CoE 164

Computing Platforms

Assessments Week 03

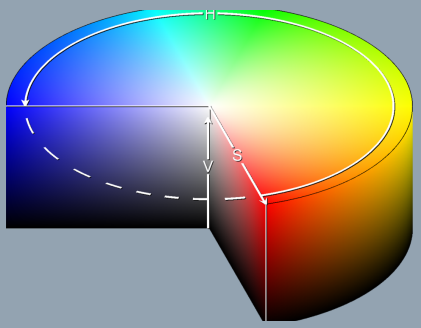
Academic Period: 2nd Semester AY 2022-2023
Workload: 3 hours
Synopsis: Rust generics and lifetimes

SE Week 03

This assessment will let you be familiar with applying object-oriented programming principles in Rust.

Problem Statement

Color is a visual perception of light. Humans recognize color as a spectrum of colors from red until violet. In analog and digital representation, colors are organized in a space used by certain devices to reproduce colors transmitted from elsewhere. Color spaces are the realization of a color model, which maps colors into a value of three tuples of numbers. In computing, the most common color models are the RGB (red, green, blue) and HSV (hue, saturation, value) models.



The RGB color model consists of three tuples of numbers (R, G, B) corresponding to the red, green, and blue values of a color, respectively. Each of these numbers are represented usually as an integer from 0 to 255 inclusive. This color model is formulated from the fact that red, green, and blue are the three basic colors perceived by the human eye. The RGB model is represented as a cube.

On the other hand, the HSV color model consists of three tuples of numbers (H, S, V) corresponding to the hue, saturation, and value of a color, respectively. The hue represents the color "type" corresponding to the color spectrum, the saturation represents the "colorfulness" of a hue corresponding to how dull or sharp the color is, and the value or brightness represents the perceived quantity of emission of light of a color. The hue is usually represented with an integer from 0 to 360 degrees inclusive while the saturation and value are usually represented with a decimal value from 0 to 1 inclusive. This color model is formulated to more accurately represent how the human eye perceives color in contrast to the physical representation of RGB. The HSV model is represented as a cylinder.

To convert an RGB color into HSV, it is necessary to first find the minimum M_{min} and maximum M_{max} values among the three RGB values, and the *chroma* C, or the range of these values.

$$M_{min} = \min(R, G, B)$$
$$M_{max} = \max(R, G, B)$$
$$C = M_{max} - M_{min}$$

After computation, we can now find the corresponding HSV tuple.

$$\alpha = 0.5(2R - G - B)$$
$$\beta = \frac{\sqrt{3}}{2}(G - B)$$
$$H = atan2(\beta, \alpha)$$
$$V = M_{max}$$
$$S = 0; V = 0$$
$$S = \frac{C}{V}; \text{otherwise}$$

Conversely, an HSV color can be converted into an RGB color by looking into the HSV cylinder and doing a geometric transform. Note that the modulo operator in $k(n)$ represents the floating-point remainder when the left value is divided by the right value.

$$f(n) = V - VS\,max(0, min(k, 4 - k, 1))$$
$$k(n) = \left(n + \frac{H}{60\text{deg}}\right) mod\ 6$$
$$(R, G, B) = (f(5), f(3), f(1))$$

You have encountered these definitions while working on a small web application that generates color palettes for graphic designers. When starting a design project, graphic designers would have to think of a set of prevailing colors in their design named a *color palette*. Selection of the colors is usually subjective. However, there are some basic rules designers follow to get the related colors governed by a system named *color theory*. Color theory looks at a color wheel, which is the same as the topmost face of the color cylinder in the HSV model. For the project, you only need to get three color schemes - chromatic, complementary, and triad colors. Chromatic colors are the same colors but have different values or brightness. Complementary colors are pairs of colors whose hue is 180 degrees apart. Finally, triad colors are three colors whose hues are 120 and 240 degrees apart.

For the module you are developing, you would like to make data structures handling the RGB and HSV values, and make routines that convert from one color model to another. In

addition, the module will also have routines that determine the chromatic, complementary, and triad colors relative to a single color.

Input

The input to the program is through a test suite, which initializes the RGB or HSV tuples and manipulates them.

Output

The output to the program is the result of the relevant manipulation. Please see and run the test suite for details.

Constraints

Input Constraints

$R, G, B, H \in Z$
$S, V \in R$
$0 \leq R, G, B < 256$
$0 \leq H \leq 360$
$0 \leq S, V \leq 1$

The number of chromatic colors requested is always at least two.

Note that all operations should be done using floating points and only rounded down to the nearest integer if the final value should be an integer.

You can assume that all of the inputs are well-formed and are always provided within these constraints. You are not required to handle any errors.

Functional Constraints

You are **required** to write the following structures and struct-specific methods in your code:

- `Rgb(u8, u8, u8)` - tuple struct corresponding to the red, green, and blue values, respectively
    - `fn max_rgb(&self) -> u8` - maximum value in RGB struct
    - `fn min_rgb(&self) -> u8` - minimum value in RGB struct
    - `fn chroma(&self) -> u8` - chroma of the RGB struct
- `Hsv(u16, f64, f64)` - tuple struct corresponding to the hue, saturation, and value, respectively
    - `fn get_k(&self, n: f64) -> f64` - implementation of the function $k(n)$

In addition, you are **required** to write the following traits and function signatures. The structs above should have these traits:

- `trait Color` - traits applied to color models for conversion between their values
    - `fn r(&self) -> u8` - red component
    - `fn g(&self) -> u8` - green component
    - `fn b(&self) -> u8` - blue component
    - `fn h(&self) -> u16` - hue component
    - `fn s(&self) -> f64` - saturation component
    - `fn v(&self) -> f64` - value component
- `trait ColorOps` - traits applied to color models for color manipulation
    - `fn chromatic(&self, n_steps: usize) -> Vec <Box <dyn Color> >` - vector with n_steps elements of chromatic colors related to the current color with the 0th index corresponding to a color with a value of zero and the (n_steps - 1)th index corresponding to a color with a value of one; colors are scaled linearly
    - `fn complement(&self) -> Box <dyn Color>` - complementary color of the current color
    - `fn triad(&self) -> [Box <dyn Color>; 3]` - triad of colors; input color at index zero, 120 degree color at index one, 240 degree color at index two

Failure to follow these functional constraints will mark your code with a score of zero.

Steps

1. Write your program in Rust. An entry point (i.e. `main` function) is not needed.
2. Download the corresponding test file named `w03c_tests.rs` and import the structs and traits into it. For this to work, make sure that all of the imports are publicly visible, your program is in the same directory as this test file, and your program has the filename `w03c.rs`.
3. Open a terminal and go to the directory where the test file is. Run the following command in your terminal to compile the tests.
   `rustc —test w03c_tests.rs`
4. Run the `w03c_tests` executable according to your operating system.
5. Submit a copy of the source code to the Week 03 submission bin. Make sure that you attach one (1) file in the bin containing the Rust source code with a .rs extension or .rs.txt extension (if UVLe doesn't accept .rs files).