INFO0004-2: Drawing geometric figures

Pr Laurent MATHY, Cyril SOLDANI

May 15, 2019

1 Introduction

In this project, you will convert geometric figure descriptions in a textual format to raster images. This project is an extension to the previous project, and uses the same simple geometric language.

2 The painting language

2.1 Overall structure

A *paint* file is a text (ASCII) file. White-space is generally ignored, but separates tokens so that `123␣456` represents the two numbers 123 and 456, and not 123456.

A `#` character introduces a comment. All characters from the `#` to the end of the line where it appears should be ignored.

A paint file begins with a `size WIDTH HEIGHT` command. The file can then contain any number of shape definitions, color definitions and paint commands, which are described below. In EBNF[1] notation, this gives:

```
paint_file = size_command, {shape_def | color_def | paint_command};
size_command = "size", number, number;
```

The two numbers given to the `size` command, *i.e.* `WIDTH` and `HEIGHT` are the width and height of the image in pixels. Consequently, they should be positive integers (they may still be given as floating-point literals, as long as they have integer value).

2.2 Numbers

Numbers in the paint language are floating-point numbers (except for the `size` command, as explained above). They can be given as floating-point or integer literals, or by taking the abscissa or ordinate of a point (see section 2.5).

```
number = ["+" | "-"], ["."], digit, {digit}
       | ["+" | "-"], digit, {digit}, ".", {digit}
       | point, ".", "x" | "y";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

All of the following are valid numbers: `123` ($= 123.0$), `.5` ($= 0.5$), `+42.` ($= 42.0$), `-3.14`.

2.3 Names

The *paint* language allows to name shapes and colors. A `name` begins with a letter, and can contain any number of letters, digits and underscores.

[1] Extended Backus-Naur form: https://en.wikipedia.org/wiki/Ebnf.

```
letter = ? any char c such that isalpha(c) is true ?;
name = letter, {letter | digit | "_"};
```

All definitions and commands can only refer to names that were already defined above in the file. It is an error to use a name that is not defined, or is defined after the location where it is used. You should report such an error where the undefined name is used.

It is also a error to try to give to a new shape the name of an existing shape, or to a color the name of an existing color. You should report such an error at the location where the name is redefined. It is however fine to use the same name for a shape and a color, those lie in different *namespaces*.

2.4 Colors

A color is given by a triplet of numbers between braces. Each number represent a color component in the RGB color space, and should be between 0.0 and 1.0. You should report an error if a color component is out of range.

One can name a color with the `color` instruction, which takes a name and a color. A color name can be used wherever a color is expected.

```
color = "{" number, number, number, "}"
      | name;
color_def = "color", name, color;
```

2.5 Points

A point is defined by a pair of coordinates between braces, by referring to a point inside a shape, or by an arithmetic expression.

```
point = "{", number, number, "}"
      | named_point
      | point_expr;
named_point = name, ".", name;
```

A named point is given by `SHAPE_NAME.POINT_NAME` where `SHAPE_NAME` is an existing shape name, and `POINT_NAME` is the name of a specific point inside that shape. *E.g.* `a_circle.c` will return the center of the circle `a_circle` (assuming `a_circle` is a valid circle). Valid point names depend on the shape type, and are described bellow.

A point can be subscripted with `.x` or `.y` to obtain respectively its abscissa and its ordinate. *E.g.* `a_circle.c.x` returns the abscissa of the center of circle `a_circle` (if it is defined). `{4 3}.y` returns 3.

Point expressions allow to do arithmetic operations on points. They use a syntax reminiscent of scheme, where the operator precedes its operands, and the order of operations is explicit. Addition and subtraction accept between one and an arbitrary number of points. *E.g.* `(- {12 0} {2 0} {3 1})` returns the point `{7.0 -1.0}` (as $12 - 2 - 3 = 7$ and $0 - 0 - 1 = -1$).

The multiplication and division take exactly one point and one number. They apply the operation to the coordinates of the point. *E.g.* `(/ {10 5} 2)` returns point `{5.0 2.5}`.

```
point_expr = "(", "+" | "-", point, {point}, ")"
           | "(", "*" | "/", point, number, ")";
```

2.6 Shapes

Each shape instruction is directly followed by a (new) `name`, which will be bound to the created shape. It is then followed by arguments which are specific to the type of shape being defined.

There are 4 primitive shapes: circles, ellipses, rectangles and triangles. There are 4 derived shapes: shifts (translations) of a shape, rotations of a shape, unions of shapes and differences of shapes.

```
shape_def = "circ", name, point, number
          | "elli", name, point, number, number
          | "rect", name, point, number, number
          | "tri", name, point, point, point
          | "shift", name, point, name
          | "rot", name, number, point, name
          | "union", name, "{", name, {name}, "}"
          | "diff", name, name, name;
```

Circle

Arguments to the `circ` command are its center point, and its radius (which should be positive).

Among supported named points on a circle should be at least `c` for *center*, and cardinal points `n` for *north*, `nw` for *north-west*, *etc.*

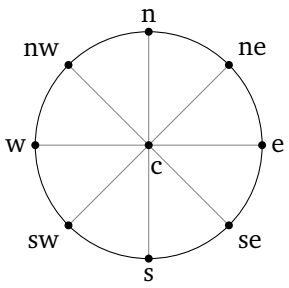

Figure 1: Points in a circle.

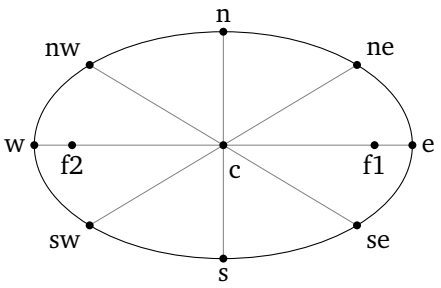

Figure 2: Points in an ellipse.

Ellipses

Arguments to the `elli` command are its center point $\{c_x, c_y\}$, its semi-major radius a, and its semi-minor radius b. Both radii should be positive.

Named points are similar to the circle ones, with the addition of `f1` and `f2` for the two focii of the ellipse (`f1` being to the right). Note that intermediate cardinal points `nw`, `ne`, `se` and `sw` result from squishing the circle of figure 1. If the ellipse is described by parametric equation

$$(x, y) = (c_x + a\cos t, c_y + b\sin t), \quad 0 \leq t < 2\pi,$$

then points `ne`, `nw`, `sw` and `se` would correspond to values of `t` of $\pi/4$, $3\pi/4$, $5\pi/4$ and $7\pi/4$ (respectively).

Rectangles

Arguments to the `rect` command are its center point, its width and its height (both of which should be positive).

Named points on a rectangle are illustrated in figure 3.

Triangle

Arguments to the `tri` command are its three vertices (in no particular order).

Named points on a triangle are its vertices `v0`, `v1` and `v2`, the middles of its sides `s01`, `s02`, `s12`, and its barycentre (or centroid, *i.e.* its center of mass) `c`.

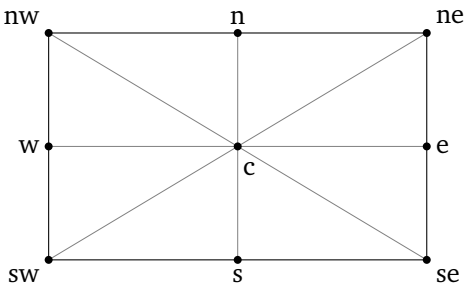

Figure 3: Points in a rectangle.

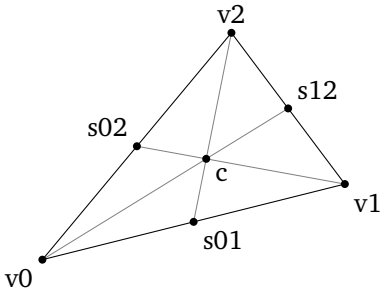

Figure 4: Points in a triangle.

Shift

Arguments to the `shift` command are a point giving the translation vector (t_x, t_y) and the name of the shape to be translated.

The resulting shape is exactly the same as the original one, but translated by (t_x, t_y), *i.e.* a point at (x, y) in the original shape will lie at $(x + t_x, y + t_y)$ in the translated shape.

Named points of a `shift` shape are the same ones as those of the original shape, translated according to the translation vector.

Rotation

Arguments to the `rot` command are the (trigonometric) angle of rotation in degrees, the point to rotate around, and the name of the shape to be rotated.

Named points of a `rot` shape are the same as the ones of the original shape, transformed according to the given rotation.

Union

Arguments to the `union` command are the names of the shapes to be grouped into a new single shape. The resulting shape is the union of all given shapes, *i.e.* a point will be in the union if it is in at least one of the given shapes.

Named points of a `union` shape are those of its first shape.

Difference

Arguments to the `diff` command are the name of the shape to be subtracted from, and the name of the shape to be subtracted. A point is in the difference if it is in the first shape, but not in the second one.

Named points in a difference are those of the first shape (even if they are not part of the resulting shape after the subtraction).

2.7 Painting commands

Arguments to the `fill` command are the name of a shape, and a color.

```
paint_command = "fill", name, color;
```

The given shape will be drawn to the image with the given color. More specifically, for all pixels in the image, if the center of that pixels is inside the shape, that pixel should be filled with the given color. There is no support for anti-aliasing, either a pixel is fully filled, or it is not at all.

The origin of the image lies at $(0, 0)$, with x direction going to the right, and y direction going to the top. A pixel is considered to have size one by one.

If a pixel center lies exactly[2] on the border of a shape, it will be considered inside it and filled. However, for differences, remember that a point is considered inside the difference if it is inside the first (subtracted from) shape, and not in the second (subtracted) shape. It results that a pixel center that is inside the first shape and on the border of the second shape is **not** considered to be inside the difference, even though it will lie on the border of the resulting shape. The corresponding pixel should thus **not** be filled.

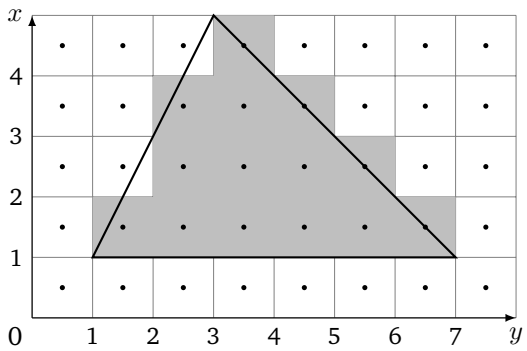

Figure 5: Triangle `tri {1,1} {7,1} {3,5}`, and corresponding fill.

Note that the shapes can be or have parts outside the image canvas. Parts outside the image canvas should simply be ignored.

Finally, the image is initially black.

3 Interface

Your program should be called `painter`, and will be called as `./painter FILE.paint`, where `FILE.paint` is the path to a file containing painting instructions.

It should output resulting image in PPM[3] format in file `FILE.ppm`. More specifically, you will use the binary `P6` variant, with 8-bit color values. All white-space should be just one space character, except for the one separating maximum color value from the raster, which should be a newline character. You can ignore gamma considerations and assume the given colors are already in the PPM RGB color space, *e.g.* color `{1 .5 .2}` will map to PPM color components of 255, 128 and 51. In general, you can convert a color component from the double-precision representation to the byte representation with `uint8_t(std::round(double_val * 255))`.

Example code that saves an image to PPM format is provided on the course web page. You can use it *as-is* or modify it as you see fit.

If there is an error in the input file, your program should abort with an error message on `stderr`.

4 Remarks

4.1 Respect of the interface

You **must scrupulously follow the given interface**. Your archive should respect the format given in section 5, your code should build `painter` with a simple `make` in your folder, and its output

[2] Up to 64-bit floating-point precision.
[3] Portable Pixel Map: http://netpbm.sourceforge.net/doc/ppm.html

should be exactly as described in section 3.

4.2 Efficiency

Your program should be able to generate images of several thousand pixels, containing hundreds of shapes, without needlessly consuming too much memory or too much CPU time.

4.3 Readability

Your code must be readable:

- Make the organisation of your code as obvious as possible. Remember you can create as many auxiliary classes and functions as you see fit.
- Use descriptive names.
- Complement your self-documenting code with comments, where appropriate.
- Choose a coding convention, and stick to it. *Consistency* is key.

4.4 Robustness

Your code must be robust. The `const` keyword must be used correctly, sensitive variables must be correctly protected, memory must be managed appropriately, the program must run to completion **without crash** (even if the user provides invalid input).

4.5 Warnings

Your code must compile **without error or warning** with `g++ -std=c++14 -Wall` on `ms8??` machines. However, we advise you to check your code also with `g++ -std=c++14 -Wall -Wextra` or, even better, `clang++ -Weverything -Wno-c++98-compat`.

4.6 Object-oriented approach

While C++ is a multi-paradigm language, and this problem can be solved using different approaches (*e.g.* a functional one), remember this is an *object-oriented* course. Try to apply the object-oriented concepts where it makes sense.

4.7 Evaluation

Your code will be evaluated based on all the above criteria. Failure to comply with any of the points mentioned in **bold face** can (and most often will) result in a mark of zero!

5 Submission

Projects must be submitted through the submission platform before **May the 17$^{\text{th}}$, 23:59 CEST**. Late submissions will be accepted, but will receive a penalty of $2^n - 1$ points (/20), where n is the number of days after the deadline (each day start counting as a full day).

You will submit a `s<ID>.tar.xz` archive of a `s<ID>` folder containing your C++ source code and a `Makefile` to compile it (with a simple `make`), where `s<ID>` is your ULiège student ID.

The submission platform will do basic checks on your submission, and you can submit multiple times, so check your submission early!