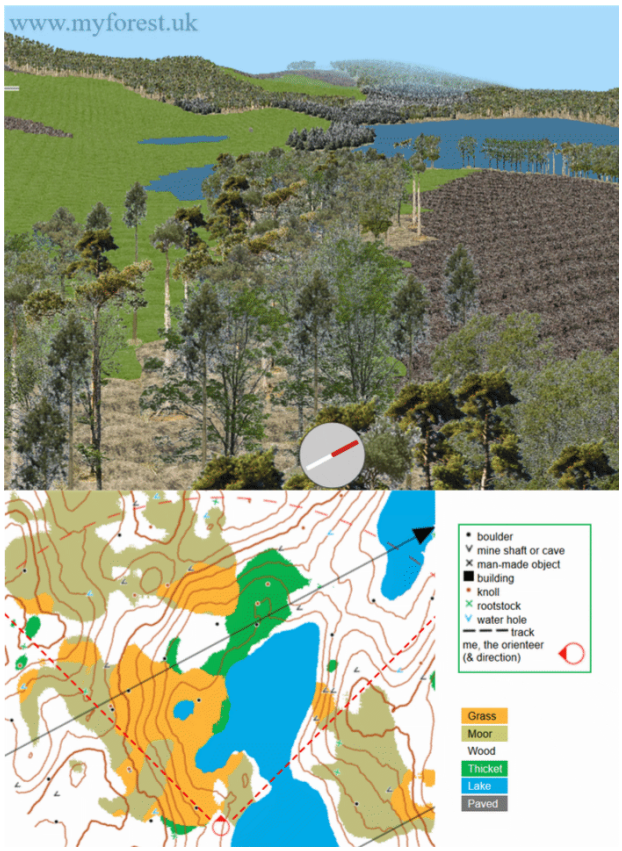
How to make limitless terrain for games in real time



The Relf Terrain Generator

This is a description for game programmers of how to generate essentially limitless terrain in real time.The terrain is not stored permanently as data. Instead it is generated by mathematical functions as a player moves around.

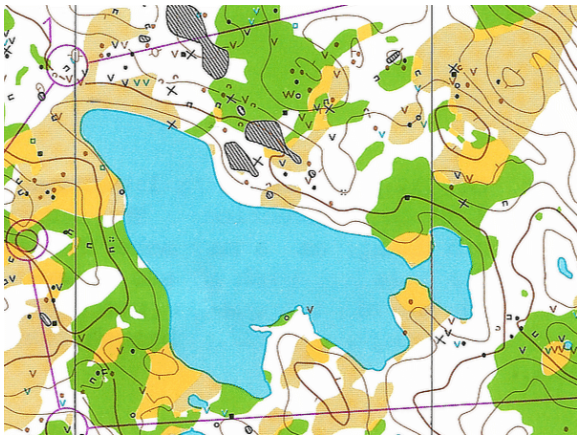
The result is not simply a height map. It includes various types of vegetation, towns, water in various forms (sea, lakes, streams) and vast numbers of objects positioned arbitrarily. Objects may also be placed deliberately and moved around as necessary. Underground there are dungeons, referred to as mines here, with mineshafts or caves for entering them.

Unlike many other terrain generators there are no chunks, tiles or voxels and players can move smoothly to any positions, not limited to whole number grid coordinates.

The terrain is essentially random but repeatably so. While I, as programmer, had to explore to find things myself, everything is the same the next time the generator runs (but it can be made completely random, different every time, as will be shown later).

The algorithm was originally developed more than 40 years ago for a simulation of the sport of orienteering, the latest version of which may be seen and explored at ***grelf.itch.io/forest*** or at ***myforest.uk*** (**The Forest**, completely free). The scene and corresponding map shown above are from The Forest running in Firefox browser.

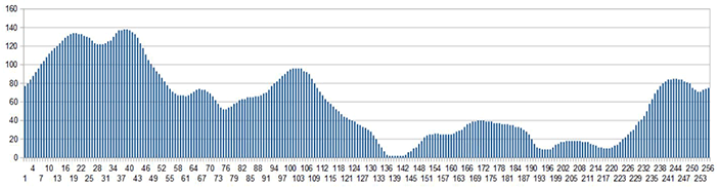
I first developed these techniques in the early 1980s when I had rather primitive versions of **The Forest** published for TRS-80, Sinclair ZX Spectrum, and BBC Microcomputers. In those days the program was written in assembly code but the present version (at the URLs on the previous page) is programmed in JavaScript to run in any HTML5 graphical browser, even on phones. The algorithm is fast. I also have versions written in Java (see github.com/grelf-net/forest for the sources) and in C++.



The map on the right here is a tiny fragment as generated in 1983.

Ground shape: heights

The starting point is a rather arbitrary 1-dimensional profile of length 256 for which the data are stored as a literal array of integer values. Charted in a spreadsheet it looks like this.



The profile is a periodic structure: its last value and slope are very similar to its start. The array is indexed by the remainder of some number (`p`, say) when divided by 256 to get the height of the terrain. The number `p` is a function of the x, y coordinates of a point on the ground (x eastwards, y northwards).

```
height = profile [p % 256]
```

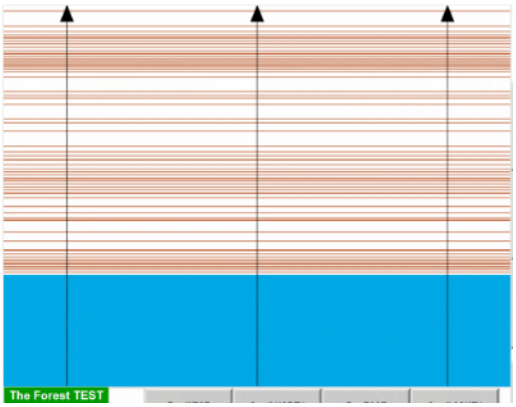
Performance is important here, so we do not want to be doing a modulus operation because that involves division. Instead a bit-wise AND is done:

```
height = profile [p & 0xff]
```

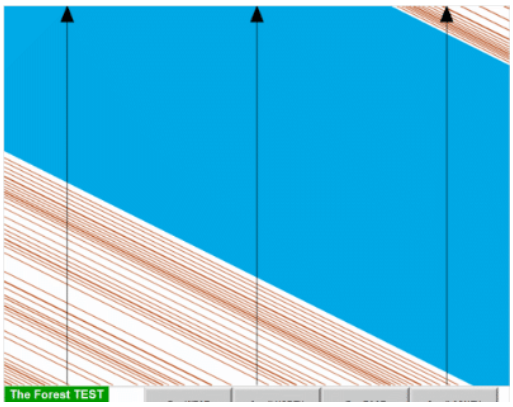
`0xff` is hexadecimal for decimal 255; written this way to emphasise what is done.

(If you look at the full JavaScript source you will see that a division by 128 is also used in the height calculation. For speed, that is replaced by a multiplication by a constant set at the start: `RECIP128 = 1 / 128;`)

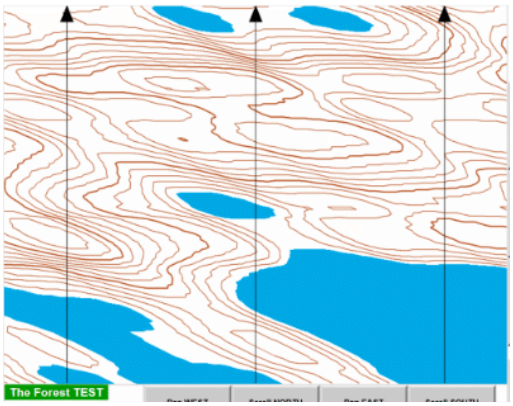
Recall that the number `p` above is a function of position. If `p = 27y` (no x-dependence), and the heights are multiplied by 5, and then heights below 204 are deemed to be lake, we get the very boring contour map shown here:



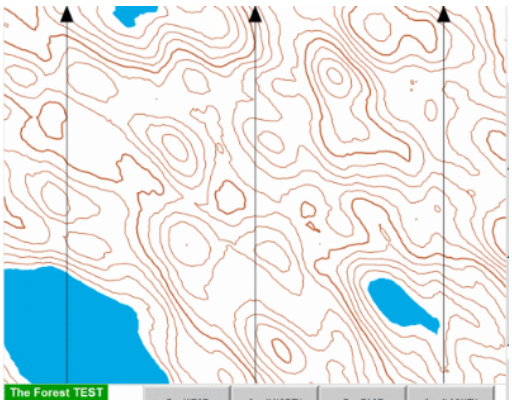
If instead `p = 13x + 26y` we get the equally boring contour map shown here, with the same pattern just lying at an angle:



But if we average those two patterns we begin to get something much more interesting:



In **The Forest** there is an average of 5 such patterns in different directions, with this result:



The interesting terrain shape is therefore calculated by something essentially like this (but using a for-loop):

```
height = sumOverI
  (profile [(A [i] * x + B [i] * y) & 0xff]) * RECIP128;
```

where `A` and `B` are arrays of parameters, each of length 5, declared as constants at the start of the program.

Notice that if the profile array were much longer but its length was still a power of 2 so that a bit-wise AND was still possible, it would make no difference to performance. Longer arrays could make for greater variety in the terrain: there can be flatter plains and sharper mountainous areas.

It is also important to note that although the initial profile repeats after 256 metres*, the fact that we are adding versions at various angles means that the resulting terrain is unlikely ever to repeat (until we run out of numerical precision of course).

* 1 pixel on the map = 1 metre on the ground = 1 element difference in the profile array.

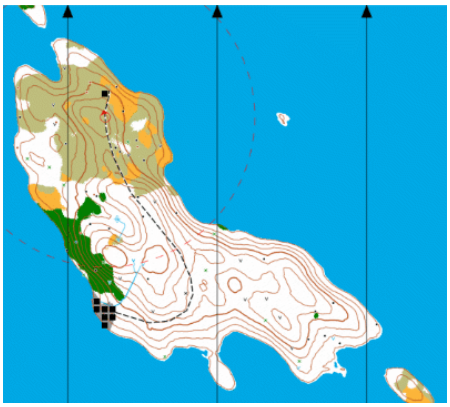
Height is calculated using the full floating point values for x and y, because the observer is not necessarily standing exactly on a whole-metre x-y grid and we want height to be as smooth a function as possible. The floating-point value of `p` interpolates linearly between adjacent elements in the profile array. Vegetation and point features (in following sections) use only rounded integer versions of x and y, for speed.

In Javascript the height calculator looks like this:

```
function calcHeight (x, y)
{
  var ht = 0;
  for (var i = 0; i < 5; i++)
  {
    var j = (AH [i] * x + BH [i] * y) * RECIP128;
    var jint = Math.floor (j);
    var jfrac = j - jint;
    var prof0 = profile [jint & 0xff];
    var prof1 = profile [(jint + 1) & 0xff];
    ht += prof0 + jfrac * (prof1 - prof0); // interpolate
  }
  return ht;
}
```

The algorithm for drawing the contours was first described in 1987 in Byte magazine by Paul Bourke. His description can be found at paulbourke.net/papers/conrec/ .
I have re-implemented the algorithm in JavaScript (and now also in Java and C++) in a form that is most efficient for my map. It performs well. Every fifth contour is thicker, as is standard for orienteering maps, to help interpretation. Streams on the map (see below) also help to see which direction is downhill.

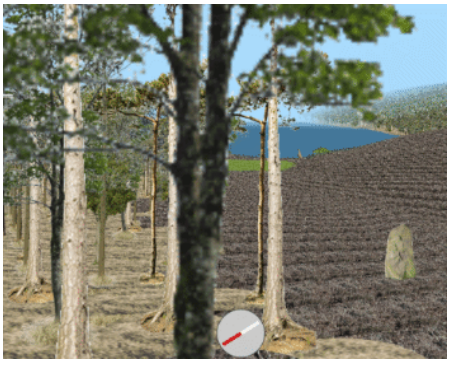
Lakes or islands



If the height is below a certain fixed value (204 in **The Forest**) there is deemed to be a lake. A rain feature was added in June 2018: the lake height slowly increases when it rains, reducing back to the original fixed value when the sun shines. (It is also possible for explorers to drain the lakes in order to find something at the bottom of one of them, if they can work out how and where to do this.)

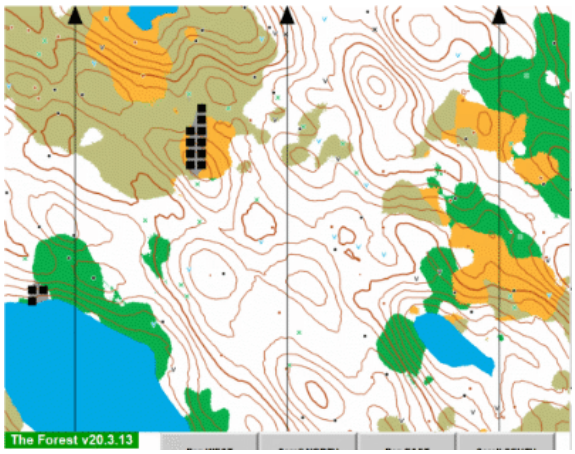
A higher lake level also makes the ground break up into islands, which can be desirable for some games.

Terrain kinds (vegetation etc)



A similar summing of profile patterns is done for determining terrain types (thicket, moor, town, etc), using the same profile but different parameter arrays a and b. If the result lies within a certain range of values, the terrain is of a certain type. Because the parameter arrays are different the patches of vegetation do not follow the contour shapes but they have similarly irregular shapes.

In **The Forest** there are only 6 possible terrain types, as in the legend displayed in the program (and shown here).



The map is drawn as closely as possible to international orienteering standards (see https://orienteering.sport/iof/mapping/). White areas are runnable woodland (mature trees). Green is thicket, slower to penetrate or even best avoided. Yellow is open grassland but ochre is moorland which is slower to run across (all taken into account in the program). Arrays of black squares represent buildings in "towns".

The program represents the terrain types as

```
const TERRAINS =
       { LAKE:0, TOWN:1, GRASS:2, MOOR:3, WOOD:4, THICKET:5 };
```

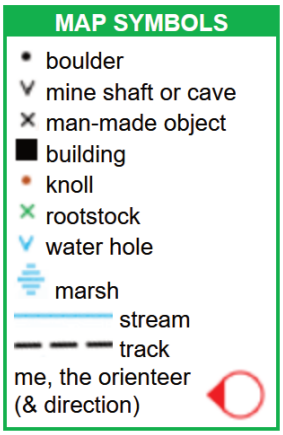
(Javascript does not have **enum**s like Java but this is doing something similar.)

```
function calcProf (a, x, b, y) // for terrain type at (x, y)
{
  var p = 0;
  for (var i = 0; i < 5; i++)
  {
    p += profile [Math.floor((a[i] * x + b[i] * y) >> 7) & 0xff];
  }
  // >> 7 is a (probably faster) way of dividing by 128
  return p;
};
```

Essentially this is the same as the height calculation function but without linear interpolation between elements of the profile array. With different parameter arrays it is used like this, after establishing that the height is above lake level:

```
var a = calcProf (A1, x, B1, y);

if (a < 120) return TOWN;

var b = calcProf (A2, x, B2, y);
var c = calcProf (A3, x, B3, y);

if (b < 255)
{
  if (c < 255) return GRASS; else return MOOR;
}
else if (c < 200) return THICKET; else return WOOD;
```

Point features on the ground



There are a number of different types of objects scattered throughout the terrain and located at specific (x, y) points. They include boulders, ponds, upturned tree roots, mine shafts (leading to systems of underground mines, of which more later) and several other things. Whether an object appears at a given point and what kind it is, are also determined from the terrain profile. They have to be sparsely arranged of course.

```
const FEATURES = {NONE:0, MINE:1, BOULDER:2,
    ROOT:3, WATERHOLE:4, KNOLL:5, X:6, CONE:7};
```

In this list X means a man-made object, such as a sculpture. CONE is one of those traffic cones, which can be moved around and are therefore not shown on the map. In the program there are several other kinds of features too.

It is vital that the determination of whether a feature is present must use the rounded whole-number coordinates x and y, otherwise there would be a near-infinity of possibilities in every metre of ground.

To determine what is present on the ground at a given (x, y) position, the program calls function `terra(x, y)` which returns an object with properties `height` (Number), `terrain` (TERRAINS.Number), `feature` (FEATURES.Number), `code` (String, 2 letters). In a lake there is also `depth` (Number).

(The code would be for an orienteering flag if there is a feature and it has a flag.)

The essence of this part of the program will now be shown in more detail, which you may wish to skip.

The sequence inside `terra(x,y)` is

- calculate height

- determine whether there is a placed terrain type (road, path or stream) or a movable feature here (see later sections), if so return {height, terrain, feature}

- if height is below lake height return {lake_height, lake, none, depth}

- determine whether there is a fixed feature here (and if so, orienteering code)

- determine the terrain type

- return {height, terrain, feature, code}

Feature determination works like this. Features must be sparsely placed on the ground and so the conditions shown here are only occasionally met.

```
xr = Math.round (x);
yr = Math.round (y);
xryr = xr * yr;
// Usual profile calculation but swapped A/B arrays
// to re-use them:
a = calcProf (B2, xr, A3, yr);
f = Math.round (a * xryr * RECIP128) & 0xfff;
```

```
if (4 === f)
{
  xyff = xryr & 0xff;
  if (xyff < 32) feature = MINE;
  else if (xyff < 128) feature = BOULDER;
  else if (xyff < 160) feature = POND;
  else if (xyff < 200) feature = KNOLL;
  else feature = ROOT;
}
else
if (8 === f && (Math.round (PI1000 * xryr) & 0xff) < 4)
    feature = X; // Man-made
else
if (16 === f && (Math.round (PI10000 * xryr) & 0xff) < 8)
    feature = CONE;
```

in which an important constant declared at start-up is

```
PI10000 = Math.PI * 10000;
```

The code looks at bit patterns. To make the bits appear to be random, groups of bits are taken from a function that includes multiplication by π, mathematical pi which is irrational: it has an unpredictable sequence of digits (or, in base 2, bits). `PI10000` essentially shifts pi up so we are not looking at its first bits.

The testing of the value of `f` in the program above ensures that we only detect rarely occurring values and so the objects will be sparse across the ground.

There are several different images for each of the ground features. The next section describes how any particular version is selected at a particular position.

Placing a mix of trees

There are trees in **The Forest** of course. They are loaded into the program as image files (from my own photos, cut out and saved in PNG format to allow transparency around them; transparency is shown by the chequered pattern here). To avoid monotony there needed to be several different tree images. Suppose there are four (in fact there are more than that). At each position on the ground one of the four is to be chosen, seemingly at random. At that position, whenever the user looks at it, maybe after moving away and coming back, it must always be the same tree out of the four.

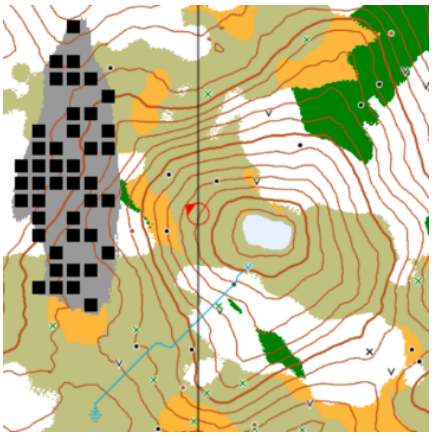
This is achieved by calculating a pair of bits (2 bits allows 4 possible values) from a function of the x and y coordinates of each point. Just as was done for point features, to make the bits appear to be random a pair of bits is taken from a function that includes multiplication by π. In JavaScript it looks like this:

```
treeSelector =
     Math.round (PI10000 * xr * yr) & 0x3;
```

A similar thing is done for the exact positions of trees within the square tiles that form the ground when a scene is displayed, so the trees do not lie always in dead straight rows. Once

again, if the explorer comes back to any place, the same tree will be seen in exactly the same slightly offset position.

Placing buildings in towns



One of the terrain types in **The Forest** is "town". A town is a paved area with 12m square buildings potentially placed at positions for which x and y are divisible by 16. So the buildings form a dense grid with alleyways 4m wide between them.

Building placement uses a similar technique to that of the trees above: a repeatable pseudo-random function determines whether a building exists at a given (x, y) position in a town. A sample of the map can be seen here.

That pale blue patch on the hill-top is snow (height above a certain level). I am also putting more variation into the buildings (again using repeatable pseudo-random seed values), as can be seen in this view from the red circle in the map above.

Enabling things to move

The terrain object has a property `placed` which is initialised simply as a `new Object()`. Recall that in JavaScript there are two quite different ways of accessing the properties of an object: either as `obj.propertyId` or as `obj ['propertyId']`, like an array indexed by a string. Behind the scenes objects are implemented as hash tables. The property names as strings are hash keys, directly forming an address within the table (content-addressable memory). The point is that given a key we can very rapidly determine whether there is an entry in the table, with no searching involved. That is exploited here, using `terrain.placed`. The key (or property name string) is formed from integer (rounded) x and y ground coordinates. They are separated by a comma, so we test whether `terrain.placed [x + ',' + y]` contains a particular value. This is very fast and it does not involve preallocating an array for all possible x and y (which would be impossible anyway because the coordinates are unbounded).

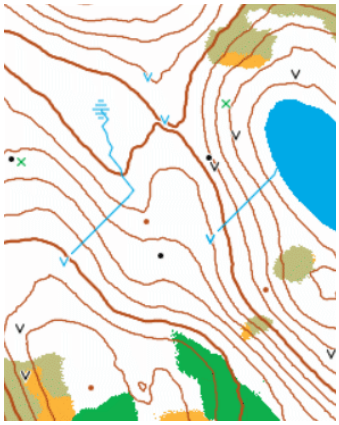
Note that the comma in the key string has a dual purpose. Firstly it causes conversion of integers x and y to strings for concatenation. Secondly it ensures that, for example, x = 123 and y = 45 does not produce the same key as x = 12 and y = 345.

This technique can be used to position any object at a required location or to indicate that an auto-generated feature has been moved, perhaps by the explorer. It does not want to be overdone because it goes against the initial principle of having auto-generated terrain. The technique has been used in **The Forest** to make helicopters stay where they land and not to be seen any longer where they started from. It has also been used to plant a self-moving mystery object on one of the tracks near the start.

Many programming languages have hash tables, often called maps, that can be used in a similar way.

Streams

Notice that the map is generated point by (x, y) point. There is no consideration of how neighbouring points may be related and therefore there are at first no linear features such as paths or streams. In fact it is much more difficult to generate a map containing such features. Only the vegetation boundaries or lake edges can be considered to be linear features. However I have recently (2020) added streams and paths.



Streams are created by testing the n x n neighbouring points around ponds (blue V on the map) to find whether any has a lower height than the pond. (n = 11 in The Forest.) If so, a stream extends to the lowest point in the neighbourhood. Repeat that process until either a lake or a hollow with no lower points is reached. The stream is described by an array of points, so its line can be drawn on the map.

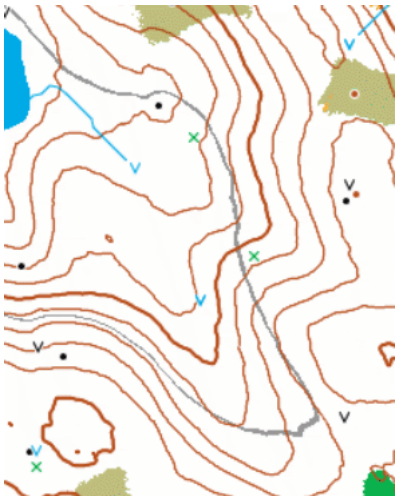
There were still programming difficulties with streams, so until version 25.1.1 of **The Forest** I considered them to be experimental. The main problem is how to deal with streams coming from ponds lying outside the displayed portion of the map or scene. How far outside should be searched for such possibilities?

For version 25.1.1 the problems were overcome by using a Javascript worker thread, scanning for streams in the background without delaying the main program. Following is more detail of how that works.

> *When a map is displayed the terrain around any pond on the map is explored as described above. If there is a stream its route is not only drawn on the map as a blue line but also (x, y) positions along the route are marked as "stream" by using the placement technique on the previous page (Enabling things to move) as long as there is not already something else marked there. That enables streams to be visible in the scene display. Otherwise it would not be possible to see streams reliably in scenes: the source of each stream may be outside the observer's visible range. The downside of this is that streams will not be visible if the observer has moved outside the area last viewed as a map. So it is necessary to scan an area around the observer whenever the observer moves. It was decided to set a maximum stream length of 300 metres so that only a certain range has to be scanned. A worker thread starts when the program is first loaded. Whenever the observer moves a message is sent to the worker giving the observer's new position. If the worker is not already scanning it begins a loop to cover the usual map area (currently 800 x 600m) plus 300m on all sides. Whenever it finds a*

stream it sends a message back to the main program giving the route data and whether the stream ends in marsh or lake. The main program is then able to mark the stream on the ground if it has not already been marked. There remained a potential problem though: could there be so many marked points that the program could run out of memory? To avoid this a count is kept of how many stream points have been placed. If that exceeds 300,000 all the stream placements are cleared and the process starts again. This number is big enough that an orienteer navigating a course will not see streams disappearing, however briefly.

Paths and roads

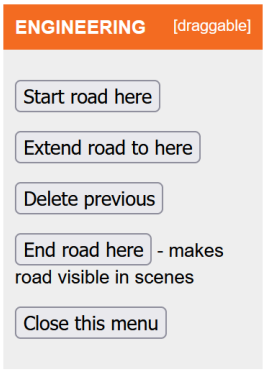


Automatically generating paths or roads is trickier. Unlike streams there is no rationality that can easily be automated for the directions they should take. In the real world they are created by people or animals repeatedly using a route they find useful. We cannot wait for a player to repeatedly follow a route in a game before turning it into a path.

The best thought I have had so far is that a line is an intersection between two surfaces. What surfaces do we have already? The contoured shape of the ground. So I have made experimental paths by putting paving slabs at every position that has the same height as another position some distance away, offset by a certain constant vector. If `DX` and `DY` are constants, the code is

```
if (terra (x,y).height === terra (x + DX, y + DY).height)
     placed [x + ',' + y] = TERRAINS.PATH;
```

This makes grey lines on the map but rather ragged.



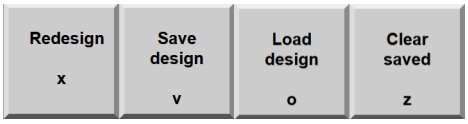
A recent addition to **The Forest** is the role of Engineer. When in this mode, clicking somewhere on the map produces a pop-up menu (really an HTML `<div>` that is usually hidden). This enables roads to be drawn as a series of straight line segments. Ending a road asks whether it is to be kept for subsequent runs of the program, in which case all saved roads are kept in the browser's local storage in JSON format.

These roads are shown on the map as dashed black lines.

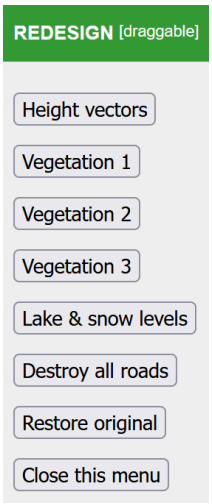
I have used this to make some permanent roads at the start.

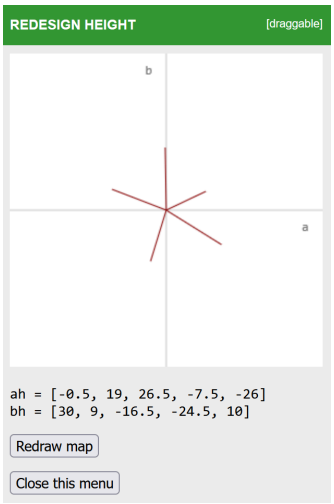
Change the terrain interactively

In a variant of **The Forest** called **My Terrain** (available free at grelf.itch.io/terrain) the engineering role has been extended to allow all of the parameters to be altered interactively by the user (as from version 24.11.1). As well as the road capabilities the Engineer sees some extra buttons:



The Redesign button produces a different menu, as shown at the left below.

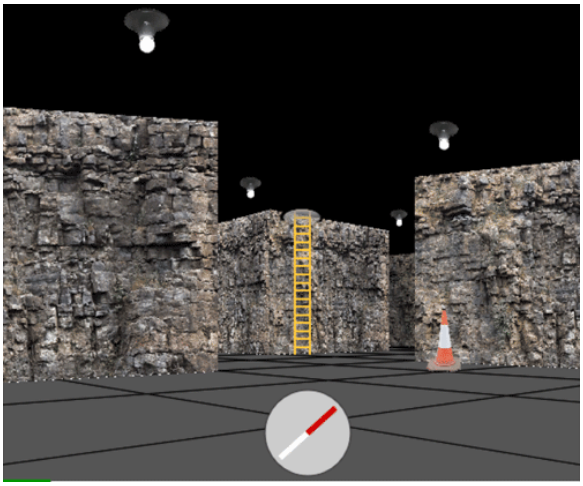




The second image here shows what the "Height vectors" option of the Redesign menu produces. Recall from earlier that the height is generated by adding the profile in 5 directions. The directions are defined by the arrays **ah** and **bh** shown below the graph. The numbers correspond to the vectors drawn in the graph. The engineers can drag the ends of the vectors and then see how the contour map changes. Making the vectors longer results in hillier terrain.

There are 3 similar sets of 5 vectors that control vegetation and point features. The 3 sets interact to produce the result so it is just a matter of experimenting to see what happens.

Underground mines / dungeons



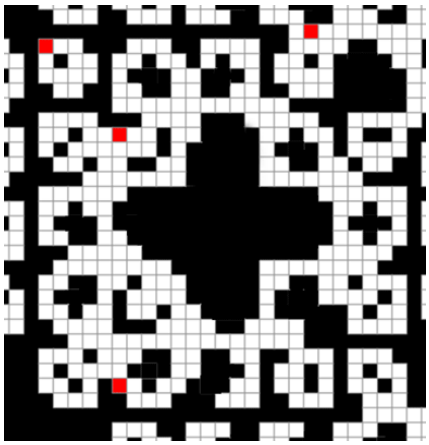
One of the types of point features on the ground is a mineshaft. Explorers (non-orienteers) can fall down the shafts if they get too close. There is a layer of mines under the whole of the terrain in a simple rectangular grid pattern. The grid squares are 16 metres across.

The determination of whether any given grid square is open (as opposed to solid rock) is extremely simple. Just determine whether the fractional part of some function of grid x and grid y is above or below some threshold value:

```
// Is a mine open at this position?
// Must be if there's a mineshaft on the ground
Mine.prototype.isOpen = function (x, y)
{
  if (this.isOpenAbove (x, y)) return true;
  var u = PI10000 * x * y;
  return (u - Math.floor (u)) > 0.4;
  // Note value 0.4 could vary to change
  // openness/connectivity
};

// There is 1 level of mines, so open above
// only if the ground has a mineshaft
Mine.prototype.isOpenAbove = function (x, y)
{
  for (var ix = x - 8; ix < x + 8; ix++)
  {
    for (var iy = y - 8; iy < y + 8; iy++)
    { if (terrain.terra (ix, iy).feature === FEATURES.MINE)
        return true;
    }
  }
  return false;
};
```

The value of u could also be used as a seed to determine various properties of each cell: wall colours or images, furnishings, and so on.

Next is a tiny fragment of map of the mine level. The cells shown in red are below mineshafts. Although the user can move in any direction, just like above ground, cells are not connected diagonally. This fragment shows a complete connected mine with 2 access shafts. At top left there is also a small mine with only one shaft. I don't know how far the other mines around the edges extend or whether they are all accessible.



If this map looks too regular there is a simple way to make it less so. In a variant of **The Forest** called **My Terrain** (at grelf.itch.io/terrain) I changed the program line which sets `u`, to this:

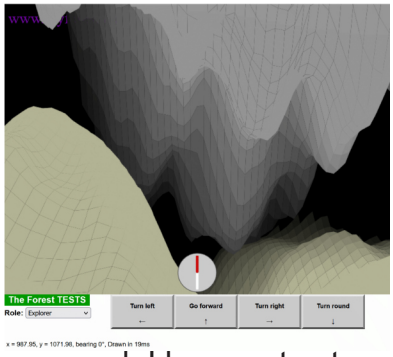
```
  var u = PI10000 * Math.sin(x) * Math.cos(y);
```

In **My Terrain** you can also view a map of the mines as you move around in them. Here is an example. You can see that the connectivity is more random. The yellow cells show where ladders occur (because there is a mineshaft above).

The mines could very easily be extended to 3 dimensions, with a z value as well as x and y. A simple demonstration of that is available at https://grelf.net/caves/cavefly.html and that program allows the connectivity threshold to be varied by the user.

Caverns

I have several times seen it written that height maps, such as I have described, cannot generate overhangs and therefore cannot have cave entrances. This is not true. I have shown how I determine that there are vertical mineshafts in some locations. If we examine the slope of the ground around such a mineshaft and find that it is steeper than some value it would not be difficult to portray a cave mouth instead, leading into a horizontal tunnel. The path of the tunnel can be determined just as in the previous section about mines. A 3D version could then easily connect sometimes downwards too, as already indicated.



Irregular caverns are easily generated too, rather than grid-based mines/dungeons. Use 2 height maps, for roof and floor. Offset them horizontally so they are not parallel and offset them vertically to adjust the amount or frequency of open spaces between the two layers. Scale factors can also be introduced to change the frequency of undulations, to be different from the smoother hills above ground. I have not yet made this available in The Forest but here is an example from my test page. It's just vanilla Javascript without added textures and only ambient lighting with my usual distant fogging.

A note about coordinates

Some 3D programs have x and z as the horizontal coordinates and y vertically. I think this is because they started with views from cameras, in which z is always a measure of depth. Objects at higher z are further from the camera and so are drawn first (the concept of z-buffer has been around from the early days of computer graphics). However, the scene being viewed is more naturally described by x and y for the ground and z vertically, so that is how my programs do it.

I think people get very confused about coordinate systems in 3D graphics.

There are two completely independent coordinate systems: one for the display screen and one for the world being depicted. Both can be described by Cartesian coordinates, x, y and z. In the following I will prefix those letters by s for screen and w for world, to make the distinction clear.

The trouble is that people start thinking from the flat display screen. It has sx horizontally, sy vertically and then we naturally use sz as distance from the viewer (eye/camera), so that farther objects are plotted before nearer ones.

On the other hand the world is best represented by wx going east, wy going north and wz upwards, all from some chosen origin (wz perhaps goes from 0 at sea level).

The screen coordinates (sx, sy, sz) for plotting any object that exists at (wx, wy, wz) in the world are derived from the world coordinates by a perspective algorithm that depends on the (wx0, wy0, wz0) position of the eye and the direction (3 angles) in which it is looking. sz is easy: the Euclidean distance from eye to object. Calculating sx and sy is more complicated.

All of this means that when generating terrain (for a game, say) it is fundamentally important to do it using the world coordinates. The first thing to calculate is a height map which should be a function obtaining wz given wx and wy:

wz = height (wx, wy). That can be a continuous function so there is a height for any floating point value of the viewer's position (the viewer may then move freely to any world position). We then go on to make other functions to determine vegetation types and discrete objects at any (wx, wy) position.

When it comes to displaying a scene for given viewing position the perspective function comes into play:

```
(sx, sy, sz) =
       perspective (wx, wy, wz, wx0, wy0, wz0, a0, b0, c0),
```

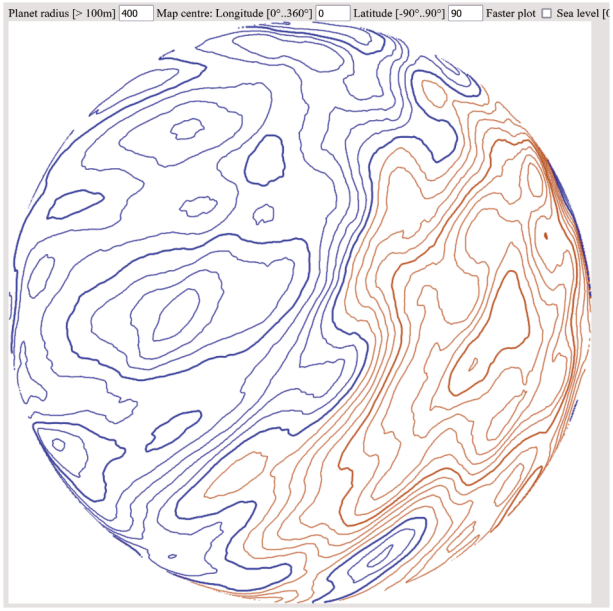
where the last 3 parameters would be Euler angles for the viewer's orientation.

It may then, and only then, be necessary to form tiles and meshes for displaying the scene in a reasonable time. Meshes are likely to be needed for feeding to a GPU, and for that we may need chunks of terrain made using the terrain generating functions:

```
    height (wx, wy), vegetation (wx, wy), feature (wx, wy).
```

I think systems such as three.js are very good but they add to the confusion by using x and z for the ground and y vertically. This implies that they start thinking from the screen rather than the world and that complicates everything.

Terrain for spherical planets



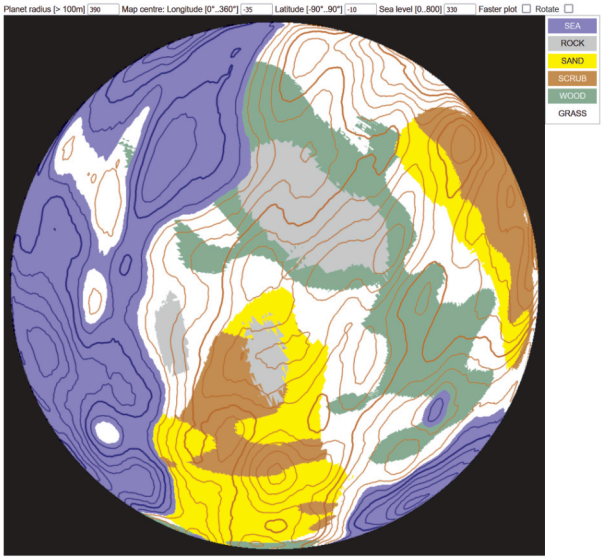
The key thing about extending to spherical surfaces is NOT to use latitude and longitude as the parameters for the generating function. The reason is that that coordinate system becomes ambiguous at the poles. The latitude there is +/-90 degrees but the longitude can have any value. As you approach the pole along different lines of longitude you will get different values for any function based on those 2 coordinates.

To make a map centred on any lat/long position you need first to find the Cartesian coordinates of that position relative to the centre of the sphere and then use those x, y and z values for the calculation. All 3 values change smoothly at the poles, as elsewhere, so there is never any singularity. The contour map on the right here is looking down on the north pole of a planet (land contours brown, sea blue).

My height calculation (page 3) is easily extended to become

```
height = sumOverI
(profile [(a[i] * x + b[i] * y + c[i] * z) & 0xff]) * RECIP128;
```

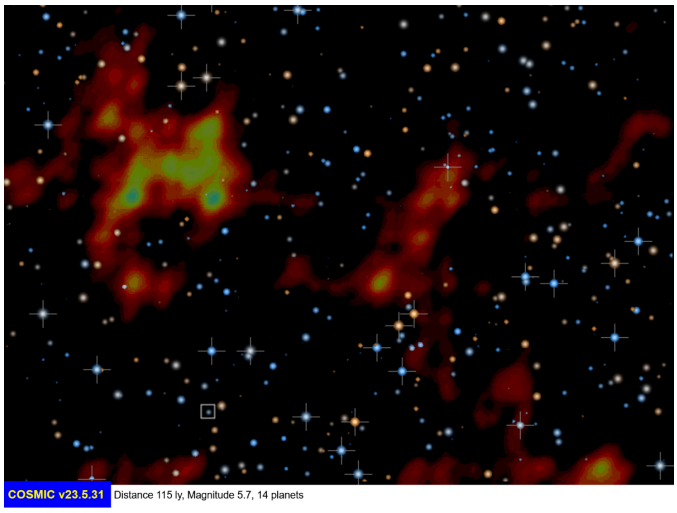
The terrain types work similarly, as shown here:



Integer values of x, y and z form circular rings on the surface, around the 3 axes. Point features can potentially be generated or placed at the intersection points of any pair of those. Placement keys must distinguish + or - values of the third coordinate.

A demonstration can be found at grelf.itch.io/planet where there are also source files to download.

3D space: nebulae and stars



Now take a step further in 3D. Make the 3 coordinates completely independent instead of being constrained to lie on the surface of a planet. I showed on page 3 how islands could be made by raising the lake level. In 3D consider an analogous threshold value such that below the threshold there is empty space but above it there is increasing density of gas, forming astronomical nebulae as if they are 3D islands. This is a very straightforward extension of my basic terrain generator.

My demonstrator may be seen at grelf.itch.io/cosmic and the full source code is available for download from that page.

The stars are scattered through 3D space in just the same way that point features, such as boulders and ponds are scattered in my forest terrain. Here is the program function responsible:

```
// If there is a star here return its data, else null.
// This is designed to produce a very very small number of
// stars per unit volume. Apparently random but always the
// same result for any given x, y, z (assumed to be integers)
Galaxy.prototype.calcStar = function (x, y, z)
{ let a = 0;
  for (let i = 0; i < 5; i++)
  { let j = (this.S [i] * x + this.T [i] * y +
             this.U [i] * z) * this.RECIP128;
    a += this.PROF [j & 0xff];
  }
```

```
  let xyz = x * y * z;
  let f = Math.round (a * xyz * this.RECIP128) & 0xfff;
  if (f === 127) //Now examine bit patterns:
  { let xyzff = xyz & 0xff;
    if (xyzff > 0 && xyzff < 8)
    { let mag = this.MIN_MAGNITUDE +
                this.MAGNITUDE_RANGE * xyzff / 7;
      let nPlanets = (xyz & 0xf00) >> 8; // 0..15
      let colour = (xyz & 0xf000) >> 12;//0..15.
        //Kind of colour index. Stars are not white!
      return new Star (x, y, z, mag, nPlanets, colour);
  } }
  return null;
};
```

Suitable values for the numbers in the algorithm were determined by experimentation.

A downside is that the 800x600 pixel view takes more than 15 seconds to draw (in Firefox, Windows 11, Surface Book 3) because it is probing 300 light years depth (screen scale is supposed to be 1 ly/pixel). But then, even if we could travel at a sizeable fraction of the speed of light it would take many months before the perspective view of such things would change noticeably.

Note that it is not the drawing of the graphics that takes time but the scanning of a block of space 800 x 600 x 300 light-years. I am therefore interested in SYCL (search for that) as a means of parallelising the generator rather than the graphics. I think SYCL has more potential than WebGL or WebGPU for this kind of thing.

Programmers interested in the source files may like to try the following exercises relating to the initial view.

1. Display the average density of stars as a (very small) number per cubic light year.

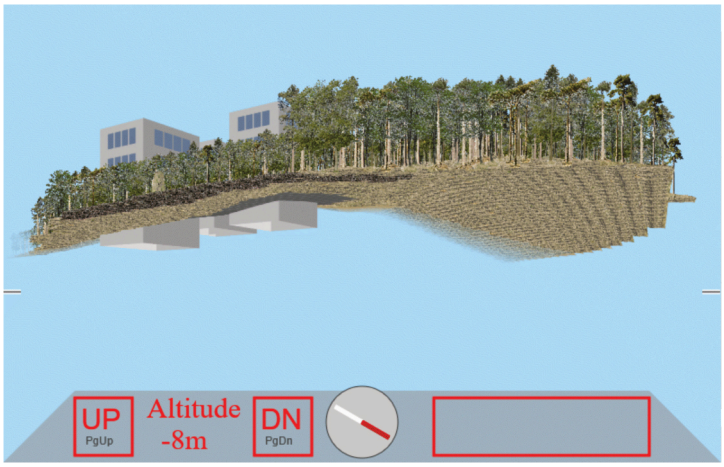
2. How far is the nearest visible star and how many planets does it have?

3. Display a table of the numbers of stars for each of the possible numbers of planets.

4. Demonstrate that there is no correlation between star colour and number of planets.

The point is that although I wrote the program that generates the stars I did not know the answers to all of those questions without writing such additional code. And the same goes for my original terrain generator - I can explore it too, and I had to when looking for locations for the orienteering courses and the treasure hunt.

Islands in the sky



A further demonstration of the versatility of my terrain generation algorithm is shown in the next image. This variant of The Forest at grelf.itch.io/skylands enables flying around, over and under islands as well as landing on them to explore them.

The only changes that had to be made to the code were

- to raise the water level to make islands in a sea rather than lakes in land,
- to not paint the sea but allow the background sky to show through,
- to enable helicopters to go below the water level when not over an island, and
- to not allow a helicopter to go up through the bottom of an island.

I am very pleased with the result.

Evolution in time

You will see from the Java source files below that the method `calcHeight()` uses two arrays of parameters, one for the x coordinate and one for y. It is very simple to add a third for time, such as time since the program started. If the factor for adding in this third contribution is quite small we get slowly evolving terrain. There is a demonstration at https://grelf.net/tt/ where the rate of evolution can be controlled and there are some other ideas too.

Changing the profile - random?

My latest version, for a new game called **Pieces of Eight**, is even more random because the terrain is different every time the program runs (though there are facilities for saving and reloading a particular map and game state). This means that objects can no longer be placed in advance. Where there might have been a building on a hillside that same (x, y) location may be under the sea the next time the program runs. This posed a very interesting programming challenge: how to position the objects I needed to place for the game, so that they will always be discoverable and usable. The player too must start on dry land and not over a mine shaft or be stuck in the wall of a building, or whatever.

I devised an algorithm for the purpose. A new function `findGoodSite(x, y)` takes as input a starting (x, y) position. It searches according to certain criteria and returns an object with new x and y and also a boolean to say whether the search was definitely successful (which I will explain further below).

The search criteria are that the new location must, if possible, be:

- on dry land, at least 10m above sea level;
- on grass or moorland or in open woodland (not in a thicket);
- not on any existing feature such as a boulder or pond;
- at least 7m from any mine shaft;
- at least 10m from any building.

There has to be some compromise because the search must not take "too long" and in particular it must not get stuck in an infinite loop. How long is too long? That is difficult to define but for my new game I need to position 8 objects plus the player at the start.

If any of the loops runs more than 1,000 times the algorithm returns with the latest best position and a false boolean so the caller might deal with that. Also console.log() shows which loop overran.

I allowed the compromise to mean that if the search is unsuccessful within certain conditions the program may alter the map to make a suitable site. My terrain generator did already have the necessary capabilities for altering small parts of the terrain. That is allowed during start-up but not after that: to be fair to the player we cannot have the map changing arbitrarily during the game. Altering the ground like this does mean that sometimes a circular patch around the positioned object is rather visible on the map. An improvement might be to use an irregular shape for the alteration.

This scheme works most of the time but the result is not perfect.

First, how is the profile changed at the start of every run?

As noted on page 2, the beginning and end of the profile array must have similar values and similar slopes. This is easily accomplished if we create the profile as a sum of sine waves with random amplitudes and, importantly, periods which are simple fractions of the length of the profile. This is like harmonics on a string except that we can also randomly vary the phase (the starting angle) for each sine wave and in that way the end points need not be zero.

My original profile, from 1983, had only positive values, in a 1-byte range (simplest for programming in assembler) but the sum of sine waves will be balanced around zero. In my old version the water level was set at 204 but now we need values around zero. A small positive value for the water level was chosen for “Pieces of Eight”, to make islands in a sea rather than lakes in a limitless land mass.

So the following function was added to the Terrain type (Javascript).

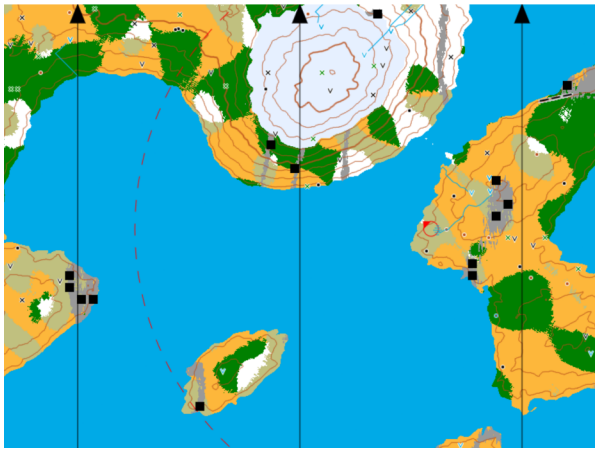
```
Terrain.prototype.makeProfile = function()
{
  let L = this.PROFILE_LENGTH, P = new Array(L);

  for (let i = 0; i < L; i++) P[i] = 0;

  let nSines = 32;
```

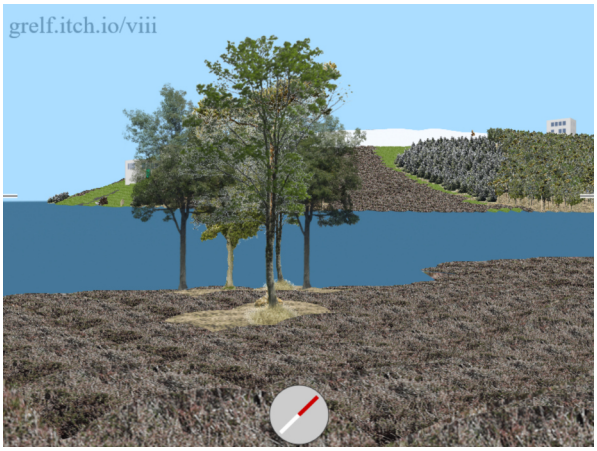
```
  let MIN_A = 16, MAX_A = 90;//Amplitude
  let MIN_N = 2, MAX_N = L / 32;//No of cycles

  for (let i = 0; i < nSines; i++)
  {
    let a = random(MIN_A, MAX_A);
    let n = Math.round(random(MIN_N, MAX_N));
    let lamda = L / n;//Wavelength
    let phase = random(0, TWO_PI);

    for (let j = 0; j < L; j++)
    {
      P[j] += Math.round((
                a * Math.sin(phase + TWO_PI * j / lamda)) / 8);

  } }

  let maxHt = 0;

  for (let i = 0; i < L; i++)
  {
    if (P[i] > maxHt) maxHt = P[i];
  }

  this.LAKE_HT0 = this.lakeHt = 34;
  this.snowHt = maxHt * 2.4;
  this.PROFILE = P;
};

function random(min, max)
{ return min + Math.random() * (max - min); }
```

The number values were the result of considerable experimentation but try others.

The profile length could be made variable too, as long as it is a power of 2.

An example of a map and corresponding scene resulting from `makeProfile()`:





Placing objects safely in the random terrain is more complicated. The code can be seen in the file https://github.com/grelf-net/viii/blob/main/terrainRA.js

Making hilltops more like mountains

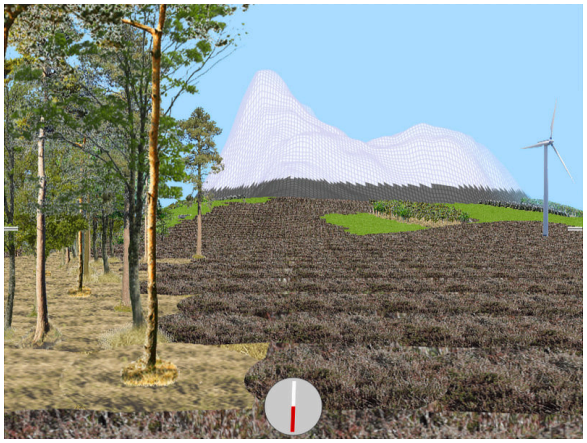
Sometimes mountains are required in the terrain. As described so far the tops of hills are all smooth and rounded. They may be covered with snow if they go above a preset snow level.

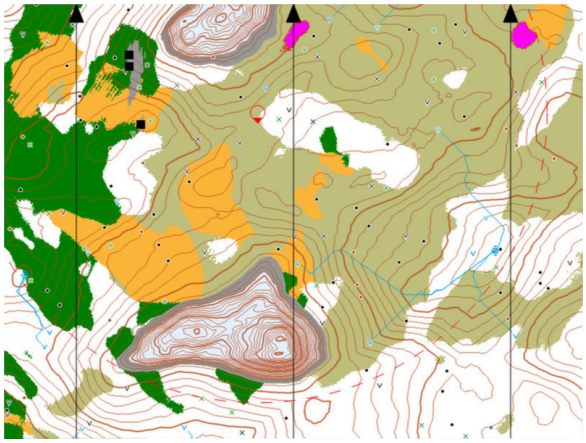
There is a simple way to make such summits look much more like mountains. In pseudocode the algorithm is as follows.

```
z = calculate height (x, y)

if z > mountainBaseHeight
{
  dz = z - mountainBaseHeight
  z = mountainBaseHeight + dz * squareRoot(dz)
}
```

I originally tried adding `dz * dz` to the base height but that produced results that were far too exaggerated. So I have ended up with dz to the power of 1.5. I have not checked whether using `Math.pow()` in Javascript would be faster or slower than `Math.sqrt()` as shown.

I have added this to my experimental version, My Terrain. Here are a resulting scene and corresponding map.





Reference sources in Java

The following files are a slightly cut-down version of my Java reference version of The Forest, as on github. This much I consider now to be public domain and I want others to use it, modify it, extend it and generally be creative.

==== File Terrain.java

```
package net.grelf.forest;

import java.util.HashMap;

/** The Relf Terrain Generator.
  *
  * This algorithm was first developed in 1983 in Z80 assembly
  * code for the Sinclair ZX Spectrum (48kB RAM).
```

```
  * It was converted to Javascript in 2014.
  * C++ and Java versions were made for comparison in 2021.
  * This Java version is the clearest of course.
  * Experiments in 2023 to implement this as non-graphical
  * WebGL did not improve speed for a map 800 x 600. WebGPU
  * may be better when generally available.
  */
public class Terrain
{
  private static final double PI1000 = Math.PI * 1000;
  private static final double PI10000 = Math.PI * 10000;
  public static final int LAKE_HT0 = 204;
  private static final double RECIP128 = 1.0 / 128.0;
  private final HashMap <String, Object> PLACED;

  /** Reference to the main program (needed in just 1 place,
    * legacy from JS). */
  public Forest FOREST;

  /** Current height of lake edges. */
  public int lakeHt;

  /** User can potentially alter these booleans. */
  public boolean bleak, woodOnly, withPaths, withStreams,
                 mapMode;

  // NB: The length of the next array MUST be a power of 2.
  // It could be much longer without affecting performance.
  // This is the original profile used in 1983 for The Forest
  // in ZX Spectrum.
  private static final int [] 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,
25,25,26,26,25,25,25,25,25,25,26,28,29,30,33,36,37,39,39,40,40,
40,39,39,39,37,37,37,36,36,36,35,35,33,33,32,30,28,25,20,15,
11,10,9,9,9,11,14,15,17,17,18,18,18,18,18,18,17,17,17,15,
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
  };

  private static final int BIT_MASK = PROFILE.length - 1;
  // All bits must be 1

  //The next arrays must be of length to match the for loops
  //in calcProf() and calcHeight()
  private static final int [] AH = {0, 13, 21, 22, 29};
  private static final int [] BH = {27, 26, 21, 11, 1};
  private static final int [] A1 = {-43, -43, -56, 31, 4};
  private static final int [] B1 = {-3, -12, 22, 2, 32};
```

```
private static final int [] A2 = {-24, -25, 60, 10, - 30};
private static final int [] B2 = {15, -54, -34, -51, -43};
private static final int [] A3 = {-51, -62, -58, -64, 33};
private static final int [] B3 = {-44, 20, 27, -64, -44};
private static final int PARAM_LENGTH = AH.length;

/** Likely to be a singleton in many programs. */
public Terrain ()
{
  this.FOREST = Forest.getInstance (); // Main program
  this.lakeHt = LAKE_HT0;
  this.bleak = false;
  this.woodOnly = false;
  this.withPaths = false;
  this.withStreams = true;
  this.mapMode = true;
  this.PLACED = new HashMap <> ();// Allows objects to move
} // Terrain

/** Position an object on the ground. */
public void place (int x, int y, Object n)
{
  PLACED.put (x + "," + y, n);
}

/** Get any object at this position, null if none. */
public Object atPlace (int x, int y)
{
  return PLACED.get (x + "," + y);
}

/** Remove any object from this position. */
public void remove (int x, int y)
{
  PLACED.remove (x + "," + y);
}

/** Remove all objects of the given kind, from the whole
  * terrain. */
public void removeAll (TERRAINS kind)
{
  for (String key : PLACED.keySet ())
  {
    if (PLACED.get (key) == kind)
    {
      PLACED.remove (key);
    }
  }
} // removeAll


/** Clear all streams from the terrain. */
public void clearStreams ()
{
  removeAll (TERRAINS.STREAM);
}
```

```
/** Get all the terrain details for the given ground
  * position. */
public Terra terra (double x, double y)
{
  int a, b;
  FEATURES feature = FEATURES.NONE;
  String code = "";
  double ht = calcHeight (x, y);

  if (this.bleak)
  {
    return new Terra (x, y, ht, 0,
                      TERRAINS.Z, FEATURES.NONE, "");
  }

  if (withPaths)
  {
    if (ht > lakeHt
     && Math.abs (ht - calcHeight (x + 4000, y + 2000)) < 2)
    {
      return new Terra (x, y, ht, 0,
                        TERRAINS.PATH, FEATURES.NONE, "");
    }
  }

  int xr = (int) Math.round (x);
  int yr = (int) Math.round (y);
  long xryr = xr * yr;

  Object pd = PLACED.get (xr + "," + yr);


  if (null != pd)
  {
    if (pd instanceof TERRAINS)
    {
      if (pd == TERRAINS.STREAM
       && FOREST.view == VIEWS.SCENE)
        return new Terra (xr, yr, ht, 0,
                          TERRAINS.STREAM, FEATURES.NONE, "");

      if (pd == TERRAINS.ROAD)
        return new Terra (xr, yr, Math.max (ht, lakeHt), 0,
                          TERRAINS.ROAD, FEATURES.NONE, "");
    }
    else if (pd instanceof FEATURES)
    {
      if (pd == FEATURES.CONE // Helicopter landed?
       || pd == FEATURES.ROCKET
       || pd == FEATURES.T)
        feature = (FEATURES) pd;
      else
      if (pd == FEATURES.NONE) // Helicopter departed?
        return new Terra (xr, yr, ht, 0,
                          TERRAINS.GRASS, FEATURES.NONE, "");
    }
  }
```

```
if (lakeHt > ht)
  return new Terra (xr, yr, lakeHt, lakeHt - ht,
                    TERRAINS.LAKE, feature, "");

if (LAKE_HT0 > ht)
  return new Terra (xr, yr, ht, 0,
                    TERRAINS.MUD, feature, "");

if (FEATURES.NONE == feature)
{
  a = calcProf (B2, x, A3, y); // NB: Swapped a/b tables
  long f = Math.round (a * xryr * RECIP128) & 0xfff;

  if (4 == f)
  {
    long 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;

    code = getCode (xr, yr);
  }
  else if (8 == f
        && (Math.round (PI1000 * xryr) & 0xff) < 4)
  {

    feature = FEATURES.X; code = this.getCode (xr, yr);
  }
  else if (16 == f
        && (Math.round (PI10000 * xryr) & 0xff) < 8)
    feature = FEATURES.CONE;
}

if (!woodOnly)
{
  a = calcProf (A1, x, B1, y);

  if (120 > a)
  {
    return new Terra (xr, yr, ht, 0,
                  TERRAINS.TOWN, FEATURES.NONE, "");
  }

  a = calcProf (A2, x, B2, y);
  b = calcProf (A3, x, B3, y);

  if (255 > a)
  {
    if (255 > b)
      return new Terra (xr, yr, ht, 0,
                  TERRAINS.GRASS, feature, code);

    return new Terra (xr, yr, ht, 0,
                  TERRAINS.MOOR, feature, code);
```

```
    }

    if (200 > b)
      return new Terra (xr, yr, ht, 0,
                        TERRAINS.THICKET, feature, code);
  }

  return new Terra (xr, yr, ht, 0,
                    TERRAINS.WOOD, feature, code); // runnable
} // terra

private int calcProf (int [] a, double x,
                      int [] b, double y)
{
  int p = 0;

  for (int i = 0; i < PARAM_LENGTH; i++)
  {
    p += PROFILE [(int) (((long) Math.floor (
                  a [i] * x + b [i] * y)) >> 7) & BIT_MASK];
  }

  return p;
} // calcProf

/** Get the ground height at the given position.
  * Interpolates between integer coordinates for smooth
  * slopes. */
protected double calcHeight (double x, double y)
{
  double ht = 0;

  for (int i = 0; i < PARAM_LENGTH; i++)
  {
    double j = (AH [i] * x + BH [i] * y) * RECIP128;
    int jint = (int) Math.floor (j);
    double jfrac = j - jint;
    int prof0 = PROFILE [jint & BIT_MASK];
    int prof1 = PROFILE [(jint + 1) & BIT_MASK];
    ht += prof0 + jfrac * (prof1 - prof0); // interpolate
  }

  return ht;
} // calcHeight

/** For orienteering control points: */
private static final String ALPHABET =
                                "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

/** Get the orienteering control code for the given ground
  * position, 2 capital letters. */
public String getCode (int x, int y)
{
  int x26 = Math.abs (x) % 26;
  int y26 = Math.abs (y) % 26;
  return ALPHABET.substring (x26, x26 + 1) +
         ALPHABET.substring (y26, y26 + 1);
```

```
  } // getCode

} // Terrain
```

==== File Terra.java

```
package net.grelf.forest;

/** To contain details of the terrain at a ground point. */
public class Terra
{
  public double x, y, height, depth; // depth only for lakes
  public TERRAINS terrain;
  public FEATURES feature; // FEATURES.NONE if none present
  public String code; // 2-letter O-control code, or ""

  public Terra (double x, double y, double height,
                double depth, TERRAINS terrain,
                FEATURES feature, String code)
  {
    this.x = x;
    this.y = y;
    this.height = height;
    this.depth = depth;
    this.terrain = terrain;
    this.feature = feature;
    this.code = code;
  }

} // Terra
```

==== File TERRAINS.java

```
package net.grelf.forest;

public enum TERRAINS
{
  LAKE, TOWN, GRASS, MOOR, WOOD, THICKET, ROAD, MUD, PATH,
  STREAM, MARSH;
}
```

==== File FEATURES.java

```
package net.grelf.forest;

public enum FEATURES
{
  NONE, MINE, BOULDER, ROOT, WATERHOLE, KNOLL, X, CONE,
  ROCKET, BUILDING;

  public String getDescription ()
  {
    switch (this)
    {
    case MINE: return "mineshaft";
    case BOULDER: return "boulder";
    case KNOLL: return "knoll";
```

```
    case ROOT: return "rootstock";
    case WATERHOLE: return "water hole";
    case X: return "man-made";
    }

    return "unknown";
  } // getDescription

} // FEATURES
```

The descriptions are really for orienteering courses. Note that cone and rocket are movable and so do not appear on the map. Buildings are self-evident and so also do not need description strings.

Graham Relf *revised December 2025*