INFO0004-1 : Project 3 (60%)
Mathematical artist

Submission before December 22

In this project, you will implement a C++ "mathematical artist" (a.k.a. "martist").

A mathematical artist creates images from random mathematical expressions. Our martists will construct complex expressions from 6 basic expressions: x, y, $\sin(\pi * \text{expr})$, $\cos(\pi * \text{expr})$, the product of two expressions, and the mean (average) of two expressions. Martists thus represent complex expressions as nested basic expressions.

Note that for all values of x and y in $[-1, 1]$, all basic expressions have values in $[-1, 1]$. Therefore, any complex expressions built from these basic expressions will also have such values in this interval.Using simple scaling, an "intensity level" in $[0, 255]$ can thus easily be obtained for any point in the $-1 \leq x, y \leq 1$ square.

Likewise, by scaling operations on x and y, this 2-by-2 square can be stretched into an w-by-h image, where "w" is the width and "h" the height of the image (in pixels).

A colour image can be obtained by using 3 random expressions, one for each colour component (red, green and blue) of the image.

1 Expressions

Of the six basic expressions, two are simple expressions (x et y), the four others are composed expressions (that is, expressions made up of two or more expressions). All expressions, complex or basic, can thus be seen as trees whose internal nodes are composed basic expressions and whose leaves are simple basic expressions.

To control the complexity of the generated images, the *depth* of an expression is defined as the maximum permitted length of each branch of the expression tree. For example, x has a depth of 1, $\sin(\pi * y)$ has a depth of 2 and $\cos(\pi * (x * \sin(\pi * y)))$ has a depth of 4.

You martist must be able to generate random expressions of a given depth. Note that the notion of depth *only* define the allowed maximum complexity, and that any expression of a given depth is also an expression of a bigger depth. For instance, y is an expression of depth 1, but is also an expression of depth 4 (and of any depth ≥ 1). The artistic "style" of your martist will depend on your interpretation of "generate a random expression of a given depth".

To "print" expressions on an I/O stream, you will use the following format, where *exp1* and *exp2* represent components in composed expressions:

Expression	**Print**
x	`x`
y	`y`
$\sin(\pi * exp1)$	`sin(pi *` *exp1*`)`
$\cos(\pi * exp1)$	`cos(pi *` *exp1*`)`
exp1 $*$ *exp2*	`(`*exp1* `*` *exp2*`)`
$(exp1 + exp2)/2$	`avg(`*exp1*`,` *exp2*`)`

To print the spec of a colour image on an I/O stream, the following format will be used:

```
red   = redexp
green = greenexp
blue  = blueexp
```

2 Images

An image is represented by an array of pixels, that will be stored as a row-first one-dimensional array, with each pixel containing 3 colour components (red, green and blue, in that order; each component being an `unsigned char`; there is no transparency component).

When you compute the mapping between image points and points in the 2-by-2 square, it is important to remember that these two "spaces" have their origins at different places: in $-1 \leq x, y \leq 1$, the origin $(0, 0)$ is obviously at the center of the square; for the image, the origin $(0, 0)$ is the top-left corner. *The mapping between the square and the image is positional*: the top-left corner in the square must map onto the top-left corner in the image, the centre of the square must map to the centre of the image, etc.

3 The `Martist` class

You will write the C++ `Martist` class that exposes the following interface:

1. `Martist(unsigned char* buffer, size_t width, size_t height, int rdepth, int gdepth, int bdepth)`; where `buffer` is a "passed-in" array to be the buffer containing the image pixels; `width` and `height` are the respective dimensions of the image (in pixels); `rdepth`, `gdepth`, `bdepth` are the depths for expressions representing the red, green and blue colour components, respectively. Note that if the depth of an expression is zero, the corresponding expression has a constant *pixel* value of 0.

2. `void redDepth(int depth)`, `int redDepth() const`, `void greenDepth(int depth)`, `int greenDepth() const`, `void blueDepth(int depth)`, and `int blueDepth() const`; to set and get the depth of the red, green and blue expressions, respectively.

3. `void seed(int seed)`; changes the Martist's mood, or in other words, seeds the randomness.

4. `void changeBuffer(unsigned char* buffer, size_t width, size_t height)`; changes the image buffer.

5. `void paint()`; generates a new (random) image.

6. the input and output stream operators, to read and write image specs (see above) on streams. Note that reading an image spec from a stream should update the depths of expressions, as well as update the image buffer.

4 Remarks

You are free to create whatever private interface for your `Martist` class, and whatever other classes or functions that you see fit. However, your Martist will only ever be accessed through the public interface described above.

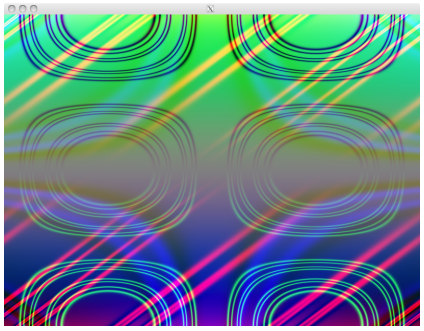
Also, all functions from the public interface should throw an `std::domain_error` exception if presented with arguments that do not make sense. Note that it is always the programmer's responsibility to make sure the image buffer is big enough.

If you take a look at the brief for project 2, you'll soon realize the expressions in both projects are identical: be wise, and re-use as much code as you can!

You may want to write a simple test application to display your Martist's images on screen. The CImg Library (`http://cimg.eu`) can be a great help here.

Here is an example of an image spec and corresponding image:

```
red   = (cos(pi * (x * y)) * avg(y, sin(pi * cos(pi * cos(pi * sin(pi * (x * avg(y, y))))))))
green = (y * cos(pi * cos(pi * cos(pi * cos(pi * cos(pi * (sin(pi * x) * cos(pi * y))))))))
blue  = sin(pi * ((y * y) * cos(pi * cos(pi * cos(pi * sin(pi * sin(pi * avg(x, y))))))))
```



Your code must be readable. Use indentation, one of the common naming conventions for variable names, and comments.

Your code must be robust. The const keyword must be used correctly, sensitive variables must be correctly protected, memory must be managed appropriately, your code must not cause a crash.

5 Evaluation

This project counts towards 60% of your final mark. In this project, you will be evaluated on: readability of your code, organization of your code (including support for incremental compilation), usage of the STL, correctness, object-oriented design, judicious and correct use of C++ language features, code reusability, and efficiency of the solution.

6 Submission

Projects must be submitted before Friday December 22, 23:59. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of $2^N - 1$ marks (where N is the number of started days after the deadline).

The public interface described above *must be* declared in a file named `martist.hpp`. You will submit your code (all appropriate files) as an archive named `martist.tar.gz` (no sub-directory!) which contains a `Makefile` (supporting incremental compilation) that generates a `martist.o` object file via the command `make martist`. This file will be linked with our test application to test your code. Be careful *NOT TO* submit your own test application!

Your code, which must scrupulously respect the public interface described above, must not generate errors or warnings on the submission platform. Failure to compile will result in an awarded mark of 0. Warnings systematically result in lost marks.

You can use C++11 features if you wish to.

Have fun...