SpinnerLab (1.0)

Kcits970

February 2, 2022

Sections

Section 1. Description

SpinnerLab is an event-driven interface for generating crease patterns of origami polygonal flasher spinners. The demonstration of this program is available here.

This program is inspired by OrigamiTourist2010[1]'s Pentagonal Flasher Spinner[2] and Tridecagonal Flasher Spinner[3]. I would also like to give special thanks to TheOrigamiVoyeur[4] for providing a tutorial[5] of the Hexagonal Flasher Spinner, as it gave me the idea of how to expand the mathematics for an arbitrary number of sides.

The purpose of this document is to explain the important parts of the code, as well as the mathematics of drawing the crease pattern.

The program is written in Java language, and compiled using Java Compiler 15.0.1. Attempts to compile the given source code with a different compiler may result in unexpected or undefined behavior.

[1] OrigamiTourist2010: https://www.youtube.com/channel/UC2ErVsuiO_W_sF8p5N0Pw-w
[2] Pentagonal Flasher Spinner: https://www.youtube.com/watch?v=kl1zcjbjcdg
[3] Tridecagonal Flasher Spinner: https://www.youtube.com/watch?v=sf810F84OOg
[4] TheOrigamiVoyeur: https://www.youtube.com/channel/UCcIwFxBVr4iqmFMXPXtgBWg
[5] Hexagonal Flasher Spinner Tutorial: https://www.youtube.com/watch?v=4o-ozyRrRbE

Section 2. Code Documentation

This section is meant to document parts of the code and how they operate. The purpose of writing this information is to aid for any future updates that I may make, whether its refactoring or adding new features.

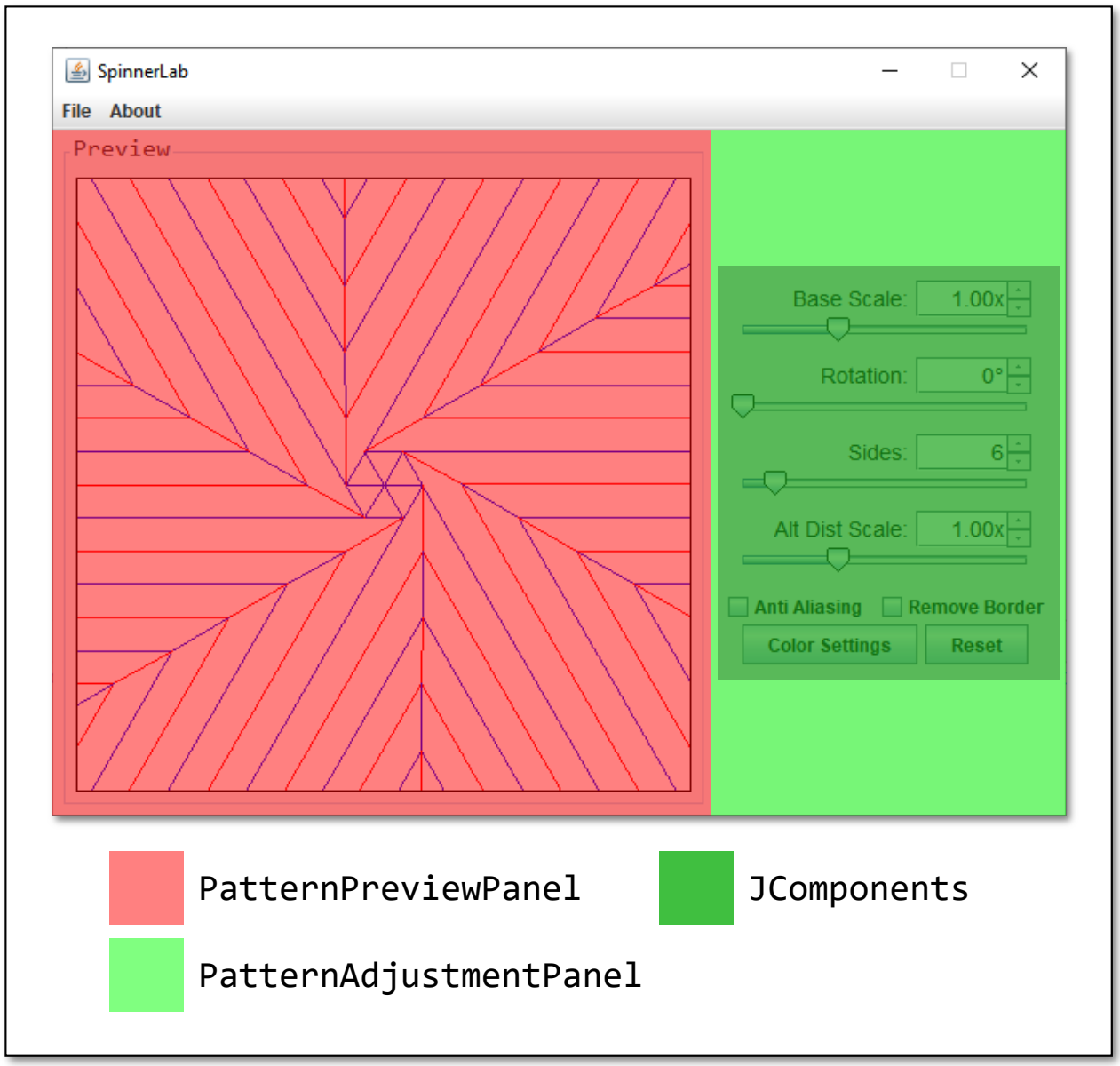
The program consists of 27 source files, distributed in 7 packages.

Package Name	Number of Files	Description
constants	1	Includes constant values referring to project data.
frame	3	Includes GUI-related classes.
frame.dialog	4	Includes GUI-related classes that extend dialogs or operate inside dialog components.
frame.main	4	Includes GUI-related classes that operate inside the outermost frame/window.
image	2	Provides classes for rendering images to an external file.
pattern	4	Provides classes for rendering a pattern onto a specified graphic.
util	8	Includes miscellaneous classes that provide utility functions.

The last source file is “Launcher.java”. Its only purpose is to launch the program, so it is not included in any packages.

Main Frame Structure

I’ll begin by explaining the structure of the main frame. `MyFrame` represents the outermost frame. The `ContentPane` of `MyFrame` contains 2 panels, which are `PatternPreviewPanel` and `PatternAdjustmentPanel`. `PatternPreviewPanel` is simply an extended `JPanel` object that renders a preview of a pattern onto itself. `PatternAdjustmentPanel` contains multiple swing components that modify the pattern upon the occurrence of a specific action. Any adjustments made with the components in `PatternAdjustmentPanel` is immediately reflected onto `PatternPreviewPanel`.



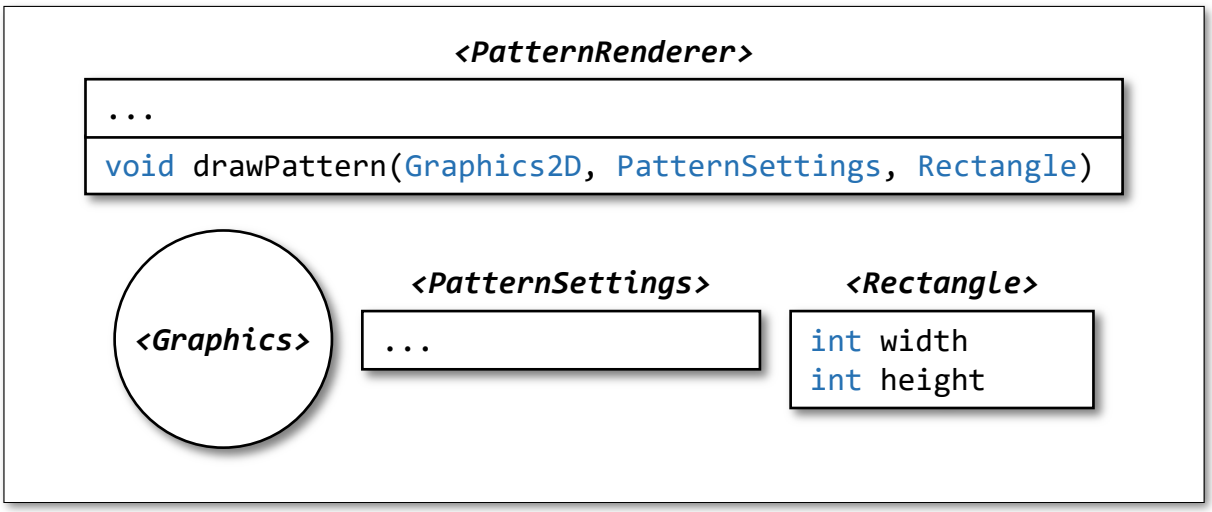
(Figure 1: Parts of frame.main.MyFrame)

Pattern Settings

In order to draw a pattern onto a certain graphic, you need an instance of `PatternSettings` and `PatternRenderer`. The `PatternSettings` object contains the needed information to draw a specific pattern. The `PatternRenderer` then takes the `PatternSettings` object as well as a Rectangle object that denotes

the render area via its drawPattern[6] method, which draws the pattern onto the specified graphic.

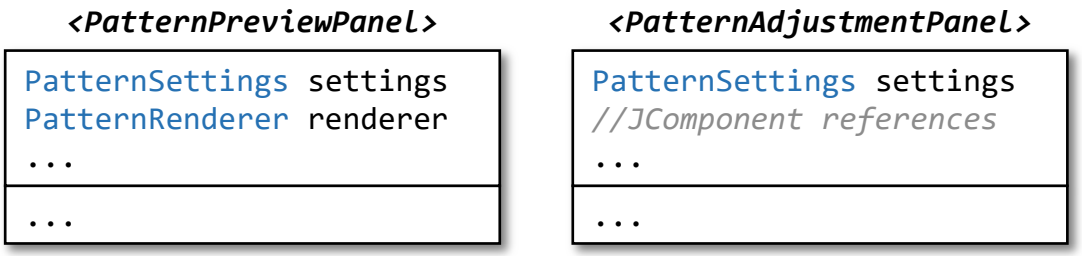
Variables of PatternSettings		
Type	**Name**	**Corresponding Setter Method**
double	baseScale	void setBaseScale(double)
int	rotation	void setRotation(int)
int	numOfSides	void setNumOfSides(int)
double	altDistScale	void setAltDistScale(double)
boolean	antiAlias	void setAntiAlias(boolean)
boolean	removeBorder	void setRemoveBorder(boolean)
Color	mountainColor	void setMountainColor(Color)
Color	valleyColor	void setValleyColor(Color)
Color	borderColor	void setBorderColor(Color)
Color	backgroundColor	void setBackgroundColor(Color)



(Figure 2: pattern.PatternRenderer.drawPattern(...))

PatternPreviewPanel contains the instance of PatternSettings and PatternRenderer, because it needs to draw the current pattern onto itself. As for PatternAdjustmentPanel, it contains the instance of PatternSettings, because the adjustments of its components need to be reflected in the settings object. It's important to note that PatternAdjustmentPanel and PatternPreviewPanel must share the same instance of PatternSettings in order for the adjustments to be reflected.

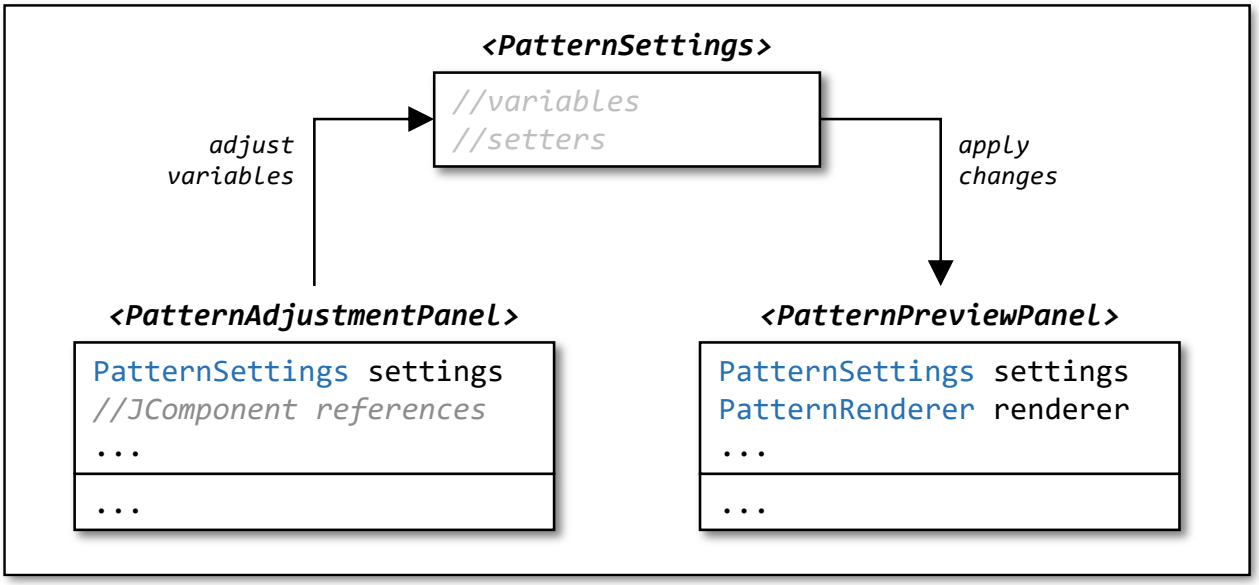
[6] PatternRenderer.drawPattern(Graphics2D, PatternSettings, Rectangle)



(Figure 3: PatternPreviewPanel and PatternAdjustmentPanel)

The Observer Pattern

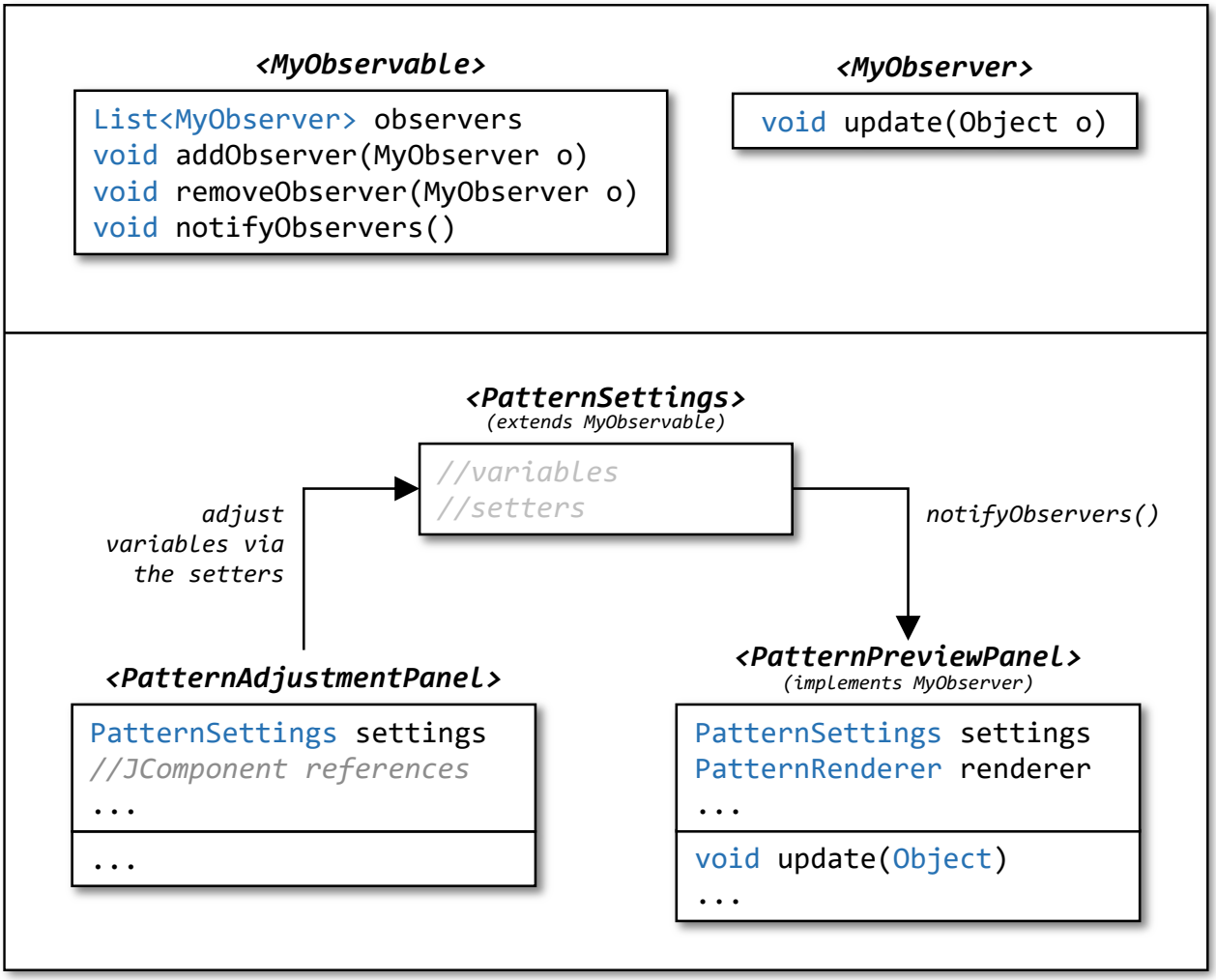
Remember the relation between `PatternAdjustmentPanel` and `PatternPreviewPanel`. The preview panel displays the pattern with the adjustments made in the adjustment panel. Notice how the settings object acts as a medium between the two panels. The adjustment panel modifies the settings object, and the preview panel redraws the pattern with the updated settings.



(Figure 4: Relation of PatternAdjustmentPanel, PatternSettings, and PatternPreviewPanel)

To ensure that `PatternPreviewPanel` gets updated with the latest settings, `PatternSettings` and `PatternPreviewPanel` are linked via the observer pattern. Each component of `PatternAdjustmentPanel` has a listener registered to it that calls the corresponding setter method of `PatternSettings`. The setter

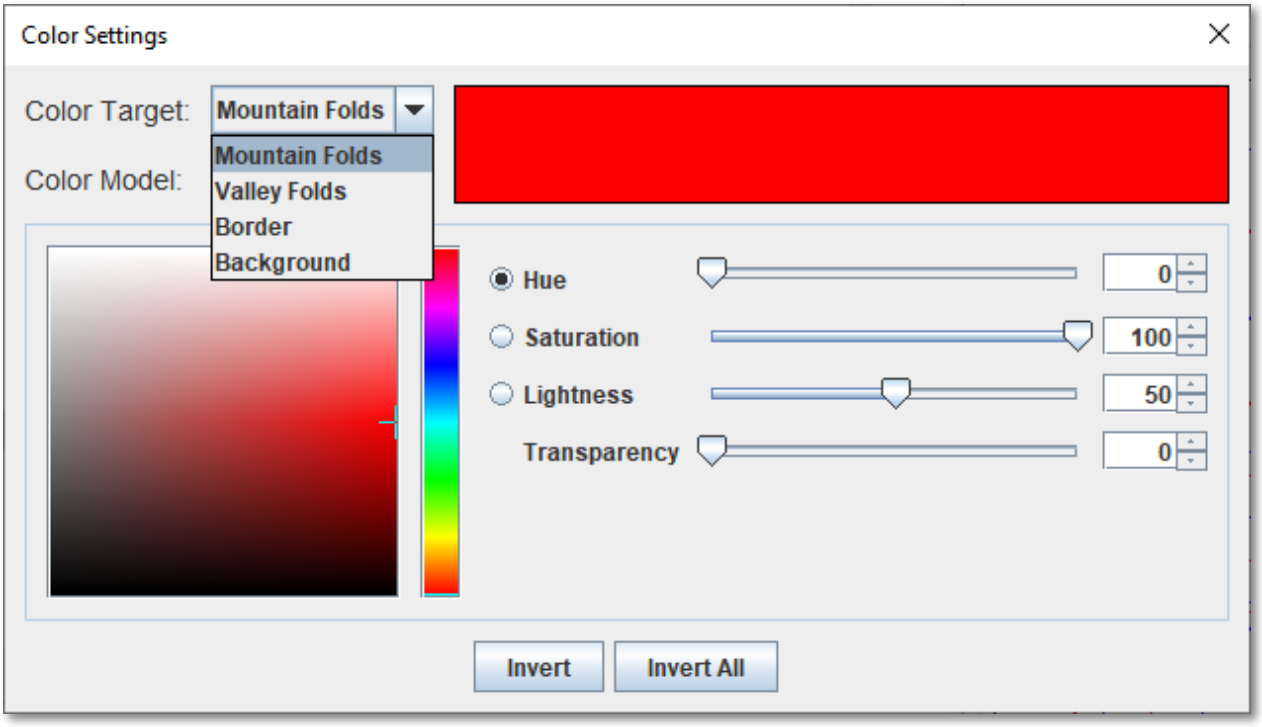
methods then call `notifyObservers()`, which consequently calls the observers' `update()` method.



(Figure 5: The observer pattern applied to figure 4.)

The Color Dialog

Because I didn't want to make the adjustment panel overly crowded with components, I created a special dialog for color adjustments, namely the `PatternColorDialog`. `PatternColorDialog` functions exactly the same as `PatternAdjustmentPanel`. It modifies the settings through the setter methods of `PatternSettings`, except that the way of modifying is structured differently.



(Figure 6: The combo box component of frame.dialog.PatternColorDialog)

The target of the color can be specified via the combo box. When changing the color, I wanted to avoid the use of large `if-else` blocks such as the code below.

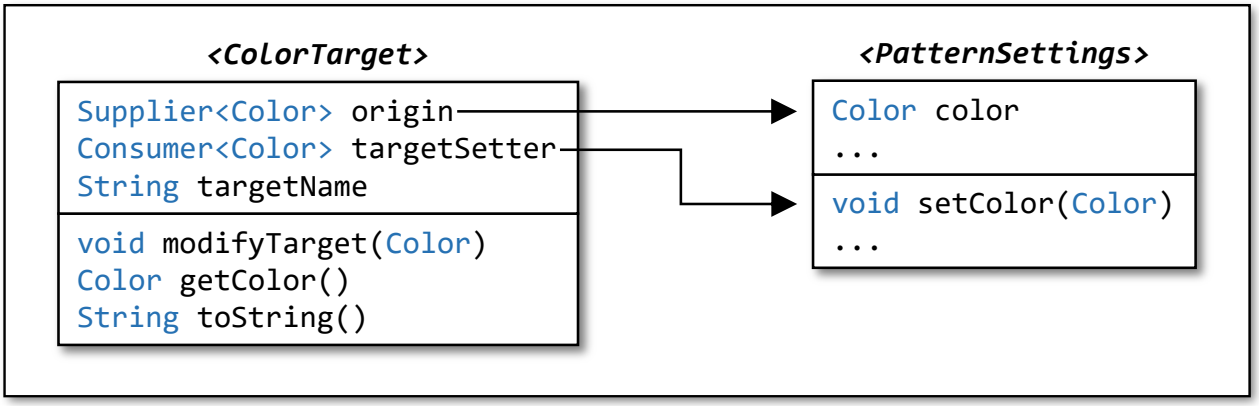
```
if (comboBox.getSelectedItem().toString().equals("Mountain Folds")) {
    //setting the color of mountain folds
} else if (comboBox.getSelectedItem().toString().equals("Valley Folds")) {
    //setting the color of valley folds
} else if (comboBox.getSelectedItem().toString().equals("Border")) {
    //setting the border color
} else {
    //setting the background color
}
```

To avoid the use of such design, I implemented a class called `ColorTarget`. `ColorTarget` uses a functional interface to retrieve and directly modify colors without the need of conditional statements.

The ColorTarget Class

`ColorTarget` takes 3 arguments to construct itself, which are the source of the color, the setter of the target color, and a string representation of the target. The source is an

instance of `Supplier<Color>`, and the target setter is an instance of `Consumer<Color>`.



(Figure 7: The functionality of origin and targetSetter.)

`origin` and `targetSetter` are similar to pointers in C. The purpose of `origin` is to point to the color variable defined in `PatternSettings`, and the purpose of `targetSetter` is to point to the corresponding setter of the target color. The implementation of this structure successfully eliminates the need of conditional statements, because the combo box now contains `ColorTarget` objects.

```
colorTargets = new JComboBox<>();
colorTargets.addItem(new ColorTarget(() -> settings.mountainColor, c -> set-
tings.setMountainColor(c), "Mountain Folds"));
colorTargets.addItem(new ColorTarget(() -> settings.valleyColor, c -> settings.set-
ValleyColor(c), "Valley Folds"));
colorTargets.addItem(new ColorTarget(() -> settings.borderColor, c -> settings.set-
BorderColor(c), "Border"));
colorTargets.addItem(new ColorTarget(() -> settings.backgroundColor, c -> set-
tings.setBackgroundColor(c), "Background"));
```

The currently selected target can be edited via a single line.

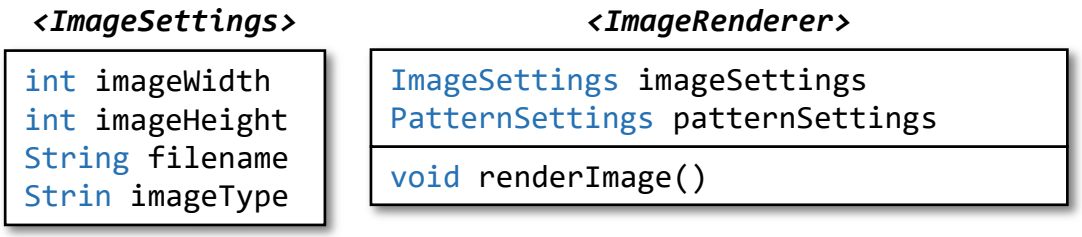
```
((ColorTarget) colorTargets.getSelectedItem()).modifyTarget(colorChooser.getColor());
```

Image Rendering

In order to write the current pattern into an external file, you need an instance of `ImageSettings` and `ImageRenderer`. `ImageSettings` simply contains information of image dimensions, image filename, and the type[7] of the image. The `ImageRenderer`

[7] PNG, JPEG, GIF, etc.

takes the ImageSettings object and the PatternSettings[8] object to write the image to an output file. The structure of ImageSettings and ImageRenderer is similar to the structure of PatternSettings and PatternRenderer.



(Figure 8: ImageSettings and ImageRenderer. Refer to figure 2 and 3 to check for similarities.)

The instance of ImageSettings is returned from frame.dialog.SaveAsDialog.getSaveSettings(). The renderImage() method of ImageRenderer is called from the ActionListener of frame.main.MyFrame.MyMenuBar.saveMenuItem.

[8] It's important to note that PatternPreviewPanel, PatternAdjustmentPanel, and ImageRenderer all need to use the same PatternSettings instance. If any one of the classes refer to a different instance, then the pattern would not be synchronized.

Section 3. Mathematics of the Pattern

This section explains the mathematics of drawing the generalized crease pattern.

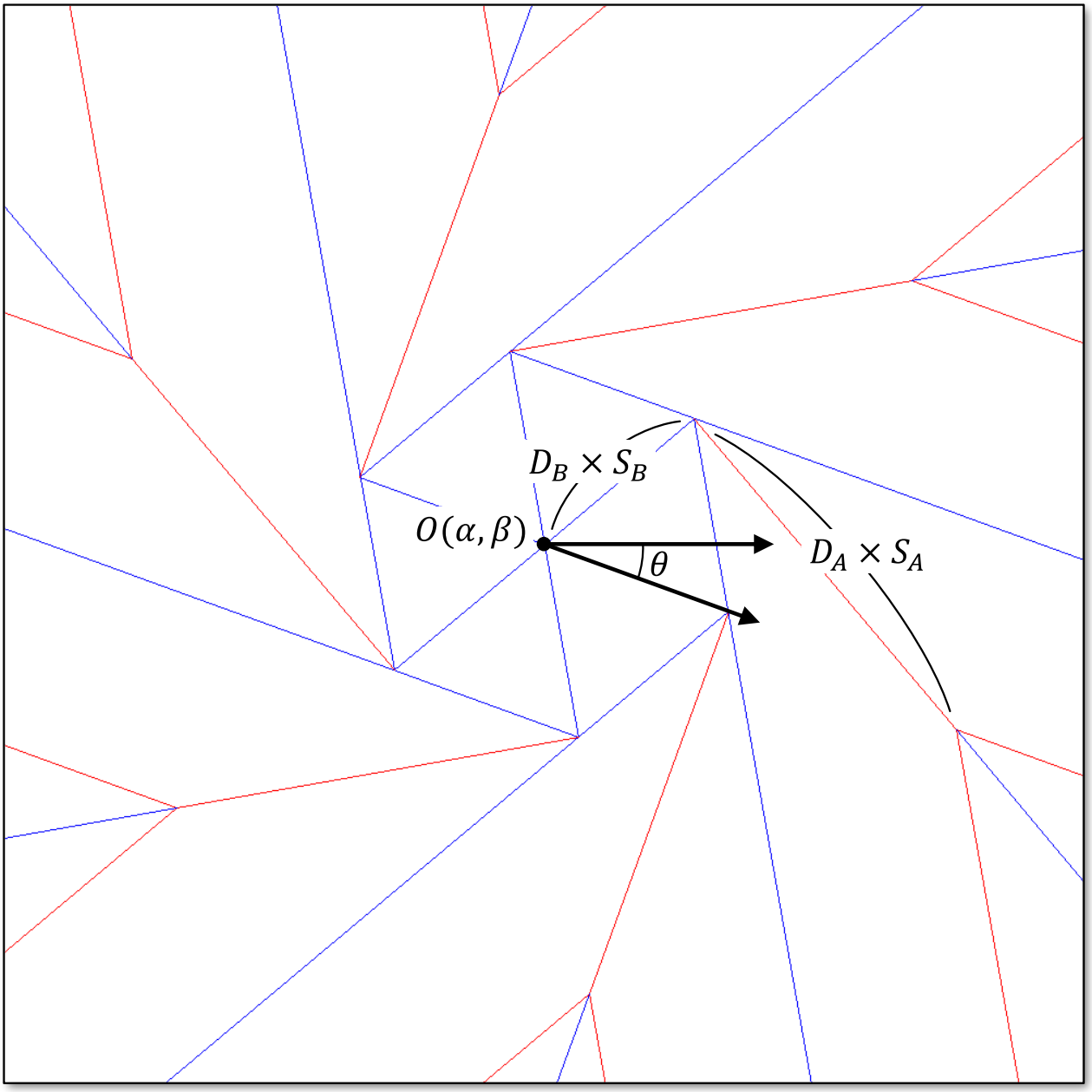
Pattern Terminology

There are a few terminologies to cover before diving into the mathematics. Note that these words are not globally accepted terms. They're merely a string of words that I applied to certain parts of the crease pattern, for the sake of convenience.

Terminology	**Description**
Origin	The center point of the crease pattern.
Base Polygon	The smallest polygon located at the origin.
Base Vertex Dist	*Base Vertex Distance*; Defines the distance of one of the vertices of the base polygon from the origin.
Alt Dist	*Alternation Distance*; Defines the distance of the alternating mountain folds and valley folds.
Base Scale	Base Vertex Distance Scale; The scalar(or multiplier) of the base vertex distance.
Rotation	The rotation of the pattern.
Sides(Edges)	The number of sides of the base polygon.
Alt Dist Scale	Alternation Distance Scale; The scalar(or multiplier) of the alternation distance.

I've assigned a simplified name(or character) for some of these terminology for convenient use.

Terminology	**Simplified Name**
Origin	$O(\alpha, \beta)$
Base Vertex Dist	D_B
Alt Dist	D_A
Base Scale	S_B
Rotation	θ
Sides(Edges)	N
Alt Dist Scale	S_A

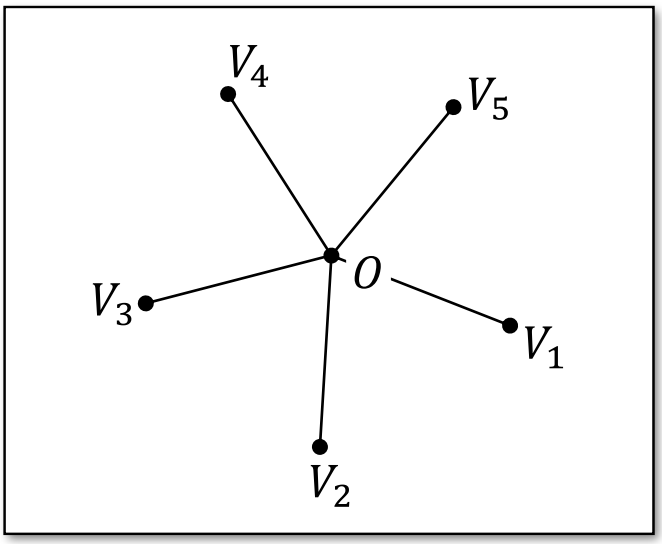


(Figure 9: Representation of the simplified names on the crease pattern. The rotation is measured backwards because the direction of the Y-axis differs in Java graphics.)

Drawing the Base

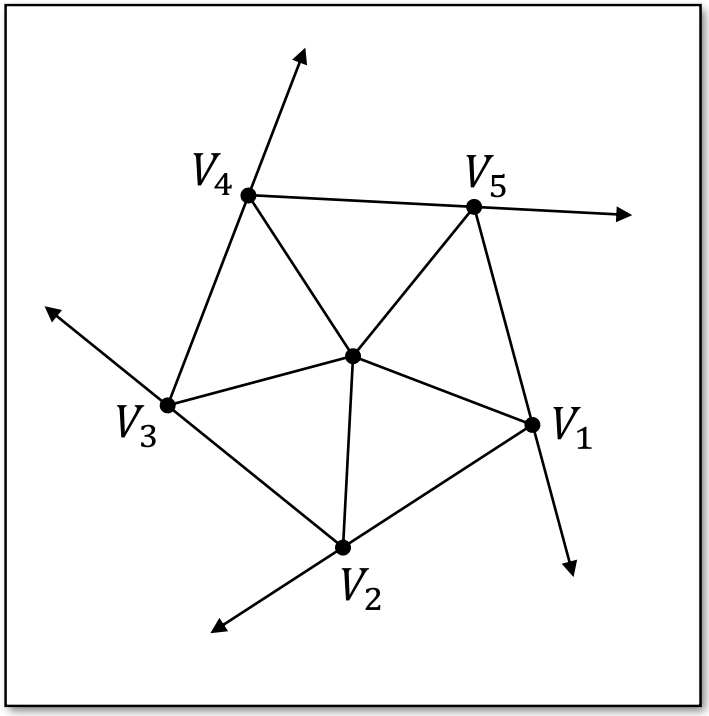
We start by plotting the vertices of the base polygon. Since the actual distance of each vertex from the origin is $D_B \times S_B$, we can use the trigonometric functions to calculate the x and y of the nth vertex.

$$V_n = \left(\alpha + D_B \times S_B \cdot \cos\left(\frac{2n\pi}{N} + \theta\right), \beta + D_B \times S_B \cdot sin\left(\frac{2n\pi}{N} + \theta\right)\right)$$



(Figure 10: Plot of the nth vertex of the base polygon.)

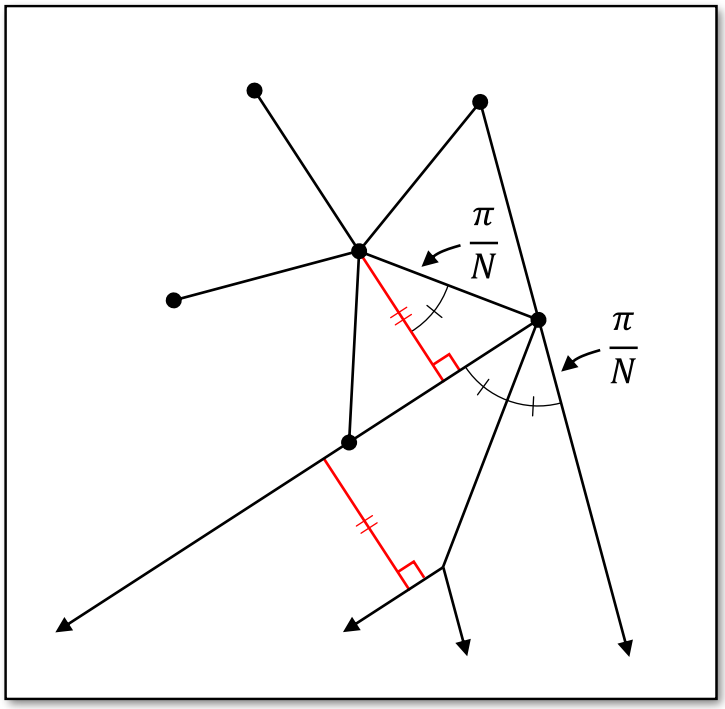
The next step is to draw a ray that goes through the adjacent vertices. In the case of figure 10, the rays would be notated as: $\overrightarrow{V_1V_2}$, $\overrightarrow{V_2V_3}$, $\overrightarrow{V_3V_4}$, $\overrightarrow{V_4V_5}$, and $\overrightarrow{V_5V_1}$.



(Figure 11: Plot of the extending rays of the adjacent vertices.)

Calculating the Alt Dist

After completing the base, we continue by drawing the alternating folds. First, we must calculate the alt dist of the pattern. If we ignore the value of the alt dist scale, then the distance of each parallel alternation is the same as the distance between the extending ray and the origin. This allows us to calculate the default alt dist value using trigonometry.



(Figure 12: Angles of the base.)[9]

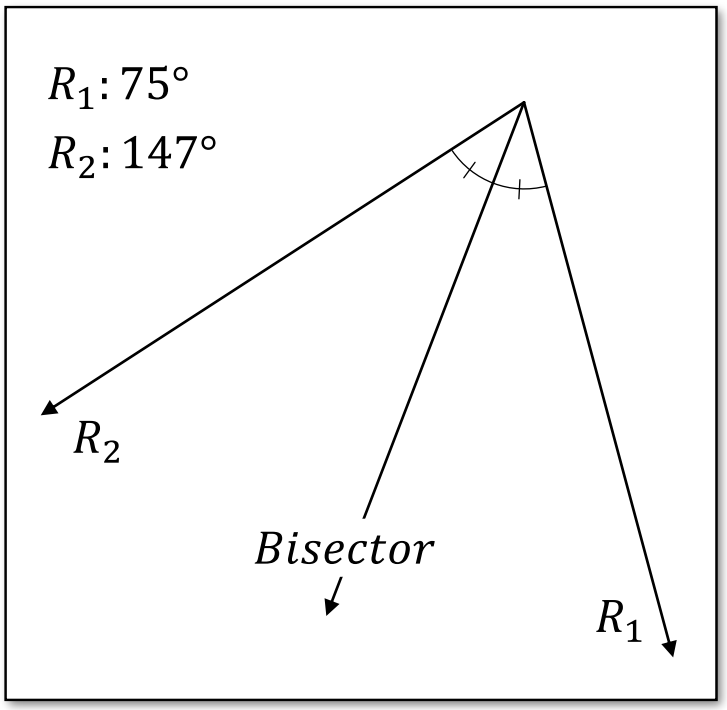
Through figure 12, we see that the alt dist(D_A) is equal to $D_B \times S_B \cdot \cot\left(\frac{\pi}{N}\right)$. Multiplying the alt dist scale(S_A) gives us the full formula of the alt dist.

$$D_A = S_A \times (D_B \times S_B) \cdot \cot\left(\frac{\pi}{N}\right)$$

Finding the Bisector Angle

[9] While it is trivial, it should be noted that the alternating rays are parallel to the base rays.

Drawing the alternating creases requires the two surrounding base rays. We'll denote the first ray as R_1, and the second ray as R_2. From the two surrounding rays, we need the angle(or direction) of the ray that divides the smaller angle between R_1 and R_2. To find this angle, we need a conditional formula.



(Figure 12: The bisector of the surrounding rays.)

$$if\ |angle(R_1) - angle(R_2)| < \pi$$

$$angle(Bisector) = \min\big(angle(R_1), angle(R_2)\big) + \left|\frac{angle(R_1) - angle(R_2)}{2}\right|$$

$$else\ if\ |angle(R_1) - angle(R_2)| = \pi$$

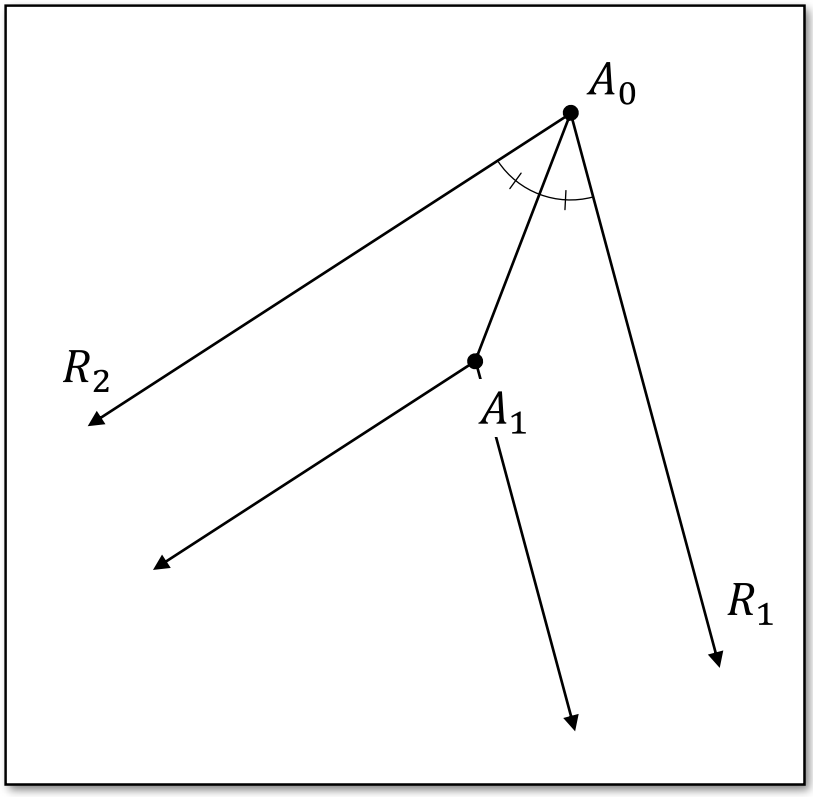
$$angle(Bisector) = undefined$$

$$else$$

$$angle(Bisector) = \min\big(angle(R_1), angle(R_2)\big) + \left|\frac{angle(R_1) - angle(R_2)}{2}\right| + \pi$$

Drawing the Alternating Creases

The only remaining task is to draw the alternating folds. We'll start by denoting the alternating origin as A_0 and the first alternation point as A_1.



(Figure 13: The alternating origin and the first alternation point.)

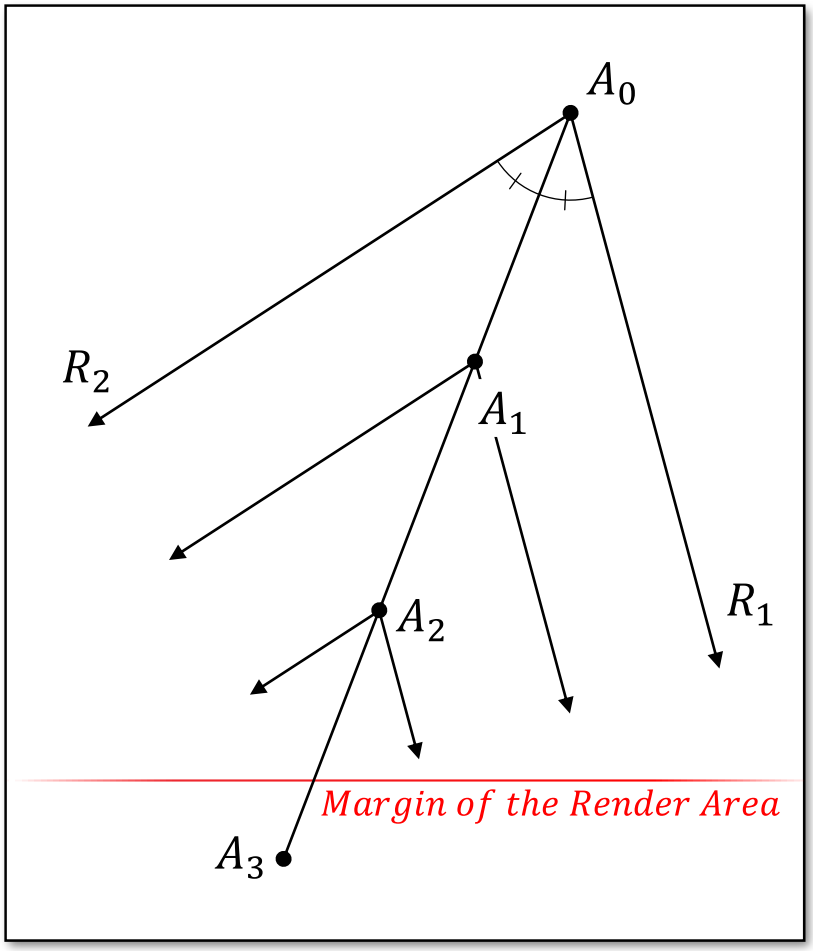
From here, we can draw 2 parallel rays from A_1, and repeat plotting the alternation points until it reaches the outside of the render area. *(If the render area is the whole grid, then this process would loop infinitely.)*

Finding the x and y of A_1 can be done with trigonometry. Keep in mind that A_0 is simply one of the base vertices, and the value of D_A and $angle(Bisector)$ have been calculated previously.

$$A_1 = \left(A_0.x + D_A \cdot \cos\left(angle(Bisector)\right), A_0.y + D_A \cdot \sin\left(angle(Bisector)\right)\right)$$

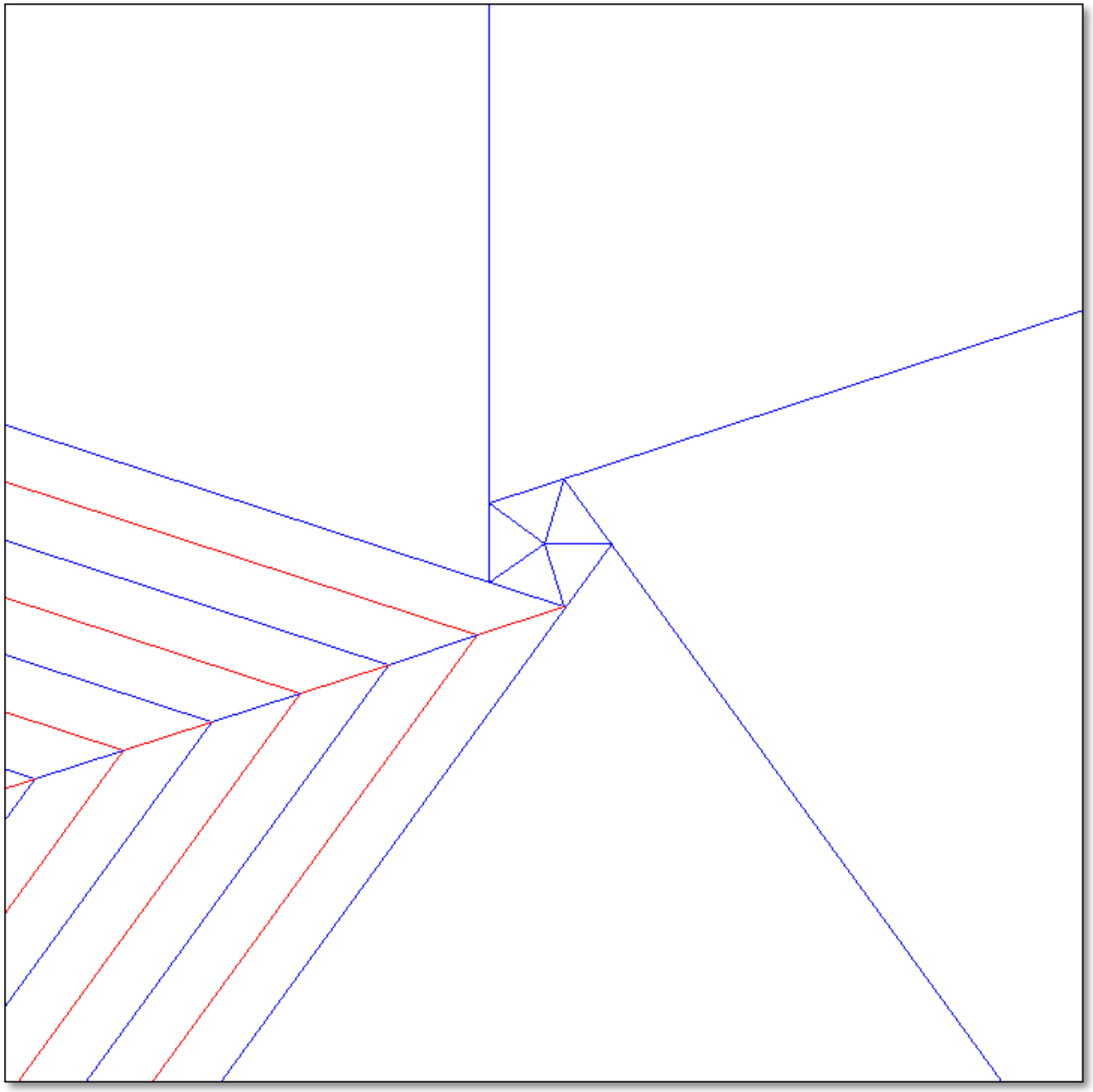
For further iterating points such as A_2, A_3, and beyond, we can simply use multiples of D_A.

$$A_k.x = A_0.x + k \times D_A \cdot \cos\left(angle(Bisector)\right)$$
$$A_k.y = A_0.y + k \times D_A \cdot \sin\left(angle(Bisector)\right)$$



(Figure 14: The iterating alternation points and the margin of the render area.)

In the case of figure 14, the repetition stops after plotting the rays associated with A_3. Plotting any further points is redundant, as the point has already crossed the margin.



(Figure 15: The completed alternation of a single pair of surrounding base rays.)

Repeating this process for the remaining pairs completes the pattern. Because of floating point numbers, computer-generated patterns are always prone to small errors or offsets. However, in most cases, this error can be ignored, as these do not hinder the folding process.