GNU LINUX MAGAZINE / FRANCE

France Métro : 5,95€ - DOM 6,45€ - BEL : 6,50€ - LUX : 6,50€ - PORT. CONT. : 6,50€ - CH : 12FS - CAN : 11$ - MAR : 65DH

L 19275 - 83 - F: 5,95 €

▶MAI ▶2006 ▶NUMÉRO 83

Greylist
Eliminez le SPAM à la racine

08 ▶

Reject 554 ! Voilà la meilleure réponse qui convient lorsqu'on tente de vous envoyer un pourriel. Eliminer la plupart des risques au niveau SMTP améliore les performances et vous protège des attaques DoS.

▶ALGORITHME ET CODE

Synthèse de textures

Découvrez la mécanique interne du greffon Texturize expliquée par son auteur

80 ▶

Administration et développement sur systèmes UNIX

→ Texturize : synthèse intelligente de textures

Manu Cornet

EN DEUX MOTS **Le plugin Texturize pour GIMP génère automatiquement de larges textures à partir d'un petit échantillon. Il s'appuie sur un algorithme fondé sur les graphes, des objets abstraits très simples à comprendre et plutôt amusants à manipuler.**

Cet article explique le fonctionnement de Texturize et expose très simplement les concepts sous-jacents, afin de comprendre comment fonctionne le processus de synthèse de textures. Nous allons commencer par voir ce que sont les graphes et comment ils se comportent, avant de les appliquer aux images. Cependant, pas besoin de ressortir vos cours de maths chéris : aucune connaissance préalable ni en mathématiques, ni sur les graphes, n'est nécessaire.

1. Ça sert à quoi ?

1.1 Le problème

Supposons que vous ayez besoin, pour votre site web, d'une image de fond qui représente des fraises (après tout, pourquoi pas ?). Malheureusement, vous n'avez pas de fraises à photographier chez vous (ce n'est pas la saison, celles du supermarché n'ont pas une bonne tête), et la seule belle image que vous ayez trouvée sur Internet est celle-ci :

Fig. 1 : L'échantillon de texture

Les fruits sont bien rouges, mais l'image n'est pas assez grande pour faire un arrière-plan...

Vous pourriez redimensionner l'image à la taille de l'écran, mais le fond obtenu serait totalement flou. Vous pourriez aussi répéter l'image plusieurs fois, mais le résultat serait désastreux : les « frontières » entre les différentes copies seraient nettement visibles.

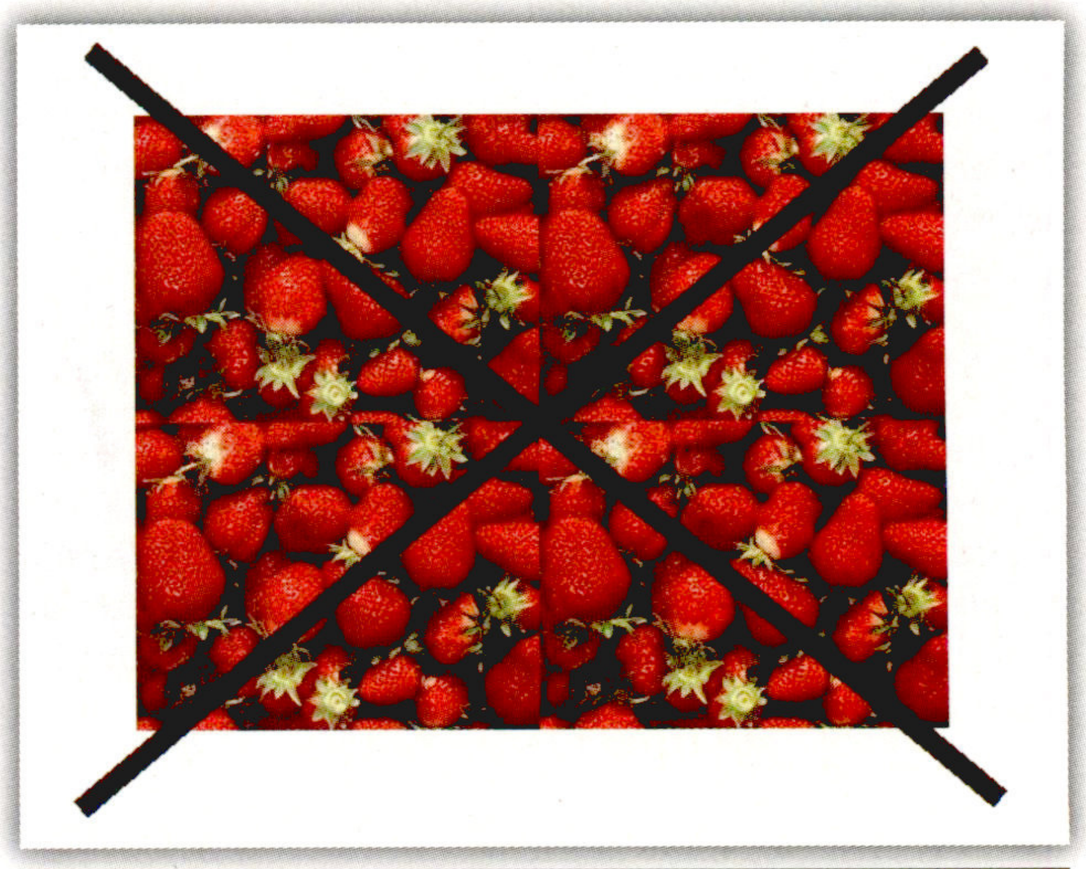

Fig. 2 : Simple répétition de l'image

1.2 La solution

Vous vous en doutez, c'est là que le *plugin* entre en jeu. Ouvrez l'image avec GIMP et passez un petit coup de Texturize (nous verrons comment l'installer et l'utiliser à la fin de cet article), choisissez la taille souhaitée, et vous obtiendrez le résultat suivant :

Fig. 3 : Résultat de la synthèse de texture

Texturize fonctionne à merveille sur des images de toutes sortes, qu'elles soient relativement complexes (des pingouins dans *Linux Mag*, c'est la moindre des choses !)

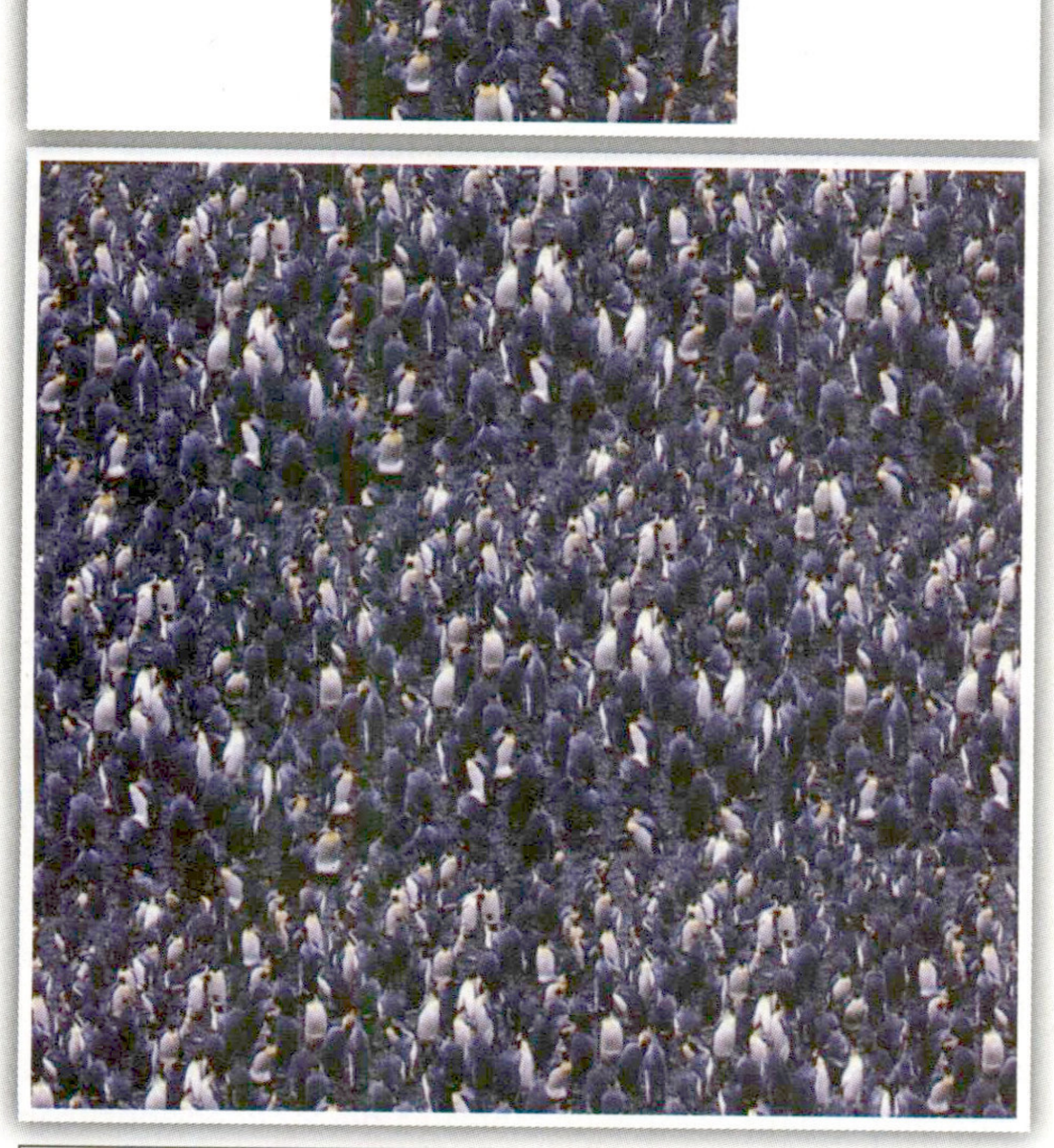

Fig. 4 : Pingouins, échantillon et résultat

ou bien plutôt périodiques :

Fig. 5 : Clavier, échantillon et résultat

(d'autres exemples sont disponibles sur la page du plugin, voir la section « Liens »).

Voyons maintenant comment le programme s'y prend pour générer ces textures, et posons pour cela quelques bases de théorie des graphes.

2. Les graphes

2.1 Qu'est-ce donc ?

Un graphe est une structure abstraite très simple, composée de seulement deux types d'objets :

- des nœuds (représentés par de petits cercles) ;
- des arêtes (représentées par des traits) qui relient ces nœuds.

Voici un exemple :

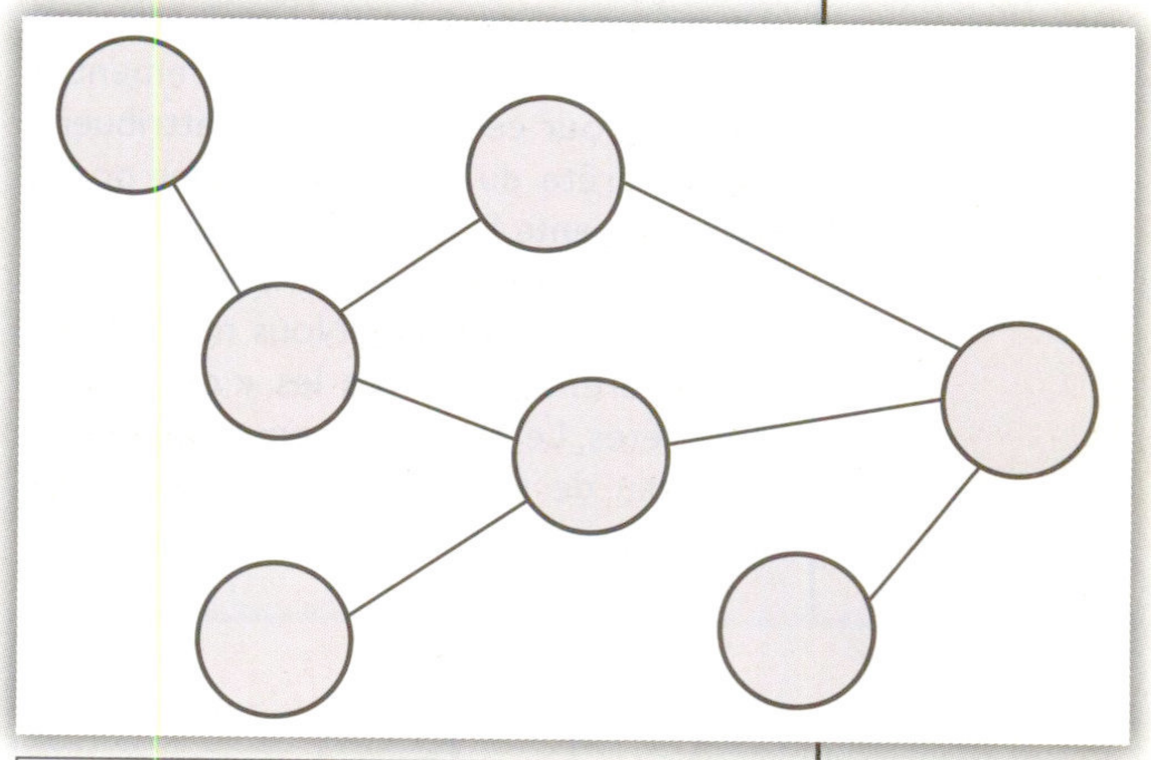

Fig. 6 : Un graphe

Les graphes constituent un moyen très simple de représenter un réseau de relations entre des objets. C'est, de fait, un outil extrêmement utilisé dans de nombreuses disciplines, de l'informatique des réseaux (relations entre des ordinateurs connectés entre eux) jusqu'à la

sociologie (liens sociaux entre les individus), en passant par la biologie (réseaux relationnels de gènes ou de protéines).

2.2 Les réseaux de transport

2.2.1 De l'eau pour Lyon

Cherchons dès maintenant à représenter une situation concrète à l'aide d'un graphe, de la façon suivante : supposons que l'on cherche à résoudre un problème de transport d'eau entre plusieurs villes françaises. Chaque nœud est une ville, et une arête reliant deux villes représente un long tuyau destiné à transporter l'eau entre ces villes. Dans la réalité, les différents tuyaux n'ont pas le même diamètre (certains peuvent transporter plus d'eau par seconde que les autres). Or, dans notre graphe, toutes les arêtes sont équivalentes (ce sont de simples traits) : il faut donc perfectionner légèrement le modèle. Pour cela, on choisit d'attribuer à chaque arête du graphe un entier positif, qui représente le débit d'eau que le tuyau peut transporter (ou bien son diamètre, cela revient au même). Nous nommerons ces différents nombres les « capacités » des arêtes. Le graphe obtenu s'appelle un « réseau de transport », et en voici un exemple simple :

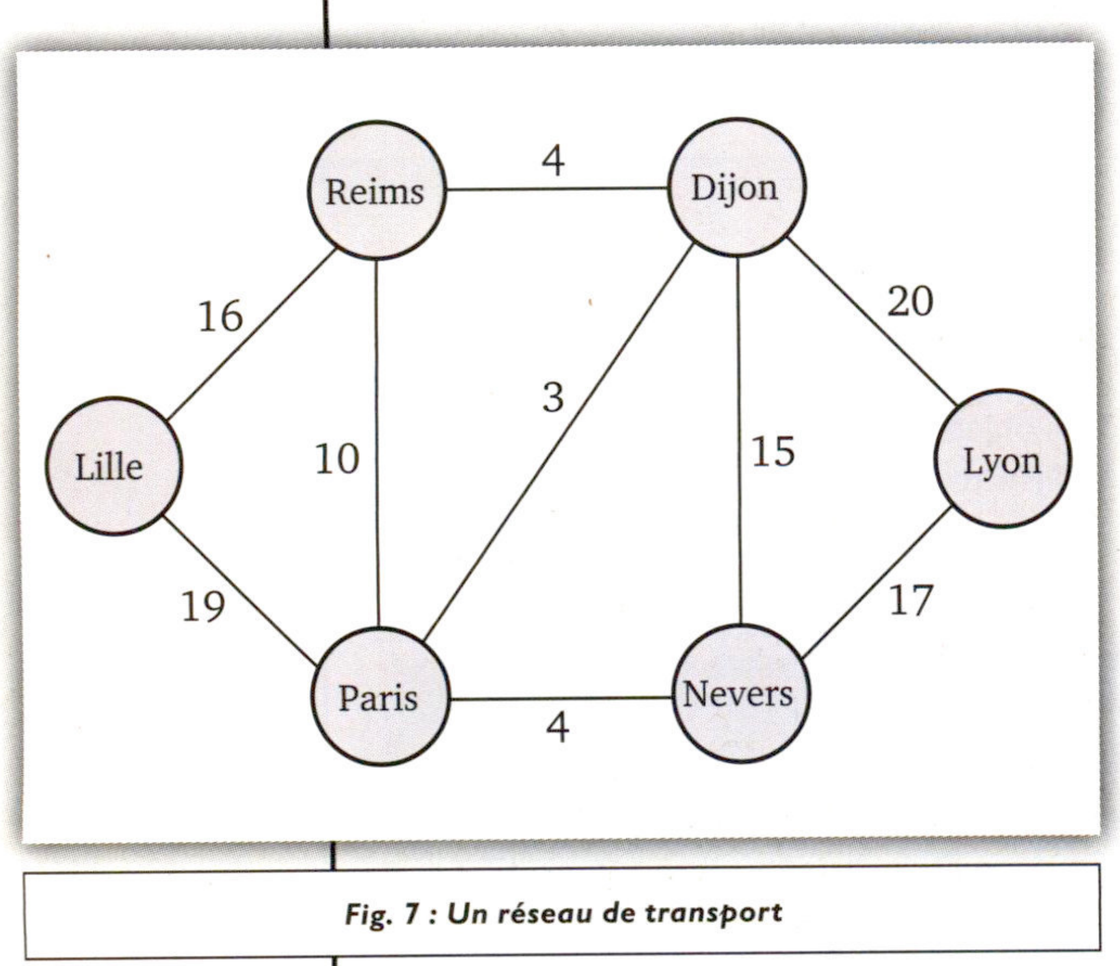

Fig. 7 : Un réseau de transport

2.2.2 Coder les graphes

Il n'est pas possible, dans le cadre de cet article, de commenter l'ensemble du code du plugin Texturize, mais il a semblé judicieux d'exposer rapidement le dilemme qui se pose immédiatement lorsqu'on cherche à mettre en pratique (dans un programme) la théorie des graphes. La section qui suit n'est pas nécessaire pour comprendre globalement le fonctionnement du plugin : si vous ne souhaitez pas vous soucier de l'implémentation, vous pouvez passer directement à la section 2.2.3.

Il existe deux façons différentes de coder les graphes dans des structures informatiques (en C), chacune comportant ses propres avantages.

Matrice d'adjacence

La façon la plus immédiate est de stocker toutes les informations d'un graphe dans un grand tableau à double entrée (une matrice carrée), dit « matrice d'adjacence ». Par exemple, pour un graphe dit « non pondéré » (aucune capacité n'est attribuée aux arêtes, comme sur la figure 6), on commence par numéroter les nœuds, puis on considère le croisement de la ligne ι (correspondant au nœud ι) et de la colonne φ (correspondant au nœud φ), que nous appellerons la case (ι, φ). Dans cette case du tableau, on inscrit 1 si les deux nœuds sont connectés par une arête, et 0 dans le cas contraire.

Remarquons d'ores et déjà que le tableau contient alors chaque information en double : la case (ι, φ) doit contenir la même information que la case (φ, ι). Ceci est dû au fait que notre modèle de graphe est « non dirigé » : dans un graphe dirigé (modèle un peu plus sophistiqué), on trace non plus de simples traits, mais des flèches, et ainsi il peut exister une arête de ι à φ (et dirigée dans ce sens), mais pas forcément de φ à ι.

Dans le cas d'un graphe pondéré (figure 7), il suffit de remplacer 1 par la capacité de l'arête lorsqu'elle existe.

Ainsi, un graphe représenté par une matrice d'adjacence se code simplement comme un tableau à deux dimensions (par exemple avec 10 nœuds et des capacités entières) :

```
#include <malloc.h>

int **graphe;
int   nb_noeuds = 10, i, j;

/* Allocation mémoire */
graphe = (int **) malloc (nb_noeuds * nb_noeuds * sizeof (int));

/* Initialisation */
for (i = 0; i < nb_noeuds; i++) {
  for (j = 0; j < nb_noeuds; j++) {
    graphe[i][j] = 0;
  }
}
```

Ajouter une arête de capacité χ entre les sommets ι et φ dans un tel graphe est donc immédiat :

```
graphe[i][j] = c;
```

(Il faut remplacer χ par 0 pour supprimer cette arête). De même, pour savoir si une arête existe ou récupérer sa capacité, un test sur le bon élément du tableau suffit.

Liste chaînée

La représentation en matrice d'adjacence est extrêmement simple, mais comporte de sérieuses limitations. Nous verrons un peu plus tard que pour les images, il nous faudra un nœud par pixel : ainsi, pour une image de 100 pixels de côté, 10 000 nœuds sont nécessaires, et il faudrait donc stocker 100 000 000 entiers dans une représentation en matrice d'adjacence. C'est beaucoup pour une image si petite ! En outre, si le graphe comporte assez peu d'arêtes, la majorité de

ces 100 000 000 entiers sont égaux à zéro. C'est clairement du gaspillage de mémoire.

Pour résoudre ce problème, on utilise une représentation en liste chaînée (c'est celle qui est utilisée dans Texturize, avec quelques raffinements) : le graphe est un tableau (à une seule dimension) de nœuds, et chaque nœud comporte un pointeur vers la liste de ses arêtes.

```
typedef struct Arete {
  int    voisin;     /* Numéro du noeud auquel l'arête est reliée
*/
  int    capacite;
  Arete *suivante;   /* Arête suivante dans la liste */
} Arete;

typedef struct Noeud {
  Arete *aretes;     /* Liste des arêtes */
  char  *nom;        /* Nom du noeud, éventuellement */
} Noeud;
```

L'initialisation d'un graphe de 10 nœuds prendrait alors la forme suivante :

```
#include <malloc.h>

Noeud *graphe;
int    nb_noeuds = 10, i;

/* Allocation mémoire */
graphe = (Noeud *) malloc (nb_noeuds * sizeof (Noeud));

/* Initialisation des liens */
for (i = 0; i < nb_noeuds; i++) {
  graphe[i]->aretes = NULL;
}
```

Dans ce type de structure, ajouter une arête de capacité χ de ι vers φ ou déterminer la capacité de l'arête reliant ι à φ est un peu plus long :

```
void ajout_arete (int i, int j, int c) {
  Arete *nouvelle_arete, *arete_courante;

  nouvelle_arete = (Arete *) malloc (sizeof (Arete));
  nouvelle_arete->capacite = c;
  nouvelle_arete->voisin = j;

  /* Si c'est la première arête de ce noeud */
  if (!(graphe[i]->aretes)) {
    nouvelle_arete->suivante = NULL;
    graphe[i]->arete = nouvelle_arete;
  } else {
    /* Si ce n'est pas la première, on l'ajoute au début de la liste */
    arete_courante->suivante = graphe[i]->aretes;
    graphe[i]->aretes = arete_courante;
  }
}
int poids_arete (int i, int j) {
  Arete *arete_courante;

  for (arete_courante = graphe[i]->aretes; arete_courante;
       arete_courante = arete_courante->suivante) {
    if (arete_courante->voisin == i)
      return arete_courante->capacite;
  }
  return 0;
}
```

Cette méthode a l'avantage d'utiliser la mémoire de façon beaucoup plus raisonnable. En revanche, le gain en mémoire se paye sur le temps d'exécution : pour tester la présence d'une arête ou déterminer sa capacité, il faut parcourir les différents maillons de la liste jusqu'à atteindre l'élément intéressant, alors qu'un simple test sur un élément du tableau suffisait dans le cas de la matrice d'adjacence.

2.2.3 Maximiser le flot

Revenons maintenant à notre problème de la figure 7 : transporter l'eau. On suppose d'une part que l'une de ces villes (par exemple, Lille) est la seule qui produise de l'eau, et on l'appelle la « source » ; d'autre part, une seconde ville (par exemple, Lyon) manque cruellement d'eau, c'est le « puits ».

Le problème consiste donc à transporter l'eau depuis la source vers le puits, à travers les tuyaux disponibles, en utilisant leur capacité de façon optimale, afin de pouvoir faire passer le maximum d'eau par seconde. Les autres villes servent uniquement de relais et ne produisent ni ne consomment d'eau.

Le problème n'est pas aussi évident qu'il en a l'air : par exemple, comme le tuyau entre Lille et Reims a une capacité de 16 unités d'eau par seconde, on aurait envie d'envoyer effectivement un débit de 16 dans ce tuyau.

Mais le tuyau suivant, reliant Reims à Dijon, n'a qu'une capacité de 4 unités par seconde, et ne pourra pas transporter toute cette eau.

On peut alors envisager de faire un détour par Paris, etc. : il faut dans tous les cas un peu de logique et quelques petits calculs pour arriver à un résultat optimal, qui indiquera précisément combien d'eau on peut effectivement faire passer dans chaque tuyau.

Vous vous demandez peut-être ce que viennent faire des tuyaux pleins d'eau dans un processus de synthèse d'images ! Mais non, il ne manque pas une page à votre magazine, les images reviendront très bientôt.

Le problème ci-dessus est connu sous le nom de « calcul du flot maximal », et il existe plusieurs algorithmes pour le résoudre.

Nous ne les détaillerons pas : ce qui est important, c'est qu'étant donné un réseau de transport (un graphe, des capacités, un puits, une source), il existe des programmes qui calculent très rapidement une solution optimale (qui n'est d'ailleurs pas unique), par exemple la suivante :

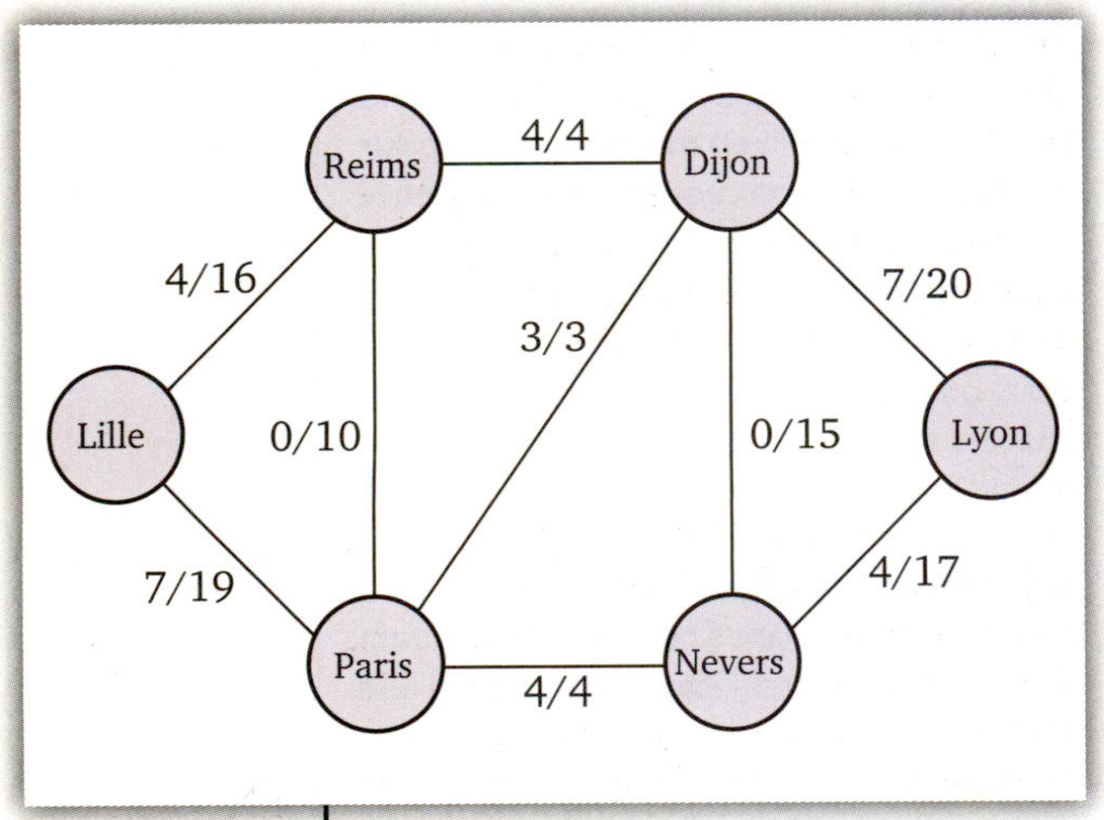

Fig. 8 : Un flot maximal

Chaque fraction α/β signifie « faire passer un débit effectif de α pour une capacité théorique de β ». Cette solution optimise les débits d'eau en utilisant au mieux les capacités des tuyaux : impossible d'envoyer davantage d'eau vers Lyon.

3. Prenons nos ciseaux

3.1 Un problème de coupe

Gardons notre réseau de transport (figure 7) et posons-nous à présent un autre problème : on cherche à **isoler totalement** la ville source (Lille) de la ville puits (Lyon) en coupant des tuyaux. Le résultat sera constitué de deux ensemble de villes qui ne seront reliés par aucun tuyau. Peu importe dans lequel des deux camps se trouveront Paris, Nevers, Reims et Dijon, l'important est que Lille soit dans un camp, et Lyon dans l'autre. Petit souci : couper des tuyaux, cela coûte cher, d'autant plus cher que le tuyau est gros. Le nouveau problème est donc : quelles arêtes (tuyaux) faut-il couper afin de séparer totalement la source du puits, sachant qu'on veut couper le moins d'arêtes possibles, de capacité (diamètre) la plus faible possible ?

3.2 C'est tout pareil ?

Cette nouvelle question, nommée « coupe minimale » semble très différente de la précédente (le flot maximal), et pourtant il s'agit d'un seul et même problème.

Dans notre exemple de flot maximal (figure 8), certaines arêtes portent une mention du type α/α : le débit d'eau que l'on fait effectivement passer dans le tuyau est exactement égal à sa capacité théorique. On dit dans ce cas que l'arête correspondante est « saturée », il n'y a plus de place pour faire passer davantage d'eau. Il existe nécessairement des arêtes saturées dans un flot maximal : s'il n'y en avait aucune, alors il suffirait de prendre n'importe quel chemin de la source vers le puits et d'y injecter un débit supplémentaire (puisque aucune des arêtes de ce chemin ne serait saturée), ce qui contredirait le caractère maximal du flot.

Et en réalité, les arêtes qui se retrouvent saturées dans un flot maximal sont précisément celles qu'il faut couper pour obtenir une coupe minimale (dans l'exemple de la figure 8, la coupe se ferait selon un axe vertical, séparant le graphe en deux parties égales).

En résumé, pour n'importe quel graphe (avec des arêtes « pondérées » par des coefficients) comportant deux nœuds particuliers (la source et le puits), un algorithme peut nous dire quelles arêtes couper afin de séparer la source et le puits à moindres frais. Nous allons maintenant appliquer cette technique aux images.

4. Synthèse intelligente de textures

4.1 Des graphes dans les images

Pour mettre en pratique ce que nous savons sur les graphes dans le domaine des images, voici comment procéder : on commence par considérer que chaque pixel de l'image est un nœud du graphe. Puis, on relie par une arête chaque nœud à ses voisins nord, sud, est et ouest lorsqu'ils existent (voir la figure 9).

Supposons que nous disposions d'une image en niveaux de gris, représentant un objet sombre sur fond clair, et que nous souhaitions séparer automatiquement l'objet du fond, autrement dit déterminer quels pixels appartiennent à l'objet, et quels pixels font partie du fond.

Pour simplifier les choses, considérons seulement un fragment carré, de 4 pixels de côté, extrait de cette image. Ce fragment montre à la fois des pixels clairs, foncés ou intermédiaires, et on cherche à tracer une frontière qui épouse le mieux possible la forme de l'objet, séparant cette image en deux camps.

C'est le moment d'utiliser notre technique de coupe : ajoutons artificiellement deux nœuds au graphe (la source et le puits, que l'on nomme σ et τ), puis relions la source par quelques arêtes à une zone de l'image dont nous sommes sûrs qu'elle appartient au fond clair (voir la figure 9). Faisons de même avec le puits, en le reliant à quelques pixels placés au sein de l'objet (foncé). Reste le problème des capacités des arêtes (la grosseur des tuyaux) : cette fois-ci, il ne s'agit pas de données du problème, c'est à nous de les choisir en fonction du résultat souhaité.

Facile, pour deux pixels donnés, il suffit de faire la différence de leur intensité lumineuse, et de donner à l'arête qui les relie une capacité inversement proportionnelle à cette différence. Par exemple, deux pixels très clairs (faible différence d'intensité) sont situés côte à côte : comme ils appartiennent très probablement au fond tous les deux, on n'a pas intérêt à les séparer par une frontière, si bien que l'on préfère donner à l'arête qui les relie une capacité **forte** pour qu'elle ne soit pas coupée facilement. Au contraire, deux pixels adjacents, un clair et un foncé (forte différence d'intensité) méritent

très probablement d'être séparés par la frontière : on veut donc pouvoir couper facilement l'arête qui les relie, et on lui donne une **faible** capacité.

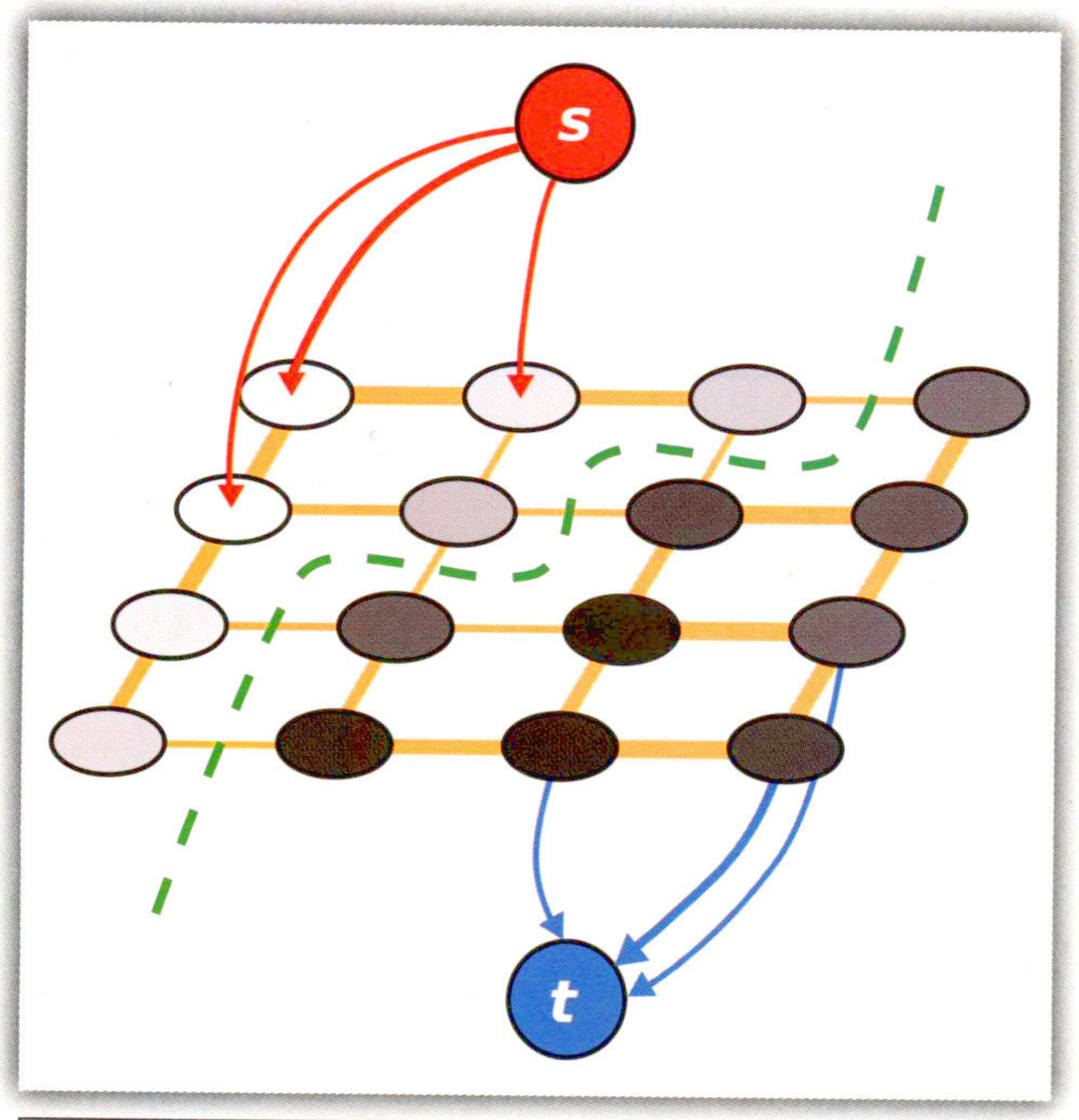

Fig. 9 : Graphe et image

4.2 Fonctionnement de Texturize

Vous avez tout compris ? Passons à la pratique.

4.2.1 Principe général

Rappelons que nous partons d'une petite image de départ (nous l'appellerons le « patch », vous allez comprendre pourquoi) et que nous cherchons à obtenir une texture plus grande (nous appellerons simplement ce résultat l'« image »). On commence donc par créer une image toute blanche, ayant les dimensions demandées par l'utilisateur. Nous allons remplir cette image petit à petit, en collant le patch de façon répétée, jusqu'à ce que l'image soit remplie. Le sens du processus peut faire penser au balayage d'une télévision à tube cathodique (blanc : partie encore vide ; gris : partie déjà remplie) :

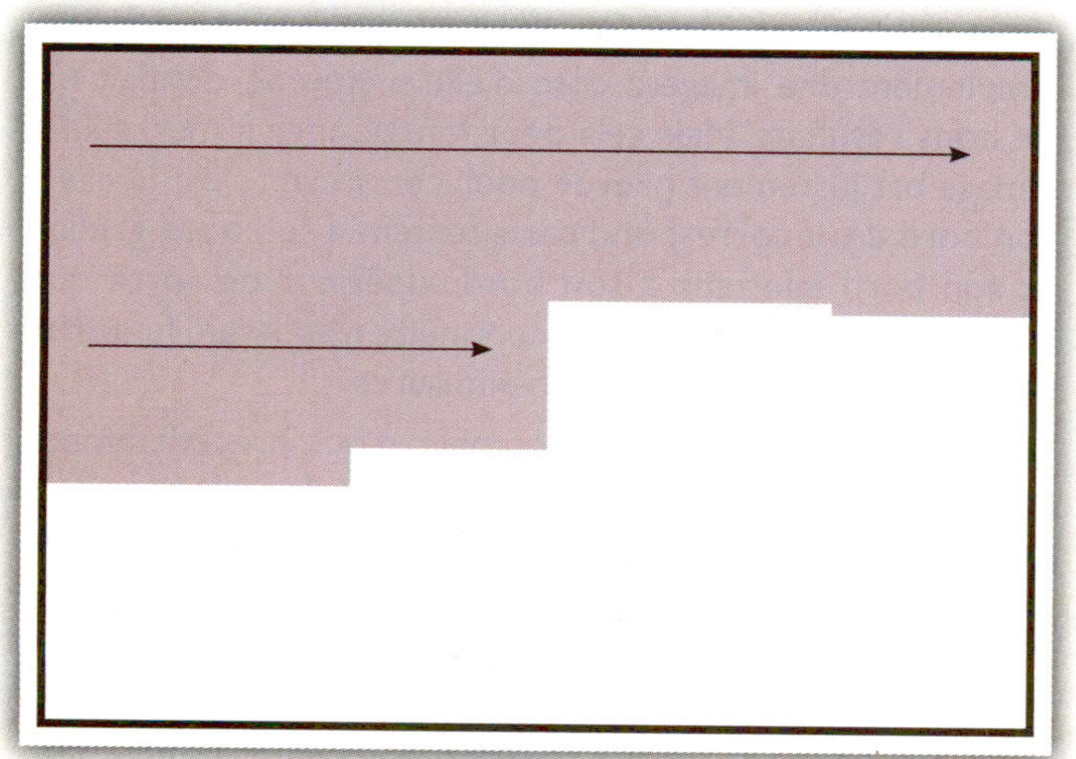

Fig. 10 : Principe général

Heureusement, Texturize n'opère pas par simple copier-coller à répétition du patch sur l'image, sinon le résultat ne serait pas vraiment meilleur que celui de la figure 2. Le programme introduit deux subtilités supplémentaires : un placement optimal de chaque patch collé, et l'utilisation d'une coupe minimale.

4.2.2 Placement du patch

Supposons qu'une partie de l'image soit déjà remplie (figure 11), et cherchons la meilleure façon de placer un nouveau patch. On commence par déterminer quel est le pixel situé le plus en haut et à gauche de la zone encore vide de l'image : c'est lui que nous devons maintenant remplir en priorité. Un peu comme si nous travaillions avec des images imprimées sur du papier transparent, on place une nouvelle copie du patch « par-dessus » ce pixel, et on observe à la fois le contenu du patch et la zone de l'image située au-dessous, que l'on distingue par transparence. Cette observation simultanée du patch et de la zone correspondante dans l'image permet de mettre en évidence les différences existant entre les deux. Or, pour que le patch se fonde le mieux possible dans l'image déjà remplie, il faut précisément que cette différence soit la plus petite possible. Par conséquent, toujours comme avec du papier, on fait glisser le patch autour de sa position d'origine, en observant à chaque fois si la différence avec la zone de l'image située en dessous est importante, et on cherche la position qui minimise cette différence (le programme réalise cela avec une simple différence pixel par pixel). Une fois cette position déterminée, on fixe le patch à cet endroit.

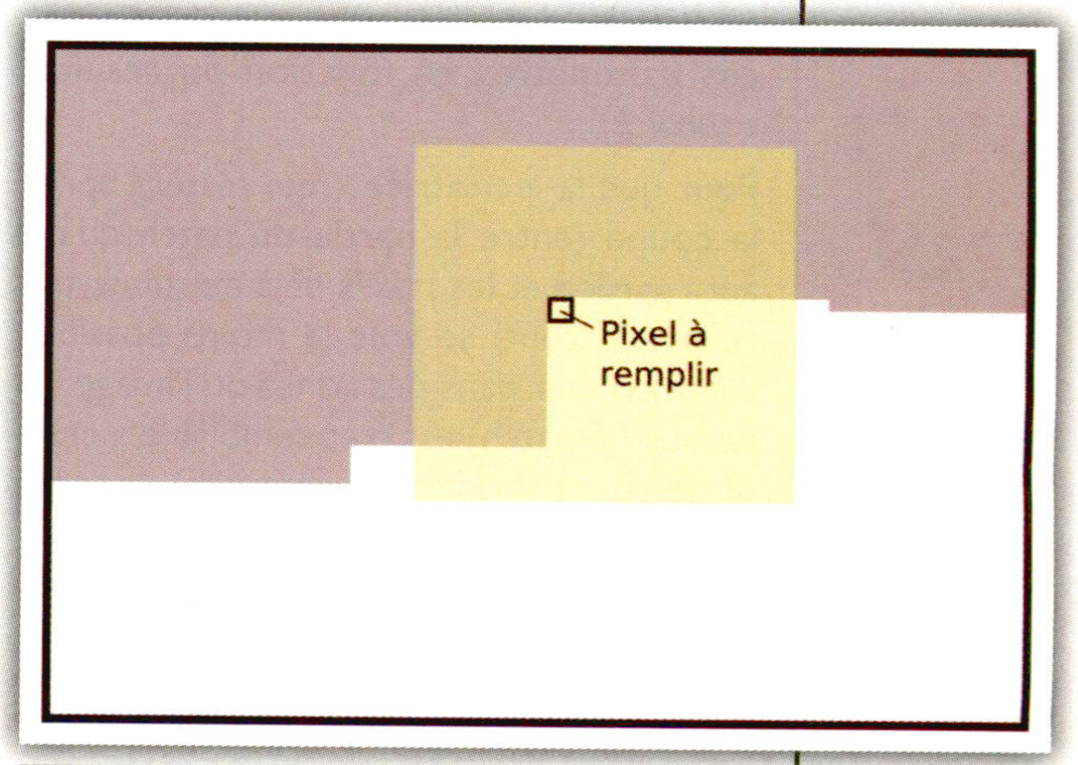

Fig. 11 : Position optimale du patch

4.2.3 Coupe minimale

Mais ce n'est pas tout ! L'étape la plus

importante est encore à venir. Car si l'on colle le patch tel quel sur notre image, la texture ne sera pas naturelle. La partie inférieure droite du patch vient se coller dans une zone encore inoccupée de l'image : pas de problème de ce côté-là. En revanche, sa partie supérieure gauche vient empiéter sur le territoire d'une partie déjà remplie de l'image ; les discontinuités seront inévitables, malgré nos précautions.

Les pixels situés dans cette zone supérieure gauche du patch ne sont pas nécessairement les bienvenus dans la nouvelle image, alors que ceux situés en bas à droite ne gênent personne et peuvent être gardés sans problème. Ainsi, on voudrait déterminer où est la frontière optimale entre le coin nord-ouest du patch (probablement indésirable) et le coin sud-est (que l'on veut garder quoi qu'il arrive). C'est précisément ce que nous avons appris à faire avec les coupes dans les graphes.

Construisons donc notre nouveau graphe ; la seule région importante est celle qui est **commune à l'image déjà remplie et au patch** (le reste ne nous intéresse pas pour le moment), et notre graphe se limitera donc à cette zone. Comme précédemment, on crée un nœud par pixel, on relie chacun à ses voisins immédiats par des arêtes, et on rajoute une source et un puits. Une fois la coupe minimale calculée, une partie des pixels sera collée dans l'image (sud-est), tandis que l'autre sera supprimée (nord-ouest).

Cependant, le problème est légèrement différent de la détection d'un contour, car nous avons ici **deux** images à la fois (le patch et la nouvelle image), superposées. Il faut donc adapter la valeur des capacités que nous allons donner aux différentes arêtes (rappelez-vous, c'est nous qui choisissons ces paramètres en fonction du résultat souhaité).

Pour que la « cicatrice » qui restera après la coupe (entre la partie du patch qu'on aura gardée et les pixels déjà existants, qui réapparaîtront) se voie le moins possible, il faut couper dans des zones où l'image du dessus (le patch) est très semblable à celle du dessous (l'image déjà construite), de sorte que les deux zones séparées par la cicatrice ne soient pas facilement discernables après la coupe. Voici donc la valeur choisie pour la capacité χ d'une arête reliant les nœuds α et β (c'est la seule équation de cet article) :

$$\chi(\alpha, \beta) = |\Pi(\alpha) - I(\alpha)| + |\Pi(\beta) - I(\beta)|$$

où $\Pi(\alpha)$ représente la valeur (en fait, les trois valeurs rouge, vert, bleu) du pixel correspondant à α dans le patch et $I(\alpha)$, dans l'image. Ainsi, si un pixel du patch ressemble beaucoup à celui de l'image située juste en dessous de lui ($\Pi(\alpha) - I(\alpha)$ est faible), et que c'est également le cas pour le pixel voisin ($\Pi(\beta) - I(\beta)$ est faible), alors l'endroit est adéquat pour une coupe, et on attribue un poids faible à l'arête qui relie les deux nœuds.

Une fois la capacité de chaque arête calculée, il ne reste plus qu'à effectuer la coupe ; voici comment la situation se présente :

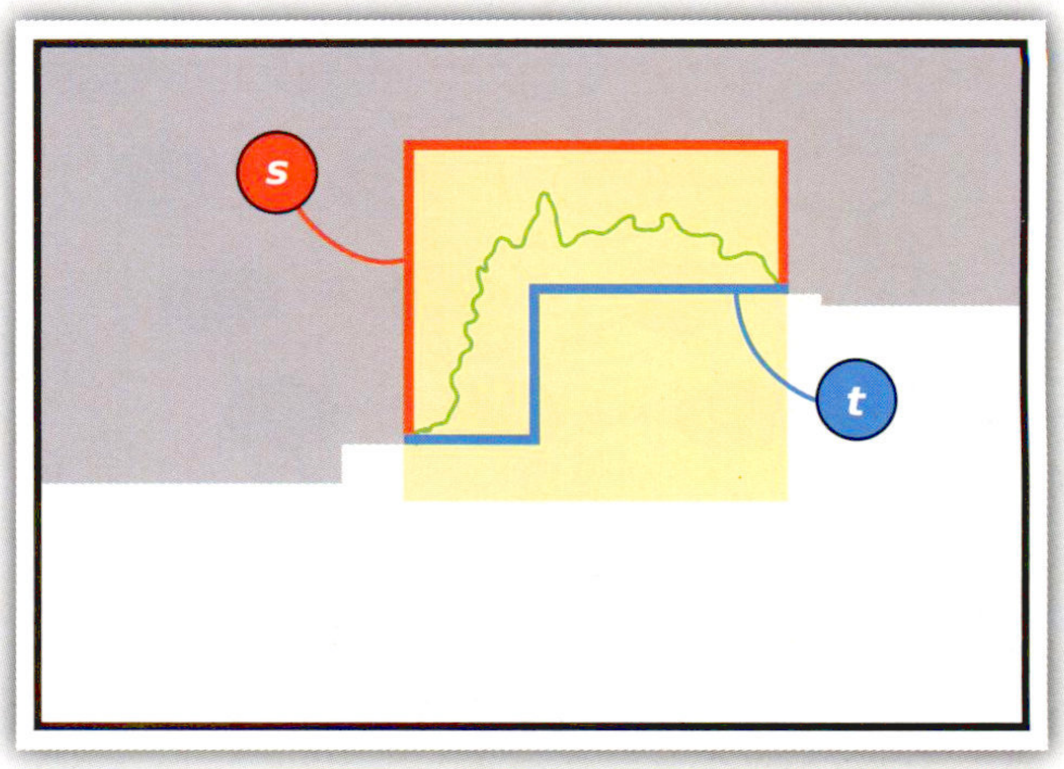

Fig. 12 : Texturize, coupe minimale

Remarquez que nous avons relié au puits les pixels que nous sommes sûrs de vouloir garder (frontières situées au sud-est de l'intersection patch/image déjà remplie, en bleu), et à la source les bords du côté opposé (en rouge). L'algorithme de coupe détermine alors la meilleure frontière séparant les deux camps (en vert) : tous les pixels situés en dessous sont collés dans la nouvelle image, les autres sont jetés à la poubelle.

Une fois cette étape terminée, on cherche le nouveau « prochain pixel à remplir », et on recommence le processus jusqu'à ce que l'image soit complète.

4.2.4 Des textures auto-similaires

Le dernier raffinement de Texturize, introduit dans la version 2.0, est de créer des images « auto-similaires » ou *tileable* en anglais. Nous avons vu, avec les fraises, que répéter simplement une image à côté d'elle-même ne donnait pas de bons résultats. Mais cela peut fonctionner à merveille si l'image est justement prévue pour cet usage : c'est le cas si son bord droit correspond parfaitement à son bord gauche, et son bord inférieur à son bord supérieur de sorte que, répétée en damier, aucune discontinuité n'apparaît. Texturize sait générer des textures auto-similaires.

Le processus est un peu plus complexe que précédemment, mais le principe général est qu'au moment de coller un patch qui dépasserait des limites de l'image (par exemple, la limite inférieure), au lieu de jeter les pixels en surplus, on les colle de l'autre côté de l'image (par exemple, en haut) afin de préserver une continuité. Cela revient à courber l'image dans les deux dimensions afin que ses bords gauche et droit d'une part, haut et bas d'autre part, se rencontrent : on colle alors des patchs non plus sur un rectangle plat, mais sur un *tore* (un beignet à

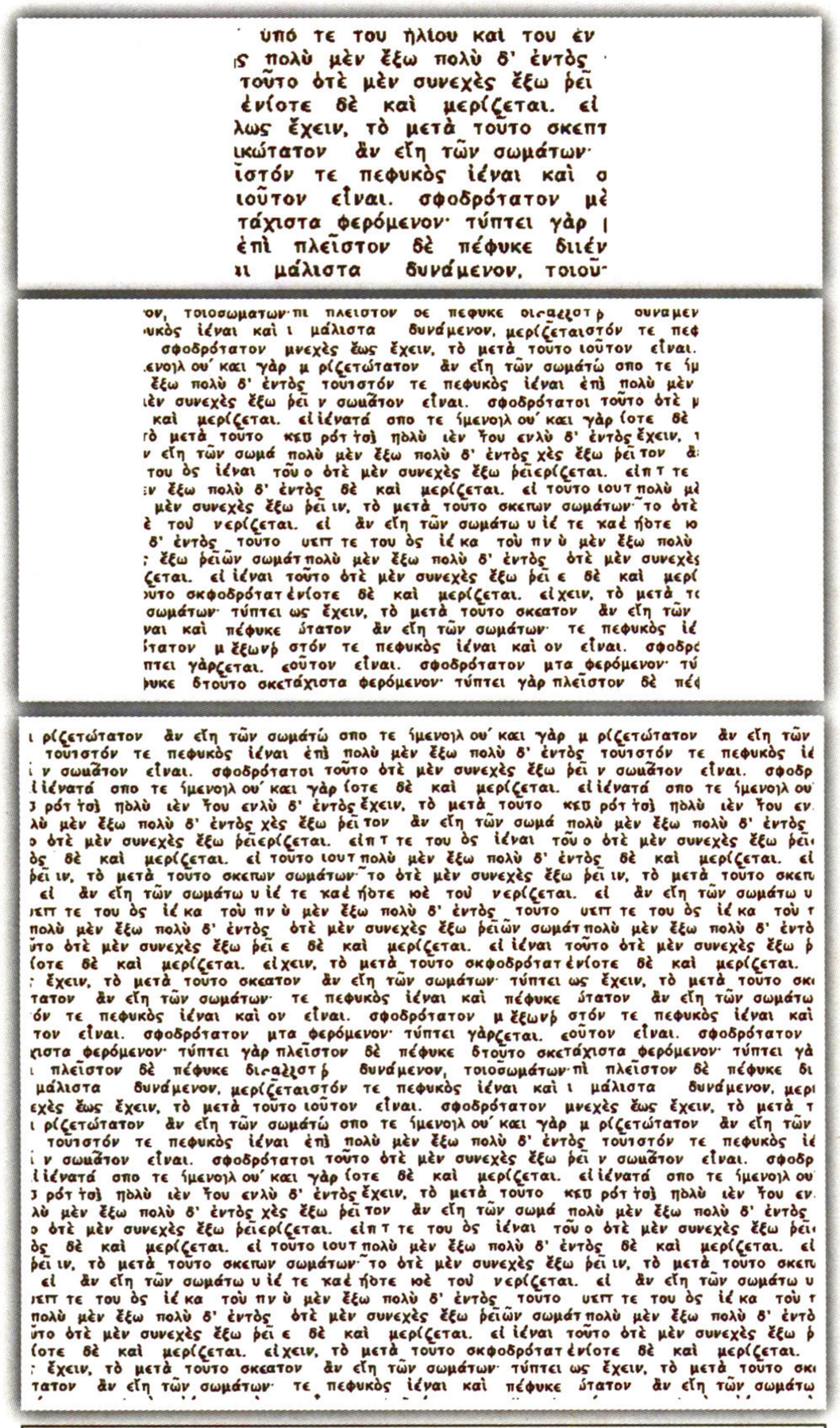

Fig. 13 : Échantillon, texture auto-similaire, répétition

l'américaine, troué en son milieu – ou une chambre à air, mais c'est moins appétissant). Cette fonction était très demandée, en particulier par les créateurs de jeux et les concepteurs de sites web : avec des textures auto-similaires, on peut obtenir des images de taille arbitraire (tout l'arrière-plan d'une longue page web ou d'un niveau de jeu vidéo), en stockant seulement une petite image, simplement par répétition. Voici par exemple ce qu'on obtient en traitant un petit morceau de texte en grec (ci-dessus figurent l'échantillon de départ, puis la texture auto-similaire, et enfin cette dernière répétée quatre fois). (Même si vous avez appris le grec, je vous garantis que vous n'y comprendrez rien !)

5. Et concrètement, comment on fait ?

Installation

Si vous travaillez sous Debian, Ubuntu, ou une distribution dérivée, installez simplement le paquet `gimp-texturize` (vérifiez qu'il s'agit bien de la version 2.0). Dans le cas contraire, la meilleure solution est de compiler le code source vous-même : un classique `./configure && make && (sudo) make install` conviendra, une fois le plugin téléchargé depuis la page web (voir la section « Liens »). Pensez aussi à installer les librairies de développement de The GIMP.

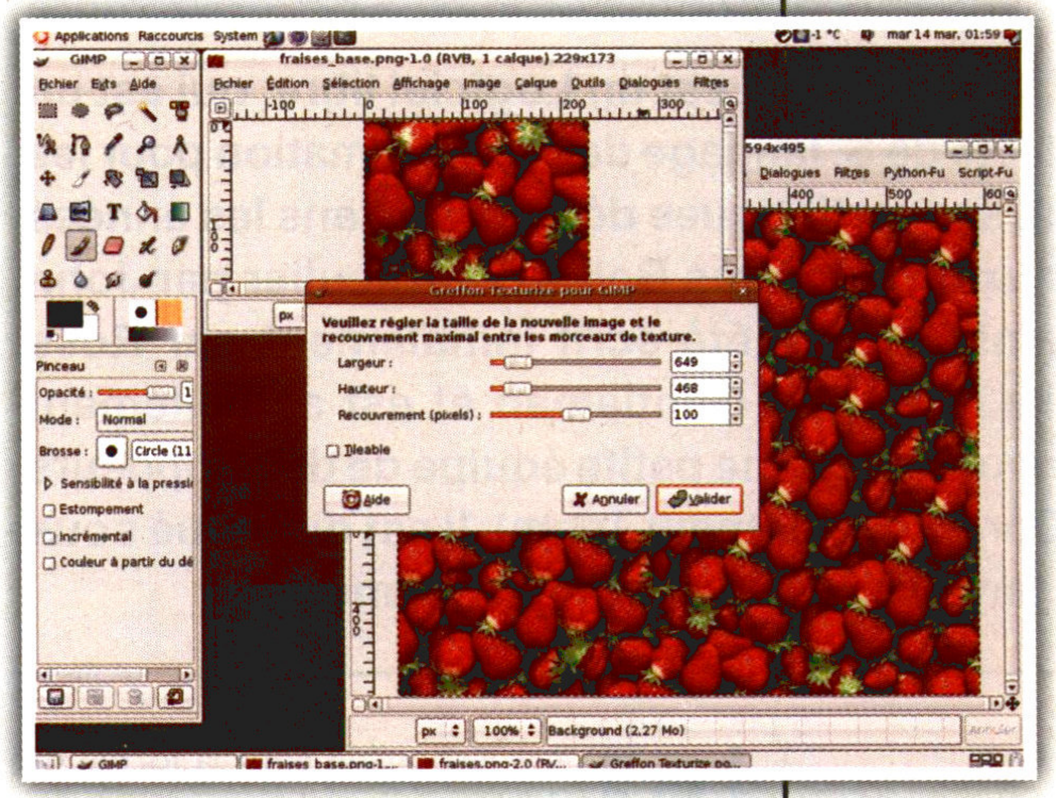

Fig. 14 : Interface de Texturize

Utilisation

Vous trouverez le plugin dans le sous-menu *Carte* du menu *Filtres* (après avoir relancé GIMP le cas échéant). Les réglages sont très simples : choisissez la taille de la nouvelle image, le taux de recouvrement (qui correspond à la taille de la zone explorée pour la recherche d'une position optimale : un taux important allonge le temps de calcul), et cochez éventuellement la case *Tileable*, le programme s'occupe du reste.

Conclusion

Texturize s'appuie sur des résultats de recherche assez récents (2003), utilise une méthode originale, et produit des textures naturelles d'excellente qualité. Que vous soyez développeur de jeux, webmestre ou simplement amateur de belles images, j'espère qu'il vous sera utile. N'hésitez pas à m'envoyer vos meilleures créations, elles seront peut-être incorporées à la page officielle !

LIENS

- Plugin Texturize pour GIMP, de Manu Cornet et Jean-Baptiste Rouquier. Page officielle du plugin (en anglais) : **www.manucornet.net/gimp_texturize.php**
- L'article de recherche sur la synthèse de textures et les *graph cuts* : **www.cc.gatech.edu/cpl/projects/graphcuttextures**

Manu Cornet,
manu@manucornet.net,