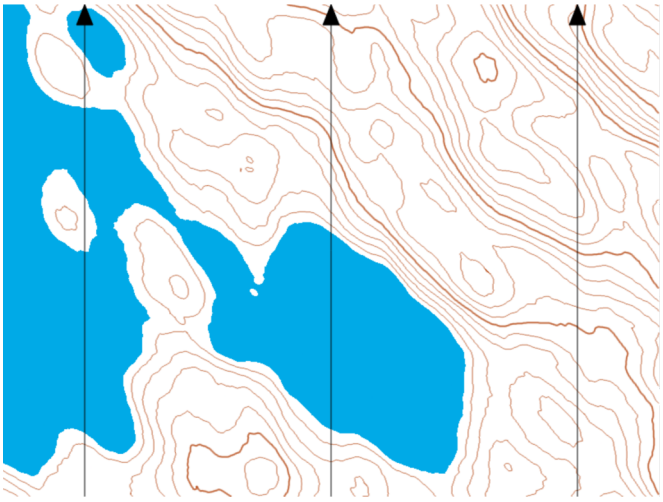
Converting terrain generation functions to Web Assembly

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. This is still work in progress but the first function, the height calculator, has been successfully converted with a significant speed gain.

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 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 is 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 will show 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.

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

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.

The second table 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 `f32` in WASM, but for indexing it needs to be a true integer of type `i32` in WASM. So two versions of `jint` will be needed.

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(){ alert("WebAssembly not available"); }

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

  try
  {
    const memory = new WebAssembly.Memory({initial: 1});
                                          // 1 page of 64kB
    const view = new DataView(memory.buffer);
    const NB = 4; // No of bytes per number, f32

    for (var i = 0; i < PROFILE.length; i++)
    {
      view.setFloat32(i * NB, PROFILE[i], true); // little endian
    }
```

```
    const AH0 = PROFILE.length * NB; // f32 => 1024
    const BH0 = AH0 + AH.length * NB; // f32 => 1044

    for (i = 0; i < AH.length; i++)
    {
      view.setFloat32(AH0 + i * NB, AH[i], true);
      view.setFloat32(BH0 + i * NB, BH[i], true);
    }

    const importObject = {js: {mem: memory}};

    WebAssembly.instantiateStreaming(
        fetch("terrain.wasm"), importObject)
        .then((obj) =>
      { waCalcHeight = obj.instance.exports.getHeight;
        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 `f32` (32-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 32-bit floats so that WASM can read them as `f32`. 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 32-bit values occupy 4 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 the data WASM will need into the memory object 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.getHeight;
```

in which `getHeight` 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, to be invoked as shown above. The assembling is done very simply in VSC with the extension described earlier. It is only necessary to right click on this file and select the option “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 programmed 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 first 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
  (global $BITMASK (mut i32) (i32.const 255))
                                   ;; PROFILE.length - 1
  (global $AH0 (mut i32) (i32.const 1024))
                                ;; byte address of start of AH
  (global $BH0 (mut i32) (i32.const 1044))
                                ;; byte address of start of BH
  (global $R128 (mut f32) (f32.const 0.0078125));; 1 / 128

  (func $getHeight (param $x f32) (param $y f32) (result f32)
    (local $ht f32) ;; ht = 0
    (local $i i32) ;; i = 0
    (local $j f32)
    (local $jint_i i32)(local $jint_f f32)(local $jfrac f32)

    (loop $loop
      ;; j = (AH[i] * x + BH[i] * y) * R128
      local.get $i
      i32.const 2
      i32.shl ;; x 4 to get byte index
      global.get $AH0
      i32.add
      f32.load ;; AH[i]
      local.get $x
      f32.mul ;; AH[i] * x
      local.get $i
      i32.const 2
      i32.shl ;; x 4 to get byte index
      global.get $BH0
      i32.add
      f32.load ;; BH[i]
      local.get $y
      f32.mul ;; BH[i] * y
      f32.add
```

```
global.get $R128
f32.mul ;; = j
local.tee $j ;; set and keep on stack

;; jint = floor(j)
f32.floor ;; jint = floor(j)
local.tee $jint_f ;; set and keep on stack
i32.trunc_f32_s ;; to signed i32
local.set $jint_i ;; set and pop from stack

;; jfrac = j - jint
local.get $j
local.get $jint_f
f32.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 2
i32.shl ;; x 4 for byte address
global.get $PROF0
i32.add
f32.load
f32.const 1
local.get $jfrac
f32.sub
f32.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 2
i32.shl ;; x 4 for byte address
global.get $PROF0
i32.add
f32.load
local.get $jfrac
f32.mul ;; prof1 * jfrac

;; ht += prof0 * (1 - jfrac) + prof1 * jfrac // interp
f32.add
local.get $ht
f32.add
local.set $ht

;; loop: add one to $i
local.get $i
i32.const 1
i32.add
local.tee $i ;; set and keep on stack

;; if $i is less than 5 branch to loop
```

```
      i32.const 5
      i32.lt_s ;; less than, signed
      br_if $loop
    )
    local.get $ht ;; result, f32
  )
  (export "getHeight" (func $getHeight))
)
(; 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:

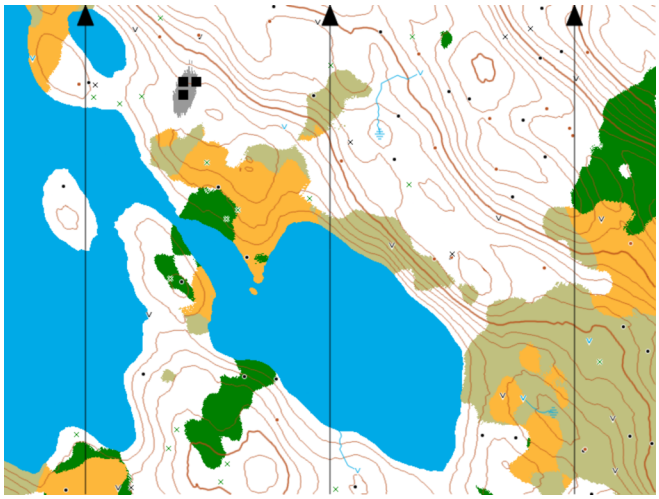
- I made all of the global declarations mutable (keyword `mut`) because I got syntax errors otherwise. I suspect that may be a problem with the VSC extension.

- 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 2 and that is a fast way of multiplying integers by 4. Remember that the values in the memory block are 32-bit (4-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. I suspect this is the reason for the very small discrepancy between JS and WA height results. I will investigate this further.

Next steps

I plan to go on 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 will be more complicated and it will make more use of bitwise operators.



I leave report.js as an exercise. It is just plain Javascript.

Graham Relf, March 2025