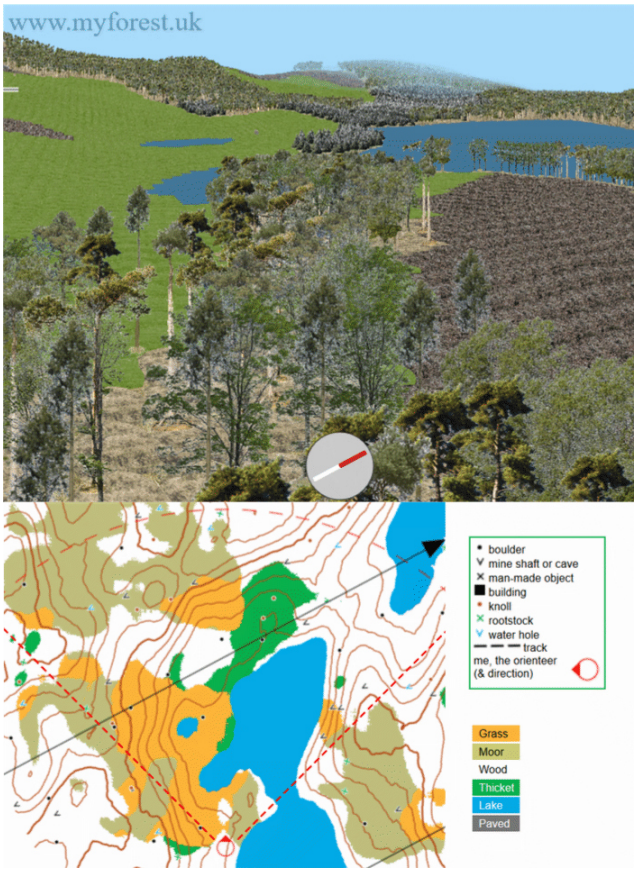
Scene display in The Forest

This document describes how scenes in The Forest (grelf.itch.io/forest) are displayed. The program is written entirely in client-side JavaScript in a single HTML5 page. It uses a 2D canvas for displaying either a detailed map of the terrain or the scene as viewed by an observer standing in the terrain (or sometimes in a helicopter above the terrain). The observer can move around of course and the view is shown in 3D perspective.



Here is an example, with a portion of the corresponding map. The observer's position is shown on the map as a red circle, pointed to show the viewing direction. This example is a helicopter view from 10 metres above the ground.

Notice how the distant part of the scene is hazy. The description to follow will include showing how that is done.

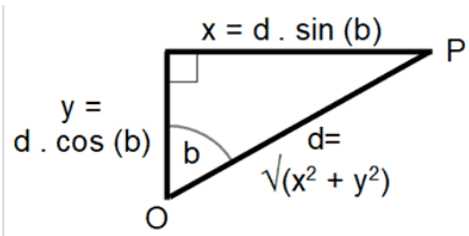
The map is described in a separate document about how the terrain is automatically generated: TerrainGeneration.pdf. It would be useful to have read that first.

Coordinates and geometry

It is important to be clear about the Cartesian coordinate system used in the program.

The observer stands at position (x, y), where x is measured eastwards and y is measured northwards. The units are considered to be metres and on the map 1 pixel represents 1 metre.

The observer is facing in a certain direction and this is measured as a compass bearing, clockwise from due north (the y-axis). When the bearing is displayed it is shown in degrees.



The convention in mathematics is to measure angles anticlockwise from the x-axis. By using compass bearings the program differs from that convention so it is important to realise the consequences, as shown in this diagram. Observer at O is looking towards point P at distance d. Note also the order of parameters for

```
b = Math.atan2 (x,y)
```

Optimise

The program has two constants declared early on, for converting between degrees and radians, so repeated divisions are avoided:

```
DEG2RAD = Math.PI / 180;  RAD2DEG = 180 / Math.PI;
```

Division is a slower operation than multiplication. The speed gains from doing that are small but every little helps when displaying something as complicated as a 3D scene.

Similarly, it is not desirable to keep on calculating sines and cosines. Whenever the observer turns to a different bearing this method of the `Observer` object is called:

```
// Calculate frequently needed fields
Observer.prototype.sincos = function ()
{ this.b = Math.round (this.b) % 360; // Always >= 0, integer
  var brad = this.b * DEG2RAD;
  this.sinb = Math.sin (brad);
  this.cosb = Math.cos (brad);
};
```

What I call the observer is known in other games as the camera but it is where the player's eyes are supposed to be in the terrain. Moving forward is really easy:

```
this.x += s * this.sinb;  this.y += s * this.cosb;
```

where s is the stride length. Repeating strides like this does not require repeatedly calculating sines and cosines.

It is important to look for optimisations wherever possible. Another example is comparing distances. There can be cases where there is no need to take square roots - just compare the squared values. Taking a square root is a slow operation.

Programming style

You may notice that my snippets of code always use `var`. This is partly because much of the code was written before `let` and `const` were added to the JavaScript standard. But I have no problem with `var` and feel no need to change. When something is meant to be constant I indicate that to myself by giving it a name which is all capitals.

Some readers may notice that I use `for` loops more often than is currently fashionable too. This is again partly for historical reasons but also, more importantly, because I want to know exactly in which order elements are being processed.

Overall object-oriented structure

There is one global object called `forest`. In the “onload” function, `init()`, several other major singleton objects are created, of types such as `Terrain`, `Map`, `Observer`, `Scene`. References to these are set as properties of `forest`, using the type name in lower case, so everything can be reached from that global object. To draw a scene, call

```
forest.scene.draw();
```

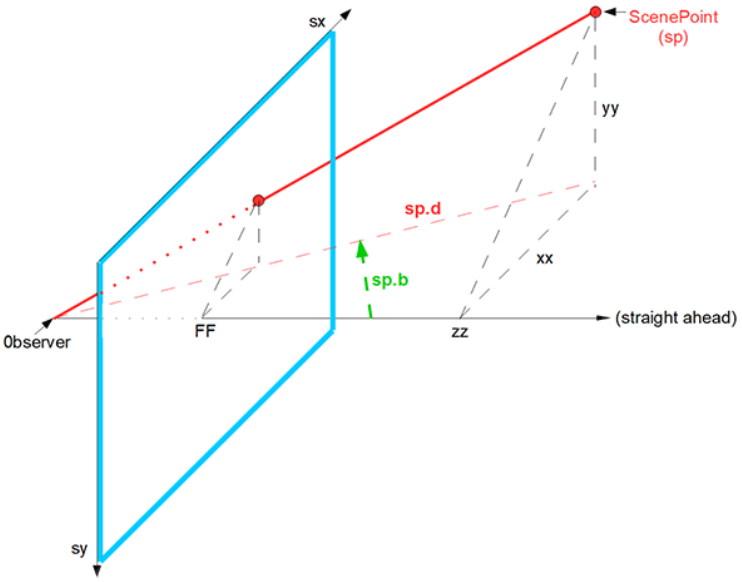
Each object type is defined in a separate JavaScript (.js) file, loaded into the HTML page by its own `<script>` element. Versioning is done by changing suffixes on the js files so they can be cached in the browser unless they change. The browser only has to download changed files.

(My `Map` type was created several years before a `Map` type was added to the JavaScript standard. My version simply hides the standard one, which is not needed but I do use the original hash-table capability of JavaScript objects.)

Perspective

The scene is drawn in 3D perspective, for which a special-purpose algorithm has been devised. The observer is generally looking more or less horizontally, so it is not necessary to have a completely general-purpose, rotating to all possible angles, algorithm. This simplification further optimises performance.

Imagine a transparent screen placed in front of the observer. It is 2-dimensional so it has 2 coordinates but these are quite independent of the terrain's coordinates so positions on the screen will be denoted (sx, sy).



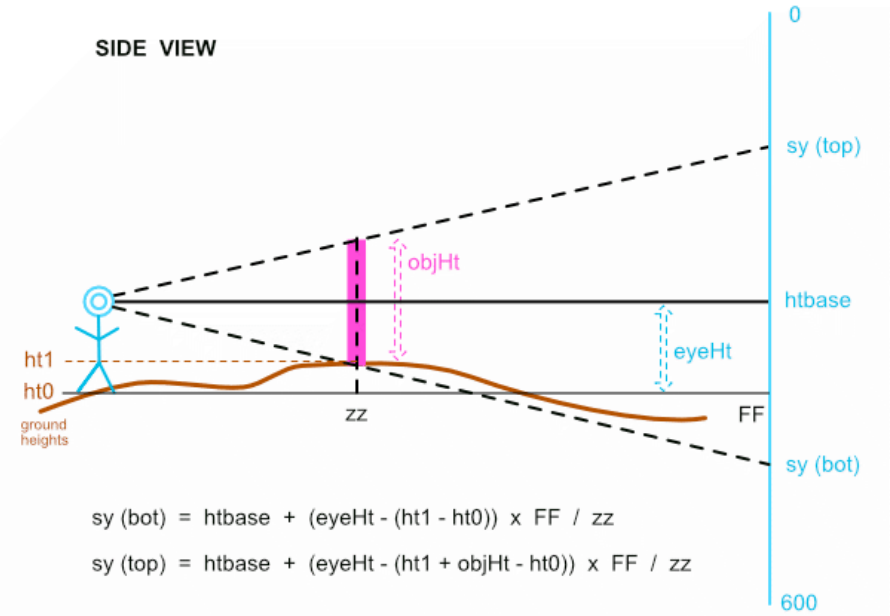
In this diagram the observer is looking through the screen at a point in the scene which we will call sp. This name is chosen because in the program there is another object type, `ScenePoint`. Instances of this type describe points in the scene, of course.

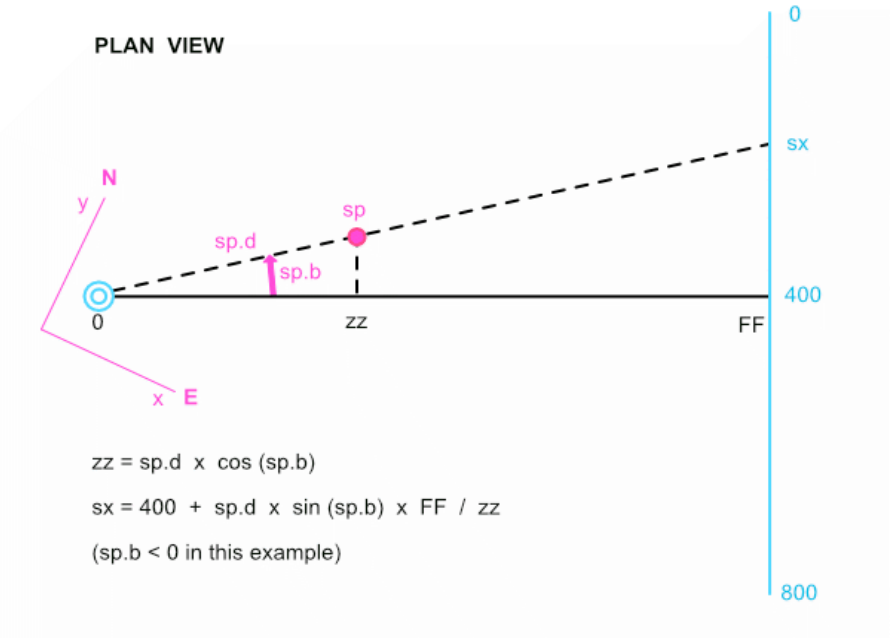
The object sp has properties x and y defining its ground position. It also has d, distance from the observer, and b, difference in bearing from the observer (it might have been better to call this db). The bearing difference is positive in a clockwise direction.

Notice also 2 points labelled on the straight-ahead axis from the observer: FF is the distance to the screen and zz is the distance to a plane parallel to the screen through the point sp.

There are two triangles in the diagram which are outlined by black dashed lines. It is important to observe that the two triangles have exactly the same shape. So the ratio of FF to zz is the same as the ratio of sx to xx and the ratio of sy to yy, if sx and sy are measured relative to the centre of the screen.

It therefore remains to set FF as a constant and calculate zz. The next 2 diagrams indicate how this can be done and work towards our JavaScript code.





These considerations result in the following method of the `scene` object.

```
Scene.prototype.getScreenXY = function (sp)
{
  var ht = sp.getTerra ().height;
  var brad = sp.b * DEG2RAD;
  var sinb = Math.sin (brad), cosb = Math.cos (brad);
  var zz = sp.d * cosb;
  if (zz < 1) zz = 1;
  var fRatio = this.FF / zz;
  var sx = fRatio * sp.d * sinb + this.wd2;
  var sy = this.htBase - fRatio *
             ((ht - this.ht0) * this.YSCALE - this.eyeHt);
  return {x:sx, y:sy};
};
```

There are a few things in this code which have not yet been explained:

1. In the Scene constructor:
   `this.wd2` is the half-width of the screen, so `sx` is relative to the centre.
   `this.obsHt`, `this.ht0` and `this.htBase` are set as in the diagram above.
   Note that adjusting `htBase` enables the observer to look up or down, within a limited range.

2. Scene constants:
   `this.FF` is set as a constant value `400 * Math.sqrt (2)`
   `this.YSCALE` is set as a constant `0.22`
   The last value was found empirically by adjusting it for best results.

3. `ScenePoint` objects have a method called `getTerra()` which returns the terrain properties at the `sp` point. If `forest.terrain.terra (sp.x, sp.y)` has already been called for this point before, its result is stored in `sp` for immediate access instead of recalculating it. This is another performance optimisation.

4. The line about `zz < 1` is to avoid occasional problems with points too close to the observer. Yes, this is a fudge but it works for our purpose.

Field of view

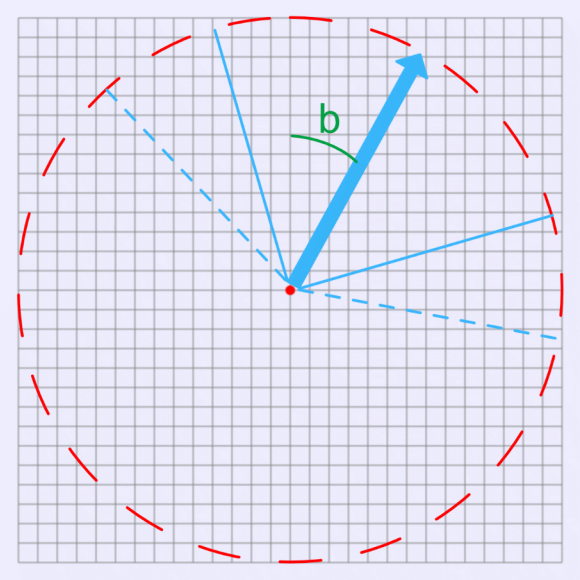
The HTML canvas is currently set as 800 x 600 pixels to give a reasonable drawing time for either a map or a scene. As machines get faster it may be reasonable to increase this size.

Setting `Scene.FF`, the distance of the screen from the observer, to 400 x √2 means that the horizontal field of view is 70.5°.

Visible range

The visible range is adjustable by the user up to a maximum of 400 metres (again, in future it might be possible to extend this but at the moment it takes too long to draw a scene out further, even on fast PCs).

The ground around the observer is treated as a grid of squares of side 1m. We therefore need an array of 801 x 801 `sp` objects out to the 400m range.



The diagram here gives an impression of the situation but with a much smaller grid. The observer is standing at the centre facing a direction on bearing b. The solid blue lines indicate the 70.5° field of view. The dashed blue lines indicate that we need a wider range to allow for objects positioned

just outside the field of view but that are wide enough for them to be seen partially.

The observer's coordinates need not necessarily be whole numbers.

To draw the scene we need to find what is in the terrain at all of the grid positions between the dashed blue lines, but not for the whole grid.

Then they must be drawn starting with the furthest and coming towards the observer. So the grid cells have to be sorted in descending order of distance from the centre.

I was amazed to find, when I first tried this in 2014, JavaScript's array sorting function could do this so fast that a scene out to 100m could be drawn in less than a second. Nevertheless further optimisations have been done since then.

More optimisations

Apart from the sorting I worried about the time taken for so many `sp` objects to be created for building a scene and then to be cleared away by the browser's garbage collector every time the observer moved.

So the first optimisation was to create all `sp` objects that might be needed (801 x 801 of them) and keep them in scope, just reassigning values to their properties whenever the observer moved.

Another improvement was to make another permanent array of grid index values, sorted on distance from the centre of the array. So the sorting no longer had to be done every time the scene was refreshed. This sorting is done in the `init()` function when the program starts. This may be a problem on older and slower machines: the browser may complain that the program is stuck. (I have not received any feedback to suggest that this is really a problem, but then feedback is rare.)

What is drawn at each grid position

First a square tile is drawn, in perspective of course. The 4 corners are run through `scene.getScreenXY()` in order to draw a quadrilateral in the 2D canvas. The shape is filled with a colour which depends on the type of terrain at that position. Then an image is drawn as a ground patch, again chosen for the type of terrain. In a wooded area the patch shown here would be used. It is scaled to cover the tile just drawn. The scaling is so easy to do with the `drawImage()` method of the 2D canvas context.

Notice the chequered background, indicating transparency. All the images are in PNG format with a transparent background. All images are from photographs of my own which I have processed to remove the background (originally in Photoshop but now I use Affinity Photo).

Then in woodland a tree would be placed on the tile. For open woodland (as opposed to a thicket) there are 8 different tree images to choose from and they are not placed centrally on each tile. The woodland is mixed and the trees are not in straight lines.

Selecting and positioning trees

It is important that when an observer returns to any position, the same tree will always be seen and always in the same slightly offset position within its grid tile. This is achieved in a similar way to the terrain generation, by using functions of the (x,y) ground position to produce repeatable pseudorandom effects.

Tree selection is achieved by calculating a group of bits (3 bits allows 8 possibilities) from a function of the x and y coordinates of each point. To make the bits appear to be random the 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). In JavaScript it looks like this:

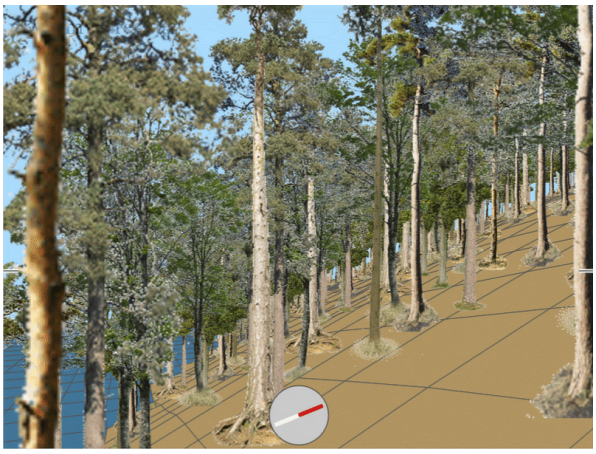
```
var rand = Math.round (PI10000 * x * y) & 0x7;
                          // 3 bits, bit-wise ANDed
```

where the constant `PI10000` has been calculated once at the start of the program:

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

That shifts pi up so we are not looking at its first bits.

The `0x` is not necessary but it is a reminder that we are not interested in decimal numbers here. `0x` means hexadecimal which is a useful way of writing bit patterns. If we needed four bits we would AND with `0xf`.



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. This scene from a test version of the program shows how trees are randomly placed, one per tile, in relation to 2m grid tiles.

The foggy or hazy horizon

It was realised that a foggy horizon would be beneficial because otherwise objects can pop into view too suddenly as they are approached. This is particularly the case when the user's horizon range is short, say 100m or less. Some users will keep the range short because the processors in their devices are not fast enough for a longer range. That means that whatever we write to produce fog must not slow scene drawing down noticeably. Nevertheless the fog has been made optional by means of a check-box in the HTML page.

How fog is represented

The distance to the horizon is initially 60 metres but can be made longer by the user via the drop-down list labelled "Visible range in scene". If the distance from the observer of any object which is to be drawn in the scene is greater than three quarters of the distance to the horizon then the object is assigned a fog factor which rises linearly from 0 at 0.75 of the horizon distance to 1 at the horizon. In the program file, scene.js, this fog factor is called `fogNo` and every `ScenePoint` object in the

arrays which describe the ground near the observer has the property **fogNo** (0..1).

The fog factor determines the degree to which the colour of any pixel in the drawn object is to approach the colour of the sky. When **fogNo** is 0 the pixels retain their original colour but when it is 1 they become the colour of the sky. Intermediate values lie proportionally within that colour range. Each of the components of the colour, red, green and blue, is scaled accordingly.

The scaling is done by two different methods, as described in the next sections.

Buildings and tiles

The method for fogging the walls and doors of buildings and for uniformly coloured tiles such as paving and lake surfaces is relatively simple because they are each drawn by filling a path, created in the graphics context, with a single colour. It is only necessary to calculate the foggy colour, interpolated between the normal colour of each surface and the colour of the sky. This is done in scene.js by the function **fogRGB()**. This takes as input parameters the 3 colour components and the fog factor. It returns a colour style string in the usual hexadecimal format, **#xxxxxx**. The code for this is quite simple:

```
function fogRGB (r, g, b, f) // f=fogNo, 0..1
{
  r += (skyR - r) * f;
  g += (skyG - g) * f;
  b += (skyB - b) * f;
  return makeCssColour (Math.floor (r), Math.floor (g),
                        Math.floor (b));
}

function makeCssColour (r, g, b) // each 0..255
{
  var rs = r.toString (16), gs = g.toString (16),
      bs = b.toString (16);
  if (r < 16) rs = '0' + rs;
  if (g < 16) gs = '0' + gs;
  if (b < 16) bs = '0' + bs;
  return '#' + rs + gs + bs;
}
```

(The standard function **rgb()** might be usable instead of my **makeCssColour()** but this works efficiently.)

Other objects, loaded as images

Objects loaded as images presented some much more interesting design problems.

As of version 18.12.2 there are 30 images to be loaded that represent objects that can appear in a ground-level outdoor scene in The Forest. These begin loading when the **Scene** object is constructed. They comprise about 1.1 megabytes of data.

It is not feasible to apply the fogging pixel by pixel to each image as it is being drawn in the scene: it would take far too long (bear in mind that there really are thousands of image drawing operations, with varying scale factors, to construct a scene). So the images have to be prepared in some way. Also it is not feasible to prepare all possible values of the fog factor. Mathematically there is a continuous range of

values from 0 to 1. Computationally the discrete range of possibilities is still large. So the first decision was to select just 8 stages of fogging on the way from 0 to 1. The continuous value of `fogNo` is changed to integers from 0 to 7:

```
var iFogNo = Math.round (fogNo * 7);
```

This means that we can have an array of 8 versions for each original image. But we do not really want to have to download 8 times as many images. Not only would that be a much larger amount of data but it enlarges and complicates the code to initiate so many downloads.

The function `loadImage()` in forest.js was changed so that the `onload` function creates an 8-element array, called `foggy`, and sets its `[0]` element to refer to this just-loaded image. The remaining elements will be undefined for now:

```
im.onload = function ()
{
  im.loaded = true;
  im.foggy = new Array (8);
  im.foggy [0] = im;
};
```

The next decision was to create the fogged versions of each image only when the need for them first arose. To do this a wrapper method was written to replace each call to `context.drawImage (im, x, y, wd, ht)` (note: the last 2 parameters there are scaled width and height: `drawImage` does scaling very efficiently). The wrapper starts like this:

```
Scene.prototype.drawImage =
          function (im, x, y, wd, ht, fogNo) // fogNo 0..1
{
  if (0 === fogNo || !this.doFog)
          this.g2.drawImage (im, x, y, wd, ht); // unfogged
  else
  {
    var fogged, iFogNo = Math.round (fogNo * 7);
    if (undefined === im.foggy [iFogNo])
       // Fogged version not yet created, create it now:
    {
```

`this.g2` is a copy within the scene object of the graphics context and `this.doFog` is a boolean controlled by a check-box in the HTML page, for whether the user wants the fog effect.

If there is no requirement for fogging then `drawImage()` is called immediately. Otherwise we check whether we have yet got the image with the required amount of fogging, indexed by `iFogNo` in an array called `foggy` that was constructed as a property of each image when it was loaded. If the required image does not exist then this is the time to create it.

Creating the fogged image requires processing each (non-transparent) pixel of the image to interpolate its colour towards that of the sky. We then need the result to be of type `Image` again because we need the scaling capabilities of `drawImage ()`. Scaling cannot be done by `context.putImageData ()`.

After processing the pixels to change their colour for fog we put the image data back

into an image by using `canvas.toDataURL()` and that involves reloading just like the initial loading of an image from file. So the result is not necessarily available immediately for display. Another design decision was therefore to display nothing if the fogged version is not yet ready. On a subsequent call for a version of this image with the same degree of fogging it will be possible to draw the right version. This simply means that the fogged objects appear in stages but it will not be long before fully fogged scenes are drawn as the user moves around.

Here is an annotated version of the finished wrapper method for `drawImage()`:

```
Scene.prototype.drawImage =
          function (im, x, y, wd, ht, fogNo) // fogNo 0..1
{
  if (0 === fogNo || !this.doFog)
    this.g2.drawImage (im, x, y, wd, ht); // unfogged
  else
  {
    var fogged, iFogNo = Math.round (fogNo * 7);
             // iFogNo 0..7

    if (undefined === im.foggy [iFogNo])
    // Fogged version not yet created, create it now:
    { // Get the image data into off-screen canvas:
      var cnv = document.createElement ('canvas');
      cnv.width = im.width;
      cnv.height = im.height;
      var g2 = cnv.getContext ('2d');
      g2.drawImage (im, 0, 0); // Full size
      var imData =
                g2.getImageData (0, 0, im.width, im.height);
      var px = imData.data;
          // Array of pixels (rgba, 4 bytes per pixel)

      for (var i = 0; i < px.length; i++)
      {
        var i3 = i + 3;

        if (0 === px [i3]) i = i3;
        // skip transparent pixel (a == 0)
        else
        {
          px [i] += (skyR - px [i]) * fogNo; i++;
          px [i] += (skyG - px [i]) * fogNo; i++;
          px [i] += (skyB - px [i]) * fogNo; i++;
        }
      } // NB: loop increment skips alpha channel, a

      g2.putImageData (imData, 0, 0);// Back into the canvas

      // LOAD the image data back into an Image object:
      fogged = new Image ();
      fogged.onload = function () { fogged.loaded = true; }
      fogged.src = cnv.toDataURL ('image/png');
                  // Transfer in PNG file format
      im.foggy [iFogNo] = fogged;
      // Do not draw - may take time to load.
      // It will be missing from scene for now
```

```
      }
      else
      {
        fogged = im.foggy [iFogNo];

        if (fogged.loaded)
          this.g2.drawImage (fogged, x, y, wd, ht);
      }
    }
  };
```

Once each fogged version of an image has been created it is likely to be used hundreds of times in drawing each scene.

Why I like HTML5 2D graphics

Working with the HTML5 standard in The Forest has shown me that it is very efficient and effective for my purposes. In particular the following capabilities are extremely useful.

- `drawImage()` not only places a corner of an image at my required position but it also allows me to specify a scaled width and height so that distant objects are drawn smaller than nearer ones, without me having to work out any geometry on individual pixels. I just set the scale in inverse proportion to distance.

- `drawImage()` takes account of 4 channels in every pixel: red, green, blue and alpha, where alpha means transparency: from 0 = transparent to 255 = opaque. This enables me to use images in PNG format which can have transparent areas. Then closer trees placed in front of more distant trees still show parts of the distant trees which should be visible, without me having to calculate this. I use Affinity Photo to prepare my images, cutting shapes out of my own photos and placing them on a transparent background.

- I also do not have to worry about parts of the image going outside the canvas. The system does the clipping as necessary.

- Clipping is even better than that. Any shapes I create as paths can be used for clipping. So, for example, tiles on the ground which I draw in perspective can clip the texture images I draw on them (scaled, etc, using `drawImage()` as usual).

Overall it is a very effective system.

Graham Relf
July 2022