Imperial College London

Individual Project Report

Department of Computing

Imperial College of Science, Technology and Medicine

Op Art Generator and Animator

Author:
Shitian Jin

Supervisor:
Dr. Peter McBrien

June 21, 2022

Submitted in partial fulfillment of the requirements for the BEng Degree of Imperial College London

Abstract

Optical Art, also known as Op Art, is a genre of visual art specifically uses optical illusions to create bizarre artistic effects. Usually op arts are made by artists using periodic arrangement of lines, shapes, colours, etc. By applying optical effect to a flat picture, it can trick viewers' eyes into thinking it is three-dimensional.

Op Art became extremely famous and popular in the 1960s. People used them as design for their clothes, CD covers, houses and so on. Op Art had even been adapted as a television commercial back then.

Regardless of op art being well-known centuries ago, it is still a magnificent form of art bringing vibrant effect and strong emotion to viewers. With the development of computing technology, digital art has become common to public, more and more people are willing to create their artwork using a computer rather by hand.

Despite of existing software that works in artistic aspects, lack of simplicity and professional focus makes Op Art artists difficult to plan and design their artwork based on computer. Therefore, there has been a high demand on designing such program that could aim better on Op Art and give better inspirations to those who work in this aspect.

This report introduces the development and evolution of an Op Art Generator program which help attracting the interest in Op Art from people and giving better understanding in Op Art for professional artists.

Acknowledgement

I would like to express my deepest appreciation to my supervisor Dr. Peter McBrien for providing me valuable opinions and guidance related to the artistic aspect throughout the duration of this project.

I would also like to thank my second marker Dr. Nicolas Wu who also provided me useful suggestions for the researching direction.

I am also grateful to my friends in college, who patiently experienced my program in the early stage and advised me brilliant ideas on how to improve it.

Last but not least, I would also like to appreciate my family for the emotional support they gave me while working on this project.

Contents

1 Introduction

Op art, the abbreviation of optical art, is a style of abstract visual art that draws in 2D uses optical illusion to give viewer a strong impression in 3D.

In early stages, op arts were drawn in black and white patterns to give a strong contrast between blocks, cause human eyes to detect an illusion of vibration, movement, flashing and so on.

Take a look at this optical art painting in Figure 1. The dots changed their widths by columns, they decrease to a minimum at a certain point and increase again, confuse viewers to think it forms into a valley or a peak. The colour gradient between black and white also cheats your brain that it looks like a 3D object rather than a plain drawing on the paper.

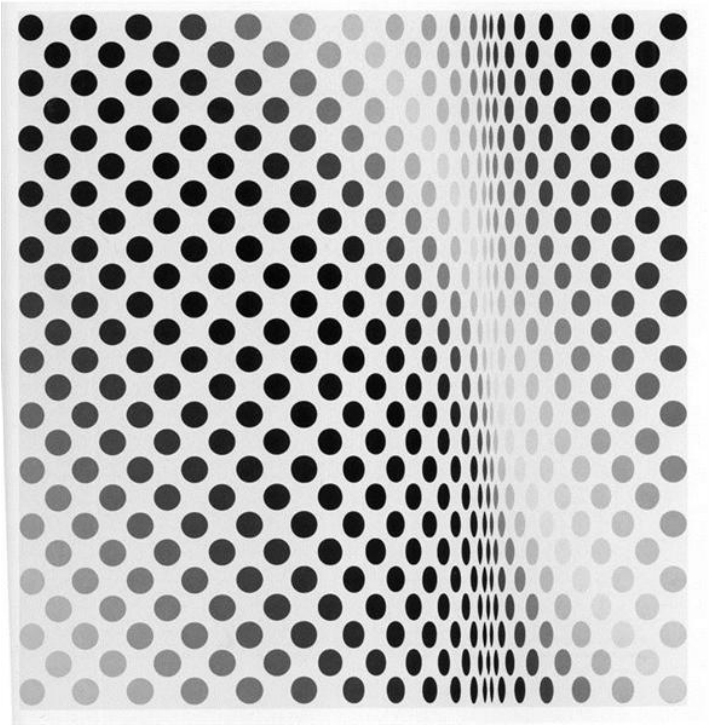


Figure 1: *Pause*, Bridget Riley, 1964 [1]

1.1 Motivation

Op Art has a long history, its aesthetic attracts so many artists to work in. However, drawing is always a time-consuming process and it usually take an artist few days or even few months to complete a single artwork. Qualities can also be affected by hand drawing, lack of imagination from creators and many other human factors.

With the development of technology and computer software, it is possible to work on digital art using computer graphics nowadays. Since computers can do perfect math calculations, logical patterns, randomness and so on that human are not good at, artworks especially op art can be done much better compare to traditional painting by hand. It also allows normal users who know nothing about art participate into op art and enjoy the process of creating an image by him/herself.

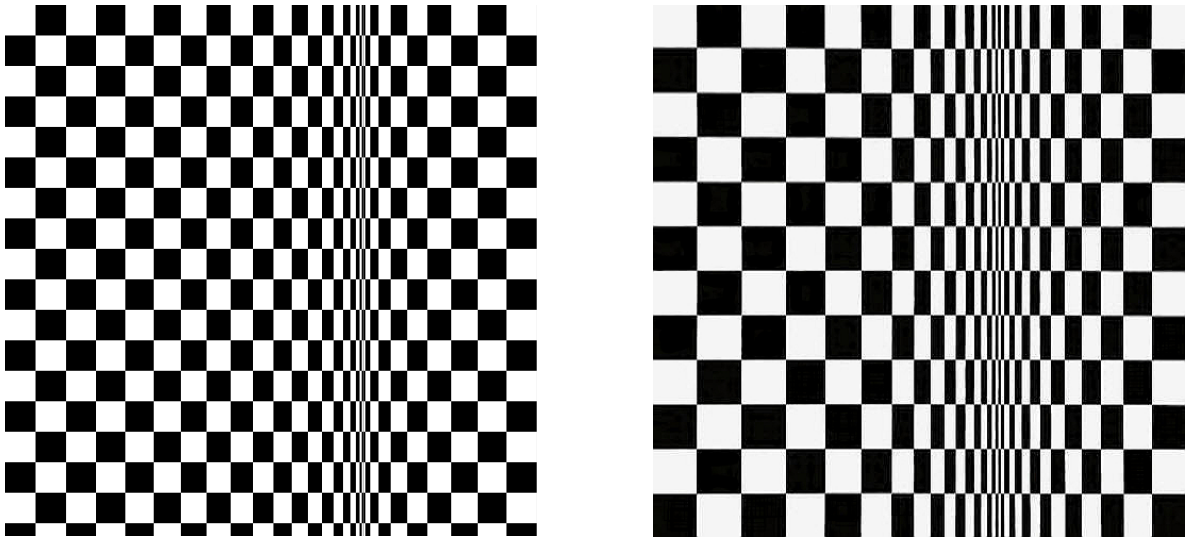
Currently, there have been plenty of art generator software in the market. Some of them are quite useful and easy to use, but still lack some functions and focus on op art.

1.2 Op Art Generator and Animator - example

This project aims to create a program that can generate op art with certain algorithms by users without knowledge of art.

In this project, an op art generator has been created using Python and its image library Pycairo [2], a Python module that provides binding to Cairo graphics library, a powerful 2D graphics library written in C. A GUI is designed to make user-friendly using Flask framework and deployed into website using Heroku.

Down below is a sample of a result image from op art generator compare to Bridget Riley's artwork *Movement in Squares* in 1961.



(a) Result from self-designed op art generator

(b) *Movement in Squares*, Bridget Riley, 1961 [3]

Figure 2: The op art generator with reference to its original work

2 Background

2.1 History

Although the term 'op art' first came out in 1964 in Time magazine to describe a style of abstract art done by an American painter Julian Stanczak [4], this form of art has already started to appear for many years.

Op art is believed originated from painting theories of Bauhaus design school in Germany around 1920s. After it was closed in 1933, lecturers include Josef Albers, one of the great abstract painters moved to America and developed op art concept further there. [5]

It was almost at the same time that the Hungarian-French artist, Victor Vasarely, also known as "grandfather" and the leader of the optical art [6] has created one of his earliest optical art *Zebra* in 1937. This painting uses black and white stripes in two different patterns to represent two zebras, give a feeling that two layers exist in this image to viewers.

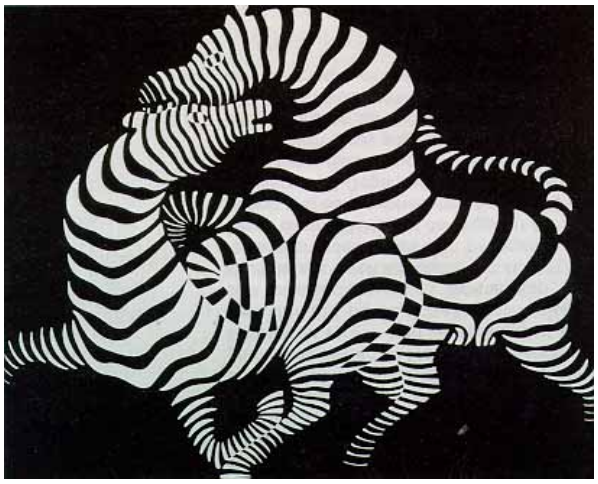


Figure 3: *Zebra*, Victor Vasarely, 1937 [7]

Op art was not well-known to the public until 1965, when an art exhibition called *The Responsive Eye* was held at New York's Museum of Modern Art (MoMA).[8] A variety of ranges of artworks, including those from British artist Bridget Riley were exhibited and later became famous. During the following years, op art became fashionable and people even use them as images for commercial contexts.

2.2 Bridget Riley

Bridget Louise Riley was born on 1931 in London, UK. She is famous for op art paintings and one of the recent op art she drawn is a poster for London 2012 Olympic Games called *Rose Rose*[9]. She now lives and works in London, Cornwall and France.

Bridget's father ran his own business of painting, which may affect Bridget's career of becoming an artist. She was an art teacher for children in her early career and later worked in several art colleges.[10]

Bridget began to paint in a semi-impressionist style until she changed to pointillism in 1958, which uses small, distinct dots of color to form the painting [11]. During the year 1960 she evolved a new form of art called op art and started to work on it until now.

2.3 Existing tools

1. **Photo editing software** such as Adobe Photoshop

A powerful image editing software that can do almost anything from basic to professional.
Advantages
Various functions for user to apply to photos or start from scratch. Features like filtering, after effects, colouring, watermarking, etc. can be applied. Users can draw op art using libraries from Adobe Stock and gain really nice images.
Disadvantages
Both Photoshop and Adobe Stock needs a subscription and may not fit for users that just want an experience specifically on op art, since photoshop can do much more than just creating op art. High occupancy of the software also means it could perform poorly on devices with average specs and hardware.

2. **Online drawing tools** such as Pixel Art Maker
A web based pixel art maker by colouring in pixels.
Advantages
The tool has a user-friendly interface and is simple to start with, accurate pixel location gives a nice feature to create a certain shape with high contrast colours so to create a nice op art image.
Disadvantages
The tool does not focus on how to make op arts, it is only a generic pixel art creation tool and using such needs certain level of art knowledge.

3. **Mobile apps** such as starryai
A mobile app that can generate artwork by artificial intelligence given information from user.
Advantages
Easy access on phones and personalised user experience. Artwork can be generated by AI by just giving the app some descriptions. No drawing technique and user skill is needed to create nice pictures.
Disadvantages
Still lack of functionalities of op arts generation, this application mainly aims for users that need inspiration in art.

4. **Image filtering software** such as Pop Art Studio
A program that can be installed locally on the computer, applying various filters to an original image user imported.
Advantages
Huge amounts of filters for user to pick and customise their images and can then be saved as files in a flash second. The software also contains an online version for cross-platform support.
Disadvantages
Only a few filters that generate op arts can be found in this program such as simple shapes and dots. It also requires to subscribe to get more features such as saving images without watermarks.

5. **Bespoke programs written by individuals**
Examples include a program written using CodePen.IO by a front-end web developer and UI designer Jorge Moreno. Users can generate exact same style of op art painting that looks similar to *Movement in Squares* by Bridget Riley in 1961.
Advantages
Easy to generate op art images as the author focuses on doing so. A single click with randomness in the program can generate op artworks without any knowledge of art.
Disadvantages
The program only focuses on generate one type of image, there could also be copyright issues if someone want to reuse the piece of code to develop further.

In summary, the project needs to provide features on generating op arts that the existing tools above cannot offer to users, such as:

- Provide an environment for users that can generate various types of op art in one platform.
- Simple user interface, users can input variables using sliders, combo boxes etc. to generate op art images in seconds.
- Images generated can be easily downloaded and saved locally in formats such as PNG, JPG and SVG files, in this way they can be accessed easily for various productive uses.
- Furthermore, sequences of images can be generated by changing variables gradually, and animations can be generated giving a further impact to human eyes.

Before we start on developing a program that can fulfill the points mentioned above, let's take a look into some op art paintings and analyse them in a mathematical way. By researching mathematics behind art, it gives us a better understanding on creating an op art generator and animator.

3 Mathematics Behind Art

Since we are focusing on designing a program that generate op artworks in a digital manner, we need to analyse the original artworks by hand and try to find the mathematical equations, geometric shapes and colour patterns behind them.

Below are several examples that have been converted into a mathematical way and applied in the op art generator and animator program:

3.1 Parallelogram

3.1.1 Sample

Here are two pictures that Bridget Riley accomplished in year 1964 and 1993 respectively. This op art is painted in colour using small pieces of parallelograms aligned perfectly in diagonal. Especially in the drawing *Fete*, zebra-like colour patterns are also applied which using two colours changing in the same column to achieve a nice visual effect.

(a) *Fete*, Bridget Riley, 1989 [12]

(b) *Nataraja*, Bridget Riley, 1993 [13]

3.1.2 Shape

We first need to know the position for each four vertices of the parallelogram. To do this, we assume the original point is at the top-left corner of the shape. In a specific example as shown in Figure 5, this refers to start at point **A** in parallelogram **ABCD**.

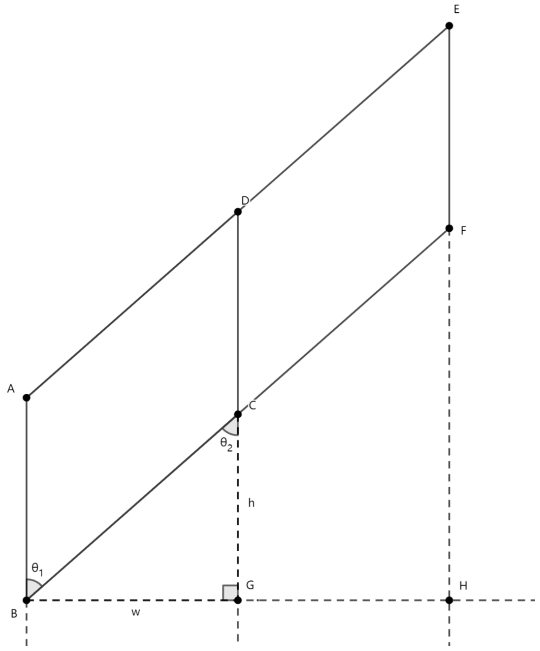


Figure 5: Analysis of parallelogram

To adjust the image, we need to set up some input arguments, this should better include:

1. The angle θ of the parallelogram. $\theta_1 = \theta_2$ since **AB** is parallel to **CD**.
2. The height of each parallelogram, which is length of **AB** or **CD**.
3. The width of each stripe needs to be, which is the width of each parallelogram as well, shown in the figure as segment **BG**.

Point **B** then moves downwards a value of argument 2, then **C** needs to move right a value of argument 3 first (now at the same point as **G**). By applying trigonometry to the graph, we can easily calculate how much point **C** needs to move upwards to form the third point, since:

$$tan\theta_2 = \frac{w}{h}$$

We can get the value of h easily by given angle (argument 1) and width (argument 2), and hence the four points of the parallelogram can be fixed by giving a starting point and three input arguments above.

Since the image is drawn with several columns and rows of parallelogram, I decided to generate it using a double loop, first record position of vertices of each row of parallelogram in one column, and then repeat the same steps for every columns (print the image from left to right). Figure 6 is the initial output that generated using this strategy.

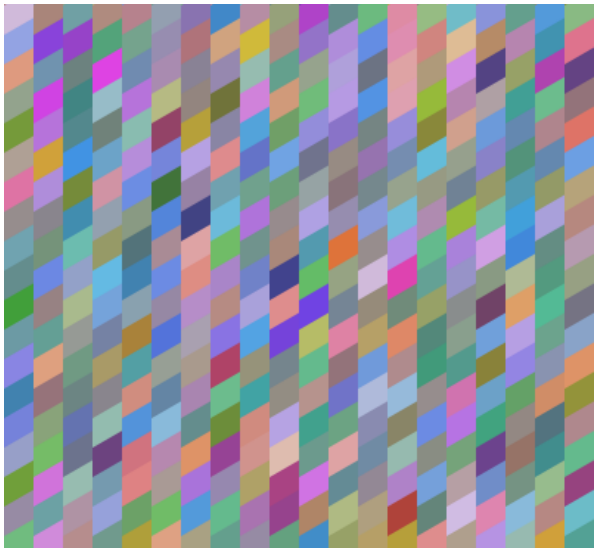


Figure 6: Initial output of parallelogram

There occurs to be a problem that in each column, the topmost parallelogram starts at the same horizontal level hence it looks quite different compare to the original artwork. The shapes are not aligned at the diagonal level. Back to Figure 5 this means **B**, **C** and **F** needs to be in the same line. To fix this, for each outer loop another offset needs to be applied to move the starting point up a bit and get them aligned. Here in Figure 5, a value of h needs to be applied.

After applying the offset, the image becomes normal and aligned, shown as Figure 7.

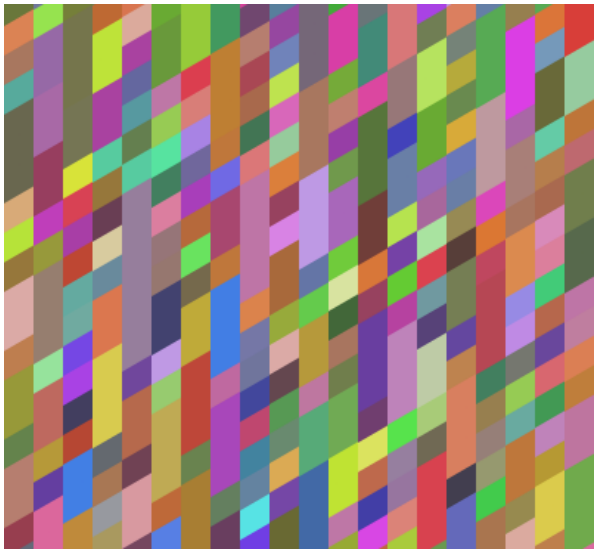


Figure 7: The result after applying an offset for each column starting point

3.1.3 Colour

As the images shown above, the program uses fully random generator for RGB values of colour. Although this brings randomness and uncertainty to the artwork, in Bridget Riley's original work she applied many zebra-like colour patterns and long parallelograms to give viewers an impact.

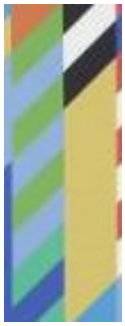


Figure 8: The zebra-like pattern (left) and long parallelograms (right) in artwork *Fete*

To achieve this effect, two further input arguments are added to the program:

1. **adjacentProbablity**. The probability that same colours will be filled to the next parallelogram, by printing parallelograms in same colour, long parallelograms can be achieved equally.
2. **adjacentList**. A list of integers for numbers of adjacent colours to appear in one occurrence, decided randomly.
3. **zebraProbability**. The probability that zebra pattern will occur. Using two random generated colours and fill in a zebra pattern.
4. **zebraList**. A list of integers for numbers of zebra pattern colours to appear in one occurrence, decided randomly.

Both *adjacentProbability* and *zebraProbability* are ranged from 0 to 1 inclusively for percentage of probability. This gives the image long parallelogram and zebra pattern features with a certain generating rate.

Here is the final image output:

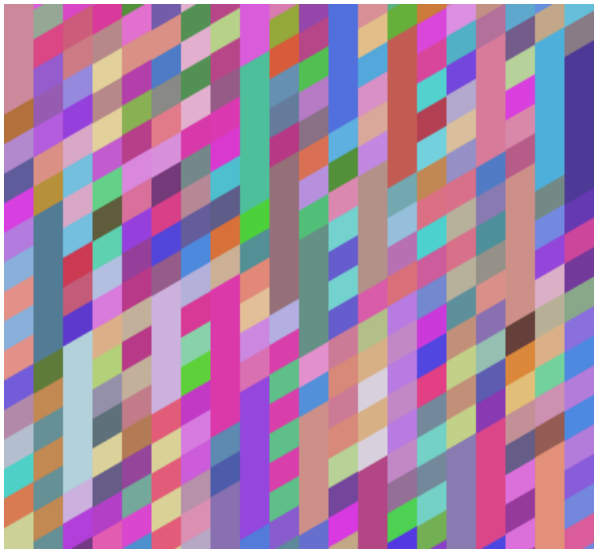


Figure 9: Final output of parallelograms with zebra pattern and long parallelograms

3.1.4 Improvement

In this op art image, more settings can be applied to make the image looks more similar to the original. The zebra patterns can be fixed to appear in certain locations rather than random, and colours could be fixed using highly contrasted ones to give a strong vibrant appearance to the viewers.

3.2 Normal Stripes

3.2.1 Sample

In the Figure 10 down below, stripes with same widths are arranged from top to bottom with five colours: black, white, blue, orange and red. Using a different colour pattern this can create a nice effect.

Figure 10: *RA 2*, Bridget Riley, 1981 [14]

3.2.2 Shape

Shapes in this picture is quite straight forward. By giving a single argument **width** of each stripe, the program can then output the image using a single loop by printing left to right.

3.2.3 Colour

In the initial plan, the five colours are chosen as the original one. Each stripe is then being filled randomly. The initial output looks like Figure 11 down below.

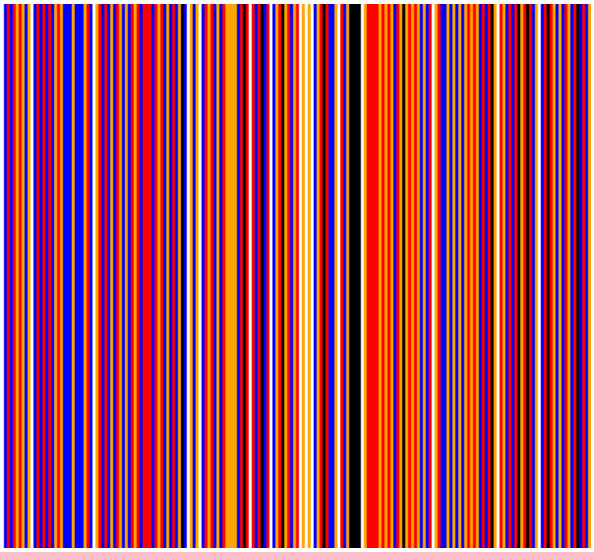


Figure 11: Initial output of stripes

We can see some of the colours are the same adjacently and the stripes do not look even and elegant compare to the original one.

After analysing Bridget Riley's artwork further, we can find out there are less black and white stripes than other coloured ones. So the possibility of filling the stripe with black or white needs to be less compare to other colours. To achieve this, a weight for each colour is added when select randomly. Furthermore, a while loop is added to prevent repeated colour so each stripe would look unique to each other.

Algorithm 1 Add weight for colours and prevent repetition

```
colour of previous stripe ← colour of current stripe
weights ← [0.32, 0.32, 0.32, 0.02, 0.02]
while previous = current do
    current ← random with weights
```

In the code above, both black and white will have a chance of only 2% to be filled in. The output would then look like below, in Figure 12.

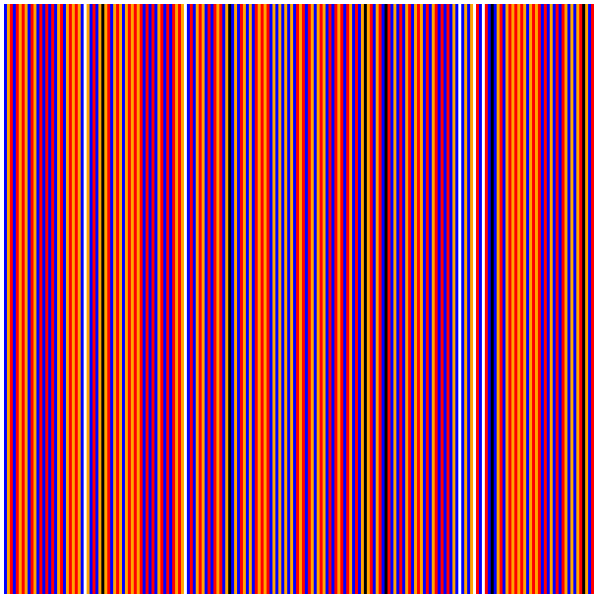


Figure 12: Stripes with weighted colour probability and no repeat

3.2.4 Improvement

Although the output looks much better after some adjustments in the program, the stripes still look too narrow compare to original hence the width for stripes could be considered further in this image. Black and white stripes can also have a certain numbers of other coloured stripes between them and get a fixed location rather than random.

3.3 Stripes with Increasing Width

3.3.1 Sample

Now consider further about the stripe pattern but only using two colours and gaps to distinguish between each other, using an increasing width approach would give viewers a strong stereoscopic feeling. Figure 13 below is an example drawn by Bridget Riley using only blue and red stripes and increasing width gradually to the middle.

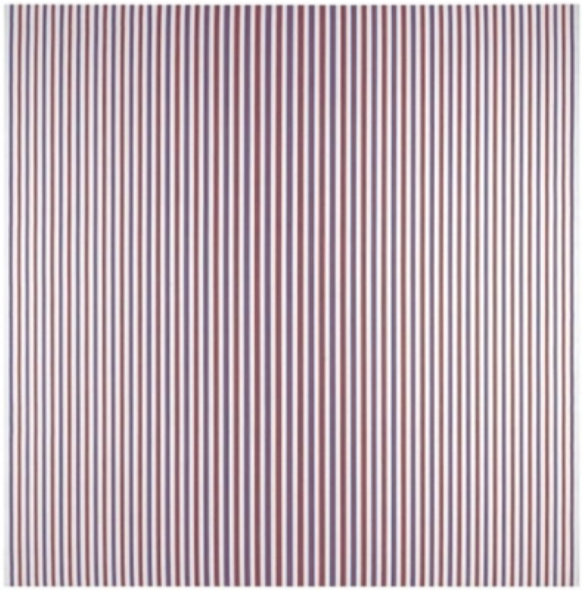


Figure 13: *Chant 2*, Bridget Riley, 1967 [15]

3.3.2 Binomial Distribution

- **Shape**

To generate this image, I first came up with an idea using binomial distribution to represent the widths of stripes at different position.

To achieve this, following arguments are needed in the program:

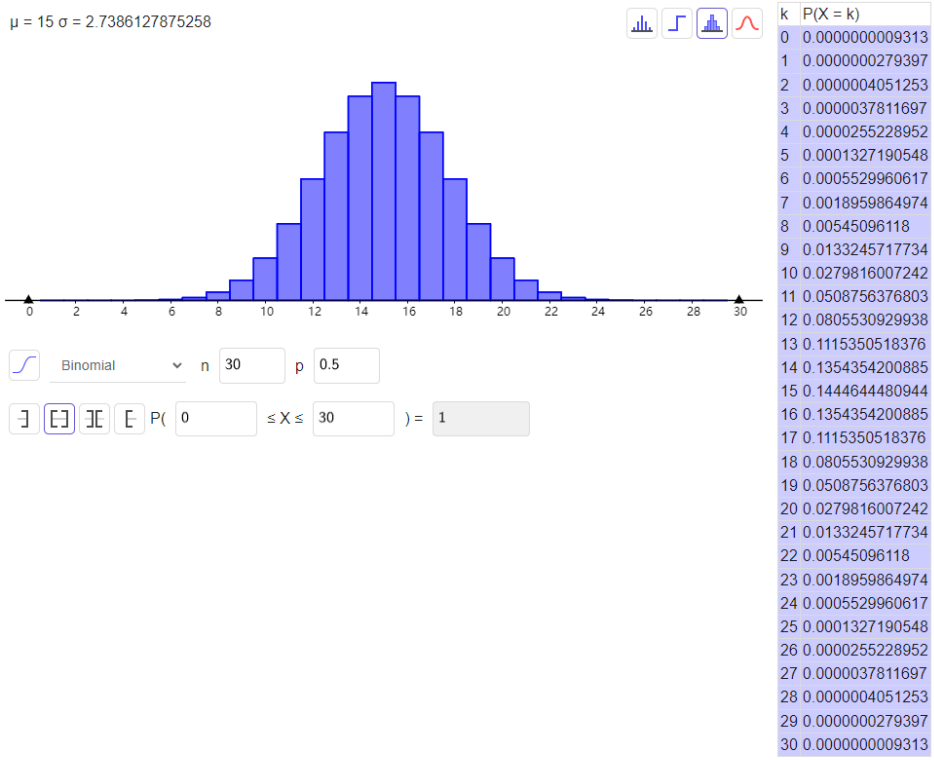
(a) The total number of stripes **n** in the image.

(b) Width of gaps between stripes (gaps are spread evenly).

The argument n could then be applied to the following binomial distribution formula, note that here $\boldsymbol{N = n - 1}$ since we are dealing with stripe index starts at 0:

$$P(x) = \binom{N}{x} \cdot p^x (1-p)^{N-x}$$

Now lets give an example to generate an image with n=31 stripes, since a symmetric image is required, we manually set the probability of success p to be 0.5. The binomial distribution graph and table would look like Figure 14 below.



k	P(X = k)
0	0.0000000009313
1	0.0000000279397
2	0.0000004051253
3	0.0000037811697
4	0.0000255228952
5	0.0001327190548
6	0.0005529960617
7	0.0018959864974
8	0.00545096118
9	0.0133245717734
10	0.0279816007242
11	0.0508756376803
12	0.0805530929938
13	0.1115350518376
14	0.1354354200885
15	0.1444644480944
16	0.1354354200885
17	0.1115350518376
18	0.0805530929938
19	0.0508756376803
20	0.0279816007242
21	0.0133245717734
22	0.00545096118
23	0.0018959864974
24	0.0005529960617
25	0.0001327190548
26	0.0000255228952
27	0.0000037811697
28	0.0000004051253
29	0.0000000279397
30	0.0000000009313

Figure 14: The binomial distribution with N=30, p=0.5

By analysing the graph and result table above, we can see a sudden increase in the middle and low probability values at both ends, which is just what we want to generate the same style of image by applying widths of stripes using values of probability *P(x)*. Since stripe widths can only be integers, a further process to round up *P(x)* is needed.

Now we consider creating the binomial distribution table as list. In Python program, this can be done using **binom** in **scipy** library.

Algorithm 2 Create a binomial distribution table

```
n ← number of stripes
p ← 0.5
data ← [0 .. (n - 1)]
distribution ← []
for i = 0 to (n - 1) do
    append (binomial pmf value at data[i] with shape parameters n and p) to distribution
```

Here we created an integer list *data* ranges from 0 to (n - 1), where *stripes* is the input argument represents number of stripes **N**, and traverse the list by applying binomial function onto each element in the list, we finally created a binomial distribution list *distribution*.

Since the float values in the list cannot be applied to widths of stripes, we need to scale the elements in list further, done like below:

Algorithm 3 Scale the stripe widths

```
stripews ← []
for i = 0 to (n - 1) do
    d ← distribution[i]
    if d · amplitude ≤ 1 then
        append 1 to stripews
    else if d · amplitude > maxwidth then
        append maxwidth to stripews
    else
        append round(d · amplitude) to stripews
```

In the pseudocode above we have two other input arguments: **amplitude** and **maxwidth**, used to scale the value into **d*amplitude** and control the maximum width it can get respectively.

- For all those values that are smaller than 1, we directly append width as 1 into the new list *stripews*.
- For those exceeds **maxwidth**, we append the maximum width instead.
- For the rest values, we append the rounded integer to make sure *stripews* is consisted of all integer elements.

After finishing the width construction we are then able to set the width of each stripe in each single loop. Now lets consider the colour it needs to output.

- **Colour**

Since the original artwork uses alternating colours of red and blue for stripes that gradually changes their widths, we can simply set a variable to represent index of each stripes and then set each stripe with **even** or **odd** index to these two colours respectively. In each loop after printing red or blue colour stripes, gaps can also be treated as stripes filled with white colour. Their widths can be defined from the input argument in the previous section(1•b).

- **Result**

After defining both shape and colour properties, the image output is produced as Figure 15 below.

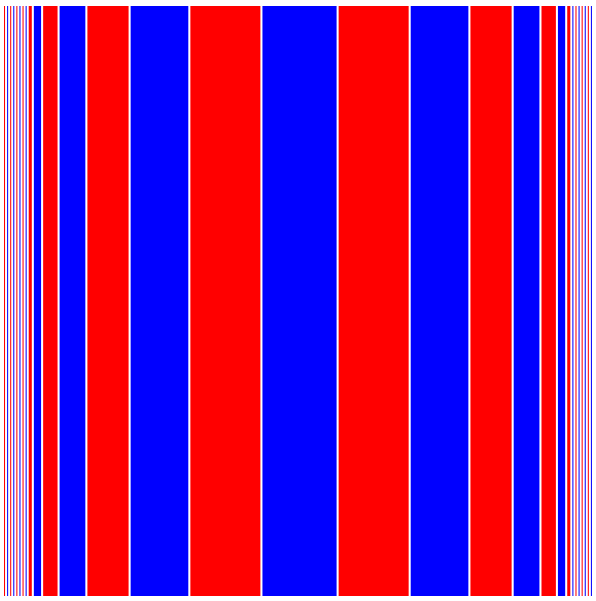
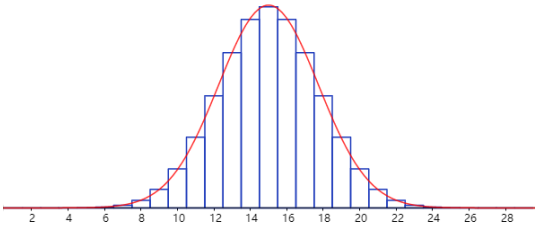


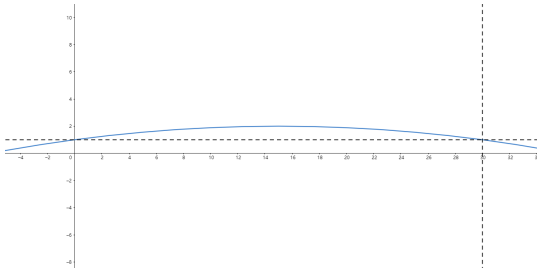
Figure 15: Image output using binomial distribution on stripe width

We can see a huge difference between this image and the original artwork, the stripes are too narrow compare to the ones near the middle. This is caused by the natural property of binomial distribution as it has a subtle increase in the middle with little change on each end. Although strategies like round up and max out have been applied, the increase rate still remains a main problem using binomial distribution.

To solve this problem, a new strategy needs to be applied to make sure stripe widths are changing, but relatively slow. A **parabola** may done quite well.



(a) Binomial Distribution



(b) Parabola

Figure 16: Property and difference of the curve shape, both when N=30

3.3.3 Parabola

- **Shape**

Parabolas are generated using quadratic equation, it has a smooth curve property compare to binomial distribution. By adapting a similar technique to represent the width of each stripe, the only difference is now we are using the equation below:

$$f(x) = a \cdot (x–h)^2 + k$$

where the parabola crosses vertex *(h, k)* with a compress rate of *a*.

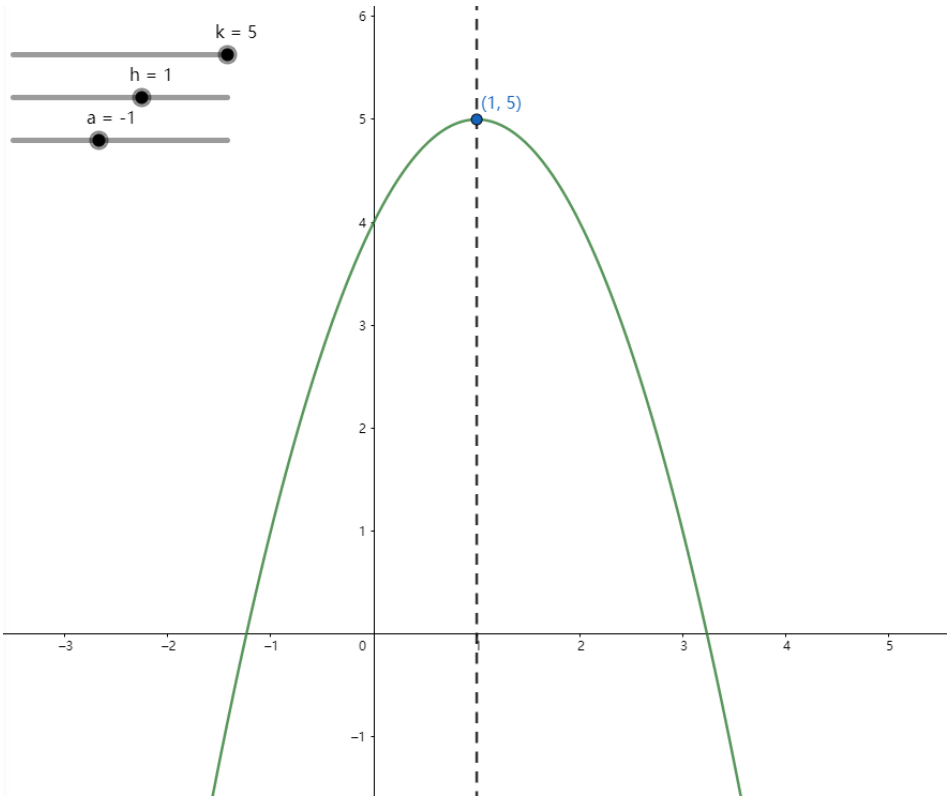


Figure 17: An example of parabola when a=-1, h=1, k=5

In this program, following input arguments are needed to generate parabola:

(a) **minWidth**, the minimum width of all stripes defines the y-interception of the parabola.

(b) **maxWidth**, the maximum width of all stripes defines the vertex's y value: k of the parabola.

(c) Number of stripes **n** defines the vertex's x value: h of the parabola. It is also the midpoint of (0,0) and (n-1,0) since the parabola is symmetric within this period.

Now consider the same example where we have N = 30 stripes (n = 31), we can easily get h = 15 and $k = maxWidth$. To get value a, since we know the parabola passes y-interception *(0, minWidth)*, we substitute the value into x and y respectively, and we get:

$$a = \frac{minWidth - maxWidth}{((n-1)/2)^2}$$

We can valid that a < 0 as *minWidth* - *maxWidth* < 0 so the parabola's shape is curving downwards.

Here are the corresponding pseudocode below:

Algorithm 4 Create a quadratic equation table

```
n ← number of stripes
c ← minimum stripe width
k ← maximum stripe width
h ← (n − 1)/2
a ← (c − k)/h^2
quadratic ← []
for x = 0 to (n - 1) do
    append a · (x − h)^2 + k to quadratic
```

After generating a sequence of values from the quadratic equation, since the values are controlled more accurately and spreaded evenly than binomial distribution, we can just round up each element in the sequence to integers:

Algorithm 5 Scale the stripe widths

```
stripews ← []
for i = 0 to (n - 1) do
    q ← quadratic[i]
    append round(q) to stripews
```

we then get the final list *stripews* for each stripe's width.

- **Colour**

  We apply the same strategy in colouring as we have done in Binomial Distribution (3.3.2).

- **Result**

  After the parabola strategy has been applied, the output provides Figure 18 below.

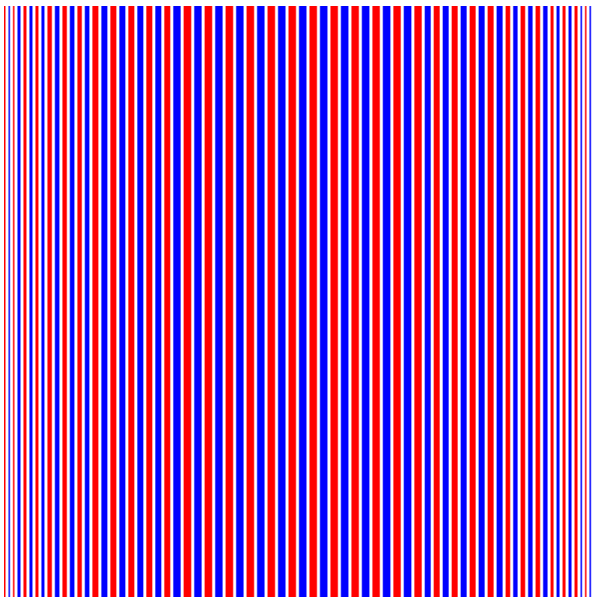


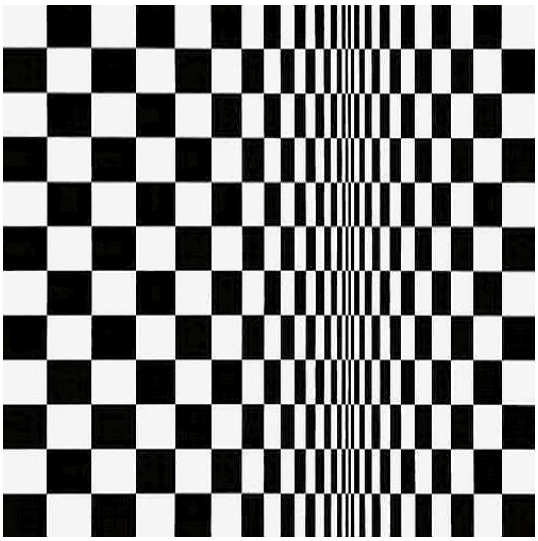
Figure 18: Image output using quadratic function on stripe width

From this image, we can see the stripes spread much more evenly and the image looks quite similar to the original artwork. The image gives viewer a strong and vibrant impact and works well in the op art aspect.

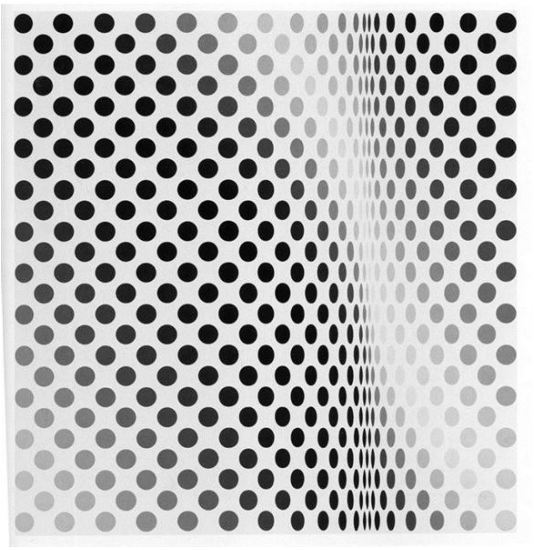
3.4 Double Quadratic Equation

3.4.1 Sample

In Bridget Riley's whole career, she has completed several artworks that have similar styles like below.



(a) *Movement in Squares*, Bridget Riley, 1961 [3]



(b) *Pause*, Bridget Riley, 1964 [1]

Despite of they are constructed of different shapes - squares on the left and dots on the right, they share a common pattern: if we take a look at each picture from left to right in **horizontal** direction, the width of each shape gradually decreases to a minimum point (we can call it a valley or a peak), and then gradually increases again to maximum. They don't need to be symmetric, as the increasing and decreasing rate of widths are different. If we take a look in **vertical**, the shapes in the same column have the same widths.

For some viewers, they may think the shapes are dented back, hence formed a valley, for others may think they've been pointed outwards hence formed a peak. Furthermore, the widths of the shapes are changing in such a way that feels like they formed curvy surfaces on two sides, therefore the picture can also be treated like an open book (or it facing downwards, in opposite). The check pattern on the left picture gives a strong vibrant view and the gradient white stripe on the right make viewers feel like the painting is twisting dynamically.

3.4.2 Shape

Regarding of the previous attempt I have done in creating an image of stripes with increasing width, I decided to use parabola again to represent the curvy changing pattern in *Movement of Squares*. Since two different quadratic equations are used for both sides of this kind of artwork, I decided to name it, in generic, *double quadratic equation*.

They have the same formula as the one in Parabola section of Stripes with Increasing Width (3.3.3), only difference is here we have two, i.e.

$$f(x_1) = a_1 \cdot (x_1 - h_1)^2 + k_1$$
$$f(x_2) = a_2 \cdot (x_2 - h_2)^2 + k_2$$

Let's imagine the double quadratic shape is a real 3D shape and the viewers are looking at its top surface, if we are trying to figure out what the width for each shape at a certain column is, we can look at its side to get the knowledge of how tall it is at each point, hence the distance to viewer's eyes when facing upwards.

With a basic idea like this, the side should look something similar to Figure 20 below, which shows how to achieve the width for each shape.

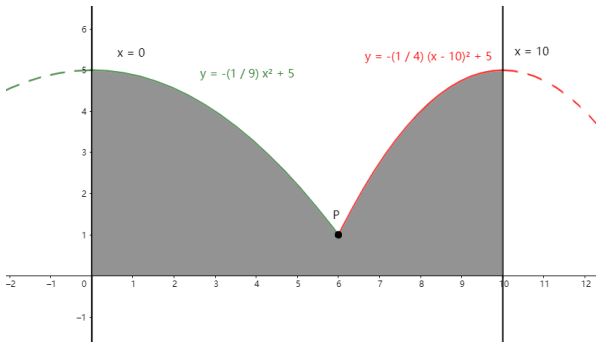


Figure 20: Two Quadratic Equations meet at point P(6,1)

To make things clear and simple,

- treat the left parabola as $f(x_1)$ and the right parabola as $f(x_2)$.
- define the shapes at the leftmost column of the generated image having an index x = 0 and reach to the maximum x value at the shapes at the rightmost column. So the value of the x-axis in the above figure represents the index of the shapes from left to right, here the maximum column is 11 (when x = 10).
- let y-axis represents the width of each shape at a certain index with x value, this corresponds to the width increasing and decreasing rate perfectly.

In this program, we require following input arguments to generate double quadratic equations:

1. **nums**, the number of columns of all shapes defines how long x-axis is going to be.
2. **minWidth**, the minimum width of all shapes defines the y value of the intersection point P.
3. **maxWidth**, the maximum width of all shapes defines the y-interception value of the parabola on the left (crosses left border of x = 0), and the y value when the parabola on the right crosses the right border of x = *nums* - 1.
4. **valley**, the column where minimum width reached defines the x value of the intersection point P.
5. **offset**, an integer power that applies to the y value to achieve a final value of the width for each column (more arithmetic details later on).

Since we want the width on the left side of x = *valley* decrease only and on the right side increase only, the left parabola $f(x_1)$ needs to have a vertex with x value less than 0 and the right parabola $f(x_2)$ with a vertex x value greater than (*nums* - 1), for convenience, let's just put their vertices on x = 0 and x = (*nums* - 1) respectively.

In short,

- $f(x_1)$ has vertex (0, *maxWidth*) corresponds to (h_1, k_1)
- $f(x_2)$ has vertex (*nums* - 1, *maxWidth*) corresponds to (h_2, k_2)

Finally, to get the compress rate a_1 and a_2 for the two parabolas, we just need to substitute their intersection point P with position value (*valley* - 1, *minWidth*), hence we get:

$$a_1 = \frac{minWidth - maxWidth}{(valley - 1)^2} \quad a_2 = \frac{minWidth - maxWidth}{((valley - 1) - (nums - 1))^2}$$

Here we can also see when $valley = \frac{1}{2} \cdot (nums + 1)$, $a_1 = a_2$ hence two parabolas are symmetric to x = *valley* - 1.

Here are the corresponding pseudocode to create two quadratic equations, the process is very similar as in the previous section:

Algorithm 6 Create two quadratic equation tables

$n \leftarrow$ number of columns $(nums) - 1$
$c \leftarrow$ minimum column width
$k \leftarrow$ maximum column width
$l \leftarrow valley - 1$
$h_1 \leftarrow 0$
$h_2 \leftarrow n$
$a_1 \leftarrow (c - k)/l^2$
$a_2 \leftarrow (c - k)/(l - n)^2$
quadratic1 $\leftarrow []$
quadratic2 $\leftarrow []$
for $x_1 = 0$ to l **do**
    append $a_1 \cdot (x_1 - h_1)^2 + k$ to *quadratic1*
for $x_2 = (l + 1)$ to n **do**
    append $a_2 \cdot (x_2 - h_2)^2 + k$ to *quadratic2*

Since values stored in the two lists *quadratic1* and *quadratic2* are non-integer numbers, we need to apply an algorithm and convert them into whole numbers. Notice we need them to be non-zero or the column won't appear, rounding them to nearest integer does not appear to be ideal, using math ceiling here could be a nicer solution.

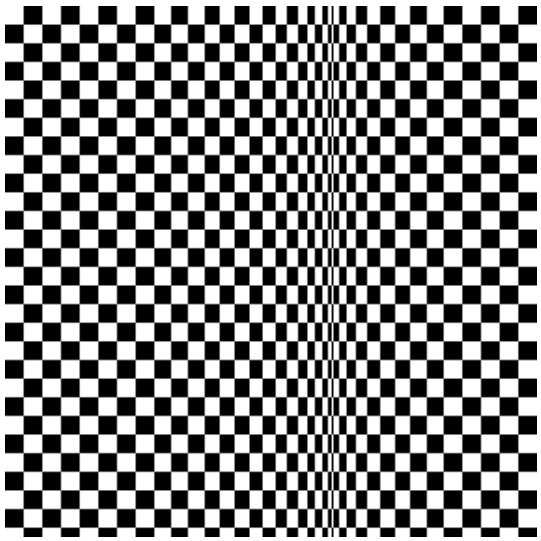
Furthermore, to make the image feels more 3D-like, we can increase the difference between widths for adjacent columns, this can be done by applying a power to all values, this is the reason we added an argument called *offset*.

Algorithm 7 Scale the column widths

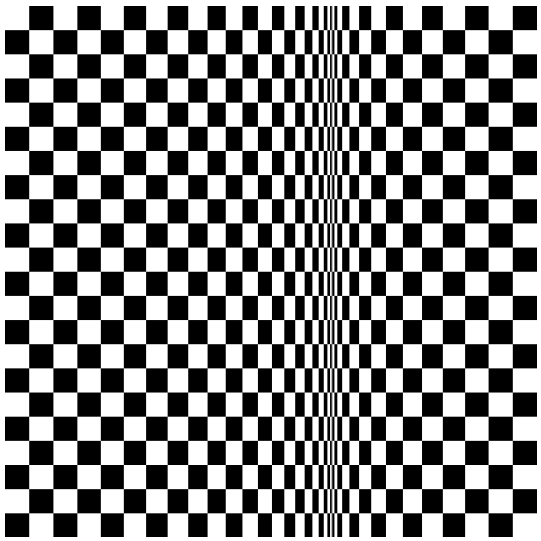
$ws_1 \leftarrow []$
$ws_2 \leftarrow []$
for $i = 0$ to l **do**
    $q_1 \leftarrow$ *quadratic1*$[i]$
    append ceiling$(q_1)^{\text{offset}}$ to ws_1
for $j = (l + 1)$ to n **do**
    $q_2 \leftarrow$ *quadratic2*$[j]$
    append ceiling$(q_2)^{\text{offset}}$ to ws_2

Something worth mention here about the above code is that since applying power to an integer could get huge easily, in the program the largest *offset* that can get and be allowed is 3. Over this value could get overflow error. When the original widths are relatively large, *offset* could only get up to 2. Hence the recommendation to achieve best output is to set *offset* to 2.

A quick peak of the differences between different *offset*s is shown below, in Figure 21:



(a) *offset* = 1

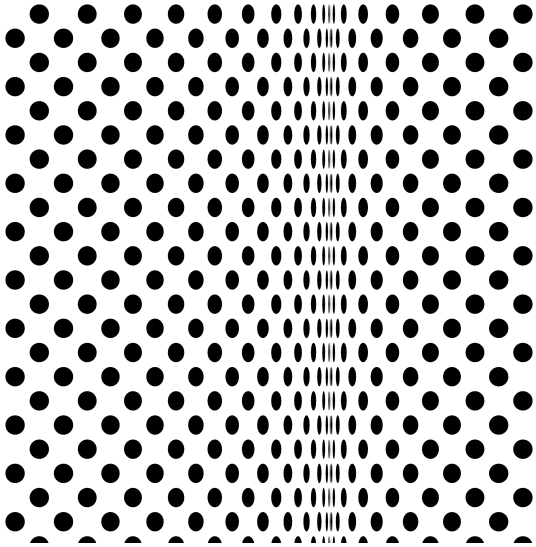


(b) *offset* = 2

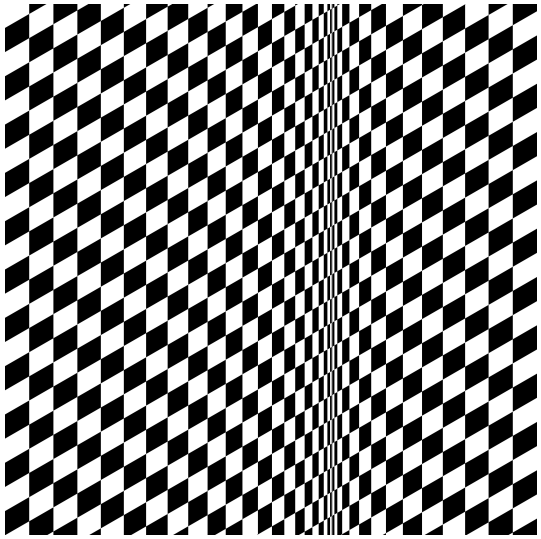
Figure 21: Different *offset* applied to same double quadratic equations

The two result images above are generated using rectangles, for each row, a rectangle with a certain colour and a specified width as we defined previously are printed, the process is then repeated in rows to complete the whole image. Moreover, we can use various sub shapes to achieve other images. Like in the artwork *Pause* from Bridget Riley, ellipses are used as sub shapes. We can even use parallelograms defined from previous sections.

Here are two images using ellipses and parallelograms respectively:



(a) Dot shapes, similar to *Pause*



(b) Parallelogram shapes

Figure 22: Different sub shapes applied to same double quadratic equations

3.4.3 Colour

According to the two artworks from Bridget Riley given in the sample section, we can see one with a pure check pattern and another with a gradient stripe in diagonal. In this section we will show how these colour patterns are achieved in a mathematical way.

- **Check**

  This colour pattern is a rather simple one, the basic idea is each sub shape is given a specific colour based on their row and column index.

  As long as we define one sub shape in the whole image (say the one located at the top-left corner), all sub shapes are defined. Let's define the column index *col* starts from 0 from left to right, and row index *row* starts from 0 from up to down. If we give the sub shape at the top left corner - the shape located at $col = 0$, $row = 0$ a white colour, then we know for all even *col* and *row*, the colour is white. Hence by checking whether *col* and *row* are even or odd numbers, white or black is given to its sub shape.

  To make things simple, we can just check whether the sum of *col* and *row* is even or odd, like the pseudocode shown below:

Algorithm 8 Check pattern colour definition

```
1: col ← columns of sub shapes
2: row ← rows of sub shapes
3: colours ← [col][row]
4: for i = 0 to (col − 1) do
5:     for j = 0 to (row − 1) do
6:         if (i + j) mod 2 = 0 then
7:             colours[i][j] ← white
8:         else
9:             colours[i][j] ← black
```

  Here the algorithm above constructs a **mono** check pattern, if we want to have check pattern with any two colours we can let the user define them and replace black or white respectively.

- **Gradient**

  Apart from the check pattern we have generated above, the shape now adds a white diagonal stripe at a certain location to it. More arguments are needed and are shown below:

  (a) **gradientPoint**, the index of the column where the centre of the stripe (the brightest point) locates at the first row ($row = 0$).

  (b) **gradientDelta**, a floating point number applies to the colour value in the gradient stripe to adjust the strength of gradient.

  (c) **gradientWidth**, defines how wide the gradient stripe needs to be.

  Here let's consider a simple situation of colour changing in the gradient stripe from black to white, this means that the R (Red), G (Green) and B (Blue) values of the shape are always the same to maintain a grey-scale transition. In order to calculate the RGB values of the shape in the range of the gradient stripe, we need to define a special index for shapes within the range of stripe. Let's call it *gradientIndex* and define,

$$\text{when } gradientIndex = 0, \text{ we have } R = G = B = gradientDelta$$

  i.e. the shape reaches the brightest colour. To make a whole stripe with width *gradientWidth*, we adjust the colour value for all shapes when their $gradientIndex \in [-\frac{1}{2} \cdot gradientWidth, \frac{1}{2} \cdot gradientWidth]$.

  Since we need a stripe in diagonal with a bottom-right direction, we need the *gradientIndex* values for shapes bottom-right-diagonal to each other remain the same, this can be done using

their *col* - *row*. As we need *gradientIndex* to be 0 at *gradientPoint* we defined, this can be calculated by difference between them, i.e.

$$gradientIndex = col - row - gradientPoint$$

hence when $gradientPoint = 0$, the stripe will start at the shape located at top-left corner of the image.

To make gradient effect, let's define a value *step* that could be multiplied to the RGB values of each shape, (notice the shapes increase and decrease RGB for half of the *gradientWidth* each)

$$step = 1 \div \frac{gradientWidth}{2} = \frac{2}{gradientWidth}$$

Now let's consider two situations according to the value of *gradientIndex*:

- When $gradientIndex \in [-\frac{1}{2} \cdot gradientWidth, 0]$, RGB values increase from 0 to *gradientDelta*
  By applying $\frac{1}{2} \cdot gradientWidth + gradientIndex$, we can get the difference between current index and lowest index.
- When $gradientIndex \in (0, \frac{1}{2} \cdot gradientWidth]$, RGB values decrease from *gradientDelta* to 0
  By applying $\frac{1}{2} \cdot gradientWidth - gradientIndex$, we can get the difference between highest index and current index.

we can actually merge these two situations into one using the formula below:

$$difference = \frac{1}{2} \cdot gradientWidth - |gradientIndex|$$

for all $gradientIndex \in [-\frac{1}{2} \cdot gradientWidth, \frac{1}{2} \cdot gradientWidth]$, $gradientIndex \in \mathbb{Z}$.

Finally, we get the formula of the R, G and B values for shapes within the gradient stripe:

$$R = G = B = step \cdot difference \cdot gradientDelta$$

When $gradientIndex \notin [-\frac{1}{2} \cdot gradientWidth, \frac{1}{2} \cdot gradientWidth]$ (the range listed above), the shape colour will be set to black.

Below is the pseudocode to achieve a black-to-white coloured gradient stripe pattern:

Algorithm 9 Gradient pattern colour definition - double quadratic equation

$col \leftarrow$ columns of sub shapes
$row \leftarrow$ rows of sub shapes
$gradientPoint \leftarrow$ location of centre of the gradient stripe
$gradientDelta \leftarrow$ brightest colour in the gradient stripe
$gradientWidth \leftarrow$ width of the gradient stripe
$colours \leftarrow [col][row]$
for $i = 0$ to $(col - 1)$ **do**
    for $j = 0$ to $(row - 1)$ **do**
        $gradientIndex \leftarrow i - j - gradientPoint$
        if $(i + j) \bmod 2 = 0$ **then**
            $colours[i][j] \leftarrow$ white
        else
            if $gradientIndex \in [-\frac{1}{2} \cdot gradientWidth, \frac{1}{2} \cdot gradientWidth]$ **then**
                $step \leftarrow 2 \ / \ gradientWidth$
                $difference \leftarrow \frac{1}{2} \cdot gradientWidth - abs(gradientIndex)$
                $r, g, b \leftarrow step \cdot difference \cdot gradientDelta$
                $colours[i][j] \leftarrow (r, g, b)$
            else
                $colours[i][j] \leftarrow$ black

Figure 23 below shows the result by applying a gradient pattern to double quadratic equations using $gradientPoint = 3$, $gradientDelta = 0.9$ and $gradientWidth = 5$:

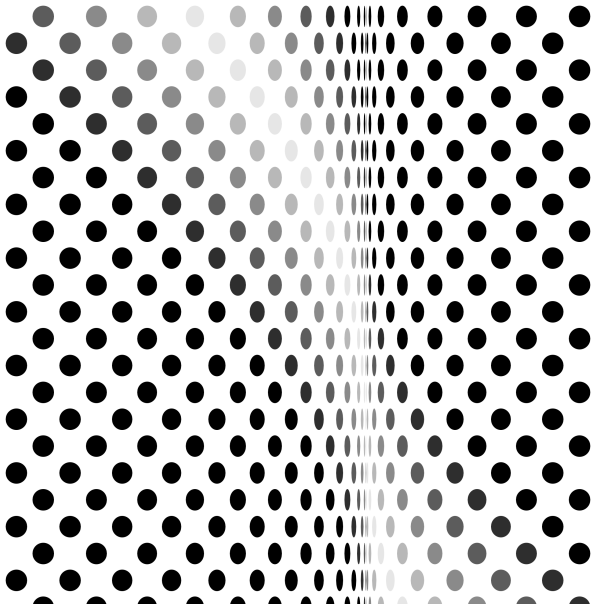
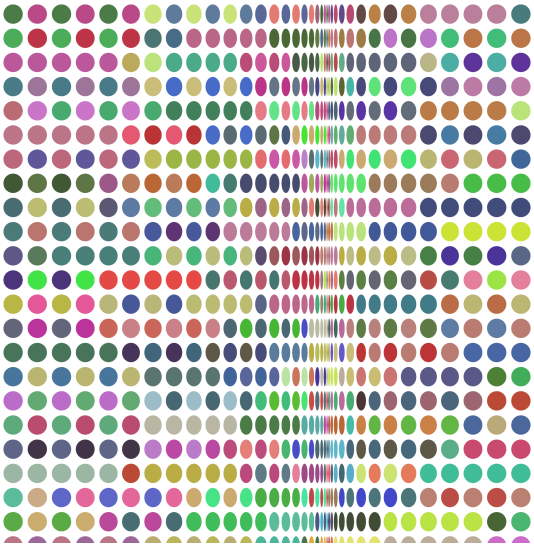


Figure 23: Double quadratic equations with gradient pattern and dot sub shapes

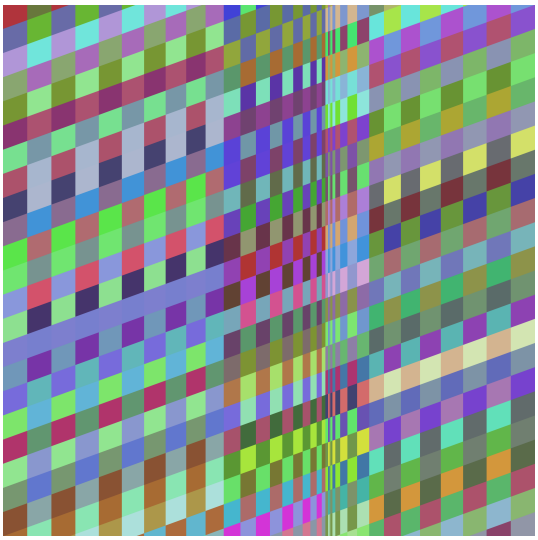
- **Random**

  By applying our existing random colour library from shape parallelogram (see section 3.1.3), we can get the same effect of colouring pattern using double quadratic equations instead. By importing the technique to change the angle of parallelograms, we can also change this property when program is in parallelogram sub shapes mode. With the application and combination of various different parameters and shape libraries, we can generate many creative images apart from Bridget Riley's original artworks.

Here are some sample outputs using random colouring pattern below:



(a) Dot sub shapes with *adjacentProbability* = 0.6



(b) Parallelogram sub shapes with *angle* = 20, *zebraProbability* = 1.0 and *zebraList* = [5]

Figure 24: Various arguments applied using double quadratic equations

3.5 Kiss

3.5.1 Sample

The shape is named after Bridget Riley's another artwork, *Kiss* finished in 1961.

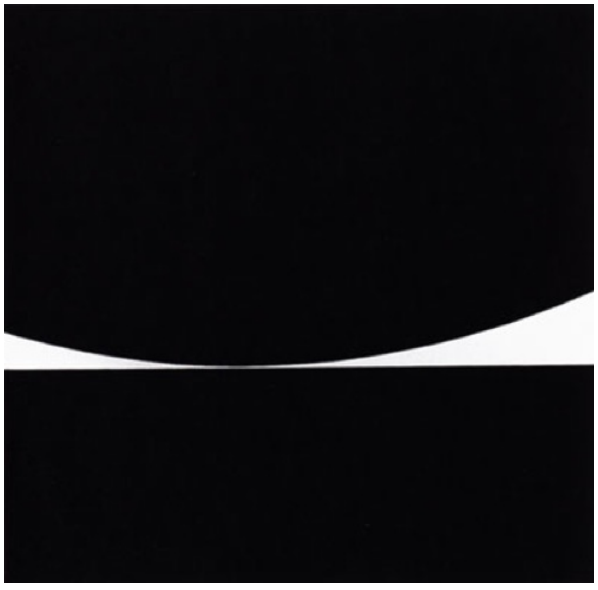


Figure 25: *Kiss*, Bridget Riley, 1961 [16]

From this artwork, we can see the curve at the top almost meets the bottom horizontal line leaving an extremely narrow gap, with the first view it may trick people think the bottom line is also a curve bending upwards, creating an optical illusion effect. With a strong contrast colour of black and white, viewers may feel that there is a gradient changing and the colour towards the gap is more grayish compare to white colour in wider spaces, hence giving a 3D effect to human eyes.

3.5.2 Shape

The shape in this program is relatively easy compare to previous shape patterns. To recreate the artwork, we need to add a shape constructed of top curve and bottom line. Similarly I used parabola to achieve the curve line. In order to make the shape adjustable, we need following arguments to proceed:

1. **base**, the distance between top of the image and the bottom line of the shape.
2. **kissPoint**, the distance between left of the image and the lowest point of the top curve (i.e. the kissing point).
3. **distance**, the distance of the narrowest gap between the top curve and the bottom line of the shape.
4. **curve**, the compress rate (the quadratic coefficient a) of the top curve.
5. **accuracy**, the step for x to be calculated as in the quadratic equation of the top curve.

To make things clear, below is a simple diagram to show these parameters:

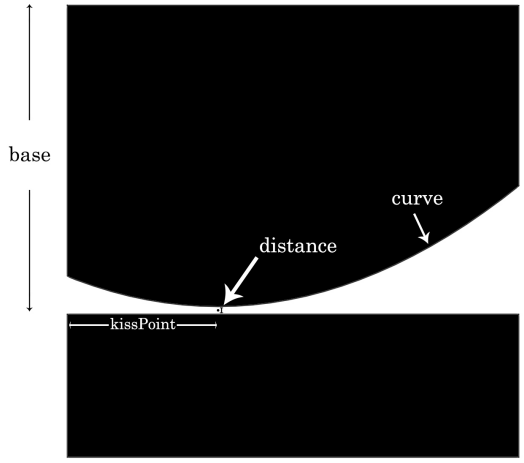


Figure 26: Diagram for kiss parameters

With these parameters we can locate and adjust the shape to form different kiss patterns.

If we take top of the image as the x axis and left as y axis, right and downwards as positive direction respectively, we will have:
the equation for the bottom line,

$$y = base$$

the equation for the curve as quadratic equation,

$$y = -\frac{1}{1000} \cdot curve \cdot (x - kissPoint)^2 + (base - distance)$$

and the corresponding pesudocode below.

Algorithm 10 Create a kiss shape

$height \leftarrow$ height of the image
$base \leftarrow$ base as given
$kissPoint \leftarrow$ kissing point distance as given
$distance \leftarrow$ gap distance as given
$curve \leftarrow$ quadratic coefficient as given
$accuracy \leftarrow$ accuracy as given
$w \leftarrow height$ ▷ Set width of image the same as height
$h \leftarrow kissPoint$
$k \leftarrow base - distance$
$a \leftarrow -\frac{1}{1000} \cdot curve$
$kiss \leftarrow []$ ▷ The kiss shape list stores coordinates of vertices
$x \leftarrow 0$
while $x < w$ **do**
    $y = a \cdot (x - h)^2 + k$
    append (x, y) to $kiss$ ▷ Add vertices of top curve to the list
    $x = x + accuracy$
append $(w, base)$ to $kiss$ ▷ Add vertices of bottom line to the list
append $(0, base)$ to $kiss$
append $(0, a \cdot h^2 + k)$ to $kiss$ ▷ Enclose the shape, back to starting point at $x = 0$

Notice we have $-\frac{1}{1000} \cdot curve$ as the quadratic coefficient since we flipped y axis upside down, and it is scaled to achieve better curve output.

3.5.3 Colour

The colour settings in the original artwork is trivial with a black background and a white shape in the middle. We can also further up this idea by applying different highly-contrast colours to the program if necessary.

By filling up background and the shape, the final output is shown below as in Figure 27:

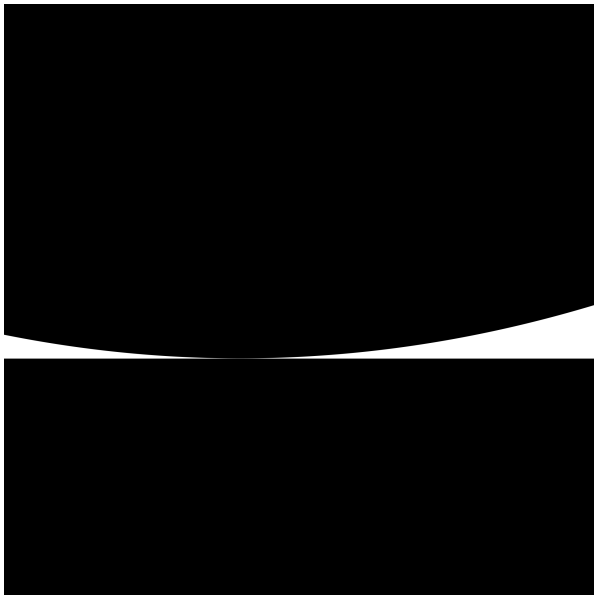


Figure 27: Generated image of kiss pattern

The image above is generated with parameter settings: $height = 2500, base = 1500, kissPoint = 1000, curve = 0.1, distance = 1, accuracy = 0.1$.

3.5.4 Further consideration

Now consider adjusting the parameters, what if we set the value *distance* to a negative value?

If we change *distance* to a negative value (say -10), we will have the curve crossing through the bottom line, and hence has two intersection point. The drawing tool will also fill up the overlapping area of the shape, so the image generated is shown as below:

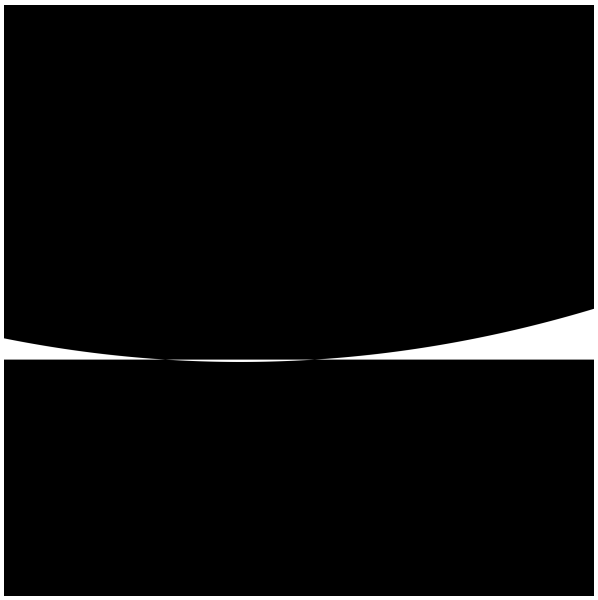


Figure 28: *distance* = -10

The output actually let viewer feels like the curve touched the bottom line twice but not crossing it, by accident this creates an optical illusion as well.

3.6 Blaze

3.6.1 Sample

Here is another shape pattern that Bridget Riley drew during her whole career with their names containing the word 'blaze'. Hence let's also name this shape pattern with the same word. Blaze normally stands for fire burning fiercely and brightly and here these artworks also give viewers a feeling of shapes bursting out of canvas strongly, like bringing an explosion.

(a) *Blaze Study*, Bridget Riley, 1962 [17]

(b) *Blaze 1*, Bridget Riley, 1962 [18]

3.6.2 Shape

At first we may only detect some random stripes facing different directions at a certain location and this seems hard to form in a mathematical equation and apply in computer programs. If we dig in details, we may find the picture is mainly constructed using several circles. However instead of showing their arcs, we can see their edges are treated as a turning point for each stripe. The stripes are spread in a way that the turning points on the same circle actually evenly divide the circle. We can also spot the continuity of each blaze stripe from the center of the innermost circle all the way to the outermost one. Hence we can compute the vertex for one stripe, and then increase its starting angle and print them out recursively.

The arguments we need to generate blaze pattern is listed below:

1. **circleCentre**, a list to store centre locations of each circle from innermost to outermost.
2. **circleRadius**, a list to store radius of each circle from innermost to outermost.
3. **angleOffset**, a list to store the angle between vertical and the direction from centre of the circle to its starting point, clockwise.
4. **angle**, the value of the angle for each evenly divided sector.

Here is a basic idea as shown below in Figure 30:

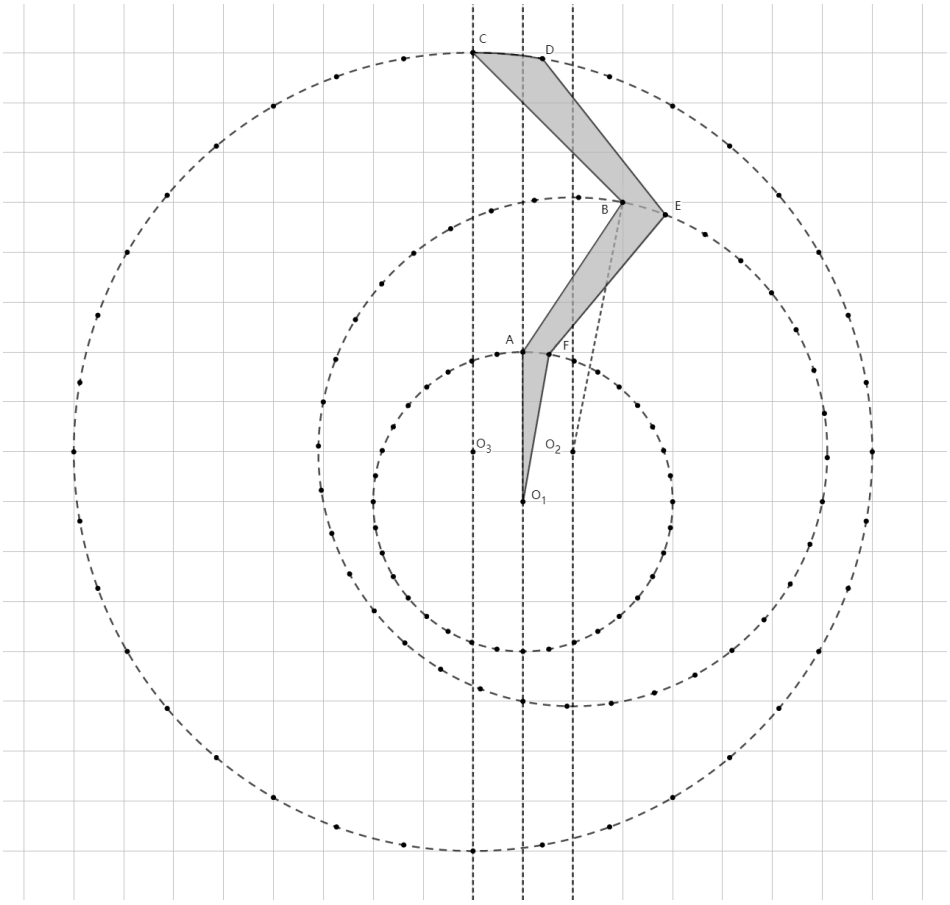


Figure 30: Basic mathematics behind blaze shapes

In the sample shown above, we have three circles O_1, O_2 and O_3 each with radius r_1, r_2 and r_3 respectively. We define the starting point for each circle using given *angleOffset*, here A, B and C respectively and divide them evenly into sectors with same angles. Here we have divided the three circles into 36 sectors each with a single sector having $angle = 10°$. Finally, starting from the centre of circle O_1, we connect each starting point with **segments** (A, B and C) until we reached outermost circle O_3, we apply an **arc** to connect to the next dividing point of circle O_3 (point D) with centre O_3 and radius r_3, we then connect the next dividing point of each circle with **segments** (D, E and F) all the way back to O_1 again. Hence, we formed the shape for a single blaze stripe $O_1ABC\overset{\frown}{C}DEFO_1$.

Similar as previous, let's define the origin point as top-left corner of the image canvas, right and downwards as positive x-axis and y-axis direction respectively. We know for a single circle O centred at point (h, k) with a radius r, we have its general equation:

$$(x - h)^2 + (y - k)^2 = r^2$$

By applying trigonometry, we can get the coordinate for all the dividing points on the circle with a certain *angleOffset*.

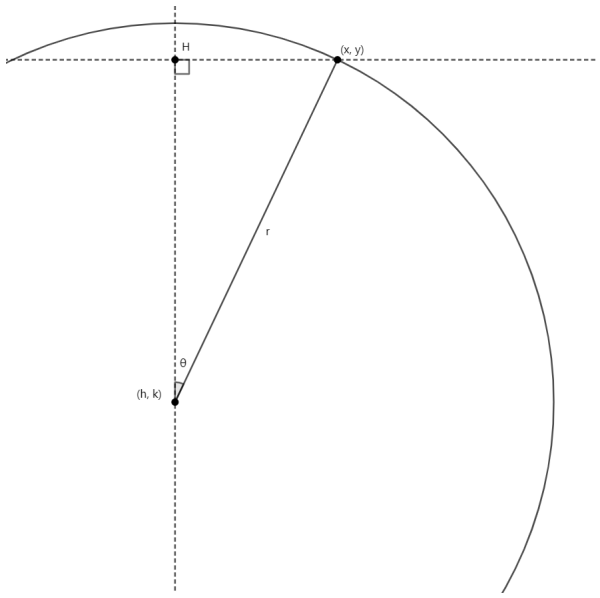


Figure 31: Trigonometry for calculating coordinates for all dividing points

Shown as Figure 31 above, if we have a circle with known centre (h, k) and its radius r, for a point with given $angleOffset = \theta$, we set its coordinate to be unknown as (x, y). Now by constructing a perpendicular line via the point to the vertical direction, we can get a right-angle triangle and hence applying trigonometry we get:

$$x = h + r \cdot sin\theta, \quad y = k - r \cdot cos\theta \quad \text{for all } \theta \in [0, 2\pi)$$

Again here the minus sign is used to calculate y-coordinate, since y-axis is flipped for the image.

With all the knowledge we have so far, we can generate each blaze stripe using pseudocode as shown below, notice we need to convert *angleOffset* and *angle* we input from degrees to radians:

Algorithm 11 Create a blaze stripe

function BLAZE_STRIPE($circleCentre, circleRadius, angleOffset, angle$)
    $cs \leftarrow circleCentre$ ▷ Abbreviate here for cleaner code
    $rs \leftarrow circleRadius$
    $\theta \leftarrow angleOffset$
    $origin \leftarrow (cs[0][0], cs[0][1])$
    $size \leftarrow$ list length of cs
    $bs \leftarrow []$
    append $origin$ to bs ▷ Add starting point
    for $i = 0$ to $(size - 1)$ **do** ▷ Add vertices going outwards to the list
        append $(cs[i][0] + rs[i] \cdot sin(\theta[i]), cs[i][1] - rs[i] \cdot cos(\theta[i]))$ to bs
        $\theta[i] = \theta[i] + angle$ ▷ Increment each value in angleOffset (θ) by $angle$
    for $j = (size - 1)$ to 0 **do** ▷ Add vertices going inwards to the list
        append $(cs[j][0] + rs[j] \cdot sin(\theta[j]), cs[j][1] - rs[j] \cdot cos(\theta[j]))$ to bs
    append $origin$ to bs ▷ Back to starting point
    return bs

The whole blaze pattern can then be generated by calling this function recursively with a loop from 0°to 360°using incremental step of *angle*:

Algorithm 12 Create the blaze pattern

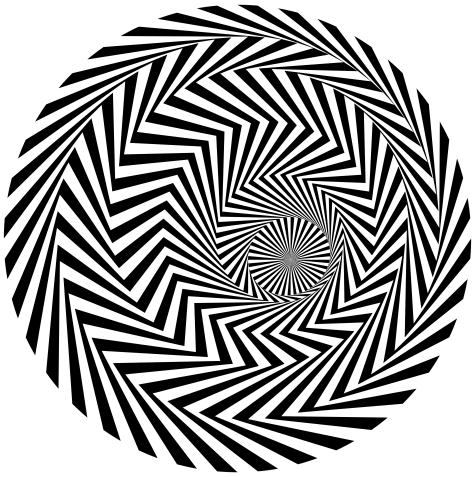
1: $a \leftarrow 0$
2: **while** $a < 360$ **do**
3: *// Extra steps to connect vertices by segments or arcs here...*
4: BLAZE_STRIPE($circleCentre, circleRadius, angleOffset, angle$) ▷ Parameters given by input
5: $a = a + angle$

Notice we did not put any recursive variables in function call, this is because the list *angleOffset* is changing its values inside the function so the while loop is only used to track how many times the function needs to be called. Furthermore, an extra step connecting the vertices with either **segments** or **arcs** is not shown here and has been abbreviated.

3.6.3 Colour

The colour pattern used in the original artwork is similar to most op arts shown previously - mono-check pattern. By tracking the index of each blaze stripe, the colour scheme can be achieved easily by applying black and white to odd and even indexes respectively. We can also add a feature that allows users to define their own colours so image with various colours can be generated as well.

Below group of images shows the final output by applying the blaze pattern algorithm compare to the original one drawn by Bridget Riley:



(a) Generated blaze pattern

(b) *Blaze 1*, Bridget Riley, 1962 [18]

Figure 32: Blaze pattern

Parameters applied:

- *circleCentre* = [(3000, 3000), (2900, 2800), (2800, 2700), (2700, 2600), (2600, 2700), (2500, 2800)]
- *circleRadius* = [400, 600, 1000, 1400, 1800, 2200]
- *angleOffset* = [0°, 300°, 340°, 320°. 350°, 320°]
- *angle* = 5°

From the original art shown on the above right, we can see the innermost circle is hollowed, this has a slight difference compare to her another blaze-pattern artwork *Blaze Study* [17] which our program

is based on. To achieve a centre-hollowed effect we can just remove codes that add the centre of the innermost circle as vertex when generating the blaze stripe (line 6 and 12 in Algorithm 11).

3.7 Waves

3.7.1 Sample

This shape is based on an artwork from Bridget Riley called *Cataract 3* finished in 1967, shown below in Figure 33:

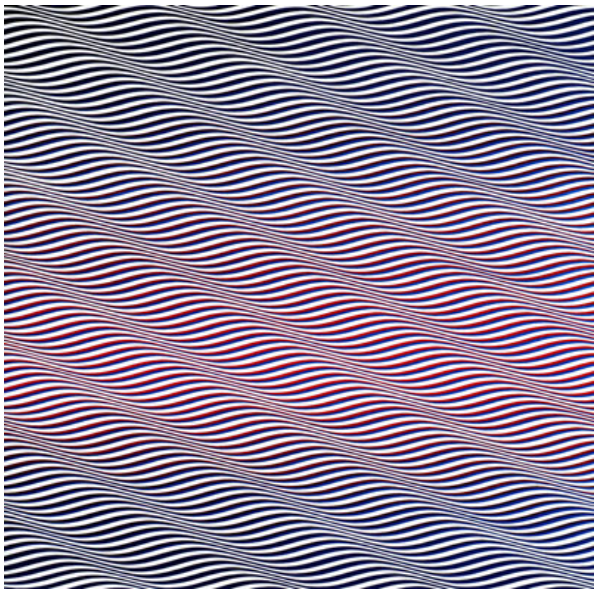


Figure 33: *Cataract 3*, Bridget Riley, 1967 [19]

By applying waves in a periodic arrangement, a three-dimensional viewing effect has created and provided to the viewers. A slight change in colour from border to the middle adds more vibrant feeling to the artwork.

3.7.2 Shape

To recreate this artwork using mathematics, we need to generate the waves by row and by applying an accumulate horizontal offset to the wave on each row, a nice effect will be achieved.

Here let's use sine wave in trigonometry and define the top-left corner of the image as the origin point, right and downwards as x-axis and y-axis positive direction respectively, as the same way as we did in the previous sections.

The arguments we need to generate each wave in this image are:

1. **amplitude**, the amplitude of the wave.
2. **period**, the period of the wave.
3. **verticalOffset**, the distance between waves, also be used to define the thickness of each wave.
4. **horizontalOffset**, the horizontal displacement between waves, also be used to define top and bottom curve displacement of each wave.
5. **accuracy**, the step for x to be calculated as in the formula of each wave.

Before we apply these parameters above to a formula, if we take a look at the artwork closer we may find in order to make the waves look more curvy, we need to form it as a shape with top curve and bottom curve as edges rather than using a single line connecting each sampling point. The shape

is then filled in with certain colour to form a nice-looking wave.

Below is a graph showing the difference between different strategies mentioned above:

(a) Without horizontal displacement

(b) with horizontal displacement

Figure 34: Difference between wave shapes

Specially in the program, I decided to give the top curve a normal horizontal displacement (*horizontalOffset*) and give the bottom curve with the value doubled, also giving the bottom curve two times *verticalOffset* plus a further value of 2. In this way the ratio of the distance between waves and the curvy effect of the wave could gain a balance for the whole image pattern.

In short we have the formulas for,
the top curve:

$$y = amplitude \cdot sin(period \cdot (x + horizontalOffset)) + verticalOffset$$

the bottom curve:

$$y = amplitude \cdot sin(period \cdot (x + 2 \cdot horizontalOffset)) + 2 \cdot verticalOffset + 2$$

With the knowledge above, we can define the wave shape in pseudocode below:

Algorithm 13 Create a wave shape

```
1:  function WAVES(width, amplitude, period, verticalOffset, horizontalOffset, accuracy)
2:      a ← amplitude
3:      p ← period
4:      v ← verticalOffset
5:      h ← horizontalOffset
6:      waves ← []
7:      x ← 0
8:      while x < width do                          ▷ Add vertices for top curve
9:          y = a · sin(p · (x + h)) + v
10:         append (x, y) to waves
11:         x = x + accuracy
12:     while x ≥ 0 do                              ▷ Add vertices for bottom curve
13:         y = a · sin(p · (x + 2 · h)) + 2 · v + 2
14:         append (x, y) to waves
15:         x = x − accuracy
16:     append (0, a · sin(p · h) + v) to waves     ▷ Back to starting point
17:     return waves
```

The value *verticalOffset* and *horizontalOffset* above is incremented each time in the recursive algorithm when calling the function:

Algorithm 14 Create the waves pattern

```
1: v ← 0
2: h ← 0
3: while v < height do
4:     // Extra steps to connect vertices and choose colours here...
5:     WAVES(width, amplitude, period, v, h, accuracy)   ▷ Parameters given by input
6:     v = v + verticalOffset
7:     h = h + horizontalOffset
```

3.7.3 Colour

Since from the original artwork we can figure out the colours are becoming redder for waves in the middle in a gradient way, we can apply a similar gradient colour pattern as the one we have applied in *Double Quadratic Equation* (section 3.4.3). To make things easier, we only need a single argument **gradientDelta** to adjust the intensity of the gradient in the image. No width or location of the gradient pattern are needed as we can define the RGB values of each wave shape depends on their location relates to the image height.

For the RGB values of a pixel, we know that changing one of its values also changes the intensity of its corresponding colour (red, green or blue respectively). So I came up with an idea by giving the program three Boolean values of whether or not change each value, different gradient colours could be achieved by mixing them:

- **isRedGradient**: whether or not change R value.
- **isGreenGradient**: whether or not change G value.
- **isBlueGradient**: whether or not change B value.

For example, if we set both *isRedGradient* and *isGreenGradient* to **True** but *isBlueGradient* to **False**, we will change R and G values, this would give the image a gradient colour of yellow.

Now let's consider what values each wave needs to have at a certain point, the colour intensity should reach its maximum at the middle with value *gradientDelta* and decrease back to black (R = G = B = 0). Similar to the gradient section of *Double Quadratic Equation*, we could define a step that is divided by half of the height of the image, i.e.,

$$step = 1 \div \frac{height}{2} = \frac{2}{height}$$

Similarly we can calculate the distance between the current wave to the middle by:

$$distance = \frac{1}{2} \cdot height - |v - \frac{1}{2} \cdot height|$$

for all $v \in [0, height)$, $v \in \mathbb{Z}$, where v is the starting point of each wave shape as we defined in *shape* section (3.7.2), and hence R, G and B values for each wave shape (when the Boolean variable is set to **True** correspondingly):

$$R = G = B = step \cdot distance \cdot gradientDelta$$

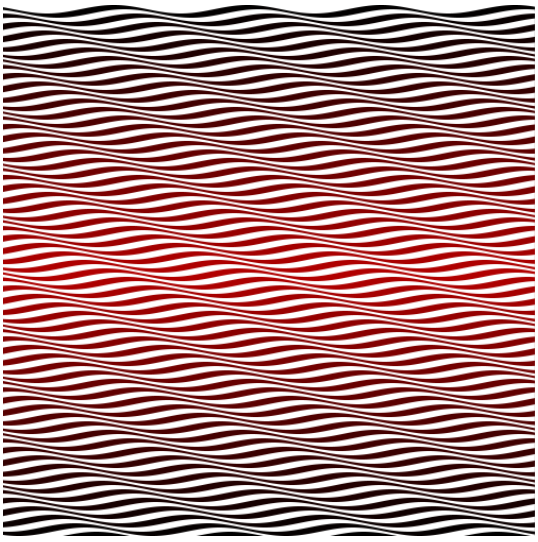
Here is the pseudocode below to achieve the waves colour pattern:

Algorithm 15 Gradient pattern colour definition - waves

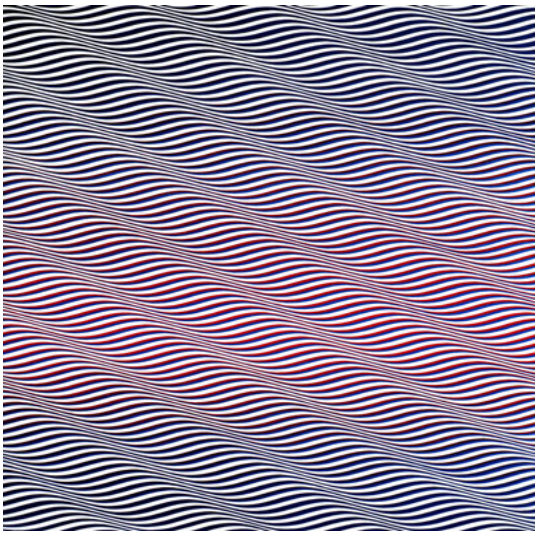
$index \leftarrow 0$
$v \leftarrow 0$
$gradientDelta \leftarrow$ brightest colour in the wave shapes
$isRedGradient \leftarrow$ whether or not red gradient
$isGreenGradient \leftarrow$ whether or not green gradient
$isBlueGradient \leftarrow$ whether or not blue gradient
$colours \leftarrow []$
while $v < height$ **do**
    if $index \bmod 2 = 0$ **then**
        $colours[index] \leftarrow$ white
    else
        $step \leftarrow 2 \ / \ height$
        $distance \leftarrow \frac{1}{2} \cdot height - abs(v - \frac{1}{2} \cdot height)$
        $gradient \leftarrow step \cdot distance \cdot gradientDelta$
        $r, g, b = 1$ ▷ Initial values for R, G and B are set to 1 as waves are black originally.
        if $isRedGradient$ **then**
            $r = gradient$
        if $isGreenGradient$ **then**
            $g = gradient$
        if $isBlueGradient$ **then**
            $b = gradient$
        $colours[index] \leftarrow (r, g, b)$
    $index = index + 1$
    $v = v + verticalOffset$

Notice the difference between the gradient pattern here and the one we used to generate in double quadratic equation, instead of defining colours based on their **indexes of row (and column)**, we calculate the values using the **height of the image**. This is because we did not define the number of rows the waves need to be generated and hence the colour should depend on location related to the image size in our waves pattern.

Here is the image our algorithm generated compare to Bridget Riley's original artwork *Cataract 3*:



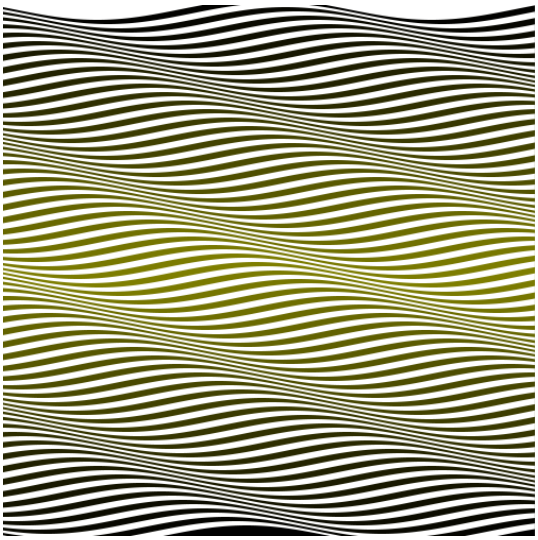
(a) Generated with a red gradient, *gradientDelta* = 0.7



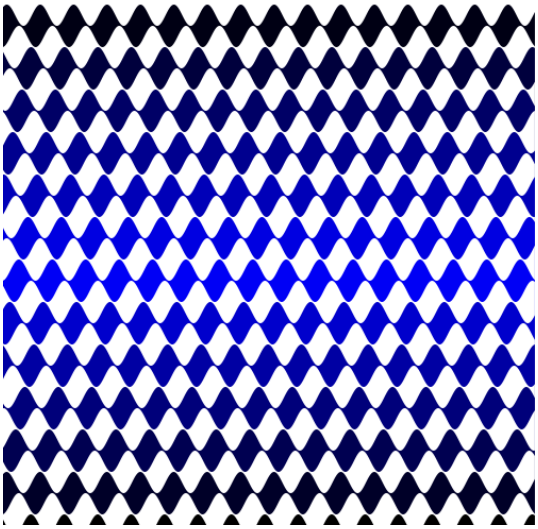
(b) *Cataract 3*, Bridget Riley, 1967 [19]

Figure 35: Waves pattern

More images with different colour gradient and other arguments are shown below in Figure 36.



(a) Yellow gradient, *gradientDelta* = 0.5, lower *period* and higher *amplitude*



(b) Blue gradient, *gradientDelta* = 1.0, extrmely large *period* and larger *verticalOffset*

Figure 36: Various waves pattern generated

3.7.4 Improvements

Until now, the algorithm could handle gradient colour with red, green and blue with their mixtures. It should be improved that users can define what colour they want and hence a specific ratio of each R, G and B value needs to be applied.

Also we set the base colour to black and white only, this could also be enhanced that users can define them.

If we check the original artwork more in details, we can see the waves in the middle are actually constructed with half coloured half black, this creates a more 3D effect compare to the current pure colour setting. There could also be a possibility that the resolution of the image affected this (a blur out) and the artwork is actually drawn using pure colours, but it remains unknown. So a better algorithm could be applied to stack two sub shapes into one.

4 Design

4.1 Starting with tradition

In early days when computer technology was still a concept, op artists use their own creativity to design shapes and patterns that form into a magnificent artwork with tools that almost have no difference with any other artists. Now imagine if we are going to create op art by hand, the steps we should do includes:

1. Prepare a blank canvas with certain size to draw on.
2. Decide basic shapes to be filled in. Most artworks are created with a single shape with different patterns, though may vary.
3. Decide the colour scheme the artwork is going to be, such as mono or coloured.
4. Decide a basic pattern and hence know where each shape is located on the canvas.
5. After decisions are made, draw the shapes repeatedly at various locations and a whole artwork is created.

4.2 Software design

Similarly to the traditional way of drawing an op art, a computer software could use certain libraries and extensions to do similar processes to draw a digital image that uses the same steps as how the artwork is traditionally made.

The figure 37 below shows a basic flowchart to generate a digital op art image by computer program:

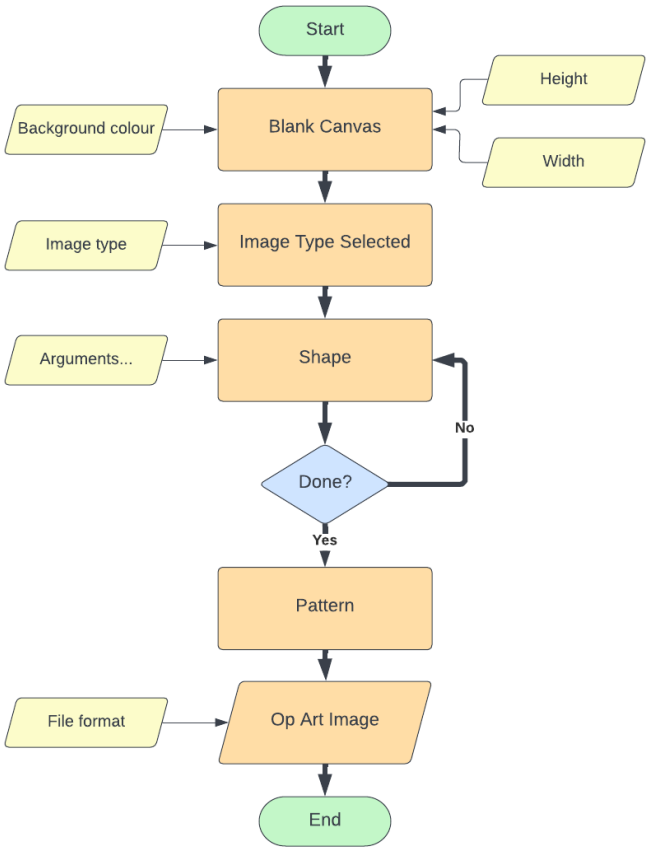


Figure 37: Basic flowchart of op art generator

According to the flowchart above, each step is defined as follows:

1. A blank canvas is created by given **height**, **width** and **background colour**.
2. The program then asks the user to choose various **image types**, which decides its following behaviours.
3. **Various arguments** are provided from user input and the corresponding shapes are printed out using a defined algorithm. For argument details and algorithms on each shape please see *Mathematics Behind Art* (Section 3).
4. Check whether the whole image is completed using a defined judgement, if not, go back to step 3.
5. Since all shapes are recursively printed out and a complete pattern is formed, the program then request the **file format** the user would like to check or save.
6. Finally, output the corresponding op art image.

From the software design flowchart shown above, we can see its similarity to the traditional process. In this way the computer could produce and create an op art image similar to hand drawing method, but with an almost perfect accuracy and efficient generating speed.

5 Implementation

5.1 Objective

In this project, my program implementation is based on the following user objective stated that:

"This project aims to use computer technology to generate images similar to those produced by Bridget Riley, but allow the computer user to adjust the parameters used to generate the images."[20]

By having the ability that users could adjust and tweak different parameters to generate op art, our program aims on helping users in following scenarios:

- For users who would like to get a basic knowledge of what op art is and play around with the generated image.
- For art learners and students to study with op art, especially artworks from Bridget Riley.
- Generate and adjust shapes and patterns of the artwork for professional artists who want to get inspirations and ideas when create their own optical illusion form of artworks.
- Save the op art image to make further use and process.

Compare to traditional hand-drawing form of op art creation, the op art generator program has following advantages:

- It generates artworks in almost no time and is way faster.
- Users can treat the program as a draft creator and plan for their artwork in advance.
- Its artwork can be saved and stored in local files in digital form, convenient to transfer or share.

5.2 Iteration

The project has been divided into several iterations, at each stage the program status was changed or new features were added or amended.

Here are three main iterations during the development of my project:

1. Background research on the abstract of op art and studying the pattern and construction of different artworks.
2. An op art generator program written in python runs under Python environment.
3. A web app deployed onto Heroku with HTML web pages communicate with Python Flask framework.

Since the first iteration does not involve in program implementation, I would like to discuss the last two main iterations in detail and how it is evolved into the final form.

5.3 Structure

5.3.1 Planning

Based on user objective, the op art generator program should be accessible in the following ways [20]:

- *"Via a GUI, allowing users to choose the general type of image produced, and set various parameters"*
- *"Via a web interface (that might embed the GUI as an applet)"*
- *"Via a scripting language, allowing the user to specify how the parameters of a image should be varied in an animation of the images"*

My initial plan to design the structure is to build an executable program that has the feature of generating op art. Since user accessibility is a key part of the software design, a user-friendly GUI that won't require much installation or preparation is very important in consideration. In this project I adapted the main idea of the second point of the user objective and would like to design a web app based online so users could visit the website directly via browsers.

5.3.2 Python executable program

In the first stage, the program was divided into several Python script files based on their different pattern of op art:

- 'parallelogram.py', shape as shown in section 3.1
- 'stripev.py', shape as shown in section 3.2
- 'binomial.py', shape as shown in section 3.3.2
- 'parabola.py', shape as shown in section 3.3.3
- 'doublequad.py', shape as shown in section 3.4
- 'kiss.py', shape as shown in section 3.5
- 'blaze.py', shape as shown in section 3.6
- 'waves.py', shape as shown in section 3.7
- '[shape_name].png/svg', the output image in user defined format.

During this stage, each python file has a main function in it and their program structure is the same as the software design mentioned in section 4.2 except there is no shape type selection step.

In each program, arguments input from user are parsed and manipulated using the following code:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-x", "--argument", default={default_value}, type={type_name})
args = parser.parse_args()
argument = args.argument
```

In the sample code above, we imported the package **argparse**, created a parser with an *argument* with its default value set to **{default_value}** and type set to **{type_name}**.

To generate an op art image, the following packages and libraries are needed for users:

- **Python interpreter**: running program in '.py' extensions with arguments attached in command line.
- **pip**: package installer for Python.
- **Pycairo**: main critical library for drawing functions and features.
- **SciPy**: the library that helps calculating binomial expansions.

As disadvantages concluded in *Evaluation* later (section 6.1), some improvements need to be made. A user-friendly GUI needs to be introduced and hence this evolves the project to the next stage.

5.3.3 Flask GUI

Flask is a web framework based on Python, it helps communicate the web front-end with back-end Python source code.[21] In this stage, web pages are designed using HTML using coding logic based on JavaScript and styling with CSS. Users can choose the shape and input the value of each argument in the web page forms, by clicking a submit button the flask framework would then passing these arguments to Python program and hence the same processes are done as in the previous stage.

To achieve using a single Python executable program that is able to cope with Flask framework, following code refactor steps are needed:

1. Define one of the python file from previous section 5.3.2 as the only library (named 'app.py').
2. Remove package **argparse** and its related parser codes.
3. Add an input argument **imageType** to define which shape user has chosen.
4. Assemble all algorithm and argument parts of codes from other shapes, put them in various section based on the value of **imageType**.
5. Add routines for each function that returns template of web pages (more details will be shown later).
6. Set up the flask app and establish a connection.

After code refactoring, we need to design the GUI based on HTML web pages, I created a template for each shape to use, and index pages for users to choose which shape to go into. Shown as follows:

- 'index.html', the home page for op art generator, user can select **imageType** or check out the help guide.
- '{imageType}.html', each shape has their own required arguments, so corresponding text boxes, check boxes and sliders are applied to design each web page.
- 'fullscreen.html', allow users to click on output images and enlarge them, they can also have a comparison with Bridget Riley's original artwork and download the generated artwork on this web page.

The next step is to connect the template web pages with flask routine, this can be done by defining certain part of the python function as a routine and let it return back a rendered template passing certain arguments to it.

Here is the sample code as follows:

```
@app.route('/home', methods = ['POST', 'GET'])
def home():
    return render_template("index.html", graph="img/home_img.jpg")
```

Here we created a route called '/home' with both methods **POST** and **GET**, the function home() returns our written web page 'index.html' and passing the argument *graph* with its value.

Now in 'index.html', we could use Jinja2 code graph to reference the value that python program has passed to it, and hence the web page could show up the corresponding image.

To visit the webpage, we run the following command in terminal:

```
flask run
```

If succeeded, we should now be able to type *"localhost:5000/home"* in a web browser and the page 'index.html' will show us the image address corresponding to the one that python passed to it.

Here is the home page of op art generator and animator:

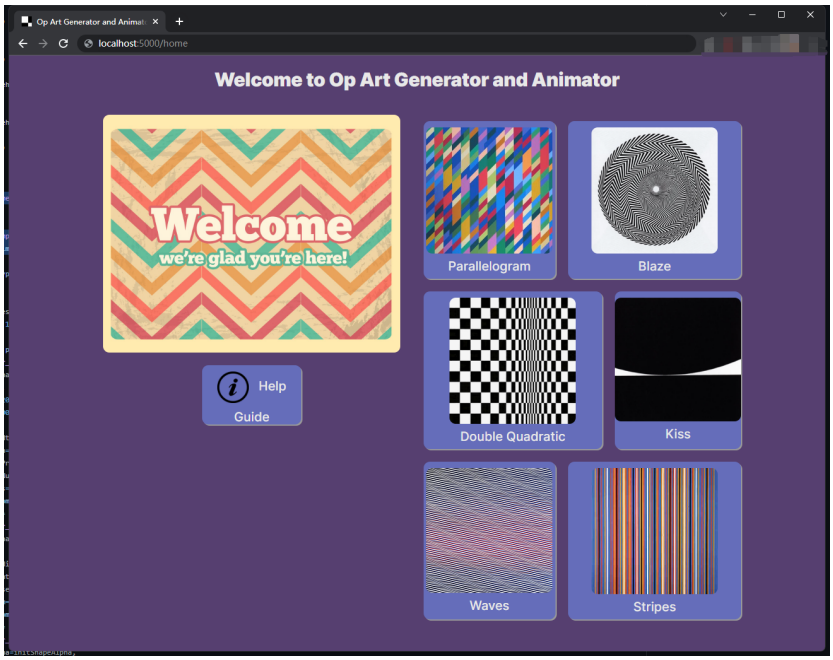


Figure 38: Home page

Similarly, we create all other routes based on our original python executable program. Below shows the basic logic for using the GUI:

1. Starting from 'index.html', users can choose image type and click on the corresponding button.
2. When a button is pressed, JavaScript embedded will bring user to **/selectType** route with given argument *imageType* from each button. The url will become ".../selectType?imageType=...".
3. The **/selectType** route will render corresponding web page based on the value of *imageType*.
4. Users are hence directed to '{imageType}.html', in which they can adjust the arguments as they like using sliders, text boxes, etc. With *graph* being home image, this is the default state when user enters.
5. The JavaScript is designed such that each time user changed a single argument, submit() function is called and the image will be updated, this strategy increases the interaction between users and the program.
6. The form where the arguments are included is designed with their tag *action* equal to value "output", this means it will find and submit to the python route called **"/output"** as well. The output() function is designed to receive all arguments submitted from the form, it also stores all algorithms that print out different shape patterns. After a new image is generated and stored at "out/output.png" (or svg based on *mode*), the function returns the same template web page. Different to route **/selectType**, the *graph* value is set to the address of the output image and hence users will see a rapid refresh on the web page with the image changed to the latest output result.
7. The route **"/output"** also passes back the arguments it received, this strategy is applied to maintain the same value users had given to python program in the previous step.
8. By using JavaScript to judge whether the image shown is an output image (its address value contains 'output'), we can allow users to click on the image and web page will call **"/fullScreen"** route in python code. Python will then render 'fullscreen.html' passing the address of the image *graph* and *mode* to it. In 'fullscreen.html' users can compare the generated image with Bridget

Riley's original artwork and download the image as well.

As we have already removed the parser that deals with the arguments in our early stage, we now use the following code to get the **POST** data from web pages in step 2 and 6:

```
# Read data from query strings, used in step 2.
imageType = request.args.get('imageType')

# Read data from web page input with name 'height_input' as an integer.
height = request.form.get('height_input', type=int)

# Read data from web page input with name 'shape_input' as a string.
shape  = request.form['shape_input']
```

Figure 39 below shows a complete UML diagram of the whole process between python back-end and HTML front-end with Flask framework:

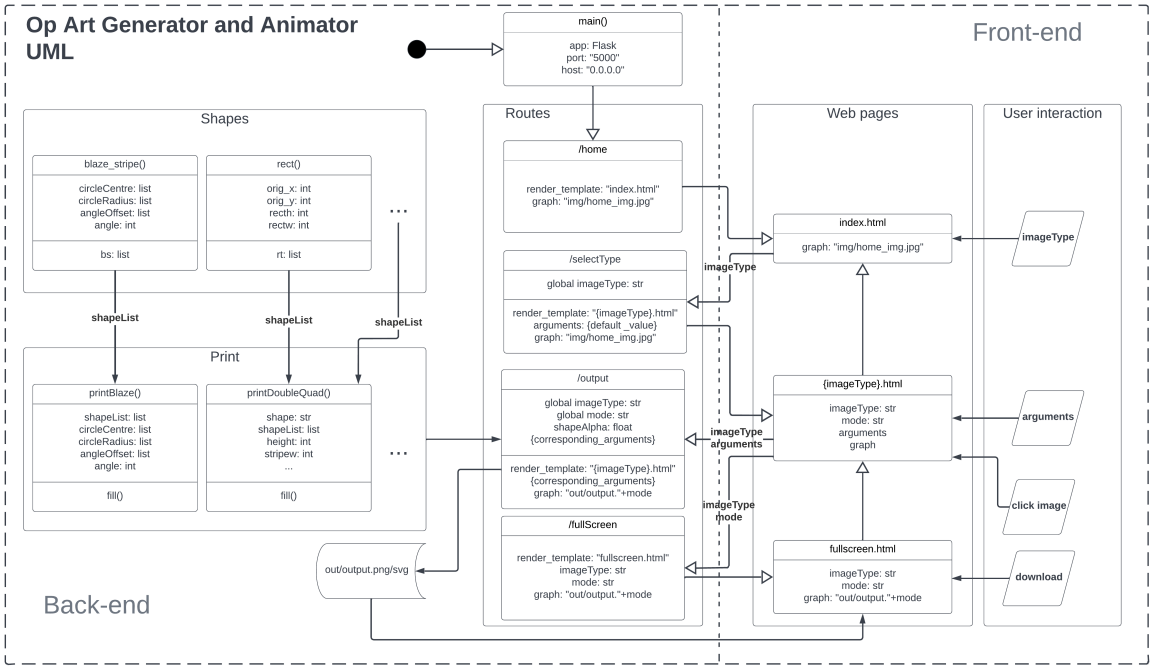


Figure 39: UML Diagram

The file structure for the project at this stage:

```
/
├── static
│   ├── css
│   │   └── main.css
│   ├── img
│   │   └── home_img.jpg
│   └── out
│       ├── output.png
│       └── output.svg
├── templates
│   ├── index.html
│   ├── {imageType}.html
│   └── fullscreen.html
└── app.py
```

After a complete GUI has been designed, we can then move on to the third stage to deploy and host the program online.

5.3.4 Deploy to Heroku

Heroku is a *platform as a service (PaaS)* that enables developers to build, run, and operate applications entirely in the cloud. [22] Compare to setting up a traditional cloud server and hosting a web service, Heroku requires minimum knowledge of network with no complicated steps. In this project, since we designed an image generating tool, there is no strict requirements on efficiency or multi-threading when hosting the website. Therefore using Heroku is a suitable choice.

Before our project is deployed onto Heroku, they need to be stored in a repository so I created one on GitHub and uploaded them. [23]

To make Heroku deployments from the repository, I logged in to the online console of Heroku and created a new project with name *'opartgenerator'*. To make the environment work before linking the project with my existing GitHub repository, a requirement text file *'requirements.txt'* and a process file *'Procfile'* need to be added to the root directory of my project.

To fill in the requirement text file, Python virtual environment needs to be created on the machine, under the environment, we can install the libraries and packages our main program *app.py* needs. After that, a command contains string of the installed libraries names is input into *'requirements.txt'*. Steps of setting up virtual environment here is abbreviated.

To fill in *'Procfile'*, we type in a single line:

```
web: gunicorn app:app
```

This code indicates the Heroku server that the web engine is run by an external library *gunicorn* (which we also included in the environment) and the main program is *app.py*.

At this point the Heroku deployment environment has been set up and after a successful build, we can visit our web app simply by input the URL: https://opartgenerator.herokuapp.com/home, where *opartgenerator* is the name of the Heroku project and *home* is our starting route of the whole program.

Problems during deployment

In the process of deploying project onto Heroku, two following problems have been occurred and resolved later on:

- Pycairo library failed to build in the virtual environment. Since the program is developed based on this main library we cannot deploy without it. I found a similar library that solved the problem called *cairocffi*, a C Foreign Function Interface (CFFI) based drop-in replacement for Pycairo. [24] On its documentation website, it stated that Pycairo library could be easily replaced by using:

```
import cairocffi as cairo
```

instead of

```
import cairo
```

In this way, no further adjustment needs to take place in the main program and cairocffi library could be built successfully in the virtual environment and hence on Heroku.

- Multi-threading problem for gunicorn default configuration. Since the arguments I designed to pass between back-end and front-end are not thread-safe, the arguments received may not up to date and this could cause bugs and runtime error in python program. At first I was not aware of the problem as it works perfectly under local Flask framework. When the website is hosted on Heroku, it occasionally returns an error page when users take actions too quick. After checking out a similar problem other developers have run into and raised on stack overflow, [25] it appears that by default, gunicorn would launch 2 workers as threads hence cause multi-threading problem when visiting data. To fix this problem, we can set worker number manually to 1 to enforce a single thread is deployed by changing our *'Procfile'* into:

```
web: gunicorn --workers 1 app:app
```

After this fix, no more web error pops out and it seems this solution has efficiently solved the problem.

6 Evaluation

6.1 First stage - Python executable program

At the initial stage of the project development, a command-line-based program had been created, some benefits during this stage includes:

- Users could use the program source code to gain fully understanding of the mathematics and algorithms behind each shape pattern and hence op art painting.
- Users could input the parameters as arguments in command line and pass them to the program with their own choice.
- The output of the images have separate names and clear classification so storing and tidying local files is handy.

Despite during this stage the third objective from users is fulfilled that users can generate images using a scripting language, this design has several disadvantages:

- It requires users to download and install several packages and libraries in advance, setting up takes time.
- Using command line for user input can be hard for users who has less knowledge in computing. Not user-friendly.
- Each shape type has a separate file and different files are stored under the same folder, this could cause difficulty in maintenance and confusion to users.
- The program runs locally and the source code could be easily broken by user accessing and editing.
- The program runs with limited platform support e.g. no mobile users could access it.

6.2 Second stage - Flask GUI

Throughout the process of this stage, a user-friendly GUI had been designed. Here are some advantages that Flask GUI can achieve:

- Instead of command line input, users can now choose which shape pattern with specified input arguments by simply click buttons, change values in text boxes, adjust modes in drop-down menus, select check boxes, etc. A better user interacting tool has been created for a user-friendly purpose.
- A web-based GUI framework creates cross-platform support on almost all devices nowadays that can use a web browser with a specified link.
- By providing a font-end web page to users they can compare the generated image with the original artwork created by artists via a single click into the picture. They can also zoom in the image to check further details.
- The program has a systematic structure and has separate parts with their own duty. It is easy for future maintenance and extension.

Although many shorts from the previous stage had been improved in this stage, some concerns are still remaining:

- Users have less contact with the source codes and algorithms now, makes it harder especially for professional users to know how each artwork is generated just by looking at a simply designed website.
- Flask framework still runs on a local environment, this is the reason we move on to deployment with Heroku.

6.3 Third stage - Heroku web app

In our final stage, a completed program has been submitted onto GitHub and deployed using Heroku. This makes the whole op art generator application run online and available to the public. With only browser required to access the website, users can generate their op arts on whichever device they like and whenever they want with almost no cost.

We know there is no one-hundred-percent perfection in designing a program and due to this, some further problems are still needed to be solved and improved:

- Since users need Internet to access the program, it is better if we also provide an offline version to download in case users are facing network problems.
- The website runs in single thread. This could cause problems such as image showing error when multiple users are accessing it at the same time. One way to fix this is using Redis, a database for storing data for separate users. [26] This could also cause the project become more complicated but is something worth considering.

7 Extension

7.1 Op Art Animator

In order to assist user gain better understanding of how each parameter they input can effect the generated image as it is changed, an animator could help making the artwork come to life and give viewers a vibrant impact.

To make this work, we can generate a sequence of images in advance when generating each shape type giving a specific argument. The images are then shown in their generated sequence with a given interval so viewers can treat it as an animation.

In the animator part, following arguments are required:

1. **isAnimator**, to allow the user to toggle on/off the animator, when it is set to off (**False**), the program will only generate one image as a normal generator regardless of the value of *imgs*, this could save total running time.
2. **interval**, the interval, in seconds, to display each image in sequence, this controls the playback speed of the animation.
3. **imgs**, the amount of images generated in total.
4. Changeable variables, users can toggle each variable on and off to decide whether it will increase by *step* or remain the same.
5. **step** of changeable variables, the amount for its variable in each image sequence increases, if the variable is toggled on.

7.2 Implementation

Due to the lack of time scale I only applied the animator to the shape *Double Quadratic Equation* with changeable variables *valley* and *gradientPoint*. Since I added an outer loop around the image generating logic for all shapes they can be extended easily. In theory, all arguments that are given from users to the program could be incremented during each loop to generate sequence of images, which give the program a good potential and let it provide lots of interesting features by doing so.

The program in *Double Quadratic Equation* will now generate following image files to the path *\static\out*:

- **out.png/svg**: the original image file shown as Op Art generator.
- **out1.png/svg, ..., outN.png/svg** for *imgs* = N, a sequence of image files based on given image format *mode*.

We then apply a **setInterval()** function in JavaScript on the web page with selected *interval* and change the *src* tag (image source) for the output window, after the last image is reached we go back to the first image and hence the animation will playback in a loop non-stop:

```
    i = 1;
    setInterval(function() {
                if (mode == "png") {
                        outputImg.src = "/static/out/output"+i+".png";
                } else if (mode == "svg") {
                        outputImg.src = "/static/out/output"+i+".svg";
                }

                if(i < imgs){
                        i++;
                } else {
                        i = 1;
                }
        }, interval*1000);
```

7.3 Improvement

Despite the animator playback could work flawlessly in local environment with an extreme short *interval*, it does not appear as so online. This is because the image would only load up after it response back to the browser and since sometimes the delay could happen up to 300ms (varied by internet connection), set *interval* below 0.4s could cause the image request cancelled and the animation will behave abnormal. Therefore it is recommended to have *interval* set greater than 0.5s.

Improvements could be made to fix this problem. One way is to load up all the sequence images after each generation onto the web page and use a slideshow technique in JavaScript. Other plugins or framework may solve this problem as well. Although this may increase the waiting time for each sequence to generate and load up, this enhances the later animation playback and is expected to be a better trade-off.

8 Ethical Issues

8.1 Legal Issues

There could be some copyright issues between the generated op art image with artists as some of them may look so similar that it is hard to distinguish between. Since op art is a form of style art which has minimum variant of shapes, colours and patterns, most paintings drawn by hand could just be as accurate as a computer program would do. Hence using a generator can cause severe issues such as violating reputation of authors.

8.2 Human

Since op art is a form of art with vibrant colours and shapes, it can sometimes cause sickness and dizziness by looking at them. Therefore it is worth to be mentioned and to warn the users such symptoms could happen when using the program to deal with op art.

References

[1] Bridget Riley. Pause. WikiArt, https://www.wikiart.org/en/bridget-riley/pause-1964, 1964.

[2] Overview - pycairo documentation. https://pycairo.readthedocs.io/en/latest/.

[3] Bridget Riley. Movement in squares. WikiArt, https://www.wikiart.org/en/bridget-riley/movement-in-squares-1961, 1961.

[4] Jon Borgzinner. Op art. *Time*, 1964.

[5] Op-art: History, characteristics. Art Encyclopedia, http://www.visual-arts-cork.com/history-of-art/op-art.htm.

[6] Roberta Smith. Victor vasarely, op art patriarch, dies at 90. *The New York Times*, 18 March 1997.

[7] Victor Vasarely. Zebra. WikiArt, https://www.wikiart.org/en/victor-vasarely/zebra-1937, 1937.

[8] The responsive eye. *New York: Museum of Modern Art*, 25 February 1965.

[9] Bridget Riley. Rose rose. WikiArt, https://www.wikiart.org/en/bridget-riley/rose-rose-london-2012-olympic-games-poster-2012, 2012.

[10] Terry Riggs. Bridget riley born 1931. Tate, https://www.tate.org.uk/art/artists/bridget-riley-1845, February 1998.

[11] Pointillism. Wikipedia, https://en.wikipedia.org/wiki/Pointillism.

[12] Bridget Riley. Fete. WikiArt, https://www.wikiart.org/en/bridget-riley/fete-1989, 1989.

[13] Bridget Riley. Nataraja. WikiArt, https://www.wikiart.org/en/bridget-riley/nataraja-1993, 1993.

[14] Bridget Riley. Ra 2. WikiArt, https://www.wikiart.org/en/bridget-riley/ra-2-1981, 1981.

[15] Bridget Riley. Chant 2. WikiArt, https://www.wikiart.org/en/bridget-riley/chant-2-1967, 1967.

[16] Bridget Riley. Kiss. WikiArt,
https://www.wikiart.org/en/bridget-riley/kiss-1961, 1961.

[17] Bridget Riley. Blaze study. WikiArt,
https://www.wikiart.org/en/bridget-riley/blaze-study-1962, 1962.

[18] Bridget Riley. Blaze 1. WikiArt,
https://www.wikiart.org/en/bridget-riley/blaze-1-1962, 1962.

[19] Bridget Riley. Cataract 3. WikiArt,
https://www.wikiart.org/en/bridget-riley/cataract-3-1967, 1967.

[20] Peter McBrien. Op-art generator and animator.
https://www.doc.ic.ac.uk/~pjm/teaching/bridget_riley.html.

[21] Pallets. Flask.
https://flask.palletsprojects.com/en/2.1.x/.

[22] Salesforce. Heroku: Cloud application platform.
https://heroku.com/.

[23] mihane-ichinose (Shitian Jin). Github - opartgenerator.
https://github.com/mihane-ichinose/OpArtGenerator.

[24] Kozea. Overview — cairocffi 1.3.0 documentation.
https://cairocffi.readthedocs.io/en/stable/overview.html.

[25] Rémi Mondenx. Why my flask backend is unstable on heroku.
https://stackoverflow.com/questions/62328835/why-my-flask-backend-is-unstable-on-heroku/62330039#62330039.

[26] Redis Ltd. Redis.
https://redis.io/.