Création d’images par algorithme génétique avec référence

Rapport de projet

Lucien Cartier-Tilet

April 29, 2019

Contents

1 Sujet **4**

2 Les méthodes utilisées **5**
2.1 Méthode naïve (méthode 1) 5
2.2 Réduction du panel des couleurs (méthode 2) 6
2.3 Une taille des formes aléatoire mais contrôlée (méthode 3) 6
2.4 Concurrence entre threads (méthode 4) 7
2.5 Collaboration entre threads (méthode 5) 7

3 Conclusion **9**

4 Annexes **10**
4.1 Code 10
4.1.1 `src/main.cc` 10
4.1.2 `include/genimg/shapes.hh` 10
4.1.3 `src/shapes.cc` 11
4.1.4 `include/genimg/methods.hh` 12
4.1.5 `src/methods.cc` 15
4.1.6 `include/genimg/parseargs.hh` 23
4.1.7 `src/parseargs.cc` 23
4.2 Images 25
4.2.1 Image de référence 25

Avant-propos

Ce document est un rapport de projet dont le sujet est décrit ci-dessous. Si vous souhaitez obtenir le code source sans avoir à recopier celui contenu dans ce fichier PDF, vous pouvez vous rendre sur son dépôt Git situé à l'adresse suivante : `https://labs.phundrak.fr/phundrak/genetic-images`

Notes sur le code

Le code source de ce projet repose sur trois dépendances :

- `boost_options`
- `spdlog`
- `opencv`

Cependant, seul `opencv` est une dépendance que l'on peut appeler « cœure » au projet ; en effet, c'est grâce à cette bibliothèque que sont exécutées toutes les opérations de manipulation d'images, allant de la simple ouverture de fichier image à la génération des images telle que décrite ci-dessous. `boost_options` est utilisé afin de parser les options du programme passées dans le terminal, et `spdlog` permet de générer une sortie utile pour la version `debug` du programme, affichant des données utiles pour un développeur travaillant sur le code source (en l'occurrence moi-même). Ces deux dernières dépendances n'ont pas d'impact sur la rapidité du programme.

1 Sujet

Le sujet de ce projet est la création d'un logiciel pouvant recréer une image fournie grâce à des générations aléatoires et successives de formes aux positions, couleurs et taille aléatoires. À chaque étape, une évaluation de la ressemblance entre l'image générée et l'image de référence est effectuée. Si l'algorithme constate que l'application d'une de ces formes aléatoire à l'image générée améliore cette ressemblance entre cette dernière et l'image de ressemblance, la modification est conservée. Sinon elle est ignorée, et une nouvelle image candidate au remplacement de l'image générée est créée. En d'autres mots, il s'agit d'un équivalent de sélection naturelle au sein des images générées aléatoirement ; seules les meilleures sont retenues et seront la base des images générées aléatoirement ultérieures. Ce processus se répètera autant de fois que l'utilisateur le souhaite.

2 Les méthodes utilisées

Plusieurs approches au problème sont possibles, allant de la simple implémentation naïve du problème à des moyen pouvant fortement accélérer la vitesse de génération de l'image ainsi que sa qualité. Deux types de formes aléatoires ont été implémentés : des triangles et des carrés. Les triangles sont pour l'instant restreint dans leur dimensions –en effet, ils tiennent tous dans un carré, leur hauteur et largeur sur les axes `x` et `y` étant égales. Il serait toutefois possible d'implémenter une version de ce programme générant une hauteur et une largeur toutes deux indépendantes l'une de l'autre pour une version future du programme. Le choix de la forme se fait via l'utilisation d'une option du programme, `-f [--form]` prenant pour valeur soit `1`, le choix par défaut, activant alors d'utilisation des carrés, ou bien la valeur `2` activant l'utilisation des triangles.

Pour évaluer la ressemblance entre deux image, j'évalue une distance euclidienne entre le vecteur de leurs pixels qui peut se résumer à ceci :

$$\sqrt{\sum_{i=0}^{n}(v_i - w_i)^2}$$

`V` étant le vecteur de pixels de l'image de référence, `W` étant le vecteur de pixels de l'image générée, et `n` la taille de ces deux vecteurs.

Les tests de temps sont réalisés sur un Lenovo Ideapad Y700, disposant d'un processeur Intel® Core™ i7-6700HQ à 2.6GHz et un turbo à 3.5GHz, composé de quatre cœurs supportant chacun deux threads, et l'ordinateur dispose d'un total de de 16Go de RAM. Le programme est compilé avec clang 8.0 et avec les options `-Wall -Wextra -Wshadow -Wpedantic -pedantic -O3 -flto`.

Voici également ci-dessous la liste des options et arguments possibles concernant l'exécution du logiciel.

```
$ ./bin/genetic-image -h
Allowed options:
  -h [ --help ]           Display this help message
  -i [ --input ] arg      Input image
  -o [ --output ] arg     Image output path (default: "output_" + input path)
  -n [ --iterations ] arg Number of iterations (default: 2000)
  -m [ --method ] arg     Method number to be used (default: 1)
  -f [ --form ] arg       Select shape (1:square, 2:triangle)
  -c [ --cols ] arg       For method 5 only, number of columns the reference
                          image should be divided into. If the value is equal
                          to 0, then it will be assumed there will be as many
                          rows as there are collumns. (default: 0)
  -r [ --rows ] arg       For method 5 only, number of rows the reference image
                          should be divided into. (default: 1)
  -S [ --submethod ] arg  Sub-method that will be used to generate the
                          individual tiles from method 5. (default: 1)
  -s [ --size ]           Enables controlled size of the random shapes
  -v [ --verbose ]        Enables verbosity
```

2.1 Méthode naïve (méthode 1)

J'ai tout d'abord implémenté la méthode naïve afin d'avoir une référence en matière de temps. Cette dernière est implémentée dans `src/methods.cc` avec la fonction `method1()`. Comme ce à quoi je m'attendais, cette méthode de génération d'images est très lente, principalement dû au fait que l'algorithme en l'état essaiera d'appliquer des couleurs n'existant pas dans l'image de référence, voire complètement à l'opposées de la palette de couleurs de l'image de référence.

Voici les options de la commande utilisée ici :

```
-i img/mahakala-monochrome.jpg -n2000 -m1 -f1 # carrés
-i img/mahakala-monochrome.jpg -n2000 -m1 -f2 # triangles
```

Voici les moyennes de temps d'exécution de cette méthode pour 2.000 améliorations :

carrés	triangles
10m 40s 615ms	4m 57s 987ms

On constate une plus grande rapidité d'exécution lors de l'utilisation de carrés. Je suppose que cela est dû à une facilité avec les triangles à créer des traits et bordures inclinées pour lesquelles plusieurs carrés seraient nécessaires, ainsi qu'une meilleure flexibilité quant aux formes pouvant être créées.

Vous trouverez en Annexes un exemple d'image générée à partir de `img/mahakala-monochrome.png` avec 2000 améliorations via cette méthode.

2.2 Réduction du panel des couleurs (méthode 2)

Constatant que la majorité des échecs d'ajout de formes de couleur par la première méthode échouent dû à une couleur incorrecte, voire n'appartenant pas à l'image de référence, j'ai décidé de restreindre les possibilités de couleurs parmis lesquelles le hasard peut choisir à la liste des couleurs présentes dans l'image de référence uniquement. Ce choix se fait donc via l'implémentation d'un set de valeurs uniques représentant les couleurs trouvées dans l'image de référence, leur détection étant réalisée avec des threads parallèles pour plus de rapidité à l'exécution. Cette méthode est celle implémentée dans la fonction `method2()` dans `src/methods.cc`.

Voici les options de la commande utilisée ici :

```
-i img/mahakala-monochrome.jpg -n2000 -m2 -f1 # carrés
-i img/mahakala-monochrome.jpg -n2000 -m2 -f2 # triangles
```

Voici le temps moyen d'exécution pour cette méthode avec 2.000 améliorations :

carrés	triangles
7m 23s 774ms	2m 57s 152ms

On peut remarquer une amélioration quant à la rapidité d'exécution du logiciel. Cependant, le résultat n'est pas aussi important qu'escompté. Je suppose que cela est dû au fait que l'algorithme précédent peut considérer un rapprochement d'une zone déjà colorée vers la couleur d'origine comme une amélioration, avec une possibilité plus large sur ce plan-là que pour le second algorithme qui se doit d'être plus précis concernant les couleurs. Une nette amélioration du résultat est toutefois visibles (voir Annexes pour une image générée à partir de `img/mahakala-monochrome.png` via la méthode 2 et avec 2000 améliorations).

À nouveau, on constate également un meilleur temps avec les triangles qu'avec les carrés.

Étant donné que cette modification ne sera à priori pas en conflit avec d'autres méthodes, cette amélioration sera conservée pour toutes les autres avancées suivantes.

2.3 Une taille des formes aléatoire mais contrôlée (méthode 3)

Une autre méthode peut être de contrôler la taille des éléments en spécifiant une taille minimale et maximale selon le nombre d'éléments posés et le nombre total d'éléments à poser. Ainsi, on pourrait privilégier tout d'abord de grandes formes en début de génération pour encourager petit à petit les formes à réduire en taille. Cela permettrait d'obtenir rapidement une représentation grossière de l'image pour ensuite pouvoir progressivement afiner les détails. La taille de la forme à appliquer est définie comme suit :

$$coef = \frac{nbIterRestantes}{totalIter}$$

$$tailleMinimale = coef \cdot \frac{min(Width, Height)}{2}$$

$$tailleMaximale = tailleMinimale \cdot 2 + 1$$

$$taille = Rand(\llbracket tailleMinimale; tailleMaximale \rrbracket)$$

Cette version du logiciel est nettement plus lente que ses versions précédentes du fait de la contrainte de taille pour les formes pouvant potentiellement améliorer l'image, et cela à un tel point que pour 2.000 améliorations, le programme tourne durant plus d'une heure. De plus, la qualité de l'image ainsi générée n'est pas nécessairement meilleure, ainsi cette méthode n'est pas nécessairement bonne. Toujours est-il que j'ai laissé pour les méthodes suivantes une option pour l'utilisateur lui permettant d'activer le contrôle de la taille des éléments s'il le souhaite via l'option `-s [--size]`. Toutefois, les temps d'exécution des méthodes suivantes n'en tiendront pas compte.

2.4 Concurrence entre threads (méthode 4)

Une utilisation de calculs parallèles pourrait être intéressante afin d'accélerer la génération des images : l'utilisation de threads mis en concurrence. À chaque tentative d'amélioration de l'image, plusieurs threads sont lancés, et chacun créera sa propre amélioration possible de l'image. Ces résultats sont récupérés et évalués, et parmi les résultats améliorant l'image générée, celle avec le meilleur score est conservée. Cela permet ainsi de multiplier les chances d'avoir une amélioration de l'image par tentative.

Voici les options de la commande utilisée ici :

```
-i img/mahakala-monochrome.jpg -n2000 -m4 -f1 # carrés
-i img/mahakala-monochrome.jpg -n2000 -m4 -f2 # triangles
```

Voici les moyennes de temps d'exécution de cette méthode pour 2.000 améliorations sans contrôle de la taille des éléments générés :

carrés	triangles
2m 30s 636ms	58s 885ms

Nous avons cette fois-ci une très nette amélioration de la vitesse d'exécution de l'algorithme, ce dernier étant jusqu'à environ trois fois plus rapide que la seconde méthode. Il s'agit d'une claire amélioration de la seconde méthode, qui elle-même présentait une amélioration de rapidité et de qualité d'image par rapport à la première méthode. En revanche, il n'y a pas de différence de qualité d'image visible au bout de ces 2.000 améliorations de visible à l'œil nu entre les deuxième et quatrième méthodes.

2.5 Collaboration entre threads (méthode 5)

Une différente approche au parallélisme peut être réalisée : plutôt que d'essayer de mettre en concurrence plusieurs threads, il serait possible d'essayer de plutôt les mettre en collaboration. Cela implique par exemple de diviser l'image d'entrée en plusieurs zones sur laquelle chacun des threads lancés travailleraient, appliquant chacun le nombre d'améliorations demandé sur sa zone dédiée. Puis, une fois que chacun des threads a terminé son travail, les différentes zones sont unifiées en une seule image. Plusieurs choix s'offrent alors à nous une fois les différents threads lancés concernant leur méthode de génération d'image, il est possible d'utiliser chacune des méthodes précédentes. Ce choix se fera via une option supplémentaire `--sub-method` ou `-S`.

J'ai choisi d'utiliser les deux méthodes les plus efficaces comme sous-méthodes, la deuxième et la quatrième. J'ai choisi la deuxième car il s'agit actuellement de la méthode la plus efficace qui ne soit pas parallélisée, et je pense que cela pourrait peut-être présenter un avantage par rapport à la quatrième méthode qui risque de se retrouvée ralentie par un trop grand nombre de threads lancés en même temps ; une image découpée en deux lignes et cinq colonnes, comme cela va être le cas ci-dessous, lancera sur le processeur de tests au maximum $2 \times 4 \times 8$ threads en même temps, soit 16 threads simultanés potentiels, deux fois plus donc qu'avec la méthode 2 en sous-méthode, qui ne tournera que sur huit threads.

Voici les options de la commande utilisée ici :

```
-i img/mahakala-monochrome.jpg -n2000 -m5 -c2 -r4 -S4 -f1 # carrés
-i img/mahakala-monochrome.jpg -n2000 -m5 -c2 -r4 -S4 -f2 # triangles
```

Voici les temps moyens d'exécution pour la cinquième méthode, utilisant en sous-méthode la seconde présentée :

carrés	triangles
6m 26s 520ms	2m 6s 865ms

Et voici les temps moyens d'exécution pour la même méthode utilisant la quatrième méthode comme sous-méthode :

carrés	triangles
4m 26s 78ms	1m 6s 984ms

Bien que les deux méthodes n'aient pas nécessairement bénéficiées d'amélioration quant à leur temps d'exécution, les images générées via cette cinquième méthode sont d'une qualité nettement supérieure aux images générées jusqu'à présent, avec un niveau de détail largement meilleur. La quatrième méthode utilisée en sous-méthode présente un avantage sur la seconde en considérant le temps d'exécution, et il semblerait que l'on puisse remarquer une légère amélioration de l'image également. Cependant, et bien que cela soit plus lent, la génération à base de carrés présente cette fois-ci un avantage par rapport à la génération à base de triangles : ces derniers cachent beaucoup plus aisément la séparation entre les différentes zones de génération de l'image que les triangles, sans doutes dû à la difficulté d'obtenir des zones de remplissages dont un des côtés soit parfaitement vertical ou horizontal avec ces derniers.

J'ai également remarqué que si l'on utilise un grand nombre de colonnes et de lignes, le programme peut mettre énormément de temps pour effectuer sa tâche. Avec l'image de test, j'ai exécuté le logiciel en divisant l'image de test en cinq colonnes et lignes, donnant un total de vingt-cinq threads exécutant chacun une méthode sur une zone de l'image ; j'ai dû manuellement arrêter le programme au bout de plus d'une heure de travail afin de ne pas perdre de temps. Je pense que cela est dû à certaines zones étant très homogènes, rendant le travail d'amélioration de cette zone très difficile quand la majorité de la zone est à peu près identique à sa version originale, chaque amélioration ne pouvant s'effectuer que sur quelques pixels tout au plus.

3 Conclusion

Plusieurs approches ont été abordées lors de ce projet quand à la meilleure méthode de génération d'images basée sur des formes aléatoires, aux couleurs aléatoires. Quelques unes d'entre elles portent sur un certain contrôle du facteur aléatoire, tandis que d'autres tentent de tirer profit du matériel utilisé afin de disposer d'une plus grande puissance de calcul. Alors que l'on a constaté qu'une limitation de l'arbitraire quant au panel de couleurs possibles est intéressante, celle de leur taille l'est moins, tout du moins avec la formule utilisée. Il serait néamoins intéressant de réitérer cette tentative avec une autre approche, comme par exemple réduire la taille maximale des éléments suite à un certain nombre d'échecs de génération de successeurs successifs.

En revanche, les méthodes consistant à répartir la charge de travail sur plusieurs threads ont été chacune couronnées de succès, et ultimement la combination d'une méthode se basant sur la concurrence entre threads elle-même lancée en collaboration avec d'autres de cette même méthode s'est prouvée être l'une des méthodes les plus efficaces afin de re-créer avec fidélité une image de référence.

Note intéressante : la génération d'image avec des triangles est plus rapide qu'une génération d'image basée sur des carrés, mais ces derniers seront la forme la plus efficace visuellement lors de cette cinquième et meilleure méthode présentée dans ce rapport.

4 Annexes

4.1 Code

4.1.1 src/main.cc

```
#include "methods.hh"
#include "parseargs.hh"

#include <cstdlib>
#include <ctime>
#include <iostream>

int main(int ac, char **av)
{
  std::srand(std::time(nullptr));
  spdlog::set_level(spdlog::level::debug);
  auto const arguments = parse_args(ac, av);
  spdlog::set_level(arguments.verbose ? spdlog::level::debug
                                      : spdlog::level::info);
  spdlog::set_pattern("[thread %t] %+");
  spdlog::debug("Input file:\t{}", arguments.input_path.native());
  spdlog::debug("Output file:\t{}", arguments.output_path.native());
  spdlog::debug("Iterations:\t{}", arguments.iterations);
  ImageManipulator image_process{arguments.input_path, arguments.output_path,
                                 arguments.iterations, arguments.shape};
  image_process.exec_method(arguments.method, arguments.controlled_size,
                            arguments.cols, arguments.rows,
                            arguments.submethod);
  image_process.write_file();
}
```

4.1.2 include/genimg/shapes.hh

```
#pragma once

#include <array>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <spdlog/spdlog.h>

class Shape
{
public:
  static constexpr int MAX_POINTS{4};
  enum class ShapeType { Square, Triangle };

  /// \brief Default constructor
  Shape() = delete;

  Shape(Shape::ShapeType const t_type);

  /// \brief Copy constructor
  Shape(const Shape &other) = default;

  /// \brief Move constructor
  Shape(Shape &&other) noexcept;

```

```
25    /// \brief Destructor
26    virtual ~Shape() noexcept = default;
27
28    /// \brief  Copy assignment operator
29    Shape &operator=(const Shape &other) = delete;
30
31    /// \brief Move assignment operator
32    Shape &operator=(Shape &&other) noexcept = delete;
33
34    /// \brief Generates a shape's points
35    void update(cv::Point &&t_max_pos, int const t_max_size,
36                int const t_min_size = 1) noexcept;
37
38    [[nodiscard]] auto get_points() const noexcept
39        -> std::array<cv::Point, Shape::MAX_POINTS> const &
40    {
41      return points_;
42    }
43
44    /// \brief Returns the type of shape described by the object
45    [[nodiscard]] auto get_type() const noexcept -> ShapeType const &
46    {
47      return type_;
48    }
49
50    [[nodiscard]] auto get_nb_points() const noexcept
51    {
52      return nb_points_;
53    }
54
55  protected:
56  private:
57    void create_square_points(cv::Point const &t_top_left,
58                              int const t_size) noexcept;
59    void create_triangle_points(cv::Point const &t_top_left,
60                                int const t_size) noexcept;
61
62    ShapeType const type_{ShapeType::Square};
63    std::array<cv::Point, Shape::MAX_POINTS> points_{
64        cv::Point{0, 0}, cv::Point{0, 0}, cv::Point{0, 0}, cv::Point{0, 0}};
65    int nb_points_{Shape::MAX_POINTS};
66  };
```

4.1.3 src/shapes.cc

```
1   #include "shapes.hh"
2
3   #include <cmath>
4   #include <utility>
5
6   using point_arr = std::array<cv::Point, 4>;
7
8   Shape::Shape(Shape::ShapeType const t_type) : type_{t_type}
9   {
10    switch (t_type) {
11    case ShapeType::Triangle: nb_points_ = 3; break;
12    case ShapeType::Square: nb_points_ = 4; break;
```

```
13     default: nb_points_ = 4; break;
14     }
15   }
16
17   Shape::Shape(Shape &&other) noexcept
18       : type_{std::move(other.type_)}, points_{std::move(other.points_)},
19         nb_points_{std::move(other.nb_points_)}
20   {
21   }
22
23   /**
24    *  Generates all the needed points for the corresponding shape described in
25    *  \ref type_.
26    *
27    *  \param t_max_pos Bottom-rightmost point of the image the shape is generated
28    *                   for
29    *  \return Array of points describing the shape
30    */
31   void Shape::update(cv::Point &&t_max_pos, int const t_max_size,
32                      int const t_min_size) noexcept
33   {
34     int const size = (rand() % (t_max_size - t_min_size)) + t_min_size + 1;
35     cv::Point const top_left
36         = {rand() % (t_max_pos.x - size + 1), rand() % (t_max_pos.y - size + 1)};
37     if (type_ == ShapeType::Triangle) {
38       create_triangle_points(top_left, size);
39     } else {  // ShapeType::Square
40       create_square_points(top_left, size);
41     }
42   }
43
44   void Shape::create_triangle_points(cv::Point const &t_top_left,
45                                      int const t_size) noexcept
46   {
47     bool top_left = rand() % 2 == 0;
48     points_       = {
49         cv::Point{top_left ? t_top_left.x : t_top_left.x + t_size, t_top_left.y},
50         cv::Point{top_left ? t_top_left.x + t_size : t_top_left.x,
51                   t_top_left.y + rand() % t_size},
52         cv::Point{t_top_left.x + rand() % t_size, t_top_left.y + t_size},
53         cv::Point{0, 0}};
54   }
55
56   void Shape::create_square_points(cv::Point const &t_top_left,
57                                    int const t_size) noexcept
58   {
59     points_ = {cv::Point{t_top_left.x, t_top_left.y},
60                cv::Point{t_top_left.x, t_top_left.y + t_size},
61                cv::Point{t_top_left.x + t_size, t_top_left.y + t_size},
62                cv::Point{t_top_left.x + t_size, t_top_left.y}};
63   }
```

4.1.4 include/genimg/methods.hh

```
1    #pragma once
2
3    #include "shapes.hh"
```

```cpp

#include <string>
#include <vector>

class ImageManipulator
{
public:
  ImageManipulator() = delete;

  /// \brief Copy contructor
  ImageManipulator(const ImageManipulator &other);

  /// \brief Move constructor
  ImageManipulator(ImageManipulator &&other) noexcept;

  /// \brief Load image from input, and prepare for output
  ImageManipulator(std::string const t_input_path,
                   std::string const t_output_path, int const iterations,
                   Shape::ShapeType const t_shape);

  /// \brief Basically makes views from image
  ImageManipulator(cv::Mat const &t_origin_image, int const t_iterations,
                   Shape::ShapeType const t_shape, int const t_x, int const t_y,
                   int const t_width, int const t_height);

  [[nodiscard]] auto operator=(ImageManipulator &other) = delete;

  [[nodiscard]] auto operator=(ImageManipulator &&other) noexcept = delete;

  /// \brief Execute the nth method on the current object
  void exec_method(int const t_nb_method, bool const t_controlled_size,
                   int const t_cols, int const t_rows, int const t_submethod);

  /**
   *  \brief Write the generated image to the output path
   *
   *  Write the generated image as a file to the specified path stored in the
   *  object
   */
  inline void write_file() const
  {
    cv::imwrite(output_path_, generated_image_);
  }

  /// \brief Returns a reference to the generated image
  [[nodiscard]] inline auto const &get_generated_image() const noexcept
  {
    return generated_image_;
  }

  /// \brief Destructor
  virtual ~ImageManipulator() noexcept = default;

protected:
private:
  // methods ////////////////////////////////////////////////////////////////

```

```cpp
/// \brief Calculates the euclidian distance between two images
[[nodiscard]] auto euclidian_distance(cv::Mat const &t_img) const noexcept
    -> double;

/// \brief Creates and returns a random color
[[nodiscard]] auto random_color() const noexcept;

/// \brief Generates a candidate for image generation improvement
[[nodiscard]] auto create_candidate(bool const t_controlled_size);

/// \brief Generates organized views of the reference image for method 5
[[nodiscard]] auto generate_tiles(int const t_cols, int const t_rows) const;

/// \brief Gets all colors from the reference image
void get_color_set();

/// \brief Threaded helper for \ref get_color_set
void threaded_get_color(int const t_h);

void draw_shape(cv::Mat &t_img, cv::Scalar &&t_color);
void create_shape() noexcept;
void create_controlled_shape() noexcept;

/// \brief Update this object's generated image
void update_gen_image(cv::Mat const &t_img, double const t_diff);

/// \brief Merges tiles generated by method5
void merge_tiles(std::vector<std::vector<ImageManipulator>> const &t_tiles);

/// \brief First method as described in the
/// [report](https://labs.phundrak.fr/phundrak/genetic-images/blob/master/report/report.pdf)
void method1();

/// \brief Second method as described in the
/// [report](https://labs.phundrak.fr/phundrak/genetic-images/blob/master/report/report.pdf)
void method2();

/// \brief Third method as described in the
/// [report](https://labs.phundrak.fr/phundrak/genetic-images/blob/master/report/report.pdf)
void method3();

/// \brief Fourth method as described in the
/// [report](https://labs.phundrak.fr/phundrak/genetic-images/blob/master/report/report.pdf)
void method4(bool const t_controlled_size);

/// \brief Fifth method as described in the
/// [report](https://labs.phundrak.fr/phundrak/genetic-images/blob/master/report/report.pdf)
void method5(bool const t_controlled_size, int const cols, int const rows,
             int const submethod);

// members /////////////////////////////////////////////////////////////////

std::vector<std::array<uchar, 3>> colors_{}; /*!< Color set from reference */
cv::Mat const reference_{};                  /*!< Reference image */
cv::Mat generated_image_{
    reference_.size().height, reference_.size().width, CV_8UC3,
    cv::Scalar(0, 0, 0)}; /*!< Working, generated image */
```

```
118   Shape shape_{Shape::ShapeType::Square};
119   mutable std::mutex
120       colors_mutex_{}; /*!< Thread mutex for color set generation */
121   std::string const output_path_{}; /*!< Write path for the generated image */
122   double diff_{euclidian_distance(generated_image_)}; /*!< Euclidian difference
123                              between \ref reference_ and \ref generated_image_ */
124   int const total_iterations_{0}; /*!< Number of iterations to perform */
125   int remaining_iter_{
126       total_iterations_}; /*!< Remaining iterations to perform */
127   int const width_{reference_.size().width};   /*!< Width of the image */
128   int const height_{reference_.size().height}; /*!< Height of the image */
129 };
```

4.1.5 src/methods.cc

```
1  #include "methods.hh"
2
3  #include <algorithm>
4  #include <future>
5  #include <optional>
6  #include <thread>
7
8  static auto const thread_nbr = std::thread::hardware_concurrency();
9
10 ///////////////////////////////////////////////////////////////////////////////
11 //                                   Public                                  //
12 ///////////////////////////////////////////////////////////////////////////////
13
14 // constructors ///////////////////////////////////////////////////////////////
15 /**
16  *  Copy constructor of \ref ImageManipulator, will copy all of its members
17  *  except for its mutex.
18  *
19  *  \param[in] other Element to copy
20  */
21 ImageManipulator::ImageManipulator(const ImageManipulator &other)
22     : colors_{other.colors_}, reference_{other.reference_},
23       generated_image_{other.generated_image_}, shape_{other.shape_},
24       output_path_{other.output_path_}, diff_{other.diff_},
25       total_iterations_{other.total_iterations_},
26       remaining_iter_{other.remaining_iter_}, width_{other.width_},
27       height_{other.height_}
28 {
29 }
30
31 /**
32  *  Move constructor of \ref ImageManipulator, will move all of the input's
33  *  members except for its mutex, a new one will be made.
34  *
35  *  \param[in] other Element to move
36  */
37 ImageManipulator::ImageManipulator(ImageManipulator &&other) noexcept
38     : colors_{std::move(other.colors_)}, reference_{std::move(
39                                              other.reference_)},
40       generated_image_{std::move(other.generated_image_)}, shape_{std::move(
41                                                                other.shape_)},
42       output_path_{std::move(other.output_path_)},
```

```cpp
      diff_{std::move(other.diff_)}, total_iterations_{other.total_iterations_},
      remaining_iter_{other.remaining_iter_}, width_{other.width_},
      height_{other.height_}
{
}

/**
 *  Creates an instance of \ref ImageManipulator based on an input path and an
 *  output path. It will load the input image from its first argument, and will
 *  write an output image when asked at the path passed as its second argument.
 *
 *  \param[in] t_input_path Path for the input, reference image
 *  \param[in] t_output_path Path to the output image to write
 */
ImageManipulator::ImageManipulator(std::string const t_input_path,
                                   std::string const t_output_path,
                                   int const t_iterations,
                                   Shape::ShapeType const t_shape)
    : reference_{cv::imread(t_input_path, cv::IMREAD_COLOR)}, shape_{Shape{
                                                                  t_shape}},
      output_path_{t_output_path}, total_iterations_{t_iterations}
{
  if (!reference_.data) {
    spdlog::critical("Could not open or find image!\n");
    exit(-1);
  }
}

/**
 *  Creates a view of the input image, and will generate an image based only on
 *  that view.
 *
 *  \param[in] t_origin_image Image to create a view from
 *  \param[in] t_iterations Number of iterations to perform on this view
 *  \param[in] t_x X value of the view's origin (top left)
 *  \param[in] t_y Y value of the view's origin (top left)
 *  \param[in] t_width Width of the view from its origin
 *  \param[in] t_height Height of the view from its origin
 */
ImageManipulator::ImageManipulator(cv::Mat const &t_origin_image,
                                   int const t_iterations,
                                   Shape::ShapeType const t_shape,
                                   int const t_x, int const t_y,
                                   int const t_width, int const t_height)
    : reference_{t_origin_image(
        cv::Range{t_y, std::min(t_y + t_height, t_origin_image.rows)},
        cv::Range{t_x, std::min(t_x + t_width, t_origin_image.cols)})},
      shape_{Shape{t_shape}}, total_iterations_{t_iterations}
{
  if (!reference_.data) {
    spdlog::critical("Could not open or find image!\n");
    exit(-1);
  }
}

// public methods ///////////////////////////////////////////////////////////////

```

```
100  /**
101   *  Execute one of the methods as described in the report. If a non-valid
102   *  method is called, the program will be terminated. The argument
103   *  `t_controlled_size` allows the program to have some control over the random
104   *  size of the squares that will be generated. The arguments `t_cols`, `t_rows`
105   *  and `submethod` are relevant to the fifth method.
106   *
107   *  \param[in] t_nb_method Method identifier
108   *  \param[in] t_controlled_size Control over the squares' size
109   *  \param[in] t_cols Number of columns the reference should be divided into
110   *  \param[in] t_rows Number of rows the reference should be divided into
111   *  \param[in] t_submethod
112   */
113  void ImageManipulator::exec_method(int const t_nb_method,
114                                     bool const t_controlled_size = false,
115                                     int const t_cols = 1, int const t_rows = 0,
116                                     int const t_submethod = 1)
117  {
118    switch (t_nb_method) {
119    case 1: method1(); break;
120    case 2: method2(); break;
121    case 3: method3(); break;
122    case 4: method4(t_controlled_size); break;
123    case 5: method5(t_controlled_size, t_cols, t_rows, t_submethod); break;
124    default:
125      spdlog::error("Requested method {} is not implemented.", t_nb_method);
126      std::exit(-1);
127    }
128  }
129
130  ///////////////////////////////////////////////////////////////////////////////
131  //                                   Private                                  //
132  ///////////////////////////////////////////////////////////////////////////////
133
134  // methods ////////////////////////////////////////////////////////////////////
135
136  /**
137   *  Calculates the euclidian distance between the reference image and the image
138   *  passed as an argument
139   *
140   *  \param t_img Image with which the distance is computed
141   *  \return double
142   */
143  [[nodiscard]] auto
144  ImageManipulator::euclidian_distance(cv::Mat const &t_img) const noexcept
145      -> double
146  {
147    double euclidian = 0.0;
148    for (auto itr1 = reference_.begin<uchar>(), itr2 = t_img.begin<uchar>();
149         itr1 != reference_.end<uchar>() && itr2 != t_img.end<uchar>();
150         ++itr1, ++itr2) {
151      euclidian += std::pow(*itr1 - *itr2, 2);
152    }
153    return std::sqrt(euclidian);
154  }
155
156  /**
```

```
157   *  \return cv::Scalar
158   */
159  [[nodiscard]] auto ImageManipulator::random_color() const noexcept
160  {
161    return cv::Scalar(rand() % 255, rand() % 255, rand() % 255);
162  }
163
164  void ImageManipulator::create_shape() noexcept
165  {
166    shape_.update(cv::Point{reference_.size().width, reference_.size().height},
167                  std::min(reference_.size().width, reference_.size().height));
168  }
169
170  void ImageManipulator::create_controlled_shape() noexcept
171  {
172    float const coef = static_cast<float>(remaining_iter_)
173                       / static_cast<float>(total_iterations_);
174    int const min_size
175        = static_cast<int>((static_cast<float>(std::min(reference_.size().width,
176                                                        reference_.size().height))
177                            / 2.0f)
178                           * coef);
179    int const max_size = min_size * 2 + 1;
180    shape_.update(cv::Point{reference_.size().width, reference_.size().height},
181                  max_size, min_size);
182  }
183
184  /**
185   *  Creates a temporary image on which a random square is drawn. If its
186   *  euclidian distance with the reference image proves to be an improvement from
187   *  the latest improvement before, then both the image and the distance are
188   *  returned. Otherwise, nothing is returned.
189   *
190   *  \param[in] t_controlled_size Enables controlled square size
191   *  \return Optional pair of cv::Mat and double
192   */
193  [[nodiscard]] auto
194  ImageManipulator::create_candidate(bool const t_controlled_size = false)
195  {
196    auto temp_img     = generated_image_.clone();
197    auto const &color = colors_[rand() % colors_.size()];
198    if (t_controlled_size) {
199      create_controlled_shape();
200    } else {
201      create_shape();
202    }
203    draw_shape(temp_img, cv::Scalar{static_cast<double>(color[0]),
204                                    static_cast<double>(color[1]),
205                                    static_cast<double>(color[2])});
206    auto new_diff = euclidian_distance(temp_img);
207    return (new_diff < diff_)
208               ? std::optional<std::pair<cv::Mat, double>>{std::make_pair(
209                   std::move(temp_img), new_diff)}
210               : std::nullopt;
211  }
212
213  /**
```

```
214  *  Generates views and stores them in a double vector so the tiles (views) are
215  *  stored by column top to bottom, and within the columns left to right.
216  *
217  *  \param t_cols Number of columns the reference image should be divided into
218  *  \param t_rows Number of rows the reference image should be divided into
219  *  \return Collection of tiles (vector<vector<ImageManipulator>>)
220  */
221 [[nodiscard]] auto ImageManipulator::generate_tiles(int const t_cols,
222                                                     int const t_rows) const
223 {
224   std::vector<std::vector<ImageManipulator>> tiles{};
225   int const tile_width  = reference_.cols / t_cols;
226   int const tile_height = reference_.rows / t_rows;
227   for (int index_x = 0; index_x < t_cols; ++index_x) {
228     std::vector<ImageManipulator> tile_col{};
229     for (int index_y = 0; index_y < t_rows; ++index_y) {
230       int const width = (index_x != t_cols - 1)
231                             ? tile_width
232                             : tile_width + reference_.cols % tile_width;
233       int const height = (index_y != t_rows - 1)
234                              ? tile_height
235                              : tile_height + reference_.rows % tile_height;
236       tile_col.emplace_back(reference_, total_iterations_, shape_.get_type(),
237                             index_x * tile_width, index_y * tile_height, width,
238                             height);
239     }
240     tiles.push_back(tile_col);
241   }
242   return tiles;
243 }
244
245 /**
246  *  Will analyse the reference image and will store each color found in member
247  *  variable \ref colors_. Works on multithreading.
248  */
249 void ImageManipulator::get_color_set()
250 {
251   for (int h = 0; h < reference_.size().height; h += thread_nbr) {
252     std::vector<std::thread> thread_list{};
253     for (auto i = 0u; i < thread_nbr; ++i) {
254       thread_list.push_back(
255           std::thread(&ImageManipulator::threaded_get_color, this, h + i));
256     }
257     std::for_each(thread_list.begin(), thread_list.end(),
258                   [](auto &th) { th.join(); });
259   }
260   colors_.shrink_to_fit();
261 }
262
263 /**
264  *  Will search for every color found in its designated column. If a new color
265  *  is found, pauses all its other similar threads, adds the new color in \ref
266  *  colors_, then resumes the other threads. Helper function for \ref
267  *  get_color_set
268  */
269 void ImageManipulator::threaded_get_color(int const t_h)
270 {
```

```
  if (t_h > reference_.size().height) {
    return;
  }
  for (int w = 0; w < reference_.size().width; w += 3) {
    std::array<uchar, 3> temp
        = {reference_.at<uchar>(t_h, w), reference_.at<uchar>(t_h, w + 1),
           reference_.at<uchar>(t_h, w + 2)};
    auto pos = std::find(std::begin(colors_), std::end(colors_), temp);
    if (pos == std::end(colors_)) {
      colors_mutex_.lock();
      colors_.push_back(std::move(temp));
      colors_mutex_.unlock();
    }
  }
}

void ImageManipulator::draw_shape(cv::Mat &t_img, cv::Scalar &&t_color)
{
  fillConvexPoly(t_img, shape_.get_points().data(), shape_.get_nb_points(),
                 t_color);
}

/**
 * Updates the object's current generated image and difference with its
 * reference by replacing them with the arguments passed in this function. This
 * function should only be called if the passed elements are improving the
 * generated image and reduce the euclidian distance between said image and its
 * reference.
 *
 * \param[in] t_img Image to replace \ref generated_image_
 * \param[in] t_diff New euclidian distance
 */
void ImageManipulator::update_gen_image(cv::Mat const &t_img,
                                        double const t_diff)
{
  diff_ = t_diff;
  t_img.copyTo(generated_image_);
  --remaining_iter_;
  spdlog::debug("remaining iter: {}\tdiff: {}", remaining_iter_, diff_);
}

/**
 * Merges the tiles generated by \ref method5 into a single image. The tiles
 * are organized by column top to bottom, within each they are stored in order,
 * left to right. They will be merged in \ref generated_image_.
 *
 * \param t_tiles Collection of tiles to be merged together
 */
void ImageManipulator::merge_tiles(
    std::vector<std::vector<ImageManipulator>> const &t_tiles)
{
  std::vector<cv::Mat> columns{};
  std::for_each(t_tiles.begin(), t_tiles.end(), [&columns](auto const &col) {
    std::vector<cv::Mat> column_arr{};
    cv::Mat column_img{};
    std::for_each(col.begin(), col.end(), [&column_arr](auto const &tile) {
      column_arr.push_back(tile.get_generated_image());
```

```cpp
    });
    vconcat(column_arr, column_img);
    columns.push_back(std::move(column_img));
  });
  hconcat(columns, generated_image_);
}

void ImageManipulator::method1()
{
  spdlog::debug("Beginning method1, initial difference: {}", diff_);
  while (remaining_iter_ > 0 && diff_ > 0.0) {
    auto temp_image = generated_image_.clone();
    create_shape();
    draw_shape(temp_image, random_color());
    if (auto const new_diff = euclidian_distance(temp_image);
        new_diff < diff_) {
      update_gen_image(temp_image, new_diff);
    }
  }
}

void ImageManipulator::method2()
{
  spdlog::debug("Beginning method2, initial difference: {}", diff_);
  spdlog::debug("Running on {} threads", thread_nbr);
  get_color_set();
  spdlog::debug("{} colors detected", colors_.size());
  while (remaining_iter_ > 0 && diff_ > 0.0) {
    if (auto result = create_candidate(); result.has_value()) {
      update_gen_image(result->first, result->second);
    }
  }
}

void ImageManipulator::method3()
{
  spdlog::debug("Beginning method3, initial difference: {}", diff_);
  spdlog::debug("Running on {} threads", thread_nbr);
  get_color_set();
  spdlog::debug("{} colors detected", colors_.size());
  while (remaining_iter_ > 0 && diff_ > 0.0) {
    auto temp_image = generated_image_.clone();
    if (auto result = create_candidate(true); result.has_value()) {
      update_gen_image(result->first, result->second);
    }
  }
}

/**
 *  \param[in] t_controlled_size Enables control over the random squares' size
 */
void ImageManipulator::method4(bool const t_controlled_size)
{
  spdlog::debug("Beginning method4, initial difference: {}", diff_);
  spdlog::debug("Running on {} threads", thread_nbr);
  get_color_set();
  spdlog::debug("{} colors detected", colors_.size());
```

```
  while (remaining_iter_ > 0 && diff_ > 0.0) {
    std::vector<std::future<std::optional<std::pair<cv::Mat, double>>>>
        results{};
    std::vector<std::pair<cv::Mat, double>> values{};
    // launch asynchronously candidate image generation
    for (size_t i = 0; i < thread_nbr; ++i) {
      results.push_back(std::async(std::launch::async,
                                   &ImageManipulator::create_candidate, this,
                                   t_controlled_size));
    }
    // if candidate is a success, store it
    std::for_each(results.begin(), results.end(), [&values, this](auto &elem) {
      if (auto res = elem.get(); res.has_value() && res->second < this->diff_) {
  values.push_back(*res);
      }
    });
    // apply best candidate
    if (values.size() > 0) {
      auto const pos
          = std::min_element(std::begin(values), std::end(values),
                             [](const auto &elem1, const auto &elem2) {
                         return elem1.second < elem2.second;
                             });
      update_gen_image(pos->first, pos->second);
    }
  }
}

/**
 *  \param[in] t_controlled_size Enables control over the random squares' size
 *  \param[in] t_cols Number of colomns the reference should be divided into
 *  \param[in] t_rows Number of rows the reference should be divided into
 *  \param[in] t_submethod Method to be used on each tile
 */
void ImageManipulator::method5(bool const t_controlled_size, int const t_cols,
                               int const t_rows, int const t_submethod)
{
  spdlog::debug("Beginning method5, initial difference: {}", diff_);
  spdlog::debug("Running on {} threads", thread_nbr);

  auto tiles = generate_tiles((t_cols != 0) ? t_cols : t_rows, t_rows);
  spdlog::debug("{} tiles", tiles.size());

  std::vector<std::thread> thread_list{};
  std::for_each(tiles.begin(), tiles.end(), [&](auto &row) {
    std::for_each(row.begin(), row.end(), [&](auto &tile) {
      thread_list.emplace_back(
          [&]() { tile.exec_method(t_submethod, t_controlled_size); });
    });
  });
  std::for_each(thread_list.begin(), thread_list.end(),
                [](auto &th) { th.join(); });
  merge_tiles(tiles);
}
```

4.1.6 include/genimg/parseargs.hh

```
#pragma once

#include "shapes.hh"

#include <filesystem>

struct ParsedArgs {
  std::filesystem::path input_path;
  std::filesystem::path output_path;
  Shape::ShapeType shape;
  int iterations;
  int method;
  int cols;
  int rows;
  int submethod;
  bool controlled_size;
  bool verbose;
};

/// \brief Parses the arguments passed to the program
[[nodiscard]] auto parse_args(int, char **) -> ParsedArgs;
```

4.1.7 src/parseargs.cc

```
#include "parseargs.hh"

#include <boost/program_options.hpp>
#include <iostream>

constexpr int DEFAULT_ITERATIONS = 2000;

using path   = std::filesystem::path;
namespace po = boost::program_options;

/**
 *  \brief Ensures correct output path
 *
 *  Checks if an output file exists, and if yes if it has an extension. In case
 *  it doesn't exist, `output_` is appended at the beginning of the input
 *  filename. If the output path does not have an extension, the type `.png` is
 *  appended at the end of the path.
 *
 *  \param[in] t_vm Arguments passed to the program
 *  \param[out] t_input Input path
 *  \param[out] t_output Output path
 */
void processFilenames(po::variables_map const &t_vm, path const &t_input,
                      path &t_output)
{
  if (!t_vm.count("output")) {
    t_output.replace_filename("output_"
                              + std::string{t_input.filename().string()});
  } else if (!t_output.has_extension()) {
    t_output.replace_extension(".png");
  }
```

```
}

/**
 * Parses the arguments given to the program, formats them and returns them as
 * a tuple. If `-h` or `--help` or a malformed argument is passed, then the
 * list of arguments and their comment will be displayed, and the program will
 * exit.
 *
 * \param[in] t_ac Number of arguments passed to the program
 * \param[in] t_av Arguments passed to the program
 * \return Tuple of path, path, int, int, int, int, int, bool and bool
 */
[[nodiscard]] auto parse_args(int t_ac, char **t_av) -> ParsedArgs
{
  ParsedArgs ret{};
  po::options_description desc("Allowed options");
  desc.add_options()("help,h", "Display this help message")(
      "input,i", po::value<path>(),
      "Input image")("output,o", po::value<path>(),
                     "Image output path (default: \"output_\" + input path)")(
      "iterations,n", po::value<int>(), "Number of iterations (default: 2000)")(
      "method,m", po::value<int>(), "Method number to be used (default: 1)")(
      "form,f", po::value<int>(), "Select shape (1:square, 2:triangle)")(
      "cols,c", po::value<int>(),
      "For method 5 only, number of columns the reference image should be "
      "divided into. If the value is equal to 0, then it will be assumed "
      "there will be as many rows as there are collumns. (default: 0)")(
      "rows,r", po::value<int>(),
      "For method 5 only, number of rows the reference image should be "
      "divided into. (default: 1)")(
      "submethod,S", po::value<int>(),
      "Sub-method that will be used to generate the individual tiles from "
      "method 5. (default: 1)")("size,s",
                                "Enables controlled size of the random shapes")(
      "verbose,v", "Enables verbosity");
  po::variables_map vm;
  po::store(po::parse_command_line(t_ac, t_av, desc), vm);
  po::notify(vm);
  if (vm.count("help") || !vm.count("input")) {
    std::cout << desc << "\n";
    std::exit(!vm.count("help"));
  }

  auto const input_path = vm["input"].as<path>();
  auto output_path
      = vm.count("output") ? vm["output"].as<path>() : input_path.filename();
  processFilenames(vm, input_path, output_path);

  ret.input_path  = input_path;
  ret.output_path = output_path;
  ret.iterations  = vm.count("iterations") ? vm["iterations"].as<int>()
                                           : DEFAULT_ITERATIONS;
  ret.method = vm.count("method") ? vm["method"].as<int>() : 1;
  switch (vm.count("form") ? vm["form"].as<int>() : 1) {
  case 2: ret.shape = Shape::ShapeType::Triangle; break;
  case 1: ret.shape = Shape::ShapeType::Square; break;
  default: ret.shape = Shape::ShapeType::Square; break;
```

```
89      }
90
91      ret.cols              = vm.count("cols") ? vm["cols"].as<int>() : 0;
92      ret.rows              = vm.count("rows") ? vm["rows"].as<int>() : 1;
93      ret.submethod         = vm.count("submethod") ? vm["submethod"].as<int>() : 1;
94      ret.controlled_size = vm.count("size");
95      ret.verbose           = vm.count("verbose");
96      return ret;
97  }
```

4.2 Images

4.2.1 Image de référence

Figure 1: Image de référence

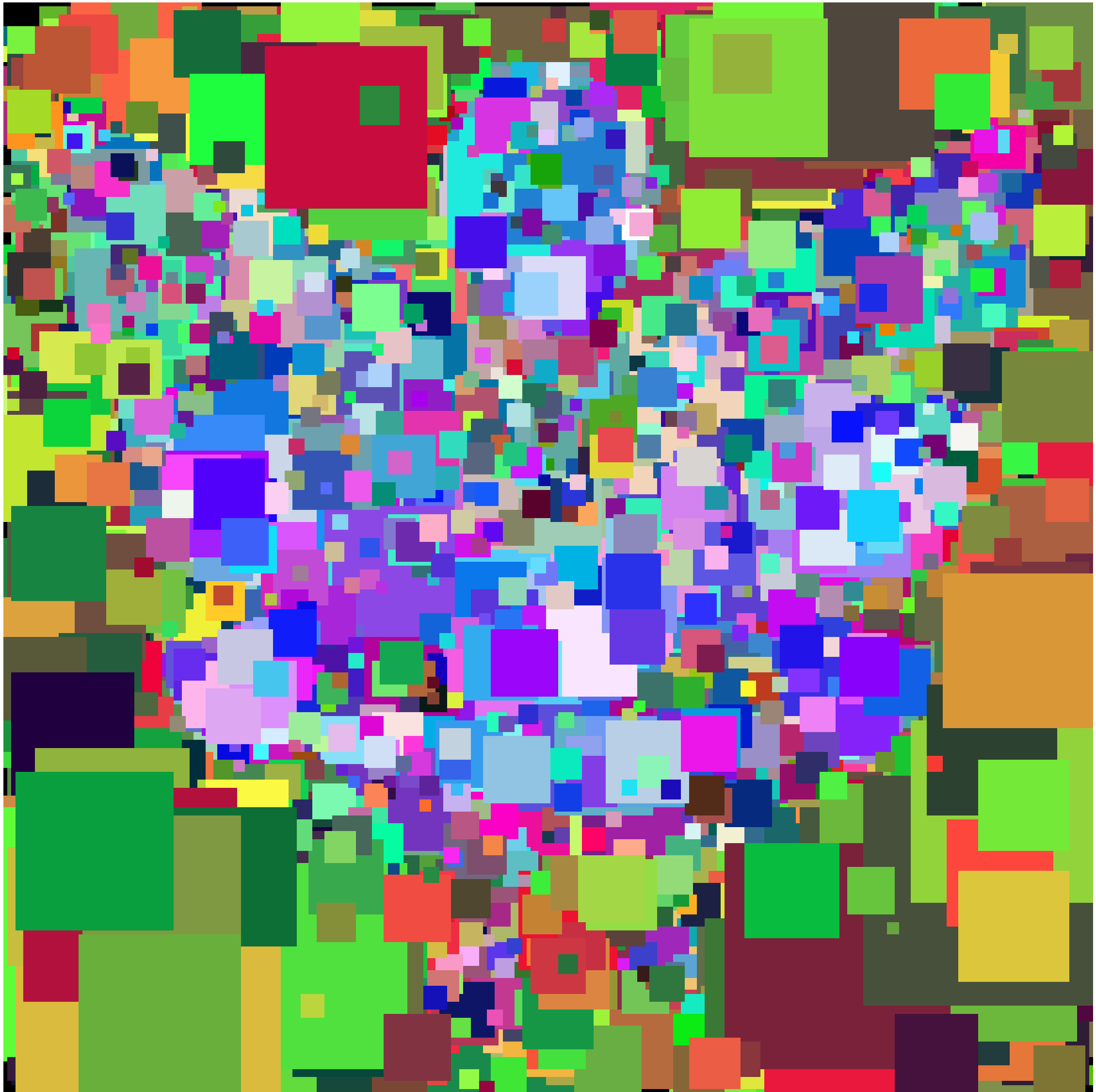

Figure 2: Méthode 1, carrés

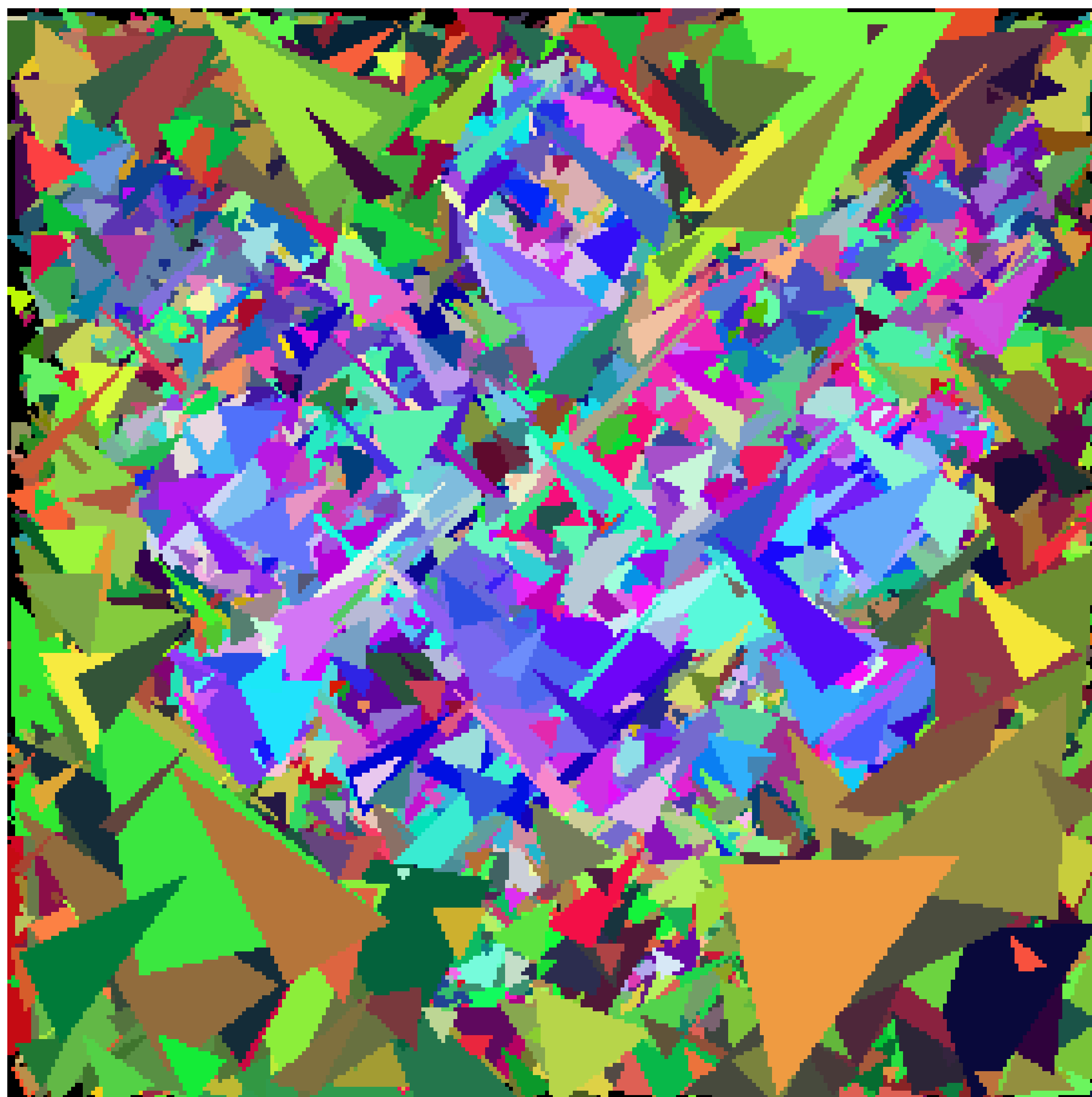

Figure 3: Méthode 1, triangles

Figure 4: Méthode 2, carrés

Figure 5: Méthode 2, triangles

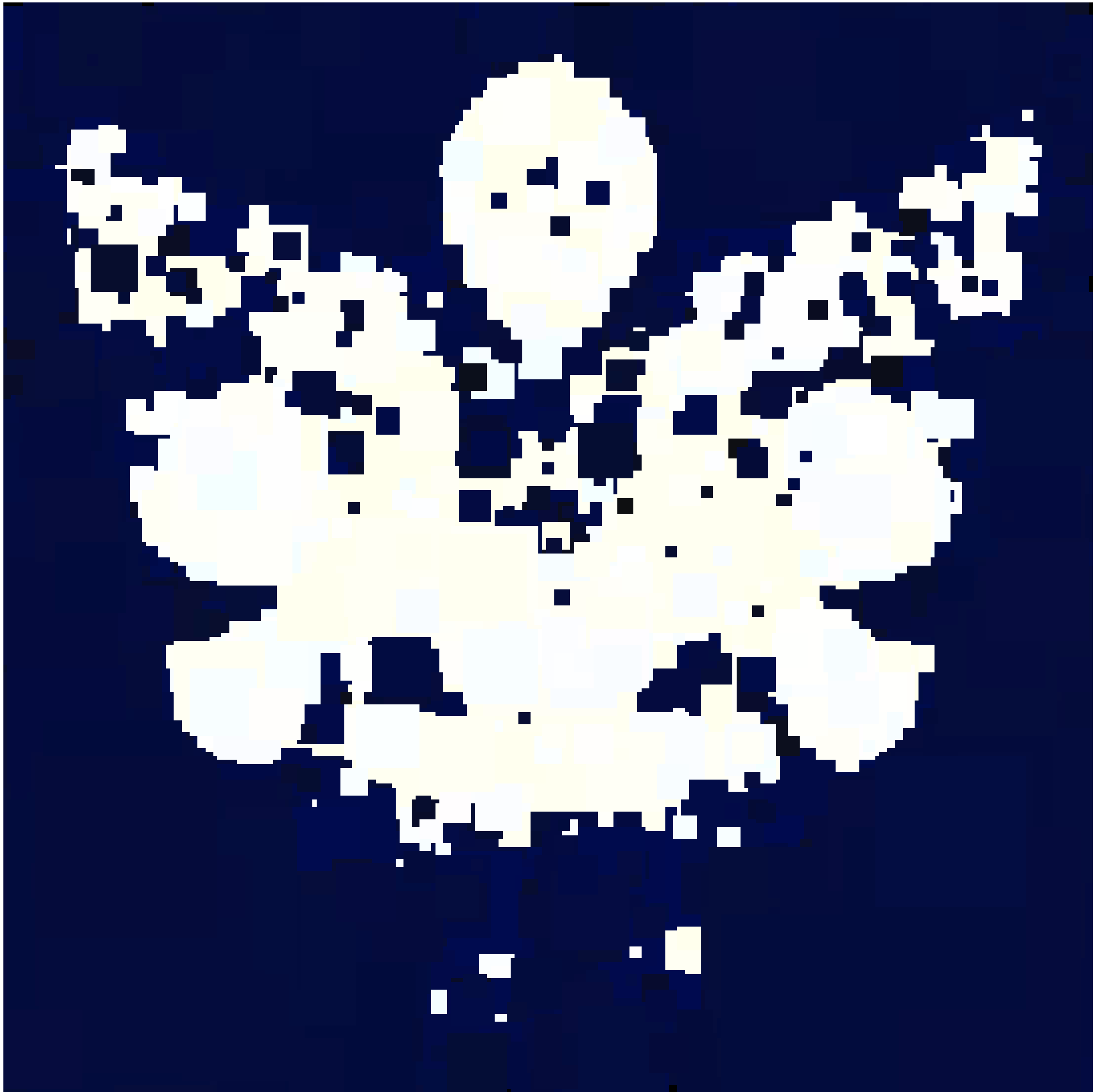

Figure 6: Méthode 4, carrés

Figure 7: Méthode 4, triangles

Figure 8: Méthode 5, sous-méthode 2, carrés

Figure 9: Méthode 5, sous-méthode 2, triangles

Figure 10: Méthode 5, sous-méthode 4, carrés

Figure 11: Méthode 5, sous-méthode 4, triangles