ECOLE NATIONNALE SUPÉRIEURE DE L'AÉRONAUTIQUE
ET DE L'ESPACE

Bureau d'Etude langage C : Réalisation d'un Casse-briques

Emmanuel Branlard - Dimitri Sluys

Le 14 Mars 2007

Table des matières

Liste des tableaux

Table des figures

Première partie
Rapport

1 Description et fonctionnalités du jeu

1.1 Mode d'emploi et recommendations

Ce jeux fonctionne sous Windows ou Linux et nécessite Java 1.4 au minimum.

Le principe est de détruire toutes les briques au départ fixes, à l'aide d'une balle qu'on envoie grâce à un curseur. Bougez le curseur avec les touches directionnelles gauche et droite du clavier et lancez la balle grâce à la touche espace. Pour quitter une partie en cours, appuyez sur la touche Q. Au cours du jeu, l'utilisateur peut faire pause en laissant appuyée la touche P. Repérez bien les diférentes briques magiques décrites sur l'écran d'accueil, certaines peuvent vous jouer des tours.

1.2 Cahiers des charges et option de jeu

1.2.1 Espace de travail et options de jeu

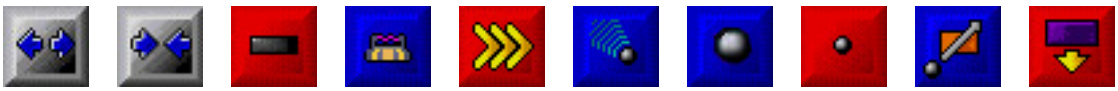
Pour notre casse brique, nous avons choisi d'utiliser des briques de tailles fixes et disposées selon un quadrillage précis. L'espace de travail est donc divisé en 200 cases (20x20). Ceci correspond à l'interface d'un casse brique classique, ce qui répondra donc aux attentes des férus de ce jeu. Toutefois, nous avons choisi de réaliser un casse brique avec des options avancées. La balle rebondit selon les lois de réflexion de Descartes sur les 3 murs formant l'enceinte de jeu (deux murs latéraux et un mur en face du curseur). Elle rebondit différemment sur le curseur, selon la position de l'impact. La partie s'arrête quand le joueur n'a plus de vie, vies qu'il perd en laissant passer la balle derrière la ligne de mouvement du curseur. S'il parvient à passer le niveau en détruisant les briques, il débutera un nouveau niveau, libéré de toute condition. Concernant les briques, il en existe deux types mais pour chaque destruction, le joueur gagne des points et augmente son score en les détruisant. On trouvera les briques magiques et les briques non magiques. Les dernières disparraissent une fois touchées sans aucun autre effet, les premières disparaissent au profit d'un bonus. Le bonus tombe sur une ligne verticale, pour en profiter, il faut l'attraper grâce au curseur. Nous avons élaboré plusieurs types de bonus. Nous avons instauré pour ces différents bonus des effets positifs et des effets négatifs. Nous avons décidé de donner des points de score à chaque fois que nous attrapons un bonus ou un malus. Les malus rapportent plus de points, ce qui pimente la partie

1.2.2 Calcul du score

40 Points par brique cassée. 100 Points par objet magique absorbé par le curseur, lorsqu'il s'agit d'un effet facilitant le jeu (ralentissement de la balle, augmentation du curseur, vie supplémentaire, etc... 700 Points par objets magiques absorbés d'influence négative (accélération de la balle, rétrécissement de celle-ci ou du curseur, etc).

1.2.3 Les bonus d'état

Les bonus d'état influencent la partie en cours soit au niveau du curseur, soit au niveau de la balle :



- "sizeup" augmente la taille du curseur, les balles se ratrappent plus facilement
- "sizedown" diminue la taille du curseur, le rattrapage des balles est plus difficile.
- "shrink" réduit directement la taille du curseur à la plus petite taille possible. Le curseur possède 4 tailles différentes, il commence par défaut à la deuxième plus petite.
- "catchme" chaque fois que la balle entre en contact avec le curseur, ce dernier la fixe à lui, le joueur est libre de la relancer quand il veut (après avoir bougé le curseur s'il le souhaite) par pression sur la touche "espace" ;
- "faster" augmente la vitesse de la balle.
- "slower" diminue la vitesse de la balle.
- "bigger" augmente la taille de la balle.
- "smaller" diminue la taille de la balle.
- "go trough" permet à la balle de ne pas rebondir sur les briques, mais de les détruire quand même, elle détruit donc toutes les briques situées sur sa trajectoire.
- "fall" fait descendre les briques d'une hauteur de briques. Ainsi, la distance entre le premier front de briques et le curseur diminue.

1.2.4 Les bonus de jeu

Les bonus de jeu ne modifient pas les paramètres dynamiques du jeu :

- "levelup" amène directement au niveau suivant, même si toutes les briques ne sont pas détruites. Cependant, le joueur doit se rappeler qu'il n'aura alors pas de bonus de score des briques non détruites.
- "lifeup" donne une vie en plus au joueur.
- " death" fait perdre une vie au joueur.
  Le nombre de vies varie est fixé a 3 à l'instant initial, toutefois, ceci peut se changer facilement.

1.2.5 Restrictions

Compte tenu de la faible portée de ce programme, il ne nous est pas apparu nécessaire d'effectuer un interface utilisateur très évolué. En effet, cette phase de création d'interface graphique destinée à l'utilisateur(Graphical User Interface - GUI) est très éloignée de la phase developpement lors de la conception d'un logiciel. Nous nous sommes résumé à l'aspect graphique de la zone de jeu, mais nous n'avons pas souhaité réaliser de menu pour lancer le jeu, sauver les meilleurs scores, lancer un niveau en particulier. Ces options n'ayant d'intérêt que pour une commercialisation du jeu.

1.2.6 Personnalisation

Notre programme utilise des images et des sons personnalisés différents de ceux qui nous ont été fournit, à l'exception de la balle et du curseur qui ont servi de base avant d'être modifiés.

2 Description du programme

2.1 Prise en main du problème

La réalisation d'un jeu nécessite plus de méthode et d'organisation que les algorithmes que l'on a pu réaliser au cours de l'année. C'est pourquoi nous avons veillé à ne pas tomber dans le piège consistant à tapper du code dès le début sans une réflexion poussée préalable. En effet, cette façon de programmer est très dangereuse car le code est modifié régulièrement à chaque ajout de nouvelles fonctionnalités. Il en devient peu lisible, peu souple, complexe, et surtout mal structuré. Ainsi, notre première approche du problème a été d'effectuer le cahier des charges, résumé à la section 1.2. Ainsi, nous avions entièrement déterminé les fonctionnalités de notre programme et c'est alors que nous avons pu commencer à réfléchir à l'organisation de celui-ci pour qu'il puisse remplir toutes nos exigences. Dans le but d'avoir un code le plus souple possible, nous avons choisi de paramétrer toutes nos valeurs afin de pouvoir les faire varier à souhait, au cours du jeu pour augmenter les modes de jeu, ou au cours de la réalisation pour faciliter la maîtrise de chacunes des valeurs. Une fois fixées les variables et les structures de données qui seront utilisées, nous nous sommes penchés sur la structure même du code, afin de le rendre le plus fonctionnel possible. Durant cette étape, nous nous sommes contentés de définir les protypes de chaque fonction : nom de la fonction, arguments, contenu et type de la valeur retournée. Dès lors, l'étape de programmation pouvait commencer, et la répartition des tâches fut on ne peut plus facile. Chacun de nous pouvait programmer indépendemmanent de l'autre en respectant les conditions que nous nous étions fixées.

2.2 Choix des variables

Dans le bu es structures vient s'ajouter la structure s_point, permettant à une fonction de retourner un couple d'entier si nécessaire. Nous précisons au passage que les noms des types et variables ont été choisis afin d'être le plus explicites possibles. Ils est courant en informatique de commencer le nom de variable par une majuscule, ou une minuscule suivi d'un underscore, correspondant au type de la variable. Par exemple, une liste de briques sera notées $l\_briques$ ou Lbriques. Nos structures suivent elles aussi cette notation, et commencent donc toutes par $s\_$. Les pointeurs quant à eux sont désignés par la lettre p accolée en début du nom de variable.

```
struct s_liste { int id; int x;int y;int type; struct s_liste * suivant;};
typedef struct s_liste * liste;

struct s_point{int x; int y;};
typedef struct s_point point;

struct s_balle{int x; int y; int a; int b;int pas;int taille;};

struct s_curseur{int taille; char tid; int x;int y;int pas;};

struct s_param{char quitter;
               char perdu;
               char gagne;
               char engagement;
               char catchme;
               char gosrou;
               char nb_vies;
               char sleep;
               char niv;
               int score;};
```

Les différents champs de ces structures seront décrits das la troisième section de ce document. Nous pouvons toutefois justifier le choix de celles-ci. Le type liste était imposé pour les briques, nous avons donc utilisé celui-là. Il permet d'utiliser la place mémoire réservée au nombre exact de briques présentes dans un niveau, et non au nombre maximal de briques (ici 400). Il parait logique et pratique de rassembler au sein d'un même élément, toutes les caractéristiques d'un objet. Or ces divers éléments n'étant pas tous du même type, les tableaux ou les listes ne conviennent pas. Nous avons donc utilisé des structures permettant d'enregistrer plusieurs variabels de types différentes.

2.2.1 Les variables globales

Les sources du programmes contiennent dans un seul fichiers, tous les paramètres globaux nécessaires. Voici la liste de ceux-ci :

```
/*dimensions de la zone de jeu*/
#define L 960
#define H 800
#define Bord_gauche 30
#define Bord_droit 30
#define Bord_haut 20

/* nombre de niveaux*/
#define N_MAX_NIV 3

/*dimensions des briques*/
#define Lb 45
#define Hb 25

/*dimensions curseur*/
#define Lc_INIT 128
#define Hc 16
#define PASc_INIT 12 /* pas de deplacement du curseur*/

/*dimensions bille initiale*/
#define Dballe 15

/* paramètres */
#define NB_VIES 3
#define SLEEP 5  /*raffraichissement*/
#define V_DESC 4 /*vitesse descente des briques magiques*/
#define V_INIT 6  /*vitesse balle initiale*/
#define A_INIT 1  /*composante en x du vecteur vitesse initial*/
#define B_INIT -3 /*composante en y du vecteur vitesse initial*/
```

Bien sûr ceci n'est pas optimal, car il s'agit d'une facilité pour le programmateur, mais non pour l'utilisateur, compte tenu du fait que ces données sont remplacées directement par le compilateur et donc ne sont plus modifiables après cette compilation. L'idée consisterait à intégrer tous ces paramètres dans un fichier de configuration, que le programme chargerait à chaque utilisation et placerait par exemple dans une structure. Ainsi, même après la compilation, il aurait été possible de changer la taille de la fenêtre, la position du curseur initial, etc... Toutefois, compte tenu de la portée de ce programme, nous avons jugé qu'il était inutile que l'utilisateur ait accès à ces paramètres.

2.3 Structure générale du programme

Il apparaît important de noter que le principe de base que nous avons adopté est celui d'un code constitué de multiples fonctions, simples. Ainsi, tout problème est décomposé en sous problèmes qui font appels à des fonctions élémentaires. C'est en quelque sorte le principe du diviser pour règner("divide to conquer"), cher aux informaticiens.

2.3.1 Le découpage en plusieurs fichiers

Nous avons utilisés plusieurs sous fichiers afin de clarifier et d'espacer le code, mais aussi d'accélérer la compilation. La liste des fichiers utilisés est la suivante :

- main.c : contient le moteur du jeu et le *main*, qui lance le jeu et le ferme
- dynamique.c : contient les fonctions ayant un aspect dynamique : deplacement d'objet, gestion des rebonds des briques magiques et des touches.
- listes.c : contient toutes les fonctions manipulant la liste de briques
- affichage.c : contient toutes les fonctions nécessaire pour afficher ou effacer des objets (curseur, balle, etc..
- init.c : contient toutes les fonctions d'initialisation et de finalisation
- auxiliaires.c : contient toutes les fonctions auxiliaires. Ce sont souvent de petites fonctions effectuant une opération en particulier.
- niveau.c : fichier responsable du chargement et de l'affichage d'un niveau
- evenements.c : gère les évènements tels que le fait de perdre ou de quitter
- prototypes.h : contient les prototypes de toutes les fonctions ainsi que les définitions numériques des constantes.
- graphic.c, key.h, graphic.h : fichiers fournis.

2.3.2 Relations entre les fichiers, structure du programme

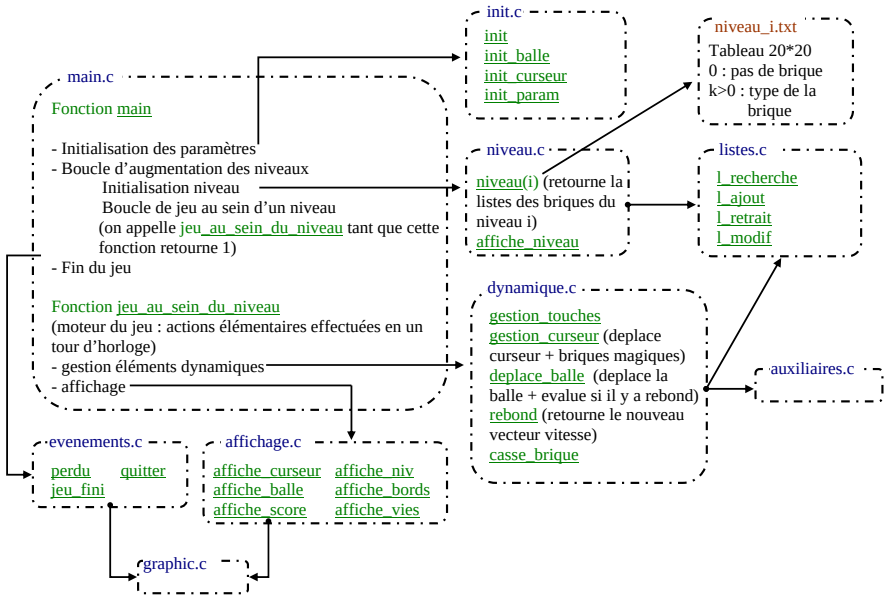


FIG. 1 – Organigramme des fonctions principales et des relations entre les fichiers

2.3.3 Prototypes des fonctions contenues dans chaque fichiers :

/*main.c*/

```
int jeu_au_sein_du_niveau(liste * l,struct s_balle * pballe,
                          struct s_curseur * pcurseur,struct s_param * pparam);
void main();
```

/* init.c : initialisation et fin*/

```
void init();
void init_balle(struct s_balle * pballe,int x,int y);
void init_curseur(struct s_curseur * pcurseur,int x_curseur,int y_curseur, int pas_curseur);
void init_param(struct s_param * pparam,int niv,int n,int score);
void fin();
```

/* dynamique.c */

```
void casse_brique(liste *pl,int id);
void rebond(struct s_balle * pballe, int lim, char type);
int gere_rebond(liste l,struct s_balle * pballe,int id2, int direction,
                int type,struct s_param * pparam);
void gestion_touches(struct s_curseur * pcurseur,struct s_param * pparam);
void gestion_curseur(struct s_curseur * pcurseurn,liste *pl,
                     struct s_param * pparam,struct s_balle * pballe);
void deplace_balle(struct s_balle * pballe,struct s_curseur * pcurseur,
                   liste ** ppl,struct s_param * pparam);
```

/* affichage.c */

```
void affiche_score(struct s_param * pparam);
void affiche_vies(struct s_param * pparam);
void affiche_bords();
void affiche_curseur(struct s_curseur * pcurseur);
void affiche_balle(struct s_balle * pballe);
void affiche_niv(struct s_param * pparam);
void efface_balle(struct s_balle * pballe);
void efface();
```

/* listes.c */

```
int   l_recherche(liste l, int x);
liste l_retrait(liste l, int x);
void  l_affichage(liste l);
void  l_ajout(liste * ppl, int id, int x,int y, int type);
void  l_modif(liste *pl, int id, int x, int y, int type);
```

/* niveau.c*/

```
void affiche_niveau(liste l);
liste niveau(int i);
```

/* evenements.c*/

```
void perdu(struct s_param * pparam);
void quitter();
void jeu_fini();
```

/* auxiliaires.c */

```
int cadrant (int a, int b);
point idtopos (int id);
int postoid (int x, int y);
void normalise_balle(struct s_balle * pballe, int a,int b);
void ajuste_taille(struct s_curseur* pcurseur);
```

2.4 Complexité et taille mémoire des algorithmes

2.4.1 Optimisation de la place mémoire

Afin d'éviter de prendre trop de place en mémoire, nous avons fait attention à déclarer un minimum de variables locales, et d'utiliser à chaque fois le type le plus adapté à ces variables. Les entiers de moins de 255 ont été codés avec un type char, par exemple. De même, la totalité des arguments sont passés par adresse et donc modifiés en place, afin d'éviter toute copie inutile en mémoire.

2.4.2 Problème dû à l'utilisation d'images

Toutefois, lors du developpement du jeu, et notamment dans la phase finale de test, il est apparu que le jeu était gourmand en mémoire et ceci provoquait un problème au niveau des fonctions java, et arrètait le jeu au bout d'un temps assez long. La cause de ce problème de la mémoire est l'affichage d'images à chaque cycle du moteur de jeu où les images sont chargées à chaque fois. Ceci nous a guidé dans une première optimisation de l'affichage : afficher les bords, les vies, le niveau, uniquement lorsque cela est nécessaire. Le gain en mémoire fut alors notable mais semble t-il insuffisant.

2.4.3 Solutions proposées

Nous avons proposé différentes solutions :

Première solution : Le problème précédent n'intervient pas, ou alors à très faible échelle lorsqu'on n'utilise pas des images mais des rectangles de couleurs à l'aide de la fonction fillRect. C'est pourquoi nous avons réalisé un second programme utilisant cette fonction d'affichage.

Seconde solution : Vu que c'est l'affichage des briques à chaque appel qui augmente la taille mémoire, il s'agit d'afficher les briques uniquement si nécéssaire. Ainsi, le chargement du niveau ne se fait que lorsque le niveau débute, et à chaque fois que le joueur perd une vie. Lorsque la balle rebondit sur une brique, on efface cette brique à l'aide de clear_rect. La balle quand à elle est effacée en dessinant un disque noir à l'emplacement de la balle, juste avant le déplacement de celle-ci. Le curseur, le score sont effacés par des clearRect, à chaque boucle du moteur de jeu. Ceci fonctionne parfaitement et utilise une mémoire quasi constante au cours du jeu. Le seul problème réside dans l'affichage des briques magiques. En effet celles-ci sont censées descendre vers le bas de l'écran, et elles ne doivent pas effacer les briques non magiques situées sous elles. Nous avons alors dégagé quatres solutions à ce second problème :

- annuler les options magiques
- faire descendre les options magiques sur le coté, et donc, ces briques n'effacent pas les briques non-magiques dans leur descente. Cette solution est très efficace et fournit de très bon résultats mémoire.
- afficher toutes les briques, tant qu'il existe des briques magiques en train de descendre. Cette option est intermédiaire et fournit des résultats mitigés. En effet, si le jeu est fluide et rapide lorsqu'aucune brique magique ne descend, il est au contraire ralenti dès q'une brique descend.

– gérer la descente des briques magiques : regarder devant quelles briques elle est, afficher la brique magique, et redessiner à l'affichage suivant les briques devant lesquelles elle était passée. Ceci fournirait une solution convenable.

Nous avons réaliser des programmes dans chacun de ces cas (a l'exception du dernier) pour pouvoir les comparer et permettre un jeu prolongé.

2.4.4 Bilan

Nous avons effectué de nombreux efforts pour optimiser la place mémoire que prendrait notre programme. Ceci nous fut bénéfique, toutefois, nous n'avions pas envisager que la fonction DrawImage chargerait à chaque appel les images. Il suffirait de modifer cette fonction pour que les images ne soient chargées qu'une seule fois, au démarrage (comme les sons).

3 Principes de fonctionnement des différents composants

3.1 Paramètres de jeu

Les paramètres de jeu contiennent toutes les informations relatives au déroulement de la partie en cours.

3.1.1 Rappel de la structure

```
struct s_param{char quitter;
               char perdu;
               char gagne;
               char engagement;
               char catchme;
               char gosrou;
               char nb_vies;
               char niv;
               int score;};
```

3.1.2 Brève description des paramètres aux noms non-canoniques

- engagement : vaut 1 si la balle doit rester collée au curseur jusqu'à ce que l'utilisateur appuie sur espace
- catchme : vaut 1 s'il y a engagement à chaque reception de la balle
- gosrou : vaut 1 si la balle passe à travers les briques sans rebondir

3.2 Gestion des niveaux

Comme nous l'avons décidé dans le cahier des charges, nous avons choisi d'utiliser des briques de tailles fixes et disposées selon un quadrillage précis. L'espace de travail est donc diviser en 200 cases (20x20). A chaque case nous attribuons un identifiant (id), qui est un entier compris entre 0 et 399. Ainsi, chaque brique dispose d'un identifiant unique, et connaissant l'identifiant de la brique, nous connaissons sa position (coordonnées du coin supérieur gauche). Il est souvent très important d'utiliser ce type d'identifiant, comme par exemple pour identifier de façon unique les lignes d'une base de données. Les niveaux sont inscrits dans des fichiers intitulés niveau_i et situés dans le repertoire niveaux/. Un niveau est un tableau de 20 par 20, où chaque case correspond à une brique. Cette case contient un char qui a pour valeur le type d'une brique. Dans le fichier, un changement de case se repère par une tabulation.

Les niveaux se créent ou se modifient très facilement à l'aide d'un tableur par exemple. Le programme charge le niveau lorsque le joueur arrive à celui-ci. Chaque niveau est donc chargé qu'une seule fois. Le chargement du niveau consiste à lire le fichier niveau_i, et de créer la liste l_briques contenant les briques magiques ou non qui sont présentes dans le niveau.

0	Absence de brique
1	brique bleue
2	brique turquoise
3	brique rouge
4	brique verte
5	vie supplémentaire (lifeup)
6	taille curseur ++ (sizeup)
7	taille curseur – (sizedown)
8	garde la balle (catchme)
9	passe au niveau suivant (levelup)
10	descend les briques d'une hauteur de brique (fall)
11	accelère la balle (faster)
12	passe à travers (gosrou pour go through)
13	ralentit la balle (slower)
14	ramène le curseur à la taille minimale (shrink)
15	perd une vie (death)
16	taille de balle – (smaller)
17	taille de balle ++ (bigger)

TAB. 1 – Description des types de briques

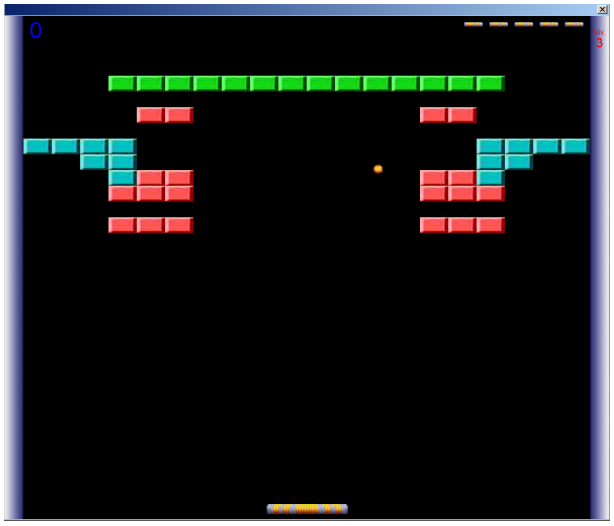


FIG. 2 – Exemple de niveau

3.3 Gestion des briques magiques

3.3.1 Rappel de la structure

```
struct s_liste { int id; int x;int y;int type; struct s_liste * suivant;};
typedef struct s_liste * liste;
```

3.3.2 Les briques magiques : des briques normales avec des coordonnées

Les briques magiques s'affichent comme les briques non magiques. L'affichage change que lorsque celles-ci sont percutées par la balle. Alors chaque brique magique est représentée par une image qui lui est propre. Comme nous venons de le voir, la fonction d'affichage des briques a juste à parcourir la liste l_briques pour afficher les briques à la fois magiques et non magiques.

Lorsqu'une brique magique est activée, on modifie la liste l_briques pour entrer les informations de positions relatives à la brique magique. Ces informations permettront de faire descendre la briques à chaque boucle du moteur de jeu. Ceci justifie l'existence des champs x et y du type struct s_liste. Ils ne sont pas utiles aux briques non magiques, c'est pourquoi nous avons longuement hésité à créer deux listes disctinctes : celles des briques magiques et celles des briques normales. Toutefois, le code nous a paru plus clair avec une seule liste.

3.4 Gestion de la balle

3.4.1 Rappel de la structure

```
struct s_balle{int x; int y; int a; int b;int pas;int taille;};
```

3.4.2 Déplacement de la balle

Toutes les informations relatives à la balle sont contenues au sein de la variable balle de type struct s_balle. Ses paramètres sont sa position et sa vitesse. Nous avons décomposé le vecteur vitesse en un vecteur unitaire de composantes (a,b) et en un pas qui représente la norme de celui-ci en pixels. Cette représentation offre une grande souplesse d'utilisation. Par exemple, pour accélérer ou ralentir la balle, il suffit d'augmenter le pas. Par ailleurs, l'expression des fonctions de rebonds s'exprime fort facilement à l'aide de ces informations.

3.5 Gestion des rebonds

3.5.1 Rebond sur le curseur

Le rebond sur le curseur s'effectue de manière très simple grâce aux structures mise en place. Nous n'avons pas souhaité séparer le curseur en plusieurs zones pour définir le rebond. Au contraire, nous avons utilisée une fonction de rebond qui est continue.

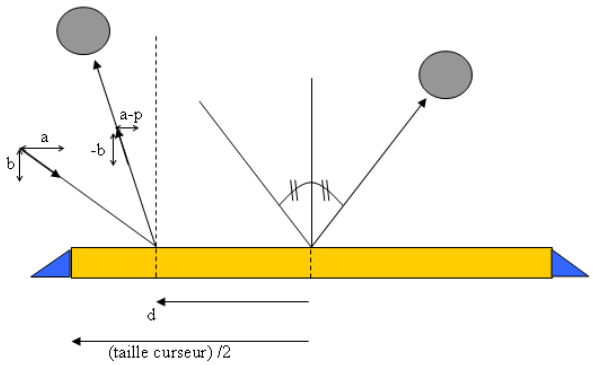


FIG. 3 – Rebond sur le curseur - notations

Nous avons souhaité utiliser un curseur ayant un certain coefficient de frottement. Ainsi les rebonds autoriseront une jouabilité fort agréable. Le vecteur vitesse, de norme pas, et dirigé selon le vecteur unitaire (a, b), devient après rebond sur le curseur :

$$\begin{pmatrix} a + \frac{pas.(d - t/2)}{\frac{t}{2}} \\ -b \end{pmatrix}$$

On notera que le vecteur n'est pas unitaire. On le norme à l'aide la fonction normalise_vitesse. Par ailleurs, les rebonds sur la partie "bleue" du curseur, sont effectués sans effets. La balle est envoyé suivant l'angle fixe de 45 degré ou 135 suivant que l'on est à gauche ou a droite. Ceci

permet de relancer facilement la balle avec un angle suffisant pour sortir des cas où la balle a une trajectoire quasi horizontale. Le comportement du curseur est donc le suivant :

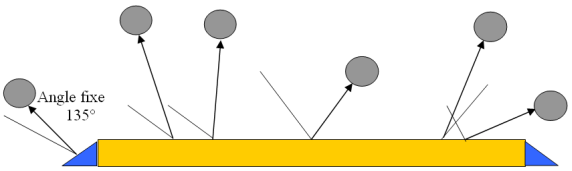


FIG. 4 – Différents rebonds sur le curseur

La fonction rebond n'étant pas continue, nous ne pouvons pas représenter l'infinité de cas possibles. Toutefois, les cas ci-dessus permettent d'intuiter le rebond de la balle sur le curseur ;

3.5.2 Rebond sur les briques

Rebond dans le cas d'un balle ponctuelle La fonction de destruction avec rebond pour les briques était une des plus difficiles à réaliser. En effet, à première vue le rebond est le même que sur un mur. Dans beaucoup de cas, oui. Mais rappellons nous que la mouvement de la balle est un phénomène discret, et donc que sur un coup de malchance, la balle pourrait traverser une brique sans la détruire, car sa position initiale et sa position finale ne touchent pas la brique en question. Considérons le shémas suivant :

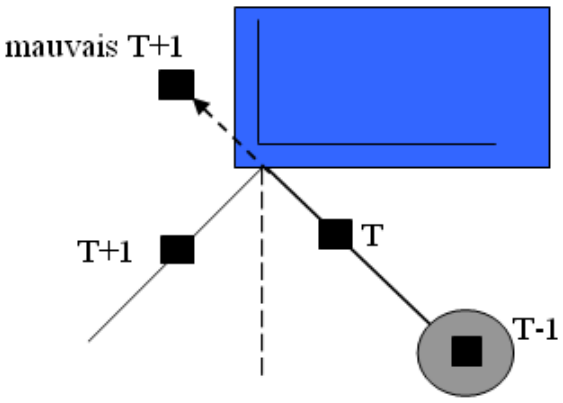


FIG. 5 – Mise en évidence du caractère discret de la propagation de la balle

On voit bien que la balle peut avoir une trajectoire erronée ; il faut bien vérifier si la balle ne traverse pas une brique, et le cas échéant, s'assurer du bon rebond de la balle sur cette dernière. Ces cas furent fastidieux à mettre en oeuvre pour s'assurer du bon rebond de la balle, de la non destruction de briques qui ne s'avèrent en réalité non atteintes etc... Nous avons finalement traité tous les cas possibles avec des tests.

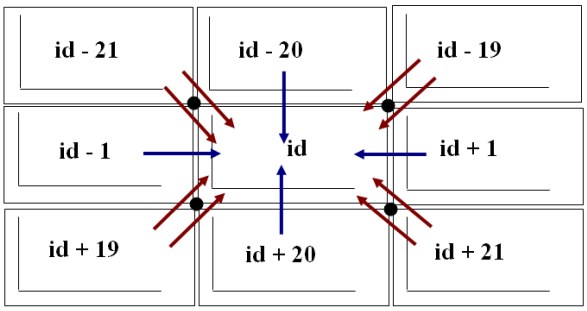


FIG. 6 – Les différents cas de rebonds possibles

Légende :
Une flèche à pour origine la balle à l'instant t, et pour arrivée la balle à l'instant t+1.
Points noirs : coins de la brique id
Flèches bleues : rebond sur la brique id (si elle existe) ne présentant pas de difficulté
Flèches rouges : situations où la balle a pu percutée une brique au milieu de son déplacement discret.

Etude d'un cas :
A l'instant t la balle est sur la case id-21 à t+1 elle est sur la case id. Si elle passe au dessus du coin supérieur gauche de la brique id, elle percutera la brique id-20 à la gauche de celle-ci si elle existe. Si elle n'existe pas, elle percutera la brique id sur son bord supérieur.

Pour vérifier si la balle passe au dessus ou en dessous d'un coin, on effectue la projection orthogonale de ce point sur la droite dirigée par le vecteur vitesse. Suivant le signe du produit scalaire entre le vecteur vitesse et le vecteur (projeté du coin -> coin), nous pouvons conclure. Dans notre programme nous avons appelé *ps* la valeur de ce produit scalaire. ($ps = -b.(xb - x0) + a.(yb - y0)$ où (xb, yb) sont les coordonnées du coin de la brique et (a, b) les coordonnées de nortre vecteur vitesse unitaire)

Prise en compte de l'epaisseur de la balle et choix des paramètres pertinents Nous savons que la balle est repérée par les coordonnées de son point supérieur gauche. Or ce n'est pas forcément ce point qui va rebondir sur une brique. Ainsi, les "paramètres pertinents pour le rebond" ne sont pas les coordonnées de la balle mais les coordonnées du point de la balle qui a des chance de percuter une brique. Pour simplifier nous choisirons les quatre points cardinaux de la balle. Suivant le cadrant du diagramme de Fresnel dans lequel evolue le vecteur vitesse, nous choisissons le bon point cardinal. Cette action est effectuée par la fonction *cadrant* qui prend en argument le vecteur vitesse et retourne le type de cadrant. La figure suivante décrit les types de cadrants :

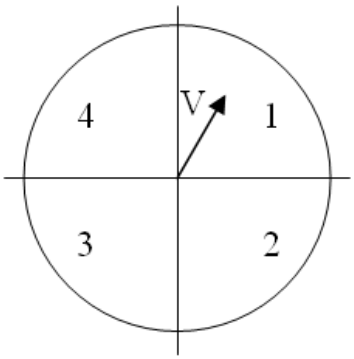


FIG. 7 – Les différents types de cadrant pour le vecteur vitesse

Conclusion

Perspectives d'amélioration du programme Au cours de ce document plusieurs propositions d'amélioration du programme ont été données, qu'il s'agisse d'options supplémentaires ou d'amélioration du code. Celles-ci n'ont pas été effectuées, soit par choix dès l'écriture du cahier des charges, soit par manque de temps. Comme nous l'avons vu dans la section 2.4 concernant la complexité, pour optimiser au mieux le programme tout en conservant un aspect graphique et une jouabilité avancée, la meilleur solution serait de modifier la fonction d'affichage des briques en chargeant une seule fois chaque image, afin de ne pas prendre trop de place mémoire. De plus, si on souhaite rendre entièrement paramétrable le jeu, il faudrait enregistrer celles-ci dans un fichier qui serait lu à chaque ouverture du programme. Enfin, en plus du jeu, peuvent être rajoutées les options suivantes : menu, liste des meilleurs score, choix du niveau, difficulté réglable...

Bilan Ce travail aura été une bonne expérience du travail en binôme au sein d'un Bureau d'Etude. Nous avons traité ce problème comme un véritable problème de conception d'un produit. Ainsi, la mise en place d'un cahier des charges a été une des priorités dans notre approche et nous somme satisfait d'être parvenu à le remplir dans son intégralité. Bien sûr, il s'agit d'un travail fort modeste face aux jeux que l'on trouve dans le commerce. Toutefois, travailler sur un tel problème a donné au projet une dimension ludique. L'intégralité des fonctionnalités apprises lors des cours de langage C nous a été utile pour ce Bureau d'Etude ainsi que les notions d'algorithmique qui nous ont été enseignées cette année. Enfin, cette expérience s'est déroulée sans aucune discorde au sein de notre binôme. Interessés par le problème, la réflexion suscitée par celui-ci n'en a été que plus agréable.

Deuxième partie

Annexes

Annexe A : Makefile

```
CC=gcc
CCOPTS=-g -Wall
PLATFORM=linux
JNI_INCL=-Iinclude -Iinclude/${PLATFORM}
JVM_CLIENT_PATH=/usr/java/jdk/jre/lib/i386/client

supanoid: main.o graphic.o niveau.o affichage.o dynamique.o listes.o
          evenements.o auxiliaires.o init.o
${CC} main.o graphic.o niveau.o affichage.o dynamique.o listes.o
      evenements.o auxiliaires.o init.o ${CCOPTS} -L${JVM_CLIENT_PATH}
       -ljvm -o supanoid

main.o: main.c
${CC} ${CCOPTS} -c main.c
niveau.o:niveau.c
${CC} ${CCOPTS} -c niveau.c
affichage.o:affichage.c
${CC} ${CCOPTS} -c affichage.c
dynamique.o: dynamique.c
${CC} ${CCOPTS} -c dynamique.c
listes.o:listes.c
${CC} ${CCOPTS} -c listes.c
evenements.o:evenements.c
${CC} ${CCOPTS} -c evenements.c
auxiliaires.o:auxiliaires.c
${CC} ${CCOPTS} -c auxiliaires.c
init.o:init.c
${CC} ${CCOPTS} -c init.c
graphic.o: graphic.c graphic.h key.h
${CC} ${CCOPTS} ${JNI_INCL} -c graphic.c

clean:
rm *.o supanoid
```

Annexe B : Fichiers .c et .h

main.c

```
#include "prototypes.h"


/*****************************************************************************/
/*  moteur du jeu  */
/*****************************************************************************/
/*retourne 1 si le jeu continue au sein du même niveau*/
/*retourne 0 si le niveau est terminé ou si game over ou si l'utilisateur veut quitter*/

/*****************************************************************************
La fonction qui gère tous les paramètres qui varient à chaque battement d'horloge,
c'est le programme qui modifie l'évolution dans un niveau
*****************************************************************************/

int jeu_au_sein_du_niveau(liste * pl,struct s_balle * pballe, struct s_curseur * pcurseur,struct s_param * ppara

    int id;

    efface();

    /* gestion des touches */
    gestion_touches(pcurseur,pparam);

    /* gestion du curseur et objets magiques */
    gestion_curseur(pcurseur,pl,pparam,pballe);

    /* gestion de la balle */
    if( (*pparam).engagement==0){/* rebond et casse_brique*/
      deplace_balle(pballe,pcurseur,&pl,pparam);
    }
    else{/* la balle reste collée au curseur*/
     (*pballe).x=(*pcurseur).x+(*pcurseur).taille/2;
     (*pballe).y=(*pcurseur).y-Dballe;
      normalise_balle(pballe,A_INIT,B_INIT);
    }
    if((*pparam).quitter || (*pparam).perdu || (*pparam).gagne){return 0;}
    else{
      /* on affiche les divers elements*/
      affiche_niveau(*pl);
      affiche_balle(pballe);
      affiche_curseur(pcurseur);
      affiche_score(pparam);
      paint();
      sleepAWhile(SLEEP);

      if(*pl==NULL){(*pparam).gagne=1; playSound("sons/gagne.wav");
                  sleepAWhile(1000);}

      return 1;
    }
}
/*********************************************
Le main appelle le moteur et gêre la boucle des niveaux
*********************************************/
int main(void)
{
/*curseur a l'instant initial*/
  const int Xc_INIT=floor(L/2-Lc_INIT/2 );
  const int Yc_INIT=floor(H-Hc-8);
  /*balle a l'instant initial*/
  int X_INIT=Xc_INIT+Lc_INIT/2;
```

```
  int Y_INIT=Yc_INIT-(Dballe);

  struct s_balle balle;
  struct s_curseur curseur;
  struct s_param param;

  liste l_briques;
  init_param(&param, 1,NB_VIES,0);

  init(); /* première fonction executée*/

  while(!param.quitter && param.niv<=N_MAX_NIV && (param).nb_vies>0 ){
  /* boucles des niveaux*/
    /* on initialise les composants du jeu : briques / curseur / balle*/
    if(!param.perdu){ l_briques=niveau(param.niv);   } /*chargement du niveau*/

    init_balle(&balle,X_INIT, Y_INIT);
    init_curseur(&curseur, Xc_INIT,Yc_INIT,PASc_INIT);
    init_param(&param, param.niv,param.nb_vies,param.score);

    affiche_bords();
    affiche_niv(&param);
    affiche_vies(&param);
    affiche_niveau(l_briques);

    while(jeu_au_sein_du_niveau(&l_briques,&balle,&curseur,&param)){    }
    if(param.gagne){
      param.niv++;
    printf("->niveau suivant !\n");
    }
    if(param.perdu){
     (param).nb_vies--;
      perdu(&param);
    }
  }
  printf("Sortie de la boucle des niveaux\n");
  if(param.gagne && param.niv==N_MAX_NIV+1){
      jeu_fini();
    }
  if(param.quitter){
     quitter();
  }

  fin(); /* dernière fonctions utilisées */
  return 1;
}
```

dynamique.c

```
#include "prototypes.h"


/*****************************************************************************/
/*  balle et curseur  */
/*****************************************************************************/


/*****************************************************************************
La fonction qui selon la commande entrée par l'utilisateur modifie l'espace de jeu
*****************************************************************************/
void gestion_touches(struct s_curseur * pcurseur,struct s_param * pparam){
    switch (getLastKeyPressed()) { // Recuperation de la touche pressee en cours
      case -1:                     // aucune touche pressÃ©e
         break;
      case VK_LEFT:
         ((*pcurseur).x)=(*pcurseur).x-(*pcurseur).pas; // le curseur va vers la gauche
         break;
      case VK_RIGHT:
         ((*pcurseur).x)=(*pcurseur).x+(*pcurseur).pas; // le curseur va vers la  droite
        break;
      case VK_SPACE:
          if( (*pparam).engagement==1){ (*pparam).engagement=0; } /*espace pour engager*/
        break;
      case VK_Q: (*pparam).quitter=1;
        break;
      case VK_P:
        while(getLastKeyPressed()==VK_P){}
        break;
      case VK_ESCAPE:
        (*pparam).quitter = 1;
        break;
    }
}

/*****************************************************************************
La fonction qui fait se mouvoir les briques magiques au sein du niveau
*****************************************************************************/
void descend_briques(liste *pl){
     while((*pl)!=NULL){
        if((*pl)->x==0){
            (*pl)->id=(*pl)->id+20;
        }
        pl=&((*pl)->suivant);
     }
}

/*****************************************************************************
La fonction qui gère non seulement l'évolution du curseur dans la zone de jeu,
mais aussi l'évolution des briques magiques, qui tombent verticalement.
 Cette fonction modifie les scores avec les briques magiques.
*****************************************************************************/
void gestion_curseur(struct s_curseur * pcurseur,liste *pl ,struct s_param * pparam,
                     struct s_balle * pballe){
     char fall=0;
     liste l=*pl;
    /* gestion evenements spéciaux */
     while(l!=NULL){
      if(l->x!=0){
        l_modif(pl,l->id,l->x,l->y+V_DESC,l->type);
        if(l->y+30>=(*pcurseur).y && (*pcurseur).x-32<=l->x && (*pcurseur).x+(*pcurseur).taille>=l->x){
          /* objet magique absorbé */
          *pl=l_retrait(*pl,l->id);
```

```
          switch(l->type){
             case 5 : (*pparam).nb_vies++;  pparam->score=pparam->score+100;break;
                       /*vie en plus */
             case 6 : (*pcurseur).tid++; ajuste_taille(pcurseur); pparam->score=pparam->score+100;
                      break;  /* curseur plus grand*/
             case 7 : (*pcurseur).tid--; ajuste_taille(pcurseur); pparam->score=pparam->score+700;
                      break;  /* curseur plus petit*/
             case 8 : (*pparam).catchme=1; pparam->score=pparam->score+100;break;
                      /* catch me */
             case 9 : (*pparam).gagne=1; pparam->score=pparam->score+100;break;
                       /* niveau suivant */
             case 10 : fall=1; pparam->score=pparam->score+700;break;
                        /* briques descendent */
             case 11 : (*pballe).pas=(*pballe).pas+3; (*pcurseur).pas=(*pcurseur).pas+3;
                       normalise_balle(pballe,(*pballe).a,(*pballe).b);
                       pparam->score=pparam->score+700;break;  /* accélération de la balle */
             case 13 : (*pballe).pas=(*pballe).pas-2; if((*pballe).pas<=0){(*pballe).pas=2;}
                       normalise_balle(pballe,(*pballe).a,(*pballe).b);
                       pparam->score=pparam->score+100; break;  /* ralentit */
             case 12 : (*pparam).gosrou=1; pparam->score=pparam->score+100; break;
                       /* passe a travers des briques */
             case 14 : (*pcurseur).tid=2; ajuste_taille(pcurseur);  pparam->score=pparam->score+700;
                       break;  /* taille mini du curseur*/
             case 15 :  (*pparam).perdu=1;  pparam->score=pparam->score+700;break;  /* perd une vie */
             case 16 :  (*pballe).taille=(*pballe).taille-4;  pparam->score=pparam->score+700;
                        break;  /* balle plus grande */
             case 17 :  (*pballe).taille=(*pballe).taille+4;  pparam->score=pparam->score+100;
                         break;  /* balle plus petite */
           }

           playSound("sons/bonus.wav");
         }
              if((l->y)>=H){*pl=l_retrait(*pl,l->id);} /*objet sorti de la fenêtre*/
       }
       l=l->suivant;
     }
     if(fall){descend_briques(pl); }

     /* eviter que le curseur sorte */
     if((*pcurseur).x<=Bord_gauche){(*pcurseur).x=Bord_gauche;}
     if((*pcurseur).x+(*pcurseur).taille>=L-Bord_droit){(*pcurseur).x=L-Bord_droit-(*pcurseur).taille;}
}


/*****************************************************************************
La fonction qui détruit une brique, vérifiant si elle doit ou non faire
apparaître une brique magique
*****************************************************************************/
void casse_brique(liste *pl,int id){
     point p;
     int type=l_recherche(*pl,id);
     p=idtopos(id);

     if(type<5 && type>0){/*brique normale*/
       *pl=l_retrait(*pl,id);
        playSound("sons/cassebrique.wav");
        }
     else{ if(type<=17 && type>0){
          l_modif(pl,id,p.x+10,p.y,type);
          playSound("sons/cassebrique.wav");
          }
     }
    clearRect(p.x,p.y,Lb,Hb);
```

```
}


/*****************************************************************************
fonction qui effectue le rebond selon le type (haut bas gauche droite)
*****************************************************************************/
void rebond(struct s_balle * pballe, int lim, char type){
/*pballe. type =1 haut et bas, type = 3 gauche, type = 4 droit*/
  switch(type){
     case 1 : (*pballe).b=-((*pballe).b); (*pballe).x=(*pballe).x+(*pballe).a; (*pballe).y=lim+1; break;
     case 2 : (*pballe).b=-((*pballe).b); (*pballe).x=(*pballe).x+(*pballe).a; (*pballe).y=lim-1; break;
     case 3 : (*pballe).a=-((*pballe).a); (*pballe).y=(*pballe).y+(*pballe).b; (*pballe).x=lim+1; break;
     case 4 : (*pballe).a=-((*pballe).a); (*pballe).y=(*pballe).y+(*pballe).b; (*pballe).x=lim-1; break;
  }
}

/*****************************************************************************
Cette fonction regarde si rebond sur brique il y a, et le cas echeant,
dans la bonne direction l'effectue
*****************************************************************************/
int gere_rebond(liste l,struct s_balle * pballe,int id2, int direction, int type,
                struct s_param * pparam){
     int x0=(*pballe).x;
     int y0=(*pballe).y;
     int xb,yb;
     point p;

     p=idtopos(id2);
     xb=p.x; yb=p.y;
     if(l_recherche(l,id2)){
       if(!(*pparam).gosrou){
          switch(direction){
              case 4 :/* rebond gd 4*/ rebond(pballe,xb-(pballe->taille),4);break;
              case 3 :/* rebond dg 3*/ rebond(pballe,xb+Lb,3);break;
              case 2 :/* rebond hb 2*/ rebond(pballe,yb-(pballe->taille),2);break;
              case 1 :/* rebond bh 1*/ rebond(pballe,yb+Hb,1);break;
          }
         /* printf("%d\n",type); */ return 1;
        }else{ casse_brique(&l,id2); (*pparam).score=(*pparam).score+40;  return 0;
        }
     }else{
          return 0;
     }
}

/*****************************************************************************
La fonction qui gère la trajectoire (vitesse et coordonnées) de la balle selon
les obstacles rencontrés
*****************************************************************************/
void deplace_balle(struct s_balle * pballe,struct s_curseur * pcurseur,
                   liste ** ppl,struct s_param * pparam){
     int x0=(*pballe).x;
     int y0=(*pballe).y;
     int a=(*pballe).a;
     int b=(*pballe).b;
     int pas=(*pballe).pas;
     int x1,y1,t,xc,yc,xb,yb;
     int id2;
     int rbd =0;
     int type, id1,id0;
     int ps;
     point p;

     xc = (*pcurseur).x;
```

```
yc = (*pcurseur).y;
t = (*pcurseur).taille;

efface_balle(pballe);

/* cas affine */
x1=x0+a;
y1=y0+b;

/* rebond sur brique */
id0=postoid(x0,y0);
switch(cadrant(a,b)){
/* suivant l'orientation du vecteur vitesse, la composante de contact n'est pas la meme*/
   case 1: id2=postoid(x1+(pballe->taille),y1); break;
   case 2: id2=postoid(x1+(pballe->taille),y1+(pballe->taille)); break;
   case 3: id2=postoid(x1,y1+(pballe->taille)); break;
   case 4: id2=postoid(x1,y1); break;
}
if(id2!=-1){
     p=idtopos(id2);
     xb=p.x; yb=p.y;
     ps=-b*(xb-x0)+a*(yb-y0);
     switch(id2-id0){
     /*on vient d'une des huits cases autours de la case que l'on s'apprete de casser*/
       case 1  :/* rebond gd 4*/  rbd=gere_rebond(*(*ppl),pballe,id2,4,1,pparam); break;
       case -1 :/* rebond dg 3*/  rbd=gere_rebond(*(*ppl),pballe,id2,3,-1,pparam);break;
       case 20 :/* rebond hb 2*/  rbd=gere_rebond(*(*ppl),pballe,id2,2,20,pparam); break;
       case -20:/* rebond bh 1*/  rbd=gere_rebond(*(*ppl),pballe,id2,1,-20,pparam); break;


       /*brique intermédiaire*/
       case -19: if(ps>0){/* on passe au dessus du coin - rebond bh*/
                      id1=id2-1;
                      rbd=gere_rebond(*(*ppl),pballe,id1,1,-192,pparam);
                      if(rbd){id2=id1;/* il y a effectivement une brique intermédiaire a casser */
                      }else{/* rebond sur la brique initiale - gd*/
                          rbd=gere_rebond(*(*ppl),pballe,id2,4,-190,pparam);
                      }
                 }else{/* on passe en dessous du coin - rebond gd*/
                      id1=id2+20;
                      rbd=gere_rebond(*(*ppl),pballe,id1,4,-191,pparam);
                      if(rbd){id2=id1;/* il y a effectivement une brique intermédiaire a casser */
                      }else{/* rebond sur la brique initiale - bh*/
                          rbd=gere_rebond(*(*ppl),pballe,id2,1,-190,pparam);
                      }
                 }
                 break;
       case 21 : if(ps>0){/* on passe au dessus du coin - rebond gd*/
                      id1=id2-20;
                      rbd=gere_rebond(*(*ppl),pballe,id1,4,212,pparam);
                      if(rbd){id2=id1;/* il y a effectivement une brique intermédiaire a casser */
                      }else{/* rebond sur la brique initiale - hb*/
                          rbd=gere_rebond(*(*ppl),pballe,id2,2,210,pparam);
                      }
                 }else{/* on passe en dessous du coin - rebond hb*/
                      id1=id2-1;
                      rbd=gere_rebond(*(*ppl),pballe,id1,2,211,pparam);
                      if(rbd){id2=id1;/* il y a effectivement une brique intermédiaire a casser */
                      }else{/* rebond sur la brique initiale - gd*/
                          rbd=gere_rebond(*(*ppl),pballe,id2,4,210,pparam);
                      }
                }
                 break;
       case 19 : if(ps>0){/* on passe en dessous du coin - rebond hb*/
```

```
                        id1=id2+1;
                        rbd=gere_rebond(*(*ppl),pballe,id1,2,191,pparam);
                        if(rbd){id2=id1;/* il y a effectivement une brique intermédiaire a casser */
                        }else{/* rebond sur la brique initiale - gd*/
                            rbd=gere_rebond(*(*ppl),pballe,id2,3,190,pparam);
                        }
                    }else{/* on passe au dessus du coin - rebond dg*/
                        id1=id2-20;
                        rbd=gere_rebond(*(*ppl),pballe,id1,3,192,pparam);
                        if(rbd){id2=id1;/* il y a effectivement une brique intermédiaire a casser */
                        }else{/* rebond sur la brique initiale - gd*/
                            rbd=gere_rebond(*(*ppl),pballe,id2,2,-190,pparam);
                        }
                    }
                    break;
          case -21 :if(ps>0){/* on passe en dessous du coin - rebond dg*/
                        id1=id2+20;
                        rbd=gere_rebond(*(*ppl),pballe,id1,3,-211,pparam);
                        if(rbd){id2=id1;/* il y a effectivement une brique intermédiaire a casser */
                        }else{/* rebond sur la brique initiale - gd*/
                            rbd=gere_rebond(*(*ppl),pballe,id2,1,-210,pparam);
                        }
                    }else{/* on passe au dessus du coin - rebond bh*/
                        id1=id2+1;
                        rbd=gere_rebond(*(*ppl),pballe,id1,1,-212,pparam);
                        if(rbd){id2=id1;/* il y a effectivement une brique intermédiaire a casser */
                        }else{/* rebond sur la brique initiale - gd*/
                            rbd=gere_rebond(*(*ppl),pballe,id2,3,-210,pparam);
                        }
                    }
                    break;
        }
        if(rbd==1){ /*  destruction de la brique  */
          casse_brique(*ppl,id2);
          (*pparam).score=(*pparam).score+40;
        }
    }


    /* cas rebond latéraux*/
    if(x1<=Bord_gauche){                    rebond(pballe,Bord_gauche,3);  playSound("sons/mur.wav");  rbd=1;}
    else{if(x1>=(L-(pballe->taille)-Bord_droit)){  rebond(pballe,L-(pballe->taille)-Bord_droit,4); playSound("

    /* cas rebond verticaux*/
    if(y1<=Bord_haut){ rebond(pballe,Bord_haut,1); playSound("sons/mur.wav"); rbd=1;}

    /* cas rebond curseur*/
    if(y1+(pballe->taille)>=yc){
     if(y1+(pballe->taille)/2<yc+Hc/2){ /* la balle est rattrapable*/
       if(x1>=xc && x1+(pballe->taille)<=xc+t){
          /* on est au dessus du curseur*/
          /* loi linéaire de rebond en fonction de la distance au centre du curseur*/

          if((*pparam).catchme){
            (*pparam).engagement=1;
            (*pballe).x=x0;
            (*pballe).y=y0;
          }else{
            playSound("sons/curseur.wav");
            rebond(pballe,yc-(pballe->taille),2); /* rebond normal*/
            normalise_balle(pballe,(*pballe).a+1.5*(2*pas*(x1-(xc+t/2)))/t ,(*pballe).b);
          }
          rbd=1;
       }else{
```

```
            /* cas rebond les bords du curseur*/
            if( xc-x1<(pballe->taille) && xc-x1>=0){ /* rebond à gauche*/
                 (*pballe).a=-5*pas/sqrt(50) ; (*pballe).b=-5*pas/sqrt(50);
                 playSound("sons/curseur.wav");
                 rbd=1;
            }
            if(x1+(pballe->taille)-xc-t<=(pballe->taille) && x1+(pballe->taille)-xc-t>=0){ /* rebond à droite
                 (*pballe).a=5*(*pballe).pas/sqrt(50) ; (*pballe).b=-5*(*pballe).pas/sqrt(50);
                  playSound("sons/curseur.wav");
                  rbd=1;
            }

        }
      }else{
        /* la bille est irratrappable*/
         /*      rebond(pballe,H-(pballe->taille),2);    */
       (*pparam).perdu=1;

      }
    }/* cas par defaut*/
    if(rbd==0){
           /* cas par defaut*/
           (*pballe).x=x1;
           (*pballe).y=y1;
     }
} /* fin deplace balle*/
```

niveau.c

```
#include "prototypes.h"
/*****************************************************************************/
/*  chargement d'un niveau  */
/*****************************************************************************/

/*****************************************************************************
La fonction qui affiche les briques, les bonus magiques (qui sont des briques)
*****************************************************************************/
void affiche_niveau(liste l){
    point p;
    if(l==NULL){printf("Aucune brique\n");}
else{
      while(l!=NULL){
        p=idtopos(l->id);
        if((l->x)==0){
         switch((l->type)%4){
            case 0 : drawImage("images/brique_1.jpg",p.x,p.y); break;
            case 1 : drawImage("images/brique_2.jpg",p.x,p.y); break;
            case 2 : drawImage("images/brique_3.jpg",p.x,p.y); break;
            case 3 : drawImage("images/brique_4.jpg",p.x,p.y); break;
           }
        }else{/*cas des briques magiques*/
          switch((l->type)){
            case 5 : drawImage("images/lifeup.jpg",l->x,l->y); break;
            case 6 : drawImage("images/sizeup.jpg",l->x,l->y); break;
            case 7 : drawImage("images/sizedown.jpg",l->x,l->y); break;
            case 8 : drawImage("images/catchme.jpg",l->x,l->y); break;
            case 9 : drawImage("images/levelup.jpg",l->x,l->y); break;
            case 10 : drawImage("images/fall.jpg",l->x,l->y); break;
            case 11 : drawImage("images/faster.jpg",l->x,l->y); break;
            case 13 : drawImage("images/slower.jpg",l->x,l->y); break;
            case 12 : drawImage("images/gosrou.jpg",l->x,l->y); break;
            case 14 : drawImage("images/shrink.jpg",l->x,l->y); break;
            case 15 : drawImage("images/death.jpg",l->x,l->y); break;
            case 17 : drawImage("images/bigger.jpg",l->x,l->y); break;
            case 16 : drawImage("images/smaller.jpg",l->x,l->y); break;
           }
        }
        l=l->suivant;
        }
   }
}

/*****************************************************************************
La fonction qui transforme les niveaux créés dans des fichiers externes sous forme
 textuelle en niveaux du casse brique par lecture de ces fichers
*****************************************************************************/
liste niveau(int i){
  int type=0,j=0;
   liste l=NULL;
        char numfic [4];
        char nomfic [24];
FILE *fichier;

sprintf(numfic,"%d",i);
strcpy(nomfic,"niveaux/niveau_");
strcat(nomfic,numfic);
strcat(nomfic,".txt");
    fichier=fopen(nomfic,"r");
printf("Chargement du niveau %d : %s\n",i,nomfic);
while(!feof(fichier)){
fscanf(fichier,"%d",&type);
    if(type!=0){
```

```
            l_ajout(&l,j,0,0,type);
        } /* le type 0 correspond à une brique vide*/
        j++;
}
fclose(fichier);

    return l;
}
```

listes.c

```
#include "prototypes.h"

/*****************************************************************************/
/*  manipulation de liste  */
/*****************************************************************************/

/*****************************************************************************
La fonction qui recherche grace a  l'identifiant la brique voulue dans la liste
*****************************************************************************/
int l_recherche(liste l, int x){
  if(l==NULL){return 0;}else{
     if(l->id==x){if(l->x==0) return l->type; else return 0; }
    else{ l=l->suivant; return l_recherche(l,x);
    }
  }
}

/*****************************************************************************
La fonction qui supprime une brique de la liste
*****************************************************************************/
liste l_retrait(liste l, int x){
  if(l==NULL) return NULL;
  else{
    if(l->id==x) return l->suivant;
    else{l->suivant=l_retrait(l->suivant,x); return l;}
  }
}

/*****************************************************************************
La fonction permet de modifier une brique reperee par id dans une liste pl
de briques, ce qui est utile pour traiter les briques magiques
*****************************************************************************/
void l_modif(liste *pl, int id, int x, int y, int type){
  if((*pl)!=NULL){
    if((*pl)->id==id){(*pl)->x=x;(*pl)->y=y;(*pl)->type=type;}
    else{l_modif(&((*pl)->suivant),id,x,y,type);}
  }
}

/*****************************************************************************
Fonction test permettant d'afficher dans le terminal le contenu de la liste des briques
*****************************************************************************/
void l_affichage(liste l){
if(l==NULL){printf("liste vide !");}
else{
  printf("\nLe contenu de la liste est :\n");
  while(l->suivant!=NULL){printf("(%d,%d):",l->id,l->type);l=l->suivant;}
  if(l->suivant==NULL){printf("%d",l->id);}
}
}

/*****************************************************************************
La fonction qui rajoute une brique dans la liste courante
*****************************************************************************/
void l_ajout(liste * ppl, int id,  int x ,int y, int type){
  liste pcase=malloc(sizeof(struct s_liste));
/*  printf("ajout de %d\n",id);*/
  pcase->id=id;
  pcase->x=x;
  pcase->y=y;
  pcase->type=type;
  pcase->suivant=*ppl;
  *ppl=pcase;
```

```
}
```

affichage.c

```
#include "prototypes.h"

/*****************************************************************************
La fonction qui affiche le curseur selon sa position et sa taille
*****************************************************************************/

void affiche_curseur(struct s_curseur * pcurseur){
/*fillRect((*pcurseur).x,(*pcurseur).y,(*pcurseur).taille,16);*/
     switch((*pcurseur).tid){
     case 5 : drawImage("images/curseur5.jpg",(*pcurseur).x,(*pcurseur).y);break;
     case 4 : drawImage("images/curseur4.jpg",(*pcurseur).x,(*pcurseur).y);break;
     case 3 : drawImage("images/curseur3.jpg",(*pcurseur).x,(*pcurseur).y);break;
     case 2 : drawImage("images/curseur2.jpg",(*pcurseur).x,(*pcurseur).y);break;
     }
}

/*****************************************************************************
La fonction qui affiche la balle selon sa vitesse et ses coordonnees
*****************************************************************************/
void affiche_balle(struct s_balle * pballe){
/*    fillOval((*pballe).x,(*pballe).y, (pballe->taille)+15,(pballe->taille)+15);*/
     switch((*pballe).taille){
      case 11 : drawImage("images/ball11.jpg",(*pballe).x,(*pballe).y); break;
      case 13 : drawImage("images/ball13.jpg",(*pballe).x,(*pballe).y); break;
      case 15 : drawImage("images/ball15.jpg",(*pballe).x,(*pballe).y); break;
      case 17 : drawImage("images/ball17.jpg",(*pballe).x,(*pballe).y); break;
      case 19 : drawImage("images/ball19.jpg",(*pballe).x,(*pballe).y); break;
 }
}

/*****************************************************************************
La fonction qui remplace l'ancienne balle en l'effacant
*****************************************************************************/
void efface_balle(struct s_balle * pballe){
    /*  fillOval((*pballe).x,(*pballe).y, (pballe->taille)+1,(pballe->taille)+1);*/
   /*   clearRect((*pballe).x,(*pballe).y, (pballe->taille),(pballe->taille));*/
}

/*****************************************************************************
La fonction qui efface les éléments, ou la fenetre de jeu
*****************************************************************************/
void efface(){
    clearRect(Bord_gauche,Bord_haut,L-Bord_gauche-Bord_droit,H);
 /*    clearRect(Bord_gauche, H-60, L-Bord_gauche-Bord_droit, 60);  /*zone curseur*/
 /*   clearRect(Bord_gauche, 0, 100, 50);  /*zone score et vies*/

}

/*****************************************************************************
La fonction qui affiche le score courant
*****************************************************************************/
void affiche_score(struct s_param * pparam){
    char s[8];
    clearRect(Bord_gauche,0,100,20);
    sprintf(s,"%d",(*pparam).score);
    setForegroundColor(BLUE);
    setFontSize(35);
    drawText(s, Bord_gauche+10,35);
}

/*****************************************************************************
La fonction qui affiche le niveau courant
*****************************************************************************/
```

```
void affiche_niv(struct s_param * pparam){
    char s[3];
    sprintf(s,"%d",(*pparam).niv);
    setForegroundColor(RED);
    setFontSize(12);
    drawText("Niv.", L-25, 30);
    setFontSize(20);
    drawText(s, L-20, 50);
}

/*****************************************************************************
La fonction qui affiche le nombre de vies courant
*****************************************************************************/
void affiche_vies(struct s_param * pparam){
   char i=1;
   for(i=1;i<=(*pparam).nb_vies;i++){
      drawImage("images/vie.jpg",L-Bord_droit-40*i,10);
   }
}

/*****************************************************************************
La fonction qui affiche les murs droit et gauche
*****************************************************************************/
void affiche_bords(){
     drawImage("images/borddroit.jpg",L-Bord_droit,0);
     drawImage("images/bordgauche.jpg",0,0);
}
```

evenements.c

```
#include "prototypes.h"


/*****************************************************************************
La fonction qui intervient quand l'utilisateur perd une vie ou n'a plus de vie
*****************************************************************************/
void perdu(struct s_param * pparam){
        playSound("sons/perdu.wav");
        if( (*pparam).nb_vies>0){
                clearRect(Bord_gauche, 0, L-Bord_droit-Bord_gauche, H);
                setForegroundColor(BLACK);
                drawImage("images/looser.jpg",L/2-200,H/2-150);
                paint();
                sleepAWhile(1000);
                clearRect(Bord_gauche, 0, L-Bord_droit-Bord_gauche, H);
        }
         else{
                clearRect(Bord_gauche, 0, L-Bord_droit-Bord_gauche, H);
                affiche_score(pparam);
                setForegroundColor(BLACK);
                drawImage("images/perdu.jpg",L/2-300,H/2-150);
                paint();
                sleepAWhile(1000);
                clearRect(Bord_gauche, 0, L-Bord_droit-Bord_gauche, H);
          }
}

/*****************************************************************************
La fonction qu'on utilise pour sortir du jeu en cours
*****************************************************************************/
void quitter(){
    setForegroundColor(BLACK);
    fillRect (0,0,L,H);
    setForegroundColor(RED);
    drawText("Merci d'avoir joué", L/2-100, H/2);
    drawText("   A bientôt", L/2-100, H/2+40);
    paint();
    sleepAWhile(1000);
}

/*****************************************************************************
La fonction qui affiche la victoire de l'utilisateur
*****************************************************************************/
void jeu_fini(){
     setForegroundColor(BLACK);
     fillRect (0,0,L,H);
    setForegroundColor(RED);
    setFontSize(35);
    drawText("Félicitations ! Vous avez fini le jeu !", L/2-250, H/2);
    paint();
    sleepAWhile(1000);
}
```

init.c

```
#include "prototypes.h"

/*****************************************************************************/
/*  Initialisation et fermeture  */
/*****************************************************************************/

/*****************************************************************************
Cette fonction initialise la fenètre de jeu, et charge les sons
*****************************************************************************************************************/
void init(){
  start(L,H);                 // creation d'une fenetre

  setBackgroundColor(BLACK);       // couleur de fond = NOIR
  setForegroundColor(RED);         // couleur de 1er plan = ROUGE

  clearRect(0, 0, L, H);       // la fenetre est painte entierement en noir

  registerKeyPressed(VK_P);
  registerKeyPressed(VK_Q);
  registerKeyPressed(VK_SPACE);
  registerKeyPressed(VK_LEFT);
  registerKeyPressed(VK_RIGHT);
  registerKeyPressed(VK_ENTER);

  registerSound("sons/perdu.wav");
  registerSound("sons/gagne.wav");
  registerSound("sons/cassebrique.wav");
  registerSound("sons/curseur.wav");
  registerSound("sons/mur.wav");
  registerSound("sons/bonus.wav");

  drawImage("images/accueil.jpg",0,0);
  paint();
  while(getLastKeyPressed()!=VK_ENTER){}
  clearRect(0, 0, L, H);       // la fenetre est painte entiÃ¨rement en noir
}

/*****************************************************************************
La fonction qui initialise la balle
*****************************************************************************/
void init_balle(struct s_balle * pballe,int x,int y){
    (*pballe).x=x;
    (*pballe).y=y;
    (*pballe).pas=V_INIT;
    normalise_balle(pballe, A_INIT, B_INIT);
    (*pballe).taille=Dballe;
}

/*****************************************************************************
La fonction pour le curseur
*****************************************************************************/
void init_curseur(struct s_curseur * pcurseur,int x_curseur,int y_curseur, int pas_curseur){
    (*pcurseur).x=x_curseur;
    (*pcurseur).y=y_curseur;
    (*pcurseur).tid=3;
     ajuste_taille(pcurseur);
    (*pcurseur).pas=pas_curseur;
}

/*****************************************************************************
La fonction pour les parametres de jeu
*****************************************************************************/
void init_param(struct s_param * pparam,int niv,int n,int score){
```

```
    (*pparam).nb_vies=n;
    (*pparam).engagement=1;
    (*pparam).catchme=0;
    (*pparam).gosrou=0;
    (*pparam).quitter=0;
    (*pparam).perdu=0;
    (*pparam).gagne=0;
    (*pparam).niv=niv;
    (*pparam).score=score;
}

/*****************************************************************************
La fonction qui gere la terminaison
*****************************************************************************/
void fin(){
  finish();
}
```

auxiliaires.c

```
#include "prototypes.h"

/*****************************************************************************/
/*  fonctions auxiliaires  */
/*****************************************************************************/

/*****************************************************************************
La fonction qui donne l'identifiant d'une brique en fonction de coordonnées
*****************************************************************************/
int postoid (int x, int y){
  int i;

  if(y<=Bord_haut || y>=Bord_haut+20*Hb){
    /* les coordonnées données ne sont pas dans la zone de brique*/
    return -1;
  }else{
    i= ((x-Bord_gauche)/Lb)+ 20*((y-Bord_haut)/Hb) ;
    return i;
  }
}

/*****************************************************************************
La fonction qui donne les coordonnees du bord superieur gauche de la brique en
fonction de son identifiant
*****************************************************************************/
point idtopos (int i){
  point p;

  p.x= Bord_gauche+(i%20)*Lb;
  p.y= Bord_haut+(i/20)*Hb;

  return p;
}

/*****************************************************************************
La fonction qui renvoie dans lequel des quatres cadrants se situe la balle
*****************************************************************************/
int cadrant (int a, int b){
    if(a>=0){
            if(b>0){return 2;} else{return 1;}
    }else{
            if(b>0){return 3;} else{return 4;}
    }
}

/*****************************************************************************
La fonction qui normalise le vecteur vitesse de la balle après calcul de trajectoire
*****************************************************************************/
void normalise_balle(struct s_balle * pballe, int a,int b){
     int pas=(*pballe).pas;
     /* Lvecteur a,b est de norme pas*/
    (*pballe).b=pas*b/sqrt(a*a+b*b);
     if((*pballe).b==0){(*pballe).b=1;}
    (*pballe).a=pas*a/sqrt(a*a+b*b);
}

/*****************************************************************************
La fonction qui gère la modifictation de taille du curseur
*****************************************************************************/
void ajuste_taille(struct s_curseur* pcurseur){
     switch((*pcurseur).tid){
      case 1:(*pcurseur).tid=2;break;
      case 2:(*pcurseur).taille=60;break;
```

```
      case 3:(*pcurseur).taille=130;break;
      case 4:(*pcurseur).taille=193;break;
      case 5:(*pcurseur).taille=300;break;
      case 6:(*pcurseur).tid=5;break;
    }
}
```

prototypes.h

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "graphic.h"
#include "key.h"


/*dimensions de la zone de jeu*/
#define L 960
#define H 800
#define Bord_gauche 30
#define Bord_droit 30
#define Bord_haut 20

/* nombre de niveaux*/
#define N_MAX_NIV 3

/*dimensions des briques*/
#define Lb 45
#define Hb 25

/*dimensions curseur*/
#define Lc_INIT 128
#define Hc 16
#define PASc_INIT 8 /* pas de deplacement du curseur*/

/*dimensions bille initiale*/
#define Dballe 15

/* paramètres */
#define NB_VIES 3
#define SLEEP 5  /*raffraichissement*/
#define V_DESC 4 /*vitesse descente des briques magiques*/
#define V_INIT 8  /*vitesse balle initiale*/
#define A_INIT 1  /*composante en x du vecteur vitesse initial*/
#define B_INIT -3 /*composante en y du vecteur vitesse initial*/


/*****************************************************************************/
/*  definitions de types  */
/*****************************************************************************/
struct s_liste { int id; int x;int y;int type; struct s_liste * suivant;};
typedef struct s_liste * liste;

struct s_point{int x; int y;};
typedef struct s_point point;


struct s_balle{int x; int y; int a; int b;int pas;int taille;};

struct s_curseur{int taille; char tid; int x;int y;int pas;};

struct s_param{char quitter;
               char perdu;
               char gagne;
               char engagement;
               char catchme;
               char gosrou;
               char nb_vies;
               char niv;
               int score;};
```

```
/*************************************************************************/
/*  entetes  */
/*************************************************************************/

/* init.c : initialisation et fin*/
void init();
void init_balle(struct s_balle * pballe,int x,int y);
void init_curseur(struct s_curseur * pcurseur,int x_curseur,int y_curseur, int pas_curseur);
void init_param(struct s_param * pparam,int niv,int n,int score);
void fin();

/* affichage.c */
void affiche_score(struct s_param * pparam);
void affiche_vies(struct s_param * pparam);
void affiche_bords();
void affiche_curseur(struct s_curseur * pcurseur);
void affiche_balle(struct s_balle * pballe);
void affiche_niv(struct s_param * pparam);
void efface_balle(struct s_balle * pballe);
void efface();

/* dynamique.c */
void casse_brique(liste *pl,int id);
void rebond(struct s_balle * pballe, int lim, char type);
int gere_rebond(liste l,struct s_balle * pballe,int id2, int direction, int type,struct s_param * pparam);
void gestion_touches(struct s_curseur * pcurseur,struct s_param * pparam);
void gestion_curseur(struct s_curseur * pcurseurn,liste *pl,struct s_param * pparam,struct s_balle * pballe);
void deplace_balle(struct s_balle * pballe,struct s_curseur * pcurseur, liste ** ppl,struct s_param * pparam);

/*  listes.c  */
int l_recherche(liste l, int x);
liste l_retrait(liste l, int x);
void l_affichage(liste l);
void l_ajout(liste * ppl, int id, int x,int y, int type);
void l_modif(liste *pl, int id, int x, int y, int type);

/* auxiliaires.c */
int cadrant (int a, int b);
point idtopos (int id);
int postoid (int x, int y);
void normalise_balle(struct s_balle * pballe, int a,int b);
void ajuste_taille(struct s_curseur* pcurseur);

/* niveau.c*/
void affiche_niveau(liste l);
liste niveau(int i);

int jeu_au_sein_du_niveau(liste * l,struct s_balle * pballe, struct s_curseur * pcurseur,struct s_param * ppara

/* evenements.c*/
void perdu(struct s_param * pparam);
void quitter();
void jeu_fini();
```