Minecraft Beacon Color Computation

Damian Nel

January 5, 2025

Abstract

In the video game Minecraft there exists a feature called the beacon. The beacon is a block that illuminates a beam of white light indefinitely along the y axis. This white light can however be coloured by stacking stained glass blocks atop the beacon.

1 Introduction

The goal of this project is to create a program that computes the most effective combination of glass stained blocks up to permutations of 5 glass blocks. The main goal of this project is to become familiar with using github repositories and writing papers using overleaf.

2 Calculation of the new beacon beam colour

2.1 The equation

The RGB vector produced by a sequence of stained glass blocks can be described by the formula :

$$\vec{C} = \frac{1}{2^n}\left(\vec{C_0} + \sum_{i=1}^{n} 2^{i-1}\vec{C_i}\right)$$

n = Total number of glass blocks - 1
$\vec{C}$ = New RGB vector formed
$\vec{C_0}$ = RGB Vector of first block in sequence
$\vec{C_i}$ = RGB Vector of current block in sum or the i+1 block in the sequence

2.2 Example of equation application

The red, green, and purple stained glass blocks are placed atop a beacon in that order. Their respective RGB values in Minecraft, represented as row vector matrices (R,G,B), are as follows:

$$\text{red} = \begin{bmatrix}176 & 46 & 38\end{bmatrix} \text{ green} = \begin{bmatrix}94 & 124 & 22\end{bmatrix} \text{ purple} = \begin{bmatrix}137 & 50 & 184\end{bmatrix}$$

$$\vec{C} = \frac{1}{2^{3-1}}(\begin{bmatrix}176 & 46 & 38\end{bmatrix} + 2^{1-1}\begin{bmatrix}94 & 124 & 22\end{bmatrix} + 2^{2-1}\begin{bmatrix}137 & 50 & 184\end{bmatrix})$$

$$\vec{C} = \frac{1}{4}(\begin{bmatrix}176 & 46 & 38\end{bmatrix} + \begin{bmatrix}94 & 124 & 22\end{bmatrix} + 2\begin{bmatrix}137 & 50 & 184\end{bmatrix})$$

$$\vec{C} = \frac{1}{4}(\begin{bmatrix}176 & 46 & 38\end{bmatrix} + \begin{bmatrix}94 & 124 & 22\end{bmatrix} + \begin{bmatrix}274 & 100 & 368\end{bmatrix})$$

$$\vec{C} = \frac{1}{4}(\begin{bmatrix}544 & 270 & 428\end{bmatrix})$$

$$\vec{C} = \begin{bmatrix}136 & 67.5 & 107\end{bmatrix}$$

2.3 Understanding the formula

In Minecraft these colours are weighted geometrically, where the next colour weighs twice as much as the previous colour with the exception of the first and second colour having the same weighting. Thus the weighted sum(C) is as follows:

$$\vec{C} = 1 \cdot \vec{C_0} + 1 \cdot \vec{C_1} + 2 \cdot \vec{C_2} + 4 \cdot \vec{C_3} + 8 \cdot \vec{C_4} + ...$$

To make the pattern more obvious we can split up the Colour Vectors and Weights into a table.

$\vec{C_1}$	1
$\vec{C_2}$	2
$\vec{C_3}$	4
$\vec{C_4}$	8
$\vec{C_i}$	2^{i-1}

The weightings can naturally be generalised as $W = 2^{i-1}$ from the second position onward. We can subsequently rewrite the sum weighted in sigma notation:

$$\vec{C} = \vec{C_0} + \sum_{i=1}^{n} 2^{i-1} \vec{C_i}$$

For the final step we will refer to our example done supra. After applying the weightings to each vector and summing them up we are left with the weighted sum vector:

$$\vec{C} = \begin{bmatrix} 544 & 270 & 428 \end{bmatrix}$$

But we know that RGB vectors are $\begin{bmatrix} 0..255 & 0..255 & 0..255 \end{bmatrix}$, so why is this happening?

In our example above 3 RGB vectors are being added in the ratio: 1 unit red, 1 unit green, and 2 units purple. By adding these 3 colours in this ratio we have create a RGB vector that is $1 + 1 + 2 = 4$ units where a unit is 1 RGB vector. This means that my current weighted sum represents 4 RGB vectors. In order for it to represent 1 vector we simply divide by 4.

This process is called normalising the vector. To normalise the RGB vector we divide by the sum of the weights. We worked out the geometric equation of the weights from the second stained glass block onward to be:

$$W = 2^{i-1}$$

The sum of a geometric sequence can be found by $S_n = \frac{a(1-r^n)}{1-r}$ where n = number of terms
With some rearrangement the general form T_n can be achieved and we can just substitute into the formula for S_n.

$$W = 1.2^{i-1}$$

$$\sum_{i=1}^{n} W = \frac{1(1-2^n)}{1-2}$$

$$\sum_{i=1}^{n} W = -1 + 2^n$$

Including the weight of the first glass block $\vec{C_0}$

$$1 + \sum_{i=1}^{n} W = 1 - 1 + 2^n = 2^n$$

Therefore dividing our weighted sum $\vec{C} = \vec{C_0} + \sum_{i=1}^{n} 2^{i-1}\vec{C_i}$ by 2^n provides our final form:

$$\vec{C} = \frac{1}{2^n}(\vec{C_0} + \sum_{i=1}^{n} 2^{i-1}\vec{C_i})$$

This formula can then be described in python and has a time complexity of $O(n)$:

```
    def calculateBeamColor(glassBlocks):
        n = len(glassBlocks) - 1

        summed_Vector = list(glassBlocks[0]) #C0
        for i in range(1,n+1):
        #n + 1 is because python range() is exclusive[x,y)
            term = multiply_RGB_Vector(glassBlocks[i], 2**(i-1))
            summed_Vector = add_RGB_Vectors(summed_Vector, term)

        C = multiply_RGB_Vector(summed_Vector, (1/(2**(n-1))))
    return round_RGB_Vector(C)
```

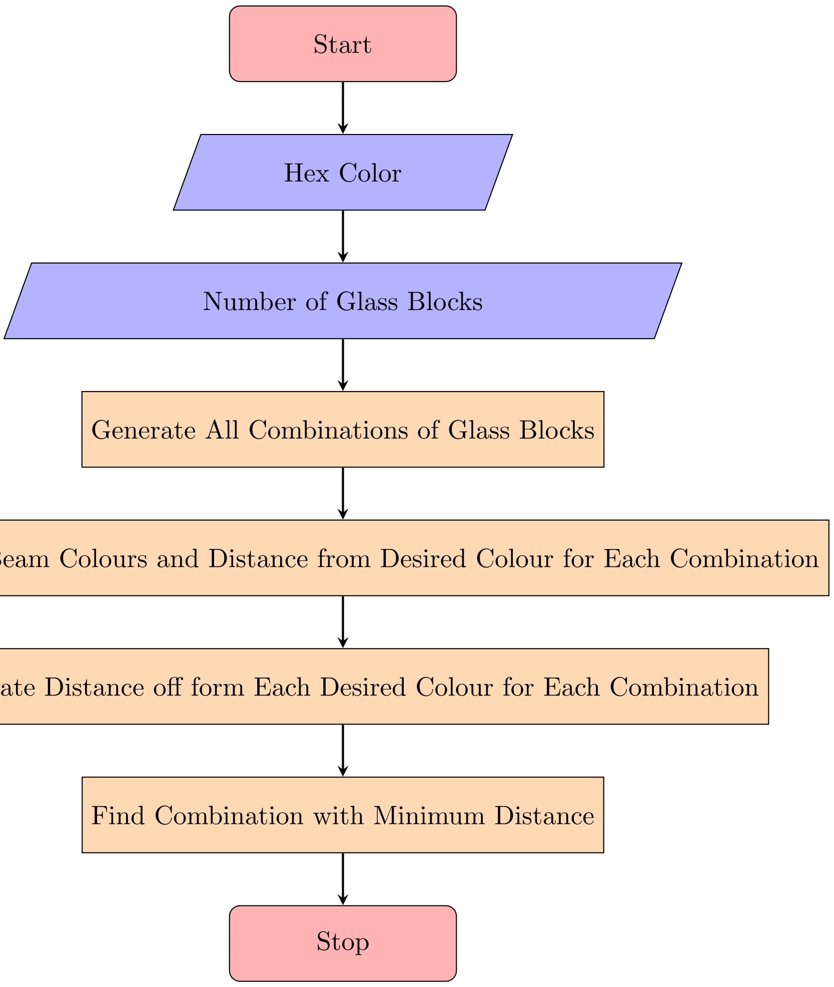
3 Initial Approach

3.1 Overview

The goal of the program is to generate a list names of stained glass blocks required to form a specific colour, where the user specifies how many glass blocks to use. I.e to form 3C6E43 with 4 glass blocks the program should output [”cyan”,”light blue”,”black”,”green”].

The initial approach is brute force computation, where every colour that can be made with n blocks is computed. The colours are then compared against the desired colour and the percentage match to the desired colour is stored. The program then retrieves the maximum percentage achieved and the respective list of RGB vectors.

3.2 Algorithm


Start
Hex Color
Number of Glass Blocks
Generate All Combinations of Glass Blocks
Calculate Beam Colours and Distance from Desired Colour for Each Combination
Calculate Distance off form Each Desired Colour for Each Combination
Find Combination with Minimum Distance
Stop

3.3 Analysis

The accuracy of the result represented by the vector distance($\bar{d}$) between the desired RGB vector and the calculated one. Understanding this distance is thus crucial to understanding the performance of the algorithm.
With RGB colours a total of $256^3 = 16777216$ different colours can be made. The program was made to work out the distance between the desired RGB and the outputted RGB for 16 589 random RGB values at n(number of glass blocks) depths of $n \in [1, 3]$. The average distance($\bar{d}$) of this computation represents the sample mean of the total sample space of 6 777 216 for n values of $n \in [1, 3]$. Additionally, the sample deviation of the set was calculated.

$$\bar{x} = \frac{1}{n} \sum i = 1^n x_i \qquad ; \qquad s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2}$$

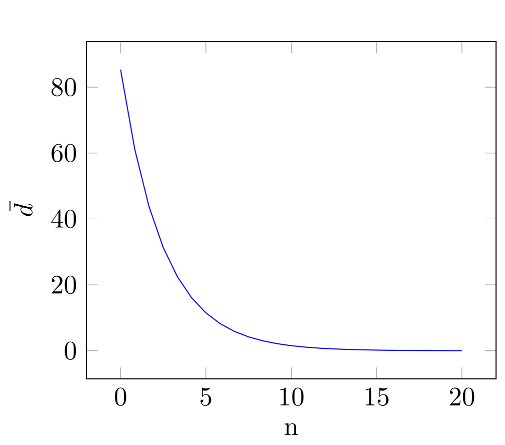
```
import random
for n in range(3):
    for i in range(16589):
        rgb.append((random.randrange(0, 256), random.randrange(0, 256),
           random.randrange(0, 256)))

    #average
    dist_sum = 0
    minDistList = []
    for color in rgb:
        bestOrder, minDist = FindGlassBlocks(n+1, color)
        minDistList.append(minDist) # for standard deviation
        dist_sum = dist_sum + minDist

    N = len(rgb)
    dBar = dist_sum/N

    #sample deviation

    squared_disance_sum = 0
    for distance in minDistList:
        squared_disance_sum += (distance-dBar)**2

    s = (1/(N-1) * squared_disance_sum)**1/2
    print(n+1,dBar,s)
```

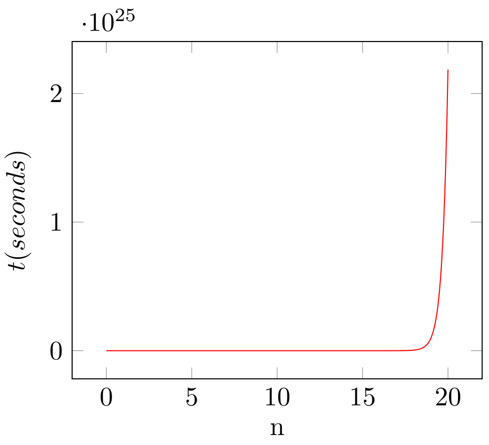
n	$\bar{x} = \bar{d}$	s
1	57.6231	220.4867
2	36.8312	205.1886
3	26.7076	254.8769

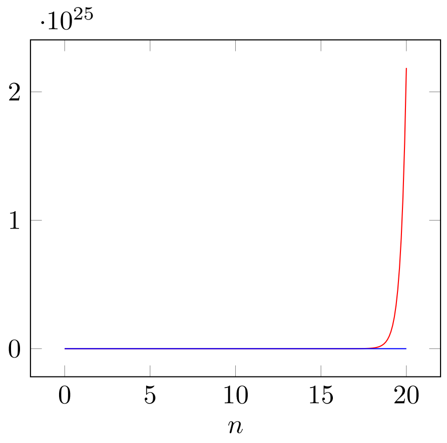
Table 1: Output of simulation to determine average vector distance at n

We can then plot the sample averages on a graph and perform an exponential regression in order to have an equation to represent the average distance in the form $\bar{d} = a(b)^{n-1}$ where n = number of glass blocks.

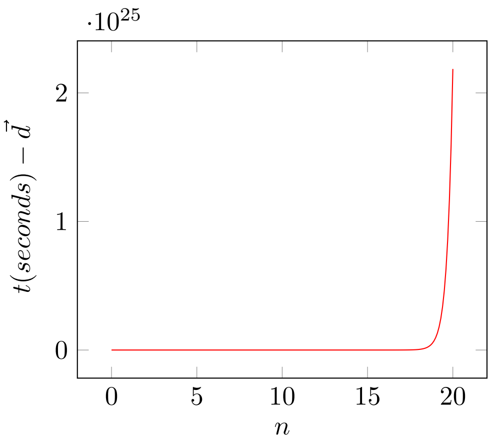
The following graphs represent the approximate average vector distance($\bar{d}$) achieved at each value of n. It can be observed that the rate at which the time function increases is drastically faster than the rate at which the approximate average vector distance is decreasing. Displaying the expensiveness of this kind of approach.







The expensiveness of a this process could be described as the rate at which time increases, less the rate at which the distance decreases. As shown below this brute force process is exponentially expensive and thus impractical for large values of n.



4 Annexture

4.1 Helper Functions

```
def multiply_RGB_Vector(color, multiplier):
    color = list(color)
    for i in range(3):
        color[i] = color[i] * multiplier
    return color
        #Time Complexity O(1)

def add_RGB_Vectors(vec1, vec2):
    new_RGB_Vector = [0,0,0]
    for i in range(3):
        new_RGB_Vector[i] = vec1[i] + vec2[i]
    return new_RGB_Vector
        #Time Complexity O(1)

def round_RGB_Vector(vec):
    for i in range(3):
        vec[i] = round(vec[i])
    return vec
        #Time Complexity O(1)
```

```
#finding average achieved for Table 1
import random
import time

start = time.time()
for i in range(16589):
    rgb.append((random.randrange(0, 256), random.randrange(0, 256), random.
        randrange(0, 256)))

rgbsum = 0
First = True
for color in rgb:
    bestOrder, minDist = FindGlassBlocks(3, color)
    rgbsum = rgbsum + minDist

end = time.time()
elapsed_time = end - start

ave_distance = rgbsum/len(rgb)
print(ave_distance, elapsed_time)
```

RGB Dictionary:
”white” : (249, 255, 254),
”light_Gray” : (157, 157, 151),
”gray” : (71, 79, 82),
”black” : (29, 29, 33),
”brown” : (131, 84, 50),
”red” : (176, 46, 38),
”orange” : (249, 128, 29),
”yellow” : (254, 216, 61),
”lime” : (128, 199, 31),
”green” : (94, 124, 22),
”cyan” : (22, 156, 156),
”light_Blue” : (58, 179, 218),
”blue” : (60, 68, 170),
”purple” : (137, 50, 184),
”magenta” : (199, 78, 189),
”pink” : (243, 139, 170)