Console de Jeu

Manuel de Programmation

version 15

0. Introduction

Ceux qui ont aimé faire des scripts pour le bot savent que le type de jeu qu'on peut réaliser avec un bot est assez limité puisqu'il ne permet que d'afficher des lignes de texte sur le chat pour faire par exemple un quizz, un scrabble, un jeu de dés, ...

Avec cette console de jeu, qui est en réalité une fenêtre animée semblable à celle d'un gameboy, nous offrons une bien plus large palette de possibilités pour créer des applications :

- fenêtre graphique de 640 x 480 en haut de l'écran de chat
- 128 objets graphiques déplaçables à l'écran
- contrôle de l'application avec la souris
- synthétiseur pour jouer des mélodies ou des bruits (tir, explosion, ..)

1. Comment écrire une application console

Commencez par télécharger le kit console ici : **console.zip**

Ce fichier zip contient :

- **run.exe** : le compilateur (il permet de convertir un script **program.txt** en **program.run** pour le faire tourner)
- **image_converter.exe** : le convertisseur d'image (il permet de convertir un .gif de max 16 couleurs en format image pour script, voir au chapitre "Les Sprites")
- **image_converter_256.exe** : le convertisseur d'image (il permet de convertir un .gif en format image 256 couleurs pour script, voir au chapitre "Les Sprites")
- **wave_converter.exe** : le convertisseur de forme d'onde (il permet de convertir un .wav de forme d'onde, voir au chapitre "Le Synthétiseur")
- **casse-brique.txt**, **galaxy-game.txt**, **pacman-game.txt**, **falcon-game.txt** : quelques exemples de scripts console.

Pour écrire une application, il faut écrire un script (par exemple **program.txt**) et ensuite, dans Windows Explorer, avec la souris, tirer et faire glisser **program.txt** sur le programme **run.exe**

Le script **program.txt** sera alors compilé et transformé en un programme exécutable **program.run** de très petite taille (< 63K) qui pourra être injecté directement dans le chat sans occasionner de ralentissements (**voir au chapitre 9 comment le bot peut lancer une application console**). Un fichier **program.lis** est également généré avec un listing du programme. Celui-ci peut être consulté en cas d'erreurs lors de l'exécution.

Le langage script est le même que celui utilisé pour le bot (**voir l'annexe 6 pour une description du langage script** et **l'annexe 7 pour un cours d'apprentissage du langage script**).

Les commandes et l'environnement de programmation de la console sont décrits dans les chapitres suivants.

2. La fenêtre

La fenêtre de la console a une taille maximale de 640 x 480.

Si votre application est destinée à être intégrée au chat, vous avez tout intérêt à ne pas utiliser cette taille sinon il ne restera plus de place pour le texte ou les webcams, surtout chez les utilisateurs qui ont un écran en résolution basse !

La commande **screen** permet de spécifier quelle taille d'écran vous souhaitez utiliser.

screen (largeur, hauteur);

- changer la taille de la fenêtre console.
- la taille maximale est de 640 x 480.
- cette commande est obligatoire au début du programme pour ouvrir une fenêtre sinon on ne voit rien !
- vous pouvez utiliser cette commande plusieurs fois au cours du programme pour modifier la taille de la fenêtre.
- une taille de 0 x 0 fait disparaitre la fenêtre sur le chat, le programme continue cependant à tourner.

Exemple:

```
proc main ()
{
   screen (160, 16);       // mettre l'écran en 160 x 16
   print ("Bonjour !");     // dire bonjour
   sync ();                // afficher à l'écran
   sleep (5000);           // attendre 5 secondes
}
```

La commande **justify** permet de justifier la fenêtre console à gauche, au centre, à droite, en haut ou en bas de l'écran chat.

justify (n);

- permet de justifier la fenêtre console par rapport à l'écran chat.

- pour n : 0 = haut gauche, 1 = haut centre, 2 = haut droite, 3 = bas gauche, 4 = bas centre, 5 = bas droite.
- par défaut, on justifie en haut gauche.

3. Le mode texte

Par défaut, l'écran texte est composé de 40 colonnes et 30 lignes (les colonnes sont numérotées de 0 à 39 et les lignes de 0 à 29).

Si vous faites des essais vous allez constater qu'il existe encore 2 colonnes cachées à droite et 2 lignes cachées en bas : elles servent à préparer du texte et à le faire apparaitre doucement à l'aide de scrolling.

Voici les commandes du mode texte :

font (largeur, hauteur);

- changer la taille de la police de caractères, ce qui change aussi le nombre de colonnes et de lignes.
- La taille par défaut est 16 x 16.
- Seules les valeurs 8, 16 et 32 sont autorisées !
- Voici les tailles les plus courantes :

font	colonnes	lignes	colonnes cachées	lignes cachées
8 x 8	80	60	4	4
16 x 16	40	30	2	2
32 x 32	20	15	1	1
8 x 16	80	30	4	2
8 x 32	80	15	4	1
...				

print ("texte");

- affiche du texte à l'écran.

sync ();

- attend 1/60 de seconde puis envoie l'écran console à la carte graphique.
- Cette commande est la plus importante de toutes car sans elle, rien ne s'affiche à l'écran !
- Chaque fois que vous avez fait une série de print et que vous voulez que ça s'affiche vous devez faire un sync();

ink (couleur);

- choisit une couleur d'écriture texte (voir liste des couleurs à l'annexe 2).
- Ink signifie 'encre' en anglais.

paper (couleur);

- choisit une couleur de fond (voir liste des couleurs à l'annexe 2).
- Paper signifie 'papier' en anglais.

at (C,L);

- déplace le curseur (là où on écrit avec print) en colonne C, ligne L.

nl ();

- commence une nouvelle ligne.

scroll (L, X, Y);

- permet un scrolling horizontal/vertical/oblique d'une ou plusieurs lignes.
  L : ligne
  X : scroll_x 0 à 32
  Y : scroll_y 0 à 32
- On ne peut scroller que des lignes de texte entières.

blit (C1, L1, nb_cols, nb_lines, C2, L2);

- permet de déplacer un bloc de texte (C1, L1) vers (C2, L2)
  C1 : colonne début
  L1 : ligne début
  nb_cols : nbre de colonnes du bloc
  nb_lines : nbre de lignes du bloc
  C2 : colonne finale
  L2 : ligne finale
- voir annexe 3 pour des exemples

fill (C, L, nb_cols, nb_lines, code);

- remplir un bloc de texte qui commence à (col, line) avec un code. C : colonne début
  L : ligne début
  nb_cols : nbre de colonnes du bloc
  nb_lines : nbre de lignes du bloc
  pour code, voir la liste des codes à l'annexe 1.
- par ex: pour effacer l'écran : paper(15); fill (0,0, 42, 32, 32);
- voir annexe 3 pour des exemples

// Exemple:

```
proc main()
```

```
{
   screen (640, 200);

   ink (56);     // écriture bleue foncé
   paper (190);  // fond bleu clair
   font (8, 8);

   // effacer tout l'écran (80 colonnes, 60 lignes)
   // comme ça on voit le fond bleu clair.
   fill (0, 0, 80, 60, 32);         // 32 = code du caractère 'blanc'

   print ("Bonjour !  Comment allez-vous !");
   nl ();
   nl ();

   print ("Quel beau temps aujourd'hui !");
   nl ();

   at (20, 10);     // aller en colonne 20, ligne 10.
   ink (23);        // écriture rouge
   print ("Oué !");

   sync();
   sleep (10000);
}
```

4. Interroger la souris

mouse (X, Y, B);

- cette commande remplit les variables :
  X, Y : la position de la souris
  B : état du bouton (0 = non appuyé, 1 = appuyé)
- si X=0 et Y=0, la souris est dans le coin supérieur gauche de l'écran.
  si X=639 et Y=479, la souris est dans le coin inférieur droit de l'écran.
  si X=-32000 et Y=-32000, la souris est hors de la fenêtre.
- attention sur le chat la position (X,Y) peut déborder hors de la fenêtre console ou devenir négative.

```
// Exemple:

proc main()
{
   var x, y, b;

   screen (640, 200);

   for (;;)
   {
      mouse (x, y, b);
      at (0,0);
      print ("x=" + str$(x) + "  y=" + str$(y) + " b=" + str$(b) + "                    ");
      sync();
```

```
   }
}
```

5. Imprimer des codes caractères à l'écran

256 codes caractères sont prédéfinis (voir le tableau à l'annexe 1).

Pour imprimer par exemple le caractère de code 1 (un smiley), utilisez :

```
print (chr$(1));
```

pour imprimer les codes 1, 2 et 3, utilisez :

```
print (chr$(1));  print (chr$(2));  print (chr$(3));
```

ou mieux :

```
print (chr$(1) + chr$(2) + chr$(3));
```

ou encore mieux :

```
print (chr$(1,2,3));
```

5.1. Créer sa propre police de caractères

Si vous n'aimez pas les caractères prédéfinis, vous pouvez créer vos propres caractères !

Sur l'image ci-dessous :

- les X sont dessinés dans la couleur ink.
- les blancs sont dessinés dans la couleur paper.

Chaque caractère occupe seulement 8 bytes de mémoire.

```
// Exemple:

image mes_lettres (8 * 8)
{
   "  XXXX  "
   " XX  XX "
   " XX  XX "
   " XXXXXX "
   " XX  XX "
   " XX  XX "
   " XX  XX "
   "        "

   " XXXX   "
   " XX  XX "
```

```
   " XX  XX "
   " XXXXXX "
   " XX  XX "
   " XX  XX "
   " XXXXX  "
   "        "

   "  XXX   "
   " XX  XX "
   " XX     "
   " XX     "
   " XX     "
   " XX  XX "
   "  XXXX  "
   "        "
}

proc main ()
{
   screen (640, 480);

   // redéfinir les caractères 65 à 67 (A à C)
   define_font (65, 67, mes_lettres);

   // essayons nos nouveaux caractères
   print ("ABC");

   sync();

   // attendre 10 secondes
   sleep (10000);
}
```

5.2. Créer des caractères de haute qualité

Pour obtenir une police de meilleure qualité, on peut dessiner le caractère en 16 x 16 points.

Notez que l'on peut créer des caractères en 8 x 8, 16 x 16 ou 32 x 32, ou bien des mélanges de ces tailles (8 x 16, 16 x 8, ..).

En 16x16, chaque caractère occupe 32 bytes de mémoire,
en 32x32, chaque caractère occupe 128 bytes de mémoire.

Evitez de gaspiller la mémoire inutilement.

// Exemple:

```
image ma_lettre (16 * 16)
{
   "    XXXXXXX    "
   "  XXXXXXXXXX   "
   "  XXX    XXX   "
   "  XXX    XXX   "
   "  XXX    XXX   "
   "  XXX    XXX   "
```

```
   "  XXXXXXXXXX  "
   "  XXXXXXXXXX  "
   "  XXX   XXX  "
   "  XXX   XXX  "
   "  XXX   XXX  "
   "  XXX   XXX  "
   "  XXX   XXX  "
   "  XXX   XXX  "
   "  XXX   XXX  "
   "          "
}

proc main ()
{
   screen (640, 480);

   // redéfinir le caractères 65 (A)
   define_font (65, 65, ma_lettre);

   // essayons notre nouveau caractère
   print ("A");

   sync();

   // attendre 10 secondes
   sleep (10000);
}
```

5.3. Créer des caractères en 4 couleurs

Jusqu'à présent, chaque caractère n'était composé que de 2 couleurs (ink et paper). Il est possible de dessiner des caractères avec 2 couleurs supplémentaires, ils prennent alors 2 fois plus de mémoire.

Sur l'image ci-dessous :

- les X sont dessinés dans la couleur ink.
- les blancs sont dessinés dans la couleur paper.
- les 1 sont dessinés dans la première couleur palette.
- les 2 sont dessinés dans la deuxième couleur palette.

Chaque caractère 8x8 en 4 couleurs occupe 16 bytes de mémoire.

// Exemple:

```
image mon_A (8 * 8, 4 colors)
{
   "  XXXX1 "
   " XX1 XX1"
   " XX1 XX1"
   " XXXXXX1"
   " XX2 XX2"
   " XX2 XX2"
   " XX2 XX2"
```

```
   "       "
}

data ma_palette
{
   24  40        // mes 2 couleurs palette
}

proc main ()
{
   screen (640, 480);

   // redéfinir le caractères 65 (A)
   define_font (65, 65, mon_A, ma_palette);

   // essayons notre nouveau caractère
   print ("A");

   sync();
   sleep (10000);
}
```

5.4. Créer des caractères en 16 couleurs

On peut augmenter le nombre de couleurs jusqu'à 16, chaque caractère 8x8 en 16 couleurs occupe alors 32 bytes de mémoire !

L'utilité principale est de pouvoir créer un décor de fond comme on crée un puzzle, en réutilisant plusieurs fois les mêmes pièces.

Sur l'image ci-dessous :

- les X sont dessinés dans la couleur ink.
- les blancs sont dessinés dans la couleur paper.
- les chiffres 1 à 9 sont dessinés dans les couleurs palette 1 à 9.
- les lettres A à E sont dessinées dans les couleurs palette 10 à 14.

```
// Exemple:

image mon_A (8 * 8, 16 colors)
{
   "abXXXX1b"
   "aXX1cXX1"
   "aXX1dXX1"
   "aXXXXXX1"
   "aXX2eXX2"
   "aXX2eXX2"
   "aXX2eXX2"
   "aaaaaaaa"
}

data ma_palette
{
   24 40 15 0 7 46 78 35 90 126 12 3 15 8  // mes 14 couleurs palette
}
```

```
proc main ()
{
   screen (640, 480);

   // redéfinir le caractères 65 (A)
   define_font (65, 65, mon_A, ma_palette);

   // essayons notre nouveau caractère
   print ("A");

   sync();
   sleep (10000);
}
```

5.5. Créer des caractères en 256 couleurs

Finalement, on peut augmenter le nombre de couleurs jusqu'à 256, chaque caractère 8x8 en 256 couleurs occupe alors 64 bytes de mémoire.

Sur l'image ci-dessous :

- les blancs sont dessinés dans la couleur paper.
- les valeurs hexadécimales 01 à FF représentent les couleurs de 1 à 255.

// Exemple:

```
image mon_A (8 * 8, 256 colors)
{
   "0101010101010101"
   "01          01"
   "01          01"
   "0101010101010101"
   "01          01"
   "01          01"
   "01          01"
   "01          01"
}


proc main ()
{
   screen (640, 480);

   // redéfinir le caractères 65 (A)
   define_font (65, 65, mon_A);

   // essayons notre nouveau caractère
   print ("A");

   sync();
   sleep (10000);
}
```

5.6. Revenir à la police par défaut

Pour annuler vos nouveaux caractères et remettre la police par défaut :

```
// revenir au défaut pour les caractère 65 à 67 (A à C)
define_font (65, 67);
```

6. Les Sprites

Les sprites sont des objets graphiques déplaçables sur l'écran. Ils permettent de créer des figurines, des bonhommes, des missiles ou tout autre objet faisant partie de l'animation.

Numéros

Vous disposez d'un maximum de 128 sprites qui portent les numéros 0 à 127.

Priorité

Les sprites ont une priorité d'affichage : si deux sprites se superposent à l'écran, le sprite de plus petit numéro cachera l'autre (sprite 1 devant sprite 2, sprite 2 devant sprite 3, etc ..). Tous les sprites sont affichés devant le décor texte.

Image

Les programmes **image_converter.exe** et **image_converter_256.exe** du .zip que vous avez téléchargé permet, en glissant un fichier gif sur lui, de convertir l'image en format image directement exploitable dans un script (voir la commande sprite_image et l'exemple ci-dessous)

image_converter.exe permet de convertir une image de 2, 4 ou 16 couleurs, tandis que **image_converter_256.exe** crée toujours une image de 256 couleurs.

Voici les commandes pour contrôler les sprites :

sprite_on (numéro_sprite, 0 ou 1);

- allume(1) ou éteint(0) le sprite.
- Au début les sprites sont éteints.
- pour allumer le sprite 5, faites : sprite_on (5, 1); pour l'éteindre faites : sprite_on (5, 0);

sprite_pos (numéro_sprite, position_x, position_y);

- positionne le sprite sur l'écran.
- X et Y spécifient le coin supérieur gauche de l'image du sprite.
- X varie entre 0 et 639, Y entre 0 et 479.
  Vous pouvez cependant aller en dehors de ces limites et même utiliser des coordonnées négatives (par ex: -20) pour afficher un sprite partiellement en dehors de l'écran.

sprite_image (numéro_sprite, image);
ou
sprite_image (numéro_sprite, image, palette);

- spécifie l'image du sprite
- contrairement aux polices de caractère, l'image d'un sprite n'est pas limitée aux tailles 8x8 ou 16x16. Vous pouvez spécifier n'importe quelle taille avec cependant un maximum de 248 x 248.
- des images de 2, 4, 16 ou 256 couleurs sont possibles comme pour les polices de caractères.
- la palette est obligatoire pour les images en 4 ou 16 couleurs.
- changer l'image d'un sprite avec la commande sprite_image permet d'animer celui-ci.
- voir ci-dessous pour un exemple

sprite_color (numéro_sprite, couleur);

- spécifie la couleur X de l'image du sprite
- la couleur X de l'image d'un sprite peut être changée à n'importe quel moment. Les autres couleurs (1, 2, ..) sont fixes, à moins de changer d'image bien sûr.

Exemple

```
// Exemple de sprite déplacé avec la souris :

image balle (15 * 15, 4 colors)
{
   "    XXXXXX    "
   "  XX11111XX   "
   "  XX1111111XX  "
   " XX112222211XX "
   " XX112222211XX "
   "XX11222222211XX"
   "XX11222222211XX"
   "XX11222222211XX"
   "XX11222222211XX"
   "XX11222222211XX"
   " XX112222211XX "
   " XX112222211XX "
   "  XX1111111XX  "
   "   XX111111XX  "
   "    XXXXXX    "
}

data couleur_balle
{
   24 56   // rouge & bleu
}

proc main ()
{
   var x, y, b;

   screen (640, 480);

   // allumer le sprite 5
   sprite_on (5, 1);
```

```
   // choisir l'image du sprite 5
   sprite_image (5, balle, couleur_balle);

   // choisir la couleur des X sur l'image balle
   sprite_color (5, 0);    // noir

   for (;;)
   {
      mouse (x, y, b);

      // positionne le sprite à l'écran
      sprite_pos (5, x, y);

      sync ();
   }
}
```

sprite_zoom (numéro_sprite, zoom_x, zoom_y);

- permet d'agrandir ou de réduire la taille d'un sprite
- des valeurs de zoom entre -8 et +8 sont autorisées :

zoom	agrandissement
0	aucun
1	X 2
2	X 4
3	X 8
4	X 16
5	X 32
6	X 64
7	X 128
8	X 256

zoom	réduction
0	aucune
-1	div 2
-2	div 4
-3	div 8
-4	div 16
-5	div 32
-6	div 64
-7	div 128
-8	div 256

- Pour doubler la hauteur du sprite 5, tapez : sprite_zoom (5, 0, 1);
  Pour doubler la largeur du sprite 5, tapez : sprite_zoom (5, 1, 0);
  Pour doubler hauteur et largeur du sprite 5 : sprite_zoom (5, 1, 1);

c = sprite_collision (numéro_sprite);

- permet de tester si un sprite est superposé à un autre sprite ou si un sprite est superposé à un code

caractère d'une couleur plus grande que 207.

- la commande sprite_collision doit être utilisée après la commande sync() car elle détecte les collisions qui ont eu lieu lors de la dernière commande sync().
- sprite_collision(n) envoie 0 (zéro) si le sprite n n'est pas entré en collision.
- sprite_collision(n) envoie la valeur du plus grand numéro de sprite qui est entré en collision avec lui. Cependant, il ignore toute collision avec un sprite de numéro inférieur à lui-même !

```
Exemple:
   n = sprite_collision(2);
   if (n > 0)
     print ("collision entre sprite 2 et sprite " + str$(n));
```

- Si vous écrivez un jeu de sprites avec 1 joueur et 100 ennemis, vous avez intérêt à donner au joueur le plus petit numéro de sprite parce que sprite_collision ne détecte que les collisions avec des numéros de sprites plus grands que lui-même !
- sprite_collision permet également de détecter les collisions avec un code caractère du décor, pour autant que ce code caractère soit imprimé avec une couleur plus grande que 207 !
  A l'annexe 2, vous voyez que seules les couleurs 0 à 207 sont définies par défaut. Vous pouvez cependant définir les couleurs supplémentaires 208 à 255 avec la commande define_color décrite ci-après.
- En créant un décor dont la zone défendue est colorée en une couleur (par exemple 208), vous pouvez détecter facilement si un sprite joueur a "touché" le décor. sprite_collision renvoie dans ce cas la couleur du décor que le sprite a touché (donc par exemple 208).

define_color (couleur, R, G, B);

- couleur : numéro de couleur à définir (0 à 255)
  R : quantité de rouge (0 à 255)
  G : quantité de vert (0 à 255)
  B : quantité de bleu (0 à 255)

7. Les listes de données

Les listes de données permettent de stocker des longues listes de valeurs que le programme peut lire quand il le souhaite. Ces listes peuvent contenir des codes graphiques, des notes de musique, ou toute autre information.

Les valeurs autorisées dans une liste de données sont :

- des nombres entre 0 et 255
- des strings (par exemple "ABC" va stocker les 3 bytes ascii 65, 66, 67)

On utilise la commande **peek** pour lire ces listes :

// Exemple :

```
data mes_valeurs
{
   2 5 12 10 250 0      // voici une liste de 6 données (obligatoirement entre 0 et 255)
}
```

```
proc main ()
{
   var n, i;

   screen (640, 480);


   n = peek (mes_valeurs);       // n vaut 2
   n = peek (mes_valeurs + 1);   // n vaut 5
   n = peek (mes_valeurs + 2);   // n vaut 10
   n = peek (mes_valeurs + 3);   // n vaut 250
   n = peek (mes_valeurs + 4);   // n vaut 0


   // Pour lire toutes les données facilement, on peut écrire une boucle :

   for (i=0; ; i++)
   {
      n = peek(mes_valeurs + i);
      if (n == 0)             // la valeur 0 indique la fin
         break;                // break permet de sortir de la boucle "for"

      print (str$(n));
      nl ();
   }

   for (;;)
      sync ();
}
```

La commande **poke**, à utiliser avec prudence, permet de modifier une valeur de la liste.

```
   poke (mes_valeurs + 1, 6);    // va remplacer la valeur 5 ci-dessus par 6.
```

Attention cependant de ne pas déborder hors de la liste des valeurs, sinon vous iriez écraser une valeur en mémoire plus loin dans le programme.

7.1. Images Dynamiques

Avec la commande poke, il est possible de modifier une image pendant que le programme console tourne. Pour cela, il faut d'abord comprendre la structure d'une image en mémoire.

```
image mon_image (3 * 2, 256 colors)
{
   "010203"
   "030104"
}
```

Les images sont stockées en mémoire en 2 parties. D'abord il y a une entête de 4 bytes qui contient :

- largeur   = 3 : largeur de l'image (arrondie vers le haut si moins de 256 couleurs)
- hauteur   = 2 : hauteur de l'image

- nb_bits   = 8 : nombre de bits utilisés pour stocker un pixel
- inutilisé = 0 : toujours zéro.

Les pixels de l'image sont stockées juste après l'entête en format dépendant du nombre de couleurs.

Le cas le plus simple est l'image à 256 couleurs : là un pixel occupe un byte (nb_bits = 8).

Pour d'autres nombre de couleurs, un pixel occupe (largueur x hauteur x nb_bits / 8) bytes.

Avec la commande poke, il est possible de modifier l'un de ces bytes. Par exemple, pour changer la couleur du premier pixel de l'image en couleur 7 :

```
   poke (mon_image + 4, 7);     // on ajoute 4 pour "passer" l'entête
```

8. Le synthétiseur

Le synthétiseur vous permet de jouer des mélodies ou d'ajouter des bruits (tir, explosion, ..) à vos applications.

Vous disposez de 4 voix indépendantes (1 à 4), d'un oscillateur pouvant générer les formes d'onde prédéfinies (triangle, dent de scie, pulse, bruit, sinusoïde) ou votre propre forme d'onde, d'un générateur d'envelope (adsr), d'options hard-sync et ring-modulation, et de filtres (passe-bas, passe-haut, passe-bande ou coupe-bande).

Le compilateur RUN.EXE peut générer un fichier .WAV (format PCM) avec le résultat des sons créés. Pour cela, il faut créer un raccourci vers run.exe (cliquer sur RUN.EXE, Copier, Coller le Raccourci) ensuite cliquez-droit sur le Raccourci, Propriétés, et ajouter à la fin du champ cible juste après run.exe : **program.txt -wav=test.wav**. Quand vous cliquez alors sur le Raccourci, program.txt sera compilé et exécuté, et test.wav sera créé.

// Voici une application simple qui joue quelques notes :

```
data chanson1
{
   70 16  // note MIDI = 70, durée = 16 x 1/64 sec
   69 16  // note MIDI = 69, durée = 16 x 1/64 sec
   67 16  // note MIDI = 67, durée = 16 x 1/64 sec
   65 16  // ...
   67 32
   67 16
    0  0  // indique la fin
}

proc main ()
{
   volume    (1, 255);
   wave      (1, 2);
   adsr      (1, 10, 10, 128, 100, 5);
   song      (1, chanson1);

   for (;;)
      sync ();
```

}

Voici la liste des commandes du synthétiseur :

volume (voix, vol)

- réglage du volume
- voix : la voix (1 à 4)
- vol : une valeur de volume entre 0 et 255.
- Par défaut, toutes les voix ont le volume 0 donc on n'entend rien.
- exemple: volume (1, 255); // voix 1 au volume maximum

wave (voix, forme_d'onde)

- réglage de la forme d'onde
- voix : la voix (1 à 4)
- forme_d'onde : (0=silence, 1=triangle, 2=dent_de_scie, 3=pulse, 4=bruit, 5=sinusoïde, ou votre propre forme d'onde)
- triangle : son flûte
  dent_de_scie : son trompette
  pulse : bruit de moteur, sirène
  bruit : explosion, tir
  sinusoïde : son doux et mélodieux
- pour la forme d'onde pulse, il faut aussi utiliser la commande pulse, voir plus loin.

- Au lieu d'une forme d'onde prédéfinie, vous pouvez également créer votre propre forme d'onde. Pour cela, vous devez disposer d'un tout petit fichier .WAV PCM avec une forme d'onde exemple (le .WAV ne peut contenir qu'au maximum 65536 samples d'une onde, le graphique de cette onde doit commencer au milieu (valeur 0), évoluer jusqu'au maximum, revenir au minimum, et remonter au milieu).

  Glissez le fichier .WAV sur le programme **wave-converter.exe** du toolkit pour obtenir les data de la forme d'onde à intégrer au script. Donnez un nom aux data (par exemple data piano) et appelez la commande wave en donnant ce paramètre, donc: wave(1,piano);

  Pour obtenir un son plus riche vous pouvez utiliser un fichier contenant plusieurs ondes successives mais uniquement un nombre d'ondes qui est une puissance de 2 (donc 1, 2, 4, 8, 16,.. ondes sont possibles). Si vous spécifiez plusieurs ondes vous devez modifier la 1ère ligne de data (wavelengths) pour spécifier le nombre d'ondes.

adsr (voix, attack, decay, sustain_volume, release, pause)

- réglage de l'enveloppe adsr (permet de choisir un instrument de musique)
- voix : la voix (1 à 4)
- Quand on frappe une touche d'un piano, le son produit passe par 5 phases durant lesquelles le volume change : attack, decay, sustain, release, pause.
- attack : durée (0 à 65535 msec) que met la note pour atteindre son volume maximum (rapide pour un piano, lent pour un violon)
- decay : durée (0 à 65535 msec) que met la note pour redescendre à son volume intermédiaire.
- sustain_volume : volume (0 à 255) intermédiaire.
- release : durée (0 à 65535 msec) que met la note pour descendre au volume zéro.

- pause : durée silencieuse (0 à 65535 msec) entre la fin de la note et la suivante.

freq (voix, frequence)

- réglage de la fréquence (hauteur) d'une note (grave ou aigu)
- voix : la voix (1 à 4)
- frequence : les valeurs entre 1 (grave) et 65535 (aigu) couvrent les fréquences entre 0.23 Hz et 15625 Hz.
- pour calculer une valeur de fréquence à partir d'une fréquence en Hz, multipliez par 4.194304 (exemple: 440 Hz x 4.194304 = 1845)
- une fréquence de 0 coupe le son.
- modifier la fréquence pendant qu'une note est jouée permet d'obtenir des effets intéressants (trémolo, sirène de pompier).

duration (voix, durée)

- réglage de la durée de la note pendant la phase 'sustain'
- voix : la voix (1 à 4)
- durée : durée (0 à 65535 msec) pendant laquelle la note reste au niveau de volume intermédiaire 'sustain'.
  une valeur de -1 signifie durée infinie.

play (voix)

- commencer à jouer une note
- voix : la voix (1 à 4)
- avant de jouer une note, il faut régler : volume, wave, adsr, freq et duration.

n = note (voix)

- permet de savoir dans quelle phase se trouve la voix du synthétiseur
- voix : la voix (1 à 4)
- renvoie les valeurs suivantes :
- 1 = attack
- 2 = decay
- 3 = sustain
- 4 = release
- 5 = pause
- 0 = la note est finie.
- pour tester si une note est finie (avant de jouer la suivante), on peut attendre comme ceci :

```
while (note(1) != 0)    // tant que la note de la voix 1 n'est pas finie
  sleep (0);            // attendre
```

midi (voix, note_midi, durée_midi)

- permet de jouer une note (cette commande combine en fait en une seule les commandes freq, duration et play)
- voix : la voix (1 à 4)
- note_midi : note MIDI (0 à 130) (0 = 8.17 Hz, 69 = 440 Hz, 127 = 12543 Hz).

- durée_midi : durée de la note MIDI en 1/64 de seconde (64 = 1 sec, 32 = 1/2 sec, 16 = 1/4 de sec, ..)

song (voix, zone_data)

- permet de jouer une mélodie entière (cette commande combine en fait en une seule les commandes midi et note, et répète cela pour chaque note)
- voix : la voix (1 à 4)
- zone_data : mélodie à jouer, encodée sous format MIDI (voir exemple plus haut).
- Cette commande démarre seulement la mélodie, ce qui signifie que le programme script peut faire d'autres choses, par exemple afficher des textes ou des sprites pendant que la mélodie continue à jouer. De plus, on peut jouer des mélodies à plusieurs voix en démarrant une commande song sur chaque voix.
- On peut tester avec la commande note() si la mélodie est finie, note() renvoie dans ce cas zéro.

pulse (voix, width)

- règle la largeur de la forme d'onde pulse
- voix : la voix (1 à 4)
- width : une valeur entre 0 (inaudible) et 32768 (onde carrée).

hard_sync (voix, cut_freq)

- produit l'effet hard-sync qui consiste à 'couper' brutalement la fin d'une forme d'onde (bruit d'un mosquito).
- voix : la voix (1 à 4)
- cut_freq : une fréquence plus petite que celle choisie pour freq.

ring_mod (voix, ring_freq)

- produit l'effet ring modulation qui consiste à ajouter des sur-sons non-harmoniques (bruit d'un gong).
- voix : la voix (1 à 4)
- ring_freq : une fréquence plus petite que celle choisie pour freq.
- ring modulation ne fonctionne que pour la forme d'onde triangle (voir commande wave).

```
// demo : hard sync et ring modulation

proc main ()
{
   var i;

   // demo 1 : hard sync

   volume    (1, 255);
   wave      (1, 1);
   adsr      (1, 3000, 2400, 0, 0, 0);
   freq      (1, 6556);
   hard_sync (1, 1835);

   duration (1, 5400);
   play(1);
```

```
   while (note(1))
      ;

   hard_sync (1, 0);


   // demo 2 : Gong (avec ring modulation)

   volume   (1, 255);
   wave     (1, 1);
   adsr     (1, 0, 1500, 0, 0, 0);
   freq     (1, 13112);
   duration (1, 900);
   ring_mod (1, 1966);

   for (i=0; i<4; i++)
   {
      play(1);
      while (note(1))
         ;
   }

   for (;;)
      sync();
}
```

filter (voix, type, cut_freq)
ou
filter (voix, type, cut_freq, resonance)

- filtrer un son
- voix : la voix (1 à 4)
- type : type de filtre :
  0 = aucun
  1 = passe-bas (laisse passer les fréquences plus petites que cut_freq, atténue les autres de 12 dB/octave)
  2 = passe-haut (laisse passer les fréquences plus grandes que cut_freq, atténue les autres de 12 dB/octave)
  3 = passe-bande (laisse passer les fréquences proches de cut_freq, atténue les autres de 6 dB/octave)
  4 = coupe-bande (atténue les fréquences proches de cut_freq de 6 dB, laisse passer les autres)
- cut_freq : fréquence du filtre.
- resonance : amplification de la fréquence cut_freq (entre -50 dB et +50 dB, normal = 0).
- utiliser le filtre sur la forme d'onde bruit permet d'obtenir des effets intéressants.
- modifier cut_freq pendant qu'une note joue permet d'obtenir des effets intéressants (décolage de fusée).

c = async ();

- envoie l'écran console à la carte graphique sans attendre 1/60 seconde.
- Cette commande est similaire à sync(), mais elle renvoie une valeur 0 ou 1.
- async renvoie 1 si elle n'a pas fini d'envoyer l'écran à la carte graphique, cela veut dire qu'il faudra refaire un autre async plus tard pour terminer le travail.
- async renvoie 0 si elle a fini d'envoyer l'écran à la carte graphique.

- async permet d'éviter la pause de 1/60 seconde, ce qui est nécessaire par exemple si on veut jouer une suite de notes au synthétiseur. On peut appeler la commande async tant qu'on veut, et dès qu'elle renvoie 0 on sait qu'on peut déplacer les objets graphiques (texte, sprites, ..).

sleep (durée);

- Cette commande attend un temps déterminé.
- durée : en millisecondes (1/1000 seconde).

```
// Exemple: synthétiseur rudimentaire

proc main ()
{
   var x, y, b, n;


   screen (640, 480);
   font (8, 8);


   // dessiner des touches

   paper (89);

   for (y=0; y<11; y++)
   {
      for (x=0; x<12; x++)
      {
         n = x + y*12;
         if (n > 128)
            break;

         at (x*5, y*5);
         print ("   ");
         at (x*5, y*5+1);
         print ("   ");
         at (x*5, y*5+2);
         print ("   ");

         at (x*5+(n<10)+(n<100), y*5+1);
         print (str$(n));
      }
   }
   sync ();


   // définir un instrument

   volume    (1, 255);
   wave      (1, 2);
   adsr      (1, 0, 0, 255, 0, 0);


   // jouer quand on clique
```

```
   for (;;)
   {
      mouse (x, y, b);
      if (b)
      {
         n = (x/8/5) + (y/8/5)*12;
         if (n >= 0 && n <= 128)
            midi (1, n, 0);
      }
   }
}
```

9. Communication avec le bot du chat

Ce chapitre décrit les commandes qui permettent à une application console d'interagir avec le bot du chat.

Commençons par un exemple simple : vous avez écrit un script console **program.txt**, vous l'avez compilé en **program.run**, et maintenant vous souhaitez le faire tourner sur le chat. Seul un bot peut lancer un script console (à l'aide de la commande console_run), vous devez donc d'abord écrire un script pour le bot.

Attention à ne pas confondre les scripts console (qui démarrent par proc main) et les scripts bot (qui démarrent par proc event) !

Voici un exemple de script bot (bot-script.txt) à placer dans le répertoire script de votre bot :

```
// bot-script.txt (ceci est le script du BOT)

proc event (session_key, userid$, sex$, has_photo, age, is_away,
                  admin, cam_on, is_bot, toc_capab, signature$,
                  action, is_myself, line$)
{
   if (action == 0 && pos(line$, "!test") > 0)
      console_run (session_key, "program.run");    // démarre la console
}
```

Placez ce script bot dans le répertoire script du bot, ajoutez-y également votre application console **program.run**. Si vous tapez **!test** sur le bot, il va lancer l'application console sur votre écran !

Ceci est bien sûr un exemple minimal. Il est possible par exemple de démarrer l'application console sur d'autres écrans de chat que le vôtre (il suffit de spécifier une session_key différente) ou encore plus intéressant : d'échanger des messages entre le bot et les applications console qui tournent chez tous les chatteurs en salle.

On utilise pour cela les commandes suivantes : (attention: ne mélangez pas les commandes bot et les commandes console !)

Commande BOT :

console_run (session_key, "ping-pong.run");

- permet au bot de démarrer une application console.
- session_key : le numéro unique du chatteur où la console sera démarrée.

Commande BOT :
console_send (session_key, phrase$);

- permet au bot d'envoyer un string à la console qu'il a démarrée.
- session_key : le numéro unique du chatteur où la console tourne.
- attention: la longueur du string (phrase$) ne peut pas dépasser 4095 caractères !
- exemple: console_send (session_key, "ça marche !");
- la console pourra lire le string à l'aide des commandes bot_message_arrived() et bot_message(), voir plus bas.

Commande CONSOLE :
b = bot_message_arrived();

- permet à la console de tester si le bot lui a envoyé un message.
- b va recevoir 1 si le bot lui a envoyé un message, sinon b va recevoir 0.
- pour lire le message, voir la commande bot_message plus bas.

Commande CONSOLE :
bot_message(m$);

- réceptionne le message que le bot lui a envoyé et le place dans m$
- si le bot n'a rien envoyé, la fonction attend que le bot lui envoie un message.
- le message a une longueur maximale de 4095 caractères.

Commande CONSOLE :
bot_event(message$);

- permet à la console d'envoyer un message au bot.
- attention: la longueur du string (message$) ne peut pas dépasser 4095 caractères !
- le bot va recevoir un évènement avec l'action 702, et le message sera dans line$, voir l'exemple ci-dessous.

Exemple:

// bot-script.txt (ceci est le script du BOT)

```
proc event (session_key, userid$, sex$, has_photo, age, is_away,
                admin, cam_on, is_bot, toc_capab, signature$,
```

```
                action, is_myself, line$)
{
   if (action == 128)
      bubble ("Bienvenue en salle, tapez !jeu ..");

   if (action == 0 && pos(line$, "!jeu") > 0)
   {
      console_run (session_key, "jeu.run");       // démarre la console
      console_send (session_key, "ça marche !");  // envoi d'une phrase à la console
      console_send (session_key, "excellent !");
   }

   if (action == 702)   // la console nous a envoyé un message dans line$
   {
      print ("action 702 : " + line$);  // afficher le message
   }
}



// jeu.txt (ceci est l'application CONSOLE)

proc main()
{
   var m$;

   screen (640, 256);

   for (;;)
   {
      if (bot_message_arrived())  // le bot nous a envoyé un message
      {
         bot_message (m$);   // lire le message

         print (m$);         // imprimer le message
         sync ();

         bot_event ("ma réponse est : " + m$);    // renvoyer une réponse au bot
      }
   }
}
```

9.1. Transférer une bannière du bot vers la console

Fonction BOT :
s$ = sprite_image$ (pseudo$, largeur, hauteur);

- permet au bot de convertir la bannière-image d'un chatteur en une image à 256 couleurs en format "sprite" pour la console.
- largeur ne peut dépasser 248, hauteur ne peut dépasser 248, largeur x hauteur ne peut dépasser 4091.
- la longueur du string calculé (s$) ne peut dépasser 4095.

- le string calculé (s$) peut contenir n'importe quelles valeurs de byte entre 0 et 255, cela dépend de la couleur de la bannière.

Exemple:

s$ = sprite_image$ (pseudo$, 168, 21);

va convertir la bannière en un string de maximum 4+(168*21) bytes. Ce string est composé d'une entête de 4 bytes (largeur, hauteur, nombre de bits par pixel et zéro), suivi de 21 fois 168 bytes. Chaque byte représente la couleur d'un pixel. (voir chapitre 7.1. Images Dynamiques).

Largeur et Hauteur sont des dimensions maximales :
- si le pseudo$ est plus grand que 168 x 21, l'image sera coupée.
- si le pseudo$ est plus petit que 168 x 21, le string calculé sera plus court.

Exemple: transfert bannière vers console

```
// script.txt

proc event (session_key, userid$, sex$, has_photo, age, is_away,
                admin, cam_on, is_bot, toc_capab, signature$,
                action, is_myself, line$)
{

   if (action == 127 || action == 128)
   {
      console_run (session_key, "banner.run");
      console_send (session_key, sprite_image$(userid$, 168, 21));
   }
}


// bannner.txt  (application console)

image banner1 (168 * 21, 256 colors)
{
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
```

```
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
   "
"
}

proc main ()
{
   var m$, length, i;

   screen (640, 120);
   paper (14);
   fill (0, 0, 40, 40, 32);
   sync();

   for (;;)
   {
      if (bot_message_arrived ())
      {
         bot_message(m$);

         // copier le string m$ dans l'image banner1
         length = len(m$);
         for (i=0; i<length; i++)
            poke (banner1 + i, asc(m$,i+1));

         sprite_on (1, 1);
         sprite_pos (1, 0, 0);
         sprite_image (1, banner1);

         sync();
      }
   }
}
```

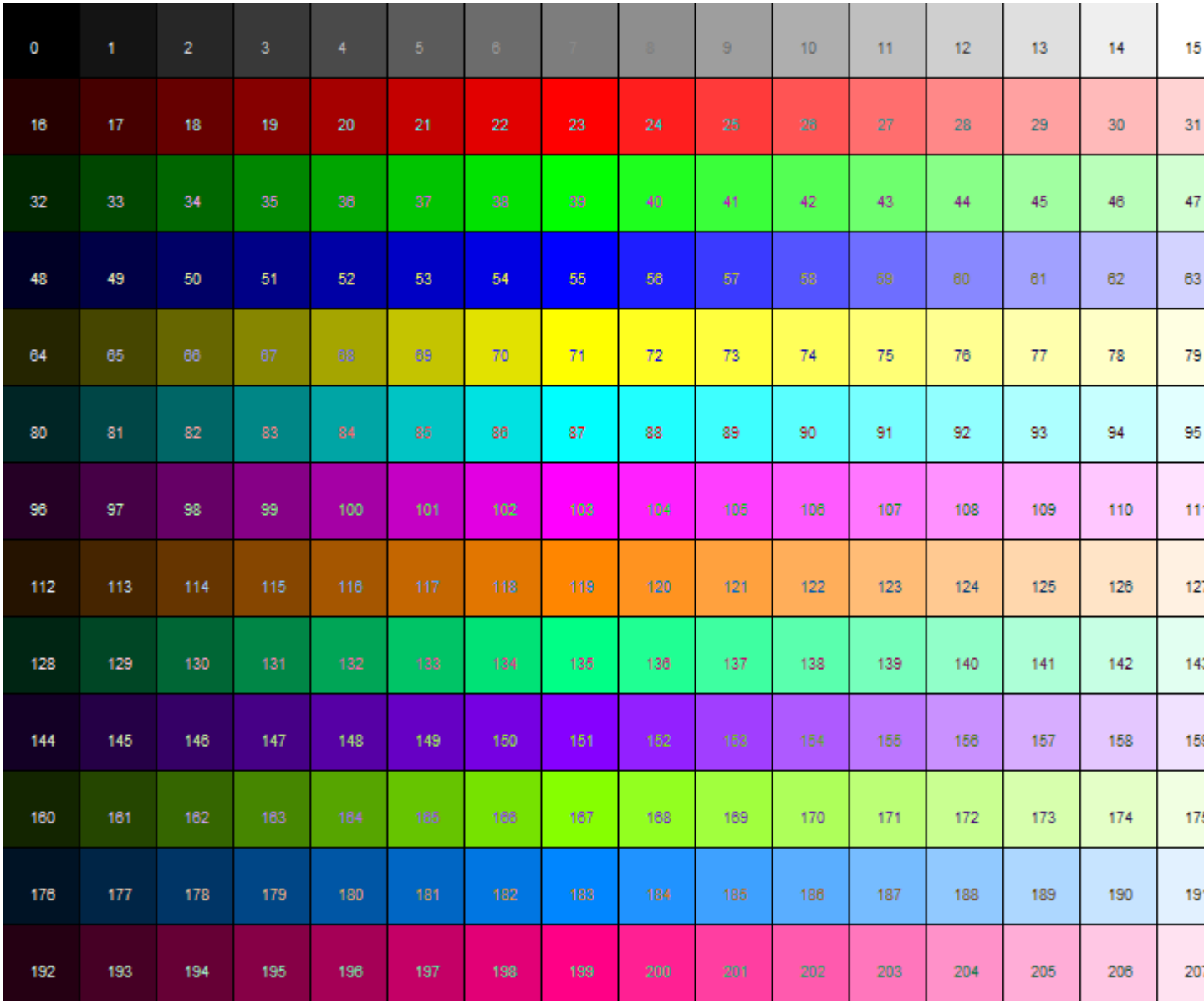
Annexe 1 : Liste des codes caractères affichables

Annexe 2 : Liste des couleurs

Vous disposez de 13 couleurs de base : gris, rouge, vert, bleu, jaune, cyan, rose, orange, vert neutre, violet, vert acide, bleu ciel et rouge carmin. Chacune de ces couleurs peut être affichée en 16 gradations (de la plus foncée à la plus claire). Voici un tableau résumé de toutes les couleurs :

  0 à 15 : 0=noir, 1-14=gris, 15=blanc
 16 à 31 : rouge (du plus foncé au plus clair)
 32 à 47 : vert (du plus foncé au plus clair)
 48 à 63 : bleu ...
 64 à 79 : jaune
 80 à 95 : cyan (vert/bleu)
 96 à 111 : rose
112 à 127 : brun/orange
128 à 143 : vert neutre
144 à 159 : violet/mauve
160 à 175 : vert acide pomme
176 à 191 : bleu ciel
192 à 207 : rouge carmin

Par exemple: pour avoir du rouge moyen, choisissez une couleur entre 16 et 31 : 23.



Annexe 3 : Maitriser le scrolling

Scrolling Horizontal
==========
pour scroller la ligne L horizontalement de 8 vers la gauche
(ce qui va afficher la moitié d'une colonne cachée de droite),
faites:

scroll (L, 8, 0);

Vous ne pouvez scroller que de 32 points. Pour aller au-delà,
il faut combiner avec la commande blit.

Scrolling Vertical
==========
Le scrolling vertical est plus compliqué à maitriser
que le scrolling horizontal.

N'oubliez pas qu'on ne peut scroller que des lignes entières.
Imaginez que vous voulez scroller doucement vers le bas
les lignes 5 à 7 de l'écran.

Pour préparer une ligne "cachée" qui va vous servir
à faire apparaitre le texte doucement, exécutez d'abord
une commande scroll sur toutes les lignes, de la
ligne 5 jusqu'en bas de l'écran (ligne 31) :

```
for (l=5; l<=31; l++)
   scroll (l, 0, 16);
```

L'effet sera que toutes les lignes seront décalées,
le contenu de la ligne 6 sera affiché à la ligne 5 !

Scrollez alors les lignes 5 à 7 doucement vers le bas, avec :

```
for (s=15; s>=0; s--)
{
   scroll (5, 0, s);
   scroll (6, 0, s);
   scroll (7, 0, s);

   for (t=0; t<10; t++)   // attendre 1/6 de sec.
      sync ();
}
```

Finalement, vous devez copier la ligne 5 vers la ligne 6,
la ligne 6 vers la ligne 7, et la ligne 7 vers la ligne 8
par :

```
blit (0, 5,    // à partir de colonne 0, ligne 5
      42, 3,   // déplacer 42 colonnes, 3 lignes
      0, 6);   // vers colonne 0, ligne 6.
```

Notez que la ligne 8, bien que ne faisant pas partie de la fenêtre,
est utilisée comme ligne cachée !

Si vous imprimez du texte sur n'importe quelle ligne en dessous
de la fenêtre, vous devez ajouter 1 à la ligne !

```
at (0, 9);
print ("juste en dessous de la fenêtre !");
```

Finalement, voici un exemple complet :

```
proc main ()
{
   var y, s, t, n, l;

   screen (640, 480);


   // afficher un texte sur toutes les lignes ..

   for (y=0; y<32; y++)
   {
```

```
      at (0, y);
      print ("Ceci est la ligne " + str$(y));
   }


   // préparation : décaler toutes les lignes
   // à partir de la ligne 5 jusqu'en bas de l'écran.

   for (l=5; l<32; l++)
      scroll (l, 0, 16);


   // pour scroller les lignes 5 à 7 vers le bas ..

   for (;;)
   {
      // afficher le nouveau texte sur la ligne 5

      at (0,5);
      n++;
      print ("ceci est ma ligne de test numéro " + str$(n));


      // scroller de 15 points (la hauteur d'une ligne)

      for (s=15; s>=0; s--)
      {
         scroll (5, 0, s);
         scroll (6, 0, s);
         scroll (7, 0, s);

         for (t=0; t<10; t++)   // attendre 1/6 de sec.
            sync ();
      }

      // descendre les 3 lignes
      blit (0, 5,    // à partir de colonne 0, ligne 5
               42, 3,   // déplacer 42 colonnes, 3 lignes
               0, 6);   // vers colonne 0, ligne 6.
   }

}
```

Annexe 4 : jeu de casse-brique avec sprites, collisions et son

```
// casse_brique.txt

image balle (6 * 6)
{
   " XXXX "
   "XXXXXX"
   "XXXXXX"
   "XXXXXX"
   "XXXXXX"
   " XXXX "
```

```
}

image raquette (1 * 8)
{
   "X"
   "X"
   "X"
   "X"
   "X"
   "X"
   "X"
   "X"
}

image brique (12 * 5, 4 colors)
{
   "111111111111"
   "1XXXXXXXXXX1"
   "1XXXXXXXXXX1"
   "1XXXXXXXXXX1"
   "111111111111"
}

data brique_color
{
   0  // bord de la brique : noir
}

data colors   // 6 couleurs de brique
{
   88 72 40 120 24 136
}


proc main ()
{
   var max_x, max_y;
   var x, y, b;               // souris
   var rx, ry, rlen;          // coordonnée et longueur raquette
   var bx, by, dx, dy, speed;  // coordonnées & direction balle
   var score, vies, i, quit, briques;
   var col[100];

   max_x = 640;
   max_y = 200;
   screen (max_x, max_y);

   volume (1, 255);
   wave (1, 1);
   adsr (1, 0, 0, 128, 0, 0);

   // raquette
   sprite_on    (0, 1);
   sprite_image (0, raquette);
   sprite_color (0, 24);   // raquette rouge

   sprite_zoom  (0, 3, 3);
```

```
rlen = 64;

// balle
sprite_on (1, 1);
sprite_image (1, balle);

// 60 briques
for (i=2; i<62; i++)
{
   sprite_on (i, 1);
   sprite_image (i, brique, brique_color);
   sprite_pos (i, ((i-2) % 10) * 55, 20 + ((i-2) / 10) * 30);
   col[i] = i % 6;
   sprite_color (i, peek(colors + col[i]));
   sprite_zoom (i, 2, 2);
   briques++;
}

// position initiale de la raquette
rx = max_x - 64;
ry = max_y / 2;

// position initiale de la balle
bx = random (0, max_x / 10);
by = 0;

// vitesse de déplacement de la balle
speed = 2;
dx = speed;
dy = speed * 2;

vies = 3;
at (25, 0);
print (dup$(chr$(1) + " ",vies) + dup$("  ", 3));

quit = 0;
for (;;)
{
   if (quit || briques==0)
      break;

   // déplacer la raquette
   mouse (x, y, b);
   if (y >= -rlen && y < max_y + rlen)
      ry = y - rlen/2 + 8;
   sprite_pos (0, rx, ry);

   // déplacer la balle
   bx += dx;
   by += dy;
   sprite_pos (1, bx, by);

   // vérifier si la balle a touché le bord
   if (bx < 0)
   {
      dx = abs(dx);
      midi (1, 75, 12);
```

```
}
if (by < 0)
{
   dy = abs(dy);
   midi (1, 75, 12);
}
if (by >= max_y-8)
{
   dy = -abs(dy);
   midi (1, 75, 12);
}
if (bx > max_x)
{
   vies--;
   at (25, 0);
   print (dup$(chr$(1) + " ",vies) + dup$("  ", 3));

   bx = random (0, max_x / 10);
   by = 0;
   dx = speed;
   dy = speed;

   if (vies <= 0)
      quit = 1;
}

// afficher tout
sync ();

// après sync, détecter les collisions entre balle et raquette
if (sprite_collision (0) == 1)   // raquette(sprite 0) a été touché par la balle(sprite 1)
{
   dx = -abs(dx);

   if (by + 3 <= ry + rlen/4)
   {
      dy = -2*speed;
   }
   else if (by + 3 >= ry + rlen*3/4)
   {
      dy = +2*speed;
   }
   else
   {
      if (dy < 0)
         dy = -speed;
      else
         dy = +speed;
   }

   midi (1, 70, 2);
}

// après sync, détecter les collisions entre balle et brique
i = sprite_collision (1);
```

```
      if (i >= 2)     // balle(sprite 1) a été touchée par brique i
      {
         dy = -dy;   // changer la trajectoire

         if (col[i] > 0)
         {
            col[i]--;
            sprite_color (i, peek(colors + col[i]));
         }
         else
         {
            sprite_on (i, 0);   // éteindre le sprite brique touché
            briques--;
         }

         midi (1, 80, 2);

         score++;
         at (12, 0);
         print ("SCORE " + str$(score));
      }
   }

   at (9, 4);
   if (briques)
   {
      print (" G A M E   O V E R");
      bot_event ("PERDU");
   }
   else
   {
      print ("well done, Champion !");
      bot_event ("GAGNE");
   }
   for (i=0; i<180; i++)
      sync();
}
```

Annexe 5 : La Démo "Samuro présente"

```
// la démo "Samuro présente"


proc centrer (ligne, texte$)
{
   at ((40 - len(texte$))/2, ligne);
   print (texte$);
}


proc presente ()
{
   var u, t;
```

```
   for (u=0; u<=30; u++)
   {
      if (u <= 15)
         ink (15-u);
      else
         ink (u-15);

      centrer (10, "Samuro présente ...");

      for (t=0; t<10; t++)
         sync ();
   }
}


proc demo1 ()
{
   var u, x;

   paper(191);
   fill (0, 0, 44, 34, 32);

   ink (38);
   centrer (12, "Une nouvelle console de jeu ...");
   sync ();

   for (u=0; u<4; u++)
   {
      for (x=0; x<32; x++)
      {
         scroll (12, x, 0);
         sync ();
      }

      for (x=32; x>0; x--)
      {
         scroll (12, x, 0);
         sync ();
      }
   }

   for (x=0; x<80; x++)
   {
      blit (0, 12, 80, 1, 1, 12);
      sync ();
   }

   paper(15);
   fill (0, 0, 44, 34, 32);
}



proc colors ()
{
   var col;

   for (col=0; col<208; col++)
```

```
    {
        paper (col);
        fill (0, 0, 44, 34, 32);
        sync ();
    }

    paper(15);
    fill (0, 0, 44, 34, 32);   // 32 = espace vide
}



proc charset ()
{
    var x, y, t;

    ink (0);
    centrer (1, "Les symboles");
    centrer (2, "============");

    for (y=0; y<16; y++)
    {
        at (0, y+5);
        print (str$(y*16));

        for (x=0; x<16; x++)
        {
            at (x+5, y+5);
            print (chr$(x+16*y));
        }
    }
    for (t=0; t<240; t++)
        sync ();
}


proc balle ()
{
    var x, y, u, dx, dy, high_y;

    x = 0;
    y = 0;

    dx = 1;
    dy = 1;

    high_y = 29;

    ink (22);
    for (u=0; u<1500; u++)
    {
        if (u > 1000)
            ink (random(0, 255));

        at (x, y);
        print (chr$(145));

        sync ();
```

```
      if (u < 1000)
      {
         at (x, y);
         print (" ");
      }

      x += dx;
      y += dy;

      if (x == 0 || x == 39)
         dx = -dx;

      if (y == 0 || y == high_y)
         dy = -dy;

      if (u == 1000)
      {
         screen (640, 480);
         high_y = 29;
      }
      else if (u > 200 && u < 1000 && y == 0 && high_y > 2)
      {
         screen (640, high_y*16);
         high_y--;
      }
   }

   ink (0);
}


proc demo2 ()
{
   var u, t, a, x, y;

   ink (56);
   centrer (0, "Voilà, c'est déjà fini !");

   for (y=0; y<60; y++)
      scroll (y, 0, 8);

   for (t=0; t<40; t++)
   {
      for (a=16; a>=0; a--)
      {
         for (y=0; y<60; y++)
            scroll (y, 0, a);
         sync ();
      }

      blit (0, 0, 80, 60, 0, 1);   // line down

      if (t == 0)
      {
         centrer (0, "                         ");
      }
```

```
      if (t == 10)
      {
         centrer (2, " oui, c'était court, je sais ...  ");
      }
   }
}


proc the_end ()
{
   var t;
   ink (0);
   paper (24);
   centrer (19, "             ");
   centrer (20, "  The End !  ");
   centrer (21, "             ");
   for (t=0; t<180; t++)
      sync ();
}


proc reduire ()
{
   var x, y, u, t;

   x = 640;
   y = 480;
   for (u=0; u<8; u++)
   {
      screen (x, y);
      for (t=0; t<30; t++)
         sync ();
      x = x / 2;
      y = y / 2;
   }
}

proc main ()
{
   screen (640, 480);

   colors ();

   presente ();
   demo1 ();
   charset ();
   balle ();
   demo2 ();
   the_end ();
   reduire ();
}
```

Annexe 6 : Le langage Script

Elements du langage script
==========

Données
=======

Le langage script manipule deux types de données :

1) les nombres entiers :

```
9  -62  136  0xFFD2
```

Les valeurs doivent être comprises entre -2147483647 et +2147483647
Le nombre 0xFFD2 est en format hexadécimal (base 16 au lieu de la base 10 habituelle).

2) les chaines de caractères :

```
""   "Bonjour"  "Oui:p"
```

la première chaine de caractères ci-dessus est une chaine vide.
la longueur maximale d'une chaine est de 8192 caractères.

Variables
=========

Une variable est une case de la mémoire du PC à laquelle on donne un nom;
cette case mémoire contient une valeur qui peut changer au fur et à mesure
que le programme se déroule.

Il existe 2 types de variables :

- celles qui contiennent un nombre entier (exemple: compteur, i, joueur)

- celle qui contiennent une chaine de caractère (exemple: fruit$, user$)
  (ces dernières se terminent toujours par un signe dollar '$')

Avant d'utiliser une variable, il faut la 'déclarer', c'est-à-dire réserver
de la place mémoire pour son contenu, à l'aide de la commande 'var'.

exemple:

```
var compteur;    // va contenir un compteur
var user$, t$;   // deux variables chaines de caractère
var fruit$[3];   // un tableau de 3 chaines : fruit$[1], fruit$[2] et fruit$[3]
```

Les variables peuvent être déclarées,
- soit tout en haut du script,
- soit au début d'une fonction ou d'une procédure.

Si elles sont déclarées tout en haut du script, elles gardent leur valeur
tant que l'application tourne; autrement, elles sont effacées à chaque fois
qu'on appelle la fonction ou procédure.

Par défaut, les variables numériques ont la valeur zéro, et les variables chaines
de caractère contiennent une chaine vide.

Procédures et Fonctions
==========

Une procédure est une commande à laquelle on peut donner des paramètres,
par exemple 'print' auquel on donne une chaine de caractères "Bonjour !"
ce qui donne :

```
print ("Bonjour !");
```

Une fonction ressemble à une procédure, mais en plus elle calcule un résultat,
par exemple 'len' qui calcule la longueur d'une chaine de caractères :

```
longueur = len ("abc");
```

Les procédures/fonctions standards suivantes existent :

calendrier

```
d$ = now$ ();          // fonction qui calcule la date du jour
                              // exemple: now$() == "2007/01/17 12:23:12"

d  = weekday (d$);     // renvoie le jour de la semaine (1=lundi, 7=dimanche) correspondant à la
date
                              // exemple: weekday("2007/01/17 12:23:12") == 2

d$ = add_date$ (d$,n);  // renvoie la date vieillie de n secondes
                              // exemple: add_date$("2007/01/17 12:23:12", 3600) == "2007/01/17
13:23:12"

n = nb_days_since_1901 (d$);  // renvoie le nombre de jours entre le 1/1/1901 et la date fournie,
                                    // ce qui peut servir à calculer le nombre de jours entre deux
dates:
                                    // nb_days_since_1901 (date2$) - nb_days_since_1901
(date1$)
```

numérique

```
n = random (a,b);      // renvoie un nombre aléatoire entre a et b
                              // exemple: random(1,50) == 12

n = abs (n);           // renvoie la valeur absolue de n
                              // exemple: abs(-2) == 2

n = sin (angle, M);    // calcule le sinus(angle) multiplié par M

n = cos (angle, M);    // calcule le cosinus(angle) multiplié par M

angle = atan2 (Y, X);  // calcule l'angle correspondant à l'arc-tangeante de (Y / X)

n = sqrt (n, M);       // calcule la racine carrée de n, multipliée par M
```

chaîne de caractère

```
n  = len(s$);          // renvoie la longueur de la chaine s$
                              // exemple: len("abc") == 3
```

```
s$ = dup$(s$,N);        // renvoie la chaine s$ dupliquée N fois (avec N >= 0)
                                // exemple: dup$("ab",3) == "ababab"

s$ = chr$(N1,N2,N3,..); // renvoie une chaine constituée des caractères ascii N1,N2,..
concaténés
                                // exemple: chr$(65,66,67) == "ABC"

c  = asc(s$[,N]);       // renvoie le numéro ascii du Nième caractère de la chaine s$ (N vaut 1 si
omis)
                                // exemple: asc("A") == 65
                                // exemple: asc("ABC",2) == 66

s$ = str$(N);           // renvoie une chaine représentant le nombre N
                                // exemple: str$(23) == "23"

n  = val(s$);           // renvoie le nombre représenté par la chaine s$ (l'inverse de str$)
                                // exemple: val("23") == 23

s$ = left$(s$,N);       // renvoie les N caractères de gauche de s$
                                // exemple: left$("ABCD",2) == "AB"

s$ = right$(s$,N);      // renvoie les N caractères de droite de s$
                                // exemple: right$("ABCD",2) == "CD"

s$ = mid$(s$,a,N);      // renvoie N caractères de s$ commençant au 'a'ième
                                // exemple: mid$("ABCDEF",2,3) == "BCD"

i  = pos(s1$,s2$);      // renvoie la position de la chaine s2$ dans s1$, ou 0 si pas trouvé
                                // exemple: pos("ABCDEF","CD") == 3
                                // exemple: pos("ABCDEF","X") == 0
```

En plus des procédures et fonctions déjà existantes décrites ci-dessus,
on peut en écrire des nouvelles; voici trois exemples :

```
// une fonction qui calcule le double de n
func double (n)
{
   var resultat;
   resultat = n * 2;
   return resultat;
}

// une fonction qui renvoie une chaine de N tirets
func tiret$ (n)
{
   return dup$("-", n);
}

// une procédure qui imprime N tirets
proc print_tiret (n)
{
   print (tiret$(n));
}
```

Opérateurs

==========

Les opérateurs permettent d'effectuer des calculs, par exemple:

```
2 + 3 * 4
```

Chaque opérateur a une priorité, par exemple * a une priorité supérieure
à +, ce qui veut dire que le calcul ci-dessus est équivalent à :

```
2 + (3 * 4)
```

et non pas à :

```
(2 + 3) * 4
```

Si ce dernier calcul est désiré, l'usage de parenthèses est requis.

Voici les opérateurs disponibles, par ordre de priorité décroissante :

opérateurs numériques

!	NON logique
*	multiplié par
/	divisé par
%	reste de la division
&	bitwise and
|	bitwise or
+	ajouter
-	soustraire
<=	plus petit que ou égal à
>=	plus grand que ou égal à
==	égal
!=	pas égal à
<	plus petit que
>	plus grand que
&&	ET logique
||	OU logique

opérateurs chaines

+	concaténer 2 chaines
<=	plus petit que ou égal à (ordre lexicographique = dictionnaire)
>=	plus grand que ou égal à
==	égal
!=	pas égal à
<	plus petit que
>	plus grand que

exemple:

```
if ("ab" < "abc")
    print ("ab vient avant abc dans le dictionnaire");
```

Instructions
==========

l'assignation

L'instruction d'assignation permet d'effectuer un calcul (à droite)
et de stocker le résultat dans la variable (à gauche).

```
result = a * b + c;          // calcule '(a * b) + c' et stocke dans la variable 'result'
line$ = "Hello" + userid$;   // concatène "Hello" et userid$ et stocke dans line$
fruit$[2] = "pomme";         // stocke la chaine "pomme" dans la case 2 du tableau fruit$
ligne$ = "i contient la valeur " + str$(i);
longueur = len(s$) + 1;
```

L'assignation avec incrémentation permet d'ajouter ou de soustraire
une valeur à la variable de gauche.

```
i += 72;             // ajoute 72 à i
j -= (b + c);
```

L'incrémentation simple permet d'ajouter 1 ou de soustraire 1 à une variable:

```
i++;       // ajoute 1 à i
j--;       // soustrait 1 à i
```

l'instruction if

L'instruction if permet de n'exécuter un groupe d'instructions
que si une condition est remplie.

```
// if avec 1 seule instruction

if (i < 10)
   print ("i trop petit");   // n'exécuter le print que si i est plus petit que 10


// if avec plusieurs instructions
// note: des { } sont obligatoires si on a plusieurs instructions

if (i < 10)
{
   print ("i trop petit");
   print ("recommencez");
}


// "if" avec partie "else"

if (i < 10)     // si i plus petit que 10
{
   print ("i trop petit");
   print ("recommencez");
}
else          // sinon
{
```

```
    print ("i égal ou supérieur à 10 ! oué");
}


// "if" avec plusieurs parties en cascade

if (i < 10)               // si i plus petit que 10
{
    print ("i trop petit");
    print ("recommencez");
}
else if (i == 10)         // sinon si i égal à 10
{
    print ("i égal à 10 ! oué");
}
else                      // sinon (i plus grand que 10)
{
    print ("i plus grand que 10 ! oué");
}


// if avec expressions complexes

if (i < 10 && j == 3 && a$ == "hello")    // si i < 10 ET j == 3 ET a$ == "Hello"
    print ("oui");

if (i < 10 || j == 3 || a$ == "hello")    // si i < 10 OU j == 3 OU a$ == "Hello"
    print ("oui");


// if avec instruction vide

if (i < 10)
    ;         // ne rien faire
else
{
    print ("oui");
}
```

l'instruction FOR

L'instruction FOR permet de répéter l'exécution d'un groupe d'instructions entre { } en augmentant la valeur d'une variable à chaque exécution.

exemples:

```
// pour imprimer les nombres de 1 à 5 :

for (i=1; i<=5; i++)
{
    print (str$(i));
}
```

imprime:
1
2
3
4
5

// pour imprimer les nombres de 5 à 1 :

```
for (i=5; i>=1; i--)
{
   print (str$(i));
}
```

imprime:
5
4
3
2
1

l'instruction WHILE

L'instruction WHILE permet de répéter l'exécution d'un groupe
d'instructions tant qu'une certaine condition est vérifiée.

```
// tant que i <= 10,
// répéter les instructions entre { }

i = 1;
while (i <= 10)
{
   print ("Bonjour");
   i++;
}


// parcourir le tableau tab[]
// jusqu'à tomber sur une valeur 6

var tab[512];

.....

i = 0;
while (i <= 512 && tab[i] != 6)   // tant que i <= 512 et que tab[i] pas égal à 6, augmenter i
{
   i++;
}
```

break et continue

```
// l'instruction 'break' fait quitter la répétition immédiatement

for (i=1; i<=6; i++)
{
   if (i == 3)
      break;
   print (str$(i));
}

imprime:
   1
   2



// l'instruction 'continue' fait passer au tour suivant

for (i=1; i<=6; i++)
{
   if (i == 3 || i == 5)   // si i égale 3 ou 5, passer au i suivant
      continue;
   print (str$(i));
}

imprime:
   1
   2
   4
   6
```

Annexe 7 : Apprendre Le langage Script

Ce cours va vous apprendre à écrire des scripts console pour le chat.

Chapitre 1 : premier script
==========

Pour créer un script, ouvrez le bloc-notes (notepad) et écrivez ceci :

```
// mon premier script console

proc main ()
{
   screen (160, 16);        // mettre l'écran en 160 x 16
   print ("Bonjour !");     // dire bonjour
   sync ();                 // afficher à l'écran
   sleep (5000);            // attendre 5 secondes
}
```

Ensuite sauvez sous un nom qui se termine par .txt, par exemple "program.txt".

Ensuite allez dans Windows Explorer et tirez avec la souris
"program.txt" sur "run.exe"
(run.exe se trouve dans le kit que vous avez téléchargé au début).

Vous allez voir une fenêtre s'ouvrir avec le mot "Bonjour !"

Voilà, vous venez d'écrire votre première application console !

Chapitre 2 : les erreurs

Durant la compilation du script, celui-ci est entièrement vérifié
et contrôlé, ce qui peut faire apparaitre des erreurs de syntaxe.

Par exemple si vous aviez oublié l'accolade finale } à la fin,
vous auriez eu, lors de la compilation, un message d'erreur
qui vous indique exactement où le problème se situe :

```
ERROR: instruction expected
AT LINE: 10 COLUMN: 1
GENERATED AT: 15/07/2008 13:42:00

        1 |
        2 | // mon premier script console
        3 |
        4 | proc main ()
        5 | {
        6 |   screen (160, 16);        // mettre l'écran en 160 x 16
        7 |   print ("Bonjour !");     // dire bonjour
        8 |   sync ();                 // afficher à l'écran
        9 |   sleep (5000);            // attendre 5 secondes
       10 |
---------^
***ERROR: instruction expected
```

Il suffit alors de corriger le script et de ré-essayer.

Chapitre 3 : style

Pour éviter les erreurs de syntaxe et aussi faciliter la lecture,
il est important d'avoir un style d'écriture clair.

Exemple :

```
// mon 2ème script

proc main ()
{
   var i, j, l$, e;

   screen (640, 480);

   for (j=0; j<16; j++)
```

```
   {
      l$ = "";
      for (i=0; i<16; i++)
      {
         e = j*16 + i;
         l$ = l$ + chr$(e);
      }
      print (l$);
      nl();
   }

   sync();
   sleep (5000);
}
```

Vous voyez, dans l'exemple ci-dessus, que les lignes entres accolades { } sont à chaque fois décalées de deux blancs à droite. C'est une habitude que vous devez absolument prendre, cela vous évite d'oublier des accolades !

Le script suivant fonctionne aussi mais est tout à fait déconseillé parce qu'on n'y comprend plus rien !

Exemple :

```
// mon 3ème script (déconseillé)

proc main ()
{
var i, j, l$, e;
screen (640, 480);
for (j=0; j<16; j++)
{
l$ = "";
for (i=0; i<16; i++)
{
e = j*16 + i;
l$ = l$ + chr$(e);
}
print (l$);
nl();
}
sync();
sleep (5000);
}
```

Quand vous postez un script sur un groupe ou un forum, changez le font en "Courier New" car avec ce font les décalages sont conservés.

l'importance des commentaires

N'hésitez pas, pour clarifier un script, à ajouter plein de lignes de commentaires

n'importe où dans votre script. Il suffit
pour cela de commencer une ligne par //

Ces lignes sont ignorées par la compilation.

Exemple:

```
// voici mon premier script
// que j'ai créé pour animer
// un peu ma salle.

// Copyright le scripteur

proc main ()
{
   screen (160, 16);        // mettre l'écran en 160 x 16

   // la commande print va afficher un texte sur le chat !
   print ("Salut les amis !");

   sync ();                 // afficher à l'écran
   sleep (5000);            // attendre 5 secondes
}
```

Chapitre 4 : les variables numériques

Une variable est une case de la mémoire RAM du PC à laquelle on donne un nom.
Chaque variable contient une valeur numérique entre -2147483648 et +2147483647.
Celle valeur change généralement au fur et à mesure que le programme se déroule.

Pour créer une variable, il faut la 'déclarer', c'est-à-dire réserver
de la place mémoire pour son contenu, à l'aide de la commande 'var'.

exemple:

```
   var compteur;    // je crée mon compteur qui contient zéro au début
```

A leur création, les variables reçoivent toujours la valeur zéro.

Pour changer la valeur d'une variable, on écrit une 'assignation'
qui ressemble à ceci :

exemple:

```
   compteur = 2;   // maintenant, compteur a la valeur 2
```

Pour changer la valeur d'une variable, il suffit de lui donner
une nouvelle valeur :

exemple:

```
compteur = 5;   // maintenant, compteur a la valeur 5
```

On peut faire des calculs aussi. Dans les exemples suivants, on peut voir :
- à droite du signe égal la valeur à calculer,
- à gauche du signe égal la variable où sera stocké le résultat du calcul.

exemples:

```
compteur = 2 + 5*7;      // maintenant, compteur a la valeur 37
compteur = (5 + 4) * 7;  // maintenant, compteur a la valeur 49
compteur = -45;          // maintenant, compteur a la valeur -45
compteur = 100 - 101;    // maintenant, compteur a la valeur -1
```

Voici comment augmenter la valeur d'une variable de 1 :

exemple:

```
compteur = compteur + 1;     // calcule compteur + 1 et stocke le résultat dans compteur
```

Il existe cependant une manière abbréviée de faire la même chose, l'incrémentation :

exemple:

```
compteur++;     // augmente compteur de 1
```

De la même façon, pour diminuer une variable de 1, on fait une décrémentation.

exemple:

```
compteur--;     // diminue compteur de 1
```

Voici comment augmenter la valeur d'une variable de 10 :

exemple:

```
compteur = compteur + 10;     // calcule compteur + 10 et stocke le résultat dans compteur
```

Il existe cependant une manière abbréviée de faire la même chose, l'assignation avec addition :

exemple:

```
compteur += 10;     // augmente compteur de 10
```

De la même façon, pour diminuer une variable de 10 :

exemple:

```
compteur -= 10;     // diminue compteur de 10
```

Chapitre 5 : les variables chaines de caractères

Il existe une 2ème sorte de variable : les variables chaines de caractères, également appelées 'string' en anglais.

Celles-ci ne contiennent pas de nombres mais des chaines de caractères (lettres, chiffres, ponctuation, etc ..).

On reconnait une variable chaine de caractère au signe dollar ($) qui est ajouté à la fin de son nom.

exemple:

```
var nom$;        // je crée une variable string
```

Pour changer la valeur d'une variable string, on fait une 'assignation' qui ressemble à ceci :

exemple:

```
nom$ = "Bonjour";     // maintenant, nom$ a la valeur "Bonjour"
```

Pour changer la valeur d'une variable, il suffit de lui donner une nouvelle valeur :

exemple:

```
nom$ = "Bonsoir";      // maintenant, nom$ a la valeur "Bonsoir"
```

On peut coller plusieurs chaines ensembles avec l'opérator "+".

exemple:

```
nom$ = "Bon" + "soir";       // maintenant, nom$ a la valeur "Bonsoir"
nom$ = nom$ + " mes amis";   // maintenant, nom$ a la valeur "Bonsoir mes amis"
```

Attention: la longueur d'une variable string ne peut pas dépasser 8192 caractères.

Lors de leur création, les variables string contienent une chaine vide
(donc de 0 caractères), ce qui est équivalent à l'exemple suivant.

exemple:

```
nom$ = "";       // maintenant, nom$ est vide
```

Chapitre 6 : conversions

On peut convertir une variable numérique en variable string à l'aide
de la fonction str$()

exemple:

```
var compteur, nom$;

compteur = 12;

nom$ = str$(compteur);    // maintenant, nom$ contient "12"

nom$ = "compteur = " + str$(compteur);    // maintenant, nom$ contient "compteur = 12"
```

A l'inverse, on peut convertir une variable string en variable numérique
à l'aide de la fonction val()

exemple:

```
var compteur, nom$;

nom$ = "12";

compteur = val (nom$);      // maintenant, compteur égale 12

compteur = val ("13");      //  maintenant, compteur égale 13

compteur = val ("ab");      //  ceci va donner une erreur qui arrête le script,
                                    //  car "ab" ne peut être converti en valeur numérique !
```

Chapitre 7 : affichages à l'écran

L'affichage à l'écran du chat se fait à l'aide de la procédure print()

exemple:

```
print ("Bonjour !");
print (nom$);
```

Il n'est pas permis de donner à print() une variable numérique, par exemple ceci ne va pas fonctionner :

exemple:

```
print (compteur);        // erreur
```

Pour afficher la valeur de compteur, il faut le convertir d'abord en string :

exemple:

```
print (str$(compteur));
print ("la valeur de compteur égale " + str$(compteur));
```

Chapitre 8 : les tableaux

A la place de réserver chaque variable séparemment, on peut réserver tout un tableau de variables.

exemple:

```
var compteur[100];       // je crée 101 compteurs
```

En fait, l'exemple ci-dessus crée les 101 compteurs suivants :

```
compteur[0]
compteur[1]
compteur[2]
compteur[3]
...
compteur[100]
```

On peut stocker des valeurs dans ces compteurs, comme ceci :

exemple:

```
compteur[5] = 12;    // stocker 12 dans le compteur numéro 5

compteur[5]++;       // augmenter le compteur 5 de 1

compteur[i] = 13;    // stocker 13 dans le compteur i
```

Le dernier exemple ci-dessus va donner une erreur si la variable "i" n'a pas de valeur comprise entre 0 et 100.

Chapitre 9 : l'instruction IF

Les conditions servent à "tester" des variables.

On peut par exemple dire "Si la variable machin est égale à 50, fais ceci"...

Mais ce serait dommage de ne pouvoir tester que l'égalité !

Il faudrait aussi pouvoir tester si la variable est inférieure à 50,
inférieure ou égale à 50, supérieure, supérieure ou égale...

Ne vous inquiétez pas, le langage script a tout prévu (mais vous n'en doutiez pas hein)

Avant de voir comment on écrit une condition de type "if... else",
il faut donc que vous connaissiez 2-3 symboles de base.
Ces symboles sont indispensables pour réaliser des conditions.

Voici un petit tableau de symboles à connaître par coeur :

Symbole	Signification
==	Est égal à
>	Est supérieur à
<	Est inférieur à
>=	Est supérieur ou égal à
<=	Est inférieur ou égal à
!=	Est différent de

Faites très attention, il y a bien 2 symboles "==" pour tester l'égalité.
Une erreur courante que font les débutants et de ne mettre qu'un symbole =.
Je vous en reparlerai un peu plus bas.

Un if simple

Attaquons maintenant sans plus tarder. Nous allons faire un test simple, qui va dire à l'ordinateur :

SI la variable vaut ça
ALORS fais ceci

Par exemple, on pourrait tester une variable "age" qui contient votre âge.
Tenez pour s'entraîner, on va tester si vous êtes majeur,
c'est-à-dire si votre âge est supérieur ou égal à 18 :

exemple:

```
if (age >= 18)
{
   print ("Vous êtes majeur !");
}
```

Le symbole >= signifie "Supérieur ou égal", comme on l'a vu dans le tableau tout à l'heure.

S'il n'y a qu'une instruction entre les accolades, alors celles-ci deviennent facultatives.
Vous pouvez donc écrire :

exemple:

```
if (age >= 18)
   print ("Vous êtes majeur !");
```

Si vous voulez tester les codes précédents pour voir comment le if fonctionne,
il faudra placer le if à l'intérieur d'une procédure main et ne pas oublier de déclarer
une variable age à laquelle on donnera la valeur de notre choix.

Ca peut paraître évident pour certains, mais apparemment ça ne l'était pas pour tout le monde
aussi ai-je rajouté cet exemple :

exemple:

```
proc main ()
{
   var age;

   screen (640, 480);

   age = 20;

   if (age >= 18)
      print ("Vous êtes majeur !");

   sync ();
   sleep (5000);
}
```

Ici, la variable age vaut 20, donc le "Vous êtes majeur !" s'affichera.

Essayez de changer la valeur initiale de la variable pour voir.
Mettez par exemple 15 : la condition sera fausse, et donc "Vous êtes majeur !" ne s'affichera
pas cette fois.

Servez-vous de ce code de base pour tester les prochains exemples du chapitre.

Une question de propreté

La façon dont vous ouvrez les accolades n'est pas importante,
votre programme marchera aussi bien si vous écrivez tout sur une même ligne.

Par exemple :

```
if (age >= 18) {  print ("Vous etes majeur !"); }
```

Pourtant, même si c'est possible d'écrire comme ça, c'est ultra déconseillé !!
En effet, tout écrire sur une même ligne rend votre code difficilement lisible.

Si vous ne prenez pas dès maintenant l'habitude d'aérer votre code,
plus tard quand vous écrirez de plus gros programmes vous ne vous y retrouverez plus !

Essayez donc de présenter votre code source de la même façon que moi :

une accolade sur une ligne, puis vos instructions (précédées de deux blancs pour les "décaler vers la droite"), puis l'accolade de fermeture sur une ligne.

Il existe plusieurs bonnes façons de présenter son code source.

Ca ne change rien au fonctionnement du programme final, mais c'est une question de "style informatique" si vous voulez. Si vous voyez un code de quelqu'un d'autre présenté un peu différemment, c'est qu'il code avec un style différent.

Le principal, c'est que son code reste aéré et lisible.

Le "else" pour dire "sinon"

Maintenant que nous savons faire un test simple, allons un peu plus loin : si le test n'a pas marché (il est faux), on va dire à l'ordinateur d'exécuter d'autres instructions.

En français, nous allons donc écrire quelque chose qui ressemble à cela :

SI la variable vaut ça
ALORS fais ceci
SINON fais cela

Il suffit de rajouter le mot else après l'accolade fermante du if.
Petit exemple :

```
if (age >= 18) // Si l'âge est supérieur ou égal à 18
{
   print ("Vous etes majeur !");
}
else   // Sinon...
{
   print ("Ah c'est bete, vous etes mineur !");
}
```

Les choses sont assez simples : si la variable age est supérieure ou égale à 18, on affiche le message "Vous êtes majeur !", sinon on affiche "Vous êtes mineur".

Le "else if" pour dire "sinon si"

On a vu comment faire un "si" et un "sinon".
Il est possible aussi de faire un "sinon si".

On dit dans ce cas à l'ordinateur :

SI la variable vaut ça ALORS fais ceci
SINON SI la variable vaut ça ALORS fais ça
SINON fais cela

Traduction:

```
if (age >= 18)       // Si l'âge est supérieur ou égal à 18
```

```
{
   print ("Vous etes majeur !");
}
else if (age > 4)    // Sinon, si l'âge est au moins supérieur à 4
{
   print ("Bon t'es pas trop jeune quand meme...");
}
else                 // Sinon...
{
   print ("Aga gaa aga gaaa gaaa");   // Langage Bébé, vous ne pouvez pas comprendre ;o)
}
```

L'ordinateur fait les tests dans l'ordre :

D'abord il teste le premier if : si la condition est vraie, alors il exécute
ce qui se trouve entre les premières accolades.

Sinon, il va au "sinon si" et fait à nouveau un test : si ce test est vrai,
alors il exécute les instructions correspondantes entre accolades.

Enfin, si aucun des tests précédents n'a marché, il exécute les instructions du "sinon".

Le "else" et le "else if" ne sont pas obligatoires. Pour faire une condition,
il faut juste au moins un "if" (logique me direz-vous, sinon il n'y a pas de condition !)

Notez qu'on peut mettre autant de "else if" que l'on veut. On peut donc écrire :

SI la variable vaut ça
ALORS fais ceci
SINON SI la variable vaut ça ALORS fais ça
SINON SI la variable vaut ça ALORS fais ça
SINON SI la variable vaut ça ALORS fais ça
SINON fais cela

Plusieurs conditions à la fois

Il peut aussi être utile de faire plusieurs tests à la fois dans votre if.
Par exemple, vous voudriez tester si l'âge est supérieur à 18 ET si l'âge est inférieur à 25.
Pour faire cela, il va falloir utiliser de nouveaux symboles :

Symbole	Signification
&&	ET
||	OU
!	NON

Test ET

Si on veut faire le test que j'ai mentionné plus haut, il faudra écrire :

exemple:

```
if (age > 18 && age < 25)
```

Les deux symboles "&&" signifient ET.

Notre condition se dirait en français :

"Si l'âge est supérieur à 18 ET si l'âge est inférieur à 25"

Test OU

Pour faire un OU, on utilise les 2 signes ||. Je dois avouer que ce signe n'est pas facilement accessible sur nos claviers. Pour le taper sur un clavier AZERTY français, il faudra faire Alt Gr + 6. Sur un clavier belge, il faudra faire Alt Gr + &.

Imaginons un programme débile qui décide si une personne a le droit d'ouvrir un compte en banque. C'est bien connu, pour ouvrir un compte en banque, il vaut mieux ne pas être trop jeune (on va dire arbitrairement qu'il faut avoir au moins 30 ans) ou bien avoir plein d'argent (parce que là même à 10 ans on vous acceptera à bras ouverts)

Notre test pour savoir si le client a le droit d'ouvrir un compte en banque pourrait être :

```
if (age > 30 || argent > 100000)
{
   print ("Bienvenue chez Picsou Banque !");
}
else
{
   print ("Hors de ma vue, misérable !");
}
```

Ce test n'est valide que si la personne a plus de 30 ans ou si elle possède plus de 100 000 euros

Test NON

Le dernier symbole qu'il nous reste à tester est le point d'exclamation.
En informatique, le point d'exclamation signifie "Non".
Vous devez mettre ce signe avant votre condition pour dire "Si cela n'est pas vrai" :

exemple:

```
if (!(age < 18))
```

Cela pourrait se traduire par "Si la personne n'est pas mineure".
Si on avait enlevé le "!" devant, cela aurait signifié l'inverse : "Si la personne est mineure".

Quelques erreurs courantes de débutant

N'oubliez pas les 2 signes ==

Si on veut tester si la personne a tout juste 18 ans, il faudra écrire :

exemple:

```
if (age == 18)
{
   print ("Vous venez de devenir majeur !");
}
```

N'oubliez pas de mettre 2 signes "égal" dans un if, comme ceci : ==

Le point-virgule de trop

Une autre erreur courante de débutant : vous mettez parfois un point-virgule
à la fin de la ligne d'un if. Or, un if est une condition, et on ne met de point-virgule
qu'à la fin d'une instruction et non d'une condition.
Le code suivant ne marchera pas comme prévu car il y a un point-virgule à la fin du if :

exemple:

```
if (age == 18); // Notez le point-virgule ici qui ne devrait PAS être là
{
   print ("Tu es tout juste majeur");
}
```

Les booléens, le coeur des conditions

Nous allons maintenant rentrer plus en détail dans le fonctionnement d'une condition de type
if... else.
En effet, les conditions font intervenir quelque chose qu'on appelle les booléens en
informatique.

C'est un concept très important, donc ouvrez grand vos oreilles (euh vos yeux plutôt)

Quelques petits tests pour bien comprendre

En cours de Physique-Chimie, mon prof avait l'habitude de nous faire commencer
par quelques petites expériences avant d'introduire une nouvelle notion.
Je vais l'imiter un peu aujourd'hui.

Voici un code source très simple que je vous demande de tester :

exemple:

```
if (1)
{
   print ("C'est vrai");
}
else
{
   print ("C'est faux");
}
```

Résultat :

```
C'est vrai
```

Mais ??? On n'a pas mis de condition dans le if, juste un nombre.
Qu'est-ce que ça veut dire ça n'a pas de sens ?

Si ça en a, vous allez comprendre.
Faites un autre test maintenant en remplaçant le 1 par un 0 :

exemple:

```
if (0)
{
   print ("C'est vrai");
}
else
{
   print ("C'est faux");
}
```

Résultat :

```
C'est faux
```

Faites maintenant d'autres tests en remplaçant le 0 par n'importe quel
autre nombre entier, comme 4, 15, 226, -10, -36 etc...
Qu'est-ce qu'on vous répond à chaque fois ? On vous répond : "C'est vrai".

Résumé de nos tests : si on met un 0, le test est considéré comme faux,
et si on met un 1 ou n'importe quel autre nombre, le test est vrai.

Des explications s'imposent

En fait, à chaque fois que vous faites un test dans un if,
ce test renvoie la valeur 1 s'il est vrai, et 0 s'il est faux.

Par exemple :

```
if (age >= 18)
```

Ici, le test que vous faites est "age >= 18".

Supposons que age vaille 23. Alors le test est vrai,
et l'ordinateur "remplace" en quelque sorte "age >= 18" par 1.

Ensuite, l'ordinateur obtient (dans sa tête) un "if (1)".

Quand le nombre est 1, comme on l'a vu, l'ordinateur dit que la condition
est vraie, donc il affiche "C'est vrai" !

De même, si la condition est fausse, il remplace age >= 18 par le nombre 0,
et du coup la condition est fausse : l'ordinateur va lire les instructions du "else".

Un test avec une variable

Testez maintenant un autre truc : envoyez le résultat de votre condition dans une variable, comme si c'était une opération (car pour l'ordinateur, c'est une opération !).

exemple:

```
var age, majeur;

age = 20;

majeur = age >= 18;

print ("majeur vaut : " + str$(majeur));
```

Comme vous le voyez, la condition age >= 18 a renvoyé le nombre 1 car elle est vraie.
Du coup, notre variable majeur vaut 1, on vérifie d'ailleurs ça en faisant un print
qui montre bien qu'elle a changé de valeur.

Faites le même test en mettant age = 10 par exemple. Cette fois, majeur vaudra 0.

Cette variable "majeur" est un booléen !

Retenez bien ceci :

On dit qu'une variable à laquelle on fait prendre les valeurs 0 et 1 est un booléen.

Et aussi ceci :

0 = Faux
1 = Vrai

Pour être tout à fait exact, 0 = faux et tous les autres nombres valent vrai
(on a eu l'occasion de le tester plus tôt).

Ceci dit, pour simplifier les choses on va se contenter de n'utiliser que
les chiffres 0 et 1, pour dire si "quelque chose est faux ou vrai".

Les booléens dans les conditions

Souvent, on fera un test "if" sur une variable booléenne :

exemple:

```
var majeur;

majeur = 1;

if (majeur)
{
   print ("Tu es majeur !");
}
else
```

```
{
   print ("Tu es mineur");
}
```

Comme majeur vaut 1, la condition est vraie, donc on affiche "Tu es majeur !".

Ce qui est très pratique, c'est que la condition se lit facilement par un être humain.
On voit "if (majeur)", ce que peut traduire par "Si tu es majeur".

Les tests sur des booléens sont donc faciles à lire et à comprendre,
pour peu que vous ayez donné des noms clairs à vos variables
comme je vous ai dit de le faire depuis le début.

Tenez, voici un autre test imaginaire :

exemple:

```
if (majeur && garcon)
```

Ce test signifie "Si tu es majeur ET que tu es un garçon".
garcon est ici une autre variable booléenne qui vaut 1 si vous êtes un garçon,
et 0 si vous êtes... une fille ! Bravo, vous avez tout compris !

Les booléens servent donc à exprimer si quelque chose est vrai ou faux.
C'est très utile, et ce que je viens de vous expliquer vous permettra
de comprendre bon nombre de choses par la suite.

Petite question : si on fait le test "if (majeur == 1)", ça marche aussi non ?

Tout à fait. Mais le principe des booléens c'est justement de raccourcir
l'expression du if et de la rendre plus facilement lisible.
Avouez que "if (majeur)" ça se comprend très bien non ?

Retenez donc : si votre variable est censée contenir un nombre,
faites un test sous la forme "if (variable == 1)".
Si au contraire votre variable est censée contenir un booléen (c'est-à-dire
soit 1 soit 0 pour dire vrai ou faux), faites un test sous la forme "if (variable)".

La condition "if... else" que l'on vient de voir est le type de condition le plus souvent utilisé.
En fait, il n'y a pas 36 façons de faire une condition. Le "if... else" permet de gérer tous les cas.

exemple:

```
if (age == 2)
{
   print ("Salut bebe !");
}
else if (age == 6)
{
   print ("Salut gamin !");
}
else if (age == 12)
{
```

```
    print ("Salut jeune !");
}
else if (age == 16)
{
    print ("Salut ado !");
}
else if (age == 18)
{
    print ("Salut adulte !");
}
else if (age == 68)
{
    print ("Salut papy !");
}
else
{
    print ("Je n'ai aucune phrase de prete pour ton age ");
}
```

Chapitre 10 : les boucles
==========

Qu'est-ce qu'une boucle ?
C'est une technique permettant de répéter les mêmes instructions plusieurs fois.

Tout comme pour les conditions, il y a plusieurs façons de réaliser des boucles.
Au bout du compte, cela revient à faire la même chose :
répéter les mêmes instructions un certain nombre de fois.

Dans tous les cas, le schéma est le même :

```
        <----------
Instructions    ^
Instructions    ^
Instructions    ^
Instructions    ^
        ----------^
```

Voici ce qu'il se passe dans l'ordre :

L'ordinateur lit les instructions de haut en bas (comme d'habitude)
Puis, une fois arrivé à la fin de la boucle, il repart à la première instruction
Il recommence alors à lire les instructions de haut en bas...
... Et il repart au début de la boucle.

Le problème dans ce système c'est que si on ne l'arrête pas,
l'ordinateur est capable de répéter les instructions à l'infini !
Il n'est pas du genre à se plaindre vous savez, il fait ce qu'on lui dit de faire ..

Et c'est là qu'on retrouve... des conditions !
Quand on crée une boucle, on indique toujours une condition.

Cette condition signifiera "Répète la boucle tant que cette condition est vraie".

Il y a plusieurs manières de s'y prendre comme je vous l'ai dit.
Voyons voir sans plus tarder comment on réalise une boucle de type "while".

La boucle while

Voici comment on construit une boucle while :

exemple:

```
while ( Condition )
{
      // Instructions à répéter
}
```

C'est aussi simple que cela. While signifie "Tant que".
On dit donc à l'ordinateur "Tant que la condition est vraie : répète les instructions entre accolades".

On veut que notre boucle se répète un certain nombre de fois.
On va pour cela créer une variable "compteur" qui vaudra 0 au début du programme et que l'on va incrémenter au fur et à mesure.

Vous vous souvenez de l'incrémentation ? Ca consiste à ajouter 1 à la variable en faisant "variable++;".

Regardez attentivement ce bout de code et, surtout, essayez de le comprendre :

exemple:

```
var compteur;

compteur = 0;

while (compteur < 5)
{
   print ("Salut !");
   compteur++;
}
```

Résultat :

```
Salut !
Salut !
Salut !
Salut !
Salut !
```

Ce code répète 5 fois l'affichage de "Salut !".

Comment ça marche exactement ?

Au départ, on a une variable compteur initialisée à 0. Elle vaut donc 0 au début du programme.

La boucle while ordonne la répétition TANT QUE compteur est inférieur à 5.
Comme compteur vaut 0 au départ, on rentre dans la boucle.

On affiche la phrase "Salut !" via un print.

On incrémente la valeur de la variable compteur, grâce à l'instruction "compteur++;".
Compteur valait 0, il vaut maintenant 1.
On arrive à la fin de la boucle (accolade fermante), on repart donc au début,
au niveau du while.

On refait le test du while : "Est-ce que compteur est toujours inférieur à 5 ?".
Ben oui, compteur vaut 1 Donc on recommence les instructions de la boucle.

Et ainsi de suite... Compteur va valoir progressivement 0, 1, 2, 3, 4, 5.

Lorsque compteur vaut 5, la condition "compteur < 5" est fausse.
Comme l'instruction est fausse, on sort de la boucle.

On pourrait voir d'ailleurs que la variable compteur augmente au fur
et à mesure dans la boucle, en l'affichant dans le print :

exemple:

```
var compteur;

compteur = 0;

while (compteur < 5)
{
   print ("la variable compteur vaut " + str$(compteur));
   compteur++;
}
```

```
La variable compteur vaut 0
La variable compteur vaut 1
La variable compteur vaut 2
La variable compteur vaut 3
La variable compteur vaut 4
```

Voilà, si vous avez compris ça, vous avez tout compris.
Vous pouvez vous amuser à augmenter la limite du nombre de boucles
("< 10" au lieu de "< 5").

Attention aux boucles infinies

Lorsque vous créez une boucle, assurez-vous toujours qu'elle peut s'arrêter
à un moment ! Si la condition est toujours vraie, votre programme ne s'arrêtera jamais !

Voici un exemple de boucle infinie :

exemple:

```
while (1)
{
   print ("Boucle infinie");
```

```
}
```

Souvenez-vous des booléens : 1 = vrai, 0 = faux. Ici, la condition est toujours vraie, donc ce programme affichera "Boucle infinie" sans arrêt !

Pour arrêter un tel programme, vous n'avez pas d'autre choix que de fermer le chat !

Faites donc très attention : évitez à tout prix de tomber dans une boucle infinie.

La boucle for

En théorie, la boucle while permet de réaliser toutes les boucles que l'on veut. Toutefois, il est dans certains cas utiles d'avoir un autre système de boucle plus "condensé", plus rapide à écrire.

Les boucles for sont très très utilisées en programmation.
Je n'ai pas de statistiques sous la main, mais sachez que vous utiliserez certainement autant de for que de while, donc il vous faudra savoir manipuler ces deux types de boucles.

Comme je vous le disais, les boucles for sont juste une autre façon de faire une boucle while.

Voici un exemple de boucle while que nous avons vu tout à l'heure :

exemple:

```
var compteur;

compteur = 0;

while (compteur < 10)
{
   print ("Salut !");
   compteur++;
}
```

Voici maintenant l'équivalent en boucle for :

exemple:

```
var compteur;

for (compteur = 0 ; compteur < 10 ; compteur++)
{
   print ("Salut !");
}
```

Quelles différences ?

Il y a beaucoup de choses entre les parenthèses après le for (nous allons détailler ça après)

Il n'y a plus de compteur++; dans la boucle.

Intéressons-nous à ce qui se trouve entre les parenthèses, car c'est là que réside
tout l'intérêt de la boucle for. Il y a 3 instructions condensées,
séparée chacune par un point-virgule :

La première est l'initialisation : cette première instruction est utilisée
pour préparer notre variable compteur. Dans notre cas, on initialise la variable à 0.

La seconde est la condition : comme pour la boucle while, c'est la condition qui dit
si la boucle doit être répétée ou pas. Tant que la condition est vraie, la boucle for continue.

Enfin, il y a l'incrémentation : cette dernière instruction est exécutée à la fin
de chaque tour de boucle pour mettre à jour la variable compteur.
La quasi-totalité du temps on fera une incrémentation, mais on peut aussi
faire une décrémentation (variable--;) ou encore n'importe quelle autre opération
(variable += 2; pour avancer de 2 en 2 par exemple).

Bref, comme vous le voyez la boucle for n'est rien d'autre qu'un condensé du while.
Sachez vous en servir, vous en aurez besoin plus d'une fois !

Chapitre 11 : procédures et fonctions de base

Qu'est-ce qu'une procédure ? ben par exemple "print" que vous connaissez bien.
Mais il en existe bien d'autres !

Vous connaissez déjà deux fonctions aussi : str$ et val, qui permettent
de convertir entre string et valeur numérique.

En fait la différence entre procédure et fonction, c'est qu'une procédure
c'est une commande, tandis qu'une fonction ça calcule quelque chose et le renvoie
comme résultat.

Nous allons étudier quelques fonctions courantes pour voir comment ça marche :

```
d$ = now$ ();          // fonction qui calcule la date du jour
                              // exemple: now$() == "2007/01/17 12:23:12"

s$ = mid$(s$,a,N);      // renvoie N caractères de s$ commençant au 'a'ième
                              // exemple: mid$("ABCDEF",2,3) == "BCD"

n = random (a,b);       // renvoie un nombre aléatoire entre a et b
                              // exemple: random(1,50) == 12
```

La fonction now$() est simple : elle renvoie la date et l'heure du PC.
Par exemple on peut écrire.

exemple:

```
   var ma_date$;

   ma_date$ = now$();   // mettre la date dans ma_date$

   print (ma_date$);
```

ce qui va imprimer par exemple:

```
2007/01/17 12:23:12
```

La fonction mid$() est très utile pour extraire un sous-string d'un string.
Par exemple imaginez que je veuille extraire juste l'heure de la date ci-dessus.

exemple:

```
var ma_date$, heure$;

ma_date$ = now$();   // mettre la date dans ma_date$

heure$ = mid$ (ma_date$, 12, 2);   // prendre une copie de la date à partir du 12 ième caractère,
                                                    // et copier 2 caractères
print (heure$);
```

ce qui va imprimer par exemple:

```
12
```

La fonction random() permet de tirer un nombre au hasard, ce qui est très utile pour les jeux.

exemple:

```
var i;

i = random (1, 100);   // choisir un nombre de 1 à 100 à mettre dans i

print ("on choisit " + str$(i));
```

Dans la documentation de la console, vous trouverez des listes
de toutes les procédures et fonctions existantes.
