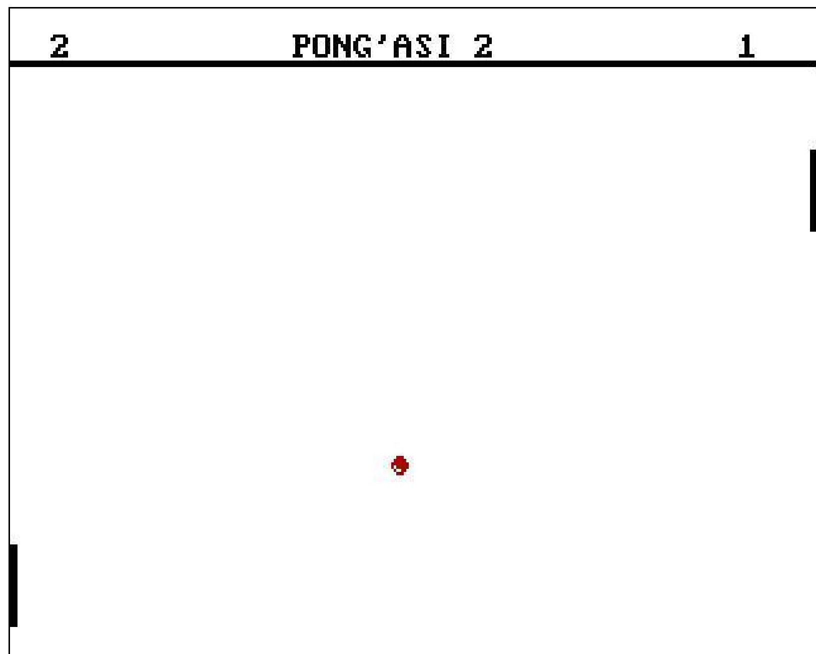
Rapport de projet : k - ASI - Brique

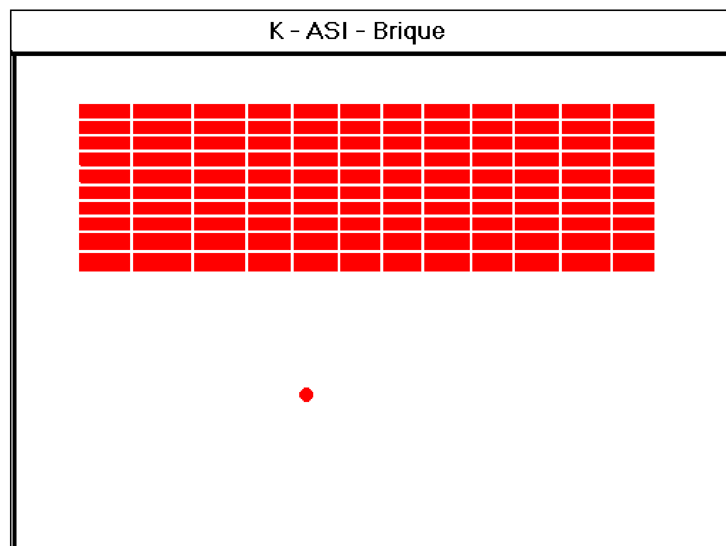
Objectif : A partir du projet Pong-ASI, transformer ce dernier en un jeu similaire, le k-ASI-brique.

1) Présentation

Le Pong-ASI, développé l'an dernier, est un jeu pour deux joueurs consistant à renvoyer une balle avec une raquette. Les raquettes sont disposées à gauche et à droite de l'écran, la balle rebondit sur les bords supérieurs et inférieurs de la zone de jeu.



Pour sa part, le k-ASI-brique consiste pour un joueur, toujours muni d'une raquette, à casser des briques en faisant rebondir une balle. La raquette cette fois ci se voit positionnée sur le bord inférieur de l'écran.



2) Travail effectué

a) Graphique

Le pong asi se jouait au départ en mode horizontal. Nous avons donc transposé les bords et la raquette de manière à situer la raquette en bas de l'écran et placé des lignes de délimitation de zone de jeu à droite, à gauche et en haut de l'écran. Nous avons aussi rajouté l'affichage des briques.

b) Score et vies

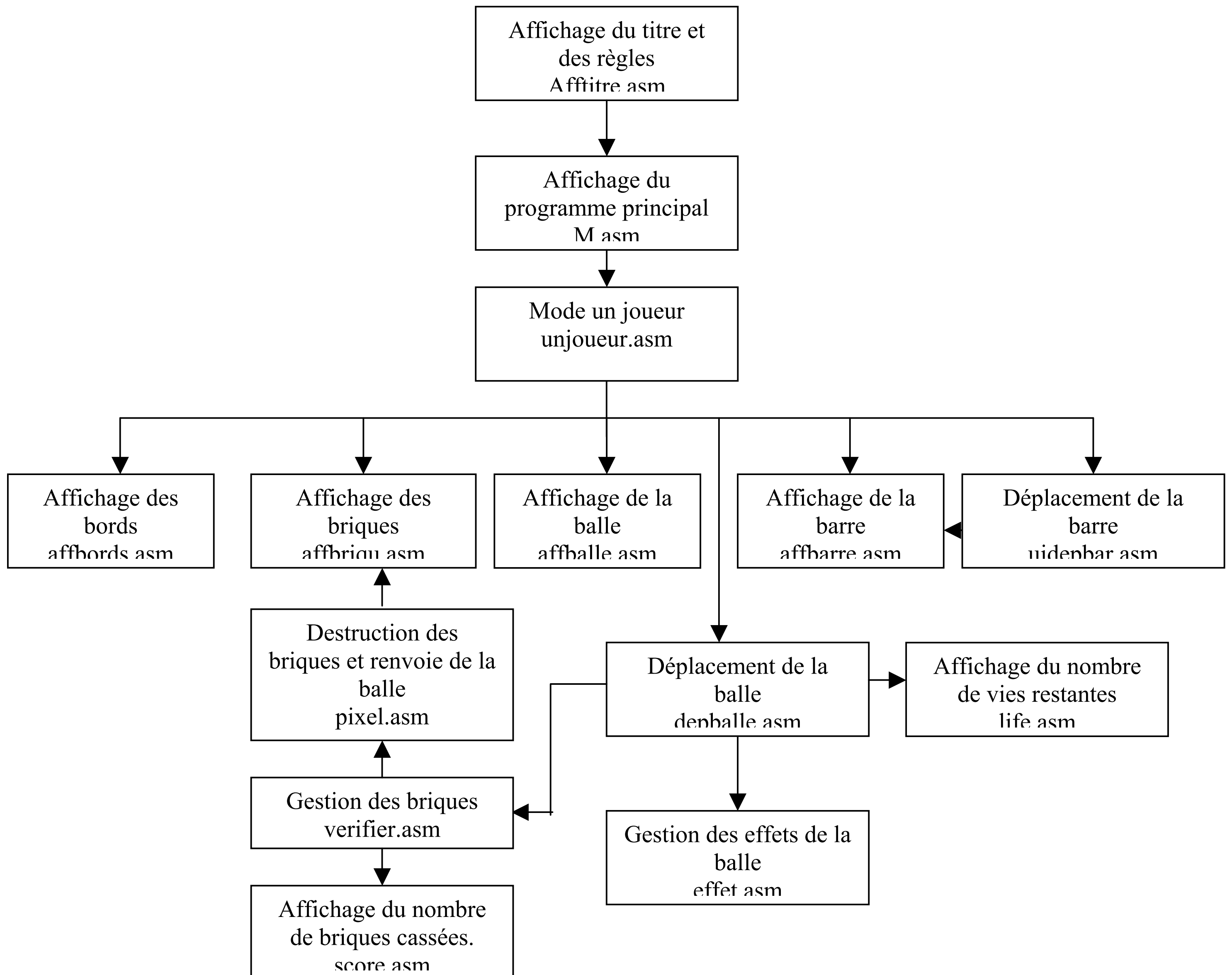
Par rapport au pong asi de l'an dernier, le but du jeu a été légèrement modifié. Il ne s'agit plus de faire perdre la balle à l'adverse un maximum de fois, mais de casser un maximum de briques avec 5 balles. Nous avons donc créé et affiché un score comptabilisant le nombre de briques cassées. Nous affichons aussi le nombre de « vies » restant. Celui-ci est initialisé à 5, et diminue chaque fois que le joueur perd la balle. Le jeu s'arrête lorsque le joueur a perdu ses 5 balles ou a cassé toutes les briques.

c) Le déplacement de la balle

Le déplacement de la balle, les effets ont aussi été modifiés et corrigés. Il a fallu intégrer la composante « brique » de manière à ce que la balle rebondisse sur une brique et l'efface. Les données relatives aux briques (position, présence…) ont été placées dans le fichier donnees.asm. Nous avons créé 4 programmes permettant de gérer les briques. Cela a sans doutes été la partie la plus délicate du projet.

3) Organigramme

Le but de l'organigramme que nous vous présentons ci-dessous est de détailler de façon claire les différentes fonctions rentrant en jeu lors du déroulement du programme et quels rôles elles tiennent au sein de ce dernier.



4) Présentation des différents fichiers

a) Le fichier appelé par l'utilisateur

Lancer.bat : ce script appelle les exécutables *afftitre.exe* et *m.exe,* déclenchant ainsi l'affichage des règles du jeu et son lancement.

b) Le fichier d'aide à la compilation et à l'édition des liens :

Link.bat : le script permet d'exécuter d'un coup la compilation et l'édition des liens des fichiers composant le jeu. A la sortie de ce script, le fichier m.exe est généré.

La présence de ce script n'est pas indispensable mais évite la fastidieuse saisie 13 lignes à chaque fois que l'on souhaite recompiler le jeu après une modification du code.

c) Les programmes générant des exécutables

m.asm : il recueille les choix du joueur en ce qui concerne la vitesse de la balle et la taille de la raquette. De même il appelle la fonction de lancement de jeu *unj*.

Afftitre.asm : réalise l'affichage le logo du casse brique ainsi que les règles du jeu.

d) Les fichiers de données

Textes.asm : Ce fichier est spécifique au programme afftitre.asm. Il contient les chaînes de caractères qui seront affichées par la suite par afftitre.asm

Données.asm : Ce fichier est utilisé par toutes les fonctions du jeu, à l'exception de afftitre.asm. Il comprend notamment des données telles que les coordonnées de la barre, de la balle ou encore des briques.

e) Les fichiers de fonctions

Affbarre.asm : Il contient la fonction qui affiche la raquette de jeu sur trois lignes et selon le paramètre `LARGEUR_BARRE` de la raquette de jeu. Il utilise comme arguments la couleur d'affichage (noir ou blanc) et l'abscisse du coin supérieur gauche de la raquette.

Affbords.asm : contient la fonction qui affiche les lignes délimitant la zone de jeu, le titre et enfin les scores.

Affballe.asm : contient la fonction qui réalise l'affichage de la balle à partir du paramètre *`couleur`* et des coordonnées *`x`* et *`y`* définies dans donnees.asm.

Affbriqu.asm : contient la fonction qui réalise l'affichage des briques. A cet effet, on emploie deux tableaux se trouvant dans données.asm. Le premier, qui s'appelle `presenceBrique`, contient une valeur indiquant si la brique est présente ou pas. Cette valeur et placée à 1 si la brique est présente, 0 dans le cas contraire. On réalise un test pour

savoir quelle est la valeur. Si le résultat retourné est 1, alors la brique est dessinée en rose, sinon on la dessine en noir. Dans ce dernier cas la brique n'apparaît pas sur le plan de jeu.
On dessine ensuite la brique en se servant du second tableau à savoir `positionsBrique`. Ce tableau contient la position en x et la position en y du coin supérieur gauche de chacune des briques. De ce fait, comme nous n'avons que 50 briques, le tableau `positionsBrique` comportera 100 valeurs. Pour pouvoir ensuite afficher la brique, on réalise une double boucle.
Dans la première, on se positionne au début de la colonne puis, dans la deuxième, on dessine la ligne en se déplaçant suivant l'axe des y. Il convient donc d'appeler cette fonction dans une boucle pour pouvoir réaliser le parcours des deux tableaux.

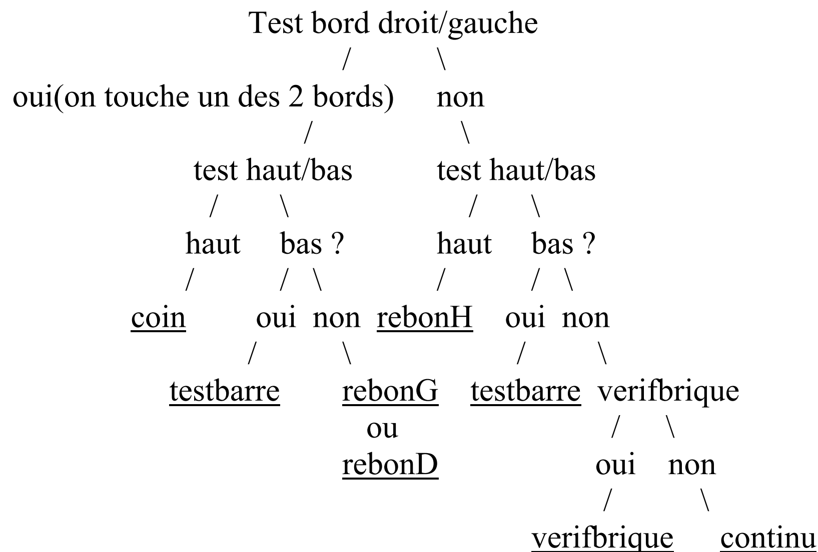
life.asm : La gestion des vies restant au joueur se réalise via la fonction *decvie* qui est définie dans le fichier life.asm. De même, cette fonction intervient sur la valeur de la variable vie définie dans le fichier données.asm. La fonction définie à l'intérieur de ce fichier, *decvie* est une fonction appelée dès que le joueur vient à perdre une balle. Dans ce cas, *decvie* décrémente la valeur de la variable vie et réalise l'affichage de cette nouvelle valeur. Une fois que cette variable arrive à zéro et que le joueur vient de nouveau de perdre une balle, la fonction détectera cette tentative de décrémentation sur une valeur déjà nulle et conduira à la fin du jeu en mettant à 1 la variable quit, paramètre déclenchant la fin immédiate du jeu. Concernant l'usage de *decvie*, il faut néanmoins remarquer que nous utilisons tout de même cette fonction pour réaliser l'affichage de la variable vie au lancement du jeu. La fonction réalisant forcément une soustraction unitaire à la variable vie, il convient pour un nombre n de vies de fixer la valeur de la variable à n+1 pour que, lors du premier affichage, la décrémentation n'ait pas d'effet sur le nombre de vies que vous auriez souhaité allouer au joueur.

score.asm : L'affichage des scores est réalisé via la fonction *affscore*, fonction définie dans le fichier score.asm. De façon basique, cette fonction se comporte comme un compteur dont le nombre maximal affichable correspond au nombre de briques présentes dans le niveau.
L'affichage du score se décompose en fait en deux sous-affichages à savoir l'affichage des unités (via la variable scoreU) et l'affichage des dizaines (via la variable scoreD). A chaque fois qu'une brique est touchée, on incrémente la valeur de scoreU puis on réalise l'appel à affscore. Cette fonction va alors tester si scoreU a dépassé le caractère ASCII 9. Si c'est le cas cela veut dire que l'on a cassé dix briques, on incrémente alors scoreD et nous remettons scoreU à zéro. Bien évidemment, *affvie* teste aussi si scoreD a atteint la valeur 5, prouvant ainsi que le joueur a cassé la totalité des briques et qu'il convient alors de quitter le jeu.

Ujdepbar.asm : contient la fonction qui déplace la barre à l'appui des touches *W* et *X*. Ce déplacement est effectué en se basant sur la position de la barre `positionbarre1`, sur le pas de déplacement `pas_deplacement` et enfin de la limite de zone de jeu (de valeur 318).

DepBalle.asm : contient la fonction qui gère le déplacement de la balle
Dans cette procédure nous devons gérer le déplacement de la balle en fonction de la présence de la barre, d'un bord ou encore des briques. Il faut donc comparer à chaque fois la position de la balle avec la position des différents objets formant l'environnement du jeu. Il y a de même certaines contraintes d'espaces à respecter, ainsi faut-il faire attention à ne pas empiéter sur les différents bords de la zone de jeu sous peine de voir l'affichage se dégrader.

Cette procédure comporte donc une série de tests et des appels aux fonctions 'effet' et 'verifbrique'(rebonds sur les briques). Il est possible de résumer cela à l'aide du schéma suivant :



Si l'on est dans un coin (haut droit ou gauche) on doit inverser les signes des 2 déplacements élémentaires.

Si l'on touche un côté, il faut inverser le signe du déplacement selon X (rebonG ou rebonD).

Si l'on touche le bord haut, il faut inverser le signe du déplacement selon Y (rebonH).

Si l'on est en bas, il faut faire le test de collision avec la barre (testbarre). Dans le cas où la balle touche effectivement la barre on appelle 'effet' sinon la balle est perdue et le joueur perd une vie.
Si aucun des tests précédents n'est concluant, on est donc à l'intérieur de l'aire de jeu. Il reste à présent à vérifier si l'on est dans la zone des briques.
Dans l'affirmative on appelle la fonction qui gère les rebonds sur les briques et l'effacement des briques(verifbrique). Sinon, la balle continue et les déplacements élémentaires restent les mêmes, il suffit alors d'incrémenter les variables de position.

verifier.asm : contient la fonction qui vérifie si la balle a touché une brique ou pas. Pour cela nous avons utilisé quatre points particuliers qui se trouvent être les quatre sommets du carré dans lequel la balle peut être inscrite. Nous augmentons à chaque fois les coordonnées de ce point de 1, et nous regardons si le pixel dont les coordonnées sont celles de ce point est allumé ou pas. Si c'est le cas, alors nous faisons appel à une fonction qui va déterminer quelle est la brique touchée et quel est le rebond que doit avoir la balle.

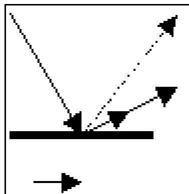
pixel.asm : c'est dans ce fichier que se trouve la fonction qui détermine le rebond que doit avoir la balle après quelle ait touché une brique. Pour cela on détermine sur quelle face se trouve le pixel qui a été testé auparavant.

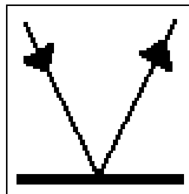
Si c'est sur le côté gauche ou droit, alors on inverse le déplacement en x, et si c'est le côté supérieur ou inférieur, alors on inverse le déplacement en y. Au final on fait appel à la fonction qui permet de savoir quelle brique a été touchée.

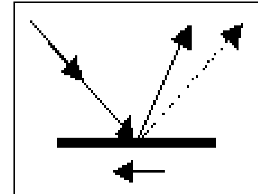
test.asm : c'est dans ce fichier que se trouve la fonction qui permet de savoir quelle brique a été touchée. Pour cela nous réalisons une boucle pour tester toutes les briques. Dans cette boucle on regarde si les coordonnées du pixel testé font qu'il se trouve dans la brique. Si c'est le cas alors on change la valeur de la brique dans le tableau `presenceBrique` de façon à ce que cette dernière ne soit plus présente sur le terrain de jeu. Ensuite on fait appel à la fonction affbrique de façon à faire disparaître la dite brique.

Unjoueur.asm : contient la fonction qui gère le passage en mode graphique 320*200, définit la vitesse du jeu, appelle les fonctions d'affichages et de déplacements, génère le lancement d'une nouvelle balle quand nécessaire et enfin gère l'arrêt du jeu et la restauration du mode graphique initial.

effet.asm : la fonction effet gère le rebond sur la raquette selon le principe suivant :







BROSSAGE Balle accélérée Trajectoire plus basse que la normale	NORMAL Vitesse constante Trajectoire de réflexion normale	LIFT Balle ralentie Trajectoire plus haute que la normale

5) Difficultés rencontrées

Le regroupement des fichiers modifiés par les uns et les autres n'a pas toujours été évident. En effet, nous avons parfois dû travailler sur les mêmes fichiers, comme donnees.asm. En intégrant un à un chaque fichier, nous avons finalement réussi à fusionner notre travail. Ceci montre néanmoins que la bonne gestion du versionning des fichiers d'un projet peut vite s'avérer délicate si l'on ne s'y attelle pas avec rigueur.
Le regroupement de nos fichiers nous a ainsi permis d'obtenir une première version du jeu intégrant des briques, une barre et des bords en place. Les personnes chargées du déplacement de la balle, des effets et de l'interaction briques balles ont alors pu tester leurs programmes, et les corriger. Cette partie était de loin la plus difficile techniquement parlant.

Une des premières difficultés pour réaliser la gestion des briques est venue avec la façon de sauvegarder les coordonnées des briques. En effet, nous ne pouvions nous servir que de 2 tableaux en même temps. De plus, comme nous utilisions déjà un tableau pour la gestion de la présence des briques, il ne nous restait alors plus qu'un seul tableau pour

pouvoir sauvegarder à la fois les positions en x et les positions en y de chacune des briques. Ce problème a été résolu en regroupant les coordonnées en x et en y dans un seul et même tableau. Une des autres difficultés a été la gestion de la destruction des briques.
Nous n'avions pas en effet d'idée au départ pour savoir comment nous allions la mettre en pratique. Puis, nous avons découvert qu'il existait une fonction permettant de savoir la couleur d'un pixel à partir de ses coordonnées. De là nous avons trouvé (non sans mal) une méthode viable pour la destruction des briques.

En ce qui concerne les effets, nous nous sommes vite rendu compte que la fonction était appelée trop tard dans le cas du programme original. En effet, dans le Pong, on faisait tout d'abord rebondir la balle et seulement après les effets se voyaient appliqués. A présent, le rebond contre la barre est assuré dans notre code via la fonction effet. Cependant, seule l'accélération est appliquée à la balle. En effet, nous ne sommes malheureusement pas arrivés à, d'une part ralentir la balle et d'autre part à changer sa trajectoire.
De même, nous avons aussi tenté -sans succès- de ne créer qu'un seul exécutable m.exe faisant appel à afftitre. Dans sa configuration de départ, le projet pong générait en effet deux exécutables totalement indépendants l'un de l'autre, car nos prédécesseurs se virent limités par le nombre maximal de caractères supportés par l'émulateur dos. Nous avons contourné ce problème de la façon suivante en remplaçant :

lil m.obj affbriqu.obj pixel.obj affballe.obj affbords.obj affbarre.obj depbarre.obj depballe.obj unjoueur.obj verifier.obj test.obj effet.obj ujdepbar.obj score.obj life.obj

par

lil m affbriqu pixel affballe affbords affbarre depbarre depballe unjoueur verifier test effet ujdepbar life score

La commande réalisée reste la même mais permet cependant d'ajouter d'autres caractères nous laissant ainsi une certaine marge de manœuvre.
Nous avons donc tenté de transformer le programme principal afftitre.asm en une procédure de type FAR. Malheureusement, nous nous sommes heurtés à ce que nous avons pensé être un problème de déclaration multiple de segments de données et de piles. En effet, les fichiers textes.asm et donnees.asm déclarent chacun un segment de données et un segment de pile. Il était impossible de fusionner les deux dans donnees.asm par exemple, car dans ce cas certains fichiers incluant donnees.asm auraient dépassé la limite fatidique des 150 lignes imposée par lasm du fait que nous n'utilisions qu'une version bridée de ce logiciel. Or, si nous les incluons tous les deux dans le même programme principal, ces deux segments entrent en conflit ce qui s'est traduit au cours de nos tests soit par un affichage aberrant, soit par le plantage du programme.

6) Les points à améliorer

La vitesse de déplacement de la balle dépend de la puissance de l'ordinateur sur lequel le jeu est exécuté. Nous l'avons paramétrée de manière à obtenir une version jouable sur les ordinateurs de l'INSA où aura lieu notre démonstration. Mais le déplacement de la balle est trop rapide pour être vraiment jouable sur nos ordinateurs personnels d'autant plus que le déplacement de la raquette est lié à une interruption système détectant l'appui sur une touche, lequel reste relativement lent nuisant encore plus à la jouabilité de l'ensemble.

Dans le même ordre d'idée et comme nous l'avons déjà mentionné ci- avant, nous ne sommes pas parvenus à appliquer tous les effets souhaités sur le déplacement de la balle. Nous parvenons à l'accélérer, mais non pas à le ralentir ou à modifier sa trajectoire.

7) L'avenir

Le casse brique est maintenant opérationnel. Cependant, il peut être encore amélioré notamment selon les axes suivant :

- ✓ Amélioration du graphique par des briques de couleurs différentes.
- ✓ Création de plusieurs tableaux de briques, un nouveau tableau apparaissant dès que le tableau courant est terminé.
- ✓ Intégration de briques incassables pour augmenter la difficulté de jeu.
- ✓ Création d'un fichier palmarès, conservant sur le disque dur les meilleurs scores.
- ✓ Possibilité d'un mode deux joueurs : les deux joueurs jouent chacun leur tour sur le même tableau, l'échange ayant lieu à chaque perte de balle.

8) Conclusion

Nous avons donc, non sans quelques difficultés, réussi à transformer le pong en casse-brique. La présence au départ du pong opérationnel et de son code relativement bien commenté (en fait, tout à fait suffisamment pour certains, pas assez pour d'autres. Les avis divergent sur ce point) a été une aide précieuse. La rédaction du rapport, et notamment de l'organigramme du jeu, a aussi contribué à améliorer le package final du jeu. En visualisant les liens entre les fichiers, nous nous sommes en effet aperçu qu'un des fichiers n'était appelé par nul autre, et était donc inutile. Ce point montre néanmoins toute la difficulté qu'il peut y avoir à repartir d'un code déjà existant. Dans notre cas, l'expérience s'est plutôt bien passée du fait de commentaires sur le code de départ, d'une taille des fichiers sources raisonnable et de la connaissance des personnes ayant participé à l'élaboration du précédent projet. Mais dans la réalité les conditions de travail auxquelles nous serons confrontées pour un travail similaire ne seront sûrement pas aussi propices. L'avantage de ce projet est au moins de nous avoir sensibilisé sur la façon dont un code pouvait évoluer pus au moins facilement en fonction du soin apporté à sa réalisation au départ.

La difficulté relative au nombre de lignes maximales compilées par lasm ne nous a pas surprise, mais nous nous y sommes cependant heurtée (d'où des codes peu aérés). Finalement, ce projet a été une bonne occasion de mettre en application le langage assembleur et ses possibilités que nous n'avions pas imaginées aussi larges.