Programmation en QBASIC

par Philippe-Henry Marcy

2003 - 2004

Table des matières

Avant-propos

Ce document est destiné à ceux qui n'ont jamais programmé. Il permet l'apprentissage des bases de la programmation à travers un langage simple : le BASIC. Les concepts présentés sont assez simplifiés et peu approfondis. Presque aucune connaissance n'est requise, que ce soit en mathématiques ou même en informatique.

Le logiciel utilisé pour concevoir des programmes est QBASIC (abréviation de Quick BASIC). Il fonctionne sous le système d'exploitation DOS, et peut aussi être utilisé avec Windows. Le but de ce cours n'est pas de vous permettre d'utiliser QBASIC au maximum de ses possibilités, qui sont d'ailleurs restreintes en comparaison avec la plupart des autres logiciels de conception. Il est simplement destiné à vous familiariser avec la programmation. Ainsi les possibilités réduites qu'offre QBASIC vous seront largement suffisantes.

Le cours est organisé en sept chapitres, qui sont désignés par des nombres romains. Chacun d'eux comprend plusieurs sections, repérées par des lettres majuscules. Certaines sections sont elles-mêmes divisées en sous-parties. Les sous-parties sont identifiées à l'aide de numéros.

Le premier chapitre du cours est réservé au fonctionnement de QBASIC, et n'aborde pas le code. Il décrit l'interface de l'éditeur, et explique ce qu'il faut savoir avant de pouvoir se lancer dans l'écriture de programmes.

Tout le reste du document concerne le code BASIC. Dans un premier temps sont présentés les différents éléments qui servent à écrire le code, afin de familiariser le débutant. Puis, la méthode d'association de ces éléments pour concevoir des applications est traitée à travers l'étude d'un exemple.

L'étape suivante est un approfondissement dans les bases de la programmation. Elle se divise en deux chapitres : l'un traite des constructions, et l'autre présente les commandes concernant l'interface visuelle entre le programme et l'utilisateur. Le sixième chapitre présente trois exemples de programmes graphiques qui illustrent les notions abordées aux deux chapitres précédents.

Enfin, dans le dernier chapitre de ce cours sont expliquées quelques commandes utiles du BASIC, qui s'avèrent souvent nécessaires en programmation, en particulier pour la conception de jeux.

Des explications détaillées sont données tout au long des chapitres pour faciliter la compréhension, et sont toutes illustrées par des exemples. Si, malgré cela, certains points restent obscurs pour vous, voici quelques conseils.

Après avoir terminé la lecture d'une section, assurez-vous d'avoir bien compris les notions que celle-ci explique avant de commencer à lire la suivante. S'il le faut, relisez-la plusieurs fois. Ensuite, mettez ces notions en pratique en écrivant des programmes. En effet, leur assimilation est souvent nécessaire à la compréhension des sections suivantes. C'est aussi valable pour les différentes sous-parties d'une section.

De la même façon, chaque chapitre de ce cours reprend des notions expliquées dans les chapitres précédents. Ainsi, il est impératif de lire tous les chapitres dans l'ordre, et de lire entièrement un chapitre avant de commencer le suivant. Malgré cela, il est probable que vous oublierez en partie certaines notions au fur et à mesure de votre lecture. Encore et toujours, le meilleur moyen de les mémoriser est d'écrire soi-même des programmes qui les appliquent.

La création de programmes est une discipline passionnante, à condition cependant de se soumettre à certaines règles. Ces règles vous seront précisées au fur et à mesure des chapitres. Elles peuvent paraître contraignantes, mais obligez-vous à les respecter. Vous vous rendrez compte, à force de programmer, qu'elles vous feront gagner du temps, et que la qualité de vos applications en sera améliorée.

Bonne lecture.

Chapitre I :
Le fonctionnement de QBASIC

Tout d'abord, nous allons nous intéresser à l'interface de l'éditeur de QBASIC, c'est-à-dire le logiciel qui vous permettra d'écrire vos programmes, puis nous verrons quelques astuces pour gagner du temps dans l'écriture du code. Pour ceux qui sont habitués à utiliser un ordinateur, il n'est pas nécessaire de lire ces explications. En revanche, les aspects traités ensuite sont plus importants : ce sont l'exécution et le débogage.

A) L'interface de QBASIC

Lorsque vous lancez le programme « QBASIC.EXE », vous voyez apparaître un écran bleu : c'est l'éditeur de QBASIC. Une fenêtre d'accueil de fond gris s'affiche. En appuyant sur *[Entrée]* vous pouvez consulter l'aide de QBASIC. Vous pourrez aussi l'afficher ultérieurement en déroulant le menu Aide ou en appuyant sur les touches *[Maj]* (symbolisée par une grosse flèche dirigée vers le haut) et *[F1]* simultanément. Pour fermer la fenêtre d'accueil, appuyez sur *[Echap]*.

L'écran bleu est divisé en deux. La partie basse, dans le cadre de laquelle il est affiché le texte « Immédiate », ne vous sera pas utile. La partie haute, qui occupe presque tout l'écran, est la fenêtre de code. C'est ici que vous devez taper toutes les instructions de vos programmes. Le titre qui est affiché en haut dans le cadre est le nom du fichier de code ouvert actuellement dans la fenêtre. Lorsqu'il est affiché « Sans_nom », c'est que vous ne l'avez pas encore sauvegardé. Il est possible de diviser cette fenêtre en deux, en cliquant sur le menu Affichage, puis Diviser la fenêtre. Cela permet de traiter simultanément plusieurs parties de votre fichier de code. Notez qu'il n'est pas possible d'ouvrir deux fichiers en même temps.

Vous pouvez utiliser les commandes du logiciel en cliquant sur les menus situés en haut, ou en utilisant les touches de raccourci correspondantes. La barre grise située en bas, appelée barre d'état, vous renseigne sur les principales touches de raccourci, ainsi que sur la position du curseur dans la fenêtre de code.

B) Création de programmes avec QBASIC

1. Ecriture du code

Comme nous l'avons vu précédemment, vous devrez taper vos instructions dans la fenêtre de code. Etant donné que vous aurez souvent à vous déplacer dans cette fenêtre pour modifier le code que vous avez tapé, il faut que vous vous familiarisiez avec les touches de déplacement et de raccourci.

Les touches de déplacement les plus utiles sont les flèches de direction. Les touches *[Début]* (symbolisée par une flèche dirigée vers le coin en haut à gauche) et *[Fin]* vous permettront de vous déplacer au début ou à la fin d'une ligne, ou bien au début ou à la fin du fichier si vous appuyez en même temps sur la touche *[Control]*. Les touches *[Page précédente]* et *[Page suivante]* (symbolisées par des flèches dirigées vers le haut ou le bas et barrées de trois traits) vous permettront de faire défiler le code page par page.

La touche *[Maj]* peut être utilisée en même temps que les touches vues précédemment pour sélectionner plusieurs caractères ou plusieurs lignes. On peut alors les supprimer, en appuyant sur la touche *[Suppr]*, ou sur la touche *[Retour arrière]* (symbolisée par une longue flèche vers la gauche, et située au-dessus de la touche *[Entrée]*). On peut aussi les couper en appuyant sur les touches *[Maj]* et *[Suppr]* simultanément, ou les copier en appuyant sur les touches *[Control]* et *[Inser]* en même temps. Pour coller un texte coupé ou copié, appuyez simultanément sur les touches *[Maj]* et *[Inser]*.

L'utilisation de ces touches vous évitera de vous servir de la souris, et par conséquent vous fera gagner un temps considérable, ce qui n'est pas négligeable, surtout en programmation.

Le code d'un programme est sauvegardé dans un fichier d'extension « *.BAS* ». Vous pouvez ouvrir ou enregistrer un fichier de code en utilisant les commandes du menu *Fichier*. Notez que la longueur du nom d'un fichier de code ne peut pas excéder 8 lettres.

2. Exécution et débogage

QBASIC étant un « interpréteur », il ne compilera pas vos programmes, mais traduira et exécutera lui-même les instructions : on dit qu'il les interprète. Cela a pour inconvénient que les programmes seront bien plus lents, et que ceux qui voudront les utiliser devront posséder QBASIC. Mais comme le but n'est ici que de créer des programmes très simples, cela ne pose pas de problème. Le moment durant lequel QBASIC interprète le code d'un programme s'appelle « l'exécution ».

Pour exécuter un programme, le fichier de code de celui-ci doit être ouvert dans la fenêtre de code. Vous devez alors utiliser la commande *Démarrer* du menu *Exécution*, ou bien appuyer sur la touche *[F5]*. Lorsque QBASIC atteindra la dernière instruction, il affichera la phrase « Appuyez sur une touche pour continuer » en bas de l'écran. Vous pouvez exécuter vos programmes à n'importe quel moment pour vous assurer de leur bon fonctionnement, et cela même si vous n'avez pas fini d'en écrire le code.

La quantité de code nécessitée par un programme étant importante, il arrivera souvent, soit que vous vous trompiez dans l'orthographe des instructions, soit que vous organisiez de façon incorrecte les éléments qui les composent : on dit alors que vous avez fait une « erreur de syntaxe ». QBASIC vous signale une erreur de syntaxe contenue dans une ligne lorsque vous déplacez le curseur sur une autre ligne. Vous pouvez alors soit choisir de rectifier l'erreur tout de suite, soit décider de le faire plus tard. Les erreurs non corrigées seront signalées à nouveau lorsque vous exécuterez le programme, et devront alors impérativement être rectifiées.

Bien sûr, vous n'arriverez pas toujours à obtenir du premier coup le résultat attendu à l'exécution. C'est pour ça qu'il est conseillé d'exécuter souvent vos programmes pour vérifier s'ils fonctionnent correctement. Lorsque ce n'est pas le cas, il faut que vous les déboguiez, c'est-à-dire que vous corrigiez les erreurs (bogues).

Il existe deux types d'erreurs survenant lors de l'exécution. La première catégorie est celle des erreurs qui sont signalées par QBASIC. Elles sont faciles à corriger, puisque l'on sait où elles se trouvent. Notez que contrairement à la plupart des logiciels de conception, QBASIC n'interrompt pas obligatoirement l'exécution du programme lorsqu'il rencontre ce genre d'erreur. Il peut simplement afficher un message d'erreur à l'écran, puis poursuivre l'exécution.

Enfin, la seconde catégorie concerne les erreurs logiques. Celles-ci sont les plus difficiles et les plus longues à corriger. En effet, il se peut que le code soit tout à fait correct, et par conséquent que QBASIC ne signale aucune erreur, mais que le programme ne fasse pas ce que vous attendez de lui. Vous devez alors vous-même comprendre ce qui n'a pas fonctionné comme vous le vouliez, et rechercher la ou les instructions erronées. C'est ce qui rend la tâche plus difficile. Prenons un exemple : vous écrivez un programme de calcul. L'ordinateur demande à l'utilisateur d'entrer deux nombres, et il affiche le résultat de leur addition. Mais ce nombre ne correspond pas. Il peut y avoir différentes causes à cela. Il se peut par exemple que vous ayez écrit par inadvertance une commande qui multiplie les nombres au lieu de les additionner, ou bien encore qui additionne le premier nombre à lui-même.

Pour corriger les erreurs survenant à l'exécution, QBASIC met à votre disposition des outils de débogage, bien que ceux-ci soient très rudimentaires. L'essentiel du débogage est l'exécution pas à pas, c'est-à-dire que celle-ci s'interrompt après l'interprétation de chaque commande, afin de trouver plus facilement quelle est celle qui contient une erreur.

Il arrive de temps en temps que l'on ait sans le vouloir écrit un programme qui ne peut pas s'arrêter de lui-même. Il est possible de forcer l'arrêt de l'exécution et de revenir à l'écran de l'éditeur de QBASIC en appuyant simultanément sur les touches *[Control]* et *[Pause]*.

Notez enfin un dernier point qui pourra vous être utile. Sauf exceptions, la plupart d'entre vous utiliseront QBASIC sous Windows. Le fait de démarrer une session DOS sous Windows ne posera aucun problème quant au fonctionnement des programmes que vous allez créer. Cependant, il se peut que certaines commandes ne fonctionnent pas exactement comme ce qui est indiqué dans ce cours. Ceci peut arriver lorsque vous utilisez QBASIC dans une fenêtre Windows. Si vous constatez une erreur de ce genre et que vous souhaitez la faire disparaître, utilisez QBASIC en plein écran. Vous pouvez le faire en cochant l'option « Plein écran » (dans le cadre « Options d'affichage ») dans les propriétés de la fenêtre. Les propriétés peuvent être affichées en cliquant sur l'icône en haut à gauche de la fenêtre.

Chapitre II :
Les éléments qui constituent le code

A présent, nous abordons le code BASIC. Pour commencer, nous allons étudier les éléments qui constituent le code de tout programme. Pour que vous compreniez bien les explications, elles sont toutes illustrées par des exemples. Recopiez ces exemples dans la fenêtre de code de QBASIC, afin de les tester. Puis, après la lecture de chacune des sections de ce chapitre, écrivez et testez vos propres petits programmes, dans le but d'assimiler correctement ce que vous venez d'apprendre.

A) La base du code : les instructions

Le code de tout programme est divisé en de nombreux éléments indépendants : les « instructions ». Une instruction occupe une ligne dans la fenêtre de code. Ainsi, lors de l'écriture du code d'un programme, il suffit d'appuyer sur la touche *[Entrée]* pour commencer une nouvelle instruction.

Chaque instruction constitue une commande. Lors de l'exécution d'un programme, ces commandes sont interprétées séparément par QBASIC, les unes à la suite des autres. L'exécution prend fin lorsque toutes les instructions du code ont été exécutées.

Voici un exemple d'instructions :

```
SCREEN 0
PRINT "Ceci est votre premier programme en QBASIC."
```

Tapez-ces deux instructions dans la fenêtre de code, et exécutez le programme. Vous constatez qu'il ne fait qu'afficher à l'écran le texte « *Ceci est votre premier programme en QBASIC.* ».

Comme vous pouvez le voir, les instructions sont elles-mêmes constituées de plusieurs éléments. Le principal élément que l'on trouve dans les instructions est le « mot-clé ». Les mots **SCREEN** et **PRINT** sont des mots-clés. On les repère facilement dans le code car ils sont écrits en majuscules. Notez que vous pouvez les écrire en minuscules. QBASIC les mettra alors automatiquement en majuscules.

Le second élément qui est utilisé pour constituer les instructions est la « donnée ». On distingue deux « types de données » : le type nombre et le type texte. Notre exemple contient les deux : le nombre 0 dans la première instruction, et le texte « *Ceci est votre premier programme en QBASIC.* », dans la seconde instruction.

Nous traiterons, dans les sections suivantes de ce chapitre, des autres éléments qui constituent les instructions. Pour l'instant, nous n'allons parler que des mots-clés et des données.

Il existe plusieurs catégories de mots-clés. **SCREEN** et **PRINT** font partie de la catégorie principale : les mots-clés qui dirigent les instructions. Ceux-ci doivent toujours être placés en début de ligne. Ils indiquent à QBASIC à quelles commandes correspondent les instructions. Dans notre exemple, le mot-clé **PRINT** précise à QBASIC que la seconde instruction correspond à la commande d'affichage à l'écran. Notez qu'un mot-clé peut être constitué de deux mots, comme le mot-clé **END IF**.

Toute instruction qui contient un de ces mots-clés dépend entièrement de celui-ci : elle doit être complétée et organisée en fonction de lui. Aussi lui donne-t-on le nom du mot-clé. Par exemple, lorsqu'une instruction débute par le mot-clé **SCREEN**, elle doit toujours être complétée par un nombre, placé après le mot-clé. On l'appelle alors « instruction **SCREEN** » (nous reparlerons de cette instruction dans le cinquième chapitre).

Parfois, ce n'est pas un seul mot-clé qui dirige une instruction, mais plusieurs : ils sont associés. L'un des mots-clés se place au début de la ligne, et l'autre peut être placé à différents endroits dans l'instruction. C'est le cas pour l'instruction **IF…THEN**, que nous verrons en détail dans une prochaine section. Elle contient le mot-clé **IF**, qui se place en début de ligne, et le mot-clé **THEN**, qui se place en fin de ligne.

Il est aussi possible que plusieurs mots-clés dirigeant des instructions différentes soient associés. Les instructions sont donc elles aussi associées. Elles forment ce que l'on appelle une « construction ». Ainsi, une instruction **IF…THEN** et une instruction **END IF** associées forment une construction.

Nous verrons plus tard qu'il existe des instructions qui ne sont dirigées par aucun mot-clé. Ce sont alors des « opérateurs » qui les dirigent.

Il existe une deuxième catégorie de mots-clés, qui, eux, ne servent pas à diriger une instruction, mais sont utilisés pour en compléter une. Par exemple, la construction **FOR…NEXT** associe deux instructions : l'une commençant par le mot-clé **FOR** (de la première catégorie) et l'autre ne contenant que le mot-clé **NEXT** (aussi de la première catégorie). L'instruction **FOR** peut être complétée par le mot-clé **STEP** (de la deuxième catégorie), mais ceci n'est pas obligatoire. La construction **FOR…NEXT** sera expliquée dans le quatrième chapitre.

Enfin, la troisième catégorie de mots-clés est la catégorie des « fonctions ». Les fonctions n'ont aucun lien direct avec les instructions, et ne sont associées à aucun autre mot-clé. Par conséquent, leur utilisation est plus libre. Contrairement aux autres mots-clés, elles n'occupent pas une place fixe dans les instructions. Elles seront présentées à la section F.

Dans notre exemple, il n'est pas difficile de repérer les instructions du programme, puisqu'il n'en contient que deux. Mais lorsqu'un programme contient de nombreuses instructions, il peut être utile de les identifier. Pour cela, on peut les numéroter. Les numéros doivent obligatoirement être placés au tout début de chaque ligne, avant même les mots-clés qui dirigent les instructions. Notez que la numérotation des lignes n'est pas obligatoire.

Les lignes de notre exemple peuvent être numérotées ainsi :

```
1 SCREEN 0
2 PRINT "Ceci est votre premier programme en QBASIC."
```

B) Affichage de données (instruction **PRINT**)

Passons maintenant à l'analyse du mot-clé **PRINT**, rencontré dans l'exemple précédent. C'est l'un des plus simples du langage BASIC, et c'est celui que vous utiliserez le plus souvent.

Comme on l'a déjà dit, **PRINT** permet d'afficher des données à l'écran. Ce nom a été choisi pour désigner la commande d'affichage car, en anglais, « print » signifie « écrire en lettres d'imprimerie » (ou « imprimer »), que l'on traduit dans le contexte par « afficher à l'écran ». Vous pourrez constater que, de la même manière, la plupart des mots-clés du code BASIC sont des mots empruntés à l'anglais, ce qui facilite leur mémorisation.

Les données affichées à l'écran peuvent être de type texte ou de type nombre. L'instruction **PRINT** peut aussi n'être complétée par aucune donnée, c'est-à-dire n'être constituée que du mot-clé. Dans ce cas, elle indique à QBASIC qu'il doit sauter une ligne à l'écran lors de l'exécution.

Dans notre exemple, c'est du texte qui est affiché. En programmation, toute donnée de type texte est appelée « chaîne de caractères », c'est-à-dire suite de caractères. Un caractère peut être une lettre, un chiffre, ou bien un symbole (un élément de ponctuation par exemple). En bref, est considéré comme caractère tout ce qui peut être entré au clavier pour former un texte, ainsi qu'une série d'autres symboles dont nous parlerons plus tard.

Vous avez remarqué que le texte est compris entre des guillemets dans l'exemple. En effet, les chaînes de caractères doivent toujours être écrites entre guillemets, afin d'être reconnues par QBASIC.

Il est possible de compléter une instruction **PRINT** par plusieurs données. Elles doivent alors être séparées par des points-virgules. Notez que lorsque toutes les données indiquées dans une instruction **PRINT** ont été affichées, QBASIC passe automatiquement à la ligne suivante de l'écran lors de l'exécution.

Voici quelques exemples d'instructions **PRINT** :

```
1 PRINT 34
2 PRINT "34"
3 PRINT "Il est actuellement"; 17; "heure"; 10
```

Notez que QBASIC affiche un espace avant et après chaque donnée de type nombre. Ainsi, la première instruction affiche à l'écran le nombre 34 précédé d'un espace, alors que la seconde instruction l'affiche contre le bord gauche de l'écran.

Ces données, que le programmeur a écrites dans les instructions **PRINT**, sont appelées des « données constantes ».

C) Stockage de données (variables)

En programmation, il est très utile de pouvoir stocker des données. Pour cela, on utilise ce que l'on appelle des « variables ». Une variable est comme une sorte de case sur une feuille de papier, dans laquelle on peut écrire du texte, ou bien un nombre. De même que les données constantes, les variables sont dites « de type texte », ou « de type nombre », selon qu'elles permettent de stocker du texte ou des nombres.

Pour repérer chaque case que l'on utilise, on lui donne un nom, que l'on écrit en-dessous d'elle sur la feuille de papier. Ce nom doit être composé uniquement de lettres minuscules ou majuscules (à l'exception des lettres accentuées), de chiffres, et de points. Il doit commencer par une lettre.

Pour des raisons de commodité, le nom est généralement choisi en fonction des données que la case est destinée à contenir. Si l'on veut stocker le prix d'un article dans une variable, il est judicieux de l'appeler **`Prix`**, ou bien **`Prix.TTC`**.

Des abrégés sont souvent utilisés pour éviter d'avoir à écrire des noms trop longs : une variable servant à stocker un pourcentage pourra être nommée **`Prcent`**, plutôt que **`Pourcentage`**.

Les deux exemples que l'on vient de voir sont des noms de variables de type nombre. Pour indiquer à QBASIC que la variable que l'on veut utiliser est de type texte, il faut ajouter le symbole **`$`** à la fin de son nom. Une variable stockant un prénom devrait être appelée **`Prenom$`**.

Deux opérations sont possibles sur les variables. D'abord, on peut écrire une donnée dans la case : on dit qu'on « affecte une valeur » à la variable. Ensuite, on peut lire ce qui est écrit dans la case, c'est-à-dire lire la valeur de la variable. Pour pouvoir écrire dans une case déjà remplie, il faut effacer le contenu de cette case. Cela signifie qu'à chaque fois que l'on affecte une valeur à une variable, la valeur qu'elle avait avant est perdue.

La façon la plus courante d'affecter une valeur à une variable est d'utiliser « l'opérateur d'affectation » : l'opérateur **`=`**. Le nom de la variable doit être écrit en début de ligne, et la valeur qui lui est affectée doit être placée à la fin. La valeur affectée peut être une donnée constante ou bien la valeur d'une autre variable. Notez qu'une instruction correspondant à une affectation n'est pas dirigée par un mot-clé.

Voici des exemples d'affectation de variables :

```
1 Nombre.1 = 1.5
2 Prenom$ = "Jean"
3 Age = 28
4 Nombre.2 = Nombre.1
```

Les trois premières lignes affectent des données constantes à des variables. La quatrième ligne affecte à la variable **`Nombre.2`** la valeur de la variable **`Nombre.1`**.

Lorsqu'une valeur a été affectée à une variable, cette valeur peut être lue par une instruction. Par exemple, **`PRINT`** permet d'afficher le contenu d'une variable. Il suffit d'écrire le nom de la variable après le mot-clé, comme si c'était une donnée constante. Plusieurs variables peuvent être utilisées dans une même instruction **`PRINT`**, et elles peuvent être mêlées à des données constantes.

Ajoutons cette ligne à l'exemple :

```
5 PRINT Prenom$; "a"; Age; "ans."
```

On a vu deux sortes de données : les données constantes, et les données variables. Les données constantes sont celles dont la valeur est écrite dans le code, alors que les données variables sont celles dont la valeur est contenue dans une case (une variable). Si vous ne saisissez pas bien la différence, regardez ces lignes :

```
1 PRINT "Ceci est une donnée constante :"; 25
2 PRINT "Ceci est une donnée constante :"; 25
3 Nombre = 1
4 PRINT "Ceci est une donnée variable :"; Nombre
5 Nombre = 2
6 PRINT "Ceci est une donnée variable :"; Nombre
```

Le nombre qu'affiche la première instruction est une donnée constante, puisque sa valeur est écrite dans l'instruction. Cette instruction peut être répétée autant de fois que l'on veut dans le programme, elle affichera toujours la même chose, car la donnée restera la même.

La quatrième instruction affiche une donnée variable. En effet, la sixième instruction, qui est identique à la quatrième, affiche une valeur différente de la variable **`Nombre`**, car la valeur de cette variable n'est pas fixe.

D) Demander à l'utilisateur d'entrer des données au clavier (instruction `INPUT`)

Il arrivera souvent que vous ayez besoin que votre programme demande à l'utilisateur d'entrer des données au clavier, pour pouvoir les utiliser. Cela se fait à l'aide de l'instruction **`INPUT`**. Celle-ci affiche les données à l'écran au fur et à mesure que l'utilisateur les entre au clavier. Il peut corriger une erreur de frappe, en appuyant sur la touche *[Retour arrière]*. Les données sont stockées dans une variable lorsque l'utilisateur appuie sur la touche *[Entrée]*.

Voici un schéma de la manière dont doit être complétée l'instruction **`INPUT`** :

`INPUT [`*`chaîne`*`, ] `*`variable`

`chaîne` et ***`variable`*** sont les éléments qui complètent l'instruction **`INPUT`**. Ce sont les « paramètres » de cette instruction.

`chaîne` est la chaîne de caractères qui est affichée à l'écran avant que l'utilisateur puisse commencer à entrer les données. Les crochets qui l'entourent dans le schéma indiquent qu'il n'est pas obligatoire de compléter l'instruction **`INPUT`** par ce paramètre. C'est la notation conventionnelle qui est utilisée dans les rubriques d'aide de la plupart des logiciels de conception.

`variable` est le nom de la variable dans laquelle sont stockées les données tapées par l'utilisateur.

Voici un exemple d'utilisation simple de **INPUT** :

```
INPUT Nombre
```

Vous pouvez remarquer que QBASIC affiche un point d'interrogation à l'écran devant les données entrées par l'utilisateur.

Voyons à présent un exemple dans lequel on complète l'instruction **INPUT** par le paramètre ***chaîne***. Tapez ce petit programme à l'écran et exécutez-le :

```
1 INPUT "Quel est votre nom ? ", Nom$
2 INPUT "Quel est votre âge ? ", Age
3 PRINT
4 PRINT "Vous vous appelez "; Nom$; " et vous êtes âgé de"; Age;
"ans."
```

A la ligne 1, **INPUT** demande à l'utilisateur d'entrer une chaîne de caractères. A la ligne 2, c'est un nombre qui est demandé. Le type de donnée que cette instruction demande est toujours le même que celui de la variable. Ainsi, quand la ligne 2 est exécutée, si l'utilisateur entre autre chose qu'un nombre, ou si le nombre qu'il entre est trop grand, QBASIC lui demande de recommencer.

E) Calculs (opérateurs arithmétiques)

Les calculs sont faits à l'aide des opérateurs arithmétiques. Les plus simples sont **+**, **–**, ***** (multiplication), et **/** (division). Des parenthèses peuvent être utilisées pour modifier l'ordre de priorité des opérateurs.

Un calcul sert généralement à l'affectation d'une variable :

```
1 Nombre.1 = 2 * 7.2
2 Nombre.2 = 16 / 7
3 Nombre.3 = (4 - 2.42) * (3.1 + 1.24)
```

Dans ces trois instructions, les calculs n'ont été faits qu'à partir de données constantes. Mais on peut aussi effectuer des calculs sur des données variables :

```
4 Nombre.4 = Nombre.1 + Nombre.2
5 Nombre.4 = Nombre.4 + 2
```

L'instruction de la ligne 4 affecte à **Nombre.4** la somme des valeurs de **Nombre.1** et de **Nombre.2**, et l'instruction de la ligne 5 ajoute 2 à la valeur de **Nombre.4** : il est possible de mêler données constantes et variables dans un calcul.

Bien que la plupart du temps, on affecte le résultat d'un calcul à une variable, ce n'est pas la seule façon d'utiliser un calcul en QBASIC. On peut aussi en insérer un dans toute instruction qui doit être complétée par une ou plusieurs données de type nombre. Par exemple, on a vu que **PRINT** permet d'afficher des données à l'écran, de type texte ou de type nombre. On peut donc afficher à l'écran le résultat d'un calcul inséré dans une instruction **PRINT** :

```
6 PRINT Nombre.3; "+"; Nombre.4; "="; Nombre.3 + Nombre.4
```

QBASIC permet d'effectuer une division entière (euclidienne), c'est-à-dire de décomposer un nombre entier en un produit de deux nombres entiers et un reste entier. Il faut pour cela utiliser les opérateurs \ et **MOD** (modulo). \ calcule le quotient entier de la division, et **MOD** calcule le reste entier. Voici comment ils s'utilisent :

```
PRINT "10 divisé par 3 =", 10 \ 3; "reste"; 10 MOD 3
```

Notez que même si **MOD** s'écrit comme un mot-clé, ce n'en est pas un : c'est un opérateur. Il existe d'autres opérateurs comme celui-ci.

L'opérateur permettant d'élever un nombre à une certaine puissance est ^. Le nombre est placé à gauche de l'opérateur, et la puissance à laquelle il doit être élevé est écrite à droite. On peut élever un nombre au carré ainsi :

```
PRINT "5 au carré est égal à"; 5 ^ 2
```

F) Fonctions

Nous avons déjà parlé des fonctions dans la section A. Il s'agit de mots-clés particuliers puisque ce sont les seuls qui ne doivent pas occuper une position prédéfinie dans les instructions. En fait, leur utilisation ressemble assez à celle des variables. Prenons un exemple :

```
INPUT "Veuillez entrer un nombre au hasard : ", Nombre1
Double = 2 * Nombre1
Nombre2 = 2 * RND
```

Les deux premières instructions sont faciles à comprendre. Le programme demande à l'utilisateur d'entrer un nombre au hasard qui est stocké dans la variable **Nombre1**. Ensuite, ce nombre est multiplié par deux et stocké dans **Double**. La troisième ligne fait la même chose : elle multiplie par deux un nombre pris au hasard, mais cette fois-ci ce n'est pas l'utilisateur qui a choisi ce nombre, c'est l'ordinateur lui-même.

La fonction **RND** est utilisée de la même manière que la variable **Nombre1** à la ligne précédente : sa valeur est lue et un calcul est effectué avec. Ce qui différencie les variables des fonctions, c'est la façon dont une valeur leur est affectée. Dans le cas des variables, c'est le programmeur qui fixe la valeur, à l'aide de l'opérateur d'affectation **=**, ou bien l'utilisateur, quand une instruction **INPUT** le lui demande. En revanche, la valeur d'une fonction est affectée par l'ordinateur lui-même.

Quand une variable est utilisée dans une instruction, QBASIC se contente de lire sa valeur dans la mémoire de l'ordinateur. Mais quand il s'agit d'une fonction, il effectue d'abord une opération, qui lui permet d'obtenir une valeur. On dit que cette valeur est « renvoyée » ou « retournée » par la fonction. Elle peut alors être utilisée comme la valeur d'une variable.

L'opération qui détermine la valeur renvoyée dépend de la fonction. Dans le cas de **RND**, il s'agit simplement de choisir un nombre au hasard. Notez, toutefois, que cette fonction ne retourne pas n'importe quels nombres. Son utilisation sera expliquée au dernier chapitre.

Il existe d'autres types de fonctions, qui effectuent des calculs pour obtenir la valeur à renvoyer. Ces calculs se font à partir d'un nombre qui doit être précisé par le programmeur. Le nombre est placé entre parenthèses, juste après le mot-clé.

La fonction **FIX** sert à tronquer des nombres décimaux. Elle s'utilise ainsi :

```
PRINT FIX(12.53)
```

Le nombre 12 s'affiche à l'écran : **FIX** a renvoyé la partie entière du nombre décimal 12,53.

G) Prise de décision (constructions IF…THEN)

Une des notions les plus importantes en programmation est la « prise de décision ». En effet, un programme ne doit pas être limité à une suite d'instructions qui reste invariable. Au contraire, dans toute application, l'ordinateur est très souvent amené à choisir entre plusieurs séries d'instructions à exécuter, en fonction de la situation.

Prenons un exemple simple et courant : au cours de l'exécution d'un programme, l'ordinateur demande à l'utilisateur s'il souhaite quitter l'application, ou s'il souhaite au contraire continuer à l'utiliser. Si l'utilisateur répond oui, QBASIC arrête l'exécution, sinon le programme poursuit son déroulement.

Les deux mots-clés qui permettent la prise de décision sont **IF** et **THEN**. Ils sont parfois associés à **ELSE**. Ces trois mots-clés signifient respectivement en anglais « si », « alors » et « sinon ». Les instructions qui les utilisent en programmation sont construites à peu près de la même façon que les phrases qui contiennent ces mots dans la langue parlée, ce qui simplifie leur utilisation.

Il existe plusieurs variantes de constructions **IF…THEN**. Voici la plus simple :

IF ***condition*** **THEN** ***instruction***

Cette construction associe deux instructions dans une même ligne : l'instruction **IF…THEN**, complétée par le paramètre ***condition***, et une seconde instruction, qui est placée après **THEN**.

condition est une expression logique, qui peut être vraie ou fausse. Si celle-ci est vraie, QBASIC exécute ***instruction***. Si elle est fausse, ***instruction*** n'est pas exécutée. Vous comprendrez mieux à travers un exemple. Regardez cette phrase :

« S'il pleut, alors je prends mon parapluie. »

Les connecteurs logiques sont « si » (**IF**) et « alors » (**THEN**). La condition est « il pleut ». Si, effectivement, il pleut, cette condition est vraie. Si, au contraire, il fait beau, elle est fausse. Si la condition est vraie, l'action (l'instruction) « je prends mon parapluie » est effectuée (exécutée).

Voici maintenant un exemple concret :

```
1 INPUT "Entrez le prix : ", Prix
2 INPUT "Voulez-vous prendre en compte la TVA (o/n) ? ", Reponse$
3 IF Reponse$ = "o" THEN Prix = Prix * 1.196
4 PRINT "Le prix est :"; Prix
```

La ligne 2 demande à l'utilisateur d'entrer une chaîne de caractères, qu'elle stocke dans la variable **Reponse$**. En fait, le programme pose une question à l'utilisateur, à laquelle il doit répondre par oui (en tapant au clavier le caractère « *o* ») ou par non (en tapant le caractère « *n* »). Puis le programme doit agir en fonction de sa réponse. Il teste la condition « l'utilisateur a répondu oui ». Si celle-ci est vraie, il augmente la valeur de la variable **Prix** de 19,6 %. Si la condition est fausse, c'est-à-dire si l'utilisateur entre au clavier autre chose que « *o* », QBASIC n'exécute pas l'instruction placée après **THEN**. Ainsi, l'utilisateur n'est pas obligé de taper « *n* » pour répondre non.

Comme on l'a dit, les conditions sont des expressions logiques. Dans la plupart des cas, ce sont des égalités, et elles sont donc formées avec l'opérateur **=**. Les termes comparés peuvent être des variables ou des données constantes. On place généralement devant le signe **=** un nom de variable. Cela permet de vérifier quelle est sa valeur. En plaçant une donnée constante après l'opérateur **=**, on teste si la variable est égale à cette donnée. En plaçant le nom d'une seconde variable, on vérifie si les deux variables sont égales.

Les conditions peuvent aussi être des inégalités. Leur organisation est la même, sauf que les opérateurs utilisés sont alors **<>**, qui signifie « différent de » (c'est l'équivalent du égal barré en mathématiques), ou bien **<** et **>** si les données sont des nombres. On peut aussi utiliser **<=** (ou **=<**) et **>=** (ou **=>**), qui signifient « inférieur ou égal à » et « supérieur ou égal à ».

Voici quelques exemples de conditions (ceci n'est pas un programme exécutable) :

```
IF Quantite = 5 THEN …
IF Nom$ = "Jean" THEN …
IF Nombre1 = Nombre2 THEN …
IF Age > 16 THEN …
IF Reponse$ <> "o" THEN …
IF Prix <= 500 THEN …
```

Voilà comment on pourrait traduire ces lignes :
SI **Quantite** est égale à 5 ALORS …
SI **Nom$** est « *Jean* » ALORS …
SI **Nombre1** et **Nombre2** sont égaux ALORS …
SI **Age** est supérieur à 16 ALORS …
SI **Reponse$** est différente de « *o* » ALORS …
SI **Prix** est inférieur ou égal à 500 ALORS …

Voyons maintenant la seconde variante. C'est la même construction, sauf que l'on ajoute le mot-clé **ELSE**, qui est suivi d'une autre instruction :

IF *condition* THEN *instruction1* ELSE *instruction2*

instruction1 est exécutée si ***condition*** est vraie. ***instruction2*** est exécutée si ***condition*** est fausse.

Modifions l'exemple précédent, en conservant toutefois les lignes 1 et 4 :

```
1 …
2 INPUT "Voulez-vous effectuer une augmentation (a) ou une réduction
(r) de ce prix ? ", Reponse$
3 IF Reponse$ = "a" THEN Prix = Prix * 1.20 ELSE Prix = Prix * 0.80
4 …
```

La ligne 2 pose une question à laquelle l'utilisateur doit répondre par « a » ou par « r ». La ligne 3 vérifie si l'utilisateur a répondu « a ». Si cette condition est vraie, l'instruction qui suit **THEN** est exécutée, c'est-à-dire que la valeur de la variable **Prix** est augmentée de 20%, sinon l'instruction qui suit **ELSE** est exécutée, c'est-à-dire que le prix est diminué de 20%.

La première variante que l'on a vue ne permet d'exécuter qu'une seule instruction lorsque la condition est vraie. Cependant, il est possible qu'il faille exécuter plusieurs instructions à une même condition. Il existe donc une construction dérivée qui s'étend sur plusieurs lignes :

IF *condition* THEN
instruction1
instruction2
…
END IF

instruction1, ***instruction2***, ainsi que toutes les instructions qui suivent, jusqu'à **END IF**, forment ce que l'on appelle un « bloc d'instructions ». Si ***condition*** est vraie, le bloc entier est exécuté, sinon, QBASIC passe directement aux instructions situées après **END IF**.

Voici un exemple de construction **IF…THEN** sur plusieurs lignes :

```
1 …
2 INPUT "Voulez-vous effectuer une réduction (o/n) ? ", Reponse$
3 IF Reponse$ = "o" THEN
4 INPUT "Entrez le pourcentage de réduction : ", Prcent
5 Prix = Prix * (1 - Prcent / 100)
6 END IF
7 …
```

Les lignes 4 et 5 ne sont exécutées que si la condition testée à la ligne 3 est vraie. L'instruction **END IF** à la ligne 6 marque la fin de la construction **IF…THEN**. La ligne 7 est donc toujours exécutée, que la condition soit vraie ou fausse, puisqu'elle est située après **END IF**.

De la même manière, la seconde variante (avec **ELSE**) peut être étendue sur plusieurs lignes. Elle comprend alors deux blocs d'instructions : le premier est situé entre les instructions **IF…THEN** et **ELSE**, et le second entre les instructions **ELSE** et **END IF**.

Complétons l'exemple :

```
1 …
2 INPUT "Voulez-vous effectuer une augmentation (a) ou une réduction
(r) de ce prix ? ", Reponse$
3 IF Reponse$ = "a" THEN
4 INPUT "Entrez le pourcentage d'augmentation : ", Prcent
5 Prix = Prix * (1 + Prcent / 100)
6 ELSE
7 INPUT "Entrez le pourcentage de réduction : ", Prcent
8 Prix = Prix * (1 - Prcent / 100)
9 END IF
10 …
```

Les lignes 4 et 5 sont exécutées si la condition testée à la ligne 3 est vraie, alors que les lignes 7 et 8 sont exécutées si elle est fausse.

Il est possible de tester plusieurs conditions dans une même construction **IF…THEN**. Cela se fait dans le cas où la première est fausse. Il faut rajouter une instruction **ELSEIF**. Son organisation est la même que celle de l'instruction **IF…THEN**. Comme **ELSE**, elle doit être placée après le bloc d'instructions exécuté si la première condition (celle de l'instruction **IF…THEN**) est vraie. Modifions la ligne 6 de l'exemple que l'on vient de voir comme ceci :

```
6 ELSEIF Reponse$ = "r" THEN
```

Les lignes 7 et 8 ne sont exécutées que si la première condition (celle testée à la ligne 3) est fausse, et si la seconde (celle testée à la ligne 6) est vraie. Si les deux sont fausses, aucune instruction n'est exécutée.

Notez qu'il est possible d'ajouter autant d'instructions **ELSEIF** que nécessaire dans une même construction **IF…THEN**. On peut aussi les associer à une instruction **ELSE**, qui sera prise en compte si aucune des conditions n'est vraie.

Une construction **IF…THEN** permet de faire ce que l'on appelle un « branchement conditionnel ». En effet, elle permet, à une certaine condition, de « brancher » (rediriger) l'exécution sur un bloc d'instructions, c'est-à-dire de demander à QBASIC d'exécuter ces instructions. Dans le cas d'une construction **IF…THEN** sans **ELSE**, si la ou les conditions sont fausses, l'exécution est branchée sur les lignes qui suivent l'instruction **END IF**. Dans le cas d'une construction avec **ELSE**, si la ou les conditions sont fausses, l'exécution est branchée sur le bloc d'instructions qui suit **ELSE**.

H) Répétition d'instructions (instruction GOTO)

Très souvent, vous serez amené à répéter plusieurs fois une même série d'instructions. Prenons l'exemple d'une application qui sert à effectuer un calcul particulier. Lorsqu'elle est lancée, elle demande à l'utilisateur de taper au clavier les nombres à partir desquels doit être fait le calcul. Une fois que le programme a calculé et affiché le résultat, il demande à l'utilisateur s'il veut effectuer à nouveau le calcul, avec des nombres différents. Si l'utilisateur répond oui, les instructions du calcul doivent à nouveau être exécutées.

Pour répéter une série d'instructions, on utilise la commande **GOTO**. Elle permet de brancher l'exécution sur la série d'instructions en question. On appelle cela un « branchement inconditionnel ». Ici aussi, on retrouve l'anglais : « go to » signifie « va à », qui s'interprète dans ce cas par « branche sur ».

La première méthode consiste à compléter l'instruction **GOTO** par le numéro de la première instruction de la série. Lorsque QBASIC rencontre la commande **GOTO**, il exécute directement l'instruction dont le numéro lui est indiqué. Il continue ensuite à exécuter toutes les instructions qui suivent celle-ci.

Plutôt que de se servir des numéros d'instructions, on peut utiliser des « étiquettes ». Une étiquette permet de repérer un emplacement précis dans le code. Chaque étiquette a un nom, qui est de la même forme qu'un nom de variable (constitué de lettres non accentuées, de points et de chiffres). Le nom de l'étiquette doit être écrit sur la ligne qui précède la première instruction de la série à répéter, suivi de deux points. C'est ce que l'on appelle la « définition de l'étiquette ». Pour brancher l'exécution sur la série d'instructions, il faut compléter l'instruction **GOTO** par le nom de l'étiquette.

Prenons l'exemple dont on a parlé, avec un calcul simple, l'addition :

```
Debut:
1 PRINT "Veuillez entrer les deux nombres à additionner."
2 INPUT "Premier nombre : ", Nombre1
3 INPUT "Second nombre : ", Nombre2
4 PRINT "Somme :"; Nombre1 + Nombre2
5 INPUT "Recommencer (o/n) ? ", Reponse$
6 IF Reponse$ = "o" THEN
7 PRINT
8 GOTO Debut
9 END IF
```

La toute première ligne, non numérotée, contient la définition de l'étiquette **Debut**. Au lancement du programme, les instructions 1 à 5 sont exécutées normalement. La ligne 6 est une instruction **IF…THEN**, qui branche l'exécution sur la ligne 7 si l'utilisateur répond qu'il veut effectuer une nouvelle addition. Si l'utilisateur répond non, le programme prend fin.

Lorsque la ligne 8 est atteinte (cela se produit uniquement si l'utilisateur a répondu oui), l'exécution est branchée sur la ligne qui suit la définition de l'étiquette **Debut**, c'est-à-dire la ligne 1. Puis QBASIC exécute à nouveau les lignes 2 à 6. Tant que l'utilisateur répond oui, les lignes 1 à 8 sont répétées.

Remarquez que l'instruction **GOTO** de la ligne 8 aurait pu être complétée par le numéro 1, à la place du nom de l'étiquette. Le résultat aurait été le même. On aurait pu aussi la compléter par le numéro 2, afin que la ligne 1 ne soit pas répétée.

Veillez toujours à ce qu'une instruction **GOTO** ne puisse être atteinte que par un branchement conditionnel (qu'elle soit comprise dans le bloc d'instructions d'une construction **IF…THEN**), ou bien qu'elle puisse être sautée par une autre instruction **GOTO**. En effet, dans l'exemple précédent, si la ligne 8 n'était pas comprise dans le bloc d'instructions de la construction **IF…THEN**, elle serait répétée sans cesse, et le programme ne s'arrêterait jamais.

On aurait aussi pu vérifier si l'utilisateur a répondu non, et dans ce cas effectuer un branchement sur la fin du code à l'aide d'une instruction **GOTO**. Ainsi, l'instruction **GOTO** qui branche l'exécution sur la ligne 1 serait sautée. Voici comment il faudrait compléter l'exemple :

```
6 IF Reponse$ = "n" THEN GOTO Fin
7 PRINT
8 GOTO Debut

Fin:
```

Notez que les définitions d'étiquettes ne constituent pas des instructions. Elles ne sont prises en compte que dans le but de servir à une instruction **GOTO**. Ainsi, l'insertion d'une définition d'étiquette au milieu d'une série d'instructions ne change en rien l'exécution de ces instructions.

Conclusion

On a vu que tout programme est composé d'instructions, pouvant être numérotées afin d'être mieux repérées. La plupart des instructions dépendent d'un mot-clé. Elles peuvent être complétées par des données constantes, des variables, des calculs, des fonctions, des expressions logiques ou d'autres mots-clés.

Vous avez appris comment communiquer avec l'utilisateur de deux façons : avec **`PRINT`** (pour afficher des données à l'écran), et avec **`INPUT`** (pour récupérer les données qu'il entre au clavier). On a aussi vu comment effectuer une prise de décision (branchement conditionnel), et comment répéter une série d'instructions (branchement inconditionnel).

Vous connaissez donc à présent les principaux éléments constitutifs du code BASIC, qui vont vous servir à écrire quelques programmes.

Chapitre III :
Premier programme

Pour vous donner une idée de la manière dont les éléments étudiés dans le chapitre précédent doivent être organisés dans un programme, nous allons étudier un exemple d'application simple et classique : un programme de calcul (calculette). Commencez d'abord par tester le programme. Il est enregistré dans le fichier « *CALCUL.BAS* ». Une fois que l'avez exécuté, regardez rapidement le code du programme copié ci-dessous. Lisez ensuite l'analyse, et examinez à nouveau le code, avec plus d'attention cette fois-ci. Une fois que vous aurez bien tout compris, écrivez quelques petits programmes vous-même pour assimiler la méthode de structuration des programmes.

A) Code du programme

```
AffMenu:
REM ***** Affichage du menu *****
10 PRINT
20 PRINT
30 PRINT "     ** Programme de calcul **"
40 PRINT
50 PRINT
60 PRINT "Voici les opérations possibles :"
70 PRINT " 1: Addition"
80 PRINT " 2: Soustraction"
90 PRINT " 3: Multiplication"
100 PRINT " 4: Division"
110 PRINT " 5: Quitter le programme"
120 PRINT

ChoixOp:
REM ***** Demande à l'utilisateur de choisir l'opération *****
130 INPUT "Entrez le numéro de l'opération à effectuer : ", NumOp
140 PRINT
150 IF NumOp = 1 THEN
160 GOTO Addition
170 ELSEIF NumOp = 2 THEN : GOTO Soustr
180 ELSEIF NumOp = 3 THEN : GOTO Multip
190 ELSEIF NumOp = 4 THEN : GOTO Divis
200 ELSEIF NumOp = 5 THEN : GOTO Fin
210 ELSE : GOTO ChoixOp
220 END IF

Addition:
REM ***** Opération d'addition *****
230 INPUT "Entrez le premier nombre : ", Nbre1
240 INPUT "Entrez le second nombre : ", Nbre2
250 PRINT
```

```
260 PRINT "Le résultat de l'addition est :"; Nbre1 + Nbre2
270 GOTO AffMenu
Soustr:
REM ***** Opération de soustraction *****
280 INPUT "Entrez le premier nombre : ", Nbre1
290 INPUT "Entrez le second nombre : ", Nbre2
300 PRINT
310 PRINT "Le résultat de la soustraction est :"; Nbre1 - Nbre2
320 GOTO AffMenu

Multip:
REM ***** Opération de multiplication *****
330 INPUT "Entrez le premier nombre : ", Nbre1
340 INPUT "Entrez le second nombre : ", Nbre2
350 PRINT
360 PRINT "Le résultat de la multiplication est :"; Nbre1 * Nbre2
370 GOTO AffMenu

Divis:
REM ***** Opération de division *****
380 INPUT "Entrez le premier nombre : ", Nbre1
390 INPUT "Entrez le second nombre : ", Nbre2
400 PRINT
410 PRINT "Le résultat de la division est :"; Nbre1 / Nbre2
420 GOTO AffMenu

Fin:
REM ***** Quitte le programme *****
```

B) Analyse de l'organisation du code

Comme vous pouvez le constater, le code est divisé en plusieurs parties, c'est-à-dire en plusieurs séries d'instructions (blocs d'instructions), séparées entre elles par des lignes vides. En effet, QBASIC permet au programmeur de laisser autant de lignes vides qu'il le souhaite dans le code. Celles-ci ne sont pas prises en compte à l'exécution. Elles servent simplement à aérer le code, pour le rendre plus lisible. Il est conseillé dans tout programme d'organiser de cette manière les instructions en plusieurs blocs. Elles ne doivent pas être regroupées au hasard : il s'agit de rassembler des instructions qui sont liées, et qui, exécutées ensemble, effectuent une seule opération. Autrement dit, chaque bloc d'instructions doit correspondre à une étape précise du programme. Par exemple, le premier bloc du programme de calcul correspond à l'étape d'affichage du menu à l'écran.

Ces séries d'instructions sont toutes précédées d'une définition d'étiquette. Le nom de chaque étiquette est choisi en fonction de ce que doit faire le bloc d'instructions auquel elle correspond. Ainsi, chacun des blocs peut être exécuté à tout moment grâce à une instruction **GOTO**. Plutôt que d'utiliser des étiquettes, on pourrait aussi indiquer le numéro de la première ligne de chaque série. Cependant, ce serait moins explicite pour le programmeur. En effet, lorsque celui-ci veut relire le code, il comprend tout de suite à quelle étape renvoie une instruction **GOTO** si elle est complétée par un nom d'étiquette, alors qu'un numéro de ligne est moins « parlant ».

On a vu précédemment que **GOTO** permet de répéter des instructions plusieurs fois. Mais ce n'est pas son unique fonction. Il sert aussi à créer des articulations dans un programme. Dans notre exemple, chaque instruction **GOTO** branche l'exécution vers une étape du programme. L'ordre dans lequel sont exécutées ces instructions correspond à l'ordre dans lequel sont exécutées les différentes étapes du programmes. Ainsi, les instructions **GOTO** commandent le déroulement du programme.

Vous avez remarqué que les lignes sont numérotées de dix en dix. C'est une précaution à prendre lorsque l'on écrit un programme. En effet, si vous avez numéroté les lignes de votre programme par des numéros consécutifs, et que vous souhaitez insérer une instruction entre deux lignes d'un bloc d'instructions, vous serez obligés de réécrire tous les numéros des lignes suivantes. Ce serait une grosse perte de temps. En numérotant les lignes de dix en dix (voire de vingt en vingt, ou plus), vous pouvez ajouter et numéroter plusieurs lignes entre chaque instruction, sans avoir besoin de réécrire tous les numéros.

Cependant, il est encore plus simple de ne pas du tout numéroter les lignes, puisque ce n'est pas obligatoire. Vous avez vu que la numérotation possède des avantages (elle permet de repérer les instructions facilement, et elle peut servir pour **GOTO**), mais elle a aussi des inconvénients (le principal est la perte de temps). C'est à vous de décider si vous devez l'utiliser ou non.

Vous avez aussi certainement observé que chaque définition d'étiquette est suivie d'une ligne commencée par le mot-clé **REM**. **REM** indique à QBASIC qu'il ne doit pas tenir compte de ce qui suit sur la ligne, car c'est une remarque du programmeur. Les étoiles ne sont pas obligatoires, mais elles permettent de faire ressortir le commentaire parmi les lignes d'instructions. Il est aussi possible d'insérer une remarque dans une ligne d'instruction. Elle doit être placée à la fin de la ligne, et précédée du caractère **'** (apostrophe), au lieu du mot-clé **REM**.

Les commentaires permettent au développeur, lorsqu'il relit son code, de savoir facilement ce que fait le bloc d'instructions en question, sans avoir à le relire en entier. Ils sont très utiles dans les longs programmes, car le code est structuré en de nombreuses parties, et dont certaines sont parfois longues. Dans ce petit programme de calcul, les commentaires sont inutiles puisque le code est court. De plus, ils ne rajoutent aucune précision à celles que donnent les étiquettes. Ils ont simplement été ajoutés à titre d'exemple.

Enfin, le dernier point à ajouter concerne la construction **IF…THEN** des lignes 150 à 220. Au lieu que chacune des instructions correspondant à un **ELSEIF** soit placée sur la ligne suivante, elle est placée sur la même ligne que ce mot-clé, séparée de lui par le caractère **:**. En effet, ce caractère permet d'écrire deux instructions sur une même ligne. En revanche, l'instruction qui suit le **THEN** doit toujours être placée sur la ligne suivante si la construction s'étale sur plusieurs lignes.

Il est ainsi possible d'écrire autant d'instructions que vous le souhaitez sur une même ligne, en les séparant chacune par `:`. Cependant il est déconseillé d'employer trop souvent ce procédé, parce que cela risquerait de rendre le code moins lisible. Il est pratique d'utiliser le caractère `:` pour associer des instructions courtes et ayant un rapport direct. C'est le cas dans notre exemple, car chacun des blocs de la construction **`IF…THEN`** n'est constitué que d'une seule instruction.

Tout ce que nous venons de voir concernant l'organisation d'un programme est facultatif, ce ne sont pas des obligations de QBASIC. Cependant, il est vivement conseillé de structurer un programme de la manière que nous avons étudiée, car cela permet de le rendre beaucoup plus clair, et par conséquent plus compréhensible. Appliquer ces règles simples rend l'écriture du code un peu plus longue, mais constitue un gain de temps considérable lors de la relecture et du débogage. La programmation demande beaucoup de temps, il faut donc en gagner au maximum.

C) Analyse du déroulement du programme

Le programme est divisé en six blocs d'instructions, étiquetés dans l'ordre : **`AffMenu`**, **`ChoixOp`**, **`Addition`**, **`Soustr`**, **`Multip`** et **`Divis`**. Notez que les noms ont été abrégés pour ne pas être trop longs. L'étiquette **`Fin`** permet de quitter le programme. En effet, sa définition occupe la dernière ligne du code, et donc lorsqu'une instruction **`GOTO`** effectue un branchement vers elle, QBASIC stoppe l'exécution du programme.

Le bloc d'instructions étiqueté **`AffMenu`** n'est constitué que d'instructions **`PRINT`**. Il affiche à l'écran le titre du programme, le menu qui indique les opérations que l'utilisateur peut demander au programme d'effectuer, ainsi que les numéros qu'il doit entrer pour cela. Tous ces éléments constituent la partie visuelle de « l'interface utilisateur » du programme, c'est-à-dire son apparence (les sons font aussi partie de l'interface utilisateur, ce que nous verrons plus tard).

Le second bloc d'instructions, **`ChoixOp`**, demande à l'utilisateur de choisir l'opération qu'il veut effectuer, en entrant le numéro correspondant. Une construction **`IF…THEN`** permet d'effectuer un branchement conditionnel en fonction de sa réponse. Si l'utilisateur répond par un numéro compris entre 1 et 4, l'exécution est branchée sur une des quatre dernières séries d'instructions, correspondant aux quatre opérations que peut effectuer le programme. S'il répond par le numéro 5, l'exécution est branchée sur l'étiquette **`Fin`**, et le programme se termine. Si l'utilisateur répond autre chose, ce bloc d'instructions est à nouveau exécuté.

Les quatre derniers blocs d'instructions correspondent aux quatre opérations basiques : l'addition, la soustraction, la multiplication et la division. Ils demandent à l'utilisateur d'entrer les deux nombres pour lesquels le calcul doit être effectué, puis affichent le résultat. Enfin, ils branchent l'exécution sur le premier bloc d'instructions. Le code est ainsi répété en boucle, jusqu'à ce que l'utilisateur décide de quitter le programme.

Chapitre IV :
Les constructions

Nous allons maintenant aborder les diverses constructions du BASIC. Elles constitueront des outils essentiels dans l'élaboration de vos applications. Vous en connaissez déjà un exemple : les constructions **IF…THEN**, qui ont été vues au chapitre deux, et qui vont être approfondies ici.

A) Prises de décision multiples (imbriquées)

Dans la plupart des cas, les prises de décision sont faites à l'aide du mot-clé **IF**. Nous avons vu qu'il existe plusieurs constructions, chacune d'elle étant adaptée à un cas précis. Le programmeur peut ainsi employer celle qui convient à ce qu'il veut faire. Mais le choix d'une de ces constructions n'est pas toujours suffisant.

Parfois, plusieurs décisions doivent être prises à la suite, chacune dépendant de celle qui a été prise avant. Il faut alors « imbriquer » plusieurs constructions **IF…THEN**, c'est-à-dire en introduire une dans un bloc d'instructions appartenant à une autre. Voici un exemple d'imbrication :

```
…
PRINT " A: Augmenter la vitesse de jeu"
PRINT " D: Diminuer la vitesse de jeu"
INPUT "Entrez votre choix : ", Reponse$
IF Reponse$ = "A" THEN
    IF Vitesse < 10 THEN
        Vitesse = Vitesse + 1
    ELSE
        PRINT "La vitesse est maximale."
        PRINT "Vous ne pouvez pas l'augmenter."
    END IF
ELSEIF Reponse$ = "D" THEN
    IF Vitesse > 1 THEN
        Vitesse = Vitesse - 1
    ELSE
        PRINT "La vitesse est minimale."
        PRINT "Vous ne pouvez pas la diminuer."
    END IF
END IF
```

Les premières lignes de cet exemple affichent des options à l'écran et demandent à l'utilisateur d'entrer son choix. La quatrième ligne est le début d'une construction **IF…THEN**, agissant en fonction de la réponse de l'utilisateur. Elle contient une instruction **ELSEIF**, et se termine par **END IF**.

Le premier bloc d'instructions correspond à celui qui est exécuté si l'utilisateur a choisi d'augmenter la vitesse, et le second correspond à celui qui est exécuté si l'utilisateur a choisi de diminuer la vitesse. Dans chacun des deux cas, une nouvelle prise de décision doit être effectuée : il faut évaluer la vitesse actuelle, pour savoir si l'utilisateur peut la modifier ou non. On a donc imbriqué une autre construction **`IF…THEN`** dans chacun des deux blocs appartenant à la première.

Remarquez que les blocs d'instructions compris dans les constructions **`IF…THEN`** sont décalés de la gauche de l'écran. On dit qu'on les a « indentés ». Cela permet de rendre le code plus clair, et de mieux s'y retrouver. On décale d'un certain nombre d'espaces vers la droite (quatre généralement) chacun des blocs de la construction principale. Puis on décale d'un cran supplémentaire les blocs de chaque construction comprise dans un bloc de la construction principale, et ainsi de suite, de sorte qu'il y ait une gradation des constructions.

Dans cet exemple, l'indentation n'apporte pas un énorme avantage au niveau du repérage des constructions, mais vous verrez qu'elle est presque indispensable lorsque les blocs d'instructions sont plus longs, et qu'il y a plusieurs constructions imbriquées. Imaginez que dans la construction qui vérifie si la vitesse n'est pas trop grande, en soit imbriquée une autre, puis qu'une autre encore soit imbriquée dans cette dernière, et ainsi de suite. Si vous n'indentez pas les blocs, il sera difficile de savoir, lors de la relecture du code, à quelle construction ils appartiennent. Au contraire, si vous le faites, vous n'aurez qu'à remonter le curseur dans le code jusqu'au moment où vous atteindrez une ligne décalée d'un cran de moins. Vous pourrez être sûr que cette ligne est une des instructions de la construction dans laquelle est comprise le bloc.

Pour indenter une ligne, il est inutile de taper plusieurs fois sur la touche *[Espace]* : la touche *[Tab]* (symbolisée par deux flèches de sens opposés, et située à gauche du clavier), permet d'afficher plusieurs espaces à la fois. On appelle cela une « tabulation ». Par défaut, QBASIC règle la tabulation à huit espaces, mais vous pouvez changer ce paramètre dans la fenêtre *Affichage*, accessible à partir du menu *Options*.

Quand vous indentez un bloc, vous devez insérer une tabulation au début de la première ligne. En revanche, ce n'est pas nécessaire que vous le fassiez pour les lignes suivantes. En effet, lorsque vous appuyez sur la touche *[Entrée]*, QBASIC place automatiquement le curseur sur la nouvelle ligne au même niveau d'indentation que la ligne précédente, à condition, cependant, que vous n'ayez pas numéroté les lignes. Pour désindenter une ligne, c'est-à-dire pour supprimer un cran de décalage, il suffit de placer le curseur sur le premier caractère de cette ligne, et d'appuyer sur la touche *[Retour arrière]*.

B) Prises de décision en évaluant une expression (construction `SELECT CASE`)

Dans certains cas, plusieurs conditions peuvent être testées les unes à la suite des autres dans une même construction **`IF…THEN`**, jusqu'à ce qu'une d'entre elles soit vraie, de sorte qu'une seule décision soit prise. C'est possible grâce au mot-clé **`ELSEIF`**.

Cette construction peut être utilisée pour évaluer une variable, et agir en conséquence. Par exemple, dans le programme de calcul que nous avons étudié, une construction de ce type évalue la variable **`NumOp`**, dans laquelle est stocké le numéro de l'opération que l'utilisateur a choisi d'effectuer, et l'exécution est branchée sur le bloc d'instructions correspondant.

L'inconvénient de cette construction, c'est qu'il faut réécrire l'égalité avec la variable pour chaque valeur à tester. Pour éviter cela, QBASIC met à votre disposition une autre construction qui permet d'évaluer une expression. C'est la construction **SELECT CASE** :

```
SELECT CASE expression
CASE valeur1
     instructions1
CASE valeur2
     instructions2
…
[CASE ELSE
     instructions3]
END SELECT
```

expression peut être une variable ou bien un calcul sur une ou plusieurs variables. C'est la valeur de cette variable ou le résultat de ce calcul qui est évalué par la construction **SELECT CASE**.

valeur1, ***valeur2***, etc. sont les valeurs à tester. Si une de ces valeurs est identique à celle de ***expression***, le bloc d'instructions correspondant est exécuté (***instructions1*** pour ***valeur1***, ***instructions2*** pour ***valeur2***, etc.)

Lorsque aucune des valeurs testées ne correspond, c'est le bloc d'instructions qui suit le mot-clé **CASE ELSE** (***instructions3***) qui est exécuté, s'il y en a un.

Reprenons l'exemple du programme de calcul avec une construction **SELECT CASE** :

```
150 SELECT CASE NumOp
160 CASE 1: GOTO Addition
170 CASE 2: GOTO Soustr
180 CASE 3: GOTO Multip
190 CASE 4: GOTO Divis
200 CASE 5: GOTO Fin
210 CASE ELSE: GOTO ChoixOp
220 END SELECT
```

Là aussi, les instructions ont été placées sur la même ligne que les mots-clés correspondants, à l'aide du caractère **:**, car pour chaque valeur testée, une seule instruction est à exécuter. Dès qu'il y en a plus d'une, il vaut mieux les placer sur des lignes séparées, en prenant soin d'indenter chaque bloc d'instructions, de la même manière que ceux des constructions **IF…THEN**.

Dans ce cas, la construction **SELECT CASE** n'est pas plus pratique que la construction **IF…THEN**, mais elle le devient lorsque l'expression à évaluer est longue à écrire, en particulier si c'est un calcul.

C) Prises de décision complexes (opérateurs logiques)

Jusqu'ici, nous n'avons étudié que des branchements conditionnels effectués après le test de conditions simples, c'est-à-dire formées d'une seule égalité ou inégalité. Mais il est aussi possible de tester des conditions complexes, c'est-à-dire constituées de plusieurs égalités ou inégalités. On utilise alors des opérateurs logiques pour associer les conditions. Ces opérateurs sont **AND**, **OR**, et **XOR**.

Les plus utilisés sont **AND** et **OR**, qui signifient en anglais « et » et « ou ». On les appelle « et logique » et « ou logique ». Ils se placent entre deux conditions afin de les associer. Plusieurs conditions associées à l'aide de ces opérateurs forment une condition générale, qui est placée entre les mots-clés **IF** et **THEN**, comme une condition simple. Prenons le cas d'une condition générale constituée de deux sous-conditions, associées par le mot-clé **AND**. La condition générale n'est vraie que lorsque les deux sous-conditions sont vraies. Si c'est l'opérateur **OR** qui est utilisé, la condition générale est vraie lorsqu'une des deux sous-conditions est vraie, ou bien lorsque les deux sont vraies. Regardez cet exemple :

```
IF Age >= 12 AND Age < 18 THEN …
IF Age < 12 OR Age >= 18 THEN …
```

Ces deux lignes peuvent être traduites :
SI **Age** est supérieur ou égal à 12 ET qu'il est inférieur à 18 ALORS …
SI **Age** est inférieur à 12 OU qu'il est supérieur ou égal à 18 ALORS …

L'opérateur **XOR** s'utilise de la même manière que **OR**. Il est appelé « ou exclusif ». Une condition générale constituée de deux sous-conditions associées par **XOR** n'est vraie que si une seule des deux sous-conditions est vraie. Elle est donc fausse si les deux sous-conditions sont vraies à la fois. Dans la deuxième ligne de l'exemple précédent, **OR** peut donc être remplacé par **XOR** sans que la condition en soit changée, puisqu'il est impossible que **Age** soit à la fois inférieur à 12 et supérieur à 18. Vous verrez par la suite que ce cas est fréquent.

De manière générale, **XOR** est utilisé plus rarement que **OR**. On l'emploie uniquement lorsque l'on doit s'assurer qu'il n'y ait qu'une seule des deux sous-conditions qui soit vraie. En voici un exemple :

```
IF Prix1 > 100 XOR Prix2 > 100 THEN …
```

Dans cet exemple, on teste si **Prix1** et **Prix2** sont supérieurs à 100. Si les deux sont supérieurs à 100 ou si les deux sont inférieurs ou égaux à 100, l'instruction qui suit **THEN** n'est pas exécutée. Si un seul des deux est supérieur à 100, l'instruction est exécutée.

Il existe un autre opérateur logique utilisé dans les conditions : **NOT**. Celui-ci s'utilise de manière un peu différente. Il se place avant une condition (ce peut être une condition simple, complexe ou une sous-condition), et permet de tester, non pas si elle est vraie, mais si elle est fausse. En effet, le mot « not » est la marque de la négation dans la langue anglaise. Il est toujours possible de remplacer une condition contenant **NOT** par une autre équivalente. **NOT** est donc généralement employé par souci de commodité et de compréhension. Par exemple, une condition construite avec l'opérateur **=** et associée à **NOT** équivaut à une condition construite avec l'opérateur **<>** :

```
IF NOT Reponse$ = "O" THEN …
IF Reponse$ <> "O" THEN …
```

Quand on construit une condition générale avec plusieurs sous-conditions, il arrive souvent qu'on ait besoin d'en regrouper certaines. Cela se fait à l'aide de parenthèses. Lorsque QBASIC rencontre des parenthèses, il teste toutes les sous-conditions qui se trouvent entre elles, et en déduit si la condition qu'elles constituent est vraie ou fausse. Celle-ci est alors utilisée comme sous-condition de la condition générale. Les exemples suivants illustrent cette utilisation :

```
IF Vitesse >= 1 AND Vitesse <= 5 AND (Vitesse >= 3 OR Diff = 1) THEN
…
IF NOT (Prenom$ = "Jean" AND Nom$ = "Dupont") THEN …
```

Dans le premier exemple, **`Vitesse`** représente la vitesse d'un jeu, et **`Diff`** la difficulté sur laquelle il a été réglé. Le programme vient de demander à l'utilisateur à quelle vitesse il souhaite jouer, et vérifie si la valeur que ce dernier a entré est correcte. La condition générale n'est vraie que si les deux premières sous-conditions, ainsi que celle comprise entre parenthèses sont vraies. Les deux premières sous-conditions vérifient si la vitesse est bien comprise 1 et 5, qui sont ses valeurs limites. La troisième n'est vraie que si la vitesse est supérieure ou égale à 3, sauf si le niveau de difficulté est 1. En effet, c'est le niveau le plus facile, et c'est le seul qui autorise l'utilisateur à baisser la vitesse en-dessous de 3.

Dans le second exemple, l'instruction **`IF…THEN`** teste si les conditions comprises entre parenthèses sont fausses. La condition générale n'est donc vraie que si les deux sous-conditions sont fausses. Elle équivaut à celle-ci :

```
IF Prenom$ <> "Jean" OR Nom$ <> "Dupont" THEN …
```

D) Boucles comptées (construction FOR…NEXT)

Nous avons vu que l'on peut répéter des instructions à l'aide du mot-clé **`GOTO`**. Cependant, **`GOTO`** ne contrôle pas le nombre de fois que le bloc d'instructions est répété. Une « boucle » **`FOR…NEXT`** le permet.

La construction est la suivante :

```
FOR variable = début TO fin [STEP pas]
    instructions
NEXT
```

Lorsque le bloc ***`instructions`*** ne contient qu'une seule instruction, il est souvent pratique de placer les trois éléments de la construction **`FOR…NEXT`** sur une seule ligne, en les séparant par des **`:`**. Dans les autres cas, il convient de les écrire sur des lignes différentes et d'indenter le bloc ***`instructions`***.

Afin de vous expliquer le fonctionnement de cette construction, prenons un exemple simple, dans lequel nous ne prenons pas en compte le paramètre **`STEP`** ***`pas`***. Ces instructions sautent dix lignes à l'écran :

```
FOR I = 1 TO 10
    PRINT
NEXT
```

I (pour « indice ») est le nom que l'on utilise généralement dans les instructions **FOR…NEXT** pour ***variable***. Cependant, vous êtes totalement libres de choisir un autre nom, ce qui dans certains cas est plus judicieux. Notez que ce peut très bien être une variable déjà utilisée dans le programme.

Lorsque QBASIC lit la première instruction (celle qui contient **FOR**), il affecte la valeur 1 (***début***) à **I** (***variable***). Puis, il exécute la commande **PRINT** (***instructions***). Lorsqu'il rencontre **NEXT**, il ajoute 1 à la variable **I**, et vérifie si sa valeur est inférieure ou égale à 10 (***fin***). Si c'est le cas, il exécute à nouveau la commande **PRINT**, et ainsi de suite. Au bout de la dixième répétition, **I** atteint la valeur 11. Comme elle est supérieure à 10, QBASIC n'exécute plus l'instruction **PRINT**. L'exécution du programme se poursuit alors sur les instructions qui suivent la construction **FOR…NEXT**.

La manière la plus simple de répéter *n* fois un bloc d'instructions est de prendre 1 comme valeur pour ***début***, et *n* comme valeur pour ***fin***. Ainsi pour répéter 10 fois l'instruction **PRINT**, on a pris 1 et 10 comme valeurs pour ***début*** et ***fin***. On aurait pu cependant prendre comme valeurs 11 et 20, le résultat aurait été le même. On peut avoir intérêt à prendre une autre valeur que 1 pour ***début*** lorsque l'on a besoin d'utiliser la valeur de ***variable*** dans le bloc ***instructions***.

Modifions un peu l'exemple, de sorte qu'il affiche la valeur de **I** à l'écran, tout en écrivant les trois instructions sur une seule ligne :

```
FOR I = 1 TO 10: PRINT I : NEXT
```

QBASIC écrit les nombres 1 à 10 à l'écran. Dans ce cas, en remplaçant ***début*** et ***fin*** par 11 et 20, on n'obtient pas le même résultat. Notez qu'après l'exécution de la construction **FOR…NEXT**, la valeur de **I** est 11, et non 10, alors que 11 n'est pas affiché à l'écran. Retenez bien ceci, car ce peut être important quand on a besoin d'utiliser ***variable*** après la boucle **FOR…NEXT**.

Revenons maintenant au paramètre **STEP** ***pas***. Il permet de préciser ce que l'on appelle le « pas d'itération » de la boucle **FOR…NEXT** : c'est la valeur qui est ajoutée à ***variable*** à chaque fois que QBASIC rencontre **NEXT**. S'il n'est pas précisé grâce à **STEP**, QBASIC le règle automatiquement à 1, comme nous l'avons vu dans l'exemple explicatif.

Cette ligne commande à QBASIC d'afficher tous les multiples de 7 compris entre 0 et 100 :

```
FOR I = 0 TO 100 STEP 7: PRINT I : NEXT
```

La dernière valeur affichée à l'écran est 98, car 98 est le plus grand multiple de 7 inférieur à 100. Notez qu'après l'exécution de la boucle **FOR…NEXT**, **I** est égal à 105.

E) Boucles conditionnelles (constructions DO…LOOP)

Il existe un second type de boucles : les boucles conditionnelles. Elles permettent de répéter un bloc d'instructions tant qu'une condition est vraie, ou jusqu'à ce qu'elle soit vraie. Ce genre de boucle se construit avec les mots-clés **DO** et **LOOP**, qui signifient « faire » (« exécuter les instructions » dans ce contexte) et « faire une boucle », ainsi que **WHILE** ou **UNTIL**, qui signifient « tant que » et « jusqu'à ce que ». Deux dispositions sont possibles :

```
DO
     instructions
LOOP {UNTIL | WHILE} condition
```

ou bien :

```
DO {UNTIL | WHILE} condition
     instructions
LOOP
```

Les accolades qui encadrent **UNTIL** et **WHILE**, ainsi que la barre qui les sépare indiquent que l'on doit utiliser l'un ou l'autre des mots-clé, mais pas les deux. C'est la notation utilisée par convention dans l'aide de QBASIC.

La disposition doit être choisie en fonction de ce que l'on veut faire. Lorsque l'on fait précéder ***condition*** du mot-clé **WHILE**, le bloc ***instructions*** est répété tant que celle-ci est vraie. Si c'est le mot-clé **UNTIL** qui est employé, le bloc est répété jusqu'à ce que ***condition*** soit vraie, c'est-à-dire tant qu'elle est fausse.

Lorsque la condition est placée sur la première ligne, elle est testée avant que le bloc ***instructions*** soit exécuté. Ainsi, si elle n'a pas la valeur que l'on attend (vraie ou fausse), le bloc n'est pas du tout exécuté. Si la condition est placée sur la dernière ligne, le bloc est exécuté au moins une fois, même si ***condition*** n'a pas la valeur attendue.

Prenons un exemple de programme pour illustrer l'utilisation de cette construction :

```
DO
    PRINT
    INPUT "Entrez un nombre : ", Nombre
    INPUT "Entrez le pourcentage à calculer : ", Prc
    PRINT Prc; "pour cent de"; Nombre; "="; Nombre * (Prc / 100)
    PRINT
    INPUT "Recommencer (O/N) ? ", Reponse$
LOOP WHILE Reponse$ = "O" OR Reponse$ = "o"
```

Le bloc d'instructions est exécuté au moins une fois. Puis, la dernière ligne teste si l'utilisateur a répondu oui à la question posée par l'instruction **INPUT** précédente. Tant que c'est le cas, il exécute à nouveau les instructions. Vous remarquez que l'on peut utiliser aussi bien des conditions complexes que des conditions simples, de la même manière que dans une construction **IF…THEN**.

Cet exemple peut aussi être écrit ainsi :

```
DO
    …
LOOP UNTIL Reponse$ <> "O" AND Reponse$ <> "o"
```

Maintenant, écrivons un autre programme similaire, mais dans lequel une question est posée avant la boucle (on suppose qu'une valeur a déjà été affectée à la variable **Nombre** dans le programme) :

```
…
INPUT "Calculer un pourcentage du nombre (O/N) ? ", Reponse$
DO WHILE Reponse$ = "O" OR Reponse$ = "o"
    PRINT
    INPUT "Entrez le pourcentage à calculer : ", Prc
    PRINT Prc; "pour cent de"; Nombre; "="; Nombre * (Prc / 100)
    PRINT
    INPUT "Recommencer (O/N) ? ", Reponse$
LOOP
```

Et, de même que pour l'exemple précédent, écrivons d'une autre façon la première ligne de la construction **DO…LOOP** :

```
…
DO UNTIL Reponse$ <> "O" AND Reponse$ <> "o"
    …
LOOP
```

Notez qu'il est possible, comme pour les constructions **FOR…NEXT**, de n'écrire la construction **DO…LOOP** qu'en une seule ligne, en utilisant le caractère **:**. Ce n'est conseillé que lorsqu'il n'y a qu'une seule instruction à répéter.

F) Branchement sur une sous-routine (instructions GOSUB et RETURN)

Dans l'exemple de programme de calcul que nous avons étudié précédemment, le code était structuré en parties, sur lesquelles on pouvait brancher l'exécution à tout moment, à l'aide d'une instruction **GOTO**. Ainsi, ces séries d'instructions pouvaient être répétées autant de fois que l'on voulait.

Cependant, une fois qu'un bloc d'instructions branché par une instruction **GOTO** a été exécuté, c'est sur les instructions qui suivent ce bloc que se poursuit l'exécution, et non pas sur celles qui suivent l'instruction **GOTO**. Pourtant, il est très souvent nécessaire de pouvoir revenir au point où l'exécution a été branchée, après que la série d'instructions branchées ont été exécutées.

Par exemple, dans chacune des parties correspondant aux opérations (addition, soustraction, multiplication et division) du programme de calcul, les trois premières lignes sont identiques. Répéter plusieurs fois trois lignes identiques n'est pas très gênant, mais ce serait une perte de temps si c'étaient des dizaines de lignes, voire plus. Dans ce cas, il est plus pratique de ne les écrire qu'une seule fois, séparément des différents blocs d'instructions dans lesquels elles sont comprises, et d'effectuer à chaque fois un branchement sur elles.

Lorsque ces instructions ont été exécutées, il faut revenir là où a été fait le branchement. Mais s'il peut avoir été fait de plusieurs endroits différents, comment peut-on alors savoir où il faut revenir ? Une solution serait d'utiliser une variable, qui indiquerait cet endroit. S'il n'y a que deux endroits possibles dans le code, cela ne pose pas de problème. Mais s'il y en a une dizaine ou plus, il faudrait utiliser une construction **SELECT CASE** très longue, et le code en serait considérablement alourdi. On ne peut donc pas revenir à l'emplacement d'où a été effectué le branchement à l'aide d'une instruction **GOTO**, puisqu'elle nécessite que l'on sache quel est cet emplacement.

Afin de résoudre ce problème, QBASIC permet d'effectuer un branchement sur ce que l'on appelle une « sous-routine ». Une sous-routine est une suite d'instructions, constituant une opération, et destinée à pouvoir être branchée plusieurs fois, à des endroits différents et sans changer l'ordre d'exécution d'un programme.

Dans notre exemple, nous avons besoin de demander à l'utilisateur d'entrer deux nombres. Cette opération doit être répétée plusieurs fois, et dans des contextes différents : au début de chacune des quatre dernières parties du programme. De plus, une fois que cette opération est effectuée, il faut revenir dans la partie qui était en cours d'exécution avant le branchement. Nous avons donc intérêt à créer une sous-routine pour cette opération.

La création de sous-routines est très simple, et s'apparente beaucoup à la création des parties du programme de calcul que nous avons vu en exemple. Une sous-routine débute par la définition d'une étiquette, suivie des instructions qui constituent l'opération. Enfin, elle est terminée par le mot-clé **RETURN**, qui indique à QBASIC de retourner à l'endroit où le branchement vers la sous-routine a été fait. Une sous-routine est branchée non pas à l'aide du mot-clé **GOTO**, mais du mot-clé **GOSUB**. **GOSUB** est l'abréviation de « go (to) sub-routine », qui signifie « branche sur la sous-routine ». On dit qu'on « appelle » une sous-routine lorsqu'on branche l'exécution sur elle.

Voici ce que donnerait la sous-routine demandant à l'utilisateur d'entrer des nombres :

```
EntreNbre:
    INPUT "Entrez le premier nombre : ", Nbre1
    INPUT "Entrez le second nombre : ", Nbre2
    PRINT
RETURN
```

Puis l'appel de cette sous-routine dans chacune des parties se fait à l'aide de cette instruction :

```
GOSUB EntreNbre
```

Puisque le programme est structuré en parties branchées par des instructions **GOTO**, cette sous-routine peut être placée n'importe où dans le code. Cependant, si elle est placée après le dernier bloc d'instructions (**Divis**), elle doit précéder l'étiquette **Fin**, pour éviter qu'elle soit exécutée avant l'arrêt du programme. De la même façon, si la sous-routine est placée avant le premier bloc d'instructions (**AffMenu**), il faut la faire précéder d'une instruction **GOTO** qui effectue un branchement vers celui-ci, pour qu'elle ne soit pas exécutée au démarrage du programme.

Conclusion

Vous connaissez à présent tous les concepts fondamentaux de la programmation en QBASIC. Les notions que vous venez d'acquérir vous permettront de créer des programmes plus élaborés et plus complexes. Entraînez-vous donc à cela dès maintenant. Si vous n'avez pas d'idée de programme dans lequel vous pouvez mettre en application ces notions, vous en trouverez des exemples dans la suite du cours. Mais essayez tout de même de vous exercer maintenant, car si vous n'assimilez pas les notions tout de suite, vous risquez d'avoir à relire certaines sections de ce chapitre par la suite.

Chapitre V :
Interface visuelle d'un programme

Il reste un domaine important que nous n'avons pas encore approfondi : c'est celui de l'interface entre le programme et l'utilisateur. C'est un élément capital pour les jeux notamment. La plus grosse partie de l'interface utilisateur des programmes QBASIC est l'interface visuelle. Nous allons donc nous intéresser aux moyens de la perfectionner.

A) Présentation des différents modes d'affichage de l'écran

Il existe deux catégories de modes d'affichage de l'écran en DOS : le mode texte et le mode graphique.

1. Mode texte

Le mode texte est le mode utilisé par défaut en DOS. C'est celui sur lequel nous avons travaillé jusqu'à présent. Comme son nom l'indique, il ne permet d'afficher à l'écran que du texte, c'est-à-dire des caractères.

Nous avons déjà défini les caractères comme étant tout ce qui peut être entré au clavier (lettres, chiffres, symboles de ponctuation), ainsi que d'autres symboles spéciaux. Il n'existe en DOS que 256 caractères, dont certains diffèrent selon la langue pour laquelle est configuré l'ordinateur. Ils sont regroupés dans ce qu'on appelle la table ASCII (*American Standard Code for Information Interchange*). Elle associe à chaque caractère un numéro compris entre 0 et 255. Vous pouvez consulter cette table dans la fenêtre d'aide de QBASIC.

L'écran est structuré sous forme de tableau. Il est divisé en cases rectangulaires de taille identique, disposées en lignes et en colonnes. Chaque case peut être vide ou contenir un caractère. Ainsi, une instruction `PRINT` ne fait qu'afficher des caractères sur plusieurs cases de l'écran. Les cases sont repérées par leur position à l'écran, c'est-à-dire par le numéro de la ligne et de la colonne dans lesquelles elles se trouvent. Les lignes sont numérotées de haut en bas et les colonnes de gauche à droite, en partant de 1. La case située le plus en haut et le plus à gauche se trouve donc dans la ligne 1 et dans la colonne 1.

Il existe plusieurs modes textes, caractérisés par le nombre de lignes et de colonnes qui constituent l'écran. Le mode standard contient 80 colonnes et 25 lignes. On dit que c'est le mode texte de format 80x25.

2. Mode graphique

Le mode graphique est le seul utilisé par les systèmes d'exploitations récents, tels que Windows. L'écran n'est plus tout à fait un tableau. Les divisions ne sont plus des cases mais des points. On appelle ces points des « pixels ». Chaque pixel de l'écran est affiché avec une certaine couleur. Ainsi, c'est en affichant avec des couleurs différentes les pixels de l'écran que l'on fait apparaître des graphismes.

Les pixels sont repérés de la même manière que les cases dans le mode texte, cependant on ne parle en mode graphique ni de colonne ni de ligne, mais respectivement « d'abscisse » et « d'ordonnée » : ce sont leurs coordonnées. Le pixel situé le plus en haut et le plus à gauche a pour abscisse 0 et pour ordonnée 0.

Il existe différents modes graphiques, caractérisés par leur « résolution », c'est-à-dire le nombre de pixels qui constituent l'écran en largeur (abscisses) et en hauteur (ordonnées), et par le nombre de couleurs différentes qui peuvent être utilisées pour afficher les pixels. Par exemple, affiché avec la résolution 640x480 pixels, l'écran est divisé en 480 rangées horizontales, comprenant chacune 640 pixels.

On peut afficher des caractères à l'écran en mode graphique de la même manière qu'en mode texte. C'est possible puisqu'il suffit pour dessiner un caractère d'afficher une série de pixels avec une certaine couleur (gris en général), sur un fond d'une autre couleur (noir en général). L'écran en mode graphique est donc aussi divisé en cases rectangulaires pouvant contenir des caractères. La taille et le nombre de ces cases varie en fonction de la résolution de l'écran.

Bien que l'on puisse afficher du texte en mode graphique, il est préférable d'utiliser le mode texte lorsque l'on n'a pas besoin d'afficher de graphismes, car toutes les fonctions du mode texte ne sont pas disponibles en mode graphique.

3. Changement de mode d'affichage de l'écran

L'écran est toujours en mode texte au lancement d'un programme, puisque l'éditeur de QBASIC fonctionne en mode texte. Le format du mode est généralement 80x25, mais ce peut aussi être 80x43 ou 80x50.

Pour passer en mode graphique, il faut utiliser la commande **`SCREEN`**. Ce mot-clé doit être suivi du numéro spécifiant le mode d'affichage. Voici un tableau simplifié des différents modes graphiques :

Numéro	Résolution	Format texte	Taille des caractères	Nombre de couleurs
7	320x200	40x25	8x8	16
8	640x200	80x25	8x8	16
9	640x350	80x25 ou 80x43	8x14 ou 8x8	16
12	640x480	80x30 ou 80x60	8x16 ou 8x8	16
13	320x200	40x25	8x8	256

Le format texte indique le nombre de colonnes et de lignes dans lesquelles on peut afficher des caractères, et la taille des caractères correspond aux dimensions en pixels des cases. Par exemple, en mode 9, les cases mesurent 8 pixels de large et 14 pixels de haut, dans le premier cas.

Remarquez que deux modes proposent deux formats texte différents. Lorsque l'affichage est basculé sur l'un de ces modes à l'aide d'une instruction **SCREEN**, c'est le premier format qui est utilisé. On peut basculer sur l'autre format avec la commande **WIDTH**. Il suffit de faire suivre ce mot-clé du nombre de colonnes puis du nombre de lignes, séparés par une virgule. N'essayez pas d'utiliser une autre combinaison de nombre de colonnes et de lignes que celles qui sont indiquées dans le tableau, QBASIC générera une erreur.

Ces lignes permettent de basculer l'affichage en mode graphique 640x350 pixels pour 16 couleurs, avec le format texte 80x43 :

```
SCREEN 9
WIDTH 80, 43
```

Les deux modes généralement les plus utilisés sont les modes 12 et 13. En effet, le mode 12 est celui qui offre la plus haute résolution (640x480 pixels), et le mode 13 est celui qui offre le plus de couleurs (256). Un autre avantage que possède ce dernier est qu'il permet d'obtenir de meilleures performances que le mode 12 au niveau de la rapidité et de la fluidité de l'affichage.

Si vous utilisez QBASIC dans une fenêtre Windows, il se peut que certaines erreurs se produisent quand vous travaillez en mode graphique. Par exemple, l'écran peut rester en mode texte lorsque vous basculez l'affichage du mode texte à un certain mode graphique. Pour résoudre ce problème, basculez l'affichage sur un autre mode graphique, qui ne pose pas ce problème, avant de le basculer sur celui avec lequel vous souhaitez travailler. Une autre erreur assez fréquente est que l'éditeur de QBASIC ne fonctionne plus normalement après l'exécution d'un programme travaillant en mode graphique. Pour éviter cela, n'oubliez pas de systématiquement repasser en mode texte à la fin de chaque programme utilisant le mode graphique. Il existe aussi une autre solution, qui est de travailler en plein écran. C'est ce qu'il est préférable de faire si vous n'avez pas besoin de vous servir d'un autre programme en même temps que QBASIC.

Le numéro 0 permet de basculer l'affichage en mode texte lorsqu'il est en mode graphique. Le format du mode texte dépend alors de la résolution du mode graphique qui était utilisé avant le basculement : ce peut être soit 80x25, soit 40x25. Pour changer le format du mode texte sans avoir à basculer l'affichage en mode graphique, vous pouvez utiliser la commande **WIDTH**, de la même manière que pour changer le format texte en mode graphique. Les combinaisons possibles sont 80x25, 80x43, 80x50, 40x25, 40x43 et 40x50.

B) Instructions du mode texte

1. Affichage de caractères ASCII (fonction CHR$)

La principale instruction utilisée en mode texte est la première dont nous avons parlé, c'est-à-dire l'instruction **PRINT**. Nous avons vu comment l'utiliser pour afficher du texte saisi au clavier dans la fenêtre de code. Cependant, certains des caractères contenus dans la table ASCII ne peuvent pas être saisis au clavier. Pour les afficher à l'écran, il faut utiliser la fonction **CHR$** dans une instruction **PRINT**. Elle doit être suivie de parenthèses comprenant le code du caractère ASCII à afficher.

Par exemple, cette instruction affiche à l'écran une note de musique :

```
PRINT CHR$(14)
```

Ces lignes affichent un petit cadre à l'écran :

```
PRINT CHR$(201); CHR$(205); CHR$(205); CHR$(187)
PRINT CHR$(186); CHR$(32); CHR$(32); CHR$(186)
PRINT CHR$(186); CHR$(32); CHR$(32); CHR$(186)
PRINT CHR$(200); CHR$(205); CHR$(205); CHR$(188)
```

Vous pouvez consulter les codes ASCII attribués aux caractères dans l'aide de QBASIC, en cliquant sur le lien *Codes de caractères ASCII* dans la *Table des matières*.

2. Déplacement du curseur (instruction LOCATE)

Vous avez dû remarquer que la commande **PRINT** n'affiche pas les caractères n'importe où à l'écran. Elle commence toujours à afficher les caractères en début de ligne, c'est-à-dire dans la colonne 1, et à chaque nouvelle instruction **PRINT** l'affichage se fait sur une nouvelle ligne.

La position de l'écran où doit commencer l'affichage, c'est-à-dire la case à partir de laquelle les caractères doivent être affichés, est repérée par ce que l'on appelle le « curseur ». Le curseur est symbolisé à l'écran par une petite barre qui clignote. Les instructions **PRINT** et **INPUT** commencent l'affichage à la position du curseur. Le curseur est déplacé d'une colonne vers la droite à chaque caractère affiché, et il est positionné au début de la ligne suivante dès que la fin de la ligne courante est atteinte, ou lorsque l'interprétation de l'instruction **PRINT** ou **INPUT** est terminée. Quand le curseur atteint la fin de la dernière ligne de l'écran, il est repositionné au début de cette ligne, et toutes les lignes de l'écran sont décalées d'un cran vers le haut.

La position du curseur, c'est-à-dire le numéro de la ligne et de la colonne dans lesquelles il se trouve, est stockée en mémoire dans des variables. Il n'est pas possible d'y accéder directement pour modifier leurs valeurs, mais on le peut le faire à l'aide de la commande **LOCATE**. Elle doit être utilisée de cette manière :

LOCATE ***ligne***, ***colonne*** [, ***curseur*** [, ***lignesup***, ***ligneinf***]]

ligne et ***colonne*** spécifient la position du curseur. Elles doivent être comprises dans les limites du format de l'écran.

curseur peut prendre deux valeurs différentes : 0 ou 1. 0 indique que le curseur est invisible à l'écran, et 1 indique qu'il est visible. Ce paramètre est facultatif. Par défaut, le curseur est invisible. Notez cependant qu'il est toujours visible lorsqu'une instruction **INPUT** est interprétée.

lignesup et ***ligneinf*** permettent de régler la taille du curseur clignotant à l'écran. ***lignesup*** précise le numéro de la ligne de pixels supérieure du curseur, et ***ligneinf*** le numéro de la ligne de pixels inférieure. Ils doivent être compris entre 0 et 31. 0 correspond à la ligne la plus haute de la case dans laquelle se trouve le curseur, et le numéro correspondant à la ligne la plus basse dépend du mode texte (si les caractères ont une taille de 8x16, ce numéro est 15). Ces paramètres sont facultatifs, tout comme ***curseur***, mais s'ils sont précisés, le paramètre ***curseur*** doit l'être aussi.

Voici un exemple d'utilisation de **LOCATE**, qui écrit « Bonjour » au milieu de l'écran :

```
WIDTH 80, 25
LOCATE 12, 36
PRINT "Bonjour"
```

L'instruction **WIDTH** permet au programmeur de s'assurer que le format du mode est bien 80x25. Si c'est déjà le cas, elle n'est pas prise en compte par QBASIC.

3. Affichage avec des couleurs (instruction COLOR)

Jusqu'à présent, dans tous les exemples de ce cours, les caractères étaient affichés à l'écran avec une couleur grise, et sur fond noir. Cependant, il est possible de changer la couleur des caractères et la couleur du fond des cases. On peut utiliser 16 couleurs différentes pour le texte, et 8 couleurs différentes pour le fond. Le changement se fait à l'aide de la commande **COLOR**. Voici les paramètres de cette instruction :

COLOR ***coultxt*** **[,** ***coulfond*****]**

coultxt est « l'attribut » (le numéro) de la couleur du texte, et ***coulfond*** l'attribut de la couleur de fond. Le paramètre ***coulfond*** est facultatif.

Voici un tableau des couleurs disponibles :

Attribut	Couleur	Attribut	Couleur	Attribut	Couleur
0	Noir	6	Marron	12	Rouge clair
1	Bleu	7	Gris clair	13	Violet clair
2	Vert	8	Gris	14	Jaune
3	Cyan	9	Bleu clair	15	Blanc
4	Rouge	10	Vert clair		
5	Violet	11	Cyan clair		

Toutes ces couleurs sont utilisables pour les caractères, mais seules les huit premières (attributs 0 à 7) peuvent servir pour la couleur de fond. Il est possible de faire clignoter les caractères en ajoutant 16 à l'attribut de couleur du texte. Notez que si vous utilisez QBASIC dans une fenêtre Windows, il est possible que les caractères ne clignotent pas, mais que la couleur de fond corresponde à une des 8 dernières couleurs du tableau (attributs 8 à 15), comme si l'on avait ajouté 8 à l'attribut de couleur de fond, au lieu d'ajouter 16 à l'attribut de couleur du texte.

Ces lignes permettent d'afficher « Bonjour » en bleu clignotant, sur fond gris clair.

```
COLOR 17, 7
PRINT "Bonjour"
```

Si vous utilisez QBASIC dans une fenêtre Windows, il se peut que le texte soit écrit en bleu fixe sur un fond blanc.

4. Effacement de l'écran (instruction CLS)

Nous avons parlé des instructions permettant d'afficher du texte, et des options disponibles pour cela, mais pas encore du moyen d'effacer les caractères affichés. En effet, il peut être préférable, pour améliorer l'interface utilisateur d'un programme, d'effacer tout ce qui est écrit plutôt que de continuer à afficher du texte sur la dernière ligne en laissant les lignes précédentes se décaler. L'instruction qui permet cela est **CLS**. Elle s'utilise sans paramètres. Ainsi, cette ligne efface l'écran et repositionne le curseur en haut à gauche :

```
CLS
```

Notez que QBASIC n'efface pas l'écran entre deux exécutions successives d'un programme. Il est donc nécessaire d'insérer une instruction **CLS** au début du code pour effacer tout ce que le programme avait écrit à l'exécution précédente, ou même ce qu'un autre programme avait écrit.

Vous avez pu constater que lorsque l'on modifie la couleur de fond, le changement ne s'applique qu'au texte affiché après l'instruction **COLOR**, et non à tout l'écran. Pour ce faire, il suffit d'effacer l'écran avec une instruction **CLS** après avoir changé la couleur de fond. En effet, **CLS** affiche toutes les cases avec la couleur de fond courante et efface tous les caractères qu'elles contiennent. Cet exemple permet d'afficher l'écran en cyan :

```
COLOR 1, 3
CLS
```

5. Définition des limites de la fenêtre de texte (instruction VIEW PRINT)

Généralement, on utilise tout l'écran pour afficher du texte. Cependant, il est possible de limiter ce que l'on appelle la « fenêtre de texte », c'est-à-dire le nombre de lignes de l'écran sur lesquelles on peut afficher du texte. Prenons comme exemple le cas où la fenêtre s'étale de la ligne 5 à la ligne 10 de l'écran. Il n'est plus possible avec une instruction **LOCATE** de positionner le curseur ailleurs que dans les lignes 5 à 10. De plus, lorsque le curseur atteint la fin de la ligne 10, il est repositionné au début de cette même ligne, et les lignes 5 à 10 sont décalées d'un cran vers le haut.

Le principal intérêt de limiter la fenêtre de texte est que cela permet de n'effacer que certaines lignes de l'écran. En effet, la commande **CLS** n'efface que la fenêtre de texte, si elle n'a pas été étendue à tout l'écran. Ainsi, lorsque vous ne voulez effacer qu'un certain nombre de lignes, vous êtes obligés de redéfinir les dimensions de la fenêtre de texte avant d'utiliser **CLS**.

La commande qui permet de le faire est **VIEW PRINT**. Elle s'utilise ainsi :

VIEW PRINT [*ldépart* TO *lfin*]

ldépart est la limite supérieure de la fenêtre de texte, et ***lfin*** est sa limite inférieure. Ces paramètres sont facultatifs. S'ils sont omis, la fenêtre est étendue à tout l'écran.

L'instruction qui correspond à l'exemple que nous avons choisi est celle-ci :

```
VIEW PRINT 5 TO 10
```

Puis, pour rétablir la fenêtre de texte à tout l'écran, il suffit d'écrire :

```
VIEW PRINT
```

C) Exemple d'utilisation du mode texte

A présent, utilisons tout ce que nous avons vu pour créer un petit programme qui affichera un écran similaire à celui de QBASIC. Ce programme est enregistré dans le fichier « *MODETXT.BAS* ». Après l'avoir exécuté, examinez attentivement son code. Le voici :

```
REM ******************************************
REM **  Exemple d'utilisation du mode texte  **
REM ******************************************

WIDTH 80, 25


REM ***** Affichage de la barre de bienvenue *****
COLOR 0, 7      'Texte : noir ; Fond : gris clair
VIEW PRINT 1 TO 2
CLS
LOCATE 1, 13
PRINT "Bienvenue dans cet exemple d'utilisation du mode texte"


REM ***** Affichage de la fenêtre *****
COLOR 7, 1      'Texte : gris clair ; Fond : bleu
VIEW PRINT 2 TO 25
CLS
```

```
REM ** Ligne du haut (bord supérieur du cadre et titre) **
Ligne$ = CHR$(218)
FOR I = 1 TO 31: Ligne$ = Ligne$ + CHR$(196): NEXT
Ligne$ = Ligne$ + " Fenêtre QBASIC "
FOR I = 1 TO 31: Ligne$ = Ligne$ + CHR$(196): NEXT
Ligne$ = Ligne$ + CHR$(191)
LOCATE 2, 1
PRINT Ligne$

REM ** Côtés (bords latéraux du cadre) **
FOR I = 3 TO 23
    LOCATE I, 1
    PRINT CHR$(179)
    LOCATE I, 80
    PRINT CHR$(179)
NEXT

REM ** Ligne du bas (bord inférieur du cadre) **
Ligne$ = CHR$(192)
FOR I = 1 TO 78: Ligne$ = Ligne$ + CHR$(196): NEXT
Ligne$ = Ligne$ + CHR$(217)
LOCATE 24, 1
PRINT Ligne$


REM ***** Affichage de la barre d'état *****
COLOR 0, 7      'Texte : noir ; Fond : gris clair
VIEW PRINT 25 TO 25
CLS


REM ***** Attend que l'utilisateur appuie sur entrée *****
COLOR 7, 1      'Texte : gris clair ; Fond : bleu
VIEW PRINT
LOCATE 3, 2
INPUT Txt$
```

Le mode texte n'est pas très compliqué à utiliser. Les quelques commandes qu'il propose permettent d'améliorer l'apparence des programmes. On pourrait par exemple les utiliser dans l'exemple de programme de calcul afin de créer une interface plus conviviale.

Vous vous rendrez compte cependant que les possibilités qu'offre ce mode d'affichage sont limitées. Il convient tout à fait lorsque c'est le travail d'arrière-plan qui est privilégié (c'est-à-dire le travail qui n'est pas visible par l'utilisateur, comme les calculs), et que l'interface sert seulement à l'échange de quelques données entre utilisateur et ordinateur. Sa simplicité d'utilisation rend le temps nécessaire à la conception de l'interface relativement court, ce qui permet de consacrer le maximum de temps aux opérations qu'effectue le programme.

Mais il y a des programmes dont l'élément principal est l'interface. C'est le cas des jeux. En effet, tout se déroule sur l'écran : l'utilisateur, qui est en permanence aux commandes du jeu, réagit en fonction de ce qu'il y voit. Dans ce cas, il faut accorder beaucoup plus de temps à la programmation de l'interface. Cela nécessite plus de possibilités, qui ne sont offertes que par le mode graphique.

D) Instructions du mode graphique

1. Compatibilité des commandes du mode texte

Nous avons dit précédemment que l'on peut afficher du texte à l'écran en mode graphique, ainsi qu'en mode texte. Certaines des instructions utilisables en mode texte sont donc compatibles avec le mode graphique.

Les commandes **PRINT** et **INPUT** peuvent être employée exactement de la même manière qu'en mode texte. Il existe aussi un curseur, qui repère la position de l'écran où doit commencer l'affichage du texte, mais celui-ci n'est pas visible, sauf lors de l'interprétation d'une commande **INPUT**. Il peut être positionné grâce à l'instruction **LOCATE**, cependant les paramètres ***curseur***, ***départ*** et ***fin*** de celle-ci n'ont plus aucune utilité, puisque le curseur n'est pas visible.

Les instructions **COLOR** et **CLS** sont spécifiques au mode d'affichage de l'écran. Leur utilisation est donc différente en mode graphique et en mode texte. Nous allons voir cela un peu plus loin.

Enfin, la commande **VIEW PRINT** est disponible en mode graphique et s'utilise comme en mode texte.

2. Généralités

Comme nous l'avons vu, l'écran en mode graphique est divisé en pixels, que l'on peut afficher avec différentes couleurs. Il existe plusieurs instructions en mode graphique pour dessiner à l'écran. Elles permettent d'afficher des formes géométriques, de copier certaines parties de l'écran, ou encore de remplir des zones bien précises, et cela en affichant avec une même couleur plusieurs pixels de l'écran. Les pixels dont la couleur doit être modifiée sont déterminés par des calculs, effectués par ces instructions.

Les formes géométriques ne peuvent être dessinées à l'écran qu'avec une couleur unique. Le programmeur peut indiquer en paramètre de l'instruction qui affiche la forme géométrique la couleur qu'il souhaite utiliser pour la dessiner. S'il ne la précise pas, c'est la couleur de tracé courante qui est utilisée. C'est aussi avec celle-ci que les commandes **PRINT** et **INPUT** affichent du texte à l'écran. On peut la modifier grâce à l'instruction **COLOR**. Dans cet exemple, c'est le rouge qui est défini comme couleur de tracé courante :

```
COLOR 4
```

Contrairement au mode texte, **COLOR** ne permet pas de définir de couleur de fond pour le texte. Il n'y a donc qu'un seul paramètre à lui préciser.

Les couleurs sont les mêmes qu'en mode texte, et elles portent les mêmes attributs. Pour le mode 13, dans lequel 256 couleurs sont disponibles, les 16 premières sont les mêmes que dans les autres modes. A vous de découvrir les 240 restantes.

3. Affichage de points (instruction PSET)

Plutôt que d'utiliser des instructions pour dessiner des formes géométriques, vous pouvez afficher à l'écran simplement des points, c'est-à-dire afficher pixel par pixel avec des couleurs choisies. C'est la commande **PSET** qui permet cela. Elle s'utilise ainsi :

```
PSET (x, y) [, couleur]
```

x et ***y*** sont les coordonnées du point à afficher (***x*** est l'abscisse et ***y*** l'ordonnée).

couleur est un paramètre facultatif. S'il est précisé, le point est affiché avec cette couleur, sinon avec la couleur de tracé courante.

Voici un exemple qui affiche un point bleu en haut à gauche de l'écran, un point blanc au milieu de l'écran, et un point rouge en bas à droite de l'écran, en mode 12 :

```
SCREEN 12
PSET (0, 0), 1
PSET (319, 239), 15
PSET (639, 479), 4
```

Cet exemple-ci permet de voir toute la palette de couleur du mode 13 :

```
SCREEN 13
FOR I = 0 TO 255
    FOR J = 0 TO 199: PSET (I, J), I: NEXT
NEXT
```

La variable **I** est à la fois utilisée pour les abscisses et les couleurs des points à afficher. En la faisant évoluer de 0 à 255, toutes les couleurs disponibles sont utilisées, et chacune est affichée sur une abscisse différente. La seconde boucle **FOR…NEXT** permet d'afficher chaque couleur sur une ligne verticale occupant toute la hauteur de l'écran, de sorte qu'elle soit bien visible.

4. Affichage de lignes et de rectangles (instruction LINE)

La forme géométrique la plus simple est la ligne droite (le segment). L'instruction qui permet d'en dessiner est **LINE**. Il suffit de lui préciser les coordonnées des deux extrémités de la ligne. Voici comment sont disposés ses paramètres :

```
LINE (x1, y1)-(x2, y2) [, couleur]
```

x1 et ***y1*** sont les coordonnées de la première extrémité de la ligne. ***x2*** et ***y2*** sont les coordonnées de la seconde extrémité.

Ces instructions affichent des lignes obliques croisées deux à deux sur toute la largeur de l'écran :

```
SCREEN 13
FOR I = -20 TO 300 STEP 40
    LINE (I, 0)-(I + 39, 199), I + 21
    LINE (I, 199)-(I + 39, 0), I + 21
NEXT
```

La boucle **FOR…NEXT** permet d'afficher des lignes tous les 40 pixels (grâce au mot-clé **STEP**), et sur toute la largeur de l'écran. En effet, **I** est utilisée pour les abscisses des extrémités des lignes. Chaque ligne s'étale sur 40 pixels de large et 200 pixels de haut. A chaque boucle, deux lignes croisées sont affichées, avec la même couleur.

Vous remarquez que certaines lignes sont dessinées en partie hors de l'écran. En effet, lors du premier tour de la boucle, l'abscisse de la première extrémité des lignes est –20, et lors du dernier tour, l'abscisse de la seconde extrémité des lignes est 339. QBASIC ne génère pas d'erreur pour cela : il se contente de n'afficher que la partie de la ligne située dans l'écran. C'est valable pour toutes les autres formes géométriques.

LINE peut aussi être utilisée pour afficher des rectangles. Des paramètres doivent être ajoutés ainsi :

LINE (*x1*, *y1*)-(*x2*, *y2*), [*couleur*], {B | BF}

x1 et ***y1*** représentent les coordonnées d'un des coins du rectangle, et ***x2*** et ***y2*** les coordonnées du coin opposé. Ce peuvent être par exemple les coordonnées du coin supérieur gauche et du coin inférieur droit.

B indique qu'un rectangle simple doit être tracé (on affiche simplement le cadre). **BF** indique qu'un rectangle rempli doit être tracé. L'un ou l'autre de ces paramètres doit être précisé.

Le paramètre ***couleur*** est facultatif. Cependant, s'il est omis, la virgule qui le précède doit être conservée. Voici comment on affiche un rectangle en utilisant la couleur de tracé courante :

```
SCREEN 13
LINE (0, 0)-(319, 199), , B
```

Si **BF** est précisé en paramètre au lieu de **B** dans cet exemple, l'instruction **LINE** remplit tout l'écran avec la couleur de tracé courante.

Cet autre exemple affiche plusieurs rectangles emboîtés avec des couleurs différentes :

```
SCREEN 12
FOR I = 0 TO 200 STEP 50
    LINE (I, I)-(639 - I, 479 - I), I / 50 + 1, BF
    LINE (I, I)-(639 - I, 479 - I), 15, B
NEXT
```

La boucle **FOR…NEXT** permet d'afficher plusieurs rectangles en décalant les coins supérieur gauche et inférieur droit de 50 pixels en abscisses et en ordonnées à chaque tour. La première instruction **LINE** dessine un rectangle rempli de couleur, et la seconde dessine son cadre en blanc. Puisque **I** évolue de 0 à 200 par intervalle de 50, diviser **I** par 50 en lui ajoutant 1 permet de faire évoluer la couleur de remplissage de 1 à 5.

5. Affichage de cercles et d'ellipses (instruction CIRCLE)

Grâce à l'instruction **CIRCLE**, vous pouvez afficher des cercles à l'écran :

CIRCLE (*x*, *y*), *rayon* [, *couleur*]

x et ***y*** sont les coordonnées du centre du cercle.

rayon doit être précisé en unités du système de coordonnées courant, qui dépend de la résolution de l'écran. En fait, l'unité des abscisses n'est pas toujours la même que l'unité des ordonnées. Cela signifie que la largeur en pixels d'un cercle que vous dessinez peut être différente de sa hauteur en pixels. Ce procédé permet de compenser la disproportion entre la largeur et la hauteur des pixels avec certaines résolutions, et d'éviter que les cercles apparaissent comme des ellipses à l'écran. C'est le cas dans tous les modes sauf le mode 12. En effet, sa résolution 640x480 correspond au format $^4/_3$ de l'écran ($^{640}/_{480} = {}^4/_3$).

J'expliquerai plus loin comment fonctionne ce système de coordonnées. Mais si vous ne comprenez pas l'intérêt d'utiliser des unités différentes en abscisses et en ordonnées, ne vous en préoccupez pas. Pour trouver le rayon qui convient au cercle que vous souhaitez afficher, cherchez-le par tâtonnement en essayant successivement différentes valeurs, plutôt que d'essayer de le calculer.

Voici un exemple d'utilisation de **CIRCLE** :

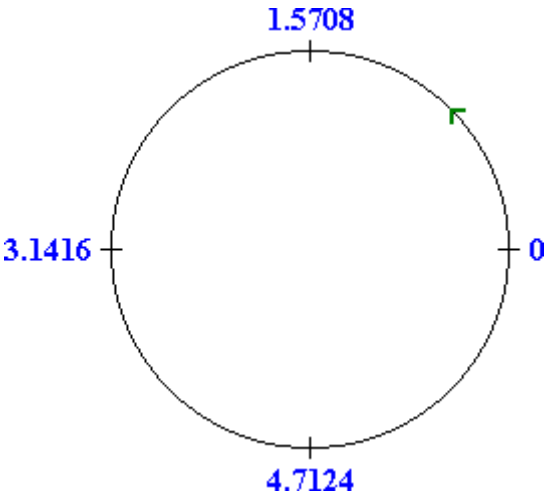
```
SCREEN 12
FOR I = 20 TO 300 STEP 20
    CIRCLE (320, 240), I, I / 20
NEXT
```

Ces instructions permettent d'afficher plusieurs cercles de même centre, avec des rayons différents et des couleurs différentes.

CIRCLE permet aussi d'afficher des arcs de cercles, c'est-à-dire des cercles incomplets. Il faut pour cela préciser en paramètres les angles de début et de fin de l'arc de cercle :

CIRCLE (*x*, *y*), *rayon*, [*couleur*], *angledéb*, *anglefin*

angledéb et ***anglefin*** doivent être en radians. Si vous ne savez pas à quoi correspondent les angles en radians, voici un schéma qui précise les valeurs de référence :



Chaque valeur d'angle représente un emplacement sur le cercle. Ces valeurs varient de 0 à un peu moins de 6.2832. Une valeur comprise entre 0 et 1.5708 représente un point situé sur le quart supérieur droit du cercle, et ainsi de suite.

L'arc de cercle est affiché entre l'angle de début et l'angle de fin, dans le sens indiqué par la flèche verte sur le schéma. Ainsi, si on prend 0 comme valeur pour l'angle de début et 1.5708 pour l'angle de fin, l'arc de cercle correspond au quart supérieur droit :

```
SCREEN 12
CIRCLE (320, 240), 100, , 0, 1.5708
```

Si on inverse l'angle de début et l'angle de fin, l'arc de cercle correspondra alors aux trois quarts de cercle complémentaires au quart supérieur droit.

Un dernier paramètre peut enfin être précisé à l'instruction **CIRCLE** : c'est l'aspect du cercle, c'est-à-dire le rapport de sa hauteur sur sa largeur. Il permet de déformer le cercle afin de lui donner la forme d'une ellipse. Ce paramètre doit être indiqué en dernier :

CIRCLE (*x*, *y*), *rayon*, [*couleur*], [*angledéb*], [*anglefin*], *aspect*

La valeur qu'il faut donner à ***aspect*** pour obtenir une certaine déformation dépend du système de coordonnées, donc de la résolution. Pour que l'ellipse soit un cercle, l'aspect doit être égal à la valeur de référence : (4/3) x (hauteur/largeur), où hauteur et largeur correspondent à la résolution de l'écran. Pour le mode 320x200, cette valeur sera (4/3) x (200/320) = 0,8333. Si l'aspect est inférieur à la valeur de référence, l'ellipse est plus large que haute, et s'il lui est supérieur, l'ellipse est plus haute que large.

Si vous ne comprenez pas ces calculs (que je n'ai d'ailleurs pas justifiés pour éviter d'alourdir le cours), contentez-vous de la méthode du tâtonnement pour trouver l'aspect adéquat.

Complétons l'exemple des cercles concentriques :

```
SCREEN 12
FOR I = 20 TO 300 STEP 20
    CIRCLE (320, 240), I, I / 20, , , .75
NEXT
```

En mode 12 (640x480), la valeur de référence est (4/3) x (480/640) = 1. L'aspect précisé dans l'exemple est 0.75, c'est-à-dire trois quarts de la valeur de référence. Ainsi la hauteur des ellipses est égale aux trois quarts de leur largeur. Cela correspond au format $^4/_3$ de l'écran ($^4/_3$ est le rapport de la largeur sur la hauteur), et les cercles ainsi déformés sont désormais tous entièrement visibles à l'écran, alors que dans l'exemple précédent les plus grands étaient coupés en haut et en bas.

6. Remplissage d'une zone de l'écran (instruction `PAINT`)

Nous venons de voir les quatre types de formes géométriques que QBASIC peut tracer à l'écran. Cependant, à part les rectangles, les instructions dont nous avons parlé ne permettent pas de les remplir.

Pour cela, c'est l'instruction **`PAINT`** qu'il faut utiliser. Elle permet de remplir l'intérieur d'une forme géométrique au contour fermé. Ce peut être un cercle, une ellipse, un polygone, ou bien une forme quelconque composée de lignes et d'arcs de cercles ou d'ellipses. Il faut impérativement que le contour de la forme soit fermé et d'une couleur unique, sinon **`PAINT`** risque de remplir tout l'écran ou presque et d'effacer les dessins situés à l'extérieur de la forme.

Pour l'utiliser, il suffit de lui indiquer les coordonnées d'un pixel situé à l'intérieur de la forme à remplir, la couleur du contour de cette forme, et la couleur de remplissage. Notez que si des graphismes ont été dessinés à l'intérieur de la zone à remplir, ils seront effacés.

`PAINT (`*`x`*`, `*`y`*`), [`*`coulremp`*`], `*`coulcont`

`x` et ***`y`*** sont les coordonnées d'un pixel situé dans la zone à remplir.

`coulremp` est la couleur de remplissage. Ce paramètre est facultatif. S'il est omis, c'est la couleur de tracé courante qui est utilisée.

`coulcont` est la couleur du contour de la zone à remplir.

Voici un exemple de remplissage de forme géométrique dessinée à l'écran (il est enregistré dans le fichier « *CHAT.BAS* ») :

```
SCREEN 13

REM ***** Corps *****
CIRCLE (160, 113), 50, 1, 2.07, 1.08, 1.5

REM ***** Tête *****
CIRCLE (145, 47), 25, 1, 3.1416, 4.7124
CIRCLE (175, 47), 25, 1, 4.7124, 0
CIRCLE (120, 25), 25, 1, 4.7124, 0
CIRCLE (200, 25), 25, 1, 3.1416, 4.7124
LINE (145, 25)-(150, 35), 1
```

```
LINE (175, 25)-(170, 35), 1
CIRCLE (160, 50), 25, 1, 1.15, 1.92, .68

REM ***** Remplissage *****
PAINT (160, 113), 2, 1
```

Les premières instructions dessinent en bleu une forme de chat, et l'instruction **PAINT** la remplit en vert.

7. Copie d'une zone de l'écran (instructions GET et PUT)

Vous aurez souvent besoin d'afficher plusieurs fois un même dessin. Pour cela, ce n'est pas la peine de réécrire à chaque fois les instructions de tracé et de remplissage. Il suffit de copier le dessin à l'aide du mot-clé **GET**, et de le coller à un autre emplacement de l'écran grâce à **PUT**.

GET permet de stocker dans une variable la couleur de tous les pixels d'une zone rectangulaire de l'écran spécifiée par le programmeur :

GET (*x1*, *y1*)-(*x2*, *y2*), *image*

x1 et ***y1*** représentent les coordonnées d'un des coins de la zone rectangulaire, et ***x2*** et ***y2*** les coordonnées du coin opposé.

image est la variable dans laquelle doit être stockée la couleur des pixels. Les attributs de couleur sont des données de type nombre. Toutes les variables de type nombre que nous avons utilisées précédemment ne permettaient de stocker qu'un seul nombre. Or ***image*** doit pouvoir stocker beaucoup plus qu'un seul nombre. Il faut donc déclarer sa taille à QBASIC avant qu'elle puisse être utilisée par l'instruction **GET**. Cela se fait à l'aide du mot-clé **DIM** :

DIM *image*(*taille*)

Cette instruction peut être placée n'importe où dans le code, mais elle doit impérativement être exécutée avant l'instruction **GET**. L'idéal est de la placer en tout début de programme.

taille doit être calculée à partir des dimensions de la zone rectangulaire. La méthode de calcul diffère selon le mode graphique. La largeur correspond à la différence des abscisses à laquelle on ajoute 1 (si ***x2*** > ***x1***, ***largeur*** = ***x2*** - ***x1*** + 1 ; si ***x1*** > ***x2***, ***largeur*** = ***x1*** - ***x2*** + 1). La hauteur correspond à la différence des ordonnées à laquelle on ajoute 1.

Pour les modes 7, 8, 9 et 12, le calcul est le suivant :

taille = (*largeur* / 8) x *hauteur*

Avant de multiplier (*largeur* / 8) par *hauteur*, il faut vérifier si le résultat de cette division est un nombre entier, et si ce n'est pas le cas, l'arrondir à l'entier supérieur. Par exemple, pour une zone de 25 pixels de large sur 50 de hauteur, on a 25 / 8 = 3.125, que l'on arrondit à 4, puis ***taille*** = 4 x 50 = 200. Notez que vous pouvez très bien déclarer ***image*** avec une taille supérieure à celle trouvée par le calcul, mais si vous la déclarez avec une taille inférieure, QBASIC générera une erreur lors de l'exécution de l'instruction **GET**.

Pour le mode 13, le calcul est différent :

taille = (*largeur* x *hauteur*) / 4

Si le nombre trouvé par le calcul n'est pas un entier, vous devez l'arrondir à l'entier supérieur. Pour une zone de 25 pixels de large sur 50 de hauteur, ***taille*** = (25 x 50) / 4 = 312.5, que l'on arrondit à 313.

PUT permet d'afficher, à un emplacement de l'écran spécifié par le programmeur, une image stockée dans une variable par une instruction **GET** :

PUT (*x*, *y*), *image*, {PSET | PRESET | AND | OR | XOR}

x et ***y*** sont les coordonnées du pixel où doit être affiché le coin supérieur gauche de l'image.

Le mot-clé indiqué en dernier paramètre précise la méthode employée pour afficher l'image. **PSET** permet de l'afficher telle qu'elle a été copiée.

PRESET indique que l'image doit être affichée en couleurs inversées. Appelons *c* l'attribut de couleur d'un des pixels à afficher. En mode 7, 8, 9 ou 12, ce pixel sera affiché avec la couleur 15 - *c*. Ainsi, au noir correspondra le blanc, au bleu correspondra le jaune, etc. En mode 13, le pixel sera affiché avec la couleur 255 - *c*.

Enfin, les mots-clés **AND**, **OR** et **XOR** précisent que l'image doit être fusionnée ou superposée avec les graphismes présents à l'écran à l'endroit où elle est affichée. Leur principe de fonctionnement est trop compliqué pour être expliqué dans le cadre de ce cours. Alors, à vous de découvrir ces méthodes particulières en les essayant.

Notez qu'il est possible de copier une zone de l'écran en utilisant un certain mode graphique, puis d'afficher l'image stockée après être passé sous un autre mode. C'est valable pour les modes 7, 8, 9 et 12, puisqu'ils fonctionnent avec le même nombre de couleurs. En revanche, le mode 13 n'est compatible avec aucun autre.

Le copier-coller effectué avec les instructions **GET** et **PUT** a cependant un inconvénient : c'est qu'il ne permet de stocker que des images strictement rectangulaires, et qu'il efface les graphismes présents à l'écran à l'endroit ou celles-ci sont affichées. Nous verrons au chapitre suivant une astuce pour faire se déplacer un objet à l'écran sans effacer le fond sur lequel il est affiché.

Notez aussi qu'il n'est pas possible de ne coller que la moitié d'une image, en faisant en sorte qu'elle se trouve à cheval sur l'écran et l'extérieur. Si vous essayez de le faire, QBASIC générera une erreur. Ainsi, en collant aux coordonnées (600, 100) une image de largeur 50, vous provoquerez l'arrêt du programme, puisque l'image déborde du côté droit (600 + 50 = 650 et 650 > 640). Bien sûr, il en est de même avec **GET**.

Reprenons l'exemple de la forme de chat. Ajoutez ces lignes au début du programme, pour déclarer la taille de la variable ***image*** :

```
SCREEN 13
DIM Chat(2815)
```

Les coordonnées de la zone copiée sont (120, 25)-(200, 163). Voici les calculs qui ont permis de déterminer la taille :

Taille = [(200 - 120 + 1) * (163 - 25 + 1)] / 4
= [(81) * (139)] / 4
= 2814.75

Puis on a arrondi 2814.75 à 2815.

A la fin du programme maintenant, ajoutez les lignes suivantes, qui copient la forme du chat et l'affichent à gauche et à droite de l'écran :

```
REM ***** Copier-coller de la forme de chat *****
GET (120, 25)-(200, 163), Chat
PUT (0, 25), Chat, PSET
PUT (239, 25), Chat, PSET
```

8. Effacement de l'écran (instruction CLS)

Ainsi qu'en mode texte, il est possible d'effacer l'écran grâce au mot-clé **CLS**. Cependant, à la différence du mode texte, il n'est pas possible de changer la couleur de fond. L'instruction **CLS** ne fait donc qu'afficher tous les pixels de l'écran avec la couleur noire.

Nous avons déjà dit que l'instruction **VIEW PRINT** peut être utilisée en mode graphique de la même manière qu'en mode texte. Cependant, même si la fenêtre de texte est plus petite que l'écran, l'instruction **CLS** effacera tout l'écran. Pour la forcer à n'effacer que la fenêtre de texte, il faut la faire suivre du nombre 2 :

```
CLS 2
```

Notez que tous les graphismes affichés dans la fenêtre de texte seront effacés en même temps que les caractères.

Conclusion

Vous savez maintenant à peu près tout sur la programmation de l'interface visuelle en QBASIC. Vous avez appris à vous servir des deux modes d'affichages, et dans la situation adaptée. Selon ce que doit faire un programme, il est plus judicieux de choisir l'un ou l'autre des modes d'affichages. Si l'interface utilisateur ne joue qu'un rôle secondaire, le mode texte est plus approprié. Si elle joue un rôle capital, c'est le mode graphique qu'il faut utiliser.

Bien sûr, il vous faudra du temps avant de pouvoir bien maîtriser les commandes qui régissent l'interface, notamment en ce qui concerne le mode graphique. Et comme pour tout le reste, il n'y a qu'en écrivant vous-même de nombreux programmes que vous y arriverez.

Chapitre VI :
Exemples de programmes graphiques

Si l'utilisation du mode texte ne vous a pas posé de problèmes particuliers, il n'en est probablement pas de même pour le mode graphique. La difficulté tient notamment au fait que vous ne pouvez dessiner des graphismes qu'en utilisant des commandes qui tracent des figures géométriques, et qu'il faut pour cela que vous vous habituiez au système des coordonnées. Voici donc des exemples de programmes qui devraient vous permettre de vous familiariser avec ce système.

Les exemples de ce chapitre sont plus « pointus » que ceux des chapitres précédents. Portez donc une attention particulière au code de ces petits programmes, et lancez-les plusieurs fois, afin de comprendre leur fonctionnement.

A) Dessin de décors

Voici deux petits programmes qui utilisent l'ensemble des commandes vues au chapitre précédent pour dessiner à l'écran un décor. Ils ne sont pas commentés dans ce cours, donc il vous faudra examiner minutieusement le code pour comprendre ce que font les instructions. Il est possible notamment d'insérer des apostrophes devant certaines commandes pour éviter qu'elles soient exécutées. De cette manière vous pourrez exécuter instruction par instruction, et voir quelles formes géométriques elles dessinent. Ainsi vous reconstituerez toutes les étapes qui permettent d'afficher le décor.

1. Parterre avec de l'herbe

Ce petit programme dessine à l'écran un rectangle d'herbe de 32 pixels de large sur 15 de hauteur, et le copie sur toute la largeur de l'écran. L'intérêt d'utiliser le mode 13 est qu'il est possible d'utiliser plusieurs nuances de vert pour l'herbe (attributs 2, 118 et 122, et d'autres encore), grâce à la palette 256 couleurs. Dans les autres modes, il n'aurait été possible d'utiliser que les attributs 2 et 10.

Son code est enregistré dans le fichier « *HERBE.BAS* ». Le voici :

```
SCREEN 13
DIM Herbe(120)      'Voir instruction GET
                    'Taille = [(31 - 0 + 1) * (179 - 165 + 1)] / 4
                    '       = [(    32    ) * (     15      )] / 4
                    '       = 120

REM ***** Dessin de l'arrière-plan (ciel et terre) *****
PAINT (0, 0), 55
LINE (0, 180)-(319, 199), 114, BF
```

```
REM ***** Dessin d'un rectangle d'herbe *****
CIRCLE (8, 165), 14, 2, 3.1416, 4.5, 1.8
CIRCLE (23, 170), 15, 122, 3.3, 4.7124, .6
CIRCLE (8, 179), 13, 118, 1.7, 3.1416, 2
LINE (5, 179)-(10, 165), 122
LINE (7, 179)-(15, 166), 2
LINE (15, 179)-(11, 169), 118
LINE (10, 179)-(15, 171), 118
LINE (14, 179)-(17, 170), 2
CIRCLE (12, 167), 12, 122, 4.8, 0, 1.5
LINE (17, 179)-(23, 174), 2
LINE (20, 179)-(25, 167), 2
LINE (26, 179)-(27, 166), 118
CIRCLE (32, 164), 15, 122, 3.2, 4.6, 1.5
CIRCLE (35, 179), 13, 2, 2.1, 3.1416, 2

REM ***** Copier-coller du rectangle d'herbe *****
GET (0, 165)-(31, 179), Herbe
FOR I = 32 TO 288 STEP 32: PUT (I, 165), Herbe, PSET: NEXT
```

1. Parterre avec des fleurs

Ce programme dessine une fleur à l'écran, et la copie à intervalles réguliers sur toute la largeur de l'écran. L'intérêt d'utiliser le mode 12 est la haute résolution qu'il offre (640x480), et donc la petite taille des pixels. En effet, le dessin d'une fleur doit être fait en détail, et ce mode est le seul qui permette d'afficher des graphismes aussi petits avec autant de précision.

Le code du programme est enregistré dans le fichier « *FLEURS.BAS* » :

```
SCREEN 12
DIM Fleur(435)      'Voir instruction GET
                    'Taille = [(37 - 3 + 1) / 8] * (449 - 363 + 1)
                    '       = [(    35    ) / 8] * (     87      )
                    '       = [   4.375 -> 5   ] * (     87      )
                    '       = 435

REM ***** Dessin de l'arrière-plan (ciel et terre) *****
PAINT (0, 0), 9
LINE (0, 450)-(639, 479), 6, BF


REM ***** Dessin d'une fleur *****
REM ** Tige **
COLOR 2
LINE (10, 400)-(12, 449), , BF
CIRCLE (21, 400), 22, , 1.6, 3.1, 2
CIRCLE (22, 400), 20, , 1.6, 3.1, 2
CIRCLE (19, 400), 20, , 1.6, 3.1, 2.5
LINE (21, 378)-(19, 380), , BF
LINE (14, 386)-(15, 384)
```

```
REM ** Feuilles **
CIRCLE (16, 420), 3, , 0, 3.1416
CIRCLE (16, 420), 6, , 4.8, 0, 2
CIRCLE (18, 437), 10, , 1.8, 2.7, 2
PAINT (16, 420)
CIRCLE (6, 400), 3, , 0, 3.1416
CIRCLE (6, 400), 6, , 3.1416, 4.6, 2
CIRCLE (4, 417), 10, , .4, 1.3, 2
PAINT (6, 400)

REM ** Pétales **
CIRCLE (10, 380), 7, 15, , , .9
CIRCLE (30, 380), 7, 15, , , .9
CIRCLE (20, 370), 7, 15, , , 1.111
CIRCLE (20, 390), 7, 15, , , 1.111
PAINT (10, 380), 15, 15
PAINT (30, 380), 15, 15
PAINT (20, 370), 15, 15
PAINT (20, 390), 15, 15

REM ** Centre **
CIRCLE (20, 380), 5, 4
PAINT (20, 380), 14, 4


REM ***** Copier-coller des fleurs *****
GET (3, 363)-(37, 449), Fleur
FOR I = 103 TO 603 STEP 100: PUT (I, 363), Fleur, PSET: NEXT


REM ***** Dessin de l'herbe *****
FOR I = 1 TO 637 STEP 4
LINE (I, 449)-(I + 2, 440), 10: NEXT
```

B) Animation

On a vu comment dessiner un décor, mais ce qui est intéressant pour créer un jeu, c'est surtout de l'animer. C'est pourquoi nous allons étudier un exemple qui illustre l'utilisation de chacune des instructions du mode graphique, en affichant à l'écran une petite animation : une voiture qui se déplace sur une route.

Le programme est enregistré dans le fichier « *ANIM.BAS* ». Lorsque vous le lancerez, il y a de fortes chances pour que l'animation soit trop rapide ou trop lente. Cela est dû au fait que la temporisation est effectuée en fonction de la vitesse de l'ordinateur. Vous pouvez régler la vitesse de l'animation en modifiant la valeur des paramètres ***fin*** des boucles **FOR…NEXT** dans la sous-routine **Temporise** (par défaut les valeurs sont 10 et 2000). Pour ralentir l'animation, il faut augmenter la valeur des deux paramètres ou d'un seul des deux, et pour l'accélérer, il faut diminuer leur valeur. Modifiez les paramètres jusqu'à ce que la vitesse de l'animation soit optimale.

Après avoir lancé le programme, examinez rapidement le code copié ci-dessous, puis lisez l'analyse. Observez alors attentivement le code afin de le comprendre. Cet exemple est le plus long et le plus compliqué du cours, et c'est par conséquent celui qui vous prendra le plus de temps.

1. Code du programme

```
SCREEN 12
DIM Fond(175)      'Voir instruction GET
                   'Dimensions des images à copier : 37x17 / 17x37 /
35x35
                   'Taille = (  35 / 8  ) * 35
                   '       = (4.375 -> 5) * 35
                   '       = 175

GOSUB DessDecor
GOTO Animation


Temporise:
REM ***** Temporisation de l'animation *****
REM  ***    Régler les paramètres des    ***
REM  ***   boucles FOR pour augmenter    ***
REM *****    ou diminuer la vitesse     *****
FOR Temp1 = 1 TO 10
    FOR Temp2 = 1 TO 2000: NEXT
NEXT
RETURN


DessDecor:
REM ** Routes **
COLOR 8      'Gris foncé
LINE (0, 180)-(540, 240), , BF
LINE (180, 0)-(240, 479), , BF
LINE (480, 0)-(540, 180), , BF
LINE (360, 240)-(420, 360), , BF
LINE (420, 300)-(639, 360), , BF
CIRCLE (465, 165), 15, , 4.7124, 0
PAINT (479, 179)
CIRCLE (480, 180), 60, 0, 4.7124, 0
PAINT (540, 240), 0, 0
CIRCLE (435, 285), 15, , 3.1416, 4.7124
PAINT (421, 299)
CIRCLE (420, 300), 60, 0, 3.1416, 4.7124
PAINT (360, 360), 0, 0

REM ** Lignes de délimitation des voies **
COLOR 15      'Blanc
FOR I = 15 TO 145 STEP 26
    LINE (I, 210)-(I + 5, 210)
    LINE (210, I)-(210, I + 5)
NEXT
```

```
FOR I = 270 TO 452 STEP 26
    LINE (I, 210)-(I + 5, 210)
    LINE (210, I)-(210, I + 5)
NEXT
LINE (478, 210)-(481, 210)
LINE (482, 209)-(483, 209)
LINE (499, 199)-(501, 197)
LINE (501, 196)-(502, 195)
FOR I = 18 TO 174 STEP 26: LINE (510, I)-(510, I + 5): NEXT
LINE (390, 270)-(390, 275)
LINE (390, 296)-(390, 301)
LINE (400, 318)-(403, 321)
FOR I = 420 TO 628 STEP 26: LINE (I, 330)-(I + 5, 330): NEXT

REM ** Passages pour piétons **
FOR I = 185 TO 235 STEP 8
    LINE (160, I)-(174, I + 2), , BF
    LINE (246, I)-(260, I + 2), , BF
    LINE (I, 160)-(I + 2, 174), , BF
    LINE (I, 246)-(I + 2, 260), , BF
    LINE (I + 180, 246)-(I + 182, 260), , BF
NEXT

RETURN


DessVoitD:
REM ***** Dessin de la voiture se dirigeant vers la droite *****
REM ** Forme **
COLOR 4     'Rouge
LINE (X + 3, Y)-(X + 31, Y + 16), , BF
LINE (X + 1, Y + 1)-(X + 2, Y + 15), , BF
LINE (X, Y + 2)-(X, Y + 14)
LINE (X + 32, Y + 1)-(X + 34, Y + 15), , BF
LINE (X + 35, Y + 2)-(X + 36, Y + 14), , BF

REM ** Contours de la carrosserie **
COLOR 0     'Noir
LINE (X + 7, Y + 2)-(X + 30, Y + 2)
LINE (X + 7, Y + 14)-(X + 30, Y + 14)
LINE (X + 7, Y + 2)-(X + 7, Y + 14)
LINE (X + 7, Y + 2)-(X + 3, Y + 4)
LINE (X + 7, Y + 14)-(X + 3, Y + 12)
LINE (X + 3, Y + 4)-(X + 3, Y + 12)
LINE (X + 24, Y + 4)-(X + 24, Y + 12)
LINE (X + 25, Y + 3)-(X + 27, Y + 3)
LINE (X + 25, Y + 13)-(X + 27, Y + 13)
LINE (X + 32, Y + 4)-(X + 32, Y + 12)
PSET (X + 31, Y + 3)
PSET (X + 31, Y + 13)

REM ** Vitres **
PAINT (X + 26, Y + 8), 9, 0
PAINT (X + 4, Y + 8), 9, 0
```

```
REM ** Phares
COLOR 14      'Jaune
LINE (X + 36, Y + 4)-(X + 36, Y + 5), , BF
LINE (X + 36, Y + 11)-(X + 36, Y + 12), , BF

RETURN


DessVoitB:
REM ***** Dessin de la voiture se dirigeant vers le bas *****
REM ** Forme **
COLOR 4      'Rouge
LINE (X, Y + 3)-(X + 16, Y + 31), , BF
LINE (X + 1, Y + 1)-(X + 15, Y + 2), , BF
LINE (X + 2, Y)-(X + 14, Y)
LINE (X + 1, Y + 32)-(X + 15, Y + 34), , BF
LINE (X + 2, Y + 35)-(X + 14, Y + 36), , BF

REM ** Contours de la carrosserie **
COLOR 0      'Noir
LINE (X + 2, Y + 7)-(X + 2, Y + 30)
LINE (X + 14, Y + 7)-(X + 14, Y + 30)
LINE (X + 2, Y + 7)-(X + 14, Y + 7)
LINE (X + 2, Y + 6)-(X + 3, Y + 4)
LINE (X + 14, Y + 7)-(X + 12, Y + 3)
LINE (X + 4, Y + 3)-(X + 12, Y + 3)
LINE (X + 4, Y + 24)-(X + 12, Y + 24)
LINE (X + 3, Y + 25)-(X + 3, Y + 27)
LINE (X + 13, Y + 25)-(X + 13, Y + 27)
LINE (X + 4, Y + 32)-(X + 12, Y + 32)
PSET (X + 3, Y + 31)
PSET (X + 13, Y + 31)

REM ** Vitres **
PAINT (X + 8, Y + 26), 9, 0
PAINT (X + 8, Y + 4), 9, 0

REM ** Phares
COLOR 14      'Jaune
LINE (X + 4, Y + 36)-(X + 5, Y + 36), , BF
LINE (X + 11, Y + 36)-(X + 12, Y + 36), , BF

RETURN


DessVoitBD:
REM ***** Dessin de la voiture se dirigeant vers le bas et la droite
*****
REM ** Forme **
COLOR 4      'Rouge
LINE (X + 2, Y + 13)-(X + 13, Y + 2)
LINE (X + 13, Y + 2)-(X + 32, Y + 21)
LINE (X + 32, Y + 21)-(X + 21, Y + 32)
LINE (X + 21, Y + 32)-(X + 2, Y + 13)
LINE (X + 2, Y + 12)-(X + 12, Y + 2)
LINE (X + 2, Y + 11)-(X + 11, Y + 2)
```

```
LINE (X + 2, Y + 10)-(X + 10, Y + 2)
LINE (X + 2, Y + 9)-(X + 9, Y + 2)
LINE (X + 32, Y + 22)-(X + 22, Y + 32)
LINE (X + 32, Y + 23)-(X + 23, Y + 32)
LINE (X + 33, Y + 23)-(X + 23, Y + 33)
LINE (X + 33, Y + 24)-(X + 24, Y + 33)
LINE (X + 34, Y + 24)-(X + 24, Y + 34)
LINE (X + 34, Y + 25)-(X + 25, Y + 34)
LINE (X + 34, Y + 26)-(X + 26, Y + 34)
LINE (X + 34, Y + 27)-(X + 27, Y + 34)
PAINT (X + 18, Y + 18)


REM ** Contours de la carrosserie **
COLOR 0      'Noir
LINE (X + 5, Y + 11)-(X + 11, Y + 5)
LINE (X + 8, Y + 14)-(X + 14, Y + 8)
LINE (X + 5, Y + 12)-(X + 6, Y + 14)
LINE (X + 12, Y + 5)-(X + 14, Y + 6)
LINE (X + 7, Y + 15)-(X + 22, Y + 30)
LINE (X + 15, Y + 7)-(X + 30, Y + 22)
PSET (X + 23, Y + 30)
PSET (X + 30, Y + 23)
LINE (X + 24, Y + 30)-(X + 30, Y + 24)
LINE (X + 19, Y + 26)-(X + 20, Y + 27)
LINE (X + 26, Y + 19)-(X + 27, Y + 20)
LINE (X + 19, Y + 25)-(X + 25, Y + 19)

REM ** Vitres **
PAINT (X + 10, Y + 10), 9, 0
PAINT (X + 26, Y + 26), 9, 0

REM ** Phares **
COLOR 14      'Jaune
LINE (X + 28, Y + 33)-(X + 29, Y + 32)
LINE (X + 33, Y + 28)-(X + 32, Y + 29)

RETURN


AffVoitD:
REM ***** Affiche la voiture qui avance vers la droite *****
GET (X, Y)-(X + 36, Y + 16), Fond
GOSUB DessVoitD
GOSUB Temporise
PUT (X, Y), Fond, PSET
RETURN
```

```
AffVoitB:
REM ***** Affiche la voiture qui avance vers le bas *****
GET (X, Y)-(X + 16, Y + 36), Fond
GOSUB DessVoitB
GOSUB Temporise
PUT (X, Y), Fond, PSET
RETURN


AffVoitBD:
REM ***** Affiche la voiture qui tourne *****
GET (X, Y)-(X + 34, Y + 34), Fond
GOSUB DessVoitBD
GOSUB Temporise
PUT (X, Y), Fond, PSET
RETURN


Animation:
REM ***** Animation de la voiture *****
REM ** Délai avant le début de l'animation **
FOR I = 1 TO 15: GOSUB Temporise: NEXT

REM ** Première ligne droite **
Y = 217
FOR X = 0 TO 176 STEP 4: GOSUB AffVoitD: NEXT

REM ** Premier tournant **
X = 182
Y = 219
GOSUB AffVoitBD
X = 187
Y = 221
GOSUB AffVoitB

REM ** Seconde ligne droite **
FOR Y = 221 TO 441 STEP 4: GOSUB AffVoitB: NEXT

REM ** Délai avant la reprise de l'animation **
FOR I = 1 TO 10: GOSUB Temporise: NEXT

REM ** Troisième ligne droite **
FOR Y = 2 TO 206 STEP 4: GOSUB AffVoitB: NEXT

REM ** Second tournant **
Y = 212
X = 189
GOSUB AffVoitBD
Y = 217
X = 191
GOSUB AffVoitD

REM ** Quatrième ligne droite **
FOR X = 191 TO 355 STEP 4: GOSUB AffVoitD: NEXT
```

```
REM ** Troisième tournant **
X = 361
Y = 219
GOSUB AffVoitBD
X = 367
Y = 221
GOSUB AffVoitB

REM ** Cinquième ligne droite **
FOR Y = 221 TO 285 STEP 4: GOSUB AffVoitB: NEXT

REM ** Dernier tournant **
Y = 285
FOR I = 1 TO 3
    Y = Y + 4
    X = X + 1
    GOSUB AffVoitB
NEXT
Y = 301
X = 372
GOSUB AffVoitB
Y = 307
X = 374
GOSUB AffVoitBD
FOR I = 1 TO 6
    X = X + 3
    Y = Y + 3
    GOSUB AffVoitBD
NEXT
X = 394
Y = 331
GOSUB AffVoitD
X = 398
Y = 333
GOSUB AffVoitD
FOR I = 1 TO 4
    X = X + 4
    Y = Y + 1
    GOSUB AffVoitD
NEXT

REM ** Dernière ligne droite **
FOR X = 414 TO 602 STEP 4: GOSUB AffVoitD: NEXT
```

2. Analyse du programme

Dans un premier temps, vous pouvez voir que la structure du code ressemble à celle de l'exemple de programme de calcul. Cependant elle est un peu différente. On peut regrouper les divers éléments du code en trois parties. La première est la partie d'initialisation du programme. Elle est constituée des quatre premières instructions (**SCREEN**, **DIM**, **GOSUB** et **GOTO**). La seconde partie est constituée de toutes les sous-routines (**Temporise**, **DessDecor**, **DessVoitD**, **DessVoitB**, **DessVoitBD**, **AffVoitD**, **AffVoitB** et **AffVoitBD**). Enfin la dernière partie est la partie principale du programme. Elle commence à l'étiquette **Animation** et se termine à la fin du programme.

Comme son nom l'indique, la première partie sert à initialiser le programme. L'instruction **SCREEN** initialise l'écran (elle passe en mode graphique 12), et l'instruction **DIM** initialise la variable **Fond** (elle déclare sa taille). L'instruction **GOSUB** appelle la sous-routine **DessDecor**, qui initialise le décor (elle le dessine à l'écran). L'initialisation du programme est alors terminée. La dernière instruction de cette partie est un branchement vers la partie principale.

La partie principale contient le code qui dirige le déroulement du programme. Dans notre exemple, le programme est une animation. Pour que le code de cette partie ne soit pas trop long, certains morceaux en ont été séparés et ont été placés dans des sous-routines : ce sont les blocs d'instructions qui sont répétées plusieurs fois. Ainsi, le code de la partie principale contient de nombreux appels à ces sous-routines.

Voyons à présent le rôle des sous-routines. La première sous-routine sert à temporiser l'animation. Elle est simplement constituée de deux boucles **FOR…NEXT** imbriquées et ne contenant aucune instruction. En effet, l'interprétation de ces boucles demande un certain temps à QBASIC, même si elles ne contiennent aucune instruction. Cela permet de créer un délai dans l'exécution. Ainsi, en insérant des appels à la sous-routine **Temporise** dans la partie principale, on peut ralentir l'animation (qui serait beaucoup trop rapide autrement).

La durée du délai créé dépend du nombre de boucles effectuées par les constructions **FOR…NEXT**. Par défaut, la première construction boucle 10 fois, et la seconde 2000 fois : 20000 boucles sont effectuées en tout. Si l'on fixe la valeur du paramètre ***fin*** de la seconde boucle à 4000 au lieu de 2000, le délai créé sera deux fois plus grand. L'animation sera donc deux fois plus lente.

Les sous-routines **DessVoitD**, **DessVoitB** et **DessVoitBD** servent à dessiner la voiture dans différentes positions : dirigée vers la droite, vers le bas, ou vers le coin en inférieur droit. Vous avez pu remarquer que toutes les coordonnées utilisées dans les instructions de dessin de ces sous-routines sont relatives à deux variables : **X** et **Y**. En effet, ces deux variables représentent la position de la voiture : elles contiennent les coordonnées de son coin supérieur gauche. Par exemple, si **X** est égal à 320 et que **Y** est égal à 240, la voiture sera dessinée avec son coin supérieur gauche situé au centre de l'écran. Ainsi, en modifiant la valeur de **X** et de **Y**, il est possible de dessiner la voiture à plusieurs endroits différents, en particulier sur la route.

Prenons un exemple : nous voulons que la voiture se déplace de la gauche vers la droite. Nous devons donc d'abord la dessiner à sa position de départ, sur la gauche de l'écran, tournée dans la direction où elle va se diriger : vers la droite. Il faut pour cela commencer par régler ses coordonnées. Pour que la voiture soit dessinée sur la gauche de l'écran, **X** doit être égal à 0. Pour choisir à quelle hauteur elle sera dessinée, il faut régler la valeur de **Y**. Celle-ci doit être comprise entre 0 et 463 (car la voiture mesure 17 pixels de hauteur : 480 - 17 = 463). Puis, pour dessiner la voiture tournée vers la droite, il faut appeler la sous-routine **DessVoitD**.

Ensuite, afin de la faire se déplacer vers la droite, nous devons augmenter la valeur de **X** régulièrement (par exemple de 4 en 4), et redessiner la voiture à chaque fois en appelant **DessVoitD**. Cependant, ce n'est pas suffisant. Si l'on se contente de faire cela, la voiture sera redessinée à chaque fois par-dessus le dessin précédent, et il restera toujours une partie du dessin précédent à l'écran. Pour éviter cela, nous devons effacer le dessin précédent à chaque fois avant de dessiner la voiture à nouveau.

Si la voiture se déplaçait sur un fond noir, il suffirait pour l'effacer d'afficher un rectangle noir par-dessus le dessin. Cependant, ici la voiture se déplace sur un décor : une route. Si l'on affichait un rectangle noir, une partie du décor serait effacée en même temps que la voiture. La solution est de redessiner le décor par-dessus la voiture. Pour cela, nous devons copier une partie du décor avant de dessiner la voiture, à l'aide de **GET**, puis le coller par-dessus la voiture pour l'effacer, grâce à **PUT**.

Ainsi, nous avons réussi à créer un effet de déplacement. Il reste encore un dernier défaut à corriger : la vitesse. En effet, sans temporisation, la voiture se déplace beaucoup trop vite. Nous devons donc appeler la sous-routine **Temporise** entre chaque déplacement de la voiture pour ralentir la vitesse de l'animation.

Les sous-routines **AffVoitD**, **AffVoitB** et **AffVoitBD** regroupent les étapes nécessaires à la création de l'effet de déplacement. Elles commencent par copier la zone rectangulaire du décor sur laquelle va être dessinée la voiture. Ensuite elles dessinent la voiture, puis elles appellent la sous-routine de temporisation. Enfin, elles effacent la voiture en collant le décor par-dessus.

Revenons au déroulement de l'animation. La première ligne du code de la partie principale contient une boucle **FOR…NEXT** qui appelle plusieurs fois de suite la sous-routine de temporisation. Cela permet de créer un délai assez long, durant lequel l'exécution est mise en attente.

Vous avez pu vous rendre compte que QBASIC met un certain temps pour changer de mode d'affichage de l'écran, après avoir interprété une instruction **SCREEN**. Cependant, il n'attend pas que le changement soit terminé pour interpréter les instructions qui suivent **SCREEN**. Ainsi, celles-ci sont exécutées avant que les pixels de l'écran soient affichés. Mettre l'exécution en attente pendant quelques temps avant de démarrer l'animation permet d'éviter que celle-ci commence avant que l'utilisateur puisse la voir s'afficher à l'écran.

Ensuite la valeur de **Y** est fixée à 217. Cela permet de dessiner la voiture dans la voie du bas de la route centrale. Une boucle **FOR…NEXT** permet de la faire se déplacer de la gauche de l'écran jusqu'au carrefour central en faisant évoluer **X** de 0 à 176, et en appelant **AffVoitD** à chaque tour. Pour toutes les lignes droites, c'est une boucle **FOR…NEXT** similaire qui fait se déplacer la voiture.

Pour les tournants, **X** et **Y** sont modifiés en même temps. La voiture est d'abord dessinée dirigée à la fois vers le bas et la droite, puis elle est dessinée dirigée vers la droite ou vers le bas, selon la direction qu'elle prend. Remarquez que la manière dont **X** et **Y** sont modifiés dans le virage courbe n'est pas calculée : c'est par tâtonnements que je l'ai déterminée.

Dans le programme, la voiture ne peut être dessinée que dirigée dans trois directions différentes, ce qui limite les possibilités de parcours que celle-ci peut effectuer. A vous d'écrire à partir des sous-routines existantes celles qui permettront de dessiner la voiture dans toutes les autres directions. Vous pourrez alors lui faire suivre le parcours qui vous plaira. Attention cependant, la programmation d'un parcours peut se révéler fastidieuse.

Pour dessiner la voiture dirigée vers la gauche, reprenez la sous-routine qui permet de la dessiner dirigée vers la droite (**DessVoitD**), et remplacez tous les **+** par des **–** dans les abscisses. Par exemple, **X + 3** deviendra **X – 3**. Attention, **X** et **Y** seront alors les coordonnées du coin supérieur droit de la voiture, et non plus du coin supérieur gauche. Pour dessiner la voiture dirigée vers le haut, reprenez la sous-routine **DessVoitB**, et remplacez tous les **+** par des **–** dans les ordonnées. **Y + 3** deviendra donc **Y – 3**. De plus, **X** et **Y** seront alors les coordonnées du coin inférieur gauche de la voiture.

Conclusion

La programmation de l'interface visuelle graphique en QBASIC n'a plus de secret pour vous. Vous êtes capables à présent de dessiner des graphismes à l'écran, notamment des décors, et aussi de créer des animations. Vous avez cependant immanquablement pu constater à quel point cela peut être long et même parfois fastidieux. Mais en vous armant de patience, vous finirez par créer vos propres animations et vous serez contents de vos résultats. Encore et toujours, c'est à force d'essais que vous y parviendrez. Et n'hésitez pas à relire les passages de ces deux derniers chapitres qui restent encore un peu obscurs pour vous.

Chapitre VII :
Quelques commandes utiles

Vous connaissez maintenant presque tout ce que vous avez à connaître sur le langage BASIC. Cependant, il vous reste encore à apprendre comment utiliser certaines commandes qui sont indispensables au développement de nombreux programmes, et plus particulièrement des jeux.

A) Nombres aléatoires (instruction `RANDOMIZE TIMER` et fonction `RND`)

Tout au long du cours, de nombreuses instructions utilisant des données de type nombre ont été étudiées : l'instruction **`PRINT`**, par exemple, ou encore les instructions de dessin dont nous avons parlé au cinquième chapitre.

Les nombres utilisés par ces instructions peuvent être soit des données constantes, soit des variables. La valeur d'une donnée constante est fixée par le programmeur, et reste invariable durant toute l'exécution du programme. Au contraire, la valeur d'une variable peut changer. Pendant l'exécution, on peut affecter à une variable un nombre constant, le résultat d'un calcul, la valeur d'une autre variable, ou bien un nombre entré au clavier par l'utilisateur.

Dans tous les cas que l'on vient de citer, ces nombres ne sont jamais choisis par l'ordinateur. Pourtant, il peut s'avérer utile de demander à l'ordinateur de choisir un nombre au hasard. QBASIC dispose pour cela d'un « générateur de nombres aléatoires ». Un « nombre aléatoire » est un nombre compris entre 0 et 1 (pouvant être égal à 0 mais pas à 1), choisi au hasard par l'ordinateur.

Avant de pouvoir être utilisé, le générateur de nombres aléatoires doit être initialisé à l'aide de la commande **`RANDOMIZE TIMER`**. Après cela, des nombres aléatoires peuvent être générés grâce au mot-clé **`RND`**. Celui-ci peut être utilisé dans n'importe quelle instruction qui manipule des nombres. Voici un exemple d'utilisation de nombre aléatoire :

```
RANDOMIZE TIMER
PRINT "Voici un nombre aléatoire :"; RND
```

Les nombres aléatoires générés par **`RND`** ne sont compris qu'entre 0 et 1, et cependant on a généralement besoin d'en obtenir dans une plus grande plage de valeurs. Pour cela, il faut multiplier **`RND`** par la valeur maximale que l'on veut que le nombre puisse avoir. Par exemple, si l'on multiplie **`RND`** par 5, le nombre obtenu sera compris entre 0 et 5. Il pourra être égal à 0 mais pas à 5.

De la même manière, si l'on veut que le nombre obtenu soit toujours supérieur à 0, il suffit d'ajouter à **`RND`** la valeur minimale que l'on veut que le nombre puisse avoir.

```
RANDOMIZE TIMER
PRINT "Voici un nombre aléatoire :"; RND * 5 + 5
```

Dans l'exemple ci-dessus, le nombre obtenu est compris entre 5 et 10. Il peut être égal à 5 mais pas à 10.

Souvent, vous aurez besoin d'utiliser des nombres entiers. Or, les nombres générés par **RND** sont décimaux. Il faut donc les tronquer, à l'aide de la fonction **FIX**. Par exemple, ces lignes simulent un lancé de dé :

```
RANDOMIZE TIMER
PRINT "Le dé est tombé sur le numéro"; FIX(RND * 6 + 1)
```

Voici un autre exemple, dans lequel des nombres aléatoires sont utilisés par des instructions de dessin :

```
SCREEN 12
RANDOMIZE TIMER
X1 = FIX(RND * 640)
Y1 = FIX(RND * 480)
FOR I = 1 to 100
    X2 = FIX(RND * 640)
    Y2 = FIX(RND * 480)
    Couleur = FIX(RND * 16)
    LINE (X1, Y1)-(X2, Y2), Couleur
    X1 = X2
    Y1 = Y2
NEXT
```

La boucle **FOR…NEXT** permet de dessiner 100 lignes avec des coordonnées et une couleur aléatoires. Les coordonnées de la première extrémité de chaque ligne dessinée sont les mêmes que celles de la seconde extrémité de la ligne précédente. Ainsi, toutes les lignes se suivent.

B) Récupération de caractères entrés au clavier (fonction INKEY$ et instruction SLEEP)

Jusqu'à présent, nous n'avons utilisé qu'un seul moyen pour récupérer des données (chaînes de caractères ou nombres) entrées au clavier par l'utilisateur : la commande **INPUT**. Son inconvénient est qu'elle affiche systématiquement à l'écran tout ce que l'utilisateur saisit au clavier. De plus, elle nécessite que celui-ci appuie sur la touche *[Entrée]* quand il a fini de taper les données, afin de les stocker.

Il existe un second moyen pour savoir ce que l'utilisateur entre au clavier : c'est de récupérer un par un les caractères qu'il saisit. La fonction **INKEY$** le permet. Elle renvoie un caractère (ou bien une combinaison de deux caractères).

Reprenons l'exemple d'utilisation du mode graphique. Lancez le programme, et essayez d'appuyer sur une touche pendant l'animation. Vous pouvez constater que cela n'a aucun effet sur son déroulement. En fait, le caractère correspondant à la touche sur laquelle vous avez appuyé est simplement stocké par l'ordinateur, en attendant d'être récupéré par la fonction **INKEY$**. Pour comprendre son fonctionnement, ajoutez cette instruction à la fin de la partie principale du code :

```
PRINT INKEY$
```

Notez qu'elle est équivalente à ces deux instructions :

```
Carac$ = INKEY$
PRINT Carac$
```

Maintenant, relancez le programme et appuyez sur la touche *[A]* pendant l'animation. A la fin de l'exécution, le caractère « *a* » est affiché à l'écran. Recommencez en appuyant cette fois simultanément sur les touches *[Maj]* et *[B]*. Le caractère « *B* » est affiché à l'écran après l'animation. Vous pouvez réessayer en entrant n'importe quel caractère, celui-ci sera affiché à l'écran à la fin du programme.

Lorsque vous entrez plusieurs caractères pendant l'animation, c'est toujours le premier que vous avez saisi qui est affiché à l'écran, à la fin du programme. Les suivants sont stockés par QBASIC après le premier, et peuvent être récupérés de la même manière que celui-ci. Pour vous en rendre compte par vous-même, réécrivez une seconde fois l'instruction que vous avez ajoutée à la fin du programme, et relancez-le. Les deux premiers caractères que vous avez entrés pendant l'animation sont affichés à l'écran.

Notez que QBASIC ne peut pas stocker plus de quinze caractères en attendant qu'ils soient récupérés par la fonction **INKEY$**. Si vous appuyez sur une touche pendant l'animation alors que quinze caractères ont été stockés, l'ordinateur émet un bip sonore. A chaque récupération d'un caractère par **INKEY$**, celui-ci est enlevé de la mémoire, et un nouveau caractère peut être stocké à sa place.

Généralement, la fonction **INKEY$** est utilisée à un moment précis d'un programme pour savoir si l'utilisateur a appuyé sur une touche particulière. Par exemple, on peut vérifier si l'utilisateur a appuyé sur la touche *[A]*. Voici ce qu'il faut écrire :

```
IF INKEY$ = "a" THEN …
```

Lorsque l'utilisateur n'appuie sur aucune touche, **INKEY$** renvoie une « chaîne nulle ». La chaîne nulle est représentée par des guillemets entre lesquels rien n'est écrit :

```
IF INKEY$ = "" THEN …
```

Ainsi, on peut utiliser une construction **DO…LOOP** pour mettre le programme en attente, c'est-à-dire d'attendre que l'utilisateur appuie sur une touche avant de poursuivre l'exécution. Il suffit de la faire boucler jusqu'à ce que **INKEY$** renvoie autre chose qu'une chaîne nulle :

```
DO: LOOP UNTIL INKEY$ <> ""
```

Certaines touches et combinaisons de touches correspondent à des caractères que l'on ne peut pas entrer au clavier. C'est le cas par exemple de la touche *[Echap]* : le caractère qui lui correspond est une flèche dirigée vers la gauche. Pour vérifier si l'utilisateur a appuyé sur une de ces touches, on doit utiliser la fonction **CHR$**, dont nous avons déjà parlé. Il faut lui préciser le code ASCII du caractère correspondant à la touche ou combinaison de touches.

Pour vérifier si l'utilisateur a appuyé sur la touche *[Echap]* (et si c'est le cas terminer l'exécution du programme), voici ce qu'il faut écrire :

```
IF INKEY$ = CHR$(27) THEN GOTO Fin
```

Voici un tableau qui précise les codes ASCII de ces touches et combinaisons de touches :

Touche(s)	Code ASCII	Touche(s)	Code ASCII
[Control] + [A]	1	*[Control] + [Entrée]*	10
[Control] + …	…	*[Entrée]*	13
[Control] + [Z]	26	*[Echap]*	27
[Retour arrière]	8	*[Control] + [Retour arrière]*	127
[Tab]	9		

Il existe encore une autre catégories de touches et combinaisons de touches qui, celles-ci, ne correspondent à aucun caractère. Elles ne sont pas désignées par un code ASCII, mais par un « code clavier ». Lorsque l'utilisateur appuie sur une de ces touches, **INKEY$** renvoie deux caractères. Le premier caractère est toujours le « caractère nul » (celui dont le code ASCII est 0). Le code ASCII du second caractère est le code clavier de la touche ou combinaison de touches. Par exemple, voici comment vérifier si l'utilisateur a appuyé sur la touche de direction vers la gauche (c'est utile dans un jeu où l'utilisateur doit pouvoir faire se déplacer un objet à l'écran) :

```
IF INKEY$ = CHR$(0) + CHR$(75) THEN …
```

Voici le tableau des codes clavier :

Touche(s)	Code clavier	Touche(s)	Code clavier
[Alt] + [Q]	16	*[Début]*	71
[Alt] + [W]	17	*[Haut]*	72
[Alt] + [E]	18	*[Page précédente]*	73
[Alt] + [R]	19	*[Gauche]*	75
[Alt] + [T]	20	*[Droite]*	77
[Alt] + [Y]	21	*[Fin]*	79
[Alt] + [U]	22	*[Bas]*	80
[Alt] + [I]	23	*[Page suivante]*	81
[Alt] + [O]	24	*[Inser]*	82
[Alt] + [P]	25	*[Suppr]*	83

[Alt] + [A]	30	*[Control] + [F1]*	94
[Alt] + [S]	31	*[Control] + …*	…
[Alt] + [D]	32	*[Control] + [F10]*	103
[Alt] + [F]	33	*[Alt] + [F1]*	104
[Alt] + [G]	34	*[Alt] + …*	…
[Alt] + [H]	35	*[Alt] + [F10]*	113
[Alt] + [J]	36	*[Control] + [Gauche]*	115
[Alt] + [K]	37	*[Control] + [Droite]*	116
[Alt] + [L]	38	*[Control] + [Fin]*	117
[Alt] + [Z]	44	*[Control] + [Page suivante]*	118
[Alt] + [X]	45	*[Control] + [Début]*	119
[Alt] + [C]	46	*[Control] + [Page précédente]*	132
[Alt] + [V]	47	*[F11]*	133
[Alt] + [B]	48	*[F12]*	134
[Alt] + [N]	49	*[Control] + [F11]*	137
[Alt] + [M]	50	*[Control] + [F12]*	138
[F1]	59		
…	…		
[F10]	68		

Nous avons vu qu'il est possible de mettre en attente un programme, grâce à une construction **DO…LOOP** qui boucle jusqu'à ce que **INKEY$** renvoie autre chose qu'une chaîne nulle. L'instruction **SLEEP** permet de faire la même chose :

```
SLEEP
```

L'avantage d'utiliser **SLEEP** est qu'il n'y a que cinq lettres à taper. Nous verrons plus tard l'inconvénient que présente cette commande par rapport à la boucle **DO…LOOP**.

C) Gestion du temps (mots-clés TIME$ et DATE$, fonction TIMER, et instructions ON TIMER , TIMER et SLEEP)

1. Date et heure

Le code BASIC possède des commandes qui permettent de gérer l'heure et la date. Ce sont **TIME$** (pour l'heure) et **DATE$** (pour la date). Il s'agit de mots-clés particuliers, puisqu'ils sont à la fois des fonctions et des instructions (ce qui signifie que l'on peut aussi bien consulter l'heure ou la date que les modifier à partir d'un programme QBASIC). En fait, ils sont utilisés comme des variables. Considérons donc que ce sont des variables créées automatiquement par QBASIC.

Les valeurs renvoyées par **TIME$** et **DATE$** sont des chaînes de caractères. La chaîne qui correspond à l'heure est de la forme « hh:mm:ss ». Par exemple, **TIME$** renvoie la chaîne « 18:15:30 » pour 18 heures 15 minutes et 30 secondes. La chaîne qui correspond à la date est de la forme « mm-jj-aaaa ». Ainsi, **DATE$** renvoie la chaîne « 11-10-2000 » pour 10 novembre 2000 (le jour et le mois sont inversés, comme dans le système anglais).

On peut afficher l'heure et la date à l'écran en introduisant directement **TIME$** et **DATE$** dans une instruction **PRINT** :

```
PRINT "Heure : "; TIME$
PRINT "Date : "; DATE$
```

Pour régler l'heure ou la date, il suffit d'affecter une chaîne de la forme correspondante à **TIME$** ou **DATE$**, comme si c'étaient des variables :

```
TIME$ = "18:15:30"
DATE$ = "11-10-2000"
```

Si vous essayez d'affecter à **TIME$** ou **DATE$** une chaîne de forme incorrecte, QBASIC générera une erreur.

TIME$** et **DATE$ renvoient des chaînes de caractères, et non des nombres, et ce sont aussi des chaînes qu'il faut leur affecter pour régler l'heure ou la date. Ainsi, il n'est pas possible d'effectuer un calcul à partir des valeurs qu'elles renvoient, ou bien de leur affecter une valeur qui a été calculée.

Il existe un second moyen de récupérer l'heure dans un programme, mais cette fois-ci sous forme numérique. La fonction à utiliser est **TIMER**. Elle renvoie l'heure exprimée en secondes (soit le nombre de secondes écoulées depuis minuit). Par exemple, s'il est minuit passé de 5 minutes et 3 secondes, **TIMER** renvoie 303 (5 x 60 + 3 = 303).

Le nombre renvoyé par **TIMER** n'est pas un entier, mais un nombre décimal dont la précision est de sept chiffres après la virgule. Cela signifie que l'heure est exprimée au dix millionième de seconde près. Cependant, ce n'est qu'un semblant de précision, car la marge d'erreur est d'environ 5 centièmes de secondes. Cela veut dire que la véritable heure peut être jusqu'à 5 centièmes de secondes en avance ou en retard par rapport à l'heure renvoyée par **TIMER**. Il ne faut donc s'en tenir qu'à une précision d'un chiffre après la virgule (au dixième près).

L'avantage de récupérer l'heure sous forme numérique est que l'on peut effectuer des calculs avec. Ainsi, on peut séparer par le calcul les heures, les minutes, les secondes, et les dixièmes de seconde, en ne tenant pas compte des six derniers chiffres après la virgule.

Prenons un exemple : **TIMER** renvoie 59472,8203125. On commence par isoler le premier chiffre après la virgule (8), qui correspond aux dixièmes de secondes. Puis on tronque le nombre renvoyé par **TIMER** (on ne garde que sa partie entière) : on obtient 59472. Ensuite, on divise ce nombre par 60 (59472 / 60 = 991 reste 12). Le reste de cette division correspond aux secondes. On divise alors le quotient par 60 (991 / 60 = 16 reste 31). Le reste de cette nouvelle division correspond aux minutes, et le quotient aux heures. Ainsi, le nombre 59472,8203125, une fois qu'il est décomposé, devient 16 heures 31 minutes 12 secondes et 8 dixièmes de seconde.

Pour isoler le chiffre des dixièmes de secondes et pour tronquer le nombre renvoyé par **TIMER**, il faut utiliser **FIX**. Puis, pour effectuer les divisions, il faut utiliser les opérateurs de division entière (\), et de reste de division (**MOD**).

Voici donc le code à taper pour obtenir séparément les heures, les minutes, les secondes et les dixièmes de seconde en décomposant le nombre renvoyé par **TIMER** :

```
Temps = TIMER
TpsEnt = FIX(Temps)                'On ne garde que la partie entière
Dixiemes = FIX((Temps - TpsEnt) * 10)   'On isole le premier chiffre
                                                    'après la virgule
Secondes = TpsEnt MOD 60    'On récupère le reste de la division n°1
TpsEnt = TpsEnt \ 60      'On récupère le quotient de la division n°1
Minutes = TpsEnt MOD 60     'On récupère le reste de la division n°2
Heures = TpsEnt \ 60      'On récupère le quotient de la division n°2
```

2. Chronométrage et temporisation

Un autre des intérêts de la fonction **TIMER** est qu'elle peut être utilisée comme chronomètre, c'est-à-dire qu'elle permet de calculer le temps qui s'est écoulé entre deux instants précis du programme. En voici un exemple :

```
Temps1 = TIMER
SLEEP
Temps2 = TIMER
PRINT "Il s'est écoulé"; FIX(Temps2 - Temps1); "secondes."
```

La première instruction récupère le nombre de secondes renvoyé par **TIMER** (au début du programme). La seconde instruction met le programme en attente, puis la troisième récupère à nouveau le nombre de secondes renvoyé par **TIMER** (après que l'utilisateur a appuyé sur une touche). Enfin, la dernière instruction affiche la différence entre les deux valeurs renvoyées par **TIMER**, soit le nombre de secondes écoulées entre le début du programme et le moment où l'utilisateur a appuyé sur une touche.

Nous l'avons vu dans l'exemple d'utilisation du mode graphique : il est parfois nécessaire de temporiser un programme. La méthode que l'on a utilisée dans cet exemple consiste à insérer des boucles **FOR…NEXT** qui ne contiennent aucune instruction. L'inconvénient de ce procédé est qu'il est très difficile en l'utilisant de régler précisément la durée du délai créé. De plus, cette durée varie en fonction de la vitesse de l'ordinateur. On peut remédier à cela, en se servant des commandes qui permettent de temporiser précisément un programme, et de sorte que les délais créés soient identiques sur tous les ordinateurs.

La première de ces commandes est **SLEEP**. Nous en avons déjà parlé dans la section précédente. Utilisée sans paramètre, elle permet de mettre un programme en attente jusqu'à ce que l'utilisateur appuie sur une touche. Mais elle sert aussi à mettre un programme en attente pendant une durée déterminée. Son rôle est alors similaire à celui des boucles **FOR…NEXT** imbriquées. Pour l'utiliser ainsi, il faut faire suivre le mot-clé **SLEEP** du nombre de secondes durant lesquelles le programme doit être mis en attente. Par exemple, cette ligne permet de suspendre l'exécution d'un programme pendant 10 secondes :

```
SLEEP 10
```

Pour tester cette commande, vous pouvez remplacer les boucles **FOR…NEXT** de la sous-routine **Temporise** par une instruction **SLEEP**, dans l'exemple d'utilisation du mode graphique.

L'inconvénient de **SLEEP** par rapport aux boucles **FOR…NEXT** imbriquées est que cette commande ne permet de régler la durée du délai qu'elle crée que par paliers d'une seconde. On ne peut donc pas avec **SLEEP** mettre un programme en attente pendant une demi seconde par exemple.

La seconde méthode dont dispose QBASIC est la programmation « d'événements » déclenchés par la minuterie. Un événement est un appel à une sous-routine, effectué automatiquement par QBASIC. Ainsi, programmer un événement déclenché par la minuterie revient à demander à QBASIC d'appeler une sous-routine à intervalles de temps réguliers. Les commandes qui servent à gérer ces événements sont **ON TIMER** et **TIMER**. La première permet de programmer un événement. La seconde permet d'activer, de désactiver ou de suspendre le déclenchement de l'événement.

Voici comment doit être complétée l'instruction **ON TIMER** :

ON TIMER(*intervalle*) GOSUB *sous-routine*

intervalle est le nombre de secondes qui s'écoulent entre chaque déclenchement de l'événement (entre chaque appel de la sous-routine). ***sous-routine*** est le nom de la sous-routine à appeler. Notez qu'un seul événement peut être programmé à la fois.

Voici comment s'utilise **TIMER** :

TIMER ON | OFF | STOP

TIMER ON active le déclenchement de l'événement programmé par **ON TIMER**. **TIMER OFF** le désactive, arrête la minuterie et la remet à zéro. **TIMER STOP** suspend le déclenchement de l'événement jusqu'à la prochaine instruction **TIMER ON**. A la différence de **TIMER OFF**, **TIMER STOP** n'arrête pas la minuterie et ne la remet pas à zéro.

Regardez cet exemple :

```
Secondes = 0
ON TIMER(1) GOSUB AffSec
TIMER ON
DO: LOOP UNTIL INKEY$ <> ""
GOTO Fin

AffSec:
Secondes = Secondes + 1
LOCATE 1, 1
PRINT "Nombre de secondes écoulées depuis le début :"; Secondes
RETURN

Fin:
```

Les trois premières lignes de code initialisent le programme. La valeur de **Secondes**, le compteur des secondes écoulées, est fixée à 0. Ensuite, un événement est programmé puis activé : toutes les secondes, QBASIC doit appeler la sous-routine **AffSec**, qui ajoute 1 au compteur **Secondes** et affiche sa valeur à l'écran.

La quatrième ligne contient une boucle **DO…LOOP** qui attend que l'utilisateur appuie sur une touche. En même temps que celle-ci est interprétée, l'événement programmé est déclenché toutes les secondes. Lorsque l'utilisateur appuie sur une touche, l'exécution passe à la cinquième ligne, qui quitte le programme.

En bref, ce programme affiche régulièrement à l'écran le nombre de secondes écoulées depuis son lancement, jusqu'à ce que l'utilisateur appuie sur une touche.

Dans cet exemple, une boucle **DO…LOOP** utilisant **INKEY$** a été préférée à la commande **SLEEP**, pour attendre que l'utilisateur appuie sur une touche. En effet, **SLEEP** permet de suspendre l'exécution d'un programme, mais le déclenchement d'un événement entraîne automatiquement la reprise de l'exécution. Ainsi, si l'on avait utilisé la commande **SLEEP** le programme se serait automatiquement arrêté au bout d'une seconde (c'est-à-dire lors du premier déclenchement de l'événement), même sans que l'utilisateur ait appuyé sur une touche.

Enfin, il existe une dernière méthode pour temporiser un programme, qui est plus précise que les deux que nous venons de voir. Il s'agit de se servir du chronométrage avec **TIMER**. Prenons un exemple :

```
Temps = TIMER
DO: LOOP UNTIL TIMER - Temps >= 1.5
```

La première instruction sert à récupérer le nombre de secondes renvoyé par **TIMER**. Puis la boucle **DO…LOOP** récupère sans cesse la valeur renvoyée par **TIMER** jusqu'à ce que la différence entre celle-ci et la première valeur récupérée soit à peu près égale à une seconde et demi (il est très peu probable qu'elle soit exactement égale à une seconde et demi, d'où l'utilisation de l'opérateur **>=** plutôt que **=**). Autrement dit, ces lignes mettent l'exécution du programme en attente pendant une seconde et demi.

Cependant, cette méthode possède un défaut qu'il faut corriger : elle ne prévoit pas le cas où l'on changerait de jour pendant la temporisation. Supposons que nous voulons suspendre l'exécution pendant cinq secondes et demi. Juste avant la suspension, il est 23 heures 59 minutes et 58 secondes. La valeur renvoyée par **TIMER** est donc sensiblement égale à 86 398. Deux secondes plus tard, il est minuit. Le compteur des secondes de l'ordinateur est remis à zéro, et **TIMER** renvoie 0. La différence entre la valeur renvoyée par **TIMER** et la première récupérée devient alors négative (- 86 398). Cette différence se rapprochera de 0 au fur et à mesure que le temps passera, mais elle ne sera jamais supérieure ou égale à 5,5. Par conséquent la construction **DO…LOOP** bouclera à l'infini.

Pour éviter que cela se produise, voici comment il faut compléter les instructions précédentes :

```
Temps = TIMER
DO: IF TIMER - Temps < 0 THEN Temps = Temps - 86400
LOOP UNTIL TIMER - Temps >= 5.5
```

L'instruction **IF…THEN** qui est dans la boucle **DO…LOOP** vérifie si la différence est négative (cela se produit lorsqu'on change de jour). Si c'est le cas, on soustrait 86400 à la variable **Temps** (c'est le nombre de secondes qui s'écoulent dans une journée). **Temps** a donc une valeur négative, qui correspond au nombre de secondes qui se sont écoulées entre le déclenchement de la temporisation et le passage au jour suivant. Soustraire **Temps** à la valeur renvoyée par **TIMER** revient donc à ajouter le nombre de secondes écoulées avant le passage au jour suivant au nombre de secondes qui se sont écoulées après. Le problème est ainsi réglé.

D) Sonorisation (instructions SOUND et PLAY)

Lorsque l'on veut créer des jeux, en plus de savoir dessiner des graphismes et temporiser le programme, il est bien utile de pouvoir rajouter des sons. QBASIC possède des commandes qui jouent du son, cependant, il ne permet pas d'utiliser la carte son. Vous pourrez donc simplement produire quelques sons dans une gamme réduite. Ils seront joués par les hauts-parleurs internes de l'ordinateur.

Il existe deux façons de faire jouer un son à l'ordinateur. La première est d'utiliser la commande **SOUND**. Chaque instruction **SOUND** joue un son unique. Il faut lui passer deux arguments, la fréquence du son et sa durée :

SOUND ***fréquence, durée***

fréquence est en hertz. Elle doit être comprise entre 37 Hz (le plus grave) et 32767 Hz (le plus aigu). ***durée*** est comptée en tops d'horloge : il y a 18,2 tops d'horloge par seconde. Elle doit être comprise entre 0 et 65535 tops (ce peut être un nombre décimal).

Vous pouvez jouer plusieurs sons à la suite en écrivant plusieurs instructions **SOUND**, mais jamais deux sons en même temps. Voici un exemple d'utilisation de **SOUND** :

```
FOR I = 440 TO 1000 STEP 5
    SOUND I, I / 1000
NEXT
```

Pour trouver la fréquence qui correspond au son que vous voulez entendre, vous devez essayer de nombreuses fois des fréquences différentes (certains calculs permettraient d'éviter cela, mais ils ne vous seront pas utiles). C'est un inconvénient de **SOUND**.

L'autre inconvénient de la commande **SOUND** est qu'elle ne permet pas de jouer des mélodies pendant que d'autres opérations sont effectuées. Lorsque QBASIC rencontre une instruction **SOUND**, il fait jouer le son au haut-parleur et interprète la ligne de code suivante. S'il s'agit aussi d'une instruction **SOUND**, il enregistre en mémoire la fréquence et la durée passées en argument. Le second son est ainsi mis en attente, et ne sera joué que lorsque le haut-parleur aura fini de jouer le premier. QBASIC lit donc la ligne suivante. S'il s'agit encore d'une instruction **SOUND**, QBASIC suspend l'exécution, car il n'est pas possible de mettre plus d'un son en attente. Lorsque l'ordinateur a fini de jouer le premier son, et qu'il joue le second, le troisième est mis en attente, et l'exécution reprend. Ainsi, il n'est pas possible de jouer plus de deux sons à la suite sans que l'exécution soit stoppée.

Il n'est donc intéressant d'utiliser **SOUND** que pour jouer un ou deux sons, c'est-à-dire pour émettre des bips sonores, qui pourront tenir lieu d'effets sonores dans un jeu par exemple.

Pour jouer des mélodies, il existe une autre commande : **PLAY**. Son utilisation est complètement différente de celle de **SOUND**. La gamme de sons est plus limitée, puisque **PLAY** ne donne pas le choix de la fréquence. Cette commande ne permet de jouer des sons qu'à des fréquences précises : ce sont alors des notes. Contrairement à **SOUND**, une seule instruction **PLAY** permet de jouer de nombreuses notes à la suite. Il n'y qu'un seul argument à passer à **PLAY** :

PLAY *mélodie$*

mélodie$ est une chaîne de caractères commandes, qui précise à **PLAY** la liste des notes à jouer, ainsi que leur longueur, à quelle octave elles doivent êtres jouées, et le tempo de la musique. Certains des caractères commandes doivent être suivis d'un nombre.

Voici les caractères commandes qui peuvent être utilisés dans la chaîne ***mélodie$*** :

Commande	Description
O	Spécifie l'octave (0 à 6)
< ou **>**	Passe à l'octave inférieure ou supérieure
C, **D**, **E**, **F**, **G**, **A**, **B**	Joue une note dans l'octave en cours
N	Joue une note sur toute la gamme (0 à 84)
# ou **+**	Dièse (pour la note précédente)
–	Bémol (pour la note précédente)
L	Fixe la durée des notes (1 à 64). 1 correspond à une ronde, 2 correspond à une blanche…
ML	Indique que les notes sont jouées à pleine durée
MN	Indique que les notes sont jouées aux $^{7}/_{8}$ de leur durée
MS	Indique que les notes sont jouées aux $^{3}/_{4}$ de leur durée
.	Indique que la note précédente est jouée aux $^{3}/_{2}$ de sa durée
T	Indique le nombre de noires par minute (32 à 255)
P	Fait une pause correspondant à un nombre spécifié de noires (1 à 64)
MF	Indique que la musique doit être jouée en premier plan
MB	Indique que la musique doit être jouée en arrière-plan

```
PLAY "MB"             'La musique est jouée en arrière-plan
PLAY "O2"             'Les notes sont jouées dans la deuxième octave
PLAY "L4"             'Les notes sont des noires
PLAY "T240"           'Tempo : 240 noires par minute
PLAY "C#D.EF.GA-"     'Joue quelques notes
```

Ces lignes sont équivalentes à celle-ci (on a tout regroupé) :

```
PLAY "MBO2L4T240C#D.EF.GA-"
```

Notez aussi que vous n'êtes pas obligé d'utiliser des majuscules. En effet, QBASIC ne fait pas la distinction entre majuscules et minuscules.

Voici deux exemples de mélodies (tirés de l'aide de QBASIC) :

```
Melodie$ = "MBO3L8ED+ED+EO2BO3DCL2O2A"
PLAY Melodie$
SLEEP
Melodie$ = "MBT180O2P2P8L8GGGL2E-P24P8L8FFFL2D"
PLAY Melodie$
```

Dans ces exemples, les mélodies sont jouées en arrière-plan. Cela signifie que QBASIC n'interrompt pas l'exécution : d'autres opérations peuvent être effectuées en même temps. Lorsque les mélodies sont jouées en premier plan, l'exécution est suspendue jusqu'à ce que QBASIC ait fini de jouer la dernière note.

Cependant, il n'est pas possible de jouer de longues mélodies en arrière-plan. Comme **SOUND**, **PLAY** met en attente les notes de la liste pendant que les précédentes sont jouées. Et le nombre de notes qu'il peut garder en attente est limité à dix-huit. Il est possible de savoir combien de notes restent en attente grâce à la fonction **PLAY** :

```
Melodie$ = "MBT180O2P2P8L8GGGL2E-P24P8L8FFFL2D"
PLAY Melodie$
DO: LOOP WHILE PLAY(0) > 0
PRINT "Fini."
```

Vous avez pu remarquer, quand la mélodie est jouée en arrière-plan, que l'exécution se termine avant que QBASIC ait fini de la jouer. Ici, ce n'est pas le cas. Les deux premières lignes donnent une mélodie à jouer à QBASIC. La troisième ligne est une construction **DO…LOOP** qui boucle tant qu'il reste des notes en attente (c'est-à-dire tant que **PLAY** renvoie un nombre non nul). Ainsi, la quatrième ligne n'est exécutée que lorsque la dernière note de la mélodie est en train d'être jouée, et seulement à ce moment l'exécution est terminée. Notez que le même résultat aurait pu être obtenu en jouant la mélodie en premier plan, la boucle **DO…LOOP** serait alors inutile. Ici, elle sert juste à montrer comment on utilise la fonction **PLAY**.

Vous avez remarqué que la fonction **PLAY** est suivi du nombre 0 entre parenthèses. En effet, il faut toujours lui passer un nombre en argument. Mais peu importe ce nombre, cela n'influera pas sur la valeur retournée par **PLAY**. A quoi sert-il donc ? Même l'aide de QBASIC ne le précise pas…

Conclusion

Toutes les commandes qui peuvent vous être utiles vous ont été présentées. Vous savez maintenant comment gérer le temps et le son dans un programme, mais aussi comment vérifier sur quelles touches appuie l'utilisateur (afin, par exemple, de lui permettre de contrôler un personnage dans un jeu). Et vous êtes capables de générer des nombres aléatoires (vous verrez qu'il est souvent utile d'obtenir des nombres au hasard).

Conclusion générale

Voilà, vous avez appris tout ce que vous aviez à savoir sur la programmation en QBASIC. Ce cours n'a certes pas présenté de manière exhaustive les commandes que ce logiciel met à votre disposition, mais ce serait une perte de temps de le faire. En effet, le langage BASIC n'est pas idéal pour concevoir des programmes complexes, et si tel est votre désir, il vaut mieux que vous appreniez un autre langage plus performant (mais à la fois plus compliqué). L'utilisation de QBASIC n'aura donc été pour vous qu'une introduction à la programmation, mais elle aura été plus agréable et plus facile que si vous aviez choisi un autre langage pour débuter.

En revanche, si vous ne souhaitez programmer que de petites applications pour le plaisir, vous pouvez très bien continuer à utiliser QBASIC, car la simplicité d'utilisation du langage BASIC vous est certainement préférable à la complexité plus prononcée des autres langages tels que le Pascal, et surtout le C++. Dans ce cas, vous pouvez continuer à approfondir vos connaissances en langage BASIC, en consultant régulièrement l'aide de QBASIC pour apprendre de nouvelles commandes. Vous aurez d'ailleurs l'occasion de vous rendre compte que l'explication de certaines commandes a été simplifiée dans ce cours, pour les besoins de l'apprentissage.

En attendant, lancez le programme « *VAISSEAU.BAS* ». C'est un jeu de vaisseau que j'ai programmé à titre d'exemple final. Toutes les commandes et notions qu'il utilise ont été vues dans ce cours (à l'exception des variables tableau). Cela vous permettra de voir le genre de programmes qui peuvent être créés avec QBASIC.

A présent, à vos clavier, et à vos programmes ! Vous venez d'entrer dans l'univers passionnant de la programmation, et peut-être un jour deviendrez-vous un professionnel en la matière.

Philippe-Henry Marcy
Pour plus d'informations sur la programmation, allez sur mon site :
http://membres.lycos.fr/firesnk/

Ou envoyez-moi un e-mail à :
fsnkemail@caramail.com