CPSC 449 Assignment 1

Due: Monday, October 16, 2017

Sample Solution Length:

- Less than 100 lines to reach the A- level, including some comments
- Approximately 130 lines with the fill color being influenced by the position within the image, including some comments

Individual Work:

All assignments in this course are to be completed individually. Students are advised to read the guidelines for avoiding plagiarism located on the course website. Students are also advised that electronic tools may be used to detect plagiarism.

Late Penalty:

Late assignments will not be accepted.

Posting Submissions for Public Viewing:

This assignment includes a creative and artistic element. As a result, we are hoping to receive numerous interesting submissions that will be worthy of showing off. We plan to post the images that are created on the course webpage so that others can view them. Your image will be posted anonymously, unless you choose to include your name as part of the image that you create. Please do not put your student number on your image. If you are **not** willing to have your image included on the website then please send an email to ben.stephenson@ucalgary.ca clearly stating such when you submit your assignment.

Background:

Piet Mondrian (March 7, 1872 – February 1, 1944) was a Dutch painter who created numerous famous paintings in the early half of the previous century that consisted of a white background, prominent black horizontal and vertical lines, and regions colored with red, yellow and blue. Three examples are shown below:

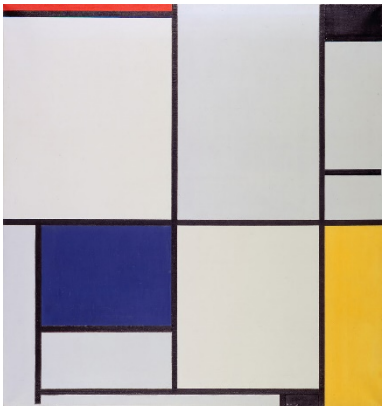
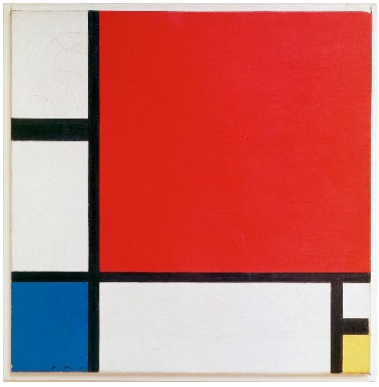
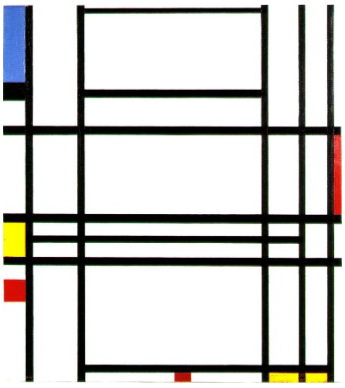


Tableau I, 1921



Composition II in Red, Blue, and Yellow, 1930



Composition No. 10, 1939-1942

Assignment Task:

Your task is to write a Haskell program that uses recursion to generate pseudo-random "art" in a Mondrian style. Your program's output will be an HTML document that contains rectangle primitives (perhaps among others) within an SVG tag. The following general strategy will be used to generate art in a Mondrian style:

If the region is wider than half the initial canvas size and the region is taller than half the initial canvas height:

Use recursion to split the region into 4 smaller regions (a vertical split and a horizontal split) with both split locations chosen randomly.

Else if the region is wider than half the initial canvas size:

Use recursion to split the region into 2 smaller regions using a vertical line with the split location chosen randomly.

Else if the region is taller than half the initial canvas size:

Use recursion to split the region into 2 smaller regions using a horizontal line with the split location chosen randomly.

Else if the region is big enough to split both horizontally and vertically, and both a horizontal and vertical split are randomly selected:

Use recursion to split the region into 4 smaller regions (a vertical split and a horizontal split) with both split locations chosen randomly.

Else if the region is wide enough to split horizontally, and a horizontal split is randomly selected:

Use recursion to split the region into 2 smaller regions using a vertical line with the split location chosen randomly.

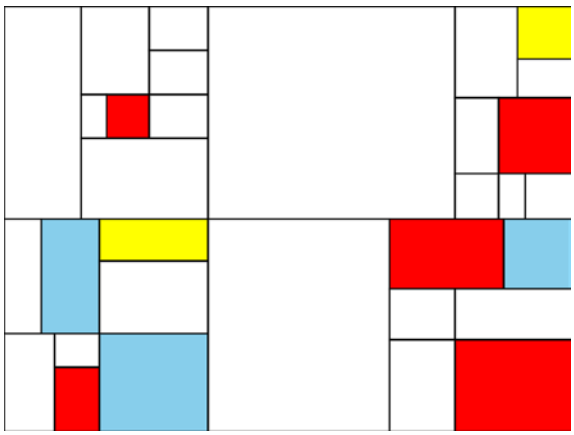
Else if the region is tall enough to split vertically, a vertical split is randomly selected:

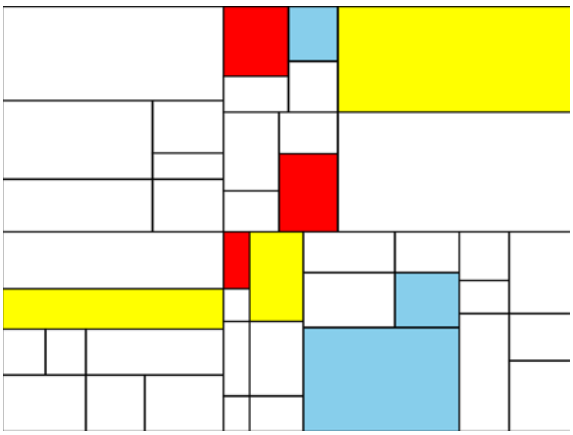
Use recursion to split the region into 2 smaller regions using a horizontal line with the split location chosen randomly.

Else:

Fill the current region (randomly, either white or colored, and if colored, with a random determination of red, blue or yellow).

A couple of images generated using this algorithm are shown below.





Use the following strategy when randomly deciding whether or not to split a region:

> Generate a random integer between 120 and the width of the region * 1.5.
> If the random integer is less than the width of the region then split the region.

While this strategy works, you might find yourself asking: Why is the random number between 120 and the width of the region * 1.5? By using 120 as the lower bound for the random number, we ensure that we never split a region that is less than 120 pixels wide (or tall when splitting in the other direction), and as such, we meet the constraint that the region is big enough to split (for my arbitrary definition of big enough). Selecting a random value that could be up to 1.5 * the width of the region, but then only performing a split when the random value is less than the width of the region, provides a random chance that a larger region will not be split into smaller regions.

Use the following strategy when splitting a region, either because it is so big that it will always get split, or because it was randomly selected to be split:

> Choose the split point, randomly, somewhere between 33% and 67% across the region (or down the region if splitting in the other direction). Choose two random split points when splitting both horizontally and vertically.
> Split the region into two smaller regions, one on the left and one on the right (or one on top and one on the bottom), or four smaller regions if splitting both horizontally and vertically
> Use recursion to fill / further split each new region

Use the following strategy to decide which color will be used to fill a region that will not be split further:

> Select a random value, r
> If r < 0.0833 then fill the region with red
> Else if r < 0.1667 then fill the region with skyblue
> Else if r < 0.25 then fill the region with yellow
> Else fill the region with white

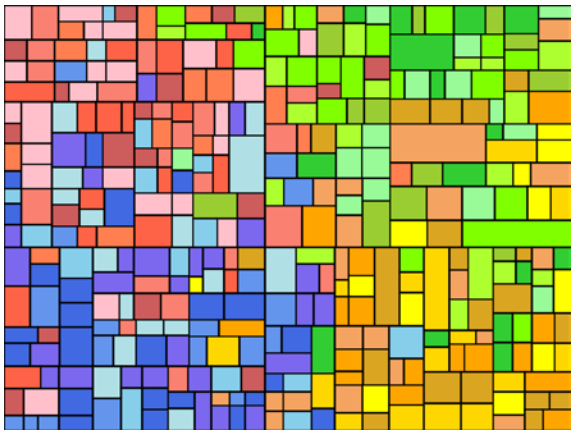
Save a copy of your program using a different name once you have the algorithm that I have provided working.

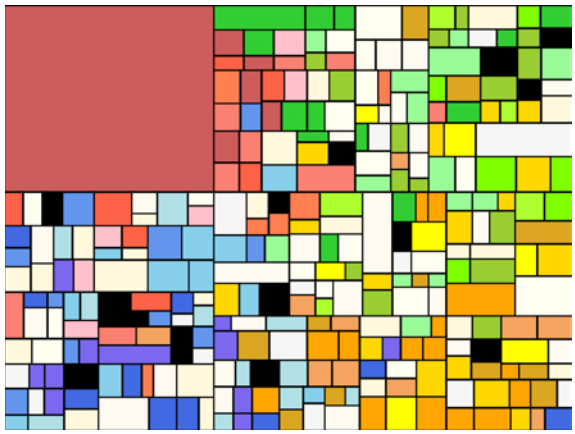
Once you have the provided algorithm working you are encouraged to adjust / expand / adapt this algorithm to generate art with your own specific style provided that is still at least vaguely Mondrian in style (meaning that it largely consists of horizontal and vertical lines and colored regions, at least the majority of which are rectangular). Ideas for customizing your work that you might want to consider include:

- Using lines of variable width
- Using a broader color palette than red, yellow and blue for the filled regions
- Changing the distribution of random numbers used when selecting the sizes of regions, colors (or anything else random). For example:
    - Instead of using numbers that are an even random distribution, you could add two random numbers together and divide by two to form a distribution that favours numbers in the middle of the random space.

    - Instead of allowing all possible values, reduce the space to numbers that are evenly divisible by 10 (or 20 or some other number) so that the random lines have more regular spacing to them.
- Using a patterned fill for some regions instead of only using solid fills
- Occasionally splitting a region into something other than rectangles
- Occasionally split a region into 3 smaller regions instead of 2 or 4

For inspiration, here are a couple of other images that I generated with different variations of my solution. In both of these images, the color used to fill the region is influenced (but not fully determined) by its location:





Implementation Details:

Randomness is an essential part of this assignment, but a function that returns a random number is contrary to the spirit of a pure functional language because a pure functional language requires each function to always return the same value when it is provided with the same parameters. While Haskell provides mechanisms for producing random numbers, they require concepts that we won't be exploring in detail in this course. To work around this limitation, I have provided some code for you to use on the course website. Specifically, I have provided code that generates a list of random numbers between 0 and 1, and then calls a (currently empty) mondrian function with a list of random values as one of its parameters. Each time you need a random number your program should take the first element from the list, and then pass the remaining elements in the list as a parameter to subsequent function calls. Similarly, one of the values returned by the mondrian function is a list. The returned list should only contain values that have not yet been used. Use each random number only once when implementing my algorithm. (It's ok to use the same random value multiple times in the A / A+ part of the assignment if that helps you achieve a desired artistic effect).

By default, the provided code will generate a different sequence of random numbers each time it is run. This can make debugging difficult. If you want the same sequence of random numbers each time your program runs, comment out "seed <- randomRIO (0, 100000 :: Int)" in main and replace it with "let seed = 0" (or let seed = some other integer).

As you implement the mondrian function (and additional helper functions), you will likely find yourself needing to write mathematical expressions that mix integers and floating point numbers. To convert

from an integer to a floating point number, use fromIntegral – it takes an integer as a parameter and returns the equivalent floating point number. To convert from a floating point number to an integer, use round – it takes a floating point number as a parameter and returns the closest integer.

Grading:

A base grade will be determined for your assignment based on its overall level of functionality, as shown in the table below. Then that base grade may be reduced if your implementation fails to use functional programming constructs in a reasonable manner, has stylistic deficiencies or other undesirable behaviour. Examples of stylistic deficiencies and other undesirable behaviour include (but are not limited to):

- Repeated code
- Magic numbers
- Missing or low quality comments
- Poor function / parameter names
- Crashing
- Generating useless output (such as rectangles that are outside of the canvas)

You **must** implement the basic algorithm described in this document **and save and submit a version of your code that implements the basic algorithm.** Then you are free to go on and modify / extend your program to demonstrate your artistic talents. Failure to submit a version of the program that implements the provided algorithm will cause us to assume that any differences occurred because you weren't able to implement the algorithm correctly rather than due to artistic choice.

Base Grade	Description
D	Uses recursion to generate a non-trivial output file that can be rendered in the browser. The generated image varies in structure (not just color) based on random values.
C	Successfully implements a significant part of the provided algorithm, but is generating wrong or incomplete results.
B	Generally implements the algorithm outlined earlier in the assignment description, but the implementation has a small shortcoming like missing one case or re-using the same random number in some situations.
A-	Implements the approach outlined earlier in the assignment description to generate images that are consistent with those shown in the assignment handout.
A	Two versions of the program are submitted. One version meets the requirements for an A- while the second version implements modest extensions / improvements to the approach described in the assignment handout to generate something that has a Mondrian style. Note that simply using random colors isn't a sufficient extension as random colors were demonstrated in class.
A+	Two versions of the program are submitted. One version meets the requirements for an A- while the second version substantially extends / improves the approach described in the assignment handout to generate something that is artistically impressive and has a Mondrian style

Note that Mondrian style will be interpreted relatively loosely in the A / A+ grade categories. You should feel free to experiment and explore, both with the features of Haskell as a programming language, and with your artistic style.

Submission Instructions:

Submit your .hs file (or files if you attempt the A / A+ part) to the Assignment 1 drop box in D2L. Since the results from your program are random, you may also want to submit the html file for your favourite image (or two) that your program has generated because we will (probably) not see the same image(s) when we test your program.