Mission 3: Beyond the Second Dimension

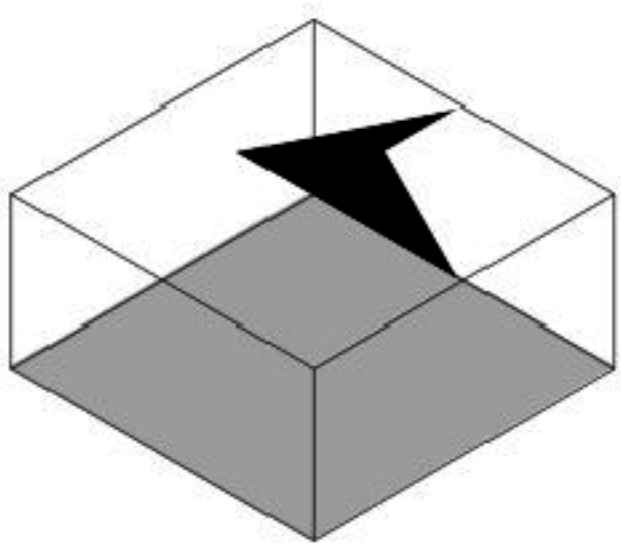
Start date: 28 August 2017
Due: 1 September 2017, 23:59

Readings:

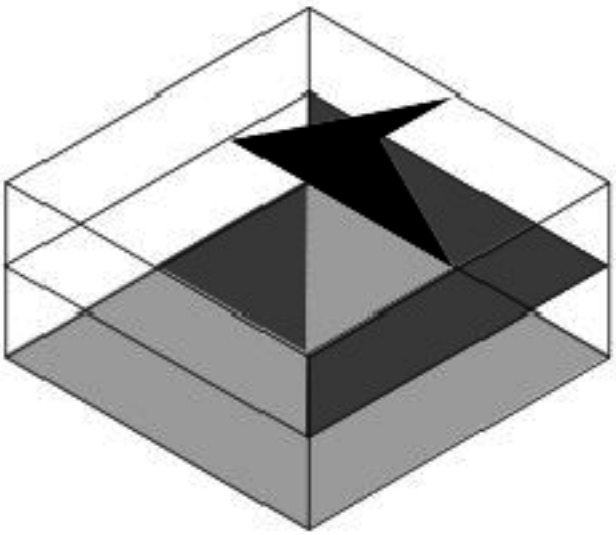
- Textbook Sections 1.1.1 to 1.1.4

In this third mission, you are expected to think spatially and show acute sense of perception. This will prove that you are able to look at problems from different angles and choose the appropriate line of attack.

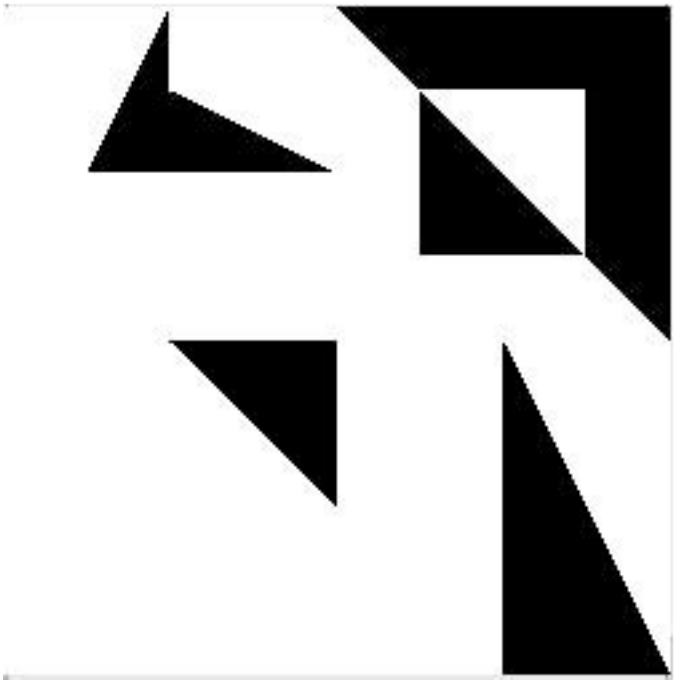
Runes (see previous missions) can be used as representations of 3D scenes. Instead of simple flat runes, the runes can possess surface components of varying depth – also known as depth maps. In this mission, darker areas are closer to you while lighter areas are further away. The following diagram illustrates how such runes – or depth maps – are interpreted. You can imagine the `nova_bb` as a pattern floating on the background:

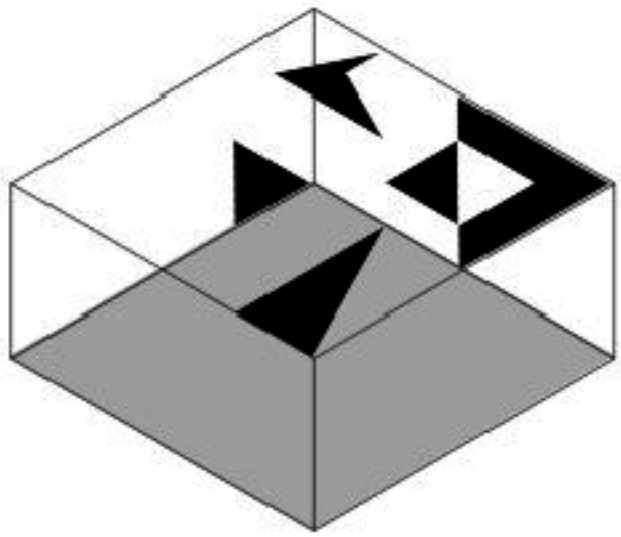


Now, observe a slightly more complicated depth map and its isometric projection. The following depth map shows the `nova_bb` rune in black (darker) and hence is closer to you than the `rcross_bb` rune in light grey.



The next one shows the isometric projection of the `mosaic` rune from Mission 1.





Now that you are familiar with visualising runes as depth maps, try generating your first anaglyph or hollusion.

Anaglyphs provide 3D stereoscopic view of 3D objects with the use of coloured lenses. We'll view them using a pair of red-cyan anaglyph glasses.

You may also try using hollusions, which are an implementation of wiggle stereoscopy, as a simple way of viewing a 3D scene.

Use the functions `anaglyph` or `hollusion` instead of `show` to generate the former instead

of depth maps. For example, to generate the anaglyph for `nova_bb`, we can execute the following program:

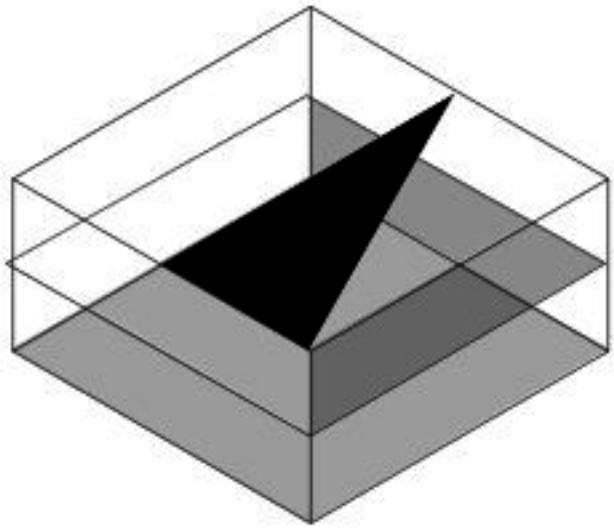
```
anaglyph(nova_bb);
```

Try these new visualisation methods on the runes you created previously.

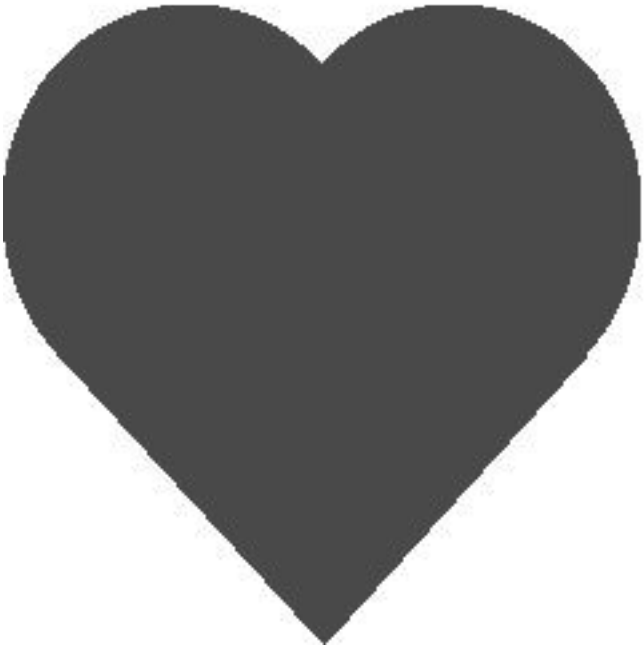
Creating Overlays

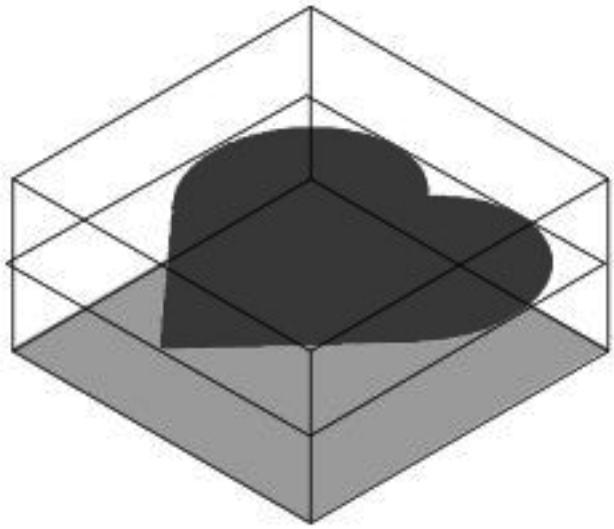
It's nice to be able to generate and view anaglyphs and hollusions, but knowing how to interpret depth maps is not enough to do this exercise. We shall now introduce several more useful transformation primitives. They are: `overlay`, `overlay_frac`, `scale`, `scale_independent`, and `translate`.

The functions `overlay` and `overlay_frac` will help you create depth maps easily. These functions overlay two runes, one on top of the other. With `overlay`, each rune will occupy half of the depth space of the resulting depth map. Figure 1 provides two examples of depth maps produced by overlaying two runes.



```
show(overlay(sail_bb, rcross_bb));
```





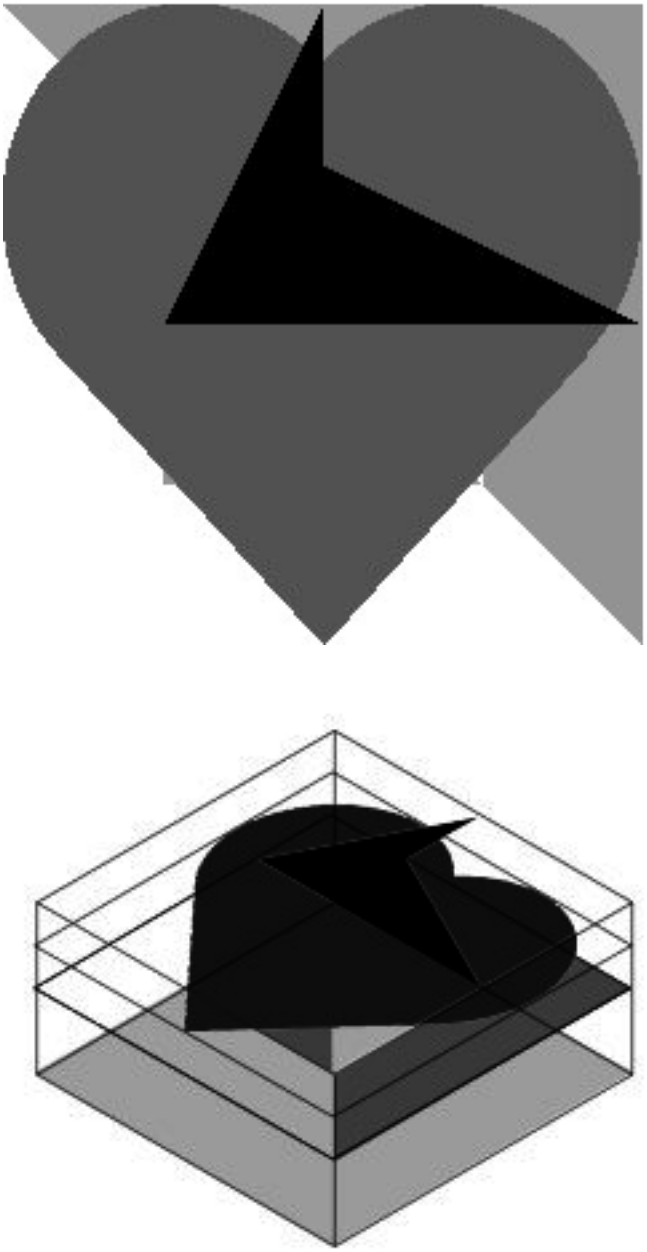
```
show(overlay(blank_bb, heart_bb));
```

Figure 1: Examples of depth maps laid out using ***`overlay`*** *function. Note that the depth space is divided equally between the two runes. In (b), notice that the first rune (occupying the top-most layer) is a blank rune* ***`blank_bb`****.*

To make things more interesting, consider the possibilities should you nest the `overlay` function. It works analogously to `stack` function introduced in lecture. `overlay` squeezes the original depth space occupied by each of the runes by half. For example, doing this:

```
overlay(overlay(nova_bb, heart_bb), rcross_bb);
```

will result in the creation of the following depth map:

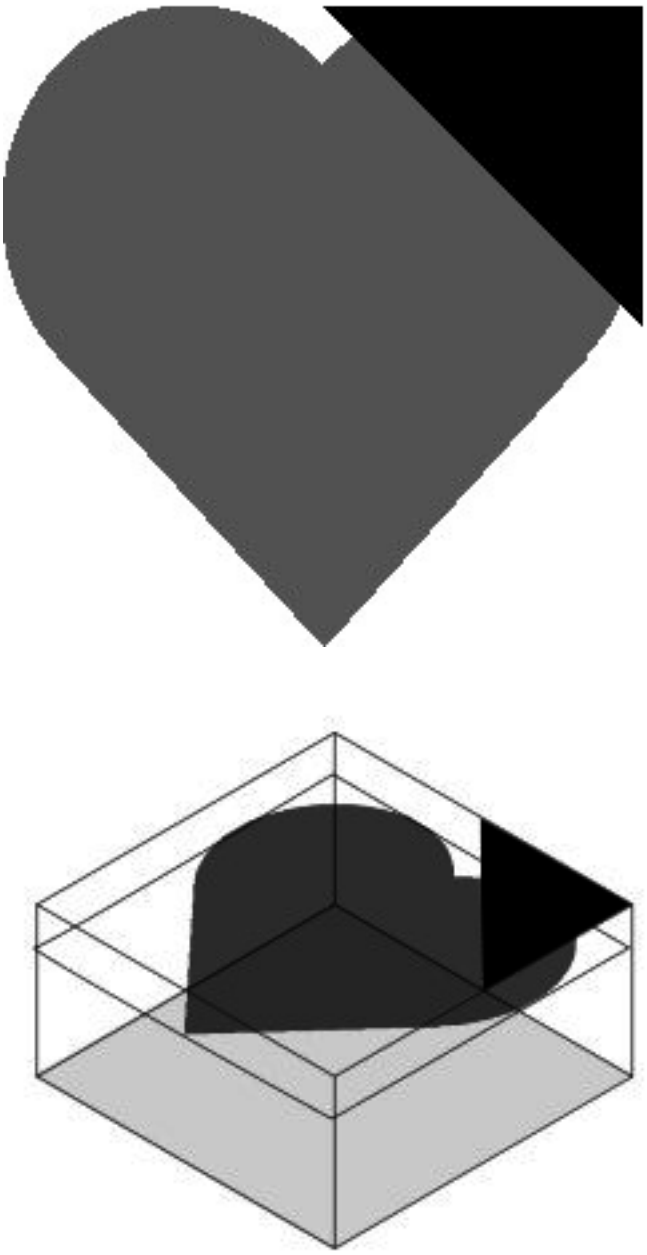


(Hint: use `show(overlay(overlay(nova_bb, heart_bb), rcross_bb));` to display it.)

The first rune, which is an overlaid `nova_bb` and `heart_bb`, occupies the top half of the depth space. The second rune, `rcross_bb`, occupies the bottom half.

The function `overlay_frac` is similar to `overlay`. However, we may also specify the fraction of the total depth space occupied by each rune. This parameter determines the fraction of the depth space occupied by the first rune; the remainder of the depth space will be occupied by the second rune. In the following example, `corner_bb` takes up the top *1/4* of the depth space, while `heart_bb` occupies the remainder *3/4*. (These two functions are so similar that we actually implement one in term of the other. Can you guess which is implemented in term

of which?



```
show(overlay_frac(1/4, corner_bb, heart_bb));
```

We have also defined `scale` and `scale_independent`. The former scales the rune according to the ratio argument, the latter allows greater flexibility as you are able to pass two ratio arguments, one for horizontal scaling and another for vertical.

```
show(scale(1/2, heart_bb));
```

```
show(scale_independent(3/4, 1/3, heart_bb));
```

Figure 2: Examples of `scale` *and* `scale_independent`*. In (a),* `heart_bb` *is scaled by* 1/2* both horizontally and vertically. In (b), it is scaled by *3/4* horizontally and *1/3* vertically.*

The last transformation tool we have is `translate`. The function translates the given rune by the given translation vector *(x, y)* where x and y are ratios of the side of the viewport. Note that positive x means translating to the right, while positive y means translating down.

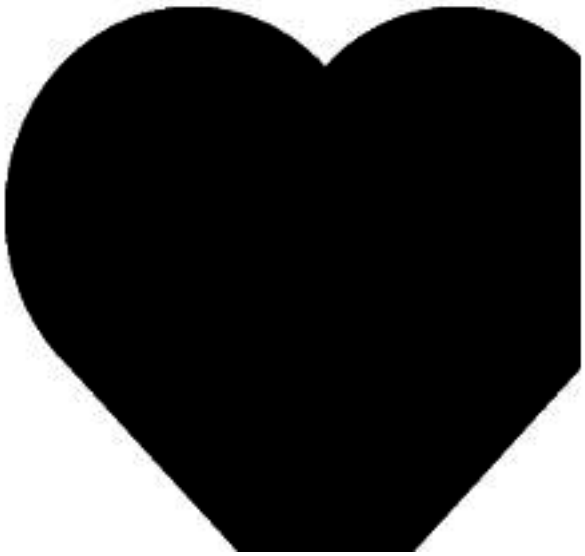


Figure 3: Example of a translated `heart_bb`*. This is the result of* `show(translate(0.1, 0.15, heart_bb));`

Try creating an anaglyph with the following program and view it with your anaglyph glasses: `anaglyph(overlay(scale(0.8, heart_bb), circle_bb));`

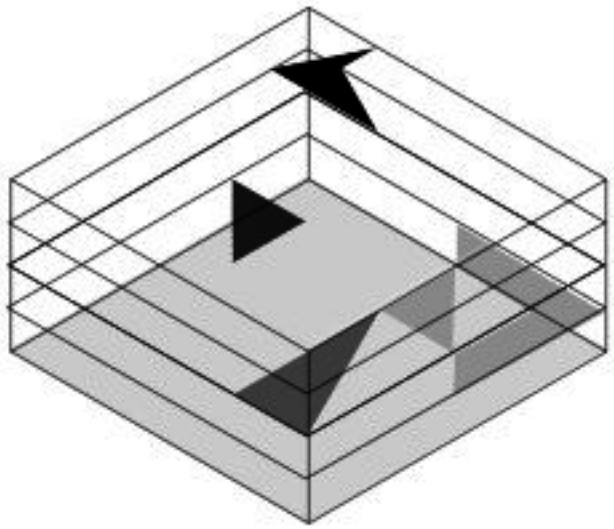
This mission has **two** tasks. **Click here for the link to the template.** Fill up your answers on the template for submission. Also ensure you are using the rune_3d tab on the top-right hand corner.

Task 1:

Write a function `steps` that takes four runes as arguments and arranges them in a *2 x 2* square, starting with the top-right corner, going clockwise, just like `mosaic` in Mission 1. However, the four runes are now placed at different depths as shown in the following example:

```
steps(rcross_bb, sail_bb, corner_bb, nova_bb);
```

will result in the following depth map:



Note that the `rcross_bb` is at the lowest level (lightest shade of grey) and `nova_bb` is at the highest level (black). Also, note that the four runes are spaced equally apart along the vertical z-axis. This means that the 4 images are evenly spaced out vertically.
Hint: You may want to make use of the blank rune `blank_bb` and the mosaic function from Mission 1. If you want to reuse mosaic you will have to redefine it.

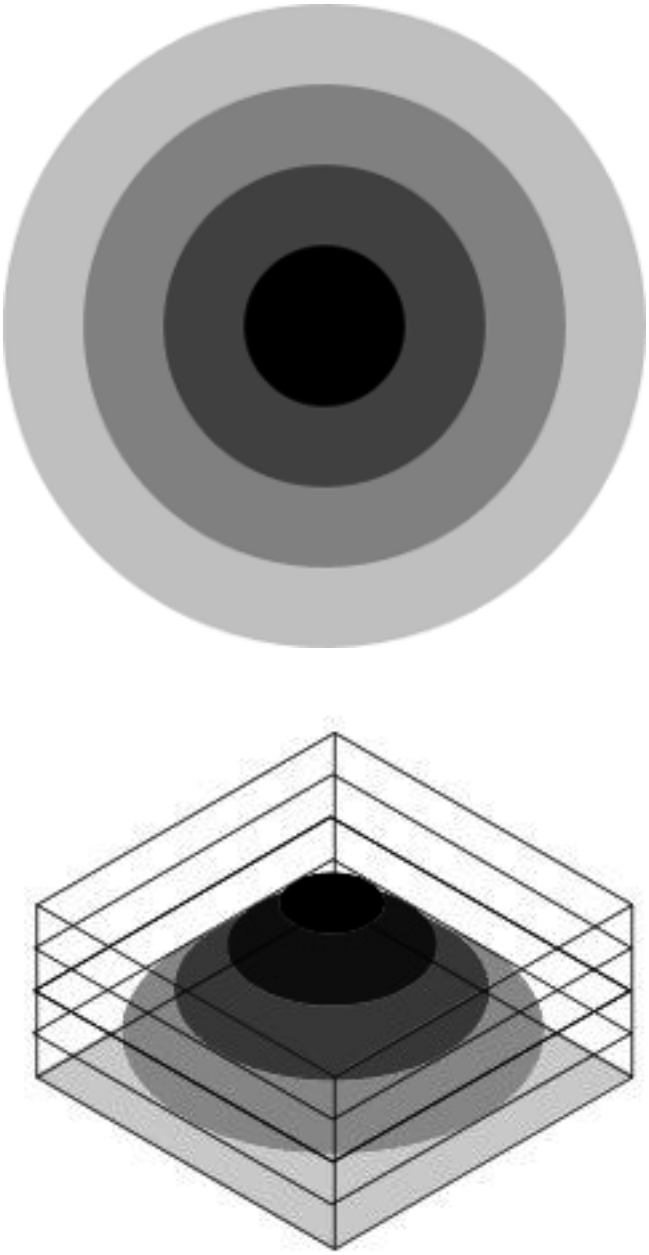
Task 2:

Implement the function `tree` that takes as arguments an integer $n > 0$ and a rune and generates a stack of n runes scaled and overlaid on top of each other.

For example, the following program

```
tree(4, circle_bb);
```

should produce the following depth map:



The generated tree must satisfy a few properties: the `circle_bb` at the top of the tree is scaled to *1/4* of its original size (the tree has 4 layers); the next lower layer is scaled by *2/4*, and so on. Note that the bottom-most layer retains its original size. The different levels of the tree must also be spaced evenly apart vertically.

Submission

To submit your work for this mission, copy the url on your browser and email it to your respective Avengers. Strictly follow to the deadlines set at the start of this file.

"IMPORTANT: Make sure that everything for your programs to work is on the left hand side (Editor) and not in the Side Content! This is because only the programs in the Editor are preserved when opening the url you have emailed to us."

Appendix: Stereograms - A Fourth Way of Achieving 3D Effects

(Read on for your own interest; understanding this is not required to complete the tasks!)

Beside depth map, anaglyph and hollusion, stereogram, or more correctly, Single Image Random Dot Stereogram (SIRDS) is another way to view 3D objects from a 2D screen. You can try this out by calling `stereogram` instead of `show`, `anaglyph` or `hollusion` in the console.

Here we will not delve in detail how stereograms work and are instead going to talk about how to view them. The basic theory behind stereogram creation is very simple, it is a problem of finding groups of points which are constrained to have the same color. In our algorithm, we simply allocate random colors to each group of constrained points, thus earning it the name: Single Image Random Dot Stereograms (SIRDS).

How to view stereograms

Two mechanisms are in play to help us see the world around us: first, our eyes roll inwards towards each other so that each eye points towards the same object (convergence); second, our eyes' lenses adjust to get a clear image (focus). The trick to viewing stereograms lies in decoupling these two mechanisms so that our eyes converge at a point behind the stereogram picture. The eyes should be looking almost in parallel at the picture, but at the same time focused on the picture. This way of viewing stereograms is known as wall-eyed viewing.

One way to view stereograms is to imagine that you are looking at an object some distance behind the stereogram. It feels somewhat like looking at your reflection in the mirror. When you are looking at your reflection, your eyes actually converge at a point behind the glass pane. If you switched between looking at the glass pane itself to looking at your reflection, then you should have gotten a sense of how it feels like to look at an object behind the stereogram. If you can do this, you will see double after some time because your eyes are looking at different places in the picture. Hopefully your brain will put two and two together and voila! you will see a beautiful 3D object.

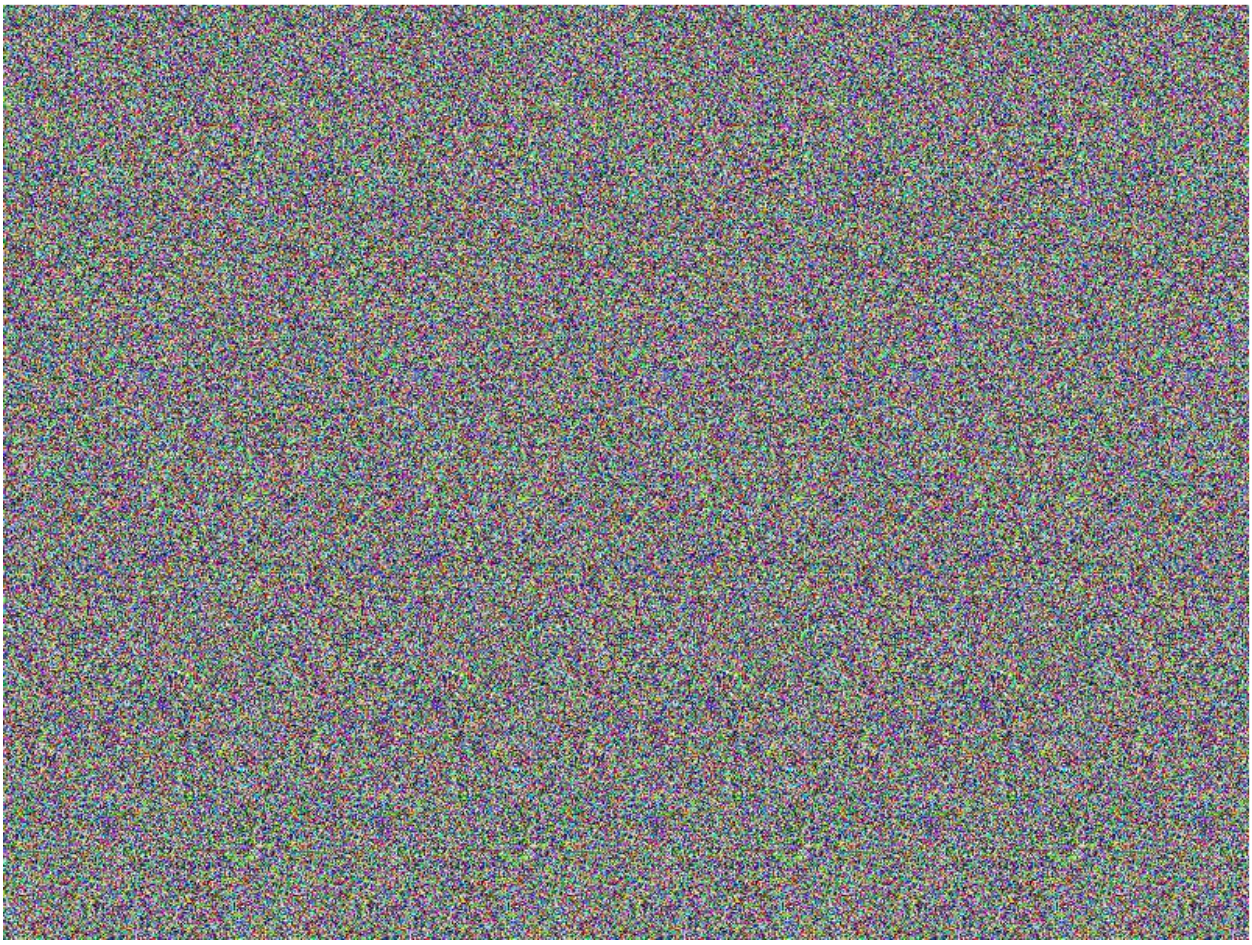
Another popular way to view stereograms is to bring the picture close to your face. It's best to bury your nose into the picture. At such a close distance, your eyes give up on trying to converge upon an object and roll back into wall-eyed viewing positions, just like if you were

looking at an object extremely far away. Now, slowly pull the picture away from your face, or your face away from the picture, but keep the same relaxed sensation in your eyes so that you see double. At some point, you will see your stereogram.

It is not natural for us to view stereograms because decoupling focus from convergence goes against our instincts and serves no useful purpose in daily life. Early humans could ill afford to decouple convergence from focus by mistake just when they were chasing their dinner or running away from a lion!

However, it is not that hard to view stereograms too—it just takes a little bit of practice, like learning how to swim or how a ride a bicycle. Remember to try easy, not hard; viewing stereograms does not involve straining your eyes. Don't worry if it does not come to you naturally—you do not need to be able to view stereograms to do the problem set.

Now, try to view the sample stereogram in the figure below!



A sample stereogram. When viewed with correct technique, a `nova_bb` *appears to "pop out".*