LECRUBIER Vincent
OLIVIER Solenne

Bureau d'étude informatique

Supanoïd

Table des matières

1 Introduction

1.1 Cahier des charges initial

L'objectif principal de ce BE est la réalisation d'un jeu vidéo de type casse-brique, nommé Supanoïd. Le cahier des charges est le suivant :

- La fenêtre doit contenir une zone de jeu et une zone de score.
- La balle doit rebondir sur tout les rebords de la zone de jeu, sauf celui du bas. Lorsque la balle touche le bord inférieur, elle disparaît, et le joueur perd une vie.
- Lorsque la balle touche une brique, elle rebondit sans effet et la brique disparaît.
- Lorsque la balle touche la raquette, elle rebondit avec effet.
- Lorsque toute les briques ont disparu, le niveau est fini et un nouveau niveau apparaît.
- Le calcul du score est libre.

La réalisation du fonctionnement de base du jeu, en suivant le cahier des charges minimal, ayant été très rapide et simple, nous avons décidé de recommencer le programme en adoptant un point de vue beaucoup plus généraliste et souple pour l'utilisateur.

1.2 Fonctionnement avancé

La configuration du jeu est basée sur l'interprétation de scripts écrits par l'auteur du niveau, permettant de définir de manière très précise le comportement du jeu, avec un grand éventail de possibilités.

Ainsi, la version finale du jeu présente les caractéristiques suivantes :

- Liberté totale de création de niveaux.
- Liberté totale de création et d'utilisation d'effets spéciaux, de graphiques, de sons.
- Liberté totale de configuration du comportement physique des objets.

La création du jeu passe donc non seulement par la programmation de l'exécutable, mais aussi par la création des scripts permettant de générer l'environnement du joueur.

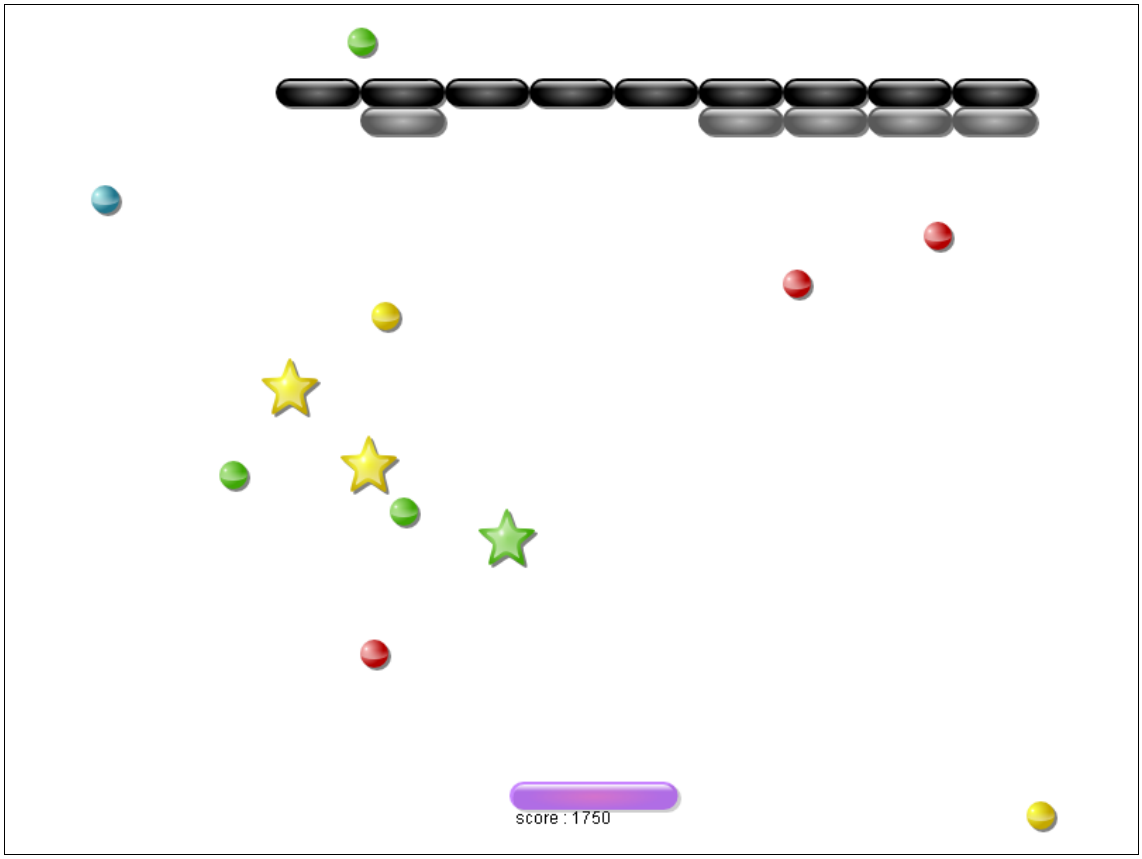


Illustration 1: Supanoid en action !

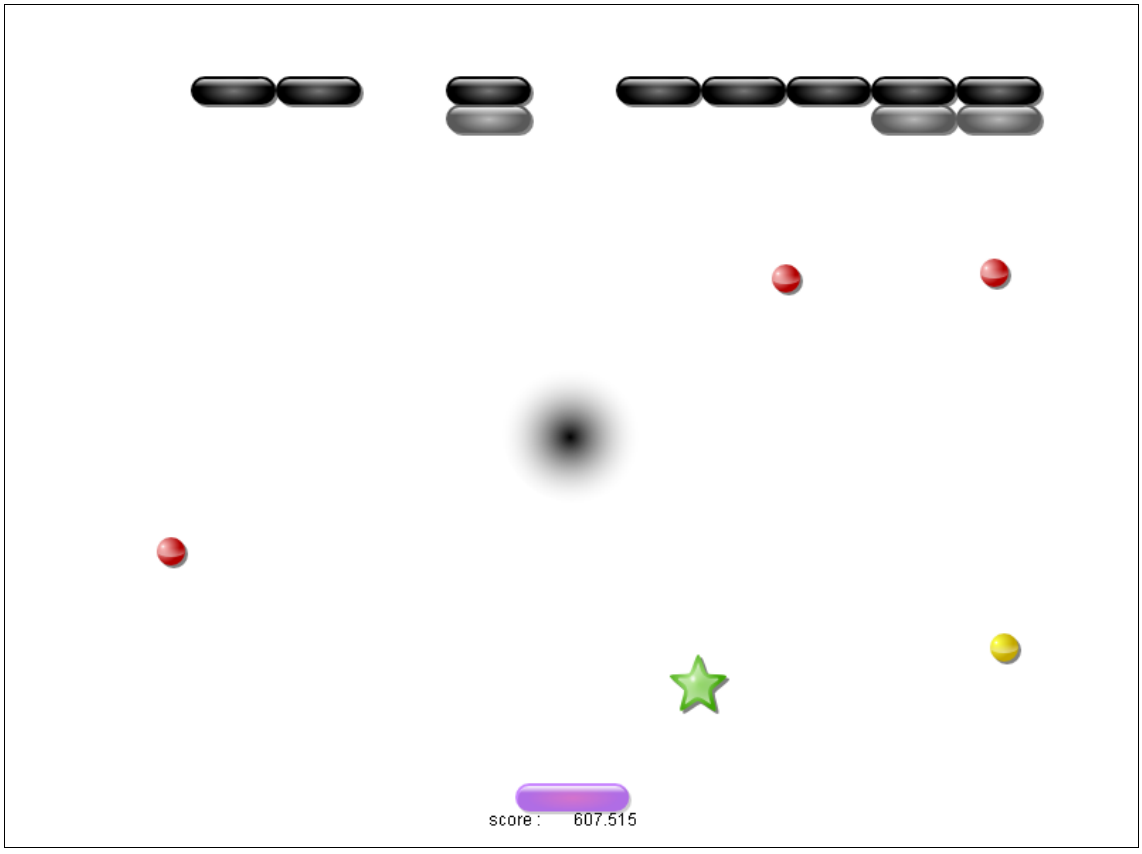


Illustration 2: Le trou noir dévore tout sur son passage...

2 Analyse

2.1 Choix de conception

2.1.1 Première approche

La conception du programme s'est fait en suivant une alternance descendante/ascendante. Le cheminement qui nous a amené a réaliser l'architecture actuelle de Supanoïd s'est déroulé chronologiquement selon les étapes suivantes :

1. Réflexion sur les différents types d'objets à introduire (Balle, Raquette, Briques) et de la structure globale du jeu.
2. Définition des propriétés des différents objets au cas par cas.
3. Les différents objets correspondant au départ à différents TDA nous semblant très similaires, nous décidons de réunir tout les objets sous un même TDA.
4. Élargissement de la notion d'objet à différentes entités liées au jeu : La fenêtre de jeu, la zone d'affichage des scores. Le programme n'aura donc à travailler que sur un unique TDA.
5. La notion d'objet généralisée introduite nous paraissant très souple, nous regrettons que toutes les caractéristiques des objets soient inscrites dans le code source du programme, et non dans des fichiers de configuration. Nous décidons de lire les niveaux dans des fichiers extérieurs. Ces fichiers commandent l'application de propriétés aux objets. (Par exemple largeur=100).
6. Nous regrettons que, de même que nous pouvons configurer les objets, nous ne pouvions configurer les actions liées à certains évènements du jeu. Nous décidons alors d'adopter non plus un langage descriptif (type a.x=b) dans les fichiers de configuration, mais de rédiger la configuration sous forme de scripts interprétés par le programme (type modifierpropriete[a,x,b]). Les scripts permettent de configurer des comportements avancés, via l'utilisation de bloc en si alors sinon etc...

2.1.2 Conception

Une fois le fonctionnement désiré du programme déterminé, il a fallu identifier les principaux sous problèmes liés à la réalisation du projet. Les principaux aspects identifiés sont :

- La gestion des propriétés physique des objets.
- L'organisation et la gestion des données générées par le programme.
- L'interface utilisateur, l'affichage, le son.
- Le paramétrage du programme, la gestion des fichiers et des commandes.

Afin d'harmoniser et de faciliter la gestion de l'information dans le programme, et la communication entre les principaux sous programmes, nous nous sommes progressivement tournés vers un type de donnée abstraite unique, regroupant tout les objets pouvant exister dans le jeu, de la fenêtre à l'affichage du score en passant par la balle et les briques.

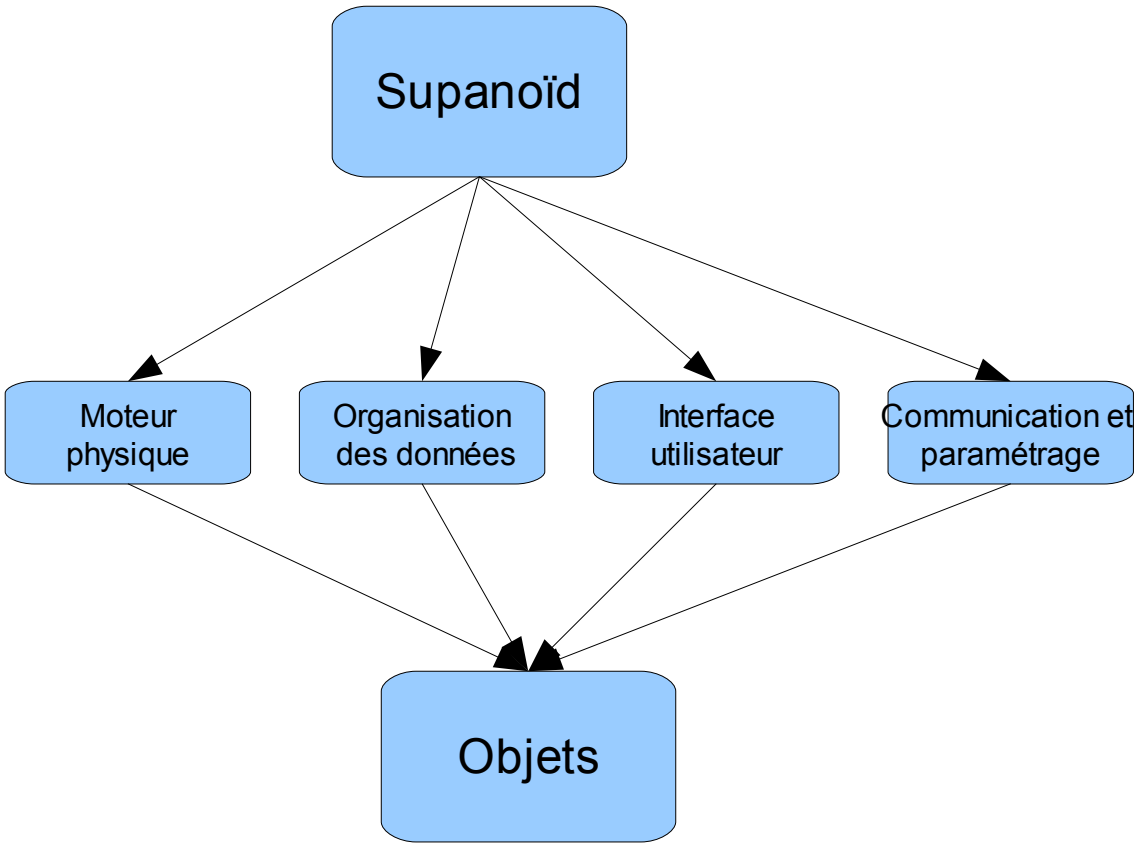


Illustration 3: Mise en commun des données et de leurs différents aspects dans Supanoïd

2.2 Types de données abstraites

Les données de Supanoïd sont toutes placées dans les cellules d'une liste doublement chaînée. A chaque TDA présenté ci dessous, nous avons associés les fonctions de base permettant :

- La création d'une nouvelle variable avec tout ses paramètres en allouant l'espace mémoire nécessaire.
- La copie d'une variable existante en mémoire.
- La suppression de la variable et la libération de la mémoire qu'elle utilisait.
- La création d'une variable contenant les paramètres par défaut.

2.2.1 Cellule

Le TDA CELLULE est l'élément de base de la liste chaînée, on y retrouve donc les éléments habituels :

- Un contenu
- Un pointeur vers l'élément suivant
- Un pointeur vers l'élément précédent

Le contenu de la cellule consiste en un identifiant et un objet, tous deux uniques et dépendants de la cellule. Ces deux types de données contenus par la cellule sont décrits plus loin. Le TDA CELLULE est défini par le code suivant :

```
typedef struct cellulestruct
{
   struct objetstruct* element;
   struct identifiantstruct* identifiant;
   struct cellulestruct* suivant;
   struct cellulestruct* precedent;
} CELLULE, *PTRCELLULE;
```

On peut donc représenter la cellule par le schéma suivant :

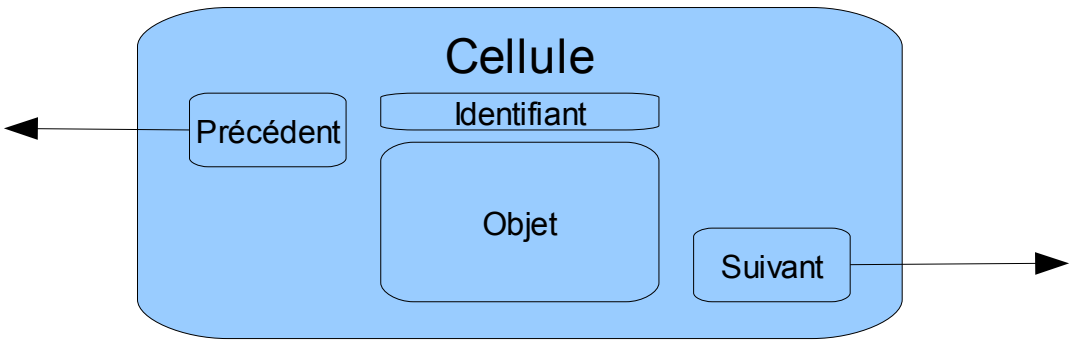


Illustration 4: Une cellule contenant un objet de Supanoïd

La liste chaînée a donc l'allure ci dessous :

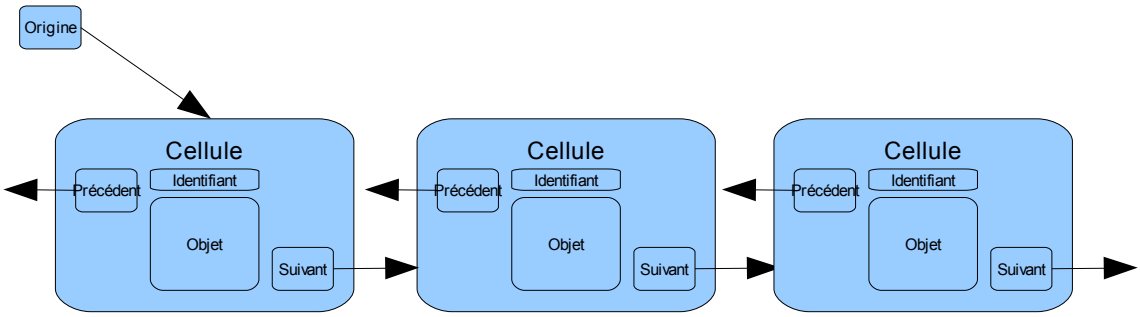


Illustration 5: Allure de la liste des objets dans Supanoïd

Il est à remarquer que dans la pratique, le contenu d'une cellule n'est pas directement inclus dans la cellule, mais qu'il est pointé depuis la cellule. Cela permet de faciliter et d'optimiser la manipulation des cellules, mais le fonctionnement global du programme se comprend aussi bien si l'on considère que l'objet et l'identifiant sont directement inclus dans la cellule.

2.2.2 Objet

Le TDA OBJET représente un objet physique, avec toutes les caractéristiques nécessaires pour obtenir un comportement réaliste, et un affichage personnalisable. Il est défini par le code suivant :

```
typedef struct objetstruct
{
   COORD position;
   COORD vitesse;
   COORD acceleration;
   int couche;
   FORME forme;
   COORD dimensions;
   float masse;
   float frottement;
   float rebondissement;
   float solidite;
   float attraction;
   float agressivite;
   char* graphique;
   char* texte;
   COULEUR couleur;
   float chronometre;
} OBJET, *PTROBJET;
```

Chaque objet est nécessairement pointé depuis une et une seule cellule afin d'être accessible par le programme.

La propriété « couche » d'un objet a un comportement particulier : Elle représente la présence de l'objet sur plusieurs « étages » superposés comme des calques sur la fenêtre de jeu. La présence de l'objet à l'étage i se notera par la présence d'un 1 dans le i-ème bit de l'écriture binaire de la propriété « couche » de l'objet.

Ainsi, on retrouve par exemple plusieurs cas de figure :

- Couche = 0 : L'objet n'existe pas réellement, bien qu'il soit présent dans la liste chainée. C'est un objet fantôme, son déplacement et ses interactions ne seront pas calculés, il ne sera pas affiché.
- Couche = 2^n : L'objet n'est présent seulement qu'à l'étage n.
- Couche = $\sum_{n=0}^{N} a_n \cdot 2^n$: L'objet est présent aux étages correspondants à $a_n = 1$

Des exemples concrets donnent :

- Couche = 3 : L'objet est présent à l'étage 0 et à l'étage 1, il pourra donc interagir avec les objets présents sur l'une de ces deux couche.
- Couche = 4 : L'objet n'est présent que sur la couche 2.

2.2.3 Identifiant

Le TDA IDENTIFIANT permet d'identifier chaque cellule sans passer par les pointeurs. Ainsi, l'utilisateur aura accès aux cellules en les désignant par leur identifiant, et non par

un pointeur, qui est un type de donnée non accessible par l'utilisateur. Un identifiant contient :

- Une chaîne de caractères « type » qui représente le « nom de famille »
- Une chaîne de caractères « nom » qui représente le « prénom » de chaque objet.
- Un entier « numéro » qui permet d'indexer les objets similaires.

IDENTIFIANT est défini par le code suivant :

```
typedef struct identifiantstruct
{
   char* type;
   char* nom;
   int numero;
} IDENTIFIANT, *PTRIDENTIFIANT;
```

Chaque identifiant est nécessairement pointé depuis une et une seule cellule afin d'être accessible par le programme.

2.2.4 Coord

Le TDA COORD permet de représenter les vecteurs en 2D. Il contient donc un doublet de réels x et y.

```
typedef struct coordstruct
{
   float x;
   float y;
} COORD, *PTRCOORD;
```

2.3 Algorithmes

2.3.1 Principal

L'algorithme principal est très simple, il se compose d'une phase d'initialisation, puis d'une boucle tournant jusqu'à ce que certains paramètres soient réunis, enfin, une phase de fin vient terminer le processus.

```
Fonction principale :
   Lire paramètres
   Lire niveau
   Initialiser
   fini=0
   Tant que fini=0
         Lire touches appuyées
         Calculer les interactions entre objets
         Calculer les mouvements et actions des objets
         Dessiner les objets
   Fin Tant que
   Terminer
```

2.3.2 Lire paramètres

L'algorithme de lecture des paramètres cherche d'abord les paramètres passés au programme s'ils existent, sinon, il demande ces paramètres à l'utilisateur. Il est à noter que le seul paramètre demandé par la version finale de Supanoïd est le nom du niveau qu'il doit lire.

```
Lire paramètres :
   Si nombre arguments passés au programme >= 1
         nom niveau = argument numéro 1
   Sinon
         demander nom niveau à l'utilisateur
   Fin Si
```

2.3.3 Lire niveau

L'algorithme de lecture du niveau se résume à l'exécution du fichier monde.supanoid contenu dans le répertoire correspondant au nom du niveau actuel.

```
Lire niveau :
   Exécuter fichier monde.supanoid
```

2.3.4 Lire touches appuyées

L'algorithme de lecture des touches appuyées est une reprise de l'algorithme présent dans le programme fourni avec le sujet du BE.

```
Lire touches appuyées :
   Selon valeur de la touche détectée par Java :
         Cas touche A
               Exécuter fichier evenements/A.supanoid
         Cas touche B
               Exécuter fichier evenements/B.supanoid
         .
         .
         .
         .
         Cas touche Z
               Exécuter fichier evenements/Z.supanoid
   Fin Selon
```

2.3.5 Calculer les interactions entre objets

L'algorithme de calcul des interactions entre objet parcourt 2 boucles imbriquées. Une première boucle parcourt chaque objet et calcule les interactions possible mettant en jeu un seul objet (Frottement, dégradation et mort). La seconde boucle parcourt, pour chaque objet, la liste des objets suivants et calcule leur interaction avec l'objet actif dans la première boucle.

En ne parcourant, dans la seconde boucle, que la liste des objets suivants, et non la liste entière de tous les objets, on évite de calculer 2 fois l'interaction entre 2 objets, une première fois dans le sens A-B et ensuite dans le sens B-A. On ne calcule ainsi que le sens A-B.

La figure suivante permet de mieux comprendre le balayage des cellules. Les interactions à objet unique calculées sont représentées en vert, les interactions à 2 objets calculées sont représentées en rouge. Les cases blanches représentent des interactions non calculées afin d'éviter les doublements.

	Objet 1	Objet 2	Objet 3	...	Objet n
Objet 1					
Objet 2					
Objet 3					
...					
Objet n					

Illustration 6: Parcours de la liste des objets pour le calcul des interactions

La description algorithmique du calcul des interactions entre objets est la suivante :

```
Calculer les interactions entre objets :
   Pour i allant de 1 au nombre d'objets
         Si objet i présent sur une couche
               Pour j allant de i+1 au nombre d'objets
                     Si objet i et objet j sur la même couche
                           Calculer collision(objet i, objet j)
                           Calculer attraction(objet i,objet j)
                     Fin Si
               Fin Pour
               Calculer frottement(objet i)
               Calculer dégradation(objet i)
               Calculer chronometre(objet i)
         Fin Si
   Fin Pour
```

2.3.6 Calculer les mouvements et actions des objets

Cette fonction parcourt de manière simple la liste des objets et calcule le mouvement de chaque objet dont les propriétés de vitesse et d'accélération, entre autres, ont été affectées par les interactions avec les autres objets. L'algorithme associé est le suivant :

```
Calculer les mouvements et actions des objets :
   Pour i allant de 1 au nombre d'objets
         Si objet i présent sur une couche
               Déplacer (objet i)
         Fin Si
   Fin Pour
```

2.3.7 Dessiner les objets

Cette fonction dessine les objets en fonction de leur couche, en commençant par les

étages les plus faibles, qui paraîtront donc en arrière plan. La fonction parcourt tous les étages et dessine les objets présents sur chacun. Un objet présent à différents étages sera donc dessiné plusieurs fois.

```
Dessiner les objets :
   Pour i allant de 1 au nombre d'étages
         Pour j allant de 1 au nombre d'objets
               Si objet j présent sur la couche i
                     Dessiner (objet j)
               Fin Si
         Fin Pour
   Fin Pour
```

2.3.8 Exécuter fichier

L'algorithme d'exécution d'un fichier parcourt toutes les lignes du fichier et exécute la commande présente sur chaque ligne.

```
Exécuter fichier nomfichier :
   fichierouvert = Ouvrir nomfichier
   Faire
         lire ligne dans fichierouvert
         Exécuter la commande de la ligne lue
   Répéter Tant que la ligne lue est non vide
   Fermer fichierouvert
```

2.3.9 Exécuter commande

Cette fonction exécute une commande passée sous forme d'une chaîne de caractères (Par exemple : « modifierpropriete[dernierobjet[],couleur,GRIS]; »)

L'exécution d'une commande passe par deux étapes :

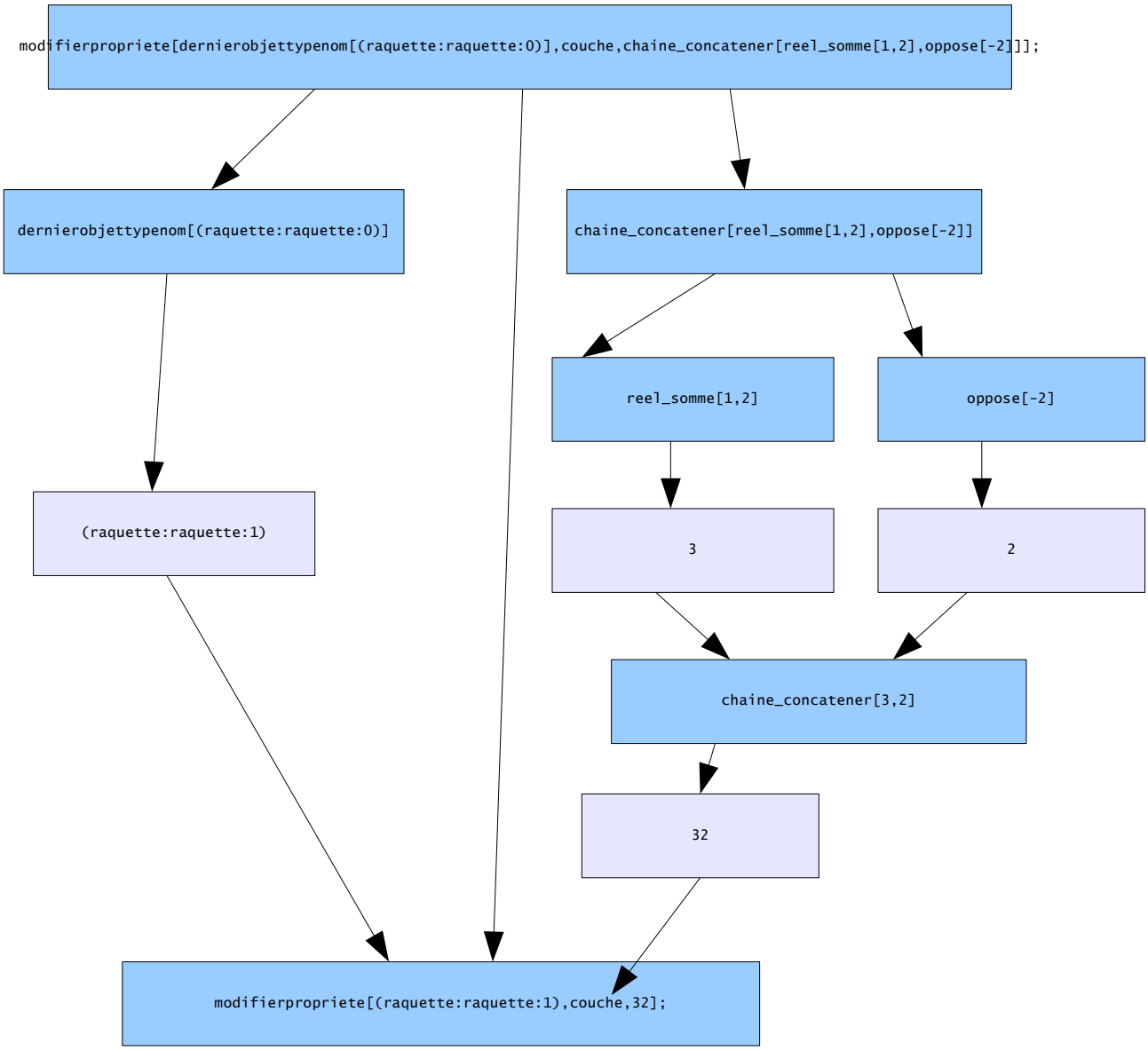
1. La mise en forme de la commande et de ses paramètres.
2. L'exécution de la commande.

La mise en forme de la ligne de commande est réalisée par une fonction qui ne sera pas décrite ici. Cette fonction renvoie en sortie une table de chaînes de caractères. La case 0 du tableau représente le nom de la commande, les cases suivantes contiennent les arguments fournis à la commande dans la ligne de commande.

Lors de l'exécution de la commande, il faut que les arguments fournis soient compréhensibles par l'interpréteur. Si ces arguments sont eux mêmes des sous commandes, alors ils sont exécutés avant d'exécuter la commande. Le processus est donc récursif.

Si la commande demandée n'existe pas dans la bibliothèque des commandes « internes » du programme, alors le programme ouvre le fichier ayant le nom de la commande.

L'interprétation d'une commande suivra le processus décrit par le schéma suivant :



L'algorithme d'exécution d'une commande est le suivant :

```
Exécuter commande lignecommande :
   Mettre en forme la commande dans un tableau
   Si la commande est valide
         Selon valeur du nom de la commande :
               Cas creerobjet
                     Appeler fonction interne creerobjet
               Cas supprimerobjet
                     Appeler fonction interne supprimerobjet
               .
               .
               .
               .
               Cas par  défaut
                     Exécuter fichier « nom_de_la_commande »
         Fin Selon
   Fin Si
   Retourner resultat
```

3 Développement

3.1 Organisation du code

Le code est divisé en plusieurs parties :

- Supanoid.h : Le fichier d'en-tête comprenant les définitions de toutes les fonctions et TDA ainsi que leur explication.
- Graphic.h, key.h, graphic.c : Les fichiers fournis avec le sujet du BE permettant de gérer l'affichage et le clavier.
- Main.c : Le fichier contenant la fonction main.
- Geometrie.c : Le fichier qui permet d'effectuer toutes sortes d'opérations géométriques en 2 dimensions, en s'appuyant principalement sur le TDA COORD pour représenter les vecteurs, et float pour représenter les réels.
- Physique. c : Moteur physique, gestion des interactions entre objets.
- Manipulations.c : Contient les fonctions nécessaires à la manipulation et la modification des différents TDA introduits, ainsi que la navigation dans la liste et la recherche d'éléments suivant différentes méthodes.
- Interface.c : Fichier qui comprend toutes les fonctions liées à l'interface utilisateur, affichage, clavier etc... Ce fichier vient en complément de graphic.h, graphic.c et key.h.
- Commandes.c : Contient les fonctions permettant d'interpréter et d'exécuter les commandes demandées par l'utilisateur.

- Actions.c : Vient en complément de commandes.c pour la réalisation des actions demandées par l'utilisateur.

Le fichier Supanoid.h contenant toutes les définitions importantes, il doit être inclus dans tous les autres fichiers du code source.

3.2 Exécution

Le fichier Supanoid.exe doit être accompagné de l'exécutable Graphics.jar dans le même dossier. Pour pouvoir avoir accès aux mondes, ceux-ci doivent être placés dans un dossier « mondes » placé aux côtés de supanoid.exe et graphics.jar. Le monde consiste en un dossier comprenant au minimum un fichier nommé « monde.supanoid » décrivant le niveau. Il est possible de passer le nom du monde a charger au démarrage en paramètre pour supanoid.exe. Ainsi, la commande :

```
supanoid.exe supaworld
```

Lancera Supanoïd et chargera le niveau « supaworld », c'est à dire, exécutera le fichier « /mondes/supaworld/monde.supanoid ».

l'exécution du programme sans argument provoquera simplement une demande de la part du programme lors de son exécution.

Les images associées à un monde doivent être placées dans le dossier « /images » au côtés de « monde.supanoid ». Les sons seront placés dans le dossier « /sons ».

Un fichier « initialisation.supanoid » peut être placé aux côtés de « monde.supanoid » afin d'exécuter des commande une fois le niveau chargé et prêt à être joué.

3.3 Plan de tests

Pour tester le fonctionnement du programme, nous avons eu recours à plusieurs méthodes :

- Création de programmes « tests » permettant de tester les fonctions critiques sans lancer le jeu.
- Mise en place d'un mode « debug » dans le jeu permettant d'exécuter les commandes une par une et d'afficher leur résultat, afin de diagnostiquer les causes d'éventuels problèmes.
- Mise en place d'un fichier « journal.supanoid » dans lequel il est possible d'imprimer les informations permettant de comprendre les problèmes rencontrés.
- Utilisation de la fonction « débogage » de l'application Dev-C++.
- Exécution du programme en fournissant des information erronées pouvant provoquer le plantage.
- Exécution du jeu en essayant de provoquer les bugs potentiels.

Ces tests nous ont permis de corriger plusieurs centaines de bugs tout au long du développement du jeu afin d'arriver à une version stable mais probablement non exempte de problèmes.

Ces tests ont aussi permis de ressentir le besoin de nouvelles fonctionnalités, qui ont alors été implémentées.

4 Exécutions

4.1 État du programme

La version actuelle du programme (V1.2) fonctionne correctement dans des conditions habituelles d'utilisation. Néanmoins, certains problèmes peuvent subsister car nous découvrons régulièrement de nouveaux bugs, causes de plantages. Cependant, le programme vérifie des conditions minimales de sécurité, il n'écrit pas sur des zones mémoires non appropriées, il ne permet pas de dépassement de mémoire etc...

Nous laissons au lecteur la possibilité d'examiner les traces d'exécution, comme par exemple le fichier journal.supanoid contenu dans chaque répertoire contenant un niveau après l'exécution de celui ci.

4.2 Perspectives

4.2.1 Performances

Le fonctionnement du jeu étant principalement basé sur la lecture de scripts, il est nécessaire d'optimiser la lecture de ceux-ci. En effet, l'ouverture et la fermeture de nombreux fichiers requiert beaucoup de mémoire et résulte en une saturation du système si le niveau est complexe.

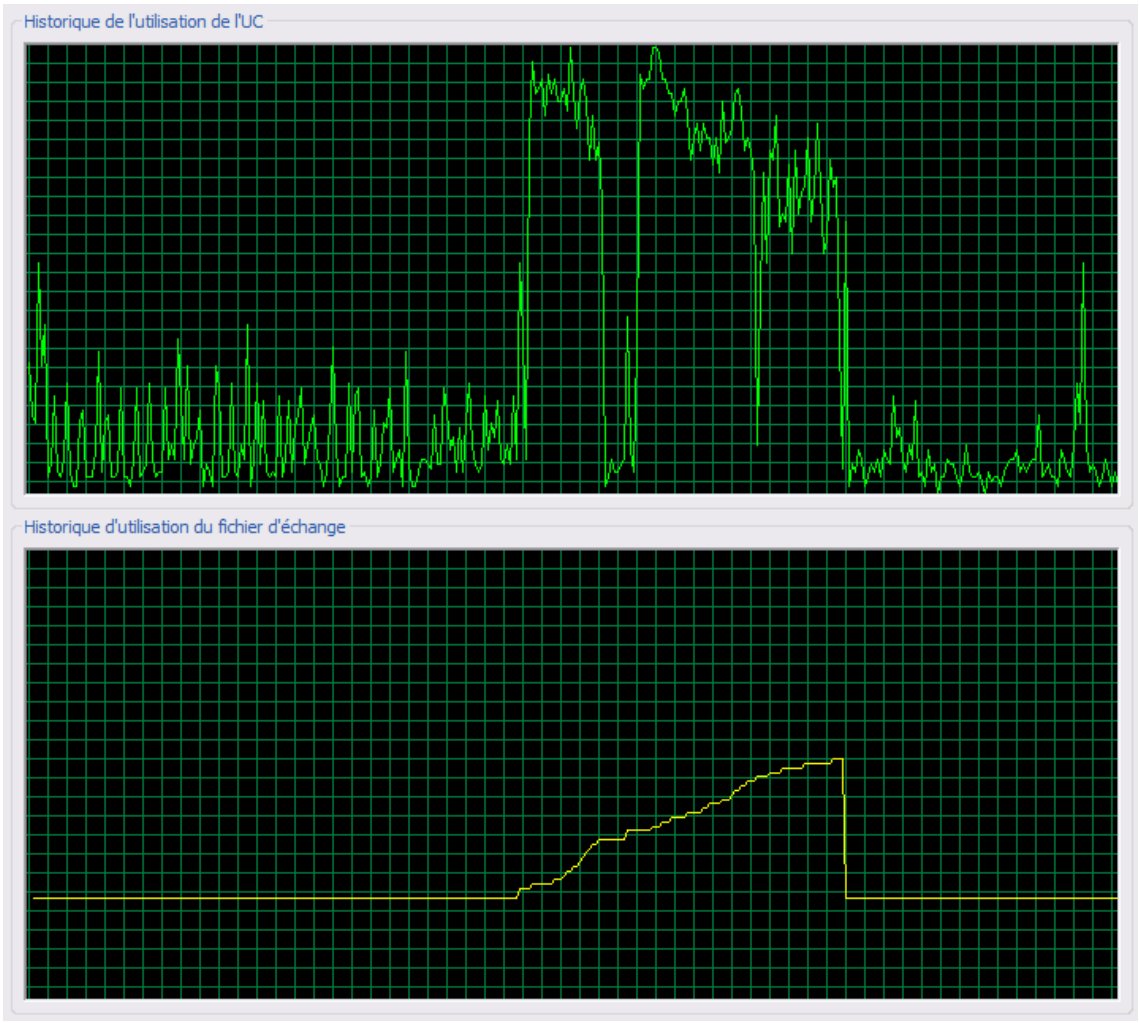


Illustration 7: Utilisation de la mémoire et du processeur lors d'une partie de Supanoïd sous Windows XP en utilisant un niveau demandant beaucoup de ressources. Plusieurs centaines de Mo utilisés !

Il serait donc souhaitable que ces fichiers soient lus par avance et ouverts qu'une seule fois chacun. Plusieurs solutions sont possibles, parmi lesquelles :

- Scanner le répertoire contenant le niveau, lire tous les scripts qui y sont présents et les stocker en mémoire avant le début du jeu.
- Ne pas scanner le répertoire, mais mettre les scripts en mémoire à leur première ouverture afin de ne plus avoir à accéder au disque lors de leurs prochaines utilisation.

4.2.2 Possibilités

De nombreuses amélioration sont envisageables, ont peut citer certains points perfectibles :

- Pour l'instant, les objets physiques du jeu ne peuvent avoir que 2 formes prédéfinies, un cercle, ou un rectangle. L'utilisation de polygones quelconques avait été envisagée au début, mais elle semblait trop complexe. Il pourrait être intéressant de pouvoir définir chaque objet avec un polygone quelconque, ce qui augmenterait les possibilités de jeu.
- De même, les objets ne peuvent pas avoir de mouvement de rotation, il serait

intéressant d'implémenter celui ci, ce qui ne serait possible qu'après avoir défini les objets par des polygones.

- L'animation des objets serait aussi un point intéressant à améliorer, bien qu'elle soit possible actuellement en utilisant des scripts et la propriété chronomètre de chaque objet.
- L'amélioration des possibilités de personnalisation des scripts, en ajoutant de nouvelles fonctions permettant de nouvelles possibilités.
- La possibilité de jouer en réseau serait elle aussi très intéressante.

5 Conclusion

Ce bureau d'étude nous a permis de comprendre les principes de base de la création d'un programme structuré en langage C. Les nombreux problèmes rencontrés nous ont incité à opter pour une méthode de travail fiable. En effet, commencer à programmer après une courte réflexion est satisfaisant à court terme, car les premiers résultats arrivent rapidement, mais cette technique arrive vite à saturation et la résolution des problèmes devient de plus en plus difficile. On comprend rapidement qu'il est plus efficace de travailler avec une phase de réflexion plus importante au départ, avant de produire le code. Cependant, même avec cette méthode, la phase la plus longue de notre projet a été le débogage. La charge de travail nous a semblé être répartie dans les proportions suivantes :

- 20% réflexion
- 10% programmation
- 20% tests
- 20% repérage des causes de problèmes
- 30% réparation des problèmes

Nous estimons qu'un travail parfaitement organisé et réfléchi aurait donné la répartition suivante :

- 40% réflexion
- 20% programmation
- 20% tests
- 10% repérage des causes de problèmes
- 10% réparation des problèmes

La programmation du jeu s'est donc avérée très intéressante et enrichissante en tant que première expérience de développement d'un logiciel dans sa quasi-intégralité.