Assignment 4 — Mandelbrot

The goal of this assignment is to implement the Mandelbrot in SaC. The work is broken down into 3 tiers, it is suggested you complete tier 1, attempt tier 2, and do tier 3 if you want an additional challenge :D.

Setup

In order to do the assignment, you need to have access to the SaC compiler and standard library, *and* the SDL library.

The SaC compiler and standard library can be downloaded from http://www.sac-home.org. The SDL library is available at https://github.com/SacBase/SDL and needs to be compiled separately, follow the instructions provided in the repository to do this.

NOTE: MacOS *does not* support SDL. Users of MacOS can use Hans' SaC installation on the RU Science faculty servers, see below for further details.

For users using Linux via Windows WSL, you might have to configure XServer to display the SDL window, see https://wiki.ubuntu.com/WSL#Running_Graphical_Applications for details on how to do this.

Assignment Guidelines

Below we provide some advice on how to complete the tiers. For tiers 1 and 2 we provide *demo* SaC programs which you can use to make sure that your implementation is producing the correct output.

Compiling the Demo Programs

Two demo programs are provided: `Fractal1-demo.sac` and `Fractal2-demo.sac`, for tiers 1 and 2, respectively. A `Makefile` is provided to allow you to easily compile these demo programs using the `make` program. Calling `make` or `make f1_demo` will build the first demo program, `make f2_demo` will build the second demo program.

Compiling the Mandelbrot program

The `Makefile` can also be used to build the Mandelbrot program as well using `make mandelbrot`.

Note: in the `mandelbrot.sac` file you will have to update the TIER macro value to 1, 2, 3 to indicate which tier you wish to compile against.

Tier 1

1. To get started, you may want to use the files `mandelbrot.sac` and `Fractal_tier1.sac` which, in essence, contain the IO-code for visualising the Mandelbrot pictures. A first running version can be obtained by implementing the missing function bodies in `Fractal_tier1.sac`:
   - `escapeTime` which implements the iteration on arrays of complex numbers, and
   - `genComplexArray` which computes a two-dimensional array of complex numbers that represent a discretisation of C^2.
2. Waiting for the final picture can be rather unpleasant if it is not clear whether the chosen fraction of C^2 yields an interesting picture and the iteration limit is high. Therefore, as a first extension, try to modify the main function in `mandelbrot.sac` so that it computes the Mandelbrot picture with increasing resolution without changing the overall size of the picture. Compute resolutions `[5,5], [10,10], ..., [320,320]` and display them consecutively in a `[320,320]` display by replicating the found values accordingly. *Hint:* define a function `stretchRgb` which takes an array of type `color[.,.]` and an integer stretching factor stretch and replicates each element of the array into a square of shape `[stretch, stretch]`.
3. The function `intArrayToMonochrome` maps all escape values into a colour by means of a *clut* (colour lookup table). Can you express this operation without a with-loop? *Hint:* you may find inspiration in one of the earlier tasks!
4. Try using the compiler option `-t mt_pth` to experiment with multi-core machines!

Tier 2

1. Now, we improve the way the colours are chosen in order to obtain smoother transitions. We will use a common approach referred to as normalized iteration counts. A normalized iteration count for a point in the complex plane z is computed by using the iteration count t and the value at that position z_t during the final iteration. Using these two values, the normalized iteration count is defined as `t_n:= (t + 1) - log2(log2(|z_t|))` for those values that escape and as 0 otherwise. The module `Fractal_tier2.sac` contains stubs for the missing functionality:

- `normalizedIterationCount` which implements the normalisation of iteration counts by taking the final computed value into account. *Hint:* the function `escapeTime` only computes the number of iterations before the value at a given position escapes. To normalize these, the final value at that position is required, as well. For this, we have provided a function `escapeValue`.
- `doubleArrayToRGB` maps the normalized iteration counts, which are double values, to an RGB colour-triple. To derive an RGB value, first scale all values such that they are in the interval `0 <= x < 360`. This value can then be used as the hue in the HSB model. *Hint:* the module `Color8` defines a function `Hsb2Rgb` that converts a HSB colour description into its corresponding RGB representation.

Tier 3

1. This Tier gives you a lot of freedom. You can come up with any non-trivial enhancement you like. This could be enhanced interactivity such as being able to generate .png–files (see https://github.com/sacbase/PNG) once you like a chosen range, the ability to keep multiple images visible, or the ability compute various Julia sets.

   Besides an implementation, please do provide a short description of what functionality you have added.

SaC on RU Science Faculty Servers

If you want, you can use SaC on the RU Science faculty servers: lilo.science.ru.nl, stich.science.ru.nl, etc.

To use SaC on Lilo, you need to add the `sac2c` binary to your `PATH`:

```
ssh -Y <username>@lilo.science.ru.nl
echo 'export PATH=$PATH:/home/hviess/.local/bin' >> ~/.bashrc
source ~/.bashrc
hash -r
sac2c -V
```

With that you should have a working copy of `sac2c`!

Note: We SSH using X-forwarding (the `-Y` flag), so that we can run the mandelbrot program using SDL.

Question?

If you have any questions about the assignment, or have issues with using SaC, you can contact Sven-Bodo or Hans on either Brightspace, SaC-User, or via email.