

# Stereograms

Andres Löh

Difficulty: medium

The topic of this assignment are *single-image random dot stereograms* (SIRDS for short). SIRDS are images that contain a “hidden” three-dimensional structure. There is lots of information about SIRDS on the web, including techniques on how to see the hidden images and many examples. See for instance Wikipedia at

<http://en.wikipedia.org/wiki/SIRDS>

for a relatively detailed explanation including lots of links to more information. Figure 1 shows an example SIRDS in black and white that contains a chessboard pattern similar to that in Figure 2. Note that stereograms are sensitive to the resolution, so you have to print the stereogram without scaling, or – if you want to watch it on screen – have to view the file at 100% and with any visual modifications (such as antialiasing) turned off.

There are many variations on the stereogram idea. There are images consisting of just random dots, but also artful pictures where even the two-dimensional variant is a joy to look at. There are images which encode several three-dimensional pictures when looked at in different ways, and there are animated stereograms as well.

In this task, we will concentrate on a really simple algorithm that produces pictures composed of pixels of more or less random colors – in other words, the two-dimensional pictures just look like (somewhat repetitive) noise.

The task consists of the following components:

- Writing a routine to output images.
- Defining a data structure that can maintain ‘links’ between pixels.
- Defining an algorithm that can compute a SIRDS from a depthmap.

Additionally, an algorithm that can (partially) reconstruct the depthmap from a SIRDS to help debugging.

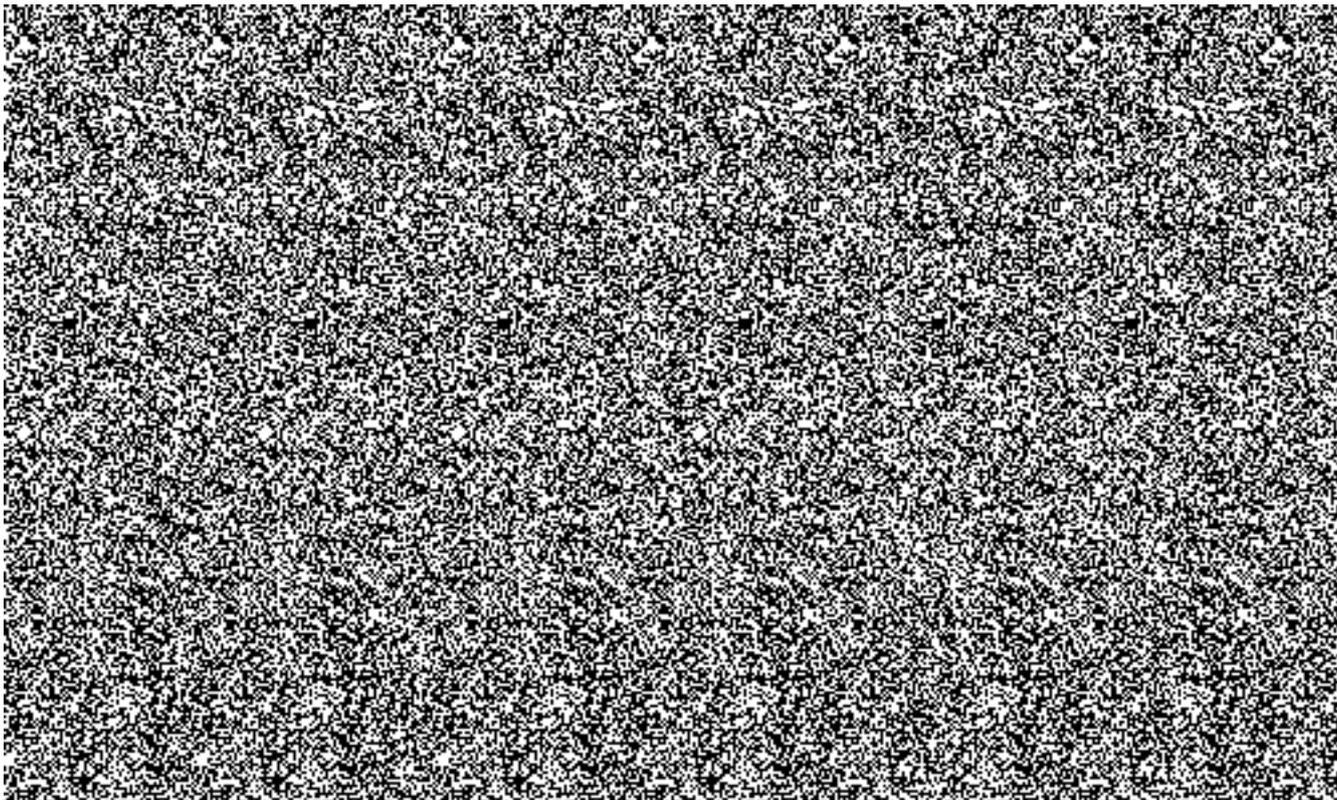


Figure 1: A black-and-white SIRDS

## Images

The first part of the task is to write code that can write images to disk, so that you can view them using an image viewer. We are going to use the PPM format to write images, because it is one of the simplest image formats available. Lots of converters exist that can automatically translate PPM files into more common graphics formats (such as for instance PNG) for you.

If for some reason, you cannot find such a conversion program or viewer for PPM, I recommend that you try to implement a writer for BMP instead, which is another really simple image format, but not quite as simple as PPM.

In essence, each PPM file consists of a header followed by the image data. The header identifies the file as a PPM file, contains the resolution and the color depth of the image. The data is an encoding of each pixel in the image, line by line.

In Haskell, we represent a single pixel by its RGB color, i.e., its red, green and blue color components, where each component ranges from 0 to 255:

```
type Color = Int
data RGB   = RGB Color Color Color
```

Note that we use RGB both as name of the datatype and as name of the single con-

structor. Such double use is allowed by Haskell and quite common. Nevertheless, the constructor `RGB` is a function, whereas the datatype `RGB` is a type. Both exist in parallel.

An image is then a list (rows) of a list (columns) of RGB colors where all rows have equally many columns:

```
type Image = [[RGB]]
```

### 1. Write functions

```
validColor :: Color -> Bool
validRGB   :: RGB   -> Bool
```

that check if a given color or RGB value is valid, i.e., if all colors involved are integers in the correct range.

### 2 (medium). Write a function

```
validImage :: Image -> Maybe (Int, Int)
```

that tests if an image is valid and, if it is, returns the resolution of the picture as a pair of the x-resolution and the y-resolution. An image is valid if all its pixels are valid *and* if all the rows have equally many columns. All these properties should be checked, and `Nothing` should be returned for all invalid images. Example:

```
*Main> validImage []
Just (0,0)
*Main> let p = RGB 0 0 0; r = [p, p, p] in validImage [r, r, r, r]
Just (3,4)
*Main> let p = RGB 0 0 0; r = [p, p, p] in validImage [r, r, r, [p]]
Nothing
*Main> validImage [[RGB 1 2 257]]
Nothing
```

### 3. The next step is to generate the PPM header for an image. For this, you have to define a function

```
ppmHeader :: (Int, Int) -> String
```

The header consists of four components. The components are separated by a space, the whole header is terminated by a newline. The first component is always the string "P6" that is an identification for the PPM file format. The second and third components are the x- and y-resolution of the image, respectively – they are therefore passed as parameter of `ppmHeader`. The final component gives the maximum color value we are using. For the purposes of this assignment, all our images will use colors from 0 to 255, so the final component is the constant string "255". Hint: Recall that you can use `show` to turn an integer into its string representation. Example:

```
*Main> ppmHeader (1024, 768)
"P6 1024 768 255\n"
```

4. Of course, we must also encode the image data. Therefore, define functions

```
encodeRGB :: RGB    -> String
ppmData   :: Image -> String
```

The former encodes a single pixel, the latter encodes a whole image by encoding all the pixels in order and concatenating the encodings (without any spaces or newlines). How is a pixel encoded? For each RGB-value, we generate a string of length 3 with one character for each component. We encode each component by using the character with the corresponding ASCII code. For this, there is a function `chr :: Int -> Char` (defined in the module `Data.Char`). Example:

```
*Main> encodeRGB (RGB 65 66 67)
"ABC"
*Main> ppmData [[RGB 65 66 67, RGB 68 69 70], [RGB 71 72 73, RGB 74 75 76]]
"ABCDEFGHIJKL"
```

Note that some ASCII code correspond to non-printable characters that will be escaped if shown by the interpreter:

```
*Main> encodeRGB (RGB 0 1 2)
"\NUL\SOH\STX"
```

5 (medium). We can now define a function that writes a PPM image to a file:

```
writePPM :: FilePath -> Image -> IO ()
```

Here, `FilePath` is an abbreviation for a string

```
type FilePath = String
```

You will have to use most of the functions defined so far. The function should test the given image for validity. If the image isn't valid, it should write an error message to the screen. If it is valid, however, it should write the PPM encoding of the image (the header concatenated with the data) to the specified file. To write a file, use the function

```
writeBinaryFile :: FilePath -> String -> IO ()
```

that is predefined in the program skeleton.

6. Verify that your PPM writer works. In the skeleton, there are predefined images `chess`, `gradient` and `circular` that you can write to files of your choice using `writePPM`. Do so, then use a converter/viewer to display the files on screen and check that they look as expected (i.e., as in Figures 2–4).

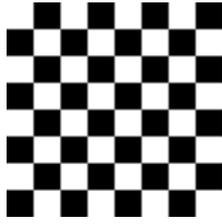


Figure 2: chessboard



Figure 3: linear gradient

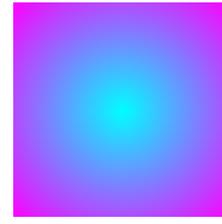


Figure 4: circular gradient

## Links

The basic idea of the SIRDS generation algorithm is that one pixel in the imaginary three-dimensional target landscape is mapped to two pixels – one for each eye – in the stereogram. To make it possible for the eyes to match up these corresponding pixels, such pixels should have the same color. The SIRDS generation algorithm determines which pixels should be ‘linked’ in this way. Let us call the distance between two linked points the ‘length’ of that link. A point in the three-dimensional space that is close to the observer leads to a short link, whereas a point that is further away leads to a long link. The idea is shown in Figure 5.

A pixel can be linked to the left and to the right at the same time. The left and right eye then perceive that pixel as different points in the three-dimensional space.

Things in the foreground tend to be perceived slightly larger than things in the background – therefore, points in the three-dimensional space that are further away are sometimes obscured (shadowed) by other points. In practice, this means that links can collide. Several three-dimensional points might be linked to the same two-dimensional positions. In this case, shorter links (corresponding to closer points) win over longer links.

7. Let us introduce types to represent the links we are interested in. A point (we are only interested in x-coordinates here, therefore points are represented as integers for now) can either be linked with another point, or unlinked:

```
data Link = Linked Int Int | Unlinked Int
```

We maintain the invariant that for the `Linked` constructor, the first point is always smaller than (i.e., to the left of) the second point. We also call the first point the *left* point, and the second the *right* point. Here is a function that tests the invariant:

```
validLink :: Link -> Bool
validLink (Linked x y) = x < y
validLink (Unlinked _) = True
```

Because – as argued above – shorter links shadow longer links, define an operator

```
(>%>) :: Link -> Link -> Bool
```

(read as ‘better’). A link is better than another if it is strictly shorter than the other. A Linked link is always better than an UnLinked point. You can choose any result for comparing two UnLinked points – it does not matter for the stereogram algorithm.

8 (medium). Next, we will implement a data structure that maintains all the links we discover. We use so-called *finite maps* for this purpose. Finite maps are defined in the module `Data.Map`. For instance, the type

```
Map Int Char
```

is a finite map from integers to characters, and can contain a finite number of associations between integers and characters. The first type is also called the *key* type, the second the *element* type. Keys of type integer can be used to look up characters.

Study the Haddock (a documentation generation tool for Haskell) documentation of the `Data.Map` module, available from

```
http://haskell.org/ghc/docs/latest/html/libraries/containers/Data-Map.html
```

For the upcoming exercises, you will need to use several functions from this module, so you should try to understand what the offered methods do and test them in GHCi on example inputs.

The module has been imported into the skeleton program you are working on using the statement

```
import qualified Data.Map as M
```

This makes all functions from the module available for use in your program and also in GHCi, but you have to prefix the names with an `M` – for instance, you have to use `M.lookup` rather than `lookup`. The reason we are doing this is that some functions on lists have the same names as their finite map counterparts, and we want to be able to distinguish between the different versions.

Check at least `empty`, `lookup`, `insert` and `delete`.

Try to understand the types of these functions. Make up example expressions such as

```
*Main> M.lookup 2 (M.insert 2 5 (M.insert 2 3 M.empty)) :: Maybe Int
*Main> M.lookup 3 (M.delete 3 (M.insert 3 6 M.empty)) :: Maybe Int
```

until you feel comfortable working with finite maps. In particular, `M.lookup` is difficult to understand, because it has a rather general type

```
M.lookup :: (Ord k, Monad m) => k -> Map k a -> m a
```

The result type can be delivered in an arbitrary monad, but for our purposes, it will suffice to think of `m` as `Maybe` here. To force this instantiation, use explicit type annotations for the result type like shown in the examples above.

9 (medium). Here is how we store all the links:

```
type Links = Map (Int, Dir) Int
data Dir   = L | R
```

The name `Links` abbreviates a finite map. The keys of the map are pairs of an integer (an x-coordinate) and a direction (L for 'left' or R for 'right'). As explained initially, any x-coordinate can be linked to a point on the left and to a point on the right. If `x` is a specific point, then `(x, R)` is the key for the point to the right, and `(x, L)` is the key for the point to the left.

An invariant is attached to our use of this data structure: if `(x, R)` is mapped to a point `x'`, then `(x', L)` should be mapped back to `x`, and vice versa.

Write a function

```
add :: Link -> Links -> Links
```

that adds a single link to the finite map such that the invariant is maintained (i.e., *two* entries have to be added to the finite map). You may assume that no other links with the same left or right point are in the `Links` data structure at the time of the insertion. Adding an `Unlinked` point should leave the finite map unchanged.

Using the given definition of

```
noLinks :: Links
noLinks = M.empty
```

add a few links to an empty finite map in GHCi and verify that your function is working. Next, write a function

```
del :: Link -> Links -> Links
```

that removes a link from the finite map. Removing an `Unlinked` point should again leave the finite map unchanged. For a `Linked` link, you may assume that the link exists in the finite map. You have to maintain the invariant, though, i.e., you will have to remove *two* entries from the finite map. Test this function as well.

10 (medium). Write a function

```
query :: Link -> Dir -> Links -> Link
```

to query the data structure of links. A call of the form `query (Linked l r) L` or `query (Unlinked r) L` should check if right point `r` is already linked with another left point. I.e., if `Linked l' r` is already in the structure, then `Linked l' r` should be returned. If there is no link in the structure that has `r` as right point, `Unlinked r` is returned. Similarly, the calls `query (Linked l r) R` or `query (Unlinked l) R` check if the left point `l` is already linked with another right point.

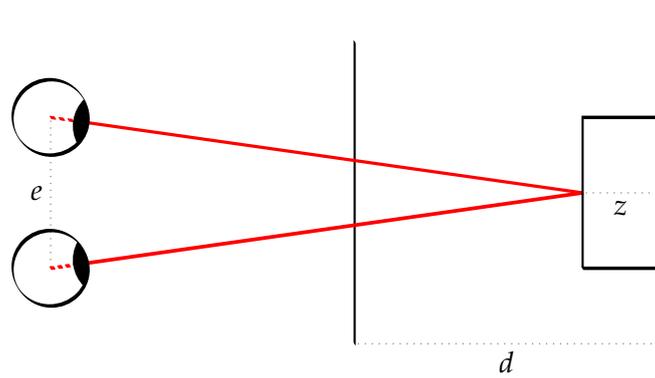


Figure 5: The idea of SIRDS generation

11 (difficult). Write a function

```
link :: Link -> Links -> Links
```

that adds a link `Linked l r` to the structure, but only if it's 'better' (according to `>%>`) than both the links that are returned by querying for `Linked l r` in both directions. If the new link is indeed better, then first the old links should be removed using `del`, and finally the new link should be added using `add`. If the new link is not better or if the new link is `Unlinked`, then the `Links` data structure should be returned without modification.

### Stereogram generation

We are now at a point where we can actually implement the generation of a SIRDS relatively easily. The input to the algorithm is a *heightmap*:

```
type HeightMap = [[Height]]
type Height     = Double
```

This is a list (lines) of lists (columns) of z-coordinates of the same size than the intended resulting image. Each of the heights is supposed to be between 0.0 and 1.0 for optimal viewing results.

12. Write a function

```
separation :: Double -> Int
```

that computes the separation of the two linked points from a given height value. Look at Figure 5. The input of this function is the height `z` of the 3D-image point from the base plane of the 3D-image (grey dashed line in the picture). The output is supposed to be the distance of the two points where the rays from the eyes to that point cross the plane of the 2D-image.

The separation can be calculated using the following formula:

$$\text{separation } z = \frac{e(d - z)}{2d - z}$$

Transcribe this formula to Haskell to define the `separation` function. The parameters `e` (the distance between the eyes) and `d` (the distance between the eyes and the 2D-image, which is equal to the distance between the 2D-image and the imaginary base plane of the 3D-image) are given in the skeleton.

For our purposes, the final result should be an integer, because we want a distance in pixels. Use the `round` function to convert a fractional number into an integral number.

**13** (difficult). Write a function that processes a single line of the stereogram and produces the according links:

```
sirdsLine :: [Height] -> Links
```

The length of the original list determines the width of the line, lets call it `width`. Start with a structure with no links. For each `x`-position `x`, you have to compute the separation `s` corresponding to the height at this position, and then link `x - (s 'div' 2)` with `x - (s 'div' 2) + s` using the function `link`, but *only* if both points are on the line, i.e., at least 0 and at most `width - 1`. The final structure of links is then returned. Hint: It is probably easiest to define an additional help function that has extra arguments for the state you have to maintain, i.e., the width, the current `x`-position and the current `Links` data structure.

**14.** At this point, you are done, because the remaining functions are all given. To complete the job, we make use of the function

```
assign :: Int -> Links -> IO [RGB]
```

that assigns random colors to a single line (the width of the line is the first argument) while respecting that linked points should get the same color. Because `assign` makes use of random numbers, and random numbers are a side-effect, its result is an `IO` type. `Assign` internally makes use of the function

```
findRightMost :: Links -> Int -> Int
```

that, given an `x`-coordinate `x`, finds the rightmost `x`-coordinate that is chain-linked with `x`, and that therefore has to be of the same color.

Finally, there is a function

```
sirds :: HeightMap -> IO Image
```

that performs the complete conversion from a heightmap into an SIRDS. For each line of the input, `sirdsLine` is called to compute all the links. Then `assign` is called to assign colors that respect the links. Study the code of these functions and try to understand it.

You can then try to run the main program. Note that the program will run *significantly* faster if you compile it with optimizations using `ghc -O --make SIRDS.hs`.

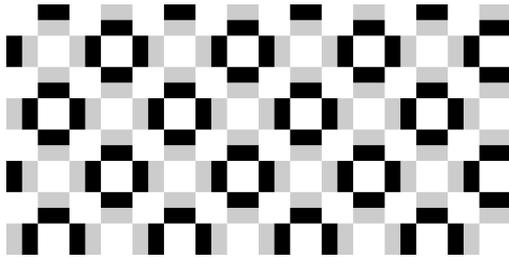


Figure 6: double chessboard pattern

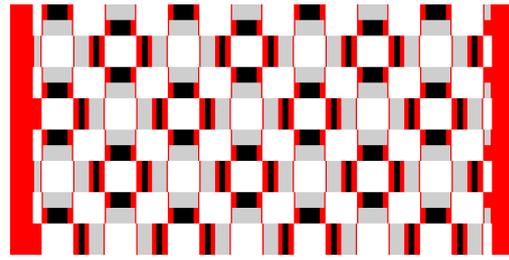


Figure 7: en- and decoded pattern

## Decoding stereograms

I'm aware that not everyone can see stereograms, and that non-working stereogram algorithms can be difficult to debug. I have therefore provided an algorithm that can recover the heightmap from a SIRDS. This is a lossy operation.

The function is called

```
decode :: Image -> Image
```

and takes an encoded (SIRDS) image as input and produces the decoded variant as a heightmap in graytones. Areas that cannot be decoded are printed in red. Red areas will typically appear at the left and right of the whole image, and to the left and right of high points, because those points shadow lower points that are directly adjacent. Here is an example, as performed by the default main routine. The `doubleChess` function can be transformed into a graytone image using the `hightmap` function, resulting in the image shown in Figure 6. If encoded as a stereogram via `sirds` and decoded back using `decode`, you will end up with something like Figure 7.

## Bonus exercises

**15** (bonus, more tricky than difficult). Add a PPM reader, and a converter from full-color PPMs into heightmaps. Then you can generate your own heightmaps with a graphics program of your choice.

**16** (bonus). Write a command-line interface that reads the input heightmap and the output PPM as command-line arguments, and then performs the requested conversion. Look at the `System.Environment` library for the necessary functions to access command line arguments.

**17** (bonus, difficult). Try to implement an SIRDS algorithm that takes a pattern as input, and does not just use random dots.

**18** (bonus, difficult). Try to make the SIRDS decoding algorithm more intelligent, so that it can handle SIRDS that have been generated by other means.