Le projet Mosaic

Artem Oboturov *
Ludwig Brummer †

le 21 Janvrier 2013

1 Description du projet

Dans le cadre du Master 2 Modélistaion Aléatoire à l'Université Paris VII Denis Diderot les étudiants doivent réaliser un projet de C++ en binôme pour montrer leur capacité de programmation. Cette année le cible du projet était la réalisation d'un photomosaïque, c'est-à-dire un programme qui reconstruit un image en remplaçant des parties de cet image par des autres images. Ce texte décrit le programme d'Artem Oboturov et Ludwig Brummer.

Dans ce qui suit on parlera de l'organisation du projet dans le chapitre 2 et donnera un mode d'emploi dans la section 3. Puis la structure de code est éxpliquée dans la partie 4 et quelques algorithmes sont discutés dans le chapitre 5. Enfin on montrera quelques résultats.

2 Organisation de travail sur projet

Un problème, que on a rencontré en début de travail sur le projet, c'était la nature distributive de l'équipe. Pour travailler effectivement à distance, on a trouvé un site specialisé en gestion de projets en ligne. Le projet a été organisé atour de site Github - le portal de travail collaborative sur les logiciels. Ce site garde le code source et donne accés vers ce code par système de contrôle de versions Git. Par ailleurs il montre tous les contributions de chaque collaborateur ligne par ligne dans le code et donne des diagrammes d'activité et productivié des collaborateurs.
Il y a un composant de gestion de projet permettant à créer des billets sur le taches de développement. De plus il est possible de faire des branches dans le projet pour travailler paralllement sur des mêmes fiches et les réunifier après.
On a créé un projet privé (non accessible par personnes non autorisées).

3 Instruction d'emploi

Dans le chapitre suivant le mode d'emploi du programme est présenté.

*oboturov@telecom-paristech.fr
†ludwig.brummer@mytum.de

3.1 La bibliothèque des carreaux

D'abord on a besoin d'une certaine quantité des images de lesquelles la programme construira la mosaïque. Pour un bon résultat on a besoin de cinquante images au moins dans le dossier "imageLibrary" dans le dossier du programme.

Tous ces images doivent avoir la même taille. Pour y arriver, on peut utiliser le programme "library_generator.exe" ou d'abord le commande "make library_generator" s'il ne se trouve pas déjà dans le dossier du programme. Le programme copiera tous les images dans le dossier "inputImages" dans le dans dossier "libraryImages" et réduira les tailles des images à un taille quadratique qui peut être spécifié dans la ligne de commande. Agrandir des images n'est pas possible avec ce programme.

Le programme peut être éxecuté avec deux different options dans la ligne de commande.

D'abord on peut spécefier la taille des images souhaité après la commande "--size" avec un nombre entier positif. La taille défaute est 100 pixels. Alors un éxecution exemplaire serait ".\library_generator --size 50".

Et puis il y a l'option "--help" qui répétera toute l'information de ces instructions.

Il faut faire attention : le programme ne va pas couper des parties des images mais seulement réduire la taille alors les proportions peuvent être changé. Ce façon de réduire a quelque avantages et disavantages mais pourrait facilement être changer par couper la partie d'image qui est "trop" et réduire après.

3.2 Le programme principal

Maintenant que nous avons une bibliothèque assez grand et de la même taille dans le dossier "libraryImages", on doit effacer tous les images dans le dossier "inputImages" sauf ceux qui sont destiné à devenir des mosaïques. Puis on peut exécuter la programme ".\m2mo_brummer_oboturov_project.exe". Là aussi il y a des options qui peuvent être ajouté dans la ligne de commande.

"--help" ou "-h" ouvrira plus ou moins les mêmes informations que ce paragraphe mais n'exécutera pas le mosaic maker.

Avec les options "--mse" ou "-a", "--meancolor" ou "-b" et "--mcmse" ou "-c" on peut choisir entre des mesures de divergence.

Avec l'option "--tilesize" et un nombre positif entier on peut choisir la taille des carreaux, c'est-à-dire les sections qui sont remplacer par des autres images.

Les options peuvent être écrit dans n'importe quel ordre. Les défautes sont la mesure de Monte Carlo et un taille de 20 pixels.

Le programme annoncera quand un mosaïque est fini. Ils sont sauvegardés dans le dossier "outputImages".

4 Le progiciel et son code source

De manière générale, la documentation pour chaque classe est réalisé en forme de commentaires des méthodes.

Dans notre travail on a largement réutilisé le projet fournit par Christian Konrad.

4.1 Réalisation des Images

Cette classe définit le concept d'Image.

Listing 1 – La classe Image

```
1  class Image
2  {
3  public:
4    typedef BlockIterator iterator;
5
6    // Defined in the project provided by C. Konrad
7    Image(string filename);
8    Image(img_size_t width, img_size_t height);
9    Image(Image &img);
10   ~Image();
11
12   // Image dimension accessor methods.
13   img_size_t get_width() const;
14   img_size_t get_height() const;
15
16   // Provide iterator over Blocks which is the Image is divided into.
17   iterator begin(const img_size_t height, const img_size_t width);
18   iterator end(const img_size_t height, const img_size_t width);
19
20   // Persist image from memory into a file.
21   void save(string filename);
22
23   // Method to flip Image horizontally.
24   void flip_horizontally();
25
26   // Accessor to a pixel at specified coordinate.
27   img_color_t& operator()
28     (img_coord_t x, img_coord_t y, img_color_layer_t layer) const;
29
30   // Scale this Image into an Image of different size (which MUST be defined
31   // in advance for a new Image).
32   Image& scale_to(Image& downscaled) const;
33 };
```

4.2 Découpage des Images en Blocs

Chaque Image peut-être représentée comme une séquence des blocs. Bloc est un rectangle correspondant à une partie de l'Image et est défini par ces coordonnées de gauche-haute et de droit-basse. C'est un concept naturel pour définir des algorithmes sur les parties matricielles des Images. Une nuance de réalisation : le bloc ne alloue pas de mémoire pour stocker des pixels d'Image auxquelles il correspond.

Listing 2 – La classe Block

```
1  class Block
```

```
2  {
3  public:
4    Block(const Image* image, const img_coord_t top, const img_coord_t left,
5      const img_size_t height, const img_size_t width);
6
7    // Accessors to Block dimensions.
8    img_coord_t get_left() const { return left; }
9    img_coord_t get_top() const { return top; }
10   img_size_t get_height() const { return height; }
11   img_size_t get_width() const { return width; }
12
13   // Get pointer to Image this Block references to.
14   const Image* get_img() const { return img; }
15
16   // Copy content from another Block into this Block.
17   void copy_content(Block& to_copy);
18 };
```

On peux parcourir des Blocs d'Image avec un iterator BlockIterator. On a conceptualisé, que l'utilisateur de la classe Image va la parcourir de façon défini par le Forward-Iterator.

Listing 3 – La classe BlockIterator

```
1  class BlockIterator
2  {
3  public:
4    BlockIterator(Image* image, const img_size_t block_height,
5      const img_size_t block_width);
6
7    // Reference to an Instance representing the End of a BlockIterator.
8    BlockIterator& end();
9
10   // Standard methods for a Forward-Iterator.
11   Block& operator*();
12   Block* operator->();
13   BlockIterator& operator++();
14   bool operator==(const BlockIterator&) const;
15   bool operator!=(const BlockIterator&) const;
16 };
```

4.3 Création de mosaïque

La mesure de divergence est une classe polymorphe. Donc, chaque mesure est défini dans une classe concret, qui réalise une distance de divergence entre deux blocs.

Listing 4 – La mesure de divergence

```
1  class DivergenceMeasure
2  {
3  public:
```

```
4     // Compare blocks with the Mean Square Error proximity measure
5     // and returns a value in [0,1].
6     virtual float compute(const Block& lhs, const Block& rhs) const = 0;
7     virtual ~DivergenceMeasure() = 0;
8   };
```

Pour augmenter la performance de parcours de la bibliothèque des images, on a fait un cache de ces images en fixant leur taille à BLOCK_SZ.

Listing 5 – L'algorothme de création de mosaïque

```
1   ImageLibrary& archive;
2   Image* target_image;
3   DivergenceMeasure* divergence;
4   int BLOCK_SZ;
5
6   Image::iterator begin = target_image->begin(BLOCK_SZ, BLOCK_SZ),
7     end = target_image->end(BLOCK_SZ, BLOCK_SZ);
8   // Scale down images from library.
9   Image library(archive.size() * BLOCK_SZ, BLOCK_SZ),
10    downscaled_img(BLOCK_SZ, BLOCK_SZ);
11  size_t cnt = 0;
12  for (ImageLibrary::storage::const_iterator image_in_lib_it = archive.begin();
13    image_in_lib_it != archive.end(); ++image_in_lib_it, ++cnt)
14  {
15    (*image_in_lib_it)->scale_to(downscaled_img);
16    Block downscaled_block(&downscaled_img, 0, 0, BLOCK_SZ, BLOCK_SZ);
17    Block(&library, 0, cnt * BLOCK_SZ, BLOCK_SZ, BLOCK_SZ)
18      .copy_content(downscaled_block);
19  }
20  // Create mosaic.
21  for (Image::iterator blockIt = begin; blockIt != end; ++blockIt)
22  {
23    float prox = 0.f;
24    Image::iterator begin = library.begin(BLOCK_SZ, BLOCK_SZ),
25        end = library.end(BLOCK_SZ, BLOCK_SZ);
26    Block chosen_tile(NULL, 0, 0, BLOCK_SZ, BLOCK_SZ);
27    for (Image::iterator libIt = begin; libIt != end; ++libIt)
28    {
29      float div_res = (*divergence).compute(*libIt, *blockIt);
30      if (div_res > prox)
31      {
32        prox = div_res;
33        chosen_tile = *libIt;
34      }
35    }
36    (*blockIt).copy_content(chosen_tile);
37    cout << "*";
38    cout.flush();
39  }
```

4.4 Exécution avec des options de ligne des commandes

Le progiciel soutien les paramètres de ligne des commandes. Cette fonctionnalité est résalisé avec la bibliothèque 'Getopt' de stdlibc.

4.5 Le library_generator

On a créé une utilité pour convertir les images arbitraires en format JPEG vers la taille de $100px \times 100px$. Pour la compiler il faut appeler 'make library_generator'. Les instructions d'utilisations sont accessible, si quelqu'un l'appelle avec un paramètre '-h'.

5 Algorithmique

5.1 L'algorithme de souséchantillonnage linéaire

L'algorithme de souséchantillonnage linéaire ou de scaling est utilisé pour changer tout les images destiné pour la bibliothque à une taille uniforme. Ici on doit se rendre compte qu'il est seulement capable de diminuer la taille des images car c'est l'usage réaliste pendant un construction d'un mosaïque. Pendant un agrandissement il y aura des fautes.

L'algorithme se trouve dans la fonction Image& Image : :scale_to(Image& downscaled) dans les fichiers image.h et image.cpp. Alors il fait partie de la classe Image comme on peut voir dans le code. La fonction reçoit comme paramétre la reference d'une image "downscaled" qui a la taille à laquelle on veut changer l'image originale.

On peut trouver le code dans le Listing 6.

Listing 6 – L'algorithme de souséchantillonnage linéaire

```
1  img_size_t  desiredw = downscaled.get_width(),
2        desiredh = downscaled.get_height();
3
4  float paceh = height/desiredh;
5  float pacew = width/desiredw;
6  float norm = paceh*pacew;
7  int y = 0;// pixel positions for big image
8  int x = 0;
9  for(int i = 0; i < desiredw; i++)
10 {
11   x=i*pacew;
12   for(int j = 0; j < desiredh; j++)
13   {
14   y = j*paceh;
15     for(img_color_layer_t layer = 0; layer < LAYER_CNT; layer++)
16     {
17       float color = 0;
18       for (int cnth = 0; cnth < int(paceh); cnth++)
19       {
20         for (int cntw = 0; cntw < int(pacew); cntw++)
21           color += (float)(*this)(x + cntw, y + cnth, layer);
```

```
22          }
23          downscaled(i, j, layer) = color/norm;
24        }
25      }
26  }
```

Grosso modo l'algorithme prend un carré des plusieurs pixels, calcule la moyenne des couleurs et met un pixel avec le couleur moyen à la place de ces pixels. C'est illustré par Figure 1. Comme ça, la taille d'un image est réduit. La prise de la moyenne est fait séparément pour chaque couleur (boucle l.15-24) en additionnant tout les intensités de ce couleur dans pixels du carré choisi (l. 21) et les divisant par la taille du carré (norm dans le code, l. 23).

Cette approche de réduction est plutôt évidente. La difficulté est de choisir les bonnes tailles pour les carrés à réduire. C'est réalisé par les lignes 4-14 et illustré par Figure 1. Ici on utilise la façon comment C++ sauvegarde et transforme des variables. Des pas, alors les longueurs des côtes du carré à réduire, sont définis comme floats aux lignes 4 et 5 car la proportion entre la hauteur respectivement la largeur d'un image et la hauteur souhaité respectivement la largeur souhaité n'est pas toujours un entier. Mais les carrés à réduire ne peuvent avoir qu'un longueur entier. Alors on définie la position à gauche et en haut dans le carré à réduire comme couplet des entiers (x, y) mais fait le calcul de la position avec des floats. Comme ça, il n'y a pas des erreurs dans le calcul, mais les positions sont automatiquement transformé en entiers et les tailles des carrés vacillent entre les deux entiers qui entourent les pas.

5.2 Les mesures de divergence

Les mesures de divergence comparent des parties de l'image originale, alors les "blocks" dans le code, avec des futurs carreaux ou "tiles" du mosaïque, alors les images de la bibliothèque. Chaque mesure retourne un nombre entre 0 et 1. 0 indique très peu ressemblance entre le block et le tile et 1 beaucoup de ressemblance. Alors le tile avec la plus grande nombre remplace le bloc.

Chaque mesure a un diffrent notion de ressemblance qui sera éxpliqué dans les prochains séctions.

5.2.1 La mesure d'erreur des moindres carrés

On peut voir les pixel d'un image comme vecteur 5-dimensionnel. Les premiéres deux dimensions sont la position du pixel dans l'image. La reste contient un code pour les couleurs. Chaque dimension représente un des couleurs rouge, vert et bleu (Codage RVB) et des valeurs entre 0 et 255 qui représente l'intensité de ce couleur. Alors un pixel avec des couleurs (0,0,255) est bleu ; (0,0,0) est noir et (255,255,255) est blanc.

La formule suivante donne la mesure de erreur des moindres carrés. C'est la difference de tout les pixels d'un block avec tout les pixels d'un tile dans la L^2-norme renormalisée par la taille $X \cdot Y$ du block (ou du tile, c'est la même) et $C = 3$ pour les couleurs.

$$err = \frac{1}{X \cdot Y \cdot C} \sum_{x,y,c} \left(\frac{block(x, y, r, v, b) - tile_{image}(x, y, r, v, b)}{255} \right)^2 \qquad (1)$$

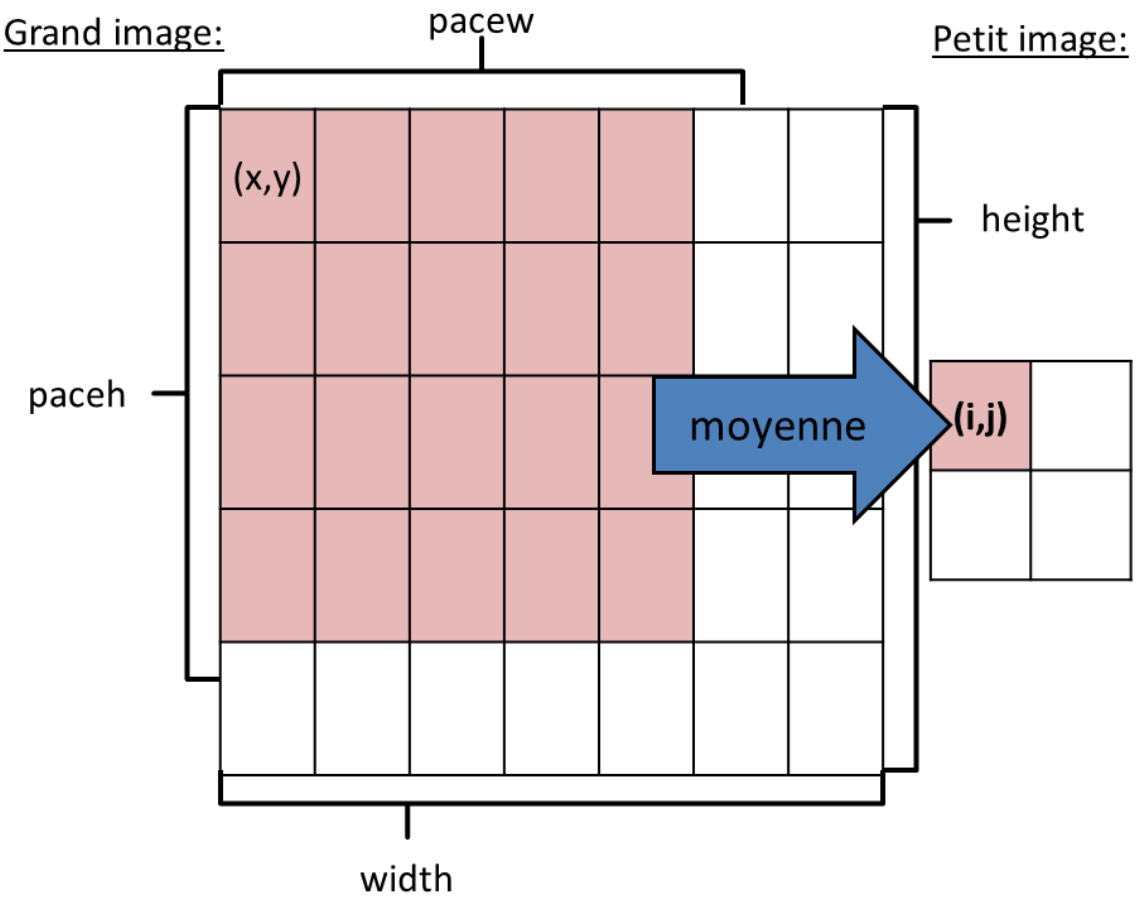

FIGURE 1 – Un illustration du choix des bonnes tailles pour les carrés à réduire

TABLE 1 – Temps d'éxecution avec des mesures differents

Mesure	guybrush.jpg	IMG_1293_small.JPG	IMG_1331.JPG
Erreur des moindres carrés	1 :13 min	0 :38 min	0 :39 min
Couleur moyenne	0 :58 min	0 :31 min	0 :29 min
Monte Carlo	0 :47 min	0 :25 min	0 :25 min

5.2.2 La mesure de couleurs moyens

La mesure de couleurs moyens est en fait très similaire à la mesure d'erreur des moindres carrés mais à la place de la L^2-norme, on prend la L^1-norme :

$$err = \frac{1}{X \cdot Y \cdot C} \sum_{x,y,c} \left| \frac{block(x,y,r,v,b) - tile_{image}(x,y,r,v,b)}{255} \right| \tag{2}$$

5.2.3 La mesure de Monte Carlo

La mesure de Monte Carlo utilise presque la même fonctionne que la mesure d'erreur des moindres carrés. La seule difference est que il ne compare pas tous les vecteurs mais décide avec une fonctionne aléatoire quelle pixels sont comparés au niveau de quelle couleur.

Cette mesure était implementé en ésperant que il est plus vite que la mesure d'erreur des moindres carrés en donnant encore un résultat acceptable.

Ensuite la mode de fonctionnement de la mesure : Soient $(Z_{x,y,c})$ une suite des variables aléatoires i.i.d. de la loi Bernoulli avec $p = 0,25$. $c \in \{r,v,b\}$ indique le niveau de couleur. Alors l'erreur est :

$$err = \frac{1}{\sum_{x,y,c} Z_{x,y,c}} \sum_{x,y,c} \left(\frac{block(x,y,c) - tile_{image}(x,y,c)}{255} \right)^2 \cdot Z_{x,y,c} \tag{3}$$

6 Resultats

Enfin les résultats avec des mesures de divergence differents seront comparé dans les disciplines vitesse et qualité. On utilisera une bibliothèque de 943 images pour transformer des images de test guybrush.jpg, IMG_1293_small.JPG, IMG_1331.JPG sur le même ordinateur avec la défaute tilesize = 20. Ils ont un taille de 800 x 1129, 800 x 600 et 800 x 600. Les durées de constuction peut être vu dans la Table 1. Alors la mesure de Monte Carlo est la plus vite comme ésperé et l'erreur des moindres carrés est le plus lente. C'est-à-dire que l'on pourrait implémenter une mesure de Monte Carlo Couleur moyenne qui est encore plus vite. De plus, le paramètre $p = 0,25$ n'est probablement pas encore optimal. On peut le réduire probablement encore sans grande perte de qualité.

Au niveau de la qualité une décision objectif n'est pas aussi simple. On peux voir des images de test dans les Figures 2, 3 et 4.

La différence entre les Figures 2 et 4 n'est pas très grand, alors on peut probablement réduire p encore plus et on préfère la mesure de Monte Carlo, car il est plus vite. La difference entre Figure 3 et les autres n'est pas aussi facile à voir. La mesure des erreurs de moindres carrés préfère apparemment des tiles plutôt monochromatique tandis que la mesure de couleurs moyens utilise des

tiles avec plus de contraste. Mais ce n'est pas facile à dire qui est mieux et reste probablement une question de situation et goût.

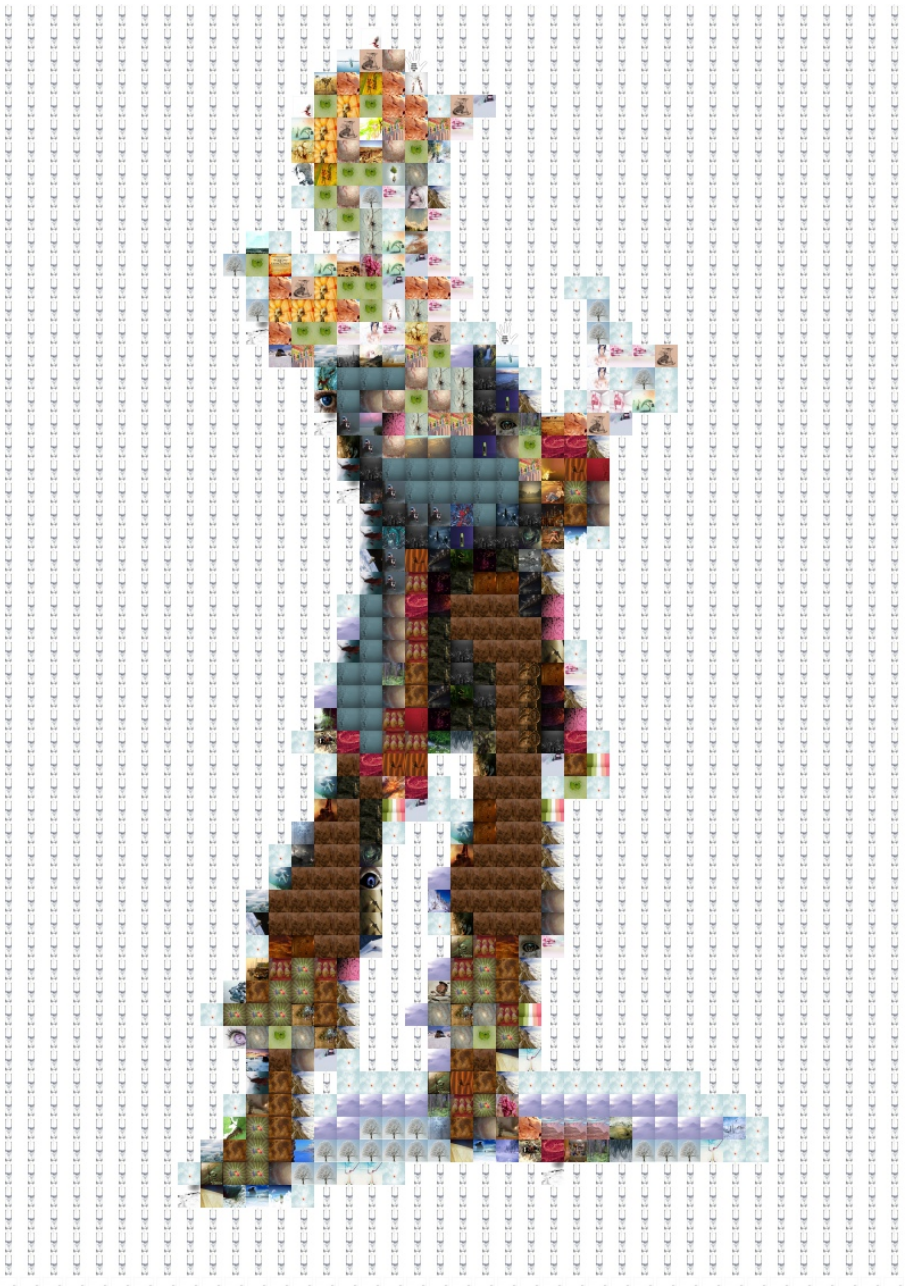

FIGURE 2 – Le résultat de l'image de test guybrush.jpg avec la mesure des moindres carrés

FIGURE 3 – Le résultat de l'image de test guybrush.jpg avec la mesure de couleurs moyens

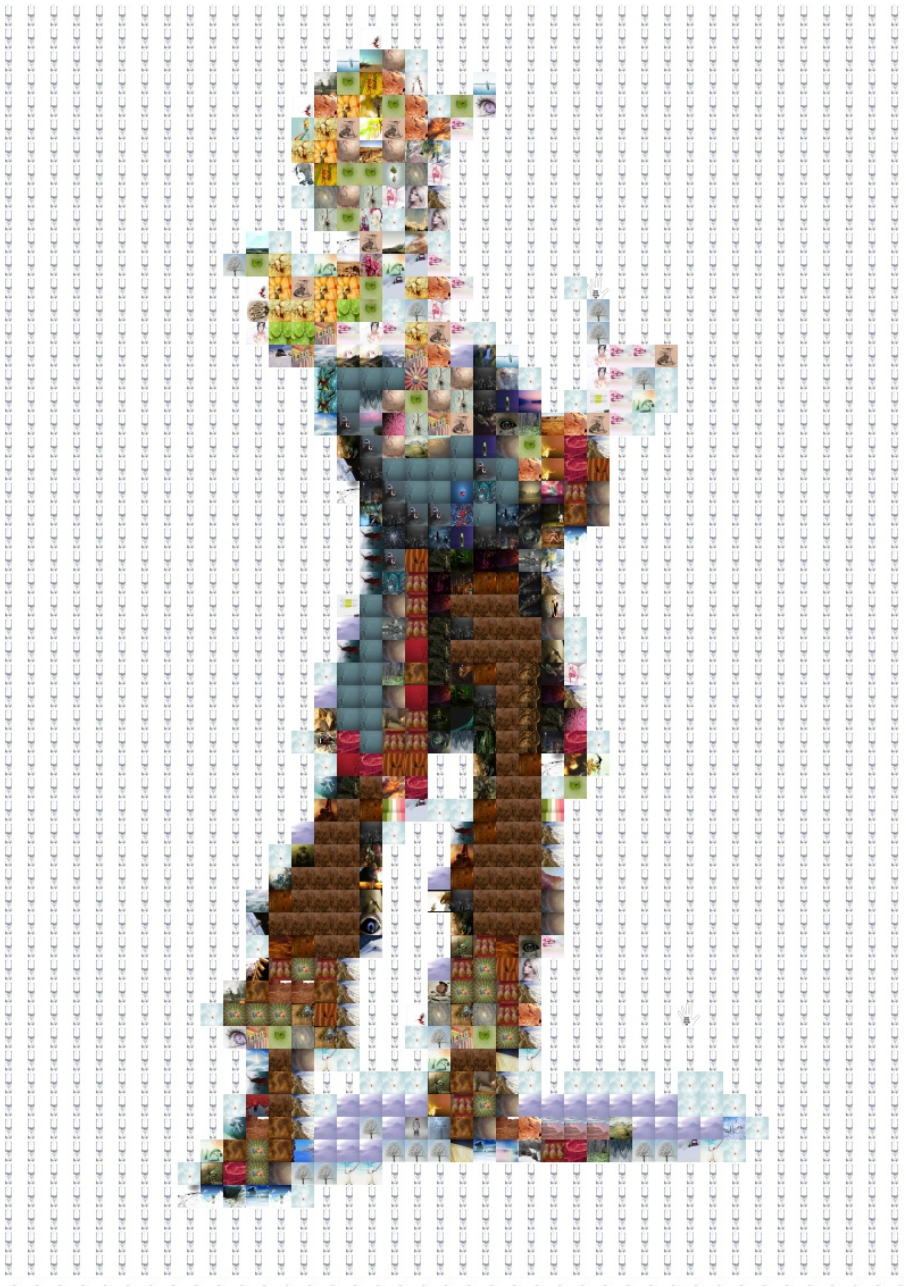

FIGURE 4 – Le résultat de l'image de test guybrush.jpg avec la mesure de Monte Carlo