
            (then ;; xyff < 160
              i32.const 5 ;; WATERHOLE
              local.set $feature
            )
            (else
              local.get $xyff
              i32.const 200
              i32.lt_s
              (if
                (then ;; xyff < 200
                  i32.const 6 ;; KNOLL
                  local.set $feature
                )
                (else
                  i32.const 4 ;; ROOT
                  local.set $feature
                )
              )
            )
          )
        )
      )
    )
  )
)
(else ;; f !== 4
  local.get $f
  i32.const 8
```

```
          i32.eq
          (if
            (then
              global.get $PI1000
              f64.load
              local.get $xryr
              f64.mul
              f64.nearest
              i64.trunc_f64_s
              i64.const 0xff
              i64.and ;; Math.round (PI1000 * xryr) & 0xff
              i64.const 12
              i64.lt_s
              (if
                (then
                  i32.const 7 ;; X
                  local.set $feature
                )
              )
            )
            (else
              local.get $f
              i32.const 16
              i32.eq
              (if
                (then
                  global.get $PI10000
                  f64.load
                  local.get $xryr
                  f64.mul
                  f64.nearest
                  i64.trunc_f64_s
                  i64.const 0xff
                  i64.and ;; Math.round (PI1000 * xryr) & 0xff
                  i64.const 8
                  i64.lt_s
                  (if
                    (then
                      i32.const 8 ;; CONE
                      local.set $feature
                    )
                  )
                )
              )
            )
          )
        )
        i32.const 24 ;; WOOD
        local.set $terrain
        (call $calcProf(global.get $A10)(local.get $x)
                  (global.get $B10)(local.get $y))
        f64.const 120
        f64.lt
        (if
          (then
            i32.const 0
```

```wasm
                    local.set $feature ;; NONE
                    i32.const 21
                    local.set $terrain ;; TOWN
                )
                (else
                    (call $calcProf(global.get $A20)(local.get $x)
                                    (global.get $B20)(local.get $y))
                    f64.const 255
                    f64.lt
                    (if
                        (then
                            (call $calcProf(global.get $A30)
                             (local.get $x)(global.get $B30)(local.get $y))
                            f64.const 255
                            f64.lt
                            (if
                                (then
                                    i32.const 22
                                    local.set $terrain ;; GRASS
                                )
                                (else
                                    i32.const 23
                                    local.set $terrain ;; MOOR
                                )
                            )
                        )
                        (else
                            (call $calcProf(global.get $A30)
                             (local.get $x)(global.get $B30)(local.get $y))
                            f64.const 200
                            f64.lt
                            (if
                                (then
                                    i32.const 25
                                    local.set $terrain ;; THICKET
                                )
                            )
                        )
                    )
                )
            )
            local.get $terrain
            i32.const 256
            i32.mul
            local.get $feature
            i32.add
        )
        (export "calcProf" (func $calcProf))
        (export "calcTerra" (func $calcTerra))
```

Conclusions

It does all work but so far only Firefox has given me any speed improvement: a good factor of about 3 for both `calcHeight()` and `calcTerra()`. But as the tables on the next page show, the story is very different in MS Edge.

This is very disappointing of course. It probably means that all Chromium-based browsers do the same.

I also tested in the Samsung browser on my phone and found that times for JS and WASM were about equal.

However, I have included the WASM file in my latest version of [The Forest](). I have made it so that during start-up my program does a timing test between the JS and WASM versions. If the WASM version takes less than half the time taken by JS then I switch to using the WASM versions of `calcHeight()` and `calcTerra()`.

4 million calls each time: **FIREFOX**

	n	mean (ms)	std.dev.
calcHeight(x, y)	100	282.09	13.47
waCalcHeight(x, y)	100	90.83	8.47
calcTerra(x, y)	100	399.75	24.42
waCalcTerra(x, y)	100	125.22	6.94

4 million calls each time: **MS EDGE**

	n	mean (ms)	std.dev.
calcHeight(x, y)	100	100.23	8.02
waCalcHeight(x, y)	100	96.4	7.19
calcTerra(x, y)	100	217.52	17.27
waCalcTerra(x, y)	100	244.11	11.11

In Firefox the console log then says "Using WASM". I would be very interested to know whether that happens in any other browsers. My hope is that they will all catch up in due course.

My test program is available to use at grelf.net/wasm and terrain.wat is there too.

Graham Relf

April 2025