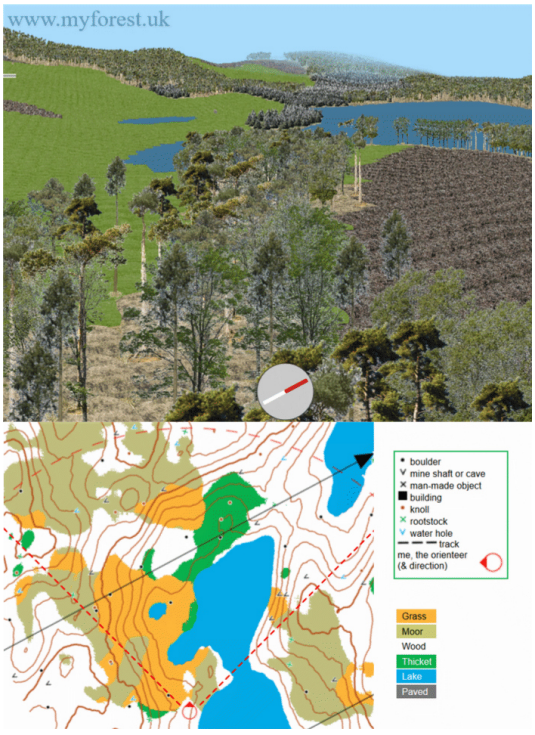
How to make limitless terrain for games in real time

This is a description for game programmers of how I generate essentially limitless terrain for a simulation of the sport of orienteering which may be seen and explored at ***grelf.itch.io/forest*** or at ***myforest.uk*** (The Forest, completely free). The terrain is not stored permanently as data. Instead it is generated by mathematical functions as the explorer or orienteer moves around.

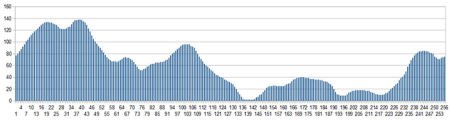


The scene and corresponding map here are from the program running in Firefox browser.

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 above) is programmed in JavaScript to run in any HTML5 graphical browser, even on phones. I also have versions written in Java and in C++.

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 (x eastwards, y northwards).

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

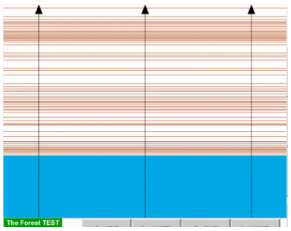
Performance is important here, so we do not want to be doing a modulus operation because that involves division. Instead a bit-wise AND is done:

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

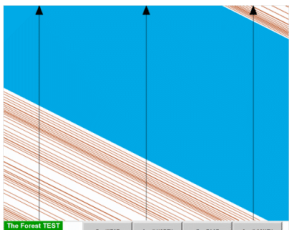
0xff is hexadecimal for decimal 255; I just write it this way to emphasise what is being done.

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

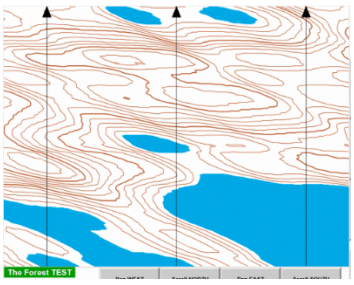
Recall that the number p above is a function of position. If p = 27y (no x-dependence), and the heights are multiplied by 5, and then heights below 204 are deemed to be lake, we get the very boring contour map shown here:



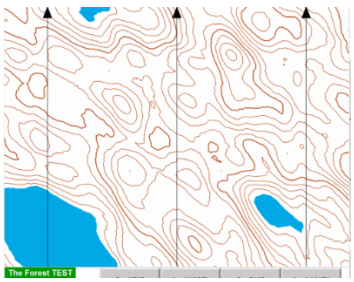
If instead p = 13x + 26y we get the equally boring contour map shown here, with the same pattern just lying at an angle:



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, 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 might be flatter plains and sharper mountainous areas.

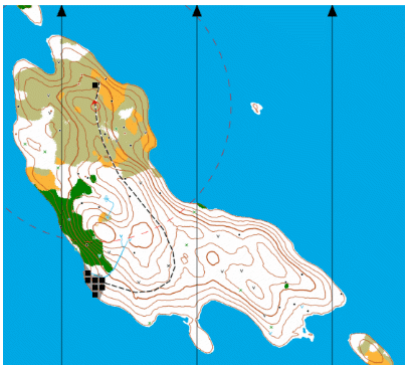
It is also important to note that although the initial profile repeats after 256 metres*, the fact that we are adding versions at various angles means that the resulting terrain is unlikely ever to repeat (until we run out of numerical precision of course).

* 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 the next sections) use only rounded integer versions of x and y, for speed.

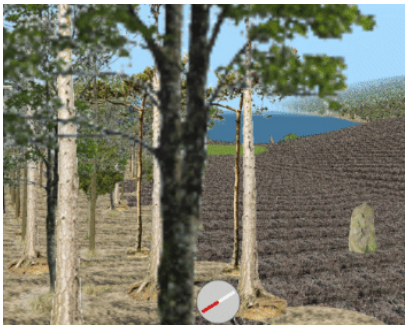
The algorithm for drawing the contours was first described in 1987 in Byte magazine by Paul Bourke. His description can be found at http://paulbourke.net/papers/conrec/ . I have re-implemented the algorithm in JavaScript in a form that is most efficient for my map. It performs well. Every fifth contour 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.

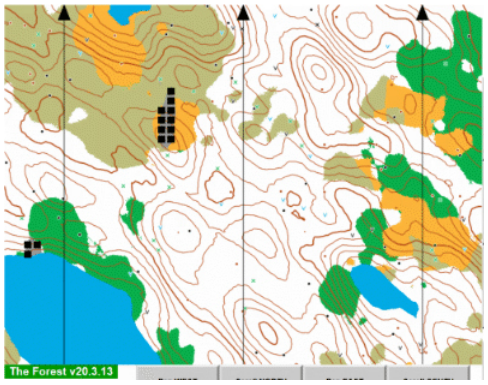


Terrain kinds (vegetation etc)

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 on the right here).



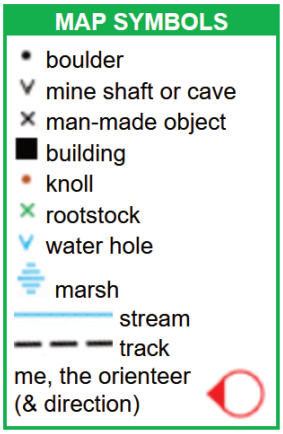
The map is drawn as closely as possible to international orienteering standards (see https://orienteering.sport/iof/mapping/). White areas are runnable woodland (mature trees). Green is thicket, slower to penetrate or even best avoided. Yellow is open grassland but ochre is moorland which is slower to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

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 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

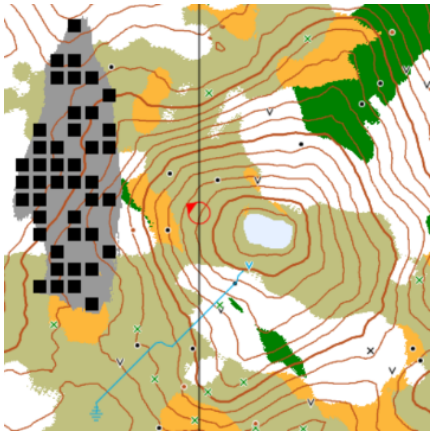
There are trees in The Forest of course. They are loaded into the program as image files (from my own photos, cut out and saved in PNG format to allow transparency around them; transparency is shown by the chequered pattern here). To avoid monotony there needed to be several different tree images. Suppose 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, 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, another feature of my newer version. 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 point shown in the map.

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 (and slightly faster than `'x' + x + 'y' + y` which I first wrote). Secondly it ensures that, for example, x = 123 and y = 45 does not produce the same key as x = 12 and y = 345.

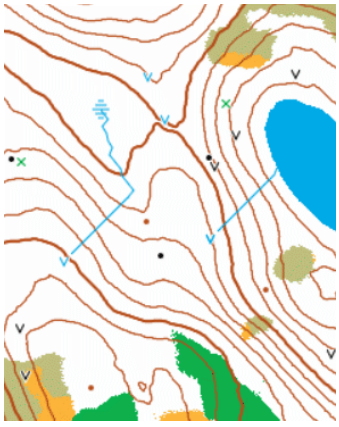
This technique can be used to position any object at a required location or to indicate that an auto-generated feature has been moved, perhaps by the explorer. It does not want to be overdone because it goes against the initial principle of having auto-generated terrain. The technique has been used in 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

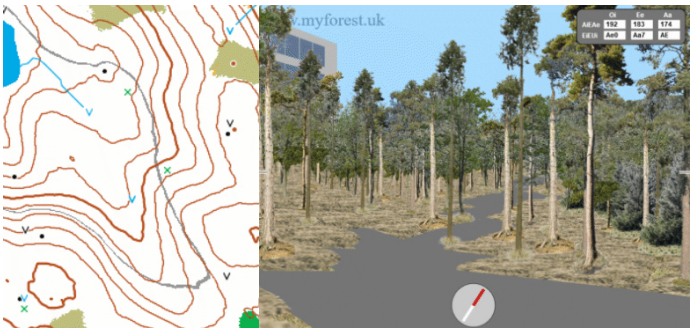
Notice that the map is generated point by (x, y) point. There is no consideration of how neighbouring points may be related and therefore there are at first no linear features such as paths or streams. In fact it is much more difficult to generate a map containing such features. Only the vegetation boundaries 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 are still programming difficulties with streams, so I consider 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?

Paths and roads

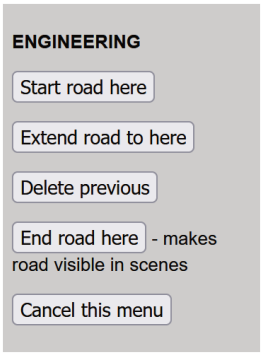


Automatically generating paths or roads is trickier. Unlike streams there is no rationality that can easily be automated for the directions they should take. In the real world they are created by people or animals repeatedly using a route they find useful. We cannot wait for a player to repeatedly follow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



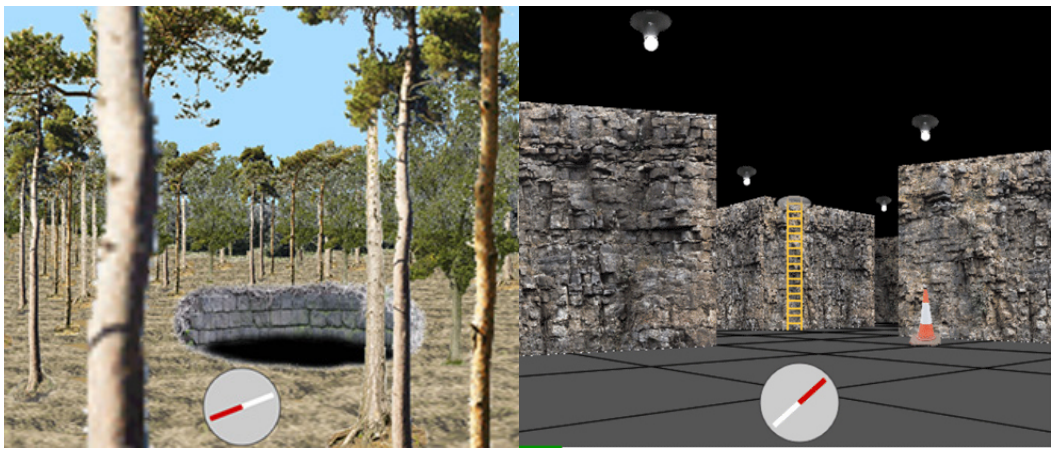
A recent addition to The Forest is the role of Engineer. When in this mode, clicking somewhere on the map produces a pop-up menu (really an HTML `<div>` that is usually hidden but in a fixed position). This enables roads to be drawn as a series of straight line segments. Ending a road asks whether it 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.

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.

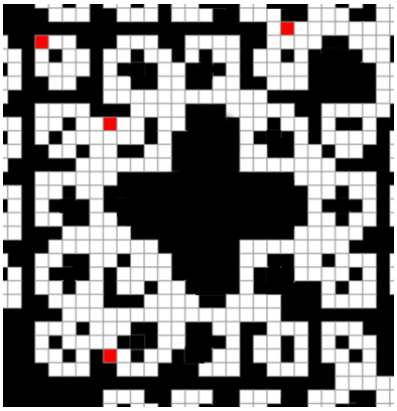


The determination of whether any given grid square is open (as opposed to solid rock) is extremely simple. Just determine whether the fractional part of some function of grid x and grid y is above or below some threshold value:

```
// 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;
};
```



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.

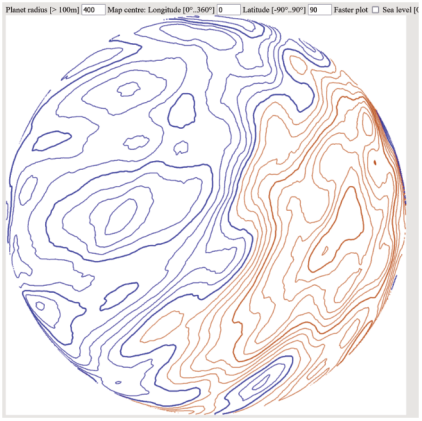
Cave entrances

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.

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.

Terrain for spherical planets



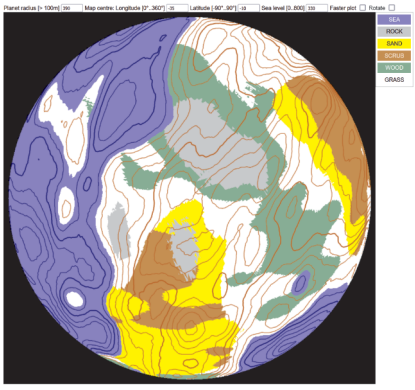
The key thing about extending to spherical surfaces is NOT to use latitude and longitude as the parameters for the generating function. The reason is that that coordinate system becomes ambiguous at the poles. The latitude there is +/-90 degrees but the longitude can have any value. As you approach the 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;
```

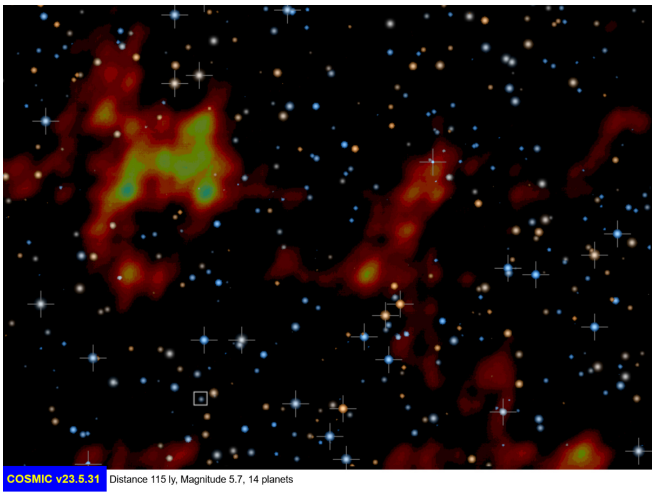
The terrain types work similarly, as shown here:



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.

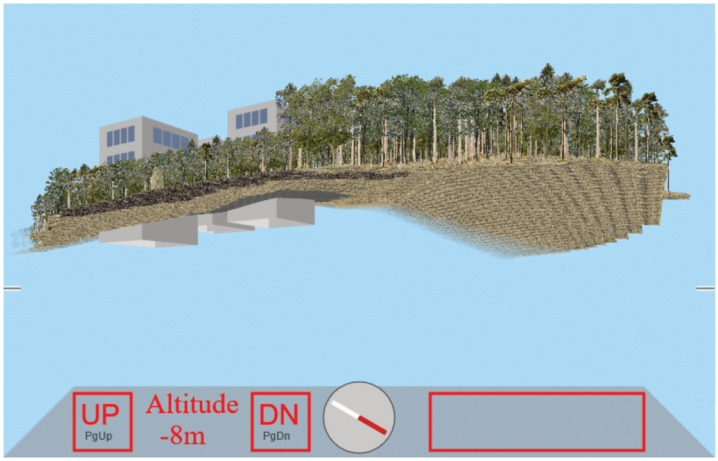
2. How far is the nearest visible star and how many planets does it have?

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 this 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.

Graham Relf, https://grelf.net, June 2023