# III. Automates cellulaires. Simulation de phénomènes naturels

Les récurrences correspondent à des phénomènes séquentiels. Elles illustrent l'évolution d'une situation globale, par exemple celle d'une population. Une autre façon de voir consiste à prendre en compte les interactions localisées de cette multitude d'individus, chacun étant influencé par son environnement. A chaque étape de temps, tous ces individus évoluent en même temps, en fonction de leur situation locale. Le processus est typiquement parallèle. On parle à ce sujet d'automates cellulaires. Ceux-ci sont censés représenter des particules, des cellules vivantes, des individus... Soumis à des règles simples dictées par leur voisinage immédiat, ils donnent lieu à des phénomènes complexes et contrastés. Apparition localisées de formes organisées à partir d'un contexte aléatoire, ou inversement développement de mouvements désordonnés et complexes qui cependant obéissent à des lois d'ensemble, partout règne l'harmonie des contraires, de l'ordre et du désordre. Cela concerne aussi bien des populations que l'évolution de formes dans le monde naturel, et même les incendies de forêt.

## 1. Feux de forêt

Voici comment simuler de façon grossière mais simple un feu de forêt. Dans un quadrillage sont placés des arbres au hasard, avec une certaine densité. Ces arbres vont avoir trois états possibles : non brûlés (en vert), en feu (en rouge), consumés (en gris). Au départ, ils sont tous verts, sauf ceux qui sont sur le front de feu initial. La règle de propagation est simple : à chaque étape de temps, un arbre en feu transmet le feu à ses quatre plus proches voisins éventuels (ceux du quadrillage). Plus précisément, les arbres sont disposés à l'intérieur d'un grand carré. On évite de placer des arbres sur la bordure extérieure du carré, afin de contenir le feu à l'intérieur : si les points du carré ont leurs coordonnées $x$ et $y$ entre 0 et $L$, les arbres sont placés avec des $x$ et des $y$ entre 1 et $L - 1$. Un front de feu est disposé sur la bordure verticale gauche (en $x = 1$), avec les arbres mis en rouge. Ensuite la propagation se fait par voisinage, étape par étape. Les arbres qui étaient en feu sont considérés à l'étape suivante comme étant consumés (en gris noir). Ils ne transmettent le feu que pendant un et un seul intervalle de temps. Le phénomène s'arrête quand il n'y a plus d'arbres en flammes.

### 1.1. Programmation

Le programme principal se réduit à la mise en place de la forêt dans le carré de côté $L$, à la mise à feu, puis à la propagation répétée, jusqu'à ce qu'il n'y ait plus d'arbres en feu, le nombre des arbres en feu étant placé dans la variable *nbarbresenfeu* qui évolue au fil des étapes de temps. Le test d'arrêt se produit lorsqu'il n'y a plus d'arbres en feu.

```
 foret();
miseafeu();   nombreetapes=0; /* la variable nombreetapes permet de connaître le temps jusqu'à
                                 l'extinction finale */
do { propagation(); nombreetapes++; }
while(nbarbresenfeu!=0);
```

Commençons par la mise en place de la forêt. Cela revient à placer un arbre (ou pas) en chaque point du quadrillage avec une probabilité égale à la densité *d* choisie (par exemple $d = 0{,}6$).[1] Les points (*i*, *j*) où se trouvent les arbres ont chacun une couleur dont l'indice est enregistré dans un tableau *p*[*i*][*j*]. Là où il y a des arbres, on fait *p*[*i*][*j*] = 1 qui est l'indice du vert, tandis que les points sans arbres restent en blanc (couleur d'indice 0),[2] ce qui sous-entend que le tableau *p*[][] a été mis à 0 en conditions initiales.

```
void foret(void)
{ int i,j,hasard;
  srand(time(NULL)); nombrearbres=0; /* ce compteur du nombre d'arbres est facultatif */
  for(i=1; i<L; i++)  for(j=1; j<L; j++)
   { hasard=rand()%1000;
     if(hasard<(int)(d*1000.)) {p[i][j]=1;nombrearbres++;}
   }
/* Ce qui suit, en italiques, suppose que l'on a intégré à SDL la bibliothèque SDL_ttf  qui permet d'écrire sur
l'écran graphique. C'est facultatif dans un premier temps */
   police=TTF_OpenFont("times.ttf",20);
   sprintf( chiffre,"densite : %3.2f", (float)nombrearbres/(float)(L*L));
   texte=TTF_RenderText_Solid(police,chiffre,cblack);  position.x=600; position.y=100;
   SDL_BlitSurface(texte,NULL,screen,&position);

  for(i=1;i<L;i++) for(j=1;j<L;j++) if (p[i][j]==1)  arbre(i,j,1);
}
```

La fonction *arbre*(*x*, *y*, *icolor*) est chargé de dessiner l'arbre au point (*x*, *y*) avec sa couleur d'indice *icolor*, verte pour le moment, sous forme d'un carré de côté *pas*. On peut prendre *pas* = 4 avec $L = 145$, ce qui revient à faire un zoom sur l'écran, un arbre étant représenté par un carré de côté 4 pixels.[3] Dans ce contexte, le carré de la forêt occupe une longueur de $145 \times 4 = 580$ pixels.

```
void arbre(int x,int y, int icolor)
{ int i,j;
  for(i=0;i<pas;i++) for(j=0;j<pas;j++)  putpixel(pas*x+i,pas*y+j,couleur[icolor]);
}
```

Cela étant fait, on place le front de feu en $x = 1$, ce qui revient à mettre en rouge (indice 2) les arbres verts de cette ligne verticale (*figure 1*), et l'on place dans la variable *nbarbresenfeu* le nombre d'arbres qui sont en flammes au départ.

```
void miseafeu(void)
{ int j;
  nbarbresenfeu=0;
```

[1] Pour faire cela, il suffit de tirer un nombre au hasard entre 0 et 999 (par exemple). Si le nombre obtenu est inférieur à 600 (pour une densité 0,6) on dessine un arbre au point (*i*, *j*), sinon on ne place pas d'arbre.

[2] Les couleurs sont ainsi enregistrées :

```
couleur[0]=SDL_MapRGB(screen->format,255,255,255);   /* blanc */
couleur[2]=SDL_MapRGB(screen->format,255,0,0);    /* rouge */
couleur[1]=SDL_MapRGB(screen->format,0,180,0);   /* vert */
couleur[3]=SDL_MapRGB(screen->format,100,100,100);   /* gris */
```

[3] Pour bien voir la propagation du feu, il est nécessaire de faire un tel zoom. Si l'on se contentait de prendre un arbre correspondant à un pixel d'écran, la propagation ne serait plus intéressante. Pourquoi ? Parce que l'on a pris une densité d'arbres qui est partout la même. Avec un nombre limité d'arbres (grâce au zoom), il n'y a pas d'uniformité dans la répartition. Par contre avec un grand nombre d'arbres, il y aurait tendance à l'uniformité dans la disposition des arbres, et la propagation deviendrait elle aussi régulière. Si l'on veut utiliser un arbre correspondant à un pixel, il convient de découper le carré de la forêt en plusieurs carrés plus petits, en faisant varier la densité dans chacun de ces carrés.

```
  for(j=1;j<L;j++)
  if (p[1][j]=1) { p[1][j]=2; arbre(1,j,2); nbarbresenfeu++; }
  for(i=1;i<L;i++) for(j=1;j<L;j++) np[i][j]=p[i][j];    /* voir ci-dessous pourquoi cela */
  SDL_Flip(screen);pause();
}
```

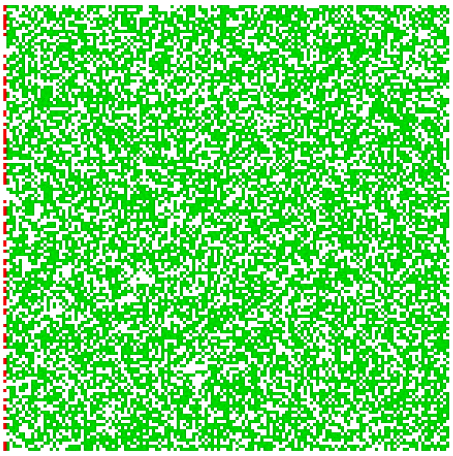

*Figure 1* : Forêt avec une certaine densité d'arbres, et mise à feu initiale à gauche.

Il reste la principale fonction, celle de la propagation, qui nécessite l'usage de deux tableaux *p* et *np*.[4] A chaque étape de la propagation, ces deux tableaux sont rendus identiques au début. Remarquons que nous avons fait cela dans la fonction de mise à feu (*voir ci-dessus la ligne en italique*), avant de lancer la propagation où l'égalité des deux tableaux est faite à la fin de chaque étape de temps. Puis on parcourt le carré de la forêt en considérant les valeurs de *p*[*i*][*j*] : si l'arbre en (*i*, *j*) est en feu, c'est un propagateur et l'on regarde ses quatre voisins éventuels. Si l'un des voisins est un arbre vert, on le met en rouge pour l'étape suivante, en faisant cette modification dans le tableau *np*, et l'on augmente de 1 la variable *nbarbresenfeu*. Mais il faut faire attention. Il se peut qu'un voisin mis à rouge dans *np* l'ait déjà été auparavant comme voisin d'un autre arbre lors du parcours séquentiel du carré. Il convient d'éviter de compter deux fois au lieu d'une ce passage au rouge. Enfin l'arbre propagateur en (*i*, *j*) va passer de rouge à gris, cette transformation est aussi faite dans le tableau *np*, tout en diminuant de 1 la variable *nbarbresenfeu*. Une fois le parcours terminé, le tableau *p* est mis à jour en intégrant les modifications mises dans *np*. On fait cela pour tous les arbres qui ont subi un changement (*np*[*i*][*j*] différent de *p*[*i*][*j*]). On peut alors relancer la propagation à l'étape suivante.

```
void propagation(void)
{ int i,j;
  for(i=0;i<L+1;i++)  for(j=0;j<L+1;j++)   np[i][j]=p[i][j];
  for(i=1;i<L;i++) for(j=1;j<L;j++)  if (p[i][j]==2)
   { if (p[i][j+1]==1 && np[i][j+1]!=2) {np[i][j+1]=2; nbarbresenfeu++; }
     if (p[i][j-1]==1 && np[i][j-1]!=2) {np[i][j-1]=2; nbarbresenfeu++; }
     if (p[i+1][j]==1 && np[i+1][j]!=2) {np[i+1][j]=2; nbarbresenfeu++; }
     if (p[i-1][j]==1 && np[i-1][j]!=2) {np[i-1][j]=2;  nbarbresenfeu++; }
     np[i][j]=3;nbarbresenfeu--; /* 3 est l'indice du gris, couleur de l'arbre consume */
   }
```

[4] Rappelons que ce genre d'algorithme de modification de paysage est synchrone (modification en parallèle) alors que l'usage d'une machine séquentielle rend indispensable le parcours séquentiel de l'écran. Ce n'est qu'après un parcours complet que l'on peut effectuer les modifications et engager l'étape suivante. D'où la nécessité de prendre deux tableaux, le deuxième enregistrant les modifications en attendant la fin du parcours. Si l'on ne prenait qu'un seul tableau, en effectuant les changements au fur et à mesure du parcours, on aboutirait à des aberrations. Par exemple, avec une rangée horizontale d'arbres où le plus à gauche serait le seul à être en feu, le parcours séquentiel de la rangée à partir du deuxième arbre mettrait le feu de proche en proche à toute la ligne d'arbres, et cela au cours d'une seule étape de temps. Ou encore en partant du premier arbre en feu, celui-ci passerait à l'état consumé, et ne mettrait le feu à aucun de ses voisins.

```
  for(i=1;i<L;i++) for(j=1;j<L;j++) if (np[i][j]!=p[i][j])
    {  p[i][j]=np[i][j];   arbre(i,j,p[i][j]);  }
  SDL_Flip(screen);SDL_Delay(20);
}
```

### 1.2. Les arbres comme récepteurs, et non propagateurs

Dans le programme précédent, chaque arbre est considéré comme un émetteur éventuel : s'il brûle, il transmet le feu alentour. On peut aussi bien considérer chaque arbre comme un récepteur : s'il est vert, et qu'il a un voisin rouge, il devient rouge. La fonction *propagation* s'écrit alors plus simplement :

```
for(i=1;i<L;i++) for(j=1;j<L;j++)
if (p[i][j]==1)
    { if (p[i][j+1]==2) {np[i][j]=2; nbarbresenfeu++; }
      else if (p[i][j-1]==2) {np[i][j]=2; nbarbresenfeu++; }
      else if (p[i+1][j]==2) {np[i][j]=2; nbarbresenfeu++; }
      else if (p[i-1][j]==2) {np[i][j]=2; nbarbresenfeu++; }
     }
else if (p[i][j]==2)   {np[i][j]=3;nbarbresenfeu--;}
for(i=1;i<L;i++) for(j=1;j<L;j++) if (np[i][j]!=p[i][j])  { p[i][j]=np[i][j];  arbre(i,j,p[i][j]);}
```

### 1.3. Résultats

Le temps mis par le feu pour s'éteindre, en termes de nombre d'étapes, est lié à la densité de la forêt. Lorsque la densité des arbres est faible, la propagation est vite stoppée, faute de voisins à qui mettre le feu, et le temps d'extinction est petit. Lorsque la densité est forte, proche de 1, la propagation s'effectue facilement, le front de feu se déplace parallèlement à lui-même, et le temps d'extinction est proche de $L$. Entre ces deux cas extrêmes, le temps d'extinction présente un maximum très net. On assiste en effet à des phénomènes capillaires, avec des développements tentaculaires du feu dans plusieurs directions (*figure 2*). Ce maximum est obtenu pour une densité des arbres de l'ordre de 0,6, le temps d'extinction dépassant dans ce cas $2L$. La relation entre la densité et le temps d'extinction peut être trouvée en procédant à une série d'essais pour chaque valeur de la densité (prise de 0,1 en 0,1). Par exemple, avec un carré de côté $L$ = 110, le temps d'extinction atteint un maximum de l'ordre de 230 pour une densité d'atbres de $d$ = 0,6. Par contre, pour $d$ = 0,4, le temps est 15, et pour $d$ = 0,80, le temps est 132 (figure 3).

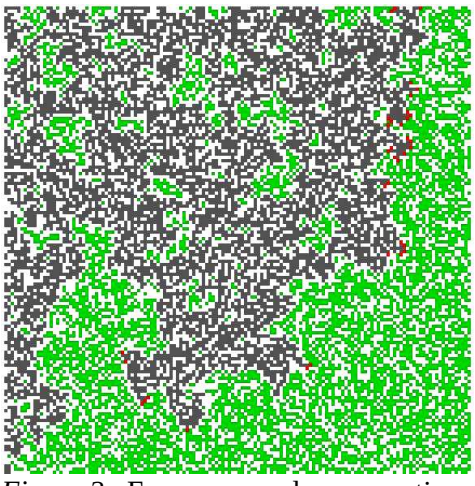

*Figure 2* : Feu en cours de propagation.

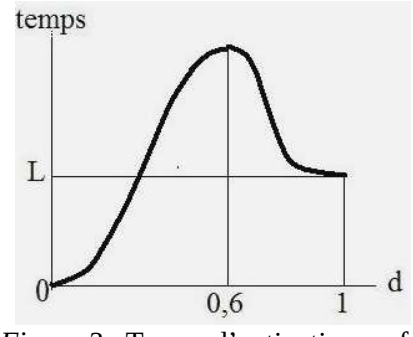


*Figure 3* : Temps d'extinction en fonction de la densité.

### 1.4. Première variante : Extinction lente

Au lieu de faire passer un arbre de l'état en feu à l'état consumé en une seule étape de temps, on va le laisser se consumer progressivement sur *P* étapes de temps (par exemple *P* = 30), en le faisant passer de rouge vif à rouge très foncé, voire noir. Cela va faire apparaître une traînée derrière les pointes rouges du feu (figure 4). Les couleurs sont ainsi définies :

```
couleur[0]=SDL_MapRGB(screen->format,255,255,255);   /* blanc */
couleur[1]=SDL_MapRGB(screen->format,0,180,0);   /* vert */
couleur[2]=SDL_MapRGB(screen->format,255,0,0);    /* rouge vif */
for(i=3;i<=P+3;i++)  /* couleurs passant de rouge  à noir */
couleur[i]=SDL_MapRGB(screen->format, 220-(int)((150.*(float)(i-3)/(float)P)),0,0);
```

Le programme principal ne change pas par rapport à ce que l'on a fait précédemment (sauf une fonction qui s'ajoute à la fin comme on le verra), pas plus que les fonctions *foret*() et *miseafeu*(). Reste la fonction de propagation, qui doit tenir compte de l'extinction lente : lorsqu'un arbre a une couleur d'indice supérieur ou égal à 2, son indice augmente de 1.

```
void propagation(void)
{ int i,j;
  for(i=1;i<L;i++) for(j=1;j<L;j++)
  if (p[i][j]==1)
   { if (p[i][j+1]==2) {np[i][j]=2; nbarbresenfeu++; }
     else if (p[i][j-1]==2) {np[i][j]=2; nbarbresenfeu++; }
     else if (p[i+1][j]==2) {np[i][j]=2; nbarbresenfeu++; }
     else if (p[i-1][j]==2) {np[i][j]=2; nbarbresenfeu++; }
    }
  else if(p[i][j]==2) {np[i][j]=3;nbarbresenfeu--;} /* début de l'extinction */
  else if (p[i][j]>2  && p[i][j] <P+3)   {np[i][j]=p[i][j]+1;} /* évolution de l'extinction */
  for(i=1;i<L;i++) for(j=1;j<L;j++) if (np[i][j]!=p[i][j])
        {  p[i][j]=np[i][j];    arbre(i,j,p[i][j]); }
  SDL_Flip(screen);SDL_Delay(20);
}
```

La propagation s'arrête lorsqu'il n'existe plus d'arbres en feu. Mais quand cela se produit, il reste des arbres qui ne sont pas encore entièrement consumés. Il convient d'ajouter une petite fonction à la fin du programme principal qui termine le processus, soit *extinctionfinale*() ainsi programmée :

```
void extinctionfinale(void)
{  int i,j,flag;
  do
  {flag=0;
   for(i=1;i<L;i++) for(j=1;j<L;j++)
  if (p[i][j]>2  && p[i][j]<P+3) {np[i][j]=p[i][j]+1; flag=1;} /* arbres pas encore complètement consumés */
  for(i=1;i<L;i++) for(j=1;j<L;j++) if (np[i][j]!=p[i][j])
        {  p[i][j]=np[i][j];    arbre(i,j,p[i][j]); }
  }
  while(flag==1);
}
```

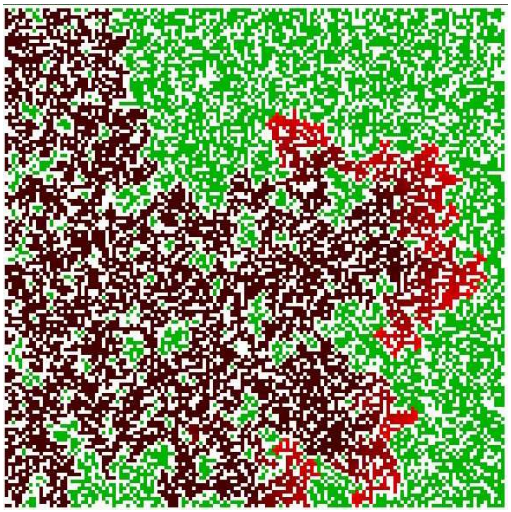

*Figure 4* : Propagation du feu avec extinction graduelle.

### 1.5. Deuxième variante : Régénérescence des arbres consumés

On considère maintenant qu'après avoir fini de se consumer, comme dans le cas précédent, les arbres redeviennent verts. L'extinction graduelle du feu se fait toujours avec les couleurs dont l'indice va de 3 à $P + 3 - 1$, tandis que la dernière couleur d'indice $P + 3$ est la même que la couleur verte d'indice 1. La palette de couleurs devient :

```
couleur[0]=SDL_MapRGB(screen->format,255,255,255);   /* blanc */
couleur[1]=SDL_MapRGB(screen->format,0,200,0);   /* vert */
couleur[2]=SDL_MapRGB(screen->format,255,0,0);    /* rouge */
for(i=3;i<3+P;i++) couleur[i]=SDL_MapRGB(screen->format,220-(int)(200./((float)P)*(float)(i-3)),0,0);
couleur[3+P]=SDL_MapRGB(screen->format,0,200,0);   /* vert comme couleur[1]*/
```

Pour simplifier, et pour donner au cheminement du feu une marge de liberté plus grande, on rend la forêt cyclique (*figure 5*). Cela signifie que ce qui déborde hors d'un côté du carré est replacé à l'intérieur de l'autre côté. Il n'y a plus de problème de bordure, et la forêt occupe maintenant un carré de sommets opposés (0, 0) et ($L - 1$, $L - 1$).

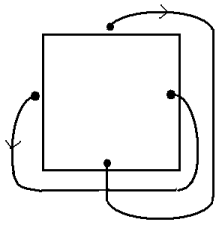

*Figure 5* : Le carré contenant la forêt est rendu cyclique.

Le programme principal s'écrit :

```
 foretcyclique();
miseafeu();
nombreetapes=0;
do
  { propagation();nombreetapes++;     }
while(nombreetapes!=100);
extinctionfinale();
```

Que va-t-il se passer ? La régénérescence régulière, et rapide des arbres (par rapport à ce qui se passe en réalité) offre de nouvelles possibilités d'expansion pour le feu. On pourrait penser que le feu

serait ainsi ranimé indéfiniment. Mais on va constater qu'il n'en est rien. Le feu va finir étouffé. Commençons par faire le programme.

- Mise en place de la forêt cyclique, la fonction *arbre*(*i,j,icolor*) restant inchangée :

```
void foretcyclique(void)
{ int i,j,hasard;
  srand(time(NULL));
  for(i=0;i<L;i++)  for(j=0;j<L;j++)     { hasard=rand()%1000;   if(hasard<(int)(d*1000.)) p[i][j]=1; }
  for(i=0;i<L;i++) for(j=0;j<L;j++) if (p[i][j]==1) arbre(i,j,1);
}
```

- Mise à feu :

Plutôt que de démarrer avec un front de feu vertical, nous allons partir d'une vingtaine d'arbres pris au hasard, auxquels est mis le feu simultanément. D'où la fonction :

```
void miseafeu(void)
{ int i,j,k;
  for(k=0;k<=20;k++)
    { do  { i=rand()%L; j=rand()%L;}  /* point (i,j) avec i et j pris au hasard entre 0 et L – 1 */
      while(p[i][j]!=1);
      p[i][j]=2; arbre(i,j,2);
    }
  for(i=0;i<L;i++) for(j=0;j<L;j++) np[i][j]=p[i][j];
  SDL_Flip(screen);
}
```

- Fonction propagation sur une étape :

La phase d'extinction d'un arbre en feu, sur *P* étapes, va être divisée en deux. Pendant la première moitié, sur *P*/2 étapes, on va considérer que l'arbre va encore propager le feu alentour, tandis que pendant la deuxième moitié, où l'arbre finit de se consumer, il ne transmet plus le feu autour de lui.

A cause de la forêt qui a été rendue cyclique, les quatre voisins du point (*i*, *j*) sont maintenant :
((*i* + 1)%*L*, *j*)
((*i* + *L* – 1)%*L*, *j*) [5]
(*i*, (*j* + 1)%*L* )
(*i*, (*j* + *L* – 1)%*L* )

Comme précédemment, à chaque étape, un arbre qui se consume voit son indice de couleur augmenter de 1, et il finit par atteindre l'indice *P* + 3. Lors de la modification finale du tableau *p*[][], via le tableau *np*[][], tous les arbres ayant la couleur *P* + 3 sont remis à la couleur 1.

```
void propagation(void)
{ int i,j;
  for(i=0;i<L;i++) for(j=0;j<L;j++)
   { if (p[i][j]==1 )
       { if (p[i][(j+1)%L]>=2 && p[i][(j+1)%L]<P/2) {np[i][j]=2;}
         else if (p[i][(j-1+L)%L]>=2  && p[i][(j-1+L)%L]<P/2) {np[i][j]=2; }
         else if (p[(i+1)%L][j]>=2 && p[(i+1)%L][j]<P/2) {np[i][j]=2;  }
         else if (p[(i-1+L)%L][j]>=2 && p[(i-1+L)%L][j]<P/2) {np[i][j]=2;}
       }
     else if (p[i][j]>=2  && p[i][j] <3+P)   {np[i][j]=p[i][j]+1;}
    }
```

[5] Comme le modulo % en langage C ne fonctionne correctement que sur des nombres positifs ou nul, on doit prendre (*i* + *L* – 1)%*L* et non pas (*i* – 1)%*L*, car il peut arriver que *i* – 1 devienne négatif.

```
  for(i=0;i<L;i++) for(j=0;j<L;j++)
  if (np[i][j]!=p[i][j])
    { p[i][j]=np[i][j];    arbre(i,j,p[i][j]);
      if (p[i][j]==3+P) p[i][j]=1;
    }
  SDL_Flip(screen);SDL_Delay(20);
}
```

- Fonction d'extinction finale :

Elle reste inchangée.

L'exécution du programme montre que le feu finit par s'éteindre, malgré le reverdissement régulier des arbres, ce qui peut sembler paradoxal. Pourquoi en est-il ainsi ?

Comme on le voit sur la *figure 6* [6], les fronts de feu issus des départs de feu se mêlent pour former des poches où le feu est emprisonné, avec une barrière formée d'arbres consumés qui empêchent le feu d'atteindre les arbres verts. Le feu meurt par asphyxie.

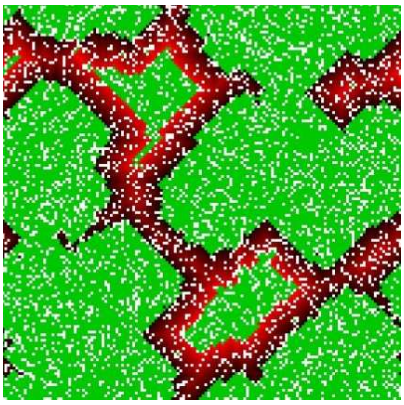

*Figure 6* : Poches de feu encerclées par des arbres consumés. Le feu s'éteint.

Cela peut s'appliquer à un autre problème, celui d'une épidémie. Dans ce contexte, les arbres deviennent des individus, disposés dans une zone géographique avec une densité *d*. Ils sont sains au départ (*en vert*), puis quelques individus malades (*en rouge*) sont disposés de-ci de-là. La transmission de la maladie se fait alors par voisinage. Quand un individu est touché, il reste contagieux pendant *P*/2 étapes de temps, puis pendant les *P*/2 étapes suivantes, il est immunisé et non contagieux, sur la voie de la guérison qui se produit juste après (couleur *P* + 3, il redevient vert). Le phénomène observé expérimentalement explique alors pourquoi l'épidémie finit par s'arrêter toute seule, non pas faute de « combattants », mais par sa propre asphyxie.[7]

### 1.6. Troisième variante : Le feu éternel

Pour empêcher l'arrêt de l'incendie, comme cela se produit dans l'exemple précédent, on doit relancer le feu régulièrement, en créant de nouveaux départs de feu localisés. Cela revient à modifier légèrement le programme principal de l'exemple précédent, toutes les autres fonctions restant identiques.

---

[6] Pour mieux voir le phénomène nous avons pris une densité forte, $d$ = 0,8, ce qui provoque un front de feu d'allure rectangulaire plutôt que circulaire, cela étant dû au fait que l'on prend seulement quatre voisins autour de chaque point.

[7] La limitation de la contagion et la fin de l'épidémie n'est en général expliquée que par la mise en quarantaine des malades, ou par la miséricorde divine.

```
nombreetapes=0;
do
   { if (nombreetapes%P==0) miseafeu();  /* nouvelles mises à feu localisées toutes les P étapes de temps */
      propagation(); nombreetapes++;
    }
while(nombreetapes!=1000);
extinctionfinale();
```

On aboutit ainsi à un phénomène périodique d'expansion-régression. Un état de l'évolution est indiqué sur la *figure 7*.

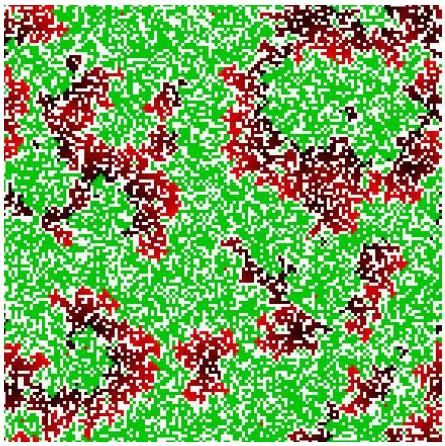

*Figure 7* : Le feu éternel.

## 2. Phénomènes d'agrégat et développement de formes coralliennes

### 2.1. Mouvement brownien

Au début du 19è siècle, le botaniste écossais Robert Brown constata que dans l'eau des particules minuscules en suspension étaient animées d'un mouvement irrégulier et incessant. Ce mouvement, qualifié de brownien, fut plus tard expliqué par le bombardement des molécules d'eau sur les particules, d'où l'aspect aléatoire de leur mouvement en ligne brisée. En termes mathématiques, le mouvement brownien correspond à la notion de fonction continue nulle part dérivable. Les trajectoires possèdent aussi un aspect d'autosimilarité: chaque partie ressemble au tout, les petits segments irréguliers et en dents de scie qui apparaissent sous un certain grossissement apparaîtront de façon analogue si l'on augmente le grossissement.

Comment simuler un tel mouvement sur ordinateur ? Même si cela apparaît simpliste, on fait en sorte qu'à chaque instant, la particule avance d'un pas dans une des quatre directions Nord, Sud, Est, Ouest. Cela s'appelle aussi le mouvement de l'homme ivre dans un quadrillage infini de rues.

### 2.2. Phénomène d'agrégat

Reprenons ces particules en suspension dans l'eau, et ajoutons quelques obstacles : des points, de petits amas, des surfaces à l'intérieur desquelles l'eau circule. On va supposer que lorsque les particules mobiles touchent un de ces obstacles, elles s'y collent à l'endroit exact de l'impact (il n'y a aucun phénomène de gravité comme pour les sédiments). L'amas initial croît peu à peu. On assiste alors à la formation d'arborescences avec des branches ou des tentacules, qui évoquent le développement de coraux, ou ressemblent à des formes de dendrites ou d'arcs électriques. Ce phénomène existe tel quel lorsqu'un dépôt de métal se fait dans une cuve à électrolyse.

Pourquoi de telles arborescences? En vertu d'une sorte d'évidence : quand on est ivre, il est plus facile de se planter au début qu'au fond d'un tunnel. Une particule soumise à un mouvement brownien

a en effet plutôt tendance à se coller au bout d'une branche plutôt que d'entrer à l'intérieur d'une cavité et de s'y coller au fond. Ce sont les bosses, les aspérités, les branches qui ont tendance à se développer.

### *Exercice 1 : Formes coralliennes*

*Programmer le développement de ces formes, dans divers contextes, en tenant compte des remarques suivantes :*

* *On distingue deux cas de figure selon la forme des obstacles mis en place initialement. Dans le premier cas, le fluide est entouré par une surface solide sur laquelle vont se coller les particules. Dans le deuxième cas, le fluide est considéré comme infini:et l'obstacle est un petit ilôt. Il se peut alors qu'une particule circule longuement sans se coller, et il convient d'arrêter son mouvement dès qu'elle sort d'un certain périmètre.*
* *Pour faciliter le programme, on lance une particule à la fois et l'on attend qu'elle se colle (ou qu'elle sorte de l'écran) avant d'en lancer une autre.*

*1) Traiter le cas où le fluide est à l'intérieur d'un cercle.*

Ce cercle est colorié en noir. Puis on lance des particules une à une, à condition que celles-ci soient à l'intérieur du cercle et aussi dans une zone qui n'est pas coloriée en noir. Si la particule a un voisin déjà colorié en noir, on la colle en la dessinant en noir. Sinon, on lance un mouvement au hasard jusqu'à ce que la particule ait un voisin en noir, auquel cas on la dessine en noir. Puis on passe à un autre lancer de particule. On voit apparaître des arborescences dont la racine est collée au cercle, et celles-ci grossissent lors des lancers successifs de particules (*figure 8*).

```
on se donne le centre (xorig,yorig) du cercle et son rayon
circle(xorig,yorig,rayon,black); circle(xorig,yorig,rayon+1,black); /* cercle de bordure */
SDL_Flip(screen) ;
for(lancer=1;lancer<15000;lancer++)  /* lancer de particules */
   { do       /* point initial où se trouve la particule */
       { r=rand()%rayon; angle=(float)(rand()%360)*M_PI/180.;
         xe=xorig+r*cos(angle);ye=yorig-r*sin(angle);
       }
     while(getpixel(xe,ye)==black);
     while(collage(xe,ye)==0)   /* mouvement du point jusqu'à son collage */
         { dir=rand()%4;
           if (dir==0) xe++;
           else if (dir==1) ye--;
           else if (dir==2) xe--;
           else if (dir==3) ye++;
         }
       putpixel(xe,ye,black);
    }
```

Avec la fonction testant si oui ou non il y a collage :

```
int collage(int xe, int ye)
{ if (getpixel(xe+1,ye)==black || getpixel(xe-1,ye)==black
      || getpixel(xe,ye+1)==black || getpixel(xe,ye-1)==black)
  return 1;
  return 0;
}
```

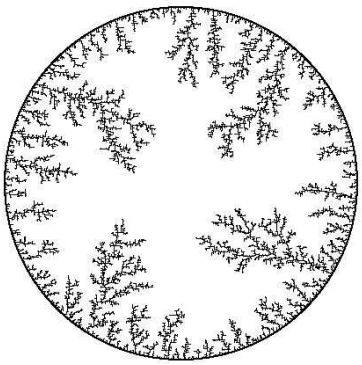

*Figure 8* : Formes arborescentes dont la racine est collée à un cercle.

*2) Création d'une forme de corail à partir d'un germe*

*Comme le germe est petit, une particule lancée au hasard peut mettre longtemps avant de se coller. On a intérêt à accélérer le phénomène de collage en procédant ainsi :*

** Placer le germe au centre O (xorig, yorig) de l'écran blanc, sous forme d'un petit disque noir, du style filldisc(xorig,yorig,3,black), et définir une zone de lancement des particules, entre deux cercles de rayon R1 et R2, par exemple R1 = 40 et R2 = R1+20.*

* *Lancer une particule au hasard dans cette zone, et colorier en rouge son point de départ déterminé par son rayon R (sa distance à O) et son angle angled en degrés puis angler en radians, ce qui donne ses coordonnées : xe = xorig + R*cos(angler); ye=yorig+R*sin(angler). Puis la particule se déplace pas à pas sur le quadrillage au hasard dans l'une des quatre directions possibles. Ce mouvement ne sera pas dessiné sur l'écran. On attend qu'il se termine. Deux cas de fin sont possibles : soit la particule sort du cercle de rayon R2 et c'est fini pour cette particule, soit elle atteint un point colorié en noir, auquel cas elle se colle sur la forme noire. Dans ce deuxième cas, on détermine la distance entre la particule qui vient de se coller et le centre O, ou plutôt son carré d2.*

* *Lancer une nouvelle particule dans la même zone, et faire comme précédemment.*

*Continuer ainsi, mais dès qu'une particule a sa distance d2 très proche de la zone de lancement, prendre une nouvelle zone de lancement, avec R1 qui augmente de 20 (l'ancienne valeur de R2) et le nouveau R2 = R1 + 20. Puis poursuivre le lancer de particules dans cette nouvelle zone. Et ainsi de suite, jusqu'à ce que la zone de lancement entre R1 et R2 atteigne les limites des l'écran. Il ne restera plus qu'à faire disparaître les points rouges correspondant aux points de départ des particules. Remarquons que le fait de colorier en rouge les points initiaux est seulement fait pour des raisons pédagogiques.*

Voici le programme, avec des résultats sur la *figure 9* :

```
filldisc(xorig,yorig,3,black);  SDL_Flip(screen);  /* le germe initial */
R1=40;compteur=0;  /* compteur va être le nombre de particules collées sur la forme */
while (R1<280)
   { R2=R1+20;
     R=R1+ rand()%20;angledegres=rand()%360;
     angleradians=(float)angledegres*M_PI/180.;
     xe=xorig+R*cos(angleradians); ye=yorig+R*sin(angleradians);
     filldisc(xe,ye,1,red);  /* point initial colorié en rouge */
     for(;;)
       { oldxe=xe;oldye=ye;
         dir=rand()%4;
         if (dir==0) xe++;else if(dir==1) ye--;else if(dir==2) xe--;else if (dir==3) ye++;
         d2=(xe-xorig)*(xe-xorig)+(yorig-ye)*(yorig-ye);
         if (d2> R2*R2) break;  /* la particule sort de la zone */
         if (getpixel(xe,ye)==black)
```

```
            { putpixel(oldxe,oldye,black); compteur++;
              if (compteur%200==0) SDL_Flip(screen);
              if (d2>(R1-10)*(R1-10))  R1+=20; /* la couronne des lancers a ses rayons qui augmentent */
              break;
            }
        }
    }
    for(i=0;i<800;i++) for(j=0;j<600;j++) if (getpixel(i,j)==red)
    putpixel(i,j,white);
```

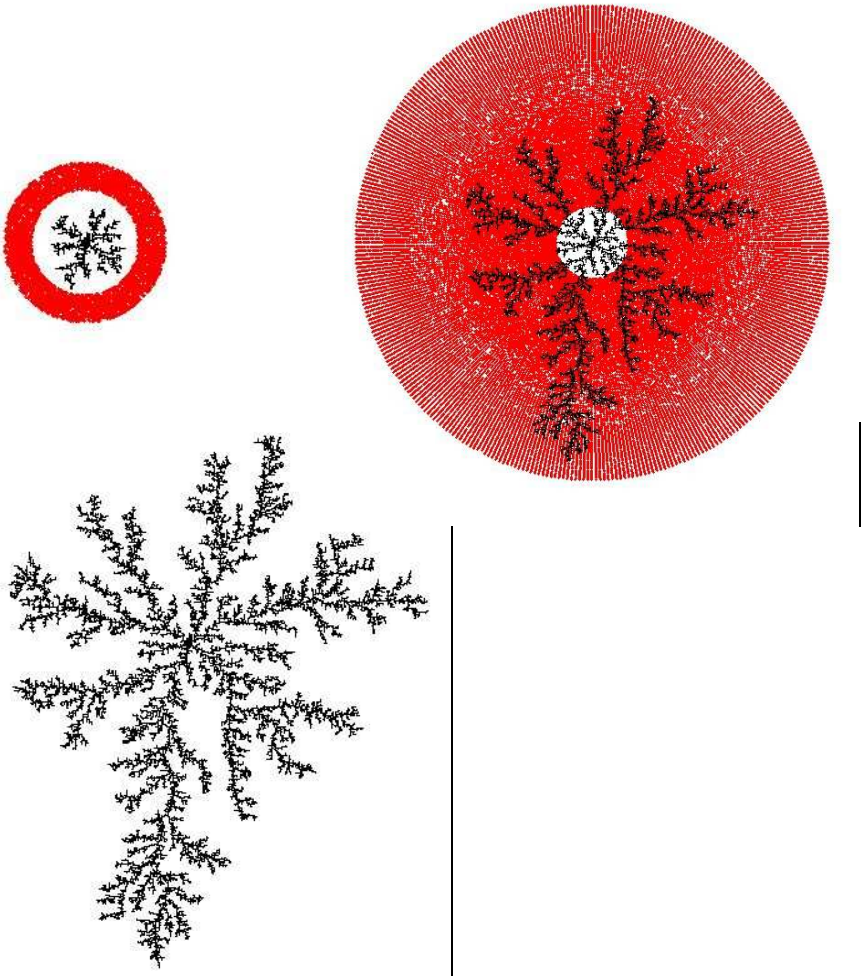

*Figure 9* : *En haut* le développement de la forme corallienne, au début *à gauche* avec la couronne des points de lancers coloriée en rouge, et en cours d'évolution *à droite*. *En bas* la forme finale.

## 3. Morphogenèse, à propos de motifs, tâches ou zébrures, qui se forment sur les peaux d'animaux

Dans ce modèle, il est considéré que la peau contient au départ deux types de cellules, les cellules actives *A*, et les cellules passives *P*. Les cellules actives ***A***, et elles seules, provoquent des transformations dans leur voisinage -comme les arbres en feu précédemment, mais de façon plus complexe. Elles contiennent d'une part un activateur qui agit dans un voisinage proche en provoquant la transformation de certaines cellules ***P*** en cellules ***A***, et d'autre part un inhibiteur qui provoque dans un voisinage plus lointain la transformation de cellules ***A*** en cellules ***P***, selon les dispositions suivantes :

* La peau est formée au départ de cellules *A* et *P*, réparties au hasard, et dans la proportion moitié-moitié par exemple, mais cela n'a en fait pas d'importance.

* Chaque point *C* de la peau qui contient une cellule *A* ou une cellule *P* est considéré comme un récepteur. On considère son voisinage proche, pour nous un carré de côté 2 $R_1$ autour du point central *C*, et son voisinage plus lointain situé entre le carré précédent et un carré de côté 2 $R_2$, avec $R_2 > R_1$.

* Les cellules actives *A* autres que la cellule *C* (du moins si *C* est active) situées dans le voisinage proche agissent chacune sur *C* proportionnellement à leur nombre *n*. D'autre part les cellules *P* situées dans le voisinage plus lointain agissent chacune sur *C* proportionnellement à leur nombre *n*'. On forme alors la quantité

$W = n - k\,n'$ , *k* étant un coefficient que l'on se donne.

Ce nombre *W* correspond à la somme des influences relatives de l'activateur et de l'inhibiteur, l'une étant positive et l'autre négative. Pour respecter les règles de notre modèle de morphogenèse, on décide que si *W* est positif, correspondant au rôle prépondérant de l'activateur, la cellule *C* est transformée en cellule *A* dans le cas où elle était *P*, et que si *W* est négatif, avec le rôle prépondérant de l'inhibiteur, la cellule *C* est transformée en cellule *P* si elle était *A*. Sinon aucun changement ne se produit.

***Exercice 2 : Programmation pour obtenir des peaux tachetées***

*Dans le programe, la peau sera représentée par un carré de côté L, en distinguant les cellules A et P par leur couleur, indexée respectivement 1 et 0, une cellule étant un point (x,y) du carré, x et y étant entre 0 et L – 1. On remplira ainsi un tableau c[x][y] contenant soit 0 soit 1 pour distinguer les deux couleurs. Puis on parcourt point par point ce carré, en prenant pour chaque point C ses voisinages proche et lointain. Ces voisinages seront considérés comme cycliques (on dit aussi torique) : si le point C est près d'une bordure du carré, la partie des voisinages situées au dehors est ramenée à l'intérieur en prenant le bord opposé.*

***1)*** *Programmer pour calculer W en chaque point, et effectuer le changement éventuel de couleur suivant le signe de W. En attendant d'avoir fini le parcours du tableau c[x][y], on utilisera un nouveau tableau newc[x][y] où l'on placera les nouvelles valeurs. Puis on actualisera en remettant le tableau newc[][] dans c[][], et on dessinera la nouvelle peau sur l'écran. Comme valeurs des paramètres, on pourra prendre $R_1 = 2$, $R_2 = 6$ et k de l'ordre de 0,2, avec L = 490 par exemple. Il convient en effet de choisir une valeur de k (dépendant de $R_1$ et $R_2$) qui aboutit à un équilibre entre les cellules A et P. Pour d'autres valeurs de k, ce sont soit les cellules A qui remplissent tout, soit les cellules P. Puis répéter l'opération précédente plusieurs fois. Constater qu'il suffit de la faire 7 ou 8 fois pour que le dessin se stabilise et donne une peau tachetée.*

Lors du parcours du voisinage de chaque cellule, on s'intéresse à chaque cellule *A* dans le grand voisinage de côté 2 $R_2$. Si en plus cette cellule *A* est dans le petit voisinage de côté 2 $R_1$, elle fait augmenter *W* de 1, sinon (on est alors entre $R_1$ et $R_2$), elle fait diminuer *W* de *k*.

```
On se donne ici L = 290 avec pas = 2, A = 1 et P = 0, et deux couleurs rouge et jaune, couleur[1] et
    couleur[0].
for(x=0;x<L;x++)  for(y=0;y<L;y++)  /* remplissage initial du carré de côté L */
   { hasard=rand()%2;if (hasard==0) c[x][y]=A; else c[x][y]=P;  } /* A = 1 et P = 0 */
for(x=0;x<L;x++) for(y=0;y<L;y++) carre(x,y,c[x][y]); /* une cellule est un petit carré sur l'écran */
SDL_Flip(screen);pause();
for(etape=1;etape<9;etape++)  /* boucle des étapes */
  { for(x=0;x<L;x++)  for(y=0;y<L;y++)
      { W=0.;
        for(dx=-R2;dx<=R2;dx++) for(dy=-R2;dy<=R2;dy++) /* parcours du grand voisinage */
        if((dx!=0 || dy!=0) && c[(x+dx+L)%L][(y+dy+L)%L]==A) /* on tombe sur une cellule A */
          { if (abs(dx)<=R1 && abs(dy)<=R1) W+=1.;   /* on a une cellule proche A */
            else W-=k;   /* on a une cellule lointaine A */
          }
```

```
            if (W>0.) nc[x][y]=A;  else if (W<0.) nc[x][y]=P; else nc[x][y]=c[x][y];
          }
      for(x=0;x<L;x++) for(y=0;y<L;y++)   {c[x][y]=nc[x][y]; carre(x,y,c[x][y]); }
     SDL_Flip(screen);
  }
```

Avec l'appoint d'un zoom effectué par la fonction *carré*() :

```
void carre(int x,int y, int icouleur) /* une cellule est représentée par un carré de côté pas */
{ int i,j;
  for(i=pas*x;i<pas*(x+1);i++) for(j=pas*y;j<pas*(y+1);j++)
  putpixel(i,j,couleur[icouleur]);
}
```

Dans ce contexte, la peau est dessinée sur l'écran par des pixels dont les coordonnées sont entre 0 et $pas \times L$ (*figure 10*).

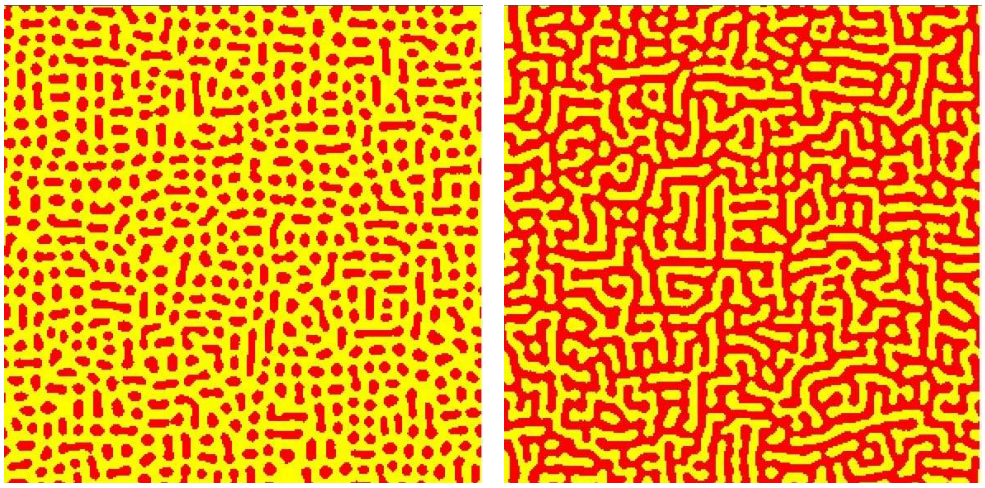

*Figure 10* : Résultats obtenus pour $k$ = 0,23 *à gauche* et $k$ = 0,16 *à droite*.

***2)*** *Comment faut-il modifier légèrement le programme pour obtenir des zébrures à prédominance horizontale, évoquant aussi des rides de sable dans le désert ?*

Il suffit de prendre des voisinages rectangulaires au lieu de carrés (*figure 11*). Sur la *figure de droite*, on a utilisé une palette de gris, indexée de 0 à 255, et à partir du dessin des zébrures en noir (indice 0) et blanc (indice 1), on fait circuler point par point un filtre qui remplace la couleur du point par une moyenne de ses quatre voisins.

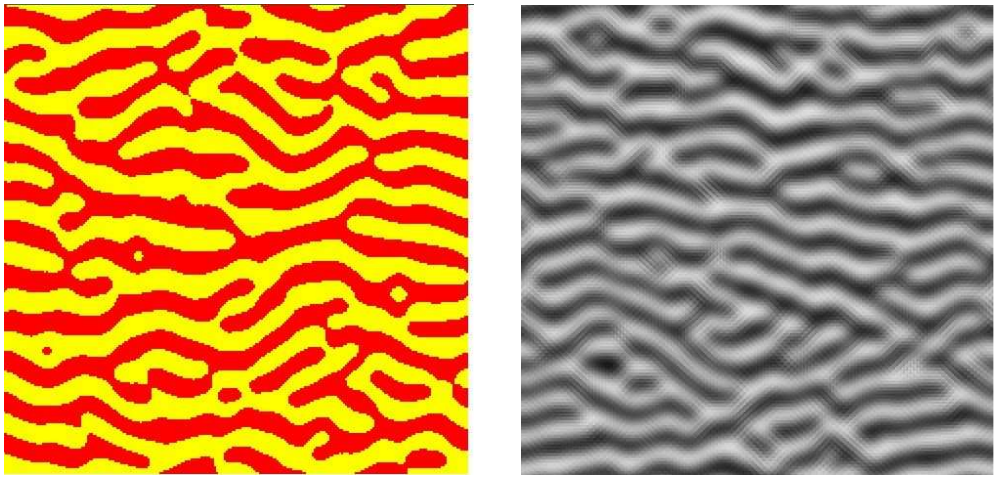

*Figure 11* : Zébrures *à gauche*, et *à droite* un effet de relief supplémentaire.

Dans ce phénomène de morphogenèse, comme dans la plupart des exemples qui précèdent, on a vu comment, à partir d'un certain désordre, se créent des formes organisées, ou du moins qui présentent entre elles un air de famille. L'inverse est tout autant possible. La désagrégation des formes, ou leur désintégration sous l'effet d'attaques diverses aboutit à l'informe, comme leur réduction en poussière.

### *Exercice 3 : Désintégration*

*Dessiner un disque au centre de l'écran, et le découper en petits carrés de côté pas. Puis procéder à la désingration du disque, avec ses petits carrés qui se dispersent tout autour. Trouver des règles simples d'évolution et programmer.*

Quelle que soit la méthode utilisée, il convient de découper le disque en petits carrés. Pour ce faire on dessine sur l'écran un disque noir sur fond blanc. Puis on parcourt le disque horizontalement et verticalement de *pas* en *pas*, cette variable *pas* étant le côté d'un petit carré. Par la même occasion, on enrigistre les coordonnées *X*[*i*], *Y*[*i*] du coin en haut à gauche de chaque petit carré, celui-ci étant numéroté par *i*. Puis on dessine ces petits carrés en rouge, et au cas où le pas est grand, il reste une zone en noir du disque initial que l'on supprimre en la mettant en blanc. Dorénavant, le disque est remplacé par des petits carrés adjacents. Ce que fait ce début de programme :

```
filldisc(xorig,yorig,100,black);  /* disque noir sur fond blanc, placé au centre de l'écran */
k=0;
for(x=10;x<700;x+=pas) for(y=10;y<500;y+=pas)
if (getpixel(x,y)==black)                /* parcours du disque de pas en pas et enregistrement des coordonnées
    { X[k]=x; Y[k]=y; nX[k]=x;nY[k]=y; k++; }                                  de chaque coin des carrés */
nbpoints=k;                              /* nombre des petits carrés */
for(i=0;i<nbpoints;i++) carre(X[i],Y[i],red);      /* coloration des carrés en rouge */
for (y=100;y<500;y++) for(x=100;x<700;x++)   /* suppression éventuelle des residus en noir */
if (getpixel(x,y)==black) putpixel(x,y,white);
SDL_Flip(screen); pause();
```

Puis on entre dans la boucle d'évolution dans le temps où les petits carrés vont se déplacer. A chaque étape, le déplacement de chaque petit carré va être fonction de ses quatre voisins, dont les coordonnées sont notés *vx*[*j*], *vy*[*j*] avec *j* allant de 0 à 3, le voisin 0 étant situé à l'est, le voisin 1 au nord, etc. Quelles règles de mouvement choisir ? On va considérer deux configurations :

- Le carré rouge a un voisin rouge et un voisin blanc de l'autre côté. On distingue quatre cas selon la position de ces voisins. Par exemple, si le carré voisin rouge est à gauche et le carré voisin blanc à droite du carré rouge considéré, on déplace ce dernier vers la droite, en l'échangeant avec le carré blanc (*figure 12*). C'est ce que fait dans le programme la fonction *gaucherouge*(). On fait de même pour les trois autres directions, avec trois fonctions analogues. Cette règle va permettre de déplacer les carrés situés sur le pourtour du disque, ou de ce qu'il en reste.

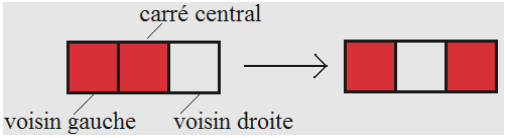


*Figure 12* : Déplacement vers la droite du carré central.

Notons que pour éviter les interférences, on utilise deux tableaux pour les déplacements, les nouvelles positions des carrés étant placées dans les tableaux *nX*[], *nY*[], puis à la fin de l'étape de parcours, les valeurs *nX*[], *nY*[] sont remises dans *X*[], *Y*[]. Notons aussi que si l'on traite en premier les cas où les voisins sont à l'horizontale, et en second ceux où les voisins sont à la verticale, le parcours séquentiel va privilégier les premiers au détriment des seconds. Aussi fait-on intervenir deux

cas au hasard, celui avec les voisins horizontaux en premier, et celui avec les voisins verticaux en premier.

- Le carré rouge a deux voisins blancs de part et d'autre. Deux cas se présentent selon que ces voisins sont à l'horizontale ou à la verticale. Tout cela arrive dès que les carrés rouges se sont séparés du disque central et s'en éloignent de plus en plus. Par exemple, si le voisin à gauche est blanc ainsi que le voisin à droite, on déplace le carré rouge en l'échangeant avec le carré blanc correspondant soit vers la droite, soit vers la gauche, selon qu'il est situé à droite ou à gauche du centre de l'écran (*figure 13*). On fait de même lorsque l'on a deux voisins blancs à la verticale, en déplaçant le carré vers le haut ou vers le bas selon que l'on est au-dessus ou au dessous du centre de l'écran.

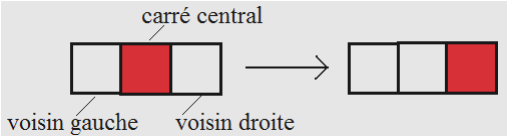


*Figure 13* : Déplacement de la case rouge dans le cas où l'on est à droite du centre de l'écran.

La suite et la fin du programme principal en découle, avec ses réusltats sur la *figure 14* :

```
for(etape=1;etape<200;etape++)  /* boucle des étapes */
   { for(i=0;i<nbpoints;i++)   if (getpixel(X[i],Y[i])==red)  /* on prend chaque petit carré rouge */
       { vx[0]=X[i]+pas;vy[0]=Y[i];  vx[1]=X[i];vy[1]=Y[i]-pas;  /* les quatre voisins */
         vx[2]=X[i]-pas;vy[2]=Y[i];   vx[3]=X[i];vy[3]=Y[i]+pas;
         q=rand()%2;  /*cas où un carré a un voisin rouge et un blanc */
         if (q==0)   /* voisins horizontaux en premier */
            { if(getpixel(vx[2],vy[2])==red) gaucherouge();
              else if(getpixel(vx[0],vy[0])==red) droiterouge();
              else if(getpixel(vx[3],vy[3])==red)  basrouge();
              else if(getpixel(vx[1],vy[1])==red)  hautrouge();
            }
         else if (q==1)  /* voisins verticaux en premier */
            { if(getpixel(vx[3],vy[3])==red)  basrouge();
              else if(getpixel(vx[1],vy[1])==red) hautrouge();
              else if(getpixel(vx[0],vy[0])==red) droiterouge();
              else if (getpixel(vx[2],vy[2])==red) gaucherouge();
            }
         else (q=rand()%2); /* cas où un carré rouge a deux voisins blancs */
         if (q ==0)
             { if(getpixel(vx[2],vy[2])==white && getpixel(vx[0],vy[0])==white)
                  droitegaucheblanc();
               else if(getpixel(vx[1],vy[1])==white && getpixel(vx[3],vy[3])==white)
                  hautbasblanc();
             }
         else if (q==1)
             { if(getpixel(vx[1],vy[1])==white && getpixel(vx[3],vy[3])==white    )
                  hautbasblanc();
               else if(getpixel(vx[2],vy[2])==white && getpixel(vx[0],vy[0])==white)
                  droitegaucheblanc();
             }
       }
     for(i=0;i<nbpoints;i++) {X[i]=nX[i]; Y[i]=nY[i]; } /* actualisation des tableaux */
     SDL_FillRect(screen,0,white);
     for(i=0;i<nbpoints;i++)   /* dessin des carrés à chaque étape */
     if (X[i]<770 && Y[i]>30 && Y[i]<570 && X[i]>30) {carre(X[i],Y[i],red);}
     SDL_Flip(screen);
}
```

Les fonctions d'appoint se présentent ainsi :

```
void gaucherouge(void)
{ if (getpixel(vx[0],vy[0])==white && getpixel(vx[0]+pas,vy[0])==white
                    && getpixel(vx[0]+2*pas,vy[0])==white && getpixel(vx[0]+3*pas,vy[0])==white)
                    {nX[i]=X[i]+(2+rand()%3)*pas;nY[i]=Y[i]; }
  else   if (getpixel(vx[0],vy[0])==white  && getpixel(vx[0]+pas,vy[0])==white
                        && getpixel(vx[0]+2*pas,vy[0])==white)
                       {nX[i]=X[i]+(2+rand()%2)*pas;nY[i]=Y[i]; }
  else  if (getpixel(vx[0],vy[0])==white && getpixel(vx[0]+pas,vy[0])==white)
                       {nX[i]=X[i]+(1+rand()%2)*pas;nY[i]=Y[i]; }
  else   if ( getpixel(vx[0],vy[0])==white)
                        {nX[i]=X[i]+pas;nY[i]=Y[i]; }
 }

void droitegaucheblanc(void)
{ if ( X[i]<xorig-rand()%50)
     {nX[i]=X[i]-pas;nY[i]=Y[i]; }
  else  if(  X[i]>xorig+rand()%50)
     {nX[i]=X[i]+pas;nY[i]=Y[i]; }
 }
```

Les autres fonctions sont de la même forme.

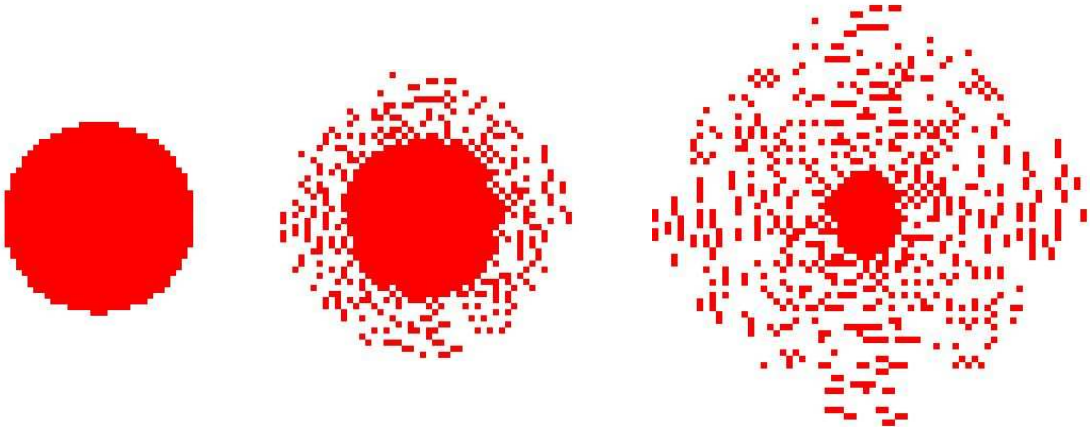

*Figure 14* : Désintégration du disque.

## 4. Plasma, ou formes nuageuses

L'état de plasma est le quatrième état de la matière, au-delà de solide, liquide et gaz, et il est obtenu à très hautes températures. Nous allons ici faire un programme qui donne un dessin évoquant artistiquement cet état de plasma, avec des formes ondoyantes aux couleurs vives. Plus prosaïquement, en utilisant une palette de niveaux de gris, le dessin ressemble à un ciel nuageux. Cela va se faire en deux temps. On commence par créer un dessin avec un dégradé continu de couleurs, par interpolation bilinéaire, puis on ajoutera des variations au hasard pour arriver au dessin final (*figure 15*).

### 4.1. Interpolation bilinéaire

Il s'agit de remplir un carré de l'écran avec un dégradé régulier de couleurs. Comme nous allons procéder par divisions successives par 2, le mieux est de donner au côté $L$ du carré une longueur qui est une puissance de 2, par exemple $L = 512$.

Dans un premier temps, il s'agit de fabriquer une palette de couleurs où apparaissent tour à tour,les trois couleurs primaires, rouge, vert et bleu, avec des dégradés entre elles, comme indiqué sur la *figure 16*, avec une palette de 768 couleurs. Par exemple, de l'indice 0 à 255, la composante rouge va de 255 à 0, la verte de 0 à 255, et la bleue reste à 0. D'où le programme :

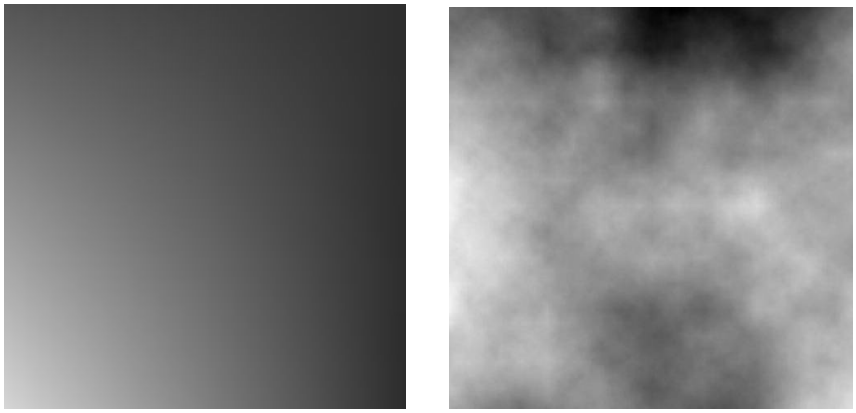

*Figure 15* : Interpolation bilinéaire à gauche, et ciel nuageux à droite


*Figure 16* : Palette cyclique de 768 couleurs, à trois dominantes.

```
for(i=0;i<768;i++)
  { if (i<256) couleur[i]=SDL_MapRGB(screen->format, 255-i, i, 0);
    else if (i<512) couleur[i]=SDL_MapRGB(screen->format, 0, 511-i, i-256);
    else if (i<768) couleur[i]=SDL_MapRGB(screen->format, i-512, 0, 767-i);
  }
for(i=0;i<768;i++) line(i,0,i,50,couleur[i]); /* dessin de la palette */
```

Passons à l'interpolation dite bilinéaire car on est en deux dimensions. Pour cela, on donne une couleur au hasard aux quatre coins du carré, ces coins étant numérotés 0, 1, 2, 3. C'est ce que fait le programme princpal suivant, avant d'appeler la fonction d'interpolation :

```
h0=rand()%768;   putpixel(0,0,couleur[h0]);     /* h_i est l'indice d'une couleur pris au hasard */
h1=rand()%768;   putpixel(L,0,couleur[h1]);    /* chaque coin du carré initial est  colorié */
h2=rand()%768;    putpixel(L,L,couleur[h2]);
h3=rand()%768;    putpixel(0,L,couleur[h3]);
interpo(0,0,L,L,h0,h1,h2,h3);
```

Reste à expliquer la fonction d'interpolation, qui assure la transition régulière des couleurs. Il s'agit de remplir l'intérieur du carré en découpant de façon récursive chaque carré en quatre carrés de côté moitié, en commençant par le carré initial. A chaque étape de ces découpages, on connaît les couleurs des quatre coins du carré concerné et l'on calcule en fonction d'elles les couleurs du centre du carré ainsi que des milieux de chaque côté. Pour le centre du carré, on fait la moyenne des quatre coins, et pour les milieux des côtés, on fait la moyenne de leurs deux extrémités. Ces moyennes vont préserver le dégradé continu et harmonieux des couleurs (*figure 17*). Puis on recommence avec les quatre petits carrés obtenus, et cela jusqu'au remplissage total du carré initial. On en déduit la fonction *interpo*(*x0,y0,x2,y2,i0,i1,i2,i3*) où les variables sont les coordonnées ($x_0$, $y_0$) et ($x_2$, $y_2$) des deux sommets opposés 0 et 2 du carré concerné, ainsi que les indices $i_0$, $i_1$, $i_2$, $i_3$ des quatre couleurs des sommets 0123.

```
void interpo(int x0,int y0,int x2,int y2,int i0, int i1, int i2, int i3)
{  int cx,cy,ic,i01,i12,i23,i30;
   if (x2-x0>1 )  /* on répète les découpages jusqu'à avoir des carrés de côté unité */
    { cx=(x0+x2)/2;cy=(y0+y2)/2;  /* centre du carré  */
      ic=(i0+i1+i2+i3)/4;  putpixel(cx,cy,couleur[ic]); /* coloriage du centre du carré */
```

```
      i01=(i0+i1)/2;  putpixel(cx,y0,couleur[i01]); /* milieu du côté 01 et coloriage*/
      i12=(i1+i2)/2;   putpixel(x2,cy,couleur[i12]); /* milieu du côté 12 */
      i23=(i2+i3)/2;   putpixel(cx,y2,couleur[i23]); /* milieu du côté 23 */
      i30=(i3+i0)/2;   putpixel(x0,cy,couleur[i30]); /* milieu du côté 30 */
      interpo(x0,y0,cx,cy,i0,i01,ic,i30);  /* rappels de la fonction interpo sur les quatre petits carrés */
      interpo(cx,y0,x2,cy,i01,i1,i12,ic);
      interpo(cx,cy,x2,y2,ic,i12,i2,i23);
      interpo(x0,cy,cx,y2,i30,ic,i23,i3);
    }
}
```

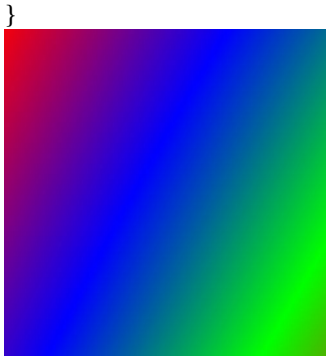

*Figure 17* : Exemple d'interpolation bilinéaire.

### 4.2. Passage au plasma

Nous allons aménager la fonction d'interpolation pour obtenir le plasma en couleurs, en rajoutant du hasard bien maîtrisé. La fonction *interpo*() précédente est remplacée par la fonction *carre*(). Le programme principal se réduit à :

```
SDL_FillRect(screen,0,couleurfond); /* on prend le blanc comme couleur de fond */
putpixel(0,0,couleur[h0]);  putpixel(L,0,couleur[h1]);
putpixel(L,L,couleur[h2]);  putpixel(0,L,couleur[h3]);
kk=0.4;  /* cette variable kk sera utilisée dans la fonction carre() pour limiter les variations de couleurs */
carre(0,0,L,L,h0,h1,h2,h3);
```

Quant à la fonction *carre*(), elle fait, comme précédemment, une moyenne des couleurs des quatre coins du carré pour colorier son centre. Mais pour les couleurs des milieux des côtés, on ne va plus se contenter de prendre la moyenne des couleurs des deux extrémités, on lui ajoute ou on lui retranche un nombre égal ou proportionnel à la longueur du côté concerné, cela avec une certaine dose de hasard. Ainsi, pour des points qui sont proches la variation de couleur est faible, et le dégradé persiste, mais pour deux points éloignés la couleur de leur milieu peut subir de grandes variations, ce qui va donner au dessin final une allure tourmentée, tout en empêchant une rupture brusque dans la coloration.

Par exemple pour le milieu du côté horizontal 01, on commence par faire la moyenne *i01* des indices des couleurs des points 0 et 1. Puis on définit la variation maximale *varmax*, égale à la longueur du côté 01. A partir de là on définit une variation *var* prise au hasard entre – *varmax*/2 et +*varmax*/2, grâce à *var* = 2*(*rand*()%(*varmax*+1)) – *varmax*. Pour éviter de trop grandes variations, on multiplie cette variation par un facteur *kk* de l'ordre de 0,3 ou 0,4, et on l'ajoute à l'indice *i01* de la couleur moyenne. Comme ces variations peuvent parfois faire sortir l'indice de couleur de la zone comprise entre les frontières 0 et 767 de la palette, tout dépassement est bloqué sur la frontière concernée. Voici ce qui est proposé pour la fonction *carre*() :

```
void carre(int x0,int y0,int x2,int y2,int i0, int i1, int i2, int i3)
{  int cx,cy, ic,i01,i12,i23,i30,varmax;   float var; Uint32 couleurc;  Uint8 iR,iG,iB;
   if (x2-x0>1 )
    {
      cx=(x0+x2)/2;cy=(y0+y2)/2;  /*  centre du carré */
```

```
        ic=(i0+i1+i2+i3)/4;  putpixel(cx,cy,couleur[ic]);
        i01=(i0+i1)/2;   /*******  i01 *******/
        varmax=x2-x0; var=2*(rand()%(varmax+1))-varmax; var=kk*var;
        i01+=var;  if(i01<0) i01=0; if(i01>767) i01=767;
        couleurc=couleur[i01];
        if (getpixel(cx,y0)==couleurfond) putpixel(cx,y0,couleurc);
        i12=(i1+i2)/2;    /*******  i12 *******/
        varmax=x2-x0; var=2*(rand()%(varmax+1))-varmax; var=kk*var; /* calcul de la variation */
        i12+=var;  if(i12<0) i12=0; if(i12>767) i12=767;  /* blocage des dépassements */
        couleurc=couleur[i12];  /* couleur du milieu du segment concerné */
        if (getpixel(x2,cy)==couleurfond)
            putpixel(x2,cy,couleurc);  /* coloriage du point s'il n'est pas déja colorié */
        i23=(i2+i3)/2;   /*******  i23  *******/
        varmax=x2-x0; var=2*(rand()%(varmax+1))-varmax; var=kk*var;
        i23+=var;  if(i23<0) i23=0; if(i23>767) i23=767;
        couleurc=couleur[i23];
        if (getpixel(cx,y2)==couleurfond) putpixel(cx,y2,couleurc);
        i30=(i3+i0)/2;  /*******  i30  *******/
        varmax=x2-x0; var=2*(rand()%(varmax+1))-varmax; var=kk*var;
        i30+=var;  if(i30<0) i30=0; if(i30>767) i30=767;
        couleurc=couleur[i30];
        if (getpixel(x0,cy)==couleurfond)putpixel(x0,cy,couleurc);

        carre(x0,y0,cx,cy,i0,i01,ic,i30);  /* rappels de la fonction carre() sur les quate petits carrés */
        carre(cx,y0,x2,cy,i01,i1,i12,ic);
        carre(cx,cy,x2,y2,ic,i12,i2,i23);
        carre(x0,cy,cx,y2,i30,ic,i23,i3);
    }
}
```

Mais ce programme donne la *figure 18*, une véritable catastrophe.

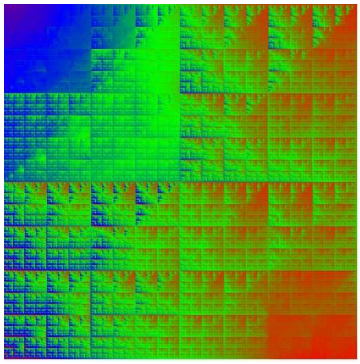

*Figure 18* : Résultat d'un programme faux.

Pourquoi ce programme ne marche pas ? Cela tient à ce qu'un même côté peut appartenir à deux carrés. Son milieu est colorié une première fois, et l'on a fait en sorte qu'il ne soit pas colorié une deuxième fois. Mais cela ne suffit pas. En effet, l'indice de couleur du milieu a été calculée une deuxième fois, et c'est lui qui est donné à l'étape suivante du programme récursif, alors qu'on doit garder l'indice obtenu la première fois, afin d'éviter toute rupture de couleur. Ainsi, lorsque le milieu d'un côté est déjà colorié, on doit récupérer sa couleur. On procède de la façon suivante, dans l'exemple du côté 01 :

```
i01=(i0+i1)/2 ;  varmax=x2-x0;  var=2*(rand()%(varmax+1))-varmax;  var=kk*var;    i01+=var;
if(i01<0) i01=0; if(i01>767) i01=767;
couleurc=couleur[i01];
if (getpixel(cx,y0)==couleurfond) putpixel(cx,y0,couleurc); /* jusqu'ici rien n'a changé */
 else     /* cas où le point a déjà été colorié une première fois */
      { couleurc=getpixel(cx,y0);  /* on récupère la couleur du point */
```

```
    SDL_GetRGB(couleurc,screen->format,&iR,&iG,&iB); /* les indices de chaque composante RGB
                                                        sont récupérés grâce à cette fonction GetRGB */
    if (iB==0) i01=(int)iG;              /* à partir des composantes RGB, on calcule son indice i01
    else if (iR==0) i01=(int)iB+256;        dans notre palette [8] */
    else if (iG==0) i01=(int)iR+512;
  }
```

En faisant de même pour chacun des autres milieux, le programme marche, avec un résultat comme sur la *figure 19 à gauche*.

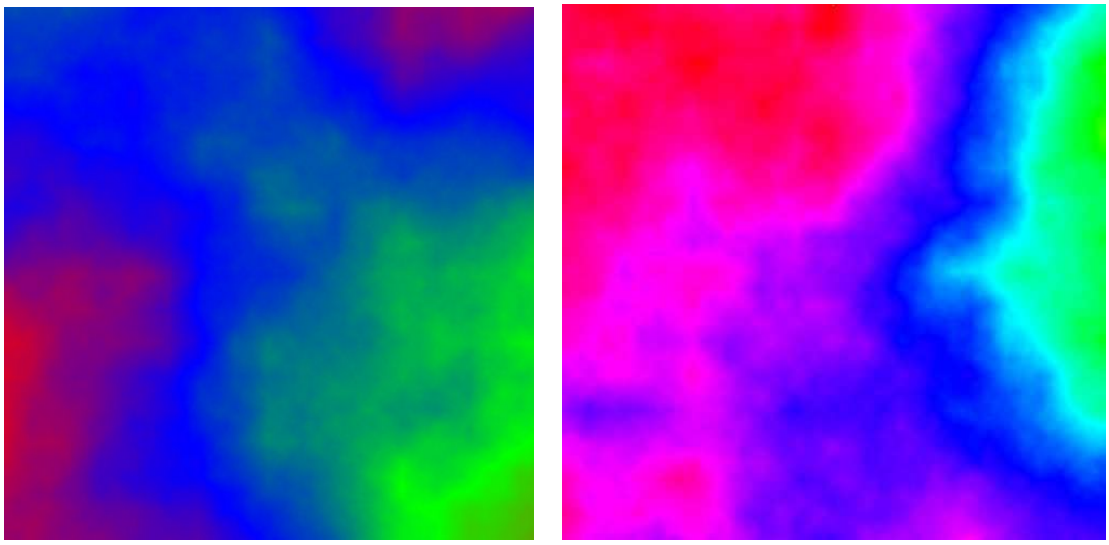

*Figure 19* : Plasma avec la palette de 768 couleurs *à gauche*, et avec la palette arc-en-ciel *à droite*.

Une fois ce dessin obtenu, il est possible d'en faire une animation, en augmentant de quelques unités, à chaque étape de temps, l'indice de couleur de chaque point du dessin, en profitant du caractère cyclique de la palette. Grâce à ce décalage cyclique, rapidement réalisé, on a l'impression d'un mouvement d'ensemble, toujours renouvelé, alors que le motif sous-jacent reste identique. C'est sans doute à cause de cela que le nom de plasma a été associé à ce mouvement perpétuel.

### *Exercice 4 : Plasma avec une palette arc-en-ciel*

*Recommencer tout cela en utilisant une palette plus large, la palette arc-en-ciel avec 1536 couleurs, définie sur la figure 20.*

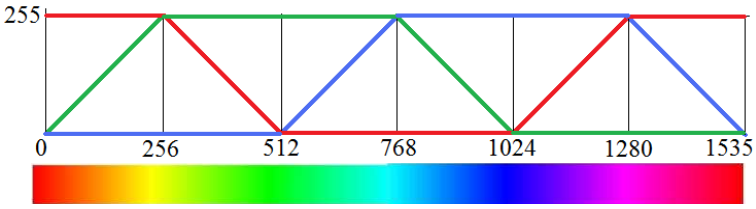


*Figure 20* : La palette arc-en-ciel.

Le graphique de la *figure 20* nous conduit à définir la palette ainsi :

```
for(i=0;i<1536;i++)
  if (i<256)  couleur[i]=SDL_MapRGB(screen->format,255,i,0);
  else if (i<512) couleur[i]=SDL_MapRGB(screen->format,511-i,255,0);
  else if (i<768) couleur[i]=SDL_MapRGB(screen->format,0,255,i-512);
```

[8] Pour comprendre, il suffit de se reporter au graphique de la *figure 16*, divisé en trois intervalles, dans chacun desquels une des composantes de couleur est nulle

```
  else if (i<1024) couleur[i]=SDL_MapRGB(screen->format,0,1023-i,255);
  else if (i<1280) couleur[i]=SDL_MapRGB(screen->format,i-1024,0,255);
  else couleur[i]=SDL_MapRGB(screen->format,255,0,1535-i);
  for(i=0;i<1536;i++)  if (i<768) line(i+10,10,i+10,100,couleur[i]); /* dessin de la palette */
  else line(i+10-768,120,i+10-768,220,couleur[i]);
```

Dans la fonction *carre*(), il convient, comme on l'a fait avec la palette de 768 couleurs, de retrouver l'indice d'une couleur de notre palette à partir de ses composantes rouge, vert, bleu. D'où cette modification du programme pour chaque milieu des côtés des carrés, le reste ne changeant pas :

```
if (getpixel(cx,y0)==couleurfond) putpixel(cx,y0,couleurc);
else
    { couleurc=getpixel(cx,y0);
      SDL_GetRGB(couleurc,screen->format,&iR,&iG,&iB);
      if (iR==255 && iB==0) i01=iG;   /* récupération de l'indice de couleur i01 */
      else if (iG==255  && iB==0) i01=511-iR;
      else if (iG==255 && iR==0) i01=iB+512;
      else if (iB==255 && iR==0) i01=1023-iG;
      else if (iB==255 && iG==0) i01=iR+1024;
      else if (iR==255 && iG==0) i01=1535-iB;
    }
```

Un résultat est montré sur la *figure 19 à droite*.

## 5. Automates cellulaires en 2 dimensions

Dans la mesure où des points ou des particules évoluent en fonction de leur voisinage, on parle d'automates cellulaires. Dans ce sens, les exemples de phénomènes naturels que nous avons vus précédemment en font partie. Mais on peut aussi définir les automates cellulaires de façon plus abstraite, en les associant à des règles précises d'évolution en fonction de leur voisinage. On doit à S. Wolfram d'avoir initié la recherche dans ce domaine.[9]

Nous donnons ici la version la plus simple relative aux automates cellulaires. A l'intérieur d'un carré sont placés des 1 et des 0 en chaque point du quadrillage. Cet univers binaire est censé représenter des cellules vivantes ou mortes, ou encore la présence ou l'absence de particules, ou bien deux états que peuvent prendre ces automates. Au départ l'ensemble des 1 peut être réduit à un point (le germe), ou être formé de plusieurs points disposés aléatoirement, ou encore être ordonné suivant une forme prédéterminée. A chaque étape de temps, l'évolution de cet univers se fait suivant des règles précises. Une cellule qui est à 0 ou à 1 va passer (ou rester) à 0 ou 1 selon la situation de son voisinage, qui peut être formé de quatre voisins ou de huit voisins (*figure 21*).

Supposons que le voisinage soit formé de quatre voisins. Une situation est définie par les chiffres 0 ou 1 de ces quatre voisins ainsi que de la cellule centrale. Cela donne $2^5 = 32$ situations possibles. A chacune de ces situations on fait correspondre une règle de passsage vers le nouvel état, à l'instant suivant, pour la cellule centrale, qui passe ou reste à 0 ou à 1. Cela fait $2^{32}$ jeux possibles. Mais on distingue deux cas particuliers.

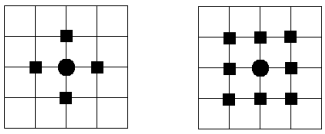

*Figure 21* : *A gauche* un voisinage-4 pour un point (le rond central), *à droite* un voisinage-8.

[9] cf. le livre *Theory and Applications of Cellular Automata* de S. Wolfram *(World Scientific 1986).*

**Premier cas :** Seule la somme des voisins intervient, ainsi que la cellule centrale. Lorsque l'on prend 4 voisins, cela donne dix situations numérotées ainsi de 0 à 9 :

0 1 2 3 4 5 6 7 8 9
**0**0 **1**0 **0**1 **1**1 **0**2 **1**2 **0**3 **1**3 **0**4 **1**4

Chaque situation est représentée par deux chiffres *ij*, avec $i + 2j$ qui est égal au numéro de cette situation. Le premier chiffre *i* correspond à l'état 0 ou 1 de la cellule centrale, le deuxième indique la somme des états des quatre cellules voisines, entre 0 et 4. Par exemple la situation 14 signifie que la case centrale est à 1 et que les quatre voisins sont aussi à 1. Pour fabriquer un jeu, on associe à chacune de ces situations un 0 ou un 1 pour définir l'état de la cellule centrale à l'instant suivant. En prenant toutes les possibilités cela fait $2^{10}$ jeux. Pour un jeu la succession de ces 0 et ces 1 peut être vue comme un nombre écrit en binaire, de longueur 10, et que l'on peut convertir en décimal. Ainsi chaque jeu a son propre code.

***Exemples***

- Code 224. Cela donne : 0 1 2 3 4 5 6 7 8 9 (numéros des situations)
  00 10 01 11 02 12 03 13 04 14 (situations)
  0 0 0 0 0 1 1 1 0 0 (code du jeu en binaire)
  $0 + 0 + 0 + 0 + 0 + 2^5 + 2^6 + 2^7 = 224$

Remarquons que le nombre en binaire est ordonné suivant les poids croissants.

Que signifie ce jeu ? En cas de surpopulation (quatre voisins présents), ou en cas d'isolement (zéro ou un voisin présent), la cellule passe ou reste à 0. Lorsqu'il y a deux ou trois voisins présents, une cellule à 1 reste à 1, et une cellule à 0 reste à 0 sauf dans le cas où il y a trois voisins auquel cas une cellule à 0 passe à 1. Cela s'appelle le jeu de la vie.

- Code 614. Il s'agit de : 00 10 01 11 02 12 03 13 04 14
  0 1 1 0 0 1 1 0 0 1
  $2 + 2^2 + 2^5 + 2^6 + 2^9$

Cela s'appelle la règle de parité : le nouvel état est la somme des états de la cellule centrale et de ses quatre voisins, ramenée modulo 2.

On peut aussi prendre un voisinage de huit voisins au lieu de quatre. Il existe alors 18 situations numérotées de 0 à 17, et $2^{18}$ jeux possibles. Comme par exemple :

- Code 261632 :

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 (numéros)
00 10 01 11 02 12 03 13 04 14 05 15 06 16 07 17 08 18 (situations)
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 (code en binaire)

Si la cellule centrale est à 0 et qu'il y a au moins cinq voisins à 1, elle passe à 1, et si la cellule centrale est à 1 et qu'il y a au plus trois voisins à 1, elle passe à 0. Elle reste inchangée dans les autres cas. Cela s'appelle la règle de la majorité.

**Deuxième cas :** Avec une cellule centrale entourée de quatre voisins, seule la somme de ces cinq cellules intervient. Cela donne 6 situations, de 0 à 5, d'où $2^6 = 64$ jeux possibles. Mais on peut aussi prendre un voisinage formé de huit cases, on trouve alors 10 situations, de 0 à 9, et $2^{10}$ jeux.

***Exemples***

- Avec un voisinage-4 : code 42

0 1 2 3 4 5
0 1 0 1 0 1

$2 + 2^3 + 2^5 = 42$. On retrouve la règle de parité.

- Avec un voisinage-8 : code 992, ce qui donne

0 1 2 3 4 5 6 7 8 9
0 0 0 0 0 1 1 1 1 1

$2^5+2^6+2^7+2^8+2^9 = 992$. On retrouve la règle de la majorité.

### *Exercice 5 : Programmation*

#### *1) Règle de la majorité*

*Partir d'une dispositon aléatoire de 0 et de 1 dans un carré de 120 sur 120. Puis utiliser le code 992 faisant intervenir la somme d'une case et de ses huit voisins. Programmer la visualisation de l'évolution du jeu, avec les cases 0 en blanc et les cases 1 en noir.*

Le code 992 est d'abord converti en binaire croissant, et ce nombre binaire placé dans un tableau *b*[]. Puis à chaque point de coordonnées (*i*, *j*) du carré, on associe un élément d'un tableau *p*[*i*][*j*] contenant 0 ou 1. Au départ, ce tableau est rempli au hasard. La fonction *dessin*() se charge de colorier les points en noir ou blanc, en pratiquant un zoom appelé *pas*. Puis à chaque étape de l'évolution, le carré est parcouru point par point, et la somme des valeurs du point et de ses huit voisins calculée. Celle-ci, par le biais du tableau *b*[], sert à trouver la nouvelle valeur au point concerné. Afin de ne pas faire d'interférence avec les anciennes valeurs, les nouvelles valeurs sont placées dans un nouveau tableau *np*[][], et quand tout le carré a fini d'être parcouru, ce nouveau tableau est envoyé dans *p*[][], ce qui permet de passer à l'étape suivante.

```
n=992;
for(i=0;i<10;i++) {b[i]=n%2;n=n/2;} /* conversion de n en binaire ascendant */
for(i=0;i<120;i++)    for(j=0;j<120;j++)  p[i][j]=rand()%2; /* remplissage aléatoire du carré écran */
dessin(); SDL_Flip(screen);pause();SDL_FillRect(screen,0,white);
for(etape=1;etape<=20;etape++)
  { for(i=0;i<120;i++) for(j=0;j<120;j++)  /* parcours du carré point par point */
       { svoisinsetcentre=p[i][j]          /* calcul de la somme des voisins */
         +p[(i-1+120)%120][j]+p[(i+1)%120][j]
         +p[(i-1+120)%120][(j+1)%120]+p[(i+1)%120][(j+1)%120]
        +p[(i-1+120)%120][(j-1+120)%120]+p[(i+1)%120][(j-1+120)%120]
        +p[i][(j-1+120)%120]+p[i][(j+1)%120];
         np[i][j]=b[svoisinsetcentre];
        }
     for(i=0;i<120;i++)  for(j=0;j<120;j++)  p[i][j]=np[i][j];
     dessin();SDL_Flip(screen);pause();SDL_FillRect(screen,0,white);
   }
```

Avec la fonction *dessin*() où chaque cellule est représentée par un carré de côté *pas* :

```
void dessin(void)
{int i,j,ii,jj;
 rectangle(0,0,pas*120,pas*120,black);
 for(i=0;i<120;i++) for(j=0;j<120;j++)   if (p[i][j]==1)  /* coloriage des cases noires */
 for(ii=pas*i;ii<pas*(i+1);ii++) for(jj=pas*j;jj<pas*(j+1);jj++)
 putpixel(ii,jj,black);
}
```

La règle de la majorité a comme effet d'agglomérer les points de même couleur, qui étaient au départ dispersés, en faisant apparaître des taches noires sur fond blanc (*figure 22*).

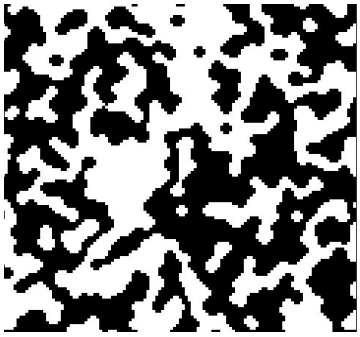

*Figure 22* : Equilibre obtenu après une diziane d'étapes, avec des taches noires sur fonc blanc.

***2) Règle de parité***

*Faire de même avec la règle de parité, dont on a vu qu'elle avait deux codes possibles, 42 ou 614.*

Si l'on prend comme code 42, il convient de prendre la somme de la case centrale et de ses quatre voisins, ce qui donne 6 situations possibles. Le programme est le même que le précédent, saut que l'on prend 4 voisins au lieu de 8 (*figure 23*).

On peut aussi prendre le code 614, où les situations consistent à ajouter la valeur de la case centrale avec celle de la somme de ses 4 voisins. Les 10 situations sont notées *ij*, et leur numéro est $i + 2j$. D'où un programme légèrement différent du précédent.

```
n=614;
for(i=0;i<10;i++) {b[i]=n%2;n=n/2;}
for(i=59;i<61;i++)    for(j=59;j<61;j++)  p[i][j]= 1;
dessin(); SDL_Flip(screen);pause();SDL_FillRect(screen,0,white);
for(etape=1;etape<=50;etape++)
  {  for(i=0;i<120;i++) for(j=0;j<120;j++)
       { svoisins= p[(i-1+120)%120][j]+p[(i+1)%120][j]
                   +p[i][(j-1+120)%120]+p[i][(j+1)%120];
         np[i][j]=b[2*svoisins+p[i][j]];
       }
     for(i=0;i<120;i++)  for(j=0;j<120;j++)  p[i][j]=np[i][j];
     if (etape%1==0) {dessin();SDL_Flip(screen);pause();SDL_FillRect(screen,0,white);}
  }
```

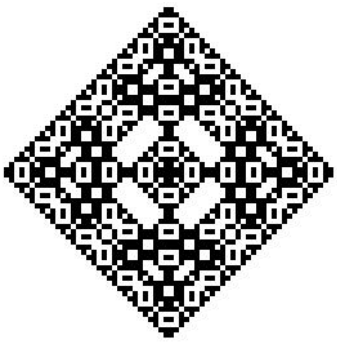

*Figure 23* : Une étape lors de l'évolution de la règle de parité.

***3) Règle du voisin unique***

*Partir d'un point, ce qui simplifie la programmation car la forme obtenue ne va faire que grossir. On applique la règle : une place vide (à 0) se remplit (se met à 1) si une et une seule des cases voisines est à 1.*

*1) Faire cela dans un voisinage-4, après avoir déterminé le code de ce jeu.*

00 10 01 11 02 12 03 13 04 14
0 1 1 1 0 1 0 1 0 1
$2 + 2^2 + 2^3 + 2^5 + 2^7 + 2^9 = 686.$

La règle du jeu fait progressivement grossir la forme au voisinage de sa circonférence, car les points déja coloriés en noir le restent définitivement. Le programme est analogue à ceux faits précédemment, le résultat est donné sur la *figure* .

*2) Faire de même dans un voisinage-8.*

00 10 01 11 02 12 03 13 04 14 05 15 06 16 07 17 08 18
0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
$2 + 2^2 + 2^3 + 2^5 + 2^7 + 2^9 + 2^{11} + 2^{13} + 2^{15} + 2^{17} = 174766$

***3)*** *Faire de même dans un voisinage hexagonal. Comment créer un tel voisinage hexagonal à partir d'une grille carrée de points ? Il suffit de prendre les points de deux en deux horizontalement et verticalement, en pratiquant en plus un décalage d'un cran toutes les deux lignes horizontales de points (figure 24).*

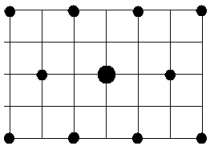

*Figure 24* : Voisinage hexagonal

A la différence des programmes précédents, qui faisaient intervenir un tableau *p*[][] avant de passer au dessin, un programme purement graphique est mieux adapté au cas actuel. A chaque étape de l'évolution, on parcourt une zone carrée de l'écran, où les points (*xx*, *ye*) sont pris de deux en deux (soit une distance enre eux de $2\times pas$), avant de faire un décalage d'un cran toutes les deux lignes, afin d'obtenir les points écran (*xe*, *ye*) du réseau hexagonal. Puis pour chacun des points non coloriés de ce réseau, on prend son voisinage hexagonal, et si l'on ne lui trouve qu'un seul voisin déjà colorié, on colorie le point. Les points ainsi coloriés (en noir dans le programme qui suit) à chaque étape ont leurs coordonnées enregristrées dans des tableaux *x*[], *y*[] avant d'être coloriés à leur tour (*figure 25*).

```
filldisc(60*pas,60*pas,pas,black); SDL_Flip(screen);pause();   /*  point initial */
for(etape=1;etape<=30;etape++)
 {compteur=0;  /* compteur des nouveaux points en noir à chaque étape */
  for(xx=2*pas;xx<120*pas;xx+=2*pas) for(ye=2*pas;ye<120*pas;ye+=2*pas)
   { if (ye%(4*pas)!=0) xe=xx+pas; else xe=xx;  /* coordonnées (xe, ye) du réseau hexagonal */
     if (getpixel(xe,ye)==white)   /* seuls les points non coloriés nous concernent */
       { cvoisin=0;
         if (getpixel(xe+2*pas,ye)!=white) cvoisin+=1;  /* comptage du nombre de voisins */
         if (getpixel(xe-2*pas,ye)!=white) cvoisin+=1;
         if (getpixel(xe+pas,ye+2*pas)!=white) cvoisin+=1;
         if (getpixel(xe+pas,ye-2*pas)!=white) cvoisin+=1;
         if (getpixel(xe-pas,ye+2*pas)!=white) cvoisin+=1;
         if (getpixel(xe-pas,ye-2*pas)!=white) cvoisin+=1;;
         if (cvoisin==1)  /* règle du voisin unique */
             { x[compteur]=xe;y[compteur++]=ye; }
       }
   }
  for(k=0;k<compteur;k++) filldisc(x[k],y[k],pas,black);
  SDL_Flip(screen);pause();  /* surtout ne pas effacer d'une étape à la suivante */
 }
```

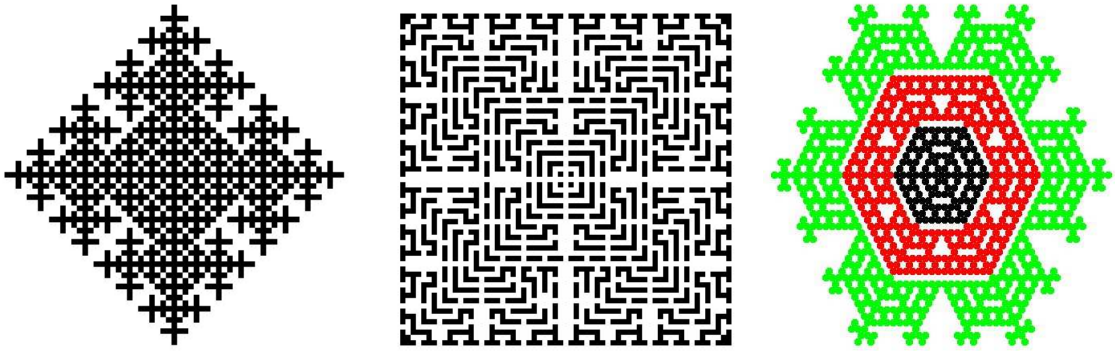

*Figure 25* : La règle du voisin unique, avec un voisinage-4 *à gauche*, un voisinage-8 *au centre*, et un voisinage hexagonal *à droite*.

***4)*** *Programmer le cas où le point central intervient, avec la somme des valeurs de ses 8 voisins, les situations étant notées ij. Essayer en particulier le code 736, à partir d'un agglomérat quelconque. On observera un mélange d'ordre et de désordre (figure 26).*

```
n=nombre;
for(i=0;i<18;i++) {b[i]=n%2;n=n/2;} /* conversion du nombre en binaire */
for(i=55;i<65;i++)    for(j=55;j<65;j++)  p[i][j]=rand()%2;  /* petit conglomérat initial de cellules */
for(etape=1;etape<=140;etape++)
  {  for(i=0;i<120;i++) for(j=0;j<120;j++)  /* parcours cyclique d'un carré de 120 sur 120 */
       { svoisins=p[(i-1+120)%120][j]+p[(i+1)%120][j]
            +p[(i-1+120)%120][(j+1)%120]+p[(i+1)%120][(j+1)%120]
            +p[(i-1+120)%120][(j-1+120)%120]+p[(i+1)%120][(j-1+120)%120]
            +p[i][(j-1+120)%120]+p[i][(j+1)%120];
         np[i][j]=b[2*svoisins+p[i][j]];
        }
      for(i=0;i<120;i++)  for(j=0;j<120;j++)  p[i][j]=np[i][j];
   }
dessin();
```

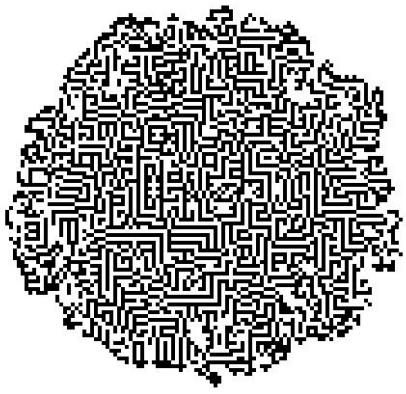

*Figure 26* : Apparition d'une forme vaguement circulaire, grossissant peu à peu, avec un certain ordonnancement dans la zone centrale.

### *Exercice 6 : Arithmétique d'un automate cellulaire en une dimension*

*Sur un segment de longueur N, dont les extrémités sont cyliques, considérer la règle simple où la valeur en une case est la somme de ses deux voisins, ramenée modulo 2. Chaque configuration peut s'écrire sous forme d'un mot de longueur N à base de 0 et de 1. Par exemple s'il y a un 1 dans les cases 0 et 2 seulement, pour N = 5, la configuration s'écrit 10100. On peut aussi bien écrire cette configuration sous la forme d'un polynôme de degré N – 1 au plus, écrit suivant les puissances croissantes. Pour 10100, le polynôme est* $1 + x^2$*, pour 11111 c'est* $1 + x + x^2 + x^3 + x^4$*.Remarquons*

*que les polynômes associés aux configurations ont tous leurs coefficients égaux à 0 ou 1 (ils sont pris modulo 2).*

*1) Partir de la configuration où le polynôme est 1 (soit un 1 dans la case 0, le reste étant à 0). Montrer que la configuration suivante s'obtient en multipliant le polynôme initial par $(x + x^{-1})$ modulo $x^N - 1$, c'est-à-dire en remplaçant $x^N$ par 1.*

A cause de la règle de voisinage, la configuration 1000...00 devient 0100...01, ou encore le polynôme initial 1 devient $x + x^{N-1} = x + x^{-1}$.

2) En gardant la même configuration initiale 1, montrer que la configuration obtenue à l'étape *n* est $(x + x^{-1})^n$. Plus généralement, en prenant une configuration initiale quelconque $P_0$, montrer que la configuration à l'étape *n* est $P_0 (x + x^{-1})^n$.

On vérifie aisément qu'une configuration quelconque *P* devient à l'étape suivante $P (x + x^{-1})$. Si la confiuration *P* a un 1 en position *i*, soit $x^i$, cele donne $x^i (x^{i-1} + x^{i+1})$, et s'il y a aussi un 1 en position *j*, cela donne $x^j (x^{j-1} + x^{j+1})$, et la nouvelle configuration s'obtient par addition, soit

$x^i (x^{i-1} + x^{i+1}) + x^j (x^{j-1} + x^{j+1})$, car le fait de travailler modulo 2 règle les problèmes d'interférences. Par exemple $1 + x^2$ devient $x^{-1} + x^3$, ou encore $x + x^{-1} + x + x^3$ en prenant séparément chacun des deux termes, et cela fait bien $x^{-1} + x^3$. On obtient donc l'évolution suivante :

$$P_0 \to P_0(x+x^{-1}) \to P_0(x+x^{-1})(x+x^{-1}) = P_0(x+x^{-1})^2 \to P_0(x+x^{-1})^3 \to \ldots$$

et à l'étape *n* : $P_0 (x + x^{-1})^n$.

*3) Pourquoi l'évolution de l'automate finit-elle par devenir périodique ?*

Parce qu'il existe un nombre fini de configurations, et qu'il est donc sûr qu'au bout d'un certain temps on va retomber sur une configuration déjà obtenue, ce qui va former une boucle parcourue indéfiniment.

*4) Développer $(x + x^{-1})^n$ suivant la formule du binôme. Puis en admettant que les combinaisons $C_{2^a}^k$ ramenées modulo 2 sont nulles pour $0 < k < 2^a$ et égales à 1 pour $k = 0$ et $2^a$,[10] calculer $(x+x^{-1})^{2^a}$ . pour $a \geq 0$. Traiter le cas où N = 7.*

$(x+x^{-1})^n = x^n + C_n^1 x^{n-2} + C_n^2 x^{n-4} + C_n^3 x^{n-6} + \ldots + x^{-n}$ . Notamment :

$(x+x^{-1})^{2^a} = x^{2^a} + x^{-2^a}$ $[x^N - 1]$

Pour $N = 7$, on a bien l'évolution

$$1 \to x + x^6 \to x^2 + x^5 \to x + x^3 + x^4 + x^6 \to x^3 + x^4 \to x^2 + x^3 + x^4 + x^5$$
$$\to x + x^2 + x^4 + x^5 \to 1 + x^2 + x^3 + x^4 + x^5 + x^6 \to x + x^6$$

[10] Pour démontrer cette propriété, partons de la relation déduite de la formule des combinaisons, soit $C_n^k = \frac{n}{k} C_{n-1}^{k-1}$ ou $k C_n^k = n C_{n-1}^{k-1}$. Appelons *g* le pdcd de *n* et *k*, soit $n = n_1 g$ et $k = k_1 g$, avec $n_1$ et $k_1$ premiers entre eux. On en déduit $k_1 C_n^k = n_1 C_{n-1}^{k-1}$. Ainsi $n_1 = n/g$ divise $k_1 C_n^k$ tout en étant premier avec $k_1$. Donc $n/g$ divise $C_n^k$. Notamment avec $0 < k < 2^a$, $C_{2^a}^k$ est divisible par $2^a$ / pgcd($2^a$, *k*) = $2^a / 2^q$ avec $q < a$ puisque $k < 2^a$, ainsi $C_{2^a}^k$ est un nombre pair, égal à 0 modulo 2.

On constate que pour les configurations numérotées à partir de la configuration 0 par les puissances de 2 égales à 1, 2, 4, 8, on trouve bien $x^{2^a} + x^{-2^a}$ . On constate aussi que dans le cas présent l'indice d'entrée dans la boucle est 1, et que la longueur de la période de la boucle est 7.

*5) On appelle sous-ordre S de 2 modulo N impair[11] la puissance de 2 d'exposant positif minimal tel que* $2^S = \pm 1$ *[N]. Par exemple pour N = 7, S est égal à 3, avec* $2^3 = 1$ *[7]. En déduire que le phénomène périodique a une période qui divise* $2^S - 1$*. Vérifier cela pour les nombres N premiers impairs 3, 5, 7, 11, 13, 17, 19, 23, 29, 31.[12]*

Comme on l'a vu au 4°, $(x + x^{-1})^{2^S} = x^{2^S} + x^{-2^S} = x^{\pm 1} + x^{\mp 1} = x + x^{-1}$. On avait $x + x^{-1}$ à l'étape 1 en partant de la configuration 1 et on retrouve cela à l'étape $2^S$. La période est donc égale à $2^S - 1$, ou à un diviseur de $2^S - 1$. On vérifie expérimentalement que la période est précisément égale à $2^S - 1$ pour ces nombres premiers :

| nombre premier | sous-ordre | période |
|---|---|---|
| 3 | 1 | 1 |
| 5 | 2 | 3 |
| 7 | 3 | 7 |
| 11 | 5 | 31 |
| 13 | 6 | 63 |
| 17 | 4 | 15 |
| 19 | 9 | 511 |
| 23 | 11 | 2047 |
| 29 | 14 | 16383 |
| 31 | 5 | 31 |

## 6. Annexe : Tracé d'équipotentielles

L'objectif est de dessiner, par exemple autour d'une forme corallienne, des courbes dites équipotentielles. Pour comprendre ce dont il s'agit, imaginons des charges électriques disposées sur une courbe fermée avec un potentiel uniforme. Celles-ci créent un champ électrique dans la périphérie de la surface, avec un potentiel qui décroît lorsque l'on s'éloigne de la surface. Les lignes d'égal potentiel sont appelées équipotentielles. Par analogie avec l'électrostatique, on retrouve cette notion de potentiel dans de nombreux phénomènes physiques, notamment le phénomène d'agrégat que l'on vient de voir, où les particules sont plus attirées dans les zones où les équipotentielles présentent des pointes ayant une forte courbure.

### 6.1. Conditions aux frontières

Dans ce genre de problème, les conditions aux frontières sont essentielles. Nous supposerons que les courbes qui constituent les frontières intérieure et extérieure de la zone concernée forment deux équipotentielles qui sont données et fixes, le potentiel $\varphi$ valant 0 sur l'une et 1 sur l'autre. Il reste à

[11] On admettra que tout nombre premier avec *N* admet un ordre *O* et un sous-ordre *S*, tels que $2^O = 1$ et $2^S = \pm 1$, ces puissances étant minimales. Il arrive soit que $S = O$ et soit que $S = O/2$. Comme 2 est premier avec tout nombre impair > 1, 2 admet un sous –ordre *S*.

[12] Pour plus de précisions, voir l'étude faite par O. Martin, A. Odyzko et S. Wolfram, intitulée *Algebraic properties of cellular automata*, dans le livre *Theory and Applications of Cellular Automata* de S. Wolfram (*World Scientific 1986*).

savoir ce qui se passe entre ces deux équipotentielles. Précisons que l'on peut aussi avoir des frontières où le potentiel $\varphi$ est connu, mais n'y est pas constant.

### 6.2. Equation de Laplace

Dans les cas les plus courants, la fonction potentiel $\varphi$ vérifie l'équation de Laplace, soit en deux dimensions avec $\varphi(x, y)$ :

$\frac{\partial^2\varphi}{\partial x^2}+\frac{\partial^2\varphi}{\partial y^2}=0$, où interviennent les dérivées secondes de $\varphi$ par rapport aux variables $x$ et $y$.

La résolution de cette équation, avec ses conditions aux frontières, donne $\varphi$ en chaque point $(x, y)$. On aura ensuite les équipotentielles en joignant tous les points ayant le même potentiel.

### 6.3. Résolution discrète de l'équation de Laplace grâce à une grille carrée

La zone où l'on cherche le potentiel est découpée suivant une grille à mailles carrées de côtés $\Delta x = \Delta y$. Un point $P$ de coordonnées $(x, y)$ de cette grille a quatre voisins dans les quatre directions : est (E), ouest (W), nord (N), sud (S). La dérivée est assimiliée à une différence finie, en bonne approximation si $\Delta x$ est suffisamment petit. Ainsi $\frac{\partial\varphi}{\partial x}$ est proche de $\frac{\varphi_E-\varphi_P}{\Delta x}$ à droite et de $\frac{\varphi_P-\varphi_W}{\Delta x}$ à gauche. Comme ces deux valeurs ne sont pas égales en général, on fait une moyenne en prenant $\frac{\varphi_E-\varphi_W}{2\Delta x}$.

Passons à la dérivée seconde $\partial^2\varphi/\partial x^2$, considérée comme la variation de la dérivée première rapportée à $\Delta x$ : $\frac{\partial^2\varphi}{\partial x^2}=\frac{\partial}{\partial x}(\frac{\partial\varphi}{\partial x})=(\frac{\varphi_E-\varphi_P}{\Delta x}-\frac{\varphi_P-\varphi_W}{\Delta x})\frac{1}{\Delta x}=\frac{\varphi_E-2\varphi_P+\varphi_W}{\Delta x^2}$

L'équation de Laplace devient :

$\varphi_E-2\varphi_P+\varphi_W+\varphi_N-2\varphi_P+\varphi_S=0$, soit

$\varphi_P=\frac{1}{4}(\varphi_E+\varphi_W+\varphi_N+\varphi_S)$

Ainsi la valeur du potentiel en un point $P$ s'obtient simplement en faisant la moyenne du potentiel de ses quatre voisins.

On est ainsi ramené à résoudre un système d'équations, comme dans l' exemple de la *figure 27* où les frontières sont un carré de potentiel 0 ainsi que le point central de potentiel 1. Les huit points intérieurs de la grille ont un potentiel $\varphi$ inconnu, mais qui obéit aux huit équations :

$$\begin{cases}\varphi_{11}=\frac{1}{4}\varphi_{12}+\frac{1}{4}\varphi_{21}\\ \varphi_{12}=\frac{1}{4}\varphi_{11}+\frac{1}{4}\varphi_{13}+\frac{1}{4}\\ \varphi_{13}=\frac{1}{4}\varphi_{12}+\frac{1}{4}\varphi_{23}\\ \ldots\ldots\ldots\end{cases}$$

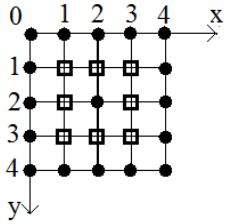


*Figure 27* : Le potentiel est connu aux frontières (ronds noirs) et inconnus aux huit points représentés par de petits carrés.

### 6.4. Méthodes pour résoudre un système d'équations

Les méthodes classiques, comme celle du pivot de Gauss, ou la *LU* (*lower upper*) décomposition de Khaletski (ou Cholevsky) deviennent lourdes dès que le nombre des équations et des inconnues dépasse quelques centaines. Aussi préfère-t-on des méthodes itératives qui travaillent par approximations successives, en se rapprochant peu à peu de la solution.[13] On distingue deux méthodes itératives, celle de Jacobi-Richardson, et celle de Gauss-Seidel. La seconde fonctionnant, de façon inespérée, en mode séquentiel, c'est elle que nous choisissons.

La méthode de Gauss-Seidel consiste, lors du parcours de la grille de gauche à droite en allant de haut en bas, à remplacer les $\varphi_{ij}^{(k-1)}$ au point *i, j* à l'étape $k-1$ par les nouveaux $\varphi_{ij}^{(k)}$ aussitôt, sans attendre la fin du parcours. Dans la grille de la *figure 28*, toute une partie a déjà été calculée pour l'étape *k* quand ce qui reste à parcourir est encore à l'étape $k-1$.

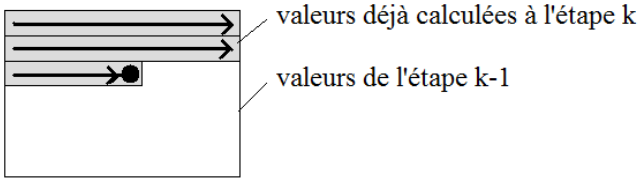


*Figure 28* : Calcul séquentiel par la méthode de Gauss-Seidel.

Cela donne des équations de la forme :

$$\varphi_{i,j}^{(k)} = \frac{\varphi_{i-1,j}^{(k)} + \varphi_{i+1,j}^{(k-1)} + \varphi_{i,j-1}^{(k)} + \varphi_{i,j+1}^{(k-1)}}{4}$$

Que se passe-t-il exactement ? Progressivement, les valeurs exactes et fixes des frontières, les seules à être justes au début du processus, diffusent à l'intérieur de la grille à partir des points situés près des frontières. Cette évolution vers des valeurs de plus en plus proches des valeurs exactes est plus ou moins rapide.

On peut accélérer le processus de convergence en ajoutant un phénomène de relaxation. On se donne un certain facteur de relaxation *r*. On commence par le même calcul que précédemment, et l'on note $\varphi_{ij}^{(k)*}$ le résultat. Puis on renforce le changement en prenant :

$$\varphi_{i,j}^{(k)} = \varphi_{i,j}^{(k-1)} + r(\varphi_{i,j}^{(k)*} - \varphi_{i,j}^{(k-1)})$$

Si l'on fait $r = 1$, on obtient $\varphi_{ij}^{(k)} = \varphi_{ij}^{(k)*}$, et l'on retrouve la méthode de Gauss-Seidel sans relaxation. Si l'on fait $r = 2$, on trouve $\varphi_{ij}^{(k)} = 2\,\varphi_{ij}^{(k)*} - \varphi_{ij}^{(k-1)}$, ce qui provoque un effet de *boost*

[13] C'est d'autant plus appréciable si la figure subit de petites variations, quelques itérations suffisent pour avoir les nouvelles équipotentielles

souvent trop élevé, dans le rapport deux pour un. Aussi prend-on en général une valeur de *r* intermédiaire, comme 1,6. La rapidité de la convergence s'en trouve nettement augmentée.

### 6.5. Algorithme

Prenons l'exemple d'une frontière extérieure carrée avec deux obstacles intérieurs en forme de cercle. La grille carrée est numérotée de 0 à *L* horizontalement, ainsi que verticalement vers le bas. Les valeurs du potentiel sont en fait des indices de couleurs prises dans une palette, par exemple entre 0 et 767. Pour tous les points de la bordure externe le potentiel *icolor*[*i*][*j*] est mis à la valeur extrême 767, dans les deux disques il est mis à 0, et ailleurs il est mis au hasard entre 0 et 767. On a ainsi déterminé les conditions initiales.

Puis on lance une boucle, où à chaque étape on parcourt l'intérieur du carré avec *i* et *j* entre 1 et *L* − 1, en évitant la bordure ainsi que les deux disques qui conservent leur potentiel. En chacun des points on fait la moyenne des quatre voisins, avec l'adjonction d'une relaxation. Pour éviter les erreurs d'arrondi lors des divisions par 4, on prend les indices de couleurs en flottants, avnt de les convertri en entiers pour le dessin. Les points de l'intérieur sont ainsi coloriés avec une couleur proportionnelle à la valeur du potentiel. L'évolution du dessin permet de juger de l'avancée de la convergence. Lorsque la figure reste stable, on arrête la boucle. Avec une grille *icolor*[*i*][*j*] de 400 sur 400 nombres flottants, le processus demande en gros 500 étapes avec une relaxation forte. On aboutit au programme suivant :

```
palette(); /* on utilise ici une palette de 768 couleurs, numérotées de 0 à 767 */
rf=1.8;   /* facteur de relaxation */
for(i=1;i<L;i++)   for(j=1;j<L;j++) icolor[i][j]=rand()%768; /* couleurs au hasard */
rectangle(0,0,L,L,couleur[767]);  /* couleur 767 sur la bordure carrée */
j=0;for(i=0;i<=L;i++) icolor[i][j]=767;
j=L;for(i=0;i<=L;i++) icolor[i][j]=767;
i=0;for(j=9;j<=L;j++) icolor[i][j]=767;
i=L;for(j=9;j<=L;j++) icolor[i][j]=767;
cx1=2.*L/3.;cy1=L/4; cx2=L/3;cy2=2*L/3.; /* centres des deux cercles */
filldisc(cx1,cy1,50,couleur[0]); /* disques de rayon 50 mis à la couleur 0 */
filldisc(cx2,cy2,50,couleur[0]);
for(i=1;i<L;i++)   for(j=1;j<L;j++)  if (getpixel(i,j)==couleur[0])
icolor[i][j]=0;
dessin();SDL_Flip(screen);pause();
for(etape=1;etape<=500;etape++)  /* boucle des étapes */
  { for(i=1;i<L;i++)for(j=1;j<L;j++)
    if ((i-cx1)*(i-cx1)+(j-cy1)*(j-cy1)> 50*50 && (i-cx2)*(i-cx2)+(j-cy2)*(j-cy2)> 50*50)
      {oldicolor[i][j]=icolor[i][j];
       icolor[i][j]=(icolor[i+1][j]+icolor[i-1][j]+icolor[i][j+1]+icolor[i][j-1])/4.; /* moyenne */
       icolor[i][j]=oldicolor[i][j]+rf*(icolor[i][j]-oldicolor[i][j]); /* relaxation */
       if (icolor[i][j]>767) icolor[i][j]=767; /* en cas de débordement */
       else if (icolor[i][j]<1) icolor[i][j]=1;
      }
   dessin();SDL_Flip(screen);
  }

void dessin(void)
{ int i,j;
  for(i=0;i<=L;i++) for(j=0;j<=L;j++)  putpixel(i,j,couleur[(int)icolor[i][j]]);
}
```

Les résultats sont donnés sur la *figure 29*, et sur la *figure 30* pour une forme coralienne.

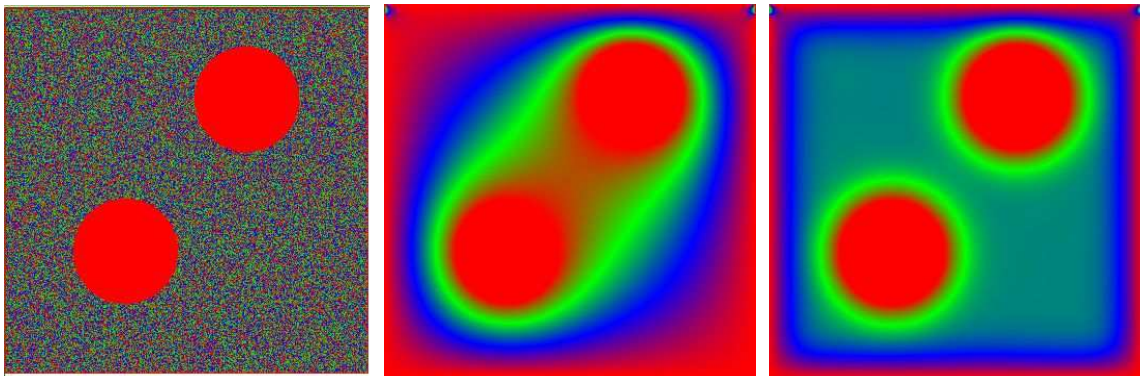

*Figure 29* : *A gauche*, le dessin initial, *au centre* l'équilibre obtenu après 500 itérations et un facteur de relaxation de 1,8, *à droite* le résultat obtenu sans relaxation après 500 itérations, pour constater qu'on on est loin de l'équilibre, seulement obtenu au bout de 3000 itérations.

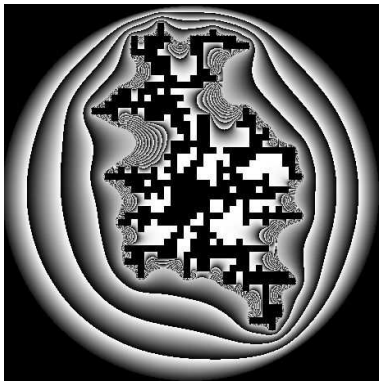

*Figure 30* : Equipotentielles en niveaux de gris autour d'une forme corallienne, la frontière externe étant un cercle.

Il convient de bien régler le facteur de relaxation *r*. En le prenant trop grand, de l'ordre de 2 et des poussières, la vitesse est forcée au point que le processus s'emballe et que l'on rate la convergence. On asssite alors à une explosion de la divergence. Au début les équipotentielles se forment dans la direction sud-est, ce qui est normal puisque le carré est parcouru de gauche à droite et de haut en bas. Au fil des itérations, les défauts se multiplient, et donnent lieu à des figures évoquant nombre de phénomènes naturels, mais qui ici n'ont aucune réalité sinon d'indiquer que la méthode est devenue fausse (*figure 31*).

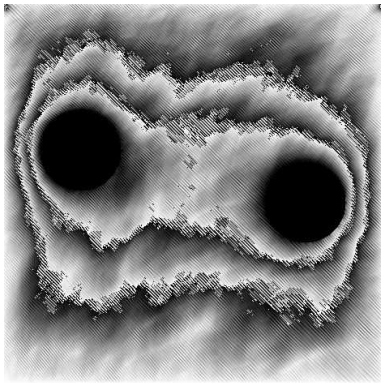

*Figure 31* : Explosion de la divergence, pour deux disques et une frontière carrée.