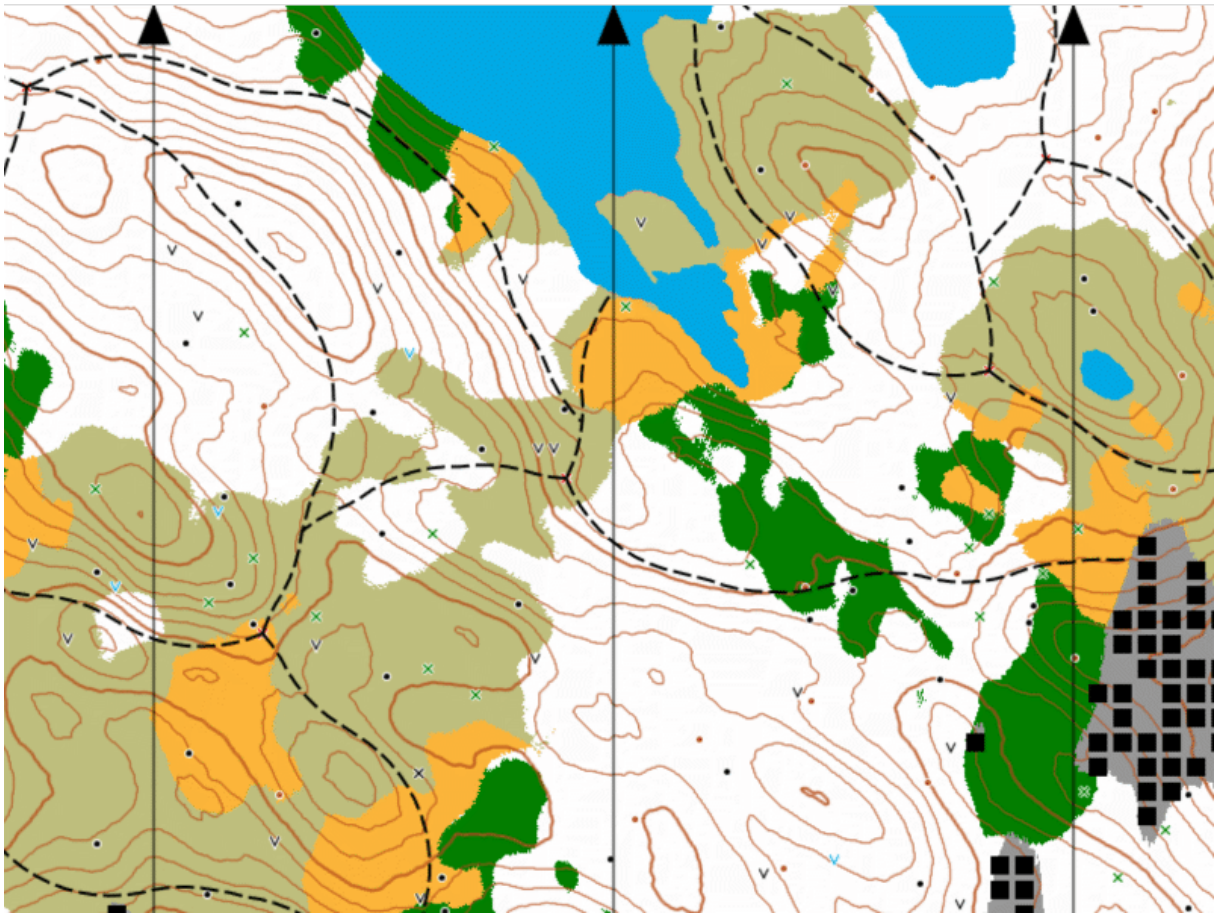
How to generate roads automatically, taking a height map into account

This document describes an algorithm I have developed for automatically generating roads in a limitless terrain system. The roads have a tendency to follow contours. In other words they try avoid having steep slopes.

The terrain system is one I have developed over many years. Its current main incarnation may be explored, completely free, at https://grelf.itch.io/forest and from that page a downloadable PDF file describes how the terrain generator works. The whole program is written in plain JavaScript as a browser client using the standard HTML5 2D canvas with no other libraries or frameworks. Here is an example of a small part of the map it can display, with some of the new auto-generated roads.



The source code for generating such roads will be described in detail. First an outline of the algorithm will be described in words. That will be followed by some background to the programming set-up. Then the full source code will be shown and explained.

The algorithm in words

1. Find starting points, scattered sparsely across the terrain (the terrain generator already has an algorithm for this).

2. For each starting point get a seed value.

3. For each starting point use the seed to determine how many paths (potential

roads) will start there: 2, 3, or 4, with 3 being more likely than 2 or 4.

4. Start each path as a vector, the directions equally spaced around the starting point but randomised slightly by an angle determined by different bits in the seed value.

5. Start a loop. Each time around extend every path by a fixed distance but in a direction determined as follows.

    - Look at the height ahead on the previous bearing and the heights 0.1 radians either side of that bearing. The new bearing will be to the point (of those 3) with height nearest to the previous height.

    - Stepping pixel by pixel along the proposed new vector check whether any of the other paths is encountered. If so, stop this path at the encounter, add it to an array of completed roads but remove it from the array of paths being looped through.

    - If the path is longer than a predetermined maximum also stop it, add it to finished roads and remove it from the loop.

    - End the loop when all paths have been removed as indicated above (or, as a protection against infinite looping, if a predetermined maximum number of loops is reached).

6. Draw the completed roads.

The most tricky part of this is checking whether another path has been encountered while trying to extend each path. The technique for doing this will be explained when we reach the relevant source code.

Next is some background to the programming set-up for this development.

The coordinate system

Positions on the ground are fixed by Cartesian coordinates: x runs from west to east and y from south to north. Units are metres and 1 metre on the ground corresponds to 1 pixel on the map.

Directions are determined as compass bearings clockwise from due north (positive y). This means that, unlike the usual maths convention (anticlockwise from positive x), we have the following relations.

```
dx = Math.sin (b); dy = Math.cos (b); b = Math.atan2 (x, y);
```

The bearing is kept in radians unless it has to be converted to degrees for display.

Experimental set-up for developing new techniques

The Forest's main HTML page is in file index.html (the usual web convention). For testing and for developments such as the new road generator there is a copy of this file called tests.html. In this copy the page title is changed, to make it clear it is the test version, and the handler for the onload event becomes a new function in a new script element within the file. In the present case the new onload handler is called `testGenerateRoads`. The code for this will be shown shortly.

It is important that the test file does not change any code in the rest of The Forest. So the new road generator has been developed on this basis. Only when it has been thoroughly tested might it be integrated into the main code files.

Some relevant objects in The Forest

There is one global object called `forest` which contains references to other important singleton objects in the system, including

- `forest.terrain`, of type `Terrain`, has methods for determining height, vegetation types (biomes) and point features at any given (x,y) position. If we only need height there is `forest.terrain.calcHeight (x,y)` but more generally `forest.terrain.terra (x, y)` returns an object containing height, terrain kind, feature, and a few other things.

- `forest.observer`, of type `Observer`, keeps track of everything to do with the user/player such as their role, position and viewing direction and it controls how they can move around.

- `forest.map`, of type `Map`, is responsible for displaying a map of any part of the terrain. (`Map` predates the addition of `Map` to JavaScript's own syntax by several years. This version simply hides the JavaScript version which we do not need.)

The onload handler, testGenerateRoads

```
function testGenerateRoads ()
{ run (1);
  var me = forest.observer;
  me.x = 19000; me.y = 1260;
  var fm = forest.map;
  fm.showCones = true;
  fm.oldPNL = fm.plotNorthLines;
  fm.plotNorthLines = generateRoads;
  toMap ();
}
```

The function `run()` is the usual onload handler of this application but calling it with a parameter (can be anything) causes it to wait for other things to be set before we continue with `toMap()`.

Scattered throughout The Forest there are traffic cones, about 10 per square kilometre. Since these would in principle be movable they are not normally shown on the map. But as it has previously been useful for testing, `Map` has a static boolean property `showCones`. For the road generation experiments the cones are used as starting points, so we need their positions to be found while drawing a map as if they were to be shown. To do that we set `showCones` to `true`.

To be able to draw roads on the map without changing the `Map` code we use a neat feature of JavaScript to intercept one of Map's existing methods, `plotNorthLines`, so that it instead uses the function to be tested, `generateRoads`, which will then invoke the original plotting function before doing its new work.

The new algorithm is implemented in `function generateRoads ()`.

The JavaScript source for the algorithm, explained

```
function generateRoads ()
  // Overriding forest.map.plotNorthLines() which would be
  // called during forest.map.draw()
{
  var fm = forest.map, g2 = fm.g2, // g2 is 2D canvas context
      startPts = fm.cones, // Array already filled by fm.draw()
      nStarts = fm.nCones; // Also already set by fm.draw()
  fm.oldPNL ();
  // Plot north lines, as done at this stage in fm.draw()
  var paths = [];
  // Each path is an array of {x, y, b, code} where b is the
  // bearing to the next {x, y} point, code is the initial
  // path index in the array but is only in path [0]
  for (var i = 0; i < nStarts; i++)
  {
    var si = startPts [i];// Seed point, in canvas coordinates
    var groundXY = fm.getGroundXY (si.x, si.y);
        // Converted to ground coordinates
    var seed = (PI1000 * groundXY.x * groundXY.y) & 0xff;
      // 8 bits. PI1000 is a global constant, Math.PI * 1000
      // Note: for given x & y the seed is always the same
    var nPaths = (seed & 3) + 1; // 1..4
    if (nPaths === 1) nPaths = 3; // 2..4 with 3 more likely
    var a = TWO_PI / nPaths; // TWO_PI is global, Math.PI * 2
    var da = seed * 0.01; // 0..2.55
    for (var j = 0; j < nPaths; j++)
    {
      var aj = a * j + da;
      paths.push (
        [{x: groundXY.x, y: groundXY.y, b: aj, code: i}]);
    }
  }
  var roads = [];// Array for collecting finalised roads
  var marks = {};// Object for finding encounters between paths
  var nLoops = 0, STEP = 10, MAX_PATH_LENGTH = 100;
  while (paths.length > 0 && nLoops < 300)
    // nLoops is an arbitrary protection
  {
    for (i = paths.length - 1; i >= 0; i--)
    { var pi = paths [i];
      var endXYB = pi [pi.length - 1]; // Last vector so far
      var b = nextBearing (endXYB, STEP); // See below
      var nextXYB = {x: endXYB.x + STEP * Math.sin (b),
                     y: endXYB.y + STEP * Math.cos (b),
                     b: b}; // See earlier note on coordinates
      pi.push (nextXYB); // Extend this path
      var road = createRoad (pi, marks);
                 // The tricky function, shown below
        // Result type Road already exists in The Forest
        // for predetermined roads (not auto-generated)
      if (road.stopped || pi.length > MAX_PATH_LENGTH)
      {
        roads.push (road);
        paths.splice (i, 1); // This is why i counts down
      }
```

```
    }
    nLoops++;
  }
  for (i = 0; i < roads.length; i++) roads [i].draw (g2);
    // Type Road already has method draw()
}

// Determine a direction which gains or loses least height
function nextBearing (xyb, d)
{
  var ft = forest.terrain;
  var h = ft.calcHeight (xyb.x, xyb.y);
  var b0 = xyb.b;
  var h0 = ft.calcHeight (xyb.x + d * Math.sin (b0),
                          xyb.y + d * Math.cos (b0));
  var b1 = b0 - 0.09;
  var h1 = ft.calcHeight (xyb.x + d * Math.sin (b1),
                          xyb.y + d * Math.cos (b1));
  var b2 = b0 + 0.09;
  var h2 = ft.calcHeight (xyb.x + d * Math.sin (b2),
                          xyb.y + d * Math.cos (b2));
  var dh0 = Math.abs (h - h0);
  var dh1 = Math.abs (h - h1);
  var dh2 = Math.abs (h - h2);
  if (dh1 < dh0 && dh1 < dh2) return b1;
  if (dh2 < dh0 && dh2 < dh1) return b2;
  return b0;
}

// Stops if another road is encountered (or lake or town)
function createRoad (path, marks)
{
  var road = new Road (); // Existing type
  road.stopped = false;   // Extra property (JS flexibility!)
  var x = path [0].x, y = path [0].y, prevX, prevY;
  road.addPoint (x, y);   // Existing method of Road
  for (var i = 1; i < path.length; i++) // Follow the path
  {
    prevX = x; prevY = y;
    x = path [i].x; y = path [i].y;
    var hit = checkPathMeeting (prevX, prevY, x, y,
                                path [0].code, marks);
    if (null !== hit)
    { // Other path encountered: see checkPathMeeting, below
      road.addPoint (hit.x, hit.y);
      road.stopped = true;
      return road;
    }
    road.addPoint (x, y);
  }
  return road;
}

// Go pixel by pixel along one vector on a path
// Returns {x, y} where other paving or lake or town
// encountered, otherwise null
function checkPathMeeting (x0, y0, x1, y1, pathCode, marks)
```

```
{
  var ft = forest.terrain;
  var dx = x1 - x0, dy = y1 - y0;
  var d = Math.sqrt (dx * dx + dy * dy);
  dx /= d; dy /= d;
  var x = x0, y = y0;
  for (var i = 0; i <= d; i++)
  {
    var xr = Math.round (x), yr = Math.round (y);
    var key = xr + ',' + yr;
    var marked = marks [key];
    if ((marked !== undefined) && (marked !== pathCode))
        // Found DIFFERENT path: stop
    {
      return {x: x, y: y};
    }
    var tt = ft.terra (xr, yr).terrain;
    if (tt === TERRAINS.LAKE || tt === TERRAINS.TOWN) // Stop
    {
      return {x: x, y: y};
    }
    marks [key] = pathCode; // Mark a point of THIS path
    x += dx; y += dy;
  }
  return null;
}
```

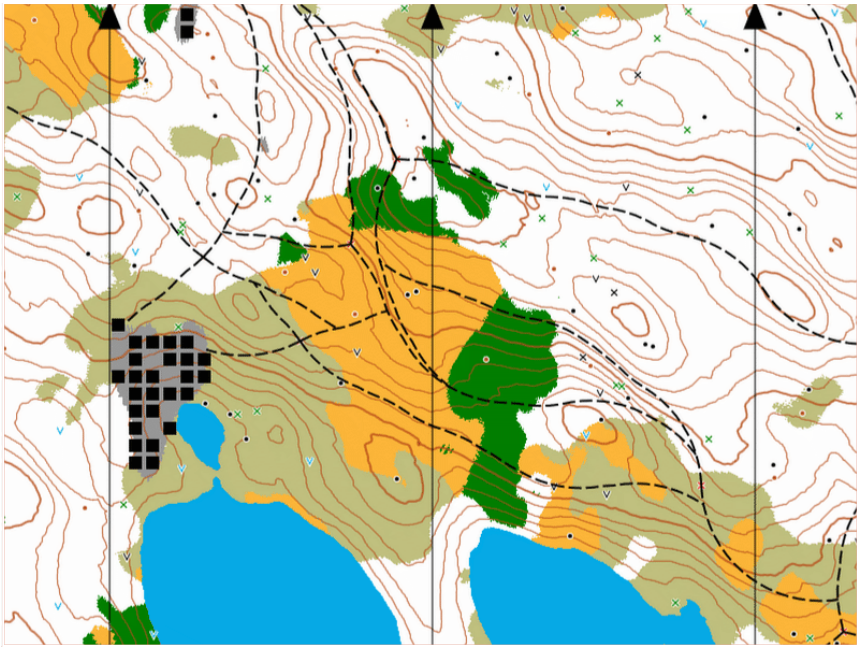
The crucial thing is the object called `marks` which is used for marking the pixels through which any path goes. It relies on a useful feature of JavaScript whereby object properties are really key/value pairs in hash tables. Form a key from the (x, y) coordinates of the ground point corresponding to a pixel on the map and put a unique value there which identifies the path. Then any other path trying to use that pixel finds a code different from its own and therefore has to stop.

How to get the sparse starting points

Cones were used in this development because they already existed. Their occurrence is defined in `forest.terrain.terra (x, y)` by the following lines of code. Something similar would be needed as a separate type of feature if the algorithm is to be integrated into The Forest.

```
  var xr = Math.round (x), yr = Math.round (y),
      xryr = xr * yr;
  //. . .
  var a = this.calcHeight (xr, yr);
  //. . .
  var f = Math.round (a * xryr * this.RECIP128) & 0xfff;
          // RECIP128 is a static constant, 1 / 128
  //. . .
  if (16 === f && ((PI10000 * xryr) & 0xff) < 8)
    feature = FEATURES.CONE;
          // PI10000 is a global constant, Math.PI * 10000
          // FEATURES is like a global enum, numbers named
```

The condition is met very rarely, so there are just a few cones per square kilometre.



In this example there were 6 starting points which resulted in 19 roads.

In Firefox / Windows 11 on a Microsoft Surface Book 3 in battery-saving mode it took about 300ms to generate the roads and draw them on this 800 x 600m map.

There is a snag…

The terrain generator creates limitless terrain, not just the area shown by the initial map. Trouble starts when we scroll the map because some of the road starting points go out of the map area and new ones appear. That means that a different set of roads is produced.

… but here's how to fix it

The important thing is to keep a record of all starting points and keep them in the same order for growing the roads from those points. Starting from the centre of the initial map (the observer's initial position) spiral outwards in a square fashion, testing every point for the existence of a road starting point. Whenever one is found append it to an array of starting points. Generate roads from this array.

Since the algorithm shown on the previous page for sparse starting points only finds about 10 per square kilometre, a large part of the terrain can be covered before the array of starting points becomes too big. Nevertheless it would be sensible to set a limit to the array size. With that in mind the function for scanning a square spiral from starting position (x0, y0) and with a maximum array size of **nMax** might go like this:

```
const AIM = {N:0, E:1, S:2, W:3}, STARTS = [];

function squiral(x0, y0, nMax)
{
  var xMin = x0 - 1, yMin = y0 - 1,
      xMax = x0 + 1, yMax = y0 + 1,
      x = x0, y = y0, aim = AIM.N;
  do
  {
    checkForStart (x, y);// pushes onto STARTS if found
    switch (AIM.aim)
    {
    case AIM.N:
```

```
        y++;
        if (y >= yMax)
        { aim = AIM.E;
          yMax++;
        }
        break;
      case AIM.E:
        x++;
        if (x >= xMax)
        { aim = AIM.S;
          xMax++;
        }
        break;
      case AIM.S:
        y--;
        if (y <= yMin)
        { aim = AIM.W;
          yMin--;
        }
        break;
      case AIM.W:
        x--;
        if (x <= xMin)
        { aim = AIM.N;
          xMin--;
        }
        break;
    }
  }
  while (STARTS.length < nMax);
}
```

There are various ways this algorithm might be used. It could run as a background thread (in JavaScript a worker thread) that sends the main thread starting positions as they are found, for the main thread to add to its `STARTS` array. A better idea might be to run it as an offline pre-process to generate a .js file defining an array of roads, the file to be included by a `<script>` element when the main program runs.

Graham Relf

8 June 2022, revised 13 October 2025