CS334 Final Project - Squares And More (SAM)

Gerardo Morales, Will Confoy

0.1 Introduction

Art is hard. How hard varies from person to person but for those who are not artistically inclined, it can be really difficult to get over the initial hurdle of making what's in your head appear on canvas. Squares And More (SAM) helps to remove this barrier, by providing a rigorous language for creating regular polygons of nearly any configuration. Additionally, SAM supports a syntax that is easy to understand for anyone fluent in English, further increasing its usefulness. Compared to the more typical artistic approach of manually drawing, it's also much easier to make adjustments as you go. This is due to the fact that SAM is a programming language, and so you can rely on its output to be consistent. Being able to tweak values to make adjustments to the art is much easier than trying to erase or overwrite something, risking damaging the rest of the piece while doing so.

SAM's usage especially lessens the burden on anyone who is disabled or has difficulty with fine motor skills. So long as you can put words on a screen, you can use SAM to produce the exact image you want, so long as that image is made up of regular polygons. While this may seem like a harsh restriction, it's not nearly as limiting as it may sound. All this coupled with the fairly natural language approach of SAM means that SAM works well with the many diverse methods by which people have found to interface with computers.

0.2 Design Principles

While designing SAM, we kept in mind two main principles: ease of use and abstraction. We wanted to ensure that SAM was easily understandable to English speakers so that it would be easy for people who hadn't seen it before to hop in and begin writing programs, and being able to put the image in their brain on canvas. Of course, there are some specifics that require user input, as it is a rigorous language. However, we did our best to abstract as much of the definitions away from the sight of the user, using the minimum definitions of shapes possible to deterministically define shapes. For example, you really only need a point, a distance, a number of sides, and an orientation to fully define any regular n-gon. Since this is a program for art, we also need to know the color of your n-gon. If you want to write multiple similar n-gons though, you'll also need to know how these things change. In particular, you need to know how the radius is changing, how the center is changing, how the orientation is changing, and how the colors change with each n-gon. With this information, SAM is able to handle all of the work that goes into actually calculating where each point needs to be on the canvas behind the scenes, so the user doesn't need to worry about it at all.

0.3 Examples

SAM is best run in the following way: "dotnet run example.sam $>$ example.svg".
Here are a few sample programs:

```
[<canvas(600,600)>]


4 4-gon with radius 120 and center(300,390), step[85,85,85],
rotation[45.0], color[#b4bcb1,#f7d85b,#fdd30b,#f0c55f],
centerDelta [(0,-30),(0,-30),(0,-30)]
```

This has been extended to multiple lines to fit on the page, but in actuality the way it would normally look is it would have the canvas part on one line, then a newline and then the n-gon line on one line. You could, however, put it all on one line. The output is below.

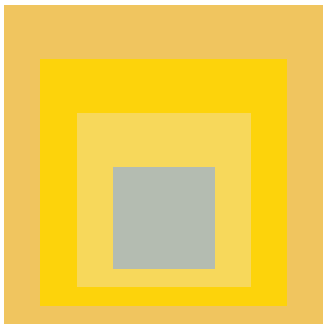

Figure 1: Homage to the Square

And another program:

```
[<canvas(400,400)>]

4 4-gon with radius 40 and center (200,200), step[5,4,4],
rotation[0.0,45.0], color[#f9c406,#FF0000,#00FF00,#0000FF],
centerDelta [(75,-50),(5,-5)]
```

As before, this has been changed to fit on the page. The output is below:

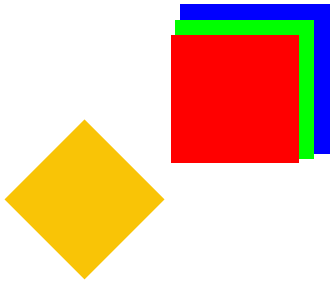

Figure 2: Rotation List Test Figure

and finally, a more complex program:

```
[<canvas (700,700)>]


8 60-gon with radius 60 and center (350,350), step [10], rotation [0.0],
color [#f9c406,#FF0000,#995668,#f9c406,#f9c406,#FF0000,#995668,#f9c406],
centerDelta[(-25,-25)]
8 60-gon with radius 60 and center (350,350), step [10], rotation [0.0],
color [#f9c406,#FF0000,#995668,#f9c406,#f9c406,#FF0000,#995668,#f9c406],
centerDelta[(25,25)]
8 60-gon with radius 60 and center (350,350), step [10], rotation [0.0],
color [#f9c406,#FF0000,#995668,#f9c406,#f9c406,#FF0000,#995668,#f9c406],
centerDelta[(25,-25)]
8 60-gon with radius 60 and center (350,350), step [10], rotation [0.0],
color [#f9c406,#FF0000,#995668,#f9c406,#f9c406,#FF0000,#995668,#f9c406],
centerDelta[(-25,25)]
```

```
8 60-gon with radius 60 and center (350,350), step [10], rotation [0.0],
color [#f9c406,#FF0000,#995668,#f9c406,#f9c406,#FF0000,#995668,#f9c406],
centerDelta[(0,25)]
8 60-gon with radius 60 and center (350,350), step [10], rotation [0.0],
color [#f9c406,#FF0000,#995668,#f9c406,#f9c406,#FF0000,#995668,#f9c406],
centerDelta[(0,-25)]
8 60-gon with radius 60 and center (350,350), step [10], rotation [0.0],
color [#f9c406,#FF0000,#995668,#f9c406,#f9c406,#FF0000,#995668,#f9c406],
centerDelta[(25,0)]
8 60-gon with radius 60 and center (350,350), step [10], rotation [0.0],
color [#f9c406,#FF0000,#995668,#f9c406,#f9c406,#FF0000,#995668,#f9c406],
centerDelta[(-25,0)]
1 60-gon with radius 60 and center (350,350), step [], rotation [0.0],
color [#0055BB], centerDelta [(0,0)]
```

Once again, this has been edited to fit on the page. The output is below:

Figure 3: Many 60-gons

0.4 Language Concepts

The core ideas can essentially be boiled down into what a regular n-gon is, and different attributes that they may have, such as orientation, size, position, and color. For some of the specific syntax, it might be useful to understand what an array is, as we use them to specify how these values change for different polygons. They should also at least be able to use hexadecimal to specify colors, even if they don't understand how it works exactly. There are plenty of online hexadecimal colors pickers to assist with this. Since we are only exposing creation of 1 type of shape, users don't need to worry about many different primitives or combined forms. Underneath the hood, there will be plenty of points, lines, and whatnot, but that's all abstracted away. For example, a polygon is a composed of a point, a radius, a color, and a rotation (in degrees). We ask the user for all of the data, but it's in English using ordinary numbers, and we handle putting it all together.

Besides that, it would be useful for users to understand some of the ways that SAM is strict and some of the ways that it is lenient. SAM is strict in the syntax that it expects- you really do need to include something for each of the tags such as step and rotation. However, you don't need to provide very much for each tag. This is especially important if you're writing a program that starts with something like "60 5-gon with"... You don't want to have to write down 60 colors or 59 centerDeltas in an array! So while you need to provide at least one value in each of these tags, if there are less values in the array than there are n-gons being drawn, SAM will take the last value in the array and use it for the rest of them.

0.5 Syntax

$$
\begin{aligned}
< expr > &:=< newline >^* [< canvas(< n >, < n >) >] < newline^* >< ngon >^+ \\
< ngon > &:=< n > \sqcup < n > -\text{gon\_with\_radius } < n > \text{ and\_center}(< n >, < n >), \\
&\qquad \text{step } < step >, \text{ rotation } < rotation >, \text{ color } < color >, \text{ centerDelta } < centerDelta >< newline >^+ \\
< n > &:=< digit >^+ \\
< negn > &:= - < n > \\
< int > &:=< n > | < negn > \\
< digit > &:= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
< f > &:=< int > . < n > \\
< step > &:= [< steps >^*] \\
< steps > &:=< n >, < n > \\
&\quad | < n > \\
< rotation > &:= [< rotations >^+] \\
< rotations > &:=< f >, < f > \\
&\quad | < f > \\
< color > &:= [< hex >^+] \\
< hex > &:= \# < hdigit >^+, \# < hdigit >^+ \\
&\quad | \# < hdigit >^+ \\
< hdigit > &:= 0 \mid 1 \mid 2 \mid \ldots | F \\
< centerDelta > &:= [< delta >^+] \\
< delta > &:= (< n >, < n >), (< n >, < n >) \\
&\quad | (< n >, < n >) \\
< newline > &:= \text{`}\backslash n\text{'}
\end{aligned}
$$

0.6 Semantics

$< n >$	A positive integer ranging from 0 to the greatest integer F# can represent
$< negn >$	A negative integer ranging from the smallest integer F# can represent to 0
$< f >$	A floating point value that can express the same values as a float in F#
$< canvas >$	Canvas is a tuple of positive integers that specifies the dimensions of the SVG canvas
$< n > n$-gon	The second n specifies the number of sides of each n-gon and the first n specifies how many of them to draw
$< radius >$	The size of the radius of the first n-gon drawn, measured from one of the corners to the center of the n-gon
$< center >$	The x and y coordinates of the center point of the first n-gon drawn
$< step >$	The change in the size of the radius of each successive n-gon
$< rotation >$	The amount each n-gon is rotated about its center point in degrees
$< color >$	The hex values corresponding to the color of each successive n-gon
$< centerDelta >$	A list of int tuples that indicates how much to change the x and y coordinates of each successive n-gon
$< ngon >$	An object representing a series of n-gons, each with the same number of sides, which contains the data from all of the above fields

0.7 Remaining Work

The majority of the remaining work lies in making the language relax some of its parsing with regard to newlines or with ints and floats. For example, for rotation, if you want to rotate 45 degrees, that needs to be 45.0 degrees currently. It would be nice to just be able to write 45 and have the language understand that. Beyond that, there are several things that would be nice to have in the future, that we might work on after the class. Among them are implementing variables and functions, as well as messing around with optional fields with default values so that there's less typing for each set of ngons. Finally, there is a feature that we had planned that we weren't able to make work. That's the distribution and granularity fields that were present (but unusable) in previous versions of the project. It ended up being a lot more difficult to implement than we expected and we never found an answer we were satisfied with. The random nature also clashes a bit with the deterministic vision of SAM, and we're not entirely sure if it's a good idea to have at all.