Math Runner

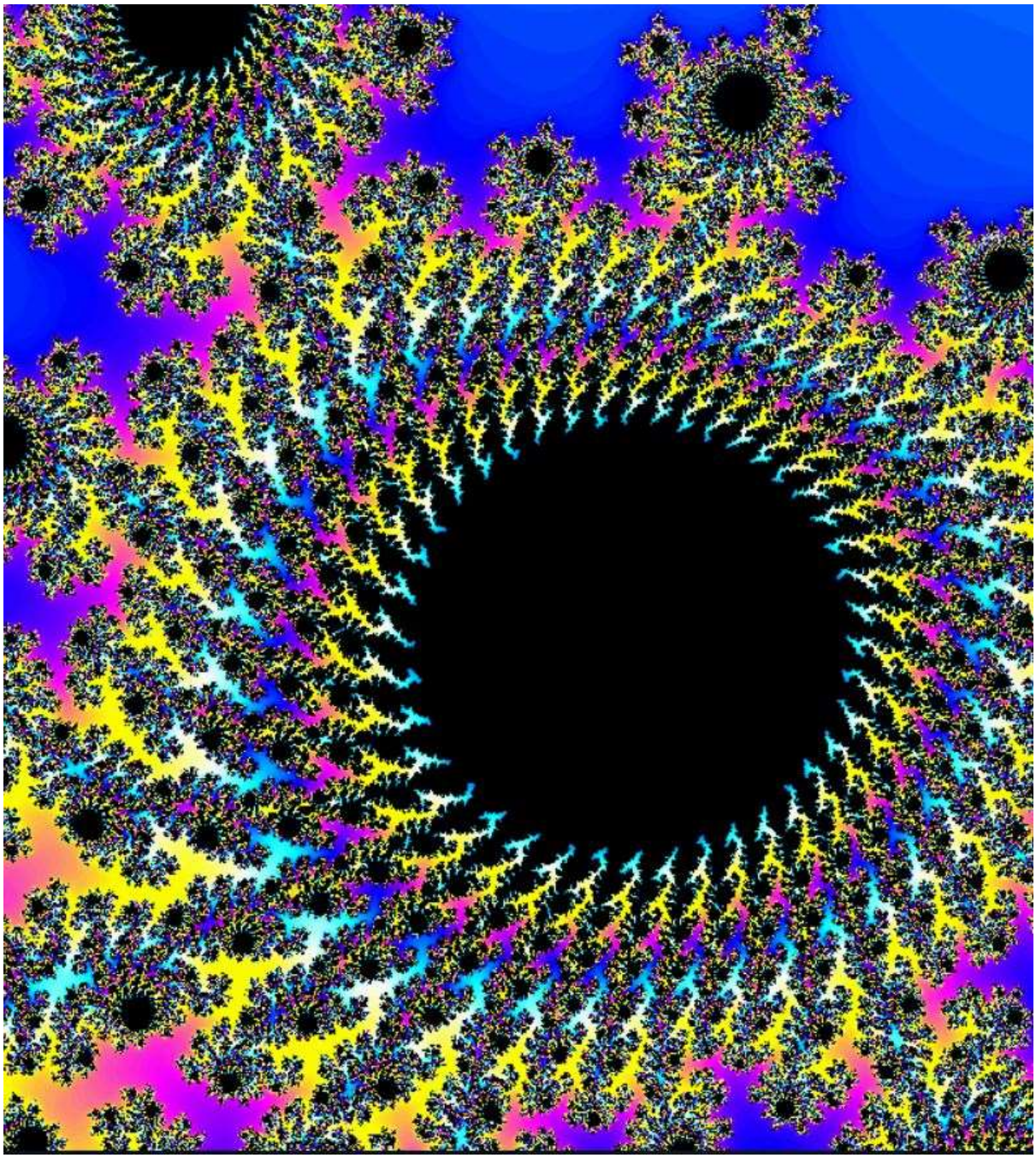
"A Psychedelic exploration of fractals through the Julia sets"

CW2 – Mathematics and Graphics for Computer Games I

Elio de Berardinis, Jean-Pascal Evette

Submission Date: 27-01-2016

No. of Pages: 13



INTRODUCTION and BACKGROUND

The Complex plane

In mathematics, the complex plane or z-plane (**Figure 1**) is a geometric representation of the complex numbers established by the real axis and the orthogonal imaginary axis. It can be thought of as a modified Cartesian plane, with the real part of a complex number represented by a displacement along the x-axis, and the imaginary part by a displacement along the y-axis [1].

In complex analysis, the complex numbers are customarily represented by the symbol z, which can be separated into its real (x) and imaginary (y) parts:

$$z = x + iy$$

For example: $z = 4 + 5i$, where x and y are real numbers, and i is the imaginary unit corresponding to $\sqrt{-1}$. In this customary notation the complex number z corresponds to the point (x, y) in the Cartesian plane. This representation is the most widely used and the most intuitive to understand the mathematical theory behind the Julia and Mandelbrot sets.

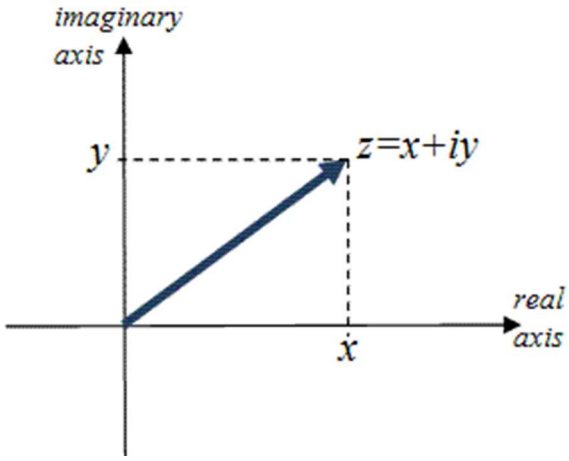


Figure 1 – The Complex Plane.

The Julia Sets

Gaston Maurice Julia (3 February 1893 – 19 March 1978) was a French mathematician who devised the formula for the Julia set while studying the iteration of polynomial functions* in the complex plane.

Generally speaking, we can apply a given polynomial function f to any complex number in the complex plane. The result of applying function f will be a different complex number or, geometrically, a different point in the complex plane. **Figure 2** shows the result of applying function f to different complex numbers where z_0 is the starting point and z_1 the resulting one.

*In mathematics, a polynomial is an expression consisting of variables (or indeterminates) and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents.

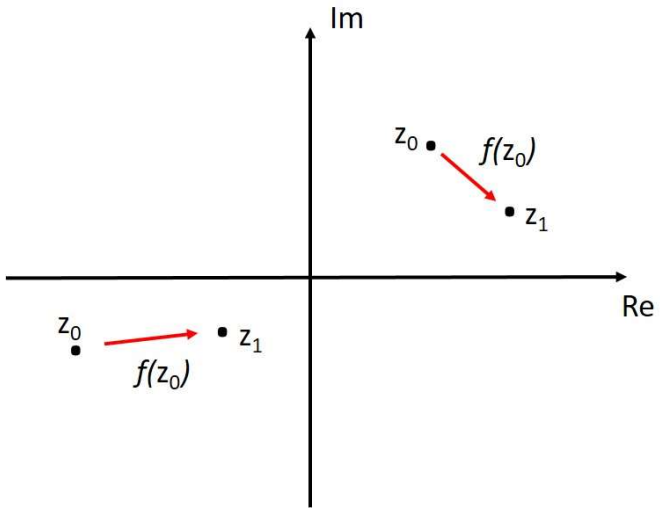


Figure 2 – Application of a generic function *f* in different points of the complex plane.

The concept of function iteration simply means the process of applying function *f* to a certain complex number z_0, taking the result z_1, applying *f* to it and repeat this process over and over again as shown in **Figure 3** up to the 4th iteration.

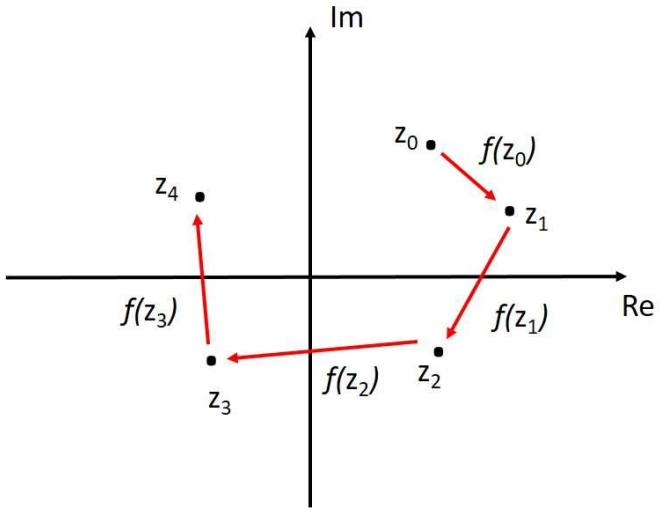


Figure 3 – Iteration of generic function *f*.

Mr. Julia was interested in finding out what happened for each complex number of the complex plane when a given function (polynomial) was iterated an infinite number of times.

He found out that 2 main different behaviours existed as we keep iterating:

1- After a certain number of iterations the resulting complex number kept increasing (in modulus) at each iteration, ultimately becoming infinitely big as the function is applied an infinite number of times (**Figure 4A**).
2- The resulting complex number stays bounded in a certain region of the complex plane. This includes the following behaviours: Converging to specific numbers called attractors (0 or non-zeros, **Figure 4B**), spiralling in an infinite loop or bouncing between 2 specific values.

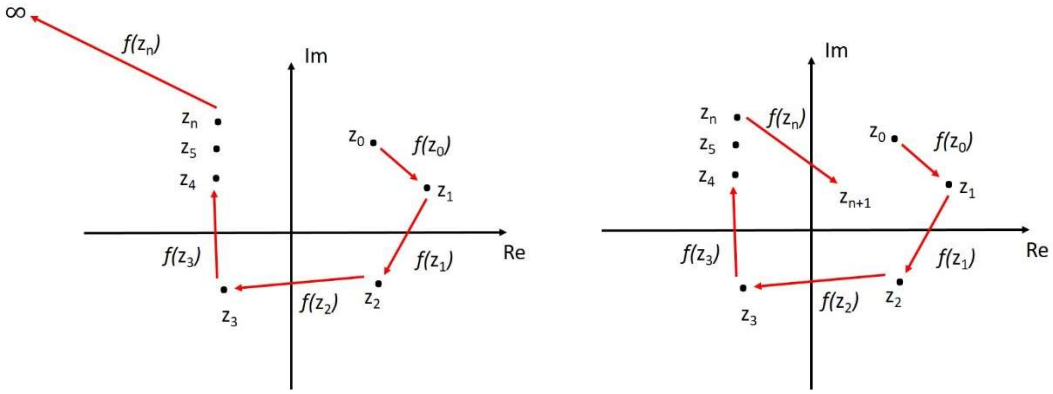


***Figure 4* – (A) Left**. After a certain number of iterations the number keeps getting indefinitely bigger. **(B) Right**. The number stays bounded and converges to a set number of attractors.

In order to graphically illustrate this phenomenon we can color-code each point of the complex plane according to its behaviour. Conventionally (and for nicer effect purposes) the points/numbers that stay bounded are coloured black and those that escape to infinity are coloured differently (i.e. blue).

Taking this concept a step ahead for nicer effects, we can color-code the "speed" at which each point escapes to infinity. For example, if the point escapes after 1 iteration we colour it blue, if it escapes after 2 light blue, 3 green, red and so on. There are many ways to implement this according to the desired effect. After we have coloured each point of the complex plane according to its behaviour under the function f we have created a Julia set.

The basic and most used function f is:

$$f(z) = z^2 + c$$

where z^2 is the classic quadratic term and $c = x + iy$ is a constant complex number. By choosing different c values we can obtain different Julia sets. Thanks to this property, the complex number c is often called "seed" of the Julia set. The basic and most intuitive (although boring) case is obtained when $c = 0$, in which f becomes the simple quadratic function $f(z) = z^2$ and the resulting Julia set is the circle of radius 1 and centre in the origin of the complex plane (**Figure 5A**). As expected all complex numbers falling within the circle stay bounded, eventually converging to 0 while those outside escape to infinity. Other examples of Julia sets obtained at different c values are shown in **Figure 5B-D**, where all the points included in the black regions represent complex numbers that stay bounded and the different colour shades represent the different speeds at which those numbers escape to infinity.

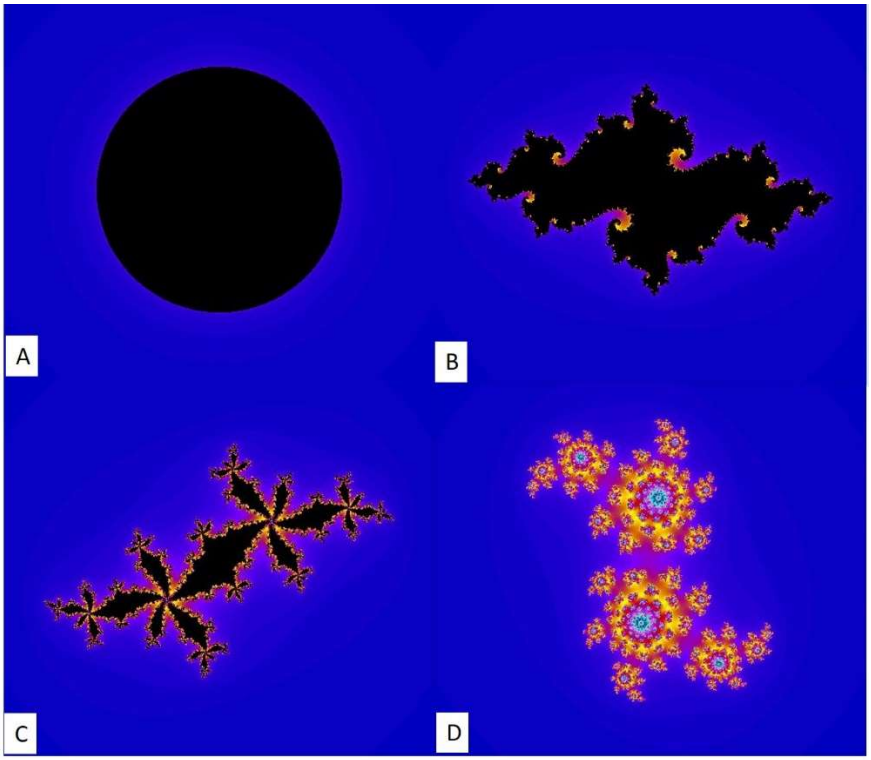


Figure 5 – Different Julia Sets obtained iterating $f(z) = z^2 + c$ in the complex plane. **(A)** c =[0,0]. **(B)** c = [-0.73,0.13]. **(C)** c = [-0.51,-0.59]. **(D)** c = [0.32,0.44]. (Images obtained with Fractal Extreme software [2]).

As we choose seeds far from the origin we get more and more interesting structures with progressively smaller, and eventually non-existing, black regions of convergence (**Figure 5D**). Besides their interesting and varied structures Julia sets have a particular and important property: they are fractals.

Fractals are mathematical (or natural) sets that exhibit a repeating pattern that displays at every scale. We can zoom into a fractal image an infinite number of times and never loose resolution [3]. If the replication is exactly the same at every scale, the fractal is called self-similar. The Julia sets have both these characteristics as shown in **Figure 6** where the set in **Figure 5C** is progressively magnified revealing a repeating pattern that never loses resolution. These properties are given by the intrinsic nature of real (and imaginary) numbers of being infinitely dense. No matter how close 2 real numbers are, there will always be another number between them.

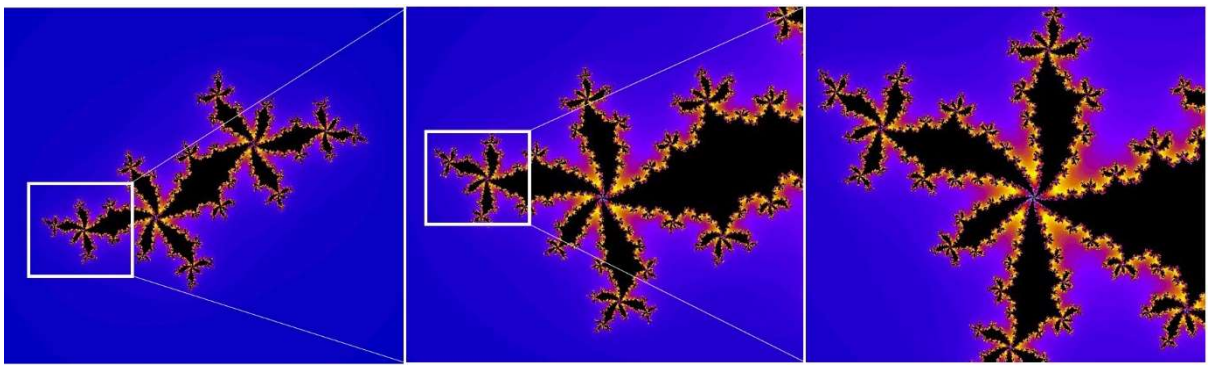


Figure 6 – Progressive magnification (left to right) of a Julia set (c = [-0.51,-0.59]). The white circle shows the magnified area. The same structure repeats itself at every scale without losing resolution. (Images obtained with Fractal Extreme software [2]).

Mr. Julia and other mathematicians were amazed by these structures and by the fact that they were generated by the intrinsic nature of numbers. They didn't invent them, they had discovered them.

As the study of these entities progressed, mathematicians noticed that there were essentially 2 main types of Julia sets: Those containing some black regions of convergence and those that had none. They labelled the first ones "Connected" (**Figure 5A-C**) and the second ones "Disconnected" (**Figure 5D**). This subdivision and its mathematical explanation gave birth to the most popular fractal known: The Mandelbrot set.

The Mandelbrot Set

The distinction between connected and disconnected Julia sets was a general way to distinguish the two different types yet not a precise one. Sometimes the black regions were small (although always comprehending more than 1 point) and spread around in certain Julia sets. It was hard to tell whether these sets would belong to the connected or disconnected category. On top of that, Mathematicians wanted a universal and mathematically rigorous way to immediately and univocally tell which category one given set (corresponding to a unique seed c value) belonged to.

The precise mathematical description of this distinction is represented in the Mandelbrot set. Its definition and its name came from the work of Adrien Douady, in tribute to the mathematician Benoit Mandelbrot [4]. The following two theorems [5] describe whether a complex number c (seed of a specific Julia set) is part of the Mandelbrot set or not (demonstration excluded):

Theorem. If $|c| > 2$, then the Julia set is a Cantor set, meaning it consists of an uncountable number of totally disconnected points. Therefore, the Mandelbrot set is contained inside $|c| \leq 2$.

Theorem. Let $f(z) = z^2 + c$

1. If the orbit of 0 remains bounded, the Julia set is connected.

2. If the orbit of 0 escapes to infinity, the Julia set is totally disconnected.

That is, a complex number c is part of the Mandelbrot set (and the corresponding Julia set is connected) if, when starting with $z = 0$ and applying the iteration repeatedly, the absolute value of $f(z_n)$ remains bounded however large n gets. For example, letting $c = 1$ gives the sequence 0, 1, 2, 5, 26,…, which tends to infinity. As this sequence is unbounded, 1 is not an element of the Mandelbrot set. On the other hand, $c = -1$ gives the sequence 0, −1, 0, −1, 0,…, which is bounded, and so −1 belongs to the Mandelbrot set.

With this tool it is enough to look at the behaviour of 0 to immediately and precisely tell whether a Julia set is connected or not. In other words, the Mandelbrot set is a "map" of all the possible Julia sets where black regions comprehend the complex numbers c that will generate connected Julia sets and other colours represent the disconnected ones. Once again, the range of colours represents the speed at which that complex number escapes to infinity.

As previously mentioned, the Mandelbrot set is a fractal and self-similar to a certain degree [6]. A rendering of the set along with some Julia sets and their seed location on it is shown in **Figure 7**.

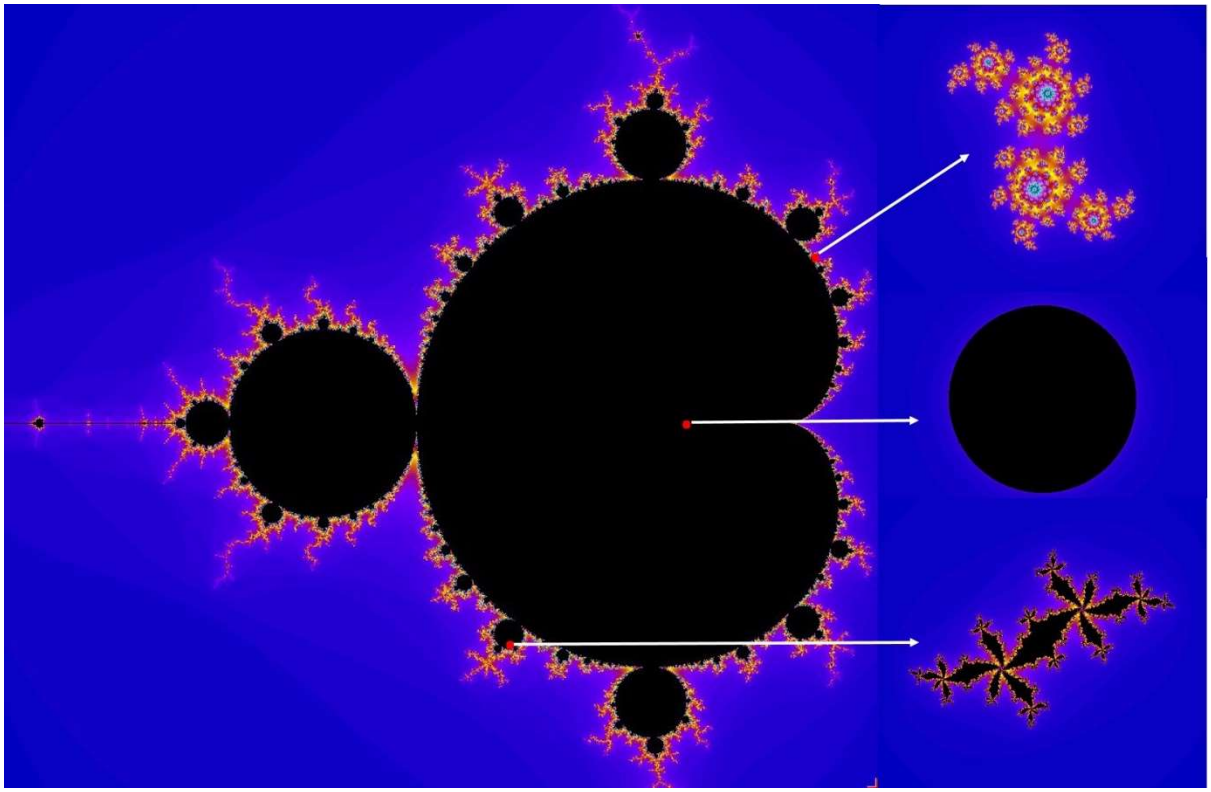


Figure 7 – The Mandelbrot set along with the seed location of the Julia sets described before.

Objectives of this work

The main objective of this coursework was to find a clever way to show to a potential user the amazing properties of fractals. In particular, to demonstrate the peculiarities of the Julia sets, their link to the Mandelbrot set and their origin in the complex plane.

Within this main scope we wanted to entertain the user and challenge him in this discovery through a light gamification of the experience.

Finally we aimed at creating a software that could easily be expanded to become a full game in future developments.

Technical details on the code implementation as well as the user experience will be discussed in the following sections.

IMPLEMENTATION and DEVELOPMENT

In this section, we will describe how the program works referring, when needed, to the line of code/Class/Function that made that feature possible in its development environment OCTET Framework [7]. As the source code is already heavily commented we will limit this aspect to the general terms and leave to the reader the possibility to integrate the information provided here with those present in the code itself.

The User experience

The player (Pink Box) is a complex number *c,* seed of the Julia set represented on the background. The "value" of the user is shown on the upper part of the screen (Parameters) separated in its real and

imaginary parts. The text display is rendered using the *text_overlay* class, which is provided by OCTET. A *text_overlay* is initialized at start-up and is then updated on every frame to reflect the changes in the real and imaginary parts. At the start the position of the user is the origin of the complex plane (0 , 0) and the corresponding Julia set is the basic circle of radius 1. Besides the numerical value of *c,* the seed location is also represented by a cross-shaped cursor in a mini-map version of the Mandelbrot set in the upper right side of the screen. A visual indication of the seed is more intuitive for the user and perfectly links the Mandelbrot set with the corresponding Julia set in the background (**Figure 8**).

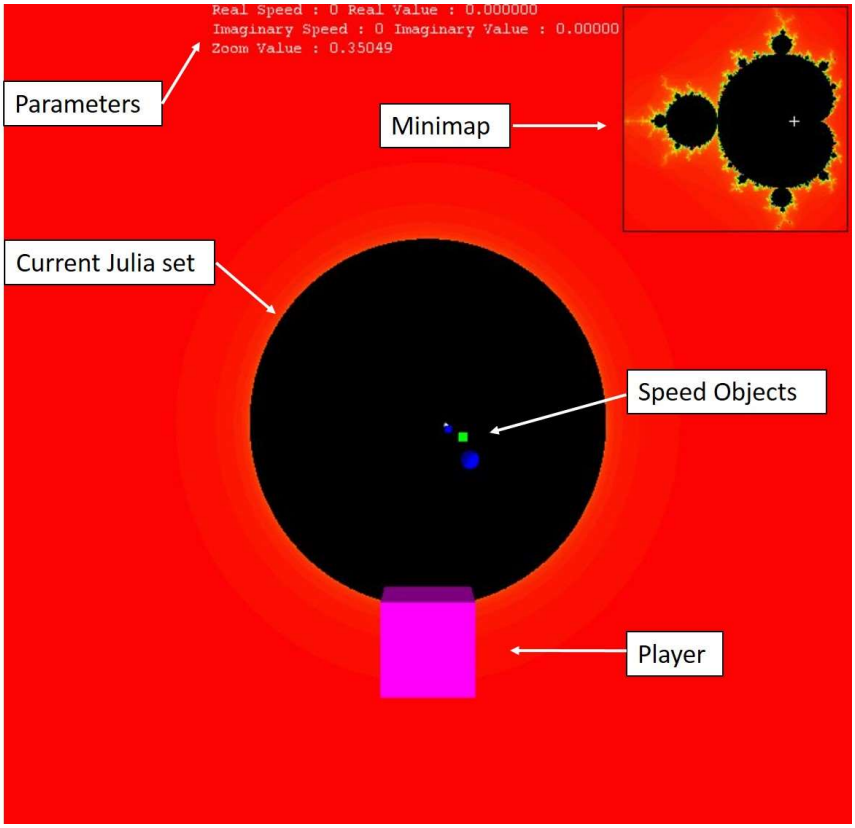


Figure 8 – Screenshot of the game scene with its components. The Julia set represented is the basic one when *c = (0, 0).*

Both the Mandelbrot set and the Julia sets are created using fragment shaders (*Mandelbrot.fs* and *Julia.fs* respectively) with code adapted from previous works [8].While OCTET does provide some tools to create shaders through header files, our implementation is instead using standard GLSL based files. Both shaders possess a set of uniforms which can then be modified at runtime to change the characteristics of the set drawn (i.e.: Zoom, location, colours, etc). Those shaders are then used as texture on some square objects which are placed in front of the player and constantly reflect the player's movement, thus creating the illusion of a background image (Julia) and a minimap (Mandelbrot).

As the program starts, the user begins traveling forward and the background image is progressively zoomed. During its travel towards the inner parts of the Julia set the player encounters different objects on its way. These objects of different shape (cubes or spheres) and colours (green, blue, white) can be picked up by the player to change its current *c* value and generate a different Julia sets in the background. The different objects and their in-game meaning is listed below:

- Green/Blue cubes: Positive/Negative acceleration of the Real part of *c*.
- Green/Blue sphere: Positive/Negative acceleration of the Imaginary part of *c*.
- White sphere: Resets both speeds to 0.

That is, picking up a green cube will make the *c* value increase its real part with a certain speed. This is represented by a continuous movement at that speed towards the right (positive *x-axis*) of the cursor on the Mandelbrot set and a continuous change of shape of the Julia set in the background.

These speed objects are continuously created at set distances from one another by calling the function *createObstacle()* in *draw_world()*. Initially the object type and colour (Sphere/Cube, Blu/Green/White) was randomly generated for each instance. After testing we realized this was frustrating for the player at this stage so we opted for a pre-set order of generation (Green and Blue Ball, Green and Blue Cubes, White Sphere). The randomness could be re-introduced in future versions of the software (check FUTURE DEVELOPMENTS section of this report).

Using the arrow keys (← ; →) the player can move left/right up to a certain distance from the centre of the screen, this way deciding whether to pick up an object or not and simply placing itself on the far right or far left to avoid picking up any.

As the user picks up an object (i.e. green cube in **Figure 9A**), it is placed directly above the player (pink cube) and kept there as long as the player doesn't move to either left or right of that position. While the object stays above the player, the corresponding speed keeps increasing. As soon as the player moves, a new object collides or the maximum speed in that particular direction is reached the object is released and the speed stops increasing or changes according to the new object. If the maximum speed in a specific direction is reached the player won't be able to pick up the relative object anymore (i.e. if the player is already travelling along the real axis with a speed of +50 it won't be able to pick up any more green cubes).

This mechanism is implemented through the collision system: When a collision is detected, the object is translated above the player (in direction of the positive Y axis) and given the same z-velocity as the player. Because the collision is only based on the X and Z axes, as long as the player does not move away from the object, the collision will keep triggering and the effect will apply.

As soon as the player moves away from the object, or if the player picks up another object, the object, not colliding with the player, will stop having the same z-velocity as the player, and thus fall behind the player. Collisions are handled in the function *handleCollisions()* called in *draw_world()*.

Any object going behind the player is then automatically deleted with the use of the function *remove_current_bonus().*

With this method the player can adjust the speed by the amount of time he keeps the object above him.

The same concept is valid in every direction (Real and Imaginary axis), with objects of the same shape but opposite sign having the result of slowing down the change (i.e. if the user is increasing its real part with a speed of +40, picking up a blue cube and keeping it up for a few seconds will make the speed be +10). Every new object colliding on the player will replace the currently held one (if any) and apply its function on the speed.

Picking up a white sphere will reset both speeds to zero fixing a specific *c* value and the corresponding Julia set in the background. The player can then decide to move again by picking up another object.

With fine tuning of the speed through the methods described above the user can navigate the complex plane and find different Julia sets to represent in the Background.

The Mandelbrot set in the map will give the user hints on where to go to get the most interesting shapes and instruct the user on the fine line (borders of the set) existing between order (black regions) and chaos (coloured regions) corresponding to connected and disconnected Julia sets. The player will soon realize how the speed has to be drastically reduced when reaching the borders of the Mandelbrot set (**Figure 9B**) compared to the black regions in order to appreciate the evolving of different Julia sets and

avoid ending up in regions of complete chaos with non-interesting shapes. Moreover, when the zoom increases the speed will have to be further decreased. As the resolution increases with the zoom a tiny change in the *c* value will show a dramatic change in the local Julia set. This hints at the infinite density of these objects and the numbers generating them. While there is a zoom-like function, the player never actually gets any closer to the background. Rather, the shader takes into account some uniforms which control both the zoom value and the location of the camera. As such, the player is always seeing the whole shader and the shader itself is changing.

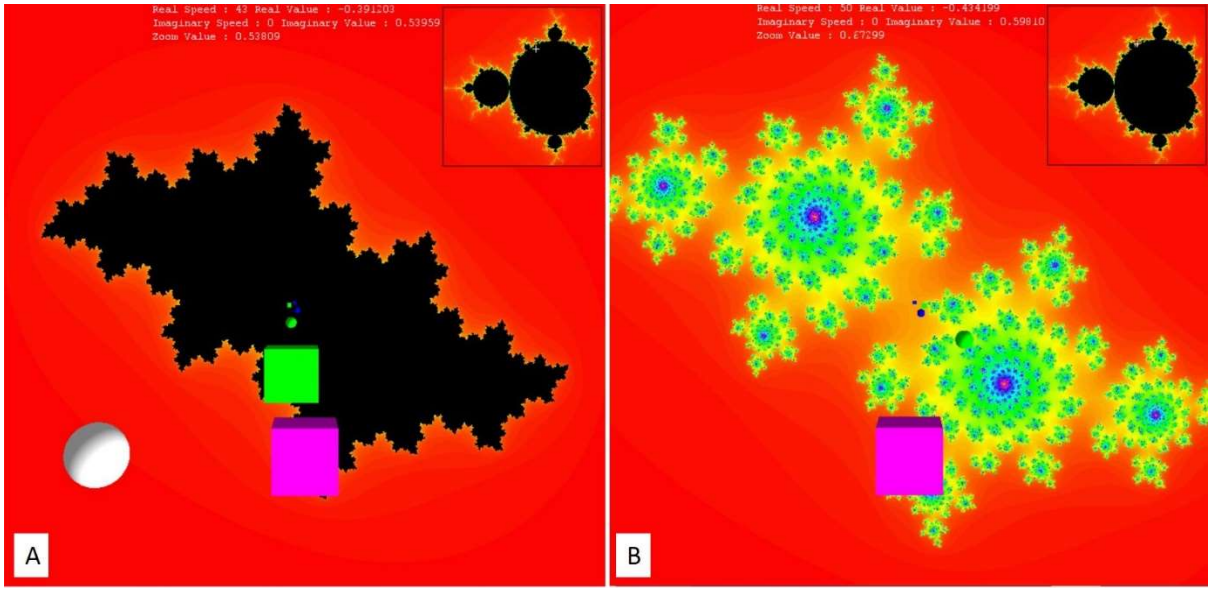


Figure 9 – **(A)** The player acquires the green cube which is placed directly above. c = (-0.39, 0.53). Zoom value of 0.5 **(B)** A Julia set generated by reaching a seed at the border of the Mandelbrot set. c = (-0.43, 0.59). Zoom value of 0.8.

The player has a few more hotkeys he can use to optimize/change the background image:

1- Arrow keys (UP,DOWN,LEFT,RIGHT) when SHIFT is held down: move the background around. This is particularly useful at high zooms to find interesting local regions of the Julia set (**Figure 10**).
2- SPACEBAR: the user can zoom out in case he wants to appreciate bigger areas of the background image. The program will then resume its automatic zoom in state.
3- f1: Choose among 6 different colour palettes available. (**Figure 11**). This is done by modifying the algorithm used to convert an HSV color (Hue, Saturation, Value) into an RGB color (Red, Green, Blue). A number of different presets have been established, but there would potentially be an infinite number of colour-sets available.
4- Ctrl/Backspace: Increase/decrease the number of colours available in the chosen Palette continuously (**Figure 12**). This is also achieved by varying a uniform parameter that is passed to the shader.

Finally, if the cursor goes beyond the borders of the Mandelbrot mini-map the value of *c* is reset to the origin (0, 0) and the speed change is set to 0 in both the Real and Imaginary axis showing the circle of radius 1 as Julia set in the background. This can be viewed as either a "Penalty" for the player for not controlling its speed or a convenient way to reset its position and speed to the initial one. It is worth mentioning that on the minimap shader (*Mandelbrot.fs*), the black border and the black cross indicating the position are not of type *GameObject*. They are all done in the fragment shader itself by comparing the uniforms with the current fragment's position.

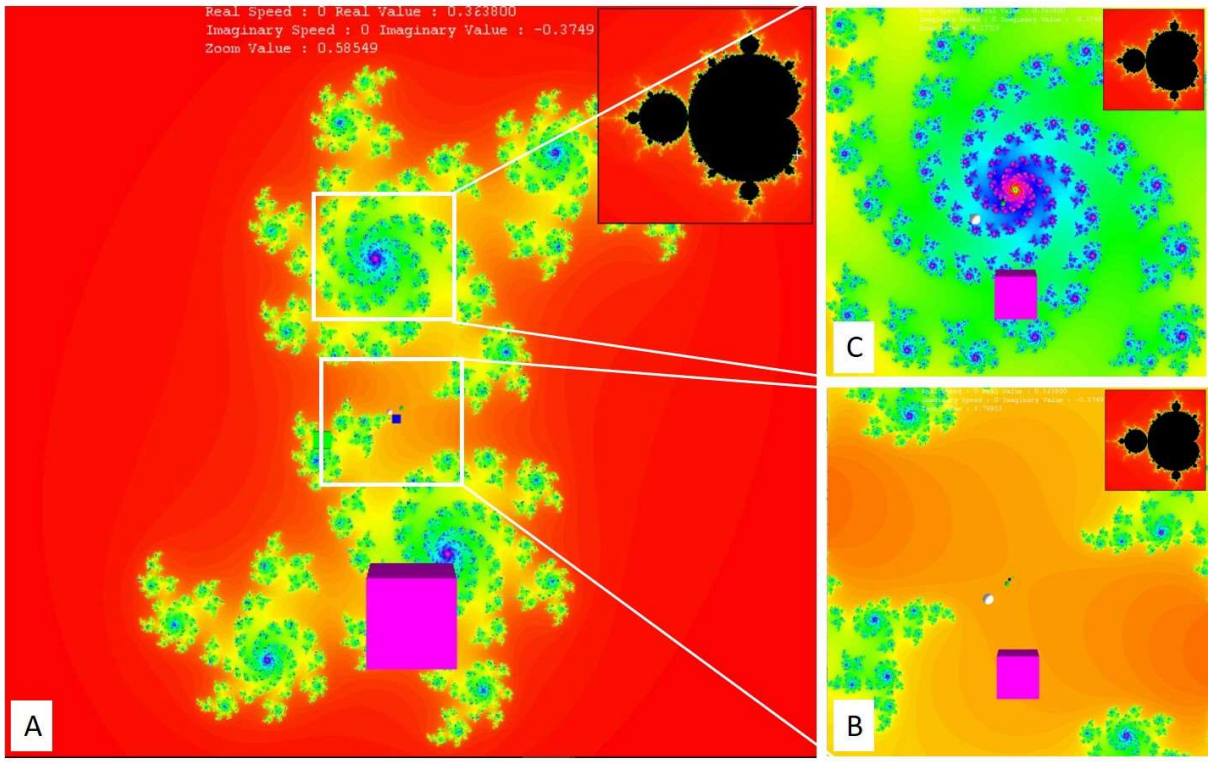


Figure 10 – **(A)** Fully disconnected Julia set [c = (0.36, -0.37)] at low zoom (0.58). **(B)** The same Julia set at higher magnification (3.8) in the default background position (zooming towards origin of complex plane). **(C)** The same Julia set at the same higher magnification but in a more interesting region found by the player by moving the background with SHIFT + (Arrow Keys).

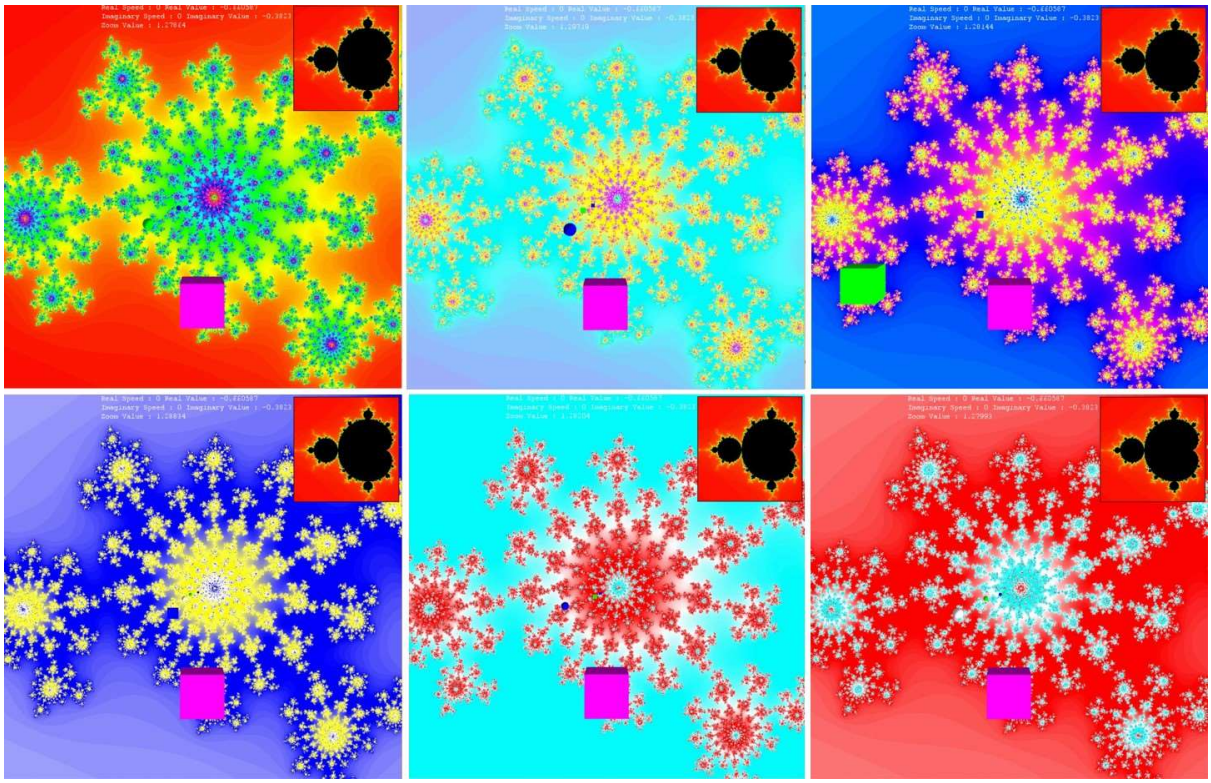


Figure 11 – Particular of a Julia set [c = (-0.66 , -0.38)] at high zoom (1.27) in the six available colour palettes selectable by pressing f1. Default palette in top left square.

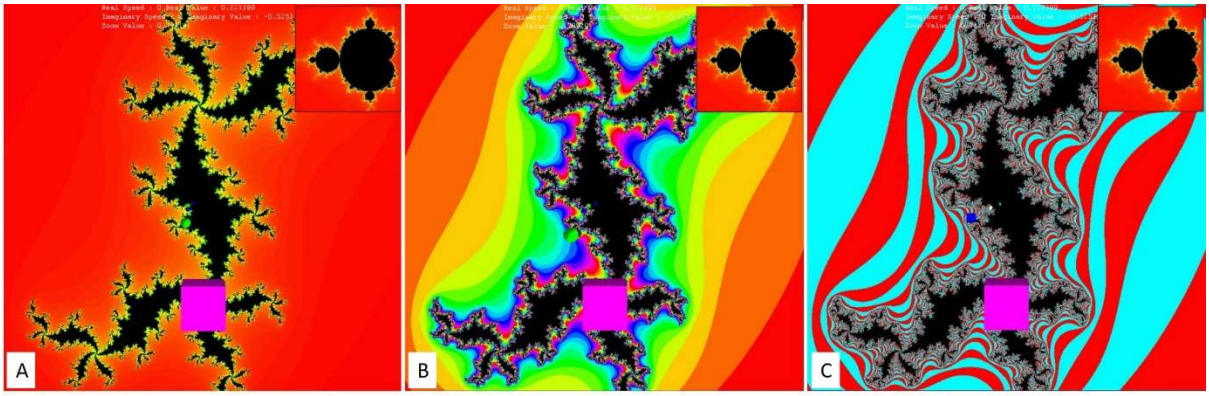


Figure 12 – Examples of decreased number of available colours in the default palette (c = [0.32, -0.52]. Zoom = 0.71). **(A)** Default. Maximum number of colours **(B)** Mid-Range. About half of the colours available **(C)** Low. Only 2 available colours.

INDIVIDUAL CONTRIBUTIONS and TEAMWORK

Individual Contributions

After both agreeing on the topic of the project being chaos and fractals we decided, based on our different skills and background, that Jean-Pascal would be the technical lead and Elio the content/gameplay lead. These roles were not strictly enforced as ideas on changes, controls, graphics and implementation came from both team members after frequent discussions, meetings and testing.

Jean-Pascal adapted the code to render the Mandelbrot and Julia sets in OCTET through shaders and built the environment to implement movement, zoom and object creation tools.

Elio added the collision system, tuned the objects generation, speed increase, zoom speed and borders interaction of the cursor in the minimap.

Both Elio and Jean-Pascal worked on tuning the colour palette system (number of colours and palette type respectively).

Towards the end of the project, Elio focused on background research making sure the mathematical theory, on which the work was based, was precise and correctly implemented. On the other hand Jean-Pascal focused on code optimization for a smooth, bug-free user experience and a clean code to be easily read and understood.

The project ended with a final meeting where the two team members reviewed the complete program and theory behind it, tested the gameplay and finalized this report by adding ideas for future iterations.

Teamwork

Overall the teamwork was good on both design and technical aspects. Every decision on gameplay mechanics was taken after testing different propositions from the two team members and compromise on divergent ideas.

There was also space for learning from one another as Elio gained some important technical skills from Jean-Pascal's coding proficiency in C++ and Jean-Pascal learned from Elio's research on the math theory underlying the project.

Moreover, when neither of us knew how to implement certain features, discussion was crucial as bouncing off ideas to one another we were often able to come up with solutions to our problems.

While the very different coding skill level surely helped one team member learn a lot from the other, it also slowed down the whole development process which would have been faster if all the people involved had the same skills. This would have given the chance to add extra features to the project that had to be left out in this version for time constraints. We were able to overcome this issues with patience, perseverance and by tuning our objectives to match our resources, limitations and time constraints. We both feel that frequent in-person meetings helped this aspect a lot as virtual chats are often frustrating and hinder a good communication and understanding. If both team members didn't work so well together and made the effort to meet frequently in person this would have probably been a big issue that could have caused the project to stall.

In the end, we both agree that working together was a positive and fruitful experience and would definitely consider doing it again on a future project.

CONCLUSIONS and FUTURE DEVELOPMENTS

In this version of the project we aimed at creating an experience that would educate the user on the amazing properties and relationship between the Julia and Mandelbrot fractals while delivering a visually engaging, psychedelic experience.

Rather than simply creating a tool to artificially explore these fractals we implemented a light gamification technique that challenges the users and, at the same time, doesn't frustrate them with particular aims, obstacles, and targets. The player is free to explore this world at its own pace by tuning its speed, resetting it or stopping it while playing with the different colour effects. There is no win or lose, just experience.

A natural future evolution of the software would be towards a more gamified experience up to a full-fledged game. We have envisioned a full-game version of this work with some or all of the features below:

- Snake-like mechanics of catching points randomly appearing on the Mandelbrot minimap triggering a higher z-speed, zoom value with a score system.
- Randomization of the speed objects.
- Add enemies/obstacles as objects to avoid.
- Add areas to avoid in the Mandelbrot minimap.
- Add objects to change the colour palette or award the possibility to change it after a certain score is reached.
- Incorporating the zoom feature as gameplay element
- Creating a save feature that would allow the user to save a picture of their favourite Julia sets.

FURTHER INFORMATION

A Demo video of the software can be found on Youtube: https://youtu.be/K9yLzbr9zr8

The source code can be found and forked on GitHub: https://github.com/JeanPascalEvette/MathRunner

REFERENCES

[1] Wikipedia: https://en.wikipedia.org/wiki/Complex_plane

[2] Fractal Extreme software. http://www.cygnus-software.com/

[3] Fractal Foundation: http://fractalfoundation.org/resources/what-are-fractals/

[4] Adrien Douady and John H. Hubbard. Etude dynamique des polynômes complexes, Prépublications mathémathiques d'Orsay 2/4 (1984 / 1985).

[5] University of Wisconsin Osh Kosh. http://www.uwosh.edu/faculty_staff/kuennene/Chaos

[6] Wikipedia: https://en.wikipedia.org/wiki/Mandelbrot_set

[7] Andy Thomason's OCTET: https://github.com/andy-thomason/octet

[8] Lode's Computer Graphics Tutorial: http://lodev.org/cgtutor/juliamandelbrot.html