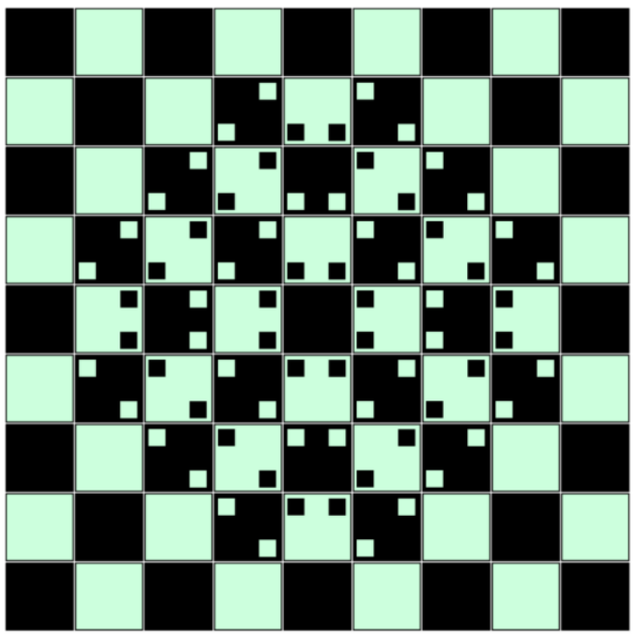
Stretched Chessboard

This is an interesting optical illusion in which a chessboard-like layout appears as if it has been stretched in the center, or it is being viewed from a lens of some sort. See below:



In reality, there is no stretching at all; it is a trick played on our eyes by the arrangement of the tiny squares inside the chessboard.

So, our task in this project is to essentially draw the above design. We should attempt to draw it using only two sprites (or costumes): a black and a white square.

Explore the program:

If you want to play with my final program to get a feel for this optical illusion, click the link specified at the end of the chapter. Try not to peek at the scripts yet, since we want to design them ourselves below.

Steps to run the program:

1. Click the Green flag.

Scratch and CS Concepts Used

When we design this program, we will make use of the following Scratch and CS concepts. Learn these concepts if you don't know them before proceeding further.

- Algorithms
  - Designing new algorithms
- Concurrency:
  - Synchronization using broadcasting
- Conditional statements:
  - Conditions: YES/NO questions
  - Relational operators (=, <, >, etc.)
  - Conditionals (IF)
  - Boolean operators (and, or, not)
- Data structures – list
  - Basic list operations
- Data types – basic
  - Integers
- Data types – strings
  - String operations
- Looping (iteration)
  - Looping – simple (repeat, for)
  - Looping – nested
  - Looping – conditional (repeat until)
- Pen Art
  - Stamp
- Procedures
  - User defined (custom)
  - Custom procedures with parameters
- Sequence
- Variables
  - Simple

High Level Design

As usual, let us take a step back and think about the big pieces of this program.

The first obvious step is to draw the chessboard-like layout. If you look carefully, it is not an 8x8 design, but a 9x9 design.

The next step is to figure out how to draw the tiny squares. One idea would be to develop a general-purpose algorithm to draw 4 tiny squares in the 4 corners of any square. This algorithm could then be modified to control, via an input parameter, which of those squares gets drawn. This input parameter could be a bit pattern, such as, "1010" in which 1 means the square is to be drawn and 0 means it is not.

Finally, we could set up a static list of 'bit patterns' which will tell us the expected "tiny square layout" for each and every cell of the chessboard.

Let us first design the data structures (i.e. data types and variables that will hold all the important information).

Data structures:

- Patterns (string): a comma-separated list of 81 strings (because there are 81 cells in the chessboard), each string is 4 letter long (e.g. 1010) and is the bit pattern for the corresponding cell in the chessboard. "Patterns" is a fixed string since we are going to draw only one chessboard layout.

- Bits (list): list of bit patterns: obtained from "Patterns" using a “split” procedure. We will borrow the “split” procedure from our library.

Feature Idea # 1: The chess board

Draw a 9x9 grid of chess-like cells.

Design:

For this, we will simply borrow an older program called "chessboard" (from the book "Practice CS Concepts with Scratch") and modify it to draw the 9x9 layout. We will use STAMP to draw the chessboard.

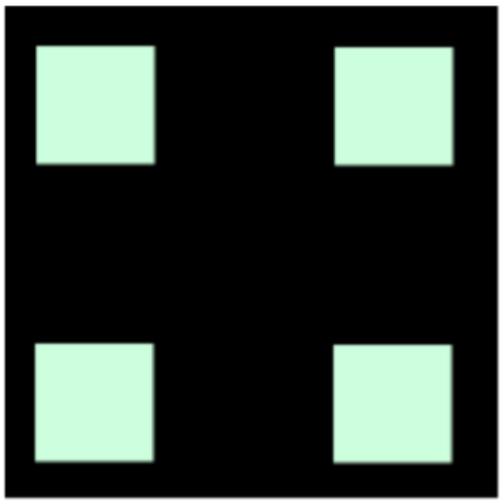
Feature Idea # 2: Tiny squares

Write a script that can draw 4 tiny squares in the 4 corners of a bigger square.

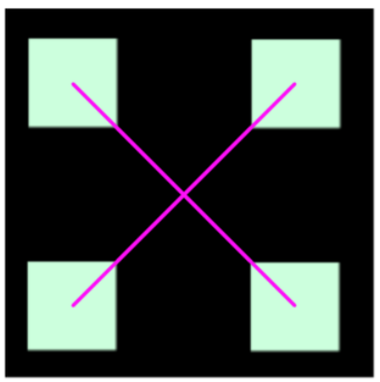
Design:

Let us do this in 2 steps:

Step 1: Draw the following pattern:



We already have 2 costumes for the white and black squares. Now we need an algorithm to draw smaller squares in the 4 corners of the bigger square. We will assume that the bigger square is already there and the costume has been switched to the opposite color. One idea is to jump to the 4 corners one by one, stamp, and return to the center. See the diagram below:



The purple lines indicate how the smaller square can jump from corner to corner.

Here is the algorithm:

```
Algorithm Tiny Squares
Resize to 1/4th original size
Calculate distance "d" from center to any one of the corners
Point towards any corner
Repeat 4
      Move d steps
      Stamp
      Move back d steps
      Turn 90
End repeat
Restore size and orientation
```

Step 2: Control which squares will be drawn.

We will modify the above algorithm by supplying an input parameter that contains a bit pattern, e.g. 1010. Starting with the square in the northwest corner and going clockwise, this parameter will determine which tiny squares would be visible.

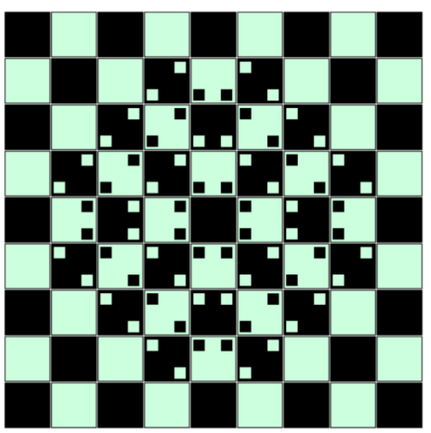
```
Algorithm Tiny Squares
Input parameter: Bits
Resize to 1/4th original size
Calculate distance "d" from center to any one of the corners
Point towards NW corner
I = 1
Repeat 4
      Move d steps
      If I'th bit in Bits is 1
            Stamp
      End if
      I = I + 1
      Move back d steps
      Turn 90
End repeat
Restore size and orientation
```

Feature Idea # 3: Stretch the chessboard

Stretch the chess board as shown in the figure at the top of this chapter.

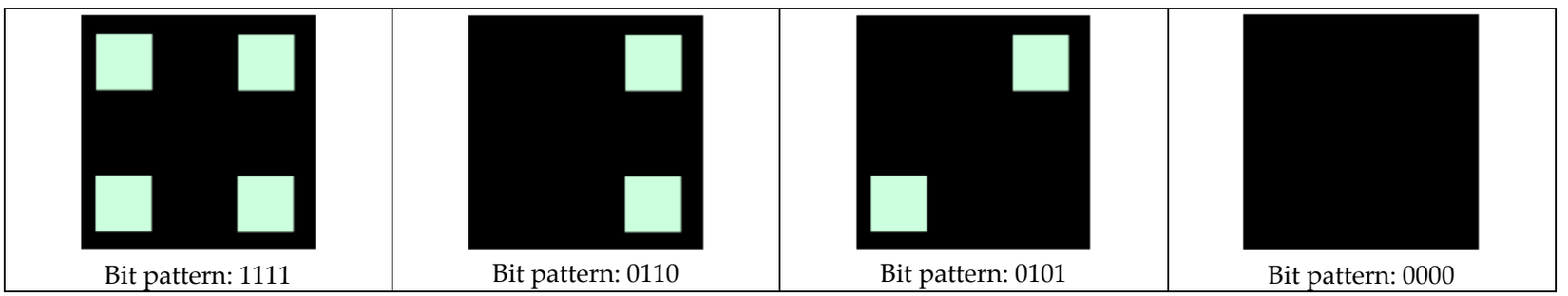
Design:

Here is the complete picture once again.



The remaining work involves drawing the tiny squares as shown in this picture. We already have a way to draw 4 tiny squares in a bigger square by using a bit pattern. Can we extend this idea for the final figure?

Here is one way we can do it. There are a total of 91 cells in this figure. We can see that each of these cells has some pattern of the tiny squares. See some examples below. The bit pattern describes which tiny squares are visible as seen clockwise starting from the NW corner.



Bit pattern: 1111 | Bit pattern: 0110 | Bit pattern: 0101 | Bit pattern: 0000

But how about tiny black squares in white cells? Well, we will let each cell figure it out by looking at its own color. The bit patterns would still be the same.

So, we could have a list of such bit patterns for the entire 9x9 chessboard. Here are all the 81 bit patterns for our optical illusion, which we will save in a string and convert to a list using the "split" function.

```
0000,0000,0000,0000,0000,0000,0000,0000,0000,
0000,0000,0000,0101,0011,1010,0000,0000,0000,
0000,0000,0101,0101,0011,1010,1010,0000,0000,
0000,0101,0101,0101,0011,1010,1010,1010,0000,
0000,0110,0110,0110,0000,1001,1001,1001,0000,
0000,1010,1010,1010,1100,0101,0101,0101,0000,
0000,0000,1010,1010,1100,0101,0101,0000,0000,
0000,0000,0000,1010,1100,0101,0000,0000,0000,
0000,0000,0000,0000,0000,0000,0000,0000,0000,

Bits = [ "0000", "0000", …, "0101", …, "1010", …, "0000" ]
```

And then, we will have a script that runs through this list and draws tiny squares in each cell according to that cell's bit pattern.

Here is the algorithm:

```
Algorithm draw optical illusion:
Input: "Bits" – list of 81 bit patterns
Pick the white costume (since the first cell is black)
I = 1
Go to first cell
Repeat 81
      Bit pattern BP = I'th item in Bits
      Call algorithm "draw tiny squares" with input BP
      Move to next cell
      Change costume
      I = I + 1
End repeat
```

Save the final version

Congratulations! You have completed all the features of the optical illusion. Compare your program with my program in the file below.

File: chessboard-stretch.

MIT website: https://scratch.mit.edu/projects/618773078/

How to run the program:

1. Click the Green flag.

Author: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 17 December 2021