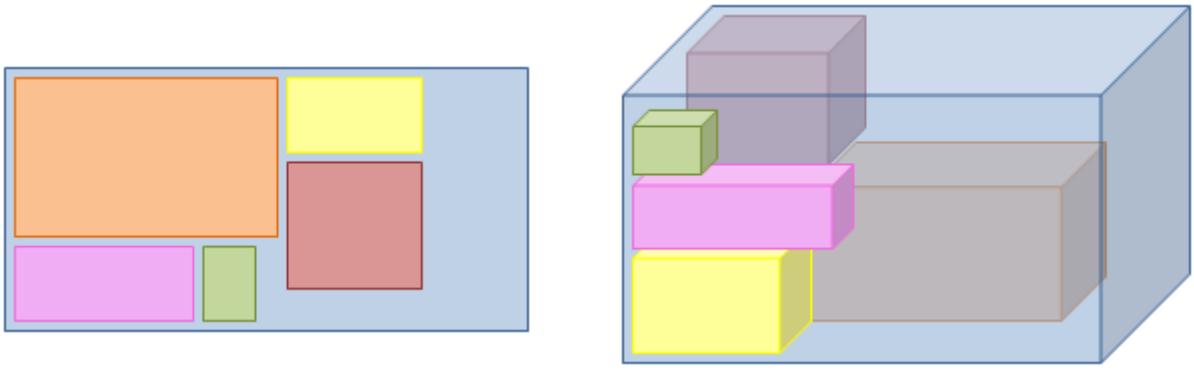
Institut **S**upérieur
d'**I**nformatique de
Modélisation et de
leurs **A**pplications

Complexe des Cézeaux
BP 125
63173 Aubière Cedex

Rapport d'ingénieur de projet de 2ème année
Filière Génie Logiciel et Systèmes Informatiques

Modélisation d'un conteneur en 3D : ISIBox



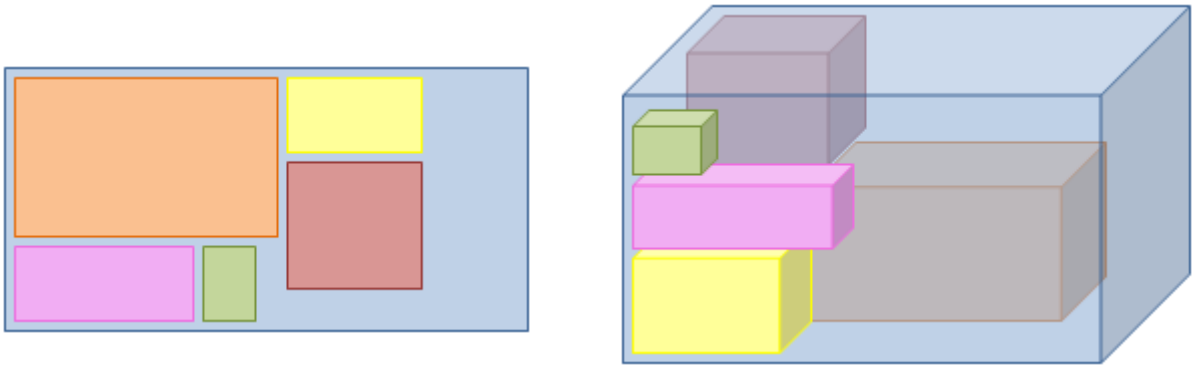
Présenté par : Christophe HAEN et Fabien THOUNY
Responsables ISIMA : Christophe DUHAMEL et Philippe LACOMME

Institut **S**upérieur
d'**I**nformatique de
Modélisation et de
leurs **A**pplications

Complexe des Cézeaux
BP 125
63173 Aubière Cedex

Rapport d'ingénieur de projet de 2ème année
Filière Génie Logiciel et Systèmes Informatiques

Modélisation d'un conteneur en 3D : ISIBox



Présenté par : Christophe HAEN et Fabien THOUNY
Responsables ISIMA : Christophe DUHAMEL et Philippe LACOMME

Remerciements

Nous tenions tout d'abord à remercier Christophe DUHAMEL et Philippe LACOMME, enseignants-chercheurs à l'ISIMA, pour la grande disponibilité dont ils ont fait preuve tout au long de ces 6 mois, ainsi que pour l'aide et les conseils dispensés, sans lesquels ce projet ne serait pas ce qu'il est aujourd'hui.

Merci également à Loïc Yon pour nous avoir fait bénéficier de son expérience avec Java3d.

Pour finir, merci aux personnes qui ont participé à la traduction de notre projet dans les différentes langues (Maria EPERJESIOVA, Corinne FION, Mathieu STEPEC).

Résumé

Pour notre deuxième année à l'ISIMA, nous avons réalisé un projet proposé et dirigé par Christophe DUHAMEL et Philippe LACOMME, enseignants-chercheurs à l'ISIMA.

Le but premier du projet est de fournir un environnement 3D permettant de visualiser le résultat d'algorithmes de résolution du problème de *Bin Packing* appliqué au domaine des transports. Les résultats avec lesquels nous faisons nos tests proviennent du projet de 3ème année de Chen DAI et Caroline LEOTOING, étudiants à l'ISIMA.

Afin de prévoir l'augmentation des besoins d'utilisation, nous avons directement travaillé dans l'optique qu'un tel logiciel sera utilisé par les entreprises de transport pour qui ce genre d'outil permet un gain de temps considérable. De plus, il correspond à un besoin réel et urgent pour les chercheurs travaillant sur le problème du *Bin Packing*. En effet, jusqu'à ce jour, ils disposaient de peu de solutions pour visualiser aisément et rapidement les résultats des algorithmes de *Bin Packing*, ce qui leur faisait perdre beaucoup de temps.

En plus des fonctionnalités graphiques évidentes comme l'affichage, le changement de vue par exemple, nous avons développé des fonctionnalités utiles aux chercheurs (recueil d'informations comme c'est le cas pour la détection des collisions), ou aux potentielles sociétés de transports (par exemple, la possibilité d'associer à chaque caisse des informations comme le destinataire, l'expéditeur ou le poids à chaque caisse).

Nous avons développé notre application en Java et nous sommes servi de l'API Java3D.
Les raisons principales liées à ce choix sont l'apprentissage aisé par rapport à d'autres solutions, et surtout la portabilité quasi-totale inhérente à Java. L'ensemble de notre travail a été réalisé sous Notepad++ , et la version finale de notre projet a été compilée en Java 5 (dernière version en date sous MAC à l'heure actuelle.).

Mots-clé : 3D, Bin Packing, IHM, Java, Notepad++, API

Abstract

For our second year at ISIMA, we worked on a project proposed and led by Christophe DUHAMEL and Philippe LACOMME, assistant professors at ISIMA.

The first goal of the project was to provide a 3D environment allowing high quality 3D representation of Bin Packing solutions. Experiments were achieved with results of Chen DAI's and Caroline LEOTOING's 3rd year project, ISIMA students as well.

To ensure evolutivity, we directly developed our software with the ambition of being used by transport companies. Furthermore, it corresponds to a real and current need of researchers working on the Bin Packing problem. Indeed, they have few solutions available which would allow them to easily and quickly visualize their results. This involved a low of time.

In addition to the basic graphical functions, like display and view change, we developed some extra functions useful to researchers (collect some information, like collisions detection), or to transport companies (for example, the possibility to add information like sender's and receiver's address, weight to each box).

The Java3D API has been used since Java provides high quality multi platform library and easy to use dedicated libraries. All our work was made with Notepad++, and the final version of our project was compiled in Java 5 (the last version available on MAC nowadays).

key-words : 3D, Bin Packing, IHM, Java, Notepad++, API

Table des matières

Table des figures

Introduction

Dans le cadre de notre deuxième année de formation d'ingénieur à l'ISIMA (Institut Supérieur d'Informatique, de Modélisation et de leurs Applications) de Clermont-Ferrand, nous avons effectué un projet de 100 heures, sous la tutelle de Christophe DUHAMEL et Philippe LACOMME, tous deux enseignants-chercheurs à l'ISIMA.

Les études actuelles de Christophe DUHAMEL et de Philippe LACOMME impliquent la résolution du problème du *Bin Packing* appliqué au domaine des transports. Rappelons que le problème du *Bin Packing* - ou problème du *Rangement/placement* consiste à optimiser le rangement d'objet dans un conteneur. Notre projet est né du besoin de nos tuteurs à vixualiser le résultat de leurs optimisations. Le but général de ce projet est donc de développer un outil de modélisation 3D dédié au domaine des transports. Cet outil devra en outre être capable de représenter en 3 dimensions (3D) le conteneur ainsi que les caisses à l'intérieur, mais également de donner la possibilité à l'utilisateur d'interagir avec tous ces éléments.

L'étude des différents cas d'utilisations possibles d'un tel logiciel vont nous permettre d'établir l'ensemble des fonctionnalités qu'il nous faudra développer pour répondre à la demande des utilisateurs, à savoir les chercheurs et éventuellement des sociétés de transports.
Les principaux besoins sont l'ajout, la suppression, le déplacement par translation ou rotation des caisses à l'intérieur du conteneur. Naturellement, des fonctions telles que la lecture des données sont indispensables. Comme nous le verrons plus tard, nous avons ajouté de nombreuses autres fonctionnalités qui donnent un surplus d'informations à l'utilisateur ou qui rendent simplement le logiciel plus ergonomique.

La première grosse partie de notre travail a été de choisir, puis d'apprendre à maitriser les outils informatiques dont nous allons nous servir pour développer notre application. La deuxième partie du projet consiste en une amélioration progressive de notre solution par l'ajout de nouvelles fonctionnalités.

Dans un premier temps nous développerons le cahier des charges d'un tel logiciel, établirons un comparatif des différentes technologies envisageables afin de justifier nos choix et finirons par une analyse de l'existant. La deuxième partie développera plus en détails la conception de ce produit, les difficultés rencontrées et les solutions apportées.

1. Partie 1 : Etude

1.1. Présentation du sujet

Le problème du bin packing, ou problème du rangement/placement, est une problématique majeure dans le domaine de la recherche, puisqu'on trouve des applications concrètes dans de très nombreux domaines. Rappelons d'abord ce problème :

« Étant donné un conteneur dont on connait la capacité maximale, en volume ou en charge par exemple, et étant donné des objets dont on connait la caractéristique propre, un volume ou un poids, quels objets doit on mettre et ou dans le conteneur pour optimiser le rangement, en termes d'espace ou de poids? »
A la base problème en 1D, nous nous intéressons ici à la généralisation du problème en 2D et 3D.

Ce problème trouve parfaitement bien sa place dans le domaine des transports. Nous ne nous intéressons pas ici à sa résolution, mais plutôt à l'étape suivante. En effet, le problème étant difficile à résoudre de manière optimale, il existe des méthodes heuristiques pour déterminer des remplissages de bonne qualité.
C'est ici qu'un logiciel permettant la visualisation 3D du résultat se révèle très utile :

- pour les chercheurs, un tel logiciel permettrait d'identifier ce qui pose éventuellement problème et ainsi améliorer leurs algorithmes et méthodes.
- pour des utilisateurs tel que les sociétés de transport l'intérêt est plutôt de voir la solution, de la modifier aisément, et d'ajouter des informations supplémentaires à chaque boite.

Christophe DUHAMEL et Philippe LACOMME sont confrontés de par leurs recherches à l'absence d'un tel logiciel d'où l'intérêt immédiat de ce projet.

Il est important dans un premier temps de bien définir les besoins, afin de choisir la technologie la plus adaptée pour y répondre. Une contrainte supplémentaire à ne pas négliger est la prise en main par l'utilisateur. Il est loin d'être aisé pour un utilisateur de travailler en 3D sur un écran et il est encore plus difficile pour un développeur de prendre en compte cette difficulté inhérente au monde de la 3D : c'est le problème de l'ergonomie.

1.2. Cahier des charges

Une définition précise des besoins des utilisateurs est indispensable. Grâce à cette étude on pourra établir une liste des fonctionnalités - nécessaires ou optionnelles – ,mais également choisir les solutions technologiques les plus adaptées à notre projet et se faire une idée de l'organisation générale de l'interface utilisateur.

Pour obtenir toutes ces informations, il convient de distinguer trois catégories d'utilisateur.

- L'utilisateur de base : ce cas correspond au cas d'utilisation le plus simple du logiciel. Le seul but de cet utilisateur consiste simplement à visualiser dans son ensemble le conteneur et les caisses placées dedans.

- L'utilisateur intermédiaire : en plus d'avoir une vision globale, cet utilisateur veut avoir une vision plus détaillée des éléments afin de tirer plus d'informations sur leur placement. C'est dans cette catégorie que se classent les chercheurs désireux de vérifier leurs algorithmes.

- L'utilisateur avancé : en plus des fonctionnalités de vue, cet utilisateur va vouloir agir sur les éléments. Il s'agit ici plutôt des entreprises.

On peut donc déjà déduire une liste exhaustive des fonctionnalités strictement nécessaires à tous ces utilisateurs :

- Entrées-sorties : il est crucial pour un utilisateur de pouvoir charger simplement un conteneur avec ses informations sans avoir besoin de le créer manuellement.
- Mouvements de caméras : afin d'obtenir des informations plus précises sur le placement des différents objets, les caméras doivent être mobiles. On doit donc penser à implémenter des fonctions de rotation de caméras – pour « tourner autour » des objets – et de translations -pour créer un effet de zoom .
- Fonctionnalités sur les éléments : on doit pouvoir ajouter, supprimer, translater ou pivoter les caisses.

Nous avons développé d'autres fonctionnalités dans le but de donner plus d'informations à l'utilisateur ou encore d'améliorer l'ergonomie de l'application :

- Collisions : nous avons mis en place un algorithme de détection des collisions en temps réel afin que l'utilisateur puisse se rendre compte d'éventuelles incohérences dans le placement des boites.
- Annuler/rétablir : ces deux fonctionnalités sont désormais omniprésentes dans tous les logiciels et se révèlent particulièrement utiles à l'usage.
- Export : l'utilisateur peut exporter ce qu'il voit à l'écran soit sous forme de fichier image (PNG) ou alors directement vers l'imprimante.
- Internationalisation : choix de la langue de l'IHM

De plus, nous avons ajouté une fenêtre des préférences, qui permet à l'utilisateur de choisir le répertoire par défaut qui lui conviendra, ainsi que la langue du logiciel. Jusqu'à présent, siw langues sont disponibles : le français, l'allemand, l'espagnol, l'anglais, le tchèque et le slovaque.

1.3 Les outils

Il nous fallait trouver un support de programmation nous permettant de répondre à la problématique. Ce support se devait tout d'abord d'être relativement aisé à appréhender, et ensuite d'être apte à concevoir un environnement virtuel en 3 dimensions.
Une première recherche effectuée sur ces critères nous amena a deux des langages principaux C++ et Java.
Toutefois, aucun des deux ne gère nativement un environnement en 3 dimensions. Pour cela, ils recourent à des librairies tierces :

- Qt (environnement fenêtré) et OpenGL pour le langage C++ (voir figure 1-1)
- L'interface de programmation (API) Java3D pour Java (l'environnement fenêtré étant géré par les composants internes AWT/Swing) (voir figure 1-2).

Tout d'abord, parlons de l'apprentissage des 2 solutions :

- La librairie OpenGL impose une programmation de bas niveau, en effet, l'affichage de simples primitives géométriques tel un parallélépipède rectangle habillé d'une texture identique sur chaque face demande de redéfinir chaque face, point par point, et d'appliquer à chacune d'elle la texture commune.
- Comparativement, Java et Java3D proposent des classes dédiées à la création de primitives géométriques. Il ne reste à renseigner que les dimensions et la texture de l'objet.

Dans le même esprit, la programmation orientée objet permet une plus grande modularité du code et donc une extensibilité et une maintenance plus facile. Bien sûr, on peut définir de nouvelles classes avec Qt et OpenGL. Le gain serait une meilleure maitrise du code, et une adaptation totale au problème, mais l'effort de développement serait bien supérieur.

Passons maintenant au chapitre de la portabilité. Les deux solutions envisagées ont l'immense mérite d'être portables sur la plupart des systèmes d'exploitations existants, Windows, Linux ou MacOS pour ne citer qu'eux. Cependant, la portabilité n'est pas définie au même niveau :

- la solution basée sur le C++ privilégie la portabilité du code, ce qui signifie qu'on n'écrira le code qu'une seule fois, mais qu'il faudra le compiler et générer un exécutable pour chaque système d'exploitation.

- La solution Java, elle-aussi, ne nécessite qu'une seule écriture du code mais, contrairement à l'autre solution, les fichiers finaux sont les mêmes pour chaque système d'exploitation une fois le code pré-compilé.

Au vu de ces différents arguments, la solution retenue fut finalement celle utilisant Java+Java3D, pour sa prédisposition à une programmation orientée objet sur l'intégralité du projet et pour sa grande portabilité.

A noter tout de même que les deux solutions envisagées n'étaient pas radicalement différentes, loin de là : Java3D se charge en fait d'effectuer la traduction du code java de haut niveau vers les API graphiques de bas niveau que sont DirectX et OpenGL. Mais ceci reste caché au développeur.

Figure 1-1 : Une fenêtre et une caisse avec texture avec Qt + OpenGL (275 lignes)

Figure 1-2 : Une fenêtre et une caisse avec texture avec Java + Java3d (150 lignes)

1.4. Etude de l'existant

De nombreux logiciels peuvent réaliser une partie du travail que nous demandent Christophe DUHAMEL et Philippe LACOMME. Nous pouvons citer par exemple Blender (voire figure 1-3) ou encore 3DSMax (voire figure 1-4) pour la visualisation et l'édition 3D.

Figure 1-3 : *Le logiciel Open Source Blender, un ensemble d'outils pour la 3D*

Blender : *site web :* http://www.blender.org/
téléchargement : http://www.blender.org/download/get-blender/
documentation : *http://www.blender.org/education-help/*

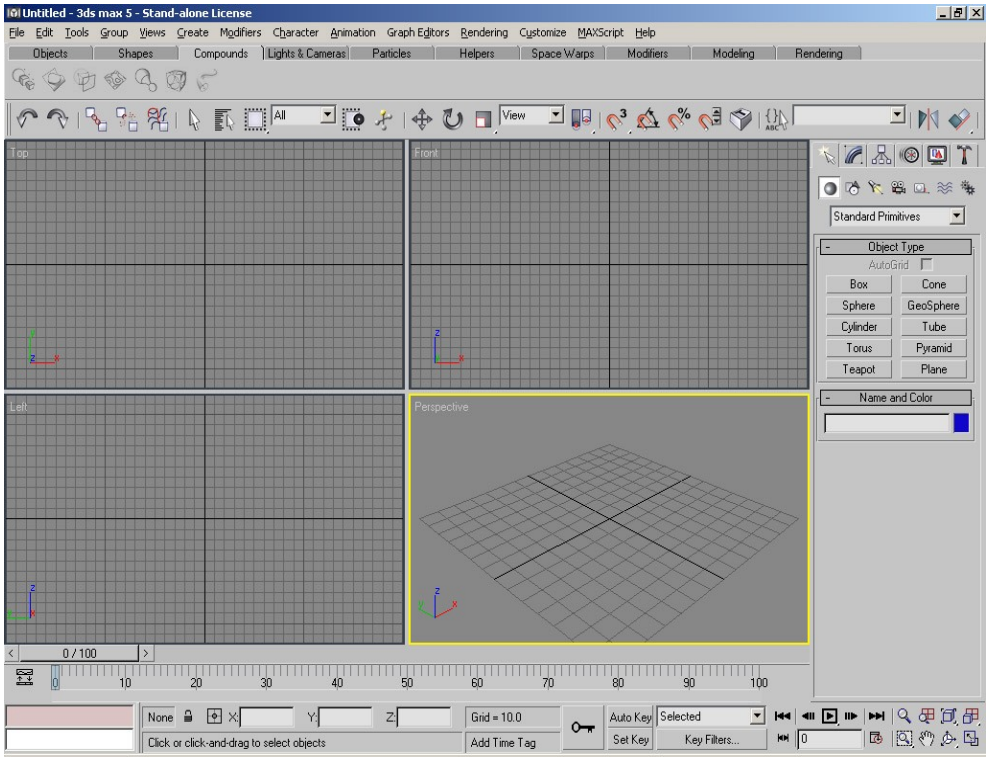


Figure 1-4 : Le logiciel 3DSMax par Autodesk

site web : http://www.autodesk.fr/adsk/servlet/index?id=10611681&siteID=458335

En plus de ne pas répondre au problème du bin packing, on peut s'apercevoir que ces logiciels ont deux gros défauts qui font qu'ils ne sont pas la solution à notre cahier des charges :

- Le besoin en ressources : ce genre de logiciels demande une très grande puissance de calcul, même lorsqu'il s'agit de créer de simples parallélépipèdes. Un ordinateur avec une bonne configuration est donc nécessaire.
- La difficulté de prise en main : la prise en main de tels outils ne se fait pas aisément, et passe par de longues heures de pratique, souvent à l'occasion de stages.

Cependant, on peut tirer certaines leçons très importantes de ces deux exemples :

- La présentation des vues : la vision en 3D sur un écran n'est pas quelque chose d'inné chez l'être humain, comme l'illustre la figure 1-5. Il est donc important de lui donner des repères dans lesquels il se sente plus à l'aise, à savoir des vues en 2D. Nous avons choisi de diviser l'écran en plusieurs parties.
  Ceci présente donc deux grands avantages :
  - c'est une aide très précieuse pour les personnes ayant quelques difficultés à travailler dans l'espace
  - c'est une source d'informations, puisqu'on voit plus clairement certaines informations.

  De plus, la vue 3D n'est finalement que l'intersection de ces 3 vues 2D.

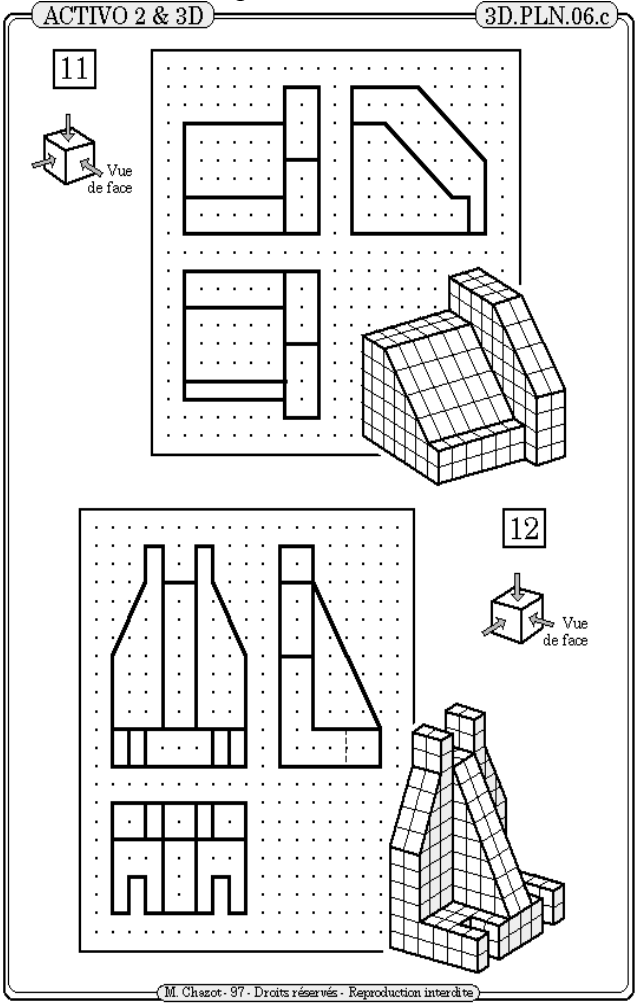


Figure 1-5 : Illustration des projections orthogonales-plan

- L'interaction avec les objets et la scène : nous nous sommes aperçus qu'un principe commun à tous ces logiciels est d'utiliser le clavier pour interagir avec la scène ou l'ensemble des objets et d'utiliser la souris pour une interaction avec un objet en particulier. Bien entendu, ces logiciels sont très aboutis et ils proposent différents types de curseurs afin de pouvoir manipuler la scène avec la souris et surtout savoir quelle est l'opération en cours (sélection, suppression, déplacement ou autre). Nous avons choisi d'adopter ce principe mais pas le système multi-curseur (pour des raisons de simplicité d'utilisation) : si le pointeur est positionné sur un élément, alors les actions seront celles de l'élément sélectionné; sinon les actions influeront sur la scène.

- L'arborescence des objets (voir figure 1-6): donner une arborescence des fichiers présents sur la scène offre deux avantages. Le premier est un confort visuel : il est aisé de savoir si un objet est présent, sans avoir à le chercher dans la scène. Le deuxième est un confort d'utilisation : il peut être difficile d'accéder à certains objets dans la scène, alors que la sélection dans une arborescence est très aisée.

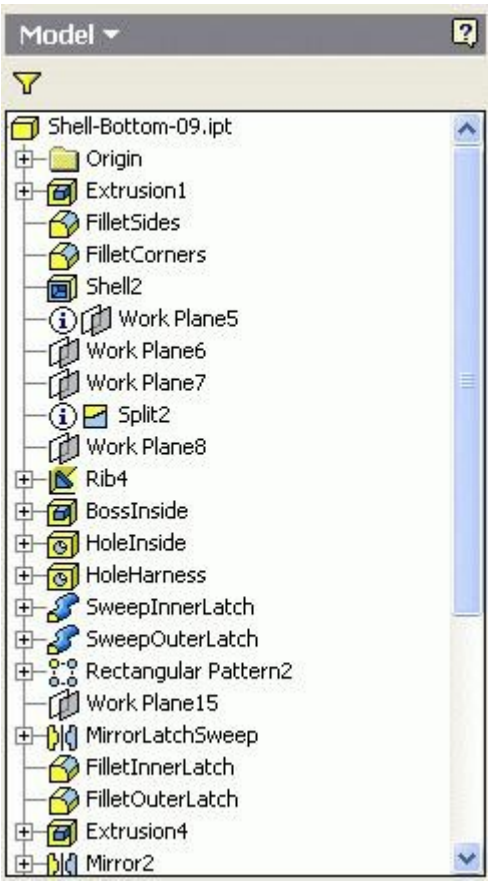


Figure 1-6 : Arborescence d'un projet Inventor

Nous avons maintenant vu les principales contraintes imposées pour ce projet, nous avons fixé les outils que nous allons utiliser et nous avons pu tirer des leçons et des conseils des outils déjà disponibles sur le marché. Voyons maintenant plus en détail les principaux points du développement de notre application, les problèmes rencontrés et les solutions que nous avons apportées.

2. Partie 2 : Développement

Une application 3D repose sur de très nombreux points charnières, et il nous serait impossible de parler ici de la totalité de ces points. Certaines parties du développement se montrent néanmoins bien plus intéressantes, et c'est là dessus que nous allons nous focaliser.

Nous commenceront cette partie en expliquant le principe de fonctionnement de Java3D. Nous détaillerons ensuite 3 des principales fonctionnalités de notre projet, à savoir la sélection d'un objet et les transformations qu'on lui fait subir, la détection des collisions, et pour finir, l'ouverture et l'enregistrement de fichier.

2.1. Fonctionnement de Java3D

Java3D est une API de Java destinée à la programmation en 3 dimensions (3D). A ce titre, elle fournit des classes et des méthodes permettant une mise en œuvre aisée de tous les éléments composant un univers 3D.

Pour chaque nouvelle scène, elle permet donc de définir un repère absolu qui permettra de placer les différents objets 3D(conteneur, caisse, etc) qu'on ajoutera par la suite.
A chaque objet est donc associé une matrice, qui contient les informations sur la position de l'objet par rapport à l'origine, mais aussi les rotations effectuées autour des axes du repère.
Mais il faut aussi visualiser la scène ainsi obtenue et la présenter à l'écran : pour cela on utilise un objet représentant une caméra que l'on place dans l'espace comme tous les autres objets.
La position et l'orientation de la caméra définissent donc ce qui est vu à l'écran.

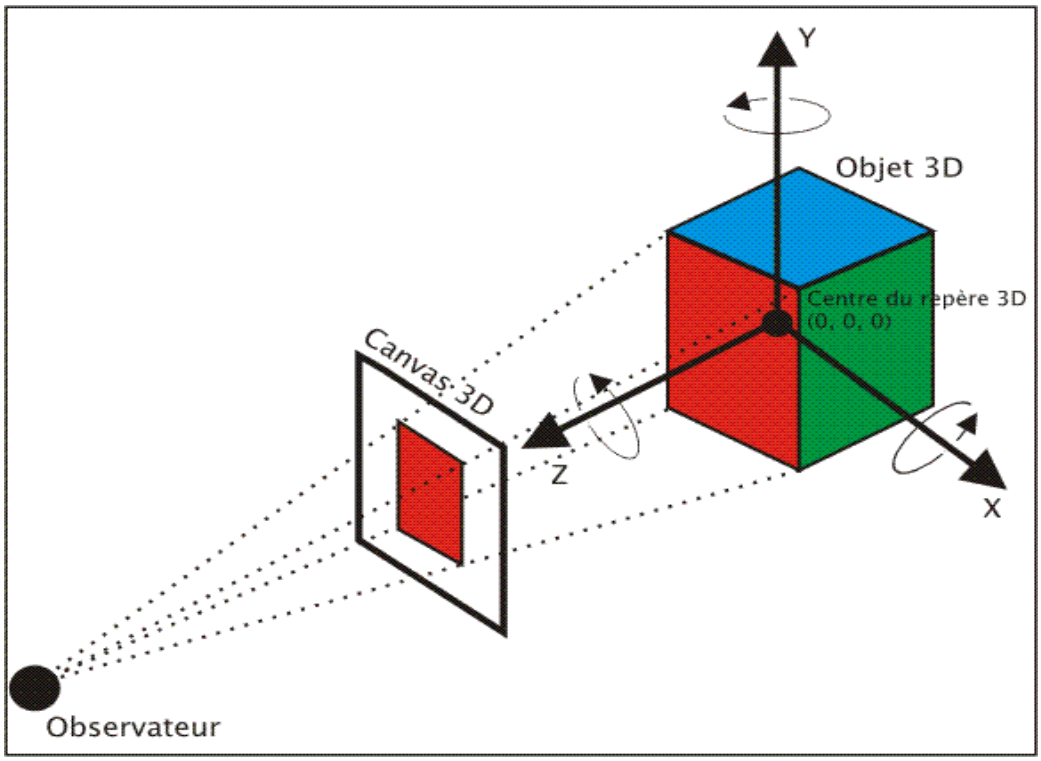


Figure 2-1 : principe de Java3D

Ensuite, il faut que l'utilisateur puisse interagir avec le scène par l'intermédiaire de son clavier et de sa souris. Pour cela, il est nécessaire de définir des comportements qui caractérisent la réaction du système à tel ou tel événement. Par exemple, un mouvement de souris peut engendrer une rotation de la caméra, et le comportement permet de faire le lien entre l'évènement et la caméra à déplacer.

Finalement, une scène 3D, se représente sous forme de graphe, avec à la racine, l'univers, et une multitude de fils composés de :

- objets 3D visibles dans la scène
- caméras servant à visualiser la scène. On peut en utiliser plusieurs pour avoir des angles de vues différents.
- comportements assurant l'interaction entre le clavier et la souris de l'utilisateur et la scène 3D.

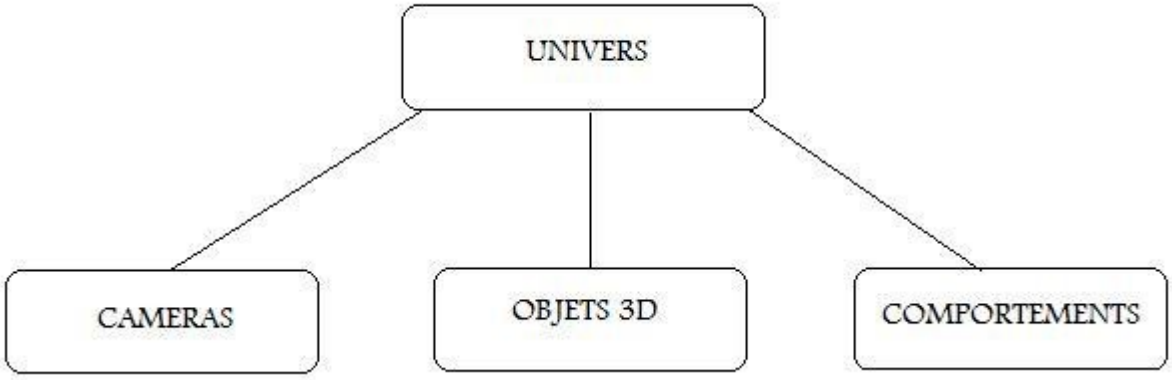


Figure 2-2 : principaux objets de Java3D

Intéressons nous maintenant aux différents comportements nécessaires pour notre application.

- rotation de caméra autour de l'origine
- sélection d'un objet
- déplacement d'un objet

Rotation de caméra autour de l'origine

A ce comportement est associé 2 évènements, en effet, nous avons choisi de pouvoir modifier la position de la caméra, par l'intermédiaire des flèches du clavier, mais aussi lorsque qu'on déplace la souris en maintenant le bouton droit appuyé.

- Commençons par les flèches du clavier :

On définit un pas de rotation d'un angle élémentaire, appelé α dans la suite, pour chaque appui sur une flèche du clavier.
A l'origine, les axes du repère sont orientés comme indiqués sur la figure précédente, la caméra et l'image rendue étant donc dans un plan parallèle au plan (O,x,y).

Intuitivement, L'appui sur la flèche de gauche ou la flèche de droite déclenchera donc une rotation de α autour de l'axe y, l'appui sur la flèche du haut ou la flèche du bas, une rotation de α autour de l'axe x.

Pour fixer les idées, prenons maintenant α = 90°.
Un appui sur la touche du haut effectuera une rotation de 90° autour de l'axe x :

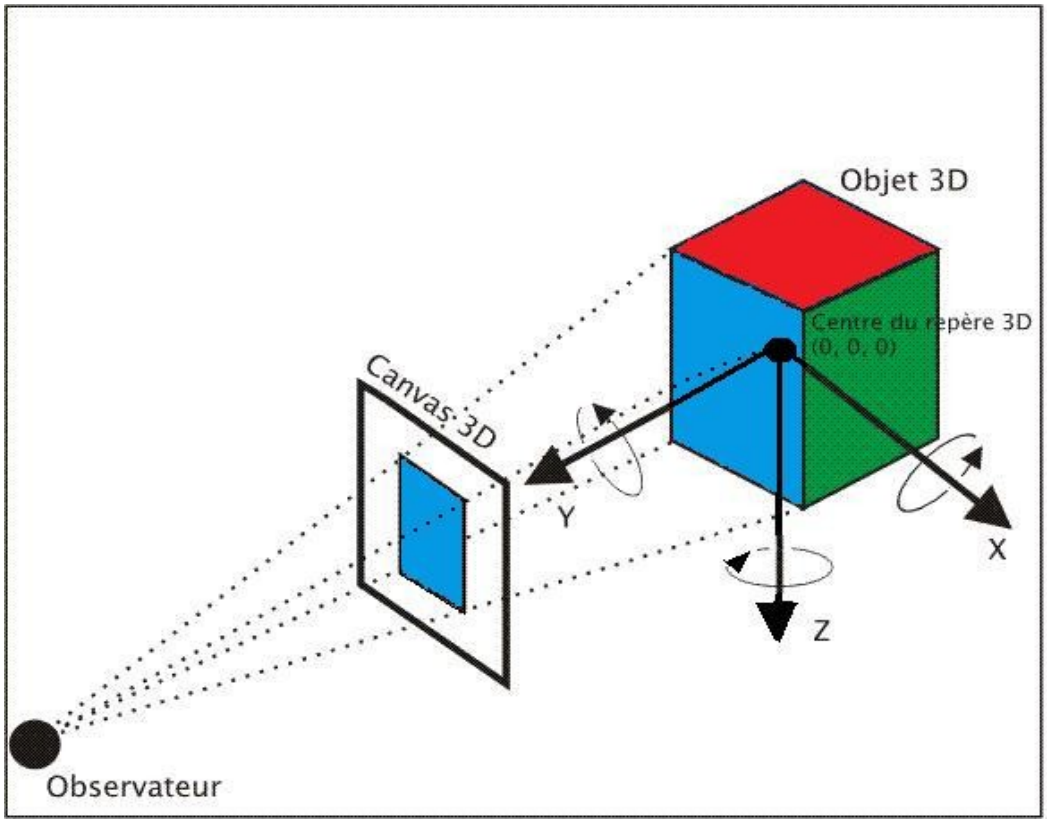


Figure 2-3 : scène de Java3D après une rotation de 90°

En examinant cette figure de plus près, on se rend compte que dans l'état actuel un appui sur la touche de gauche ou de droite, pour rester conforme à l'intuition de l'utilisateur, doit effectuer une rotation autour de l'axe z alors qu'avant la rotation, c'était l'axe y qui était concerné.
Qui plus est, on remarque aussi qu'une rotation d'un angle positif selon y dans l'ancien repère correspondra à une rotation d'un angle négatif selon z dans ce nouveau repère.

Si on extrapole, on peut dire que la rotation à effectuer dépendra donc de la position actuelle de la caméra.

La prise en compte de cela se fait en appliquant un changement de repère à la rotation élémentaire (selon x ou y) qui devient ainsi conforme à l'orientation actuelle du repère.

On peut ensuite appliquer la rotation ainsi calculée pour obtenir la nouvelle position de la caméra.

- Passons maintenant à la rotation de la caméra à la souris.

Après examen, les différences avec la rotation avec les flèches sont :

- une rotation composée de deux rotations élémentaires, une selon x et une selon y.
- l'angle variable de rotation défini par le mouvement plus ou moins ample de la souris. Imaginons, par exemple, que la souris se déplace de 100 pixels selon x. On applique alors un coefficient à ce nombre, pour obtenir l'angle de rotation correspondant.

Mais la méthode, au final, reste la même que celle utilisée pour la rotation avec les flèches.
On effectue le changement de repère sur les deux rotations élémentaires correspondant au mouvement de la souris puis on applique ces deux rotations pour obtenir la nouvelle position de caméra.

2.2. Sélection d'un objet et transformations

Un des points névralgiques de l'interface homme-machine est la possibilité de sélectionner un objet 3D à la souris. Pour cela une série d'actions se révèle nécessaire :

- récupérer la position en pixels de la souris à l'écran
- traduire la position de la souris dans le repère 3D. On rappelle que la caméra est aussi un objet de la scène, elle fournit donc une méthode permettant cette transformation.
- Calculer l'équation de la droite partant de l'œil fictif de l'utilisateur et passant par le point obtenu précédemment.
- Relever tous les objets 3D en intersection avec cette droite.
- Sélectionner l'objet le plus proche de l'utilisateur.

Heureusement, Java3D propose des comportements facilitant cette sélection, et les actions qui y sont associées, par l'intermédiaire de classes liées aux caméras.
L'implémentation s'est donc révélée des plus aisées.

Le déplacement d'un objet

Nous avons choisi pour l'utilisateur un mode de déplacement intuitif consistant en un « glisser – déposer », l'interaction se faisant par maintien du bouton gauche de la souris appuyé.

Cependant, une fois l'objet sélectionné, une problématique demeure : comment, signifier à l'objet son nouvel emplacement une fois la souris déplacée? Autrement dit, comment faire pour que l'objet suive la souris à l'écran ?

La solution apparaît finalement dans la description des actions nécessaires à la sélection d'un objet.

Les 3 premières étapes sont identiques et donnent la droite partant de l'œil fictif de l'utilisateur et passant par la position de la souris dans l'espace.

Maintenant, nous avons postulé que les objets ne pouvaient se déplacer que dans un plan parallèle à la caméra.

Ainsi, pour obtenir la nouvelle position de l'objet, il suffit de calculer les coordonnées du point d'intersection entre la droite décrite précédemment et ce plan de déplacement.

Si on place l'objet à cette position, l'œil, le pointeur de la souris, et l'objet seront donc alignés.
La répétition de cette opération, à chaque mouvement de souris, permet de garder constamment l'objet en-dessous.

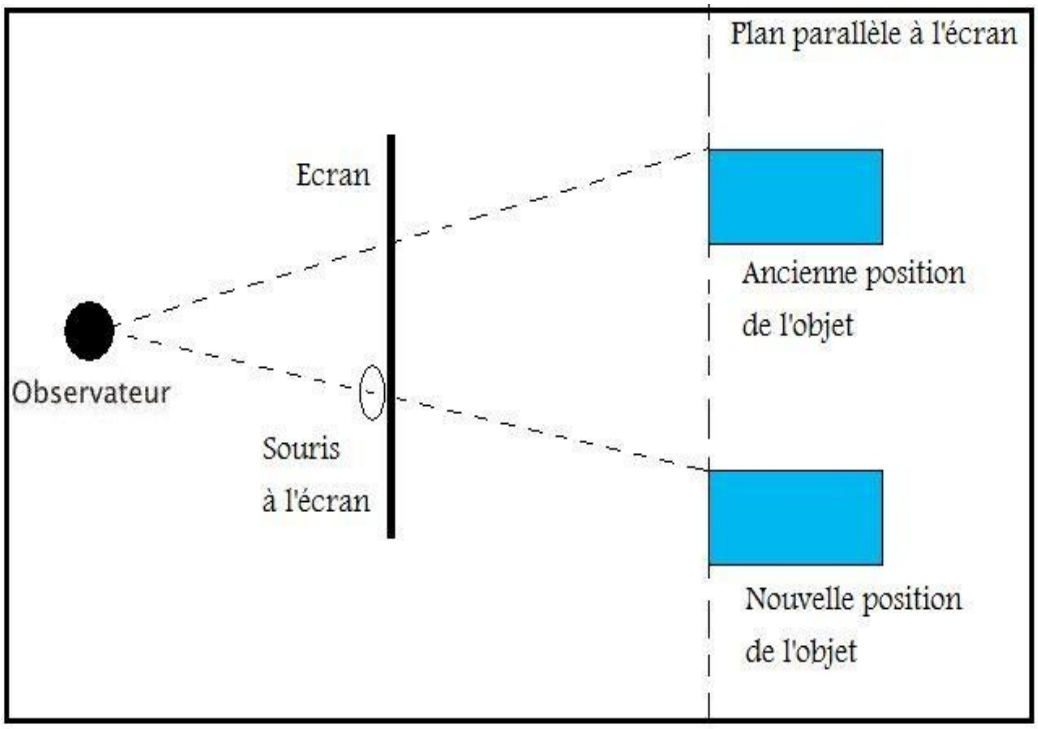


Figure 2-4 : principe du déplacement d'un objet

2.3. Détection des collisions

La détection est sans doute la partie qui nous a demandé le plus de travail en termes de recherche algorithmique. En effet, Java3D dispose d'un outil de détection de collisions, mais il ne répond pas à nos besoins en de nombreux points. Il a donc fallu écrire un algorithme parfaitement adapté. Bien que cette fonctionnalité ne soit pas cruciale, elle est d'une grande utilité pour tous les utilisateurs.

2.3.1 Principe mathématique

Le principe de notre algorithme repose sur la constatation mathématique suivante :

«deux objets sont en collision dans l'espace
si leurs 3 projections orthogonales dans le plan le sont »

Mais c'est sa contraposé qui est effectivement utilisée :

« si pour deux objets dans l'espace,
on peut trouver au moins 1 projection orthogonale dans le plan
où ils ne sont pas en collision, alors ils ne sont pas en collision dans l'espace »

Les deux images qui suivent sont un bon exemple de ce principe.

En effet, sur la figure 2-5, on voit que sur les vues 2D du dessus et de face (respectivement en haut à droite et en bas à gauche), les caisses ne sont pas en collision, ce qui implique que les caisses ne sont pas non plus en collision sur la vue 3D (en haut à gauche). C'est le principe contraposé.

Sur la figure 2-6, c'est le principe direct qui s'applique. On peut voir que les 2 caisses sont en collision sur la vue 3D, et on peut vérifier que l'on retrouve une collision sur chacune des projections orthogonales.

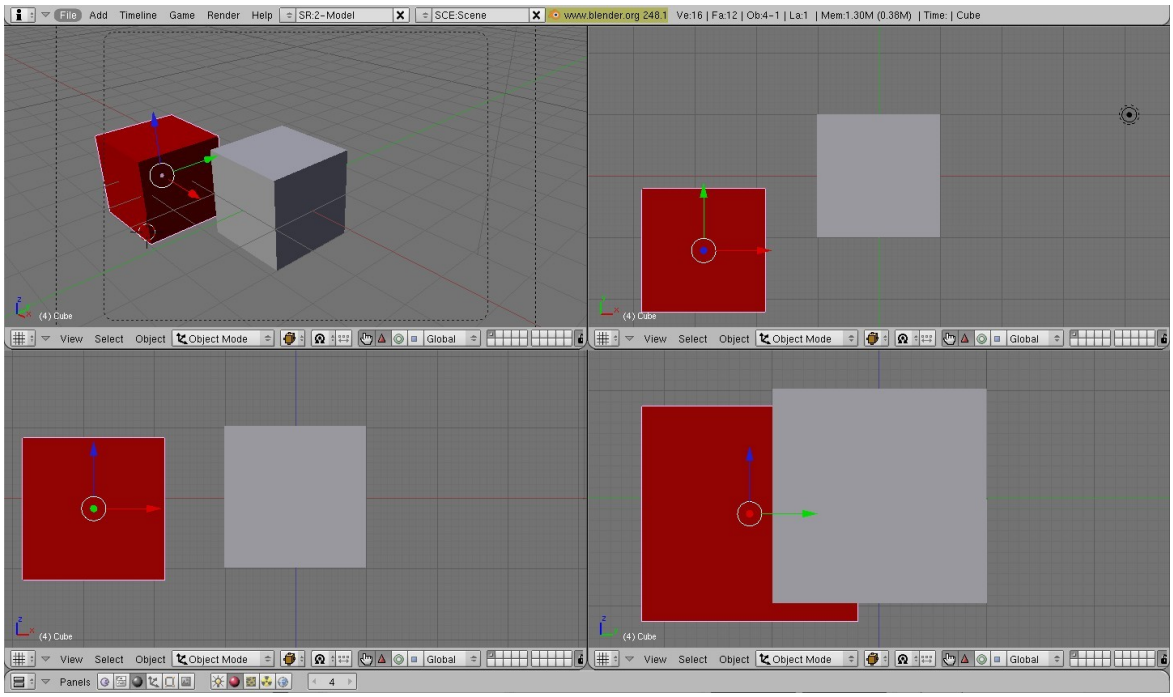


Figure 2-5 : 2 caisses sans collision 3D

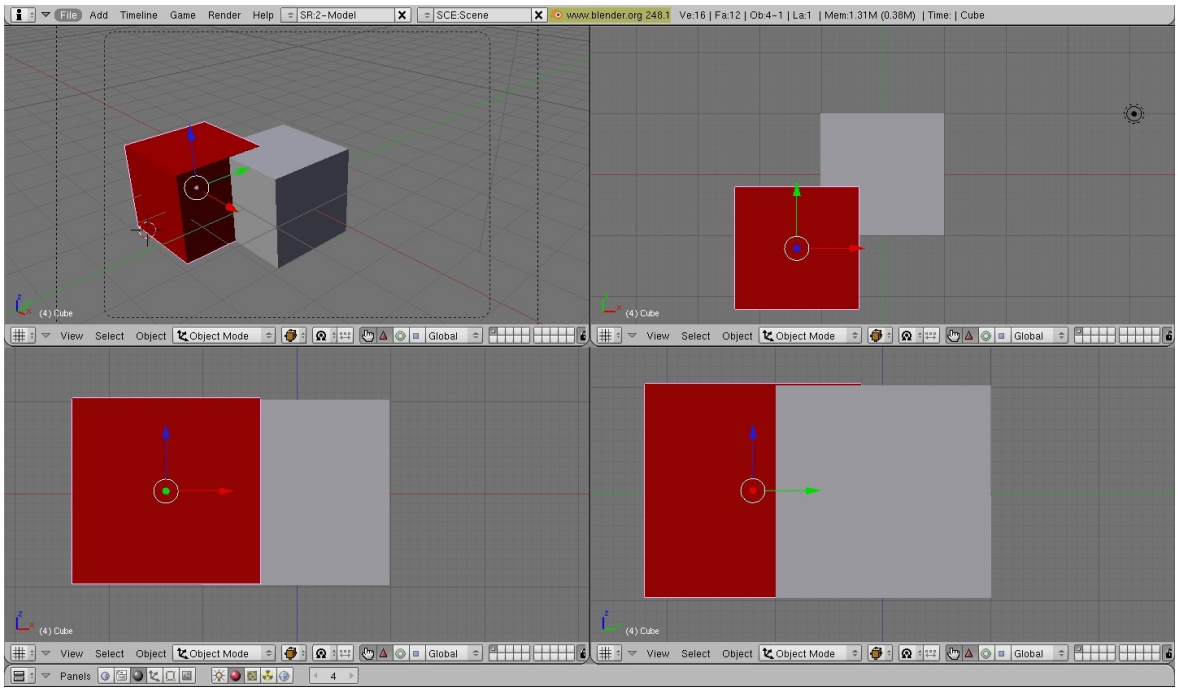


Figure 2-6 : 2 caisses en collision 3D

Maintenant que nous avions le principe général, il fallait élaborer l'algorithme qui correspondait à ce principe. Cependant, nous voulions, on plus de détecter la collision entre deux objets statiques, détecter la collision en temps réel d'objets en mouvement. Cela impliquait une contrainte supplémentaire sur notre algorithme. Il fallait en effet que celui ci soit suffisamment performant pour que les calculs déterminant la collision ne soit pas trop couteux, et perturbent ainsi les calculs de déplacement des objets, entrainant ainsi une sensation de « saccade » à l'écran.

2.3.2 Les éléments de l'algorithme

Les éléments qui vont permettre de répondre à ces contraintes vont être les suivants :

- Les éléments de position : pour chaque objet dans la scène, il existe 6 éléments de positions, 2 pour chaque axe. Les éléments de positions sont composé comme suit :

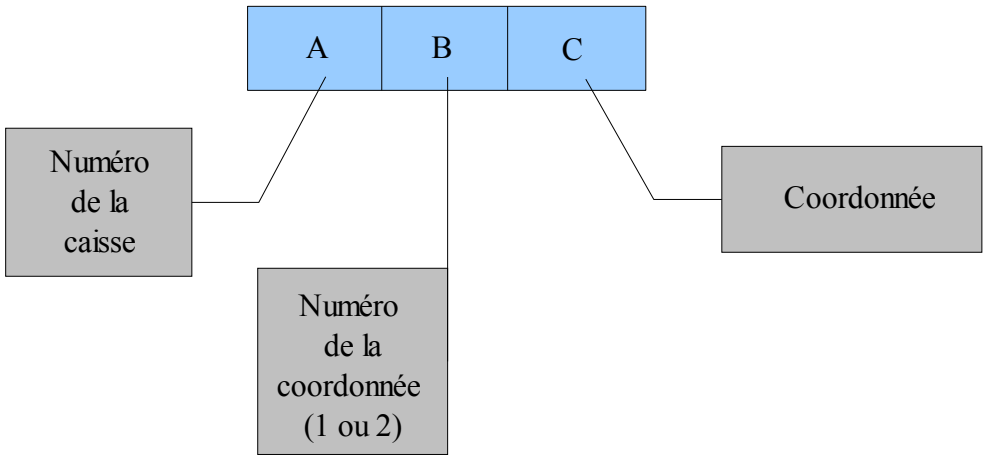


Figure 2-7 : Description de la structure d'un élément de position.

- Les listes de position : il existe 3 listes de positions, chacune correspondant à un des axes. Chaque liste contient 2 éléments de positions par objet présents dans la scène. Les listes de positions sont triées en fonction des coordonnées. Elles sont absolues, c'est à dire qu'elle ne dépendent pas d'un objet en particulier, mais de l'ensemble de la scène.

- Les éléments de mouvement : nous reviendrons plus tard sur la manière dont ils sont construits. Pour le moment, intéressons nous seulement à la structure

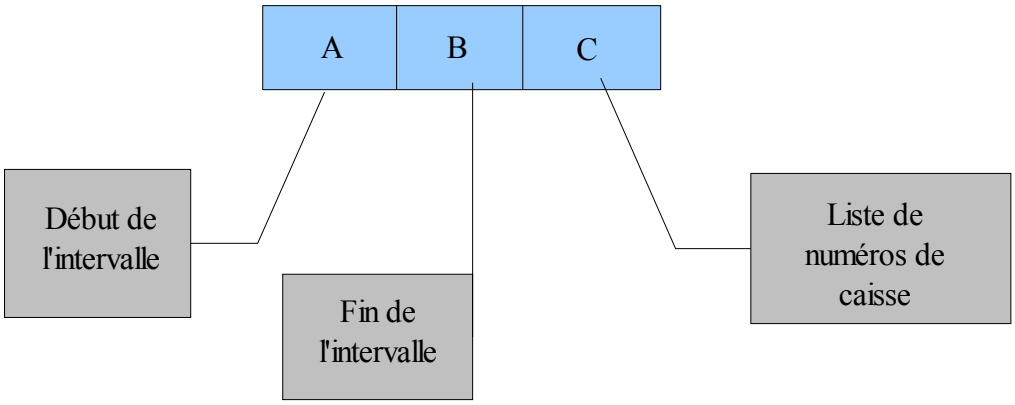


Figure 2-8 : Description de la structure d'un élément de mouvement.

- Les listes de mouvement : comme pour les listes de positions, il en existe une une par coordonnées. Cependant, les listes de mouvements sont relatives à un objet de la scène. Elles sont triées selon les intervalles des éléments de mouvements. De plus, les intervalles sont contigües (la fin de l'intervalle de l'élément N correspond au début de l'intervalle de l'élément N+1).

2.3.3 La construction et l'utilisation des éléments

Cet algorithme est possible uniquement car les les arrêtes des caisses ne sont pas une composition linéaire des 3 axes, mais uniquement suivant un seul des axes. Autrement dit, les caisses sont toujours soit parallèles, soit perpendiculaires aux parois du conteneur, et jamais en diagonale.

2.3.3.a. Les listes de position

Les listes de positions sont initialement vides, et sont remplies au fur et à mesure que l'on crée de nouvelles caisses, et vider au fur et à mesure de la suppression des caisses. Ces listes ne changent pas pendant les mouvements des caisses, mais sont mises à jour à la fin du mouvement.

A chaque ajout d'une caisse, on crée les 6 éléments de position correspondant, et on les insère dans la liste adéquat, en respectant bien l'ordre croissant des coordonnées.

On déduit de notre contraposé mathématique que l'objet ne sera pas en collision si on arrive à trouver une des listes dans laquelle les 2 éléments de positions de l'objet sont consécutifs.

Cependant, certains cas particuliers compliquent un peu l'algorithme, et force à multiplier le nombre de test. C'est le cas par exemple de deux caisses l'une dans l'autre, ou de caisses superposées, ou même accolées.

Les traitements concernant ces cas ne sont pas fait lors de l'insertion dans la liste, mais lors du test de collision. Cela nous permet d'insérer un élément de position, sans se soucier de toutes les places qu'il pourrait potentiellement occuper. Cependant, il faut prendre soin lors du test de collision de vérifier ces différentes possibilités par ordre décroissant de probabilité. On économise ainsi du temps de calcul.

Voyons comment se passent ces étapes sur un exemple. Afin de simplifier les schémas, on peut, sans nuire à la généralité, travailler sur des caisses en 2 dimensions.

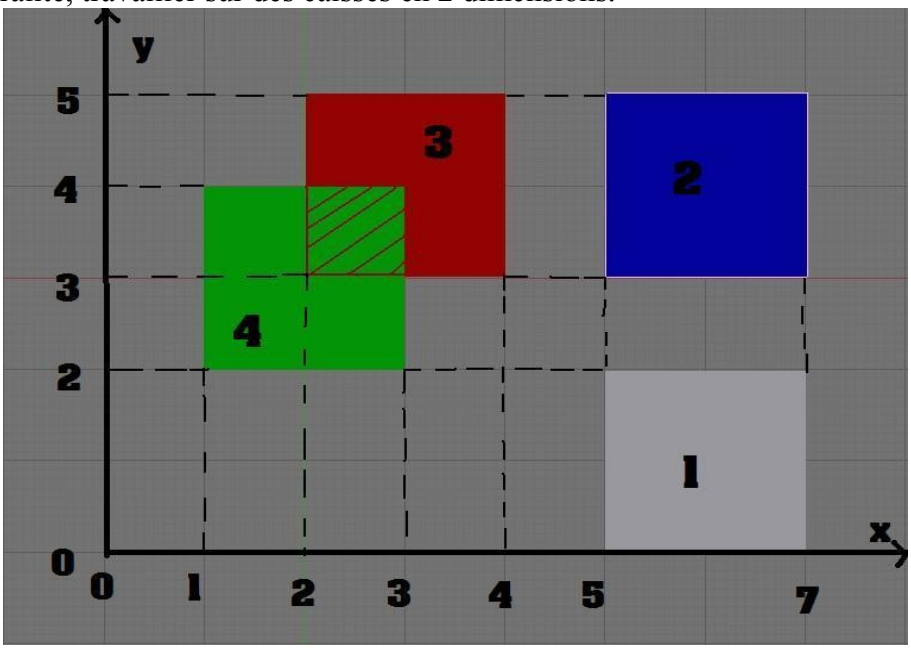


Figure 2-9 : Cas de figure en 2D

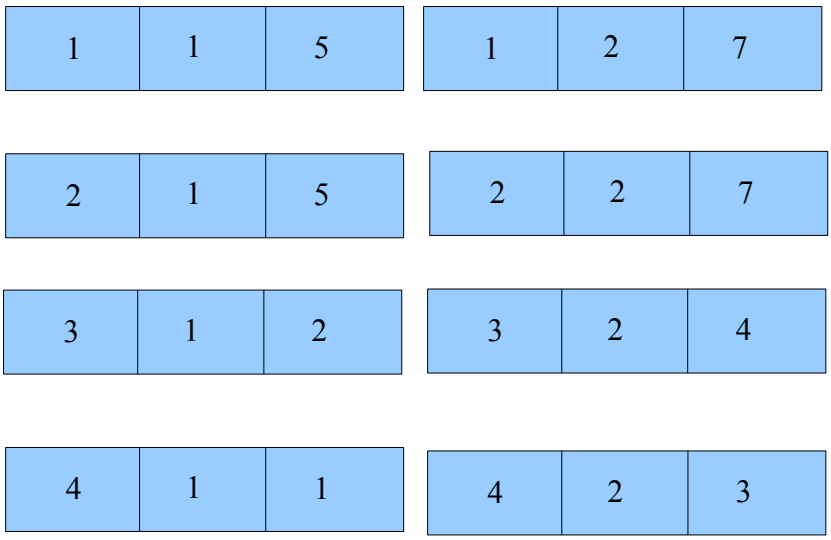


Figure 2-10 : Éléments à insérer dans la liste de position X

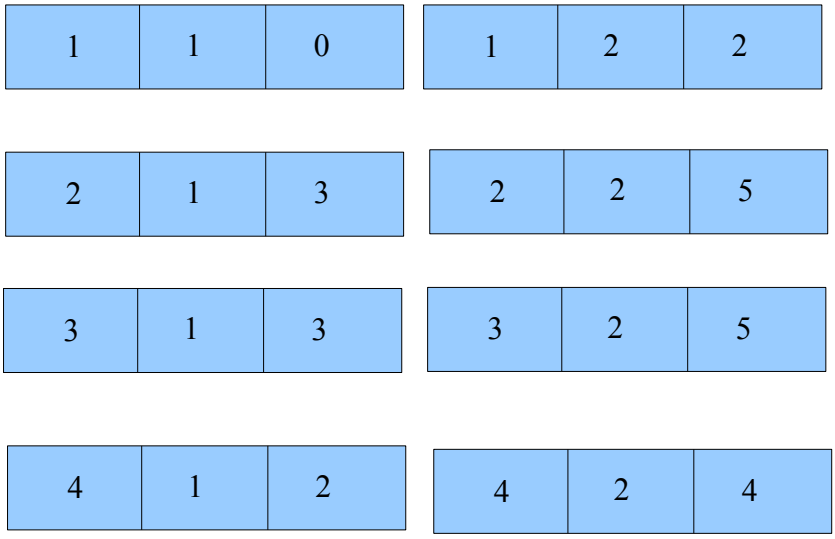


Figure 2-11 : Éléments à insérer dans la liste de position Y

On peut donc en déduire deux listes de positions X et Y qui conviennent (elles ne sont pas uniques!)

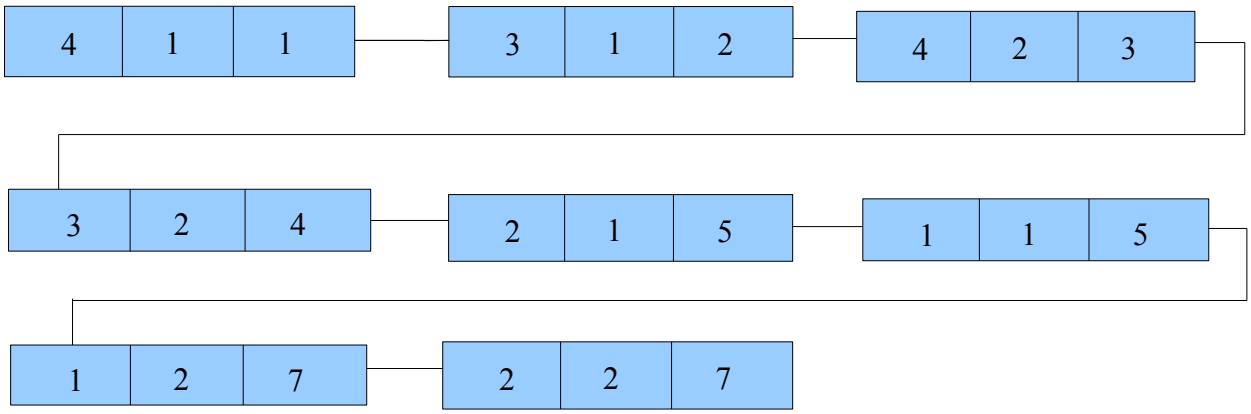


Figure 2-12 : liste de position X

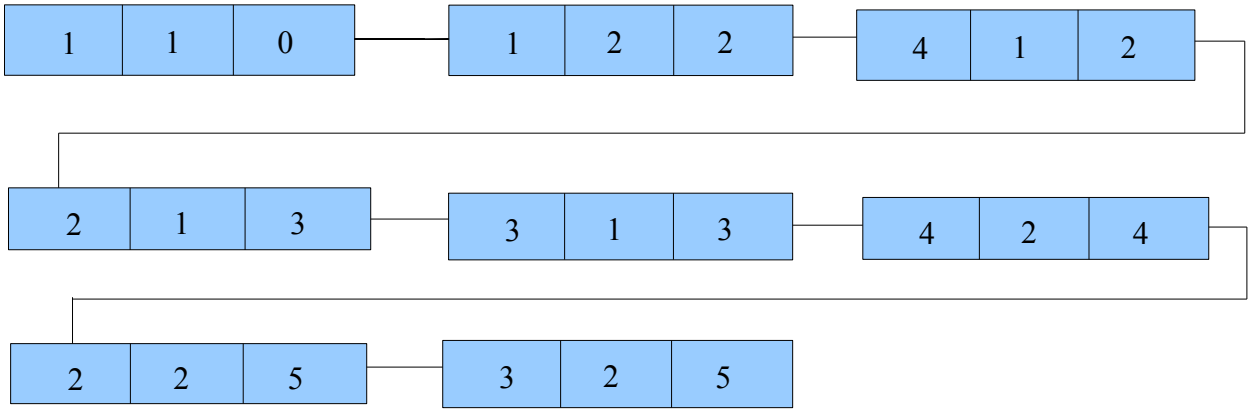


Figure 2-13 : liste de position Y

On peut déduire :

- 1 n'est pas en collision car ses deux éléments sont consécutifs dans les 2 listes
- 2 n'a aucun élément consécutif, mais c'est un des cas particulier desquels il faut se méfier (visible aux coordonnées égales)
- 3 n'a aucun élément consécutif, donc il est en collision.
- 4 n'a aucun élément consécutif, donc il est en collision.

Le diagramme d'activité de la détection de collision est le suivant :

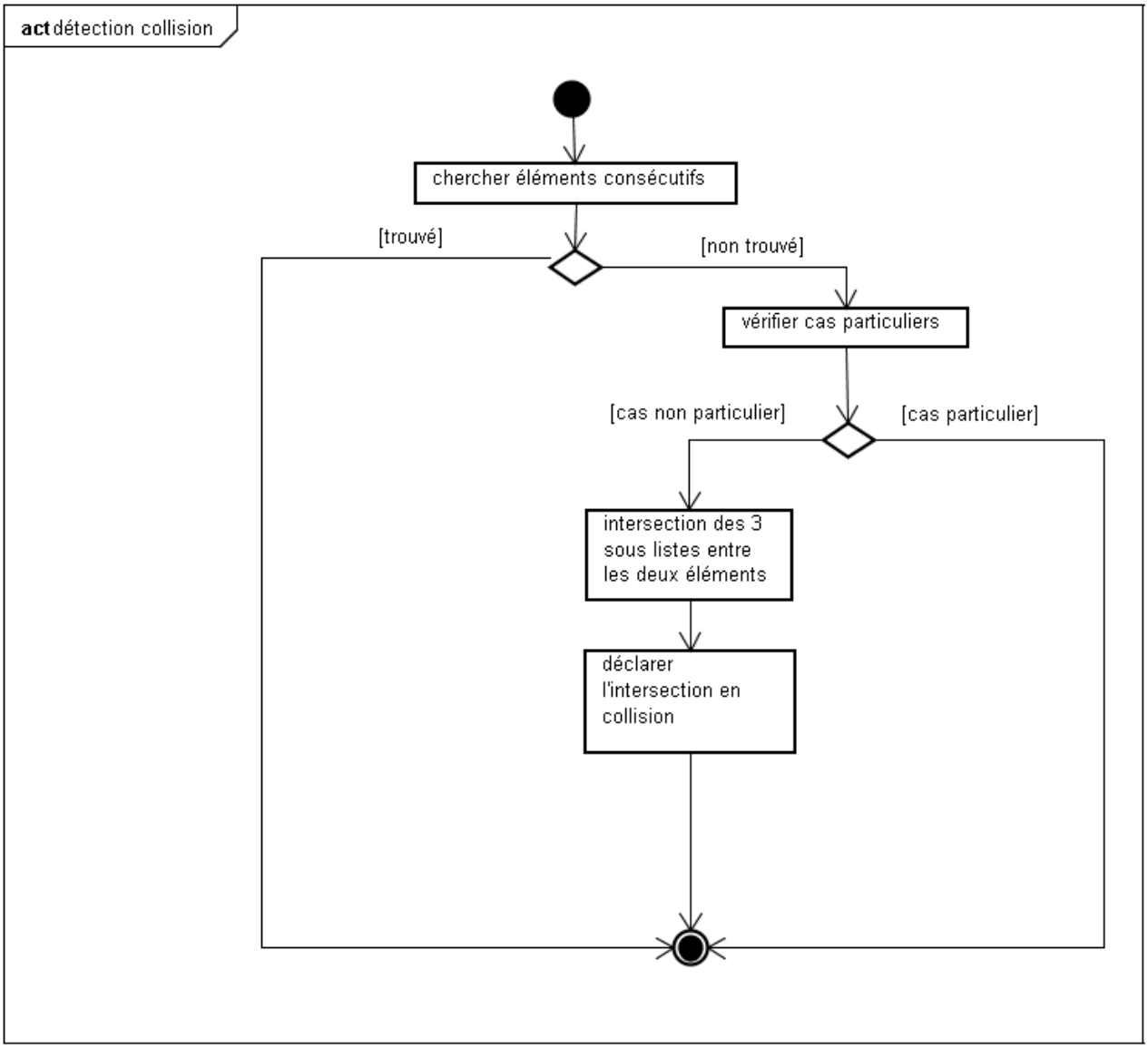


Figure 2-14 : diagramme d'activité de la détection de collisions

Les listes de positions permettent donc de détecter les collisions entre objets statiques.
Elles n'évoluent que lors de l'ajout ou de la suppression d'une caisse, et on les re-trie lorsqu'une caisse a fini d'être déplacée.

2.3.3.b. Les listes de mouvements

Les listes de mouvements sont relatives à un élément : celui que l'on a choisit de déplacer. Elles sont crées lors de la sélection d'un objet, et sont détruites lorsqu'on lâche le bouton de la souris.

Les listes de mouvement sont construites à partir des listes de position, chaque axe indépendamment l'un de l'autre. C'est à dire que pour construire la liste de mouvement de l'axe X, la liste de position de cet axe suffit, et on a pas besoin de connaître les listes de position des axes Y et Z.

Chaque élément de position n'est en fait qu'un intervalle, et une liste exhaustive de ce que l'on trouve dans cet intervalle. Une liste de mouvement permet donc de totalement couvrir un axe. Le premier et le dernier élément sont ajoutés artificiellement : le premier va de - ∞ à la première caisse, et le dernier va de la dernière caisse à +∞. Un élément mouvement est crée chaque fois que la projection sur l'axe en question ne vient plus des mêmes caisses, autrement dit, à chaque fois que l'on rencontrera ou quittera l'intervalle de projection d'une caisse.

Il devient donc évident de construire ces intervalles à partir des listes de position, puisqu'elles sont triées. Il ne faut cependant pas tenir compte de la caisse que l'on déplace.

Afin d'illustrer cet algorithme, reprenons l'exemple plan précédent, et les listes de positions que nous en avions déduites. La construction pour chacun des axes étant strictement la même nous n'allons faire l'exemple que pour l'axe X, en supposant qu'on souhaite déplacer la caisse 1 :

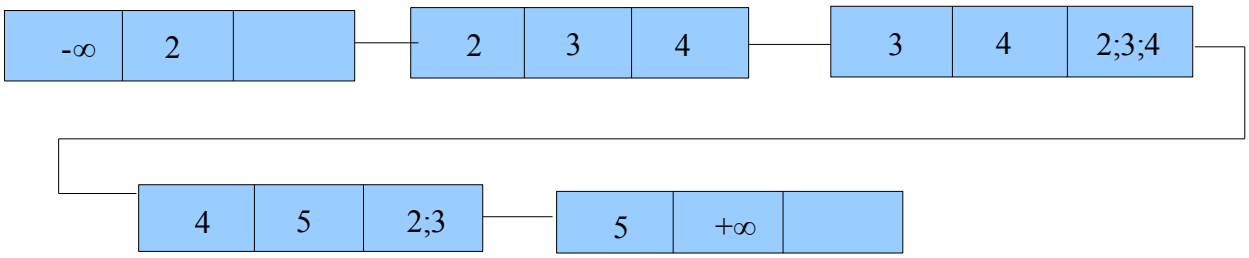


Figure 2-15 : liste de mouvement

On peut lire ça de la manière suivante « entre -∞ et 0, je ne rencontre aucun obstacle », « entre 2 et 3, je rencontre uniquement 14»... »entre 3 et 4, je rencontre 2, 3 et 4 »... Nous pouvons remarquer que la caisse 1 est totalement ignorée.

Il faut maintenant repérer les projections de la caisse que l'on bouge sur chacun des axes. On va pour se faire utiliser en tout 6 pointeurs, chaque pointeur correspondant à un coin significatif de

la caisse. On a donc 2 pointeurs par liste de mouvement, qui pointeront chacun sur une case dont l'intervalle contient la position du coin qui leur est assigné (par exemple, les 2 pointeurs qui définissent la projection de la caisse 1 sur X pointeront sur la première case).

Il faut maintenant faire évoluer ces pointeurs en même temps que la caisse se déplace, afin que à chaque instant on sache si les projections de la caisse croisent les projections d'autres objets.

Afin de permettre que 2 caisses soient accolées, un pointeur changera de caisse lorsque qu'il sera strictement supérieur (respectivement inférieur) à la valeur de fin (respectivement de début) de l'intervalle.

La détection de la collision devient alors similaire à la détection statique : à chaque fois qu'un pointeur changera de case, seront déclarées en collision toute les caisses qui sont dans l'intersection des 3 sous listes de caisse comprises entre 2 pointeurs (les sous listes de caisses sont obtenus par l'union de la 3ème case de chaque élément de mouvement placé entre les 2 pointés, ces derniers y compris)

La détection des collisions en mouvement n'est fluide que parce que les calculs d'intersection et d'union des listes ne sont fait que lors des déplacements de pointeurs. De plus, certaines techniques permettent d'accélérer l'ensemble, en évitant par exemple à l'aide de variables locales, d'accéder à chaque fois aux éléments de la liste pour connaître les bornes de l'intervalle.

2.3.4 Conclusion sur les collisions

Bien que tous ces éléments soient nécessaires pour la détection des collisions, d'autres sont cruciaux pour la gestion de ces dernières. Par exemple, chaque caisse dispose d'une liste qui contient toutes les caisses avec lesquelles est en collision, ceci pour éviter qu'une caisse soit considérée comme « isolée » simplement parce qu'elle vient de « perdre une collision » avec l'une de ces caisses. Bien qu'intéressant, lister l'ensemble des outils permettant la détection et la gestion complète des collisions est impossible ici pour des raisons évidentes de taille.

Signalons cependant que pour une vision claire des collisions, toutes les caisses détectées comme étant en collisions sont surlignées en rouge vif dans notre projet.

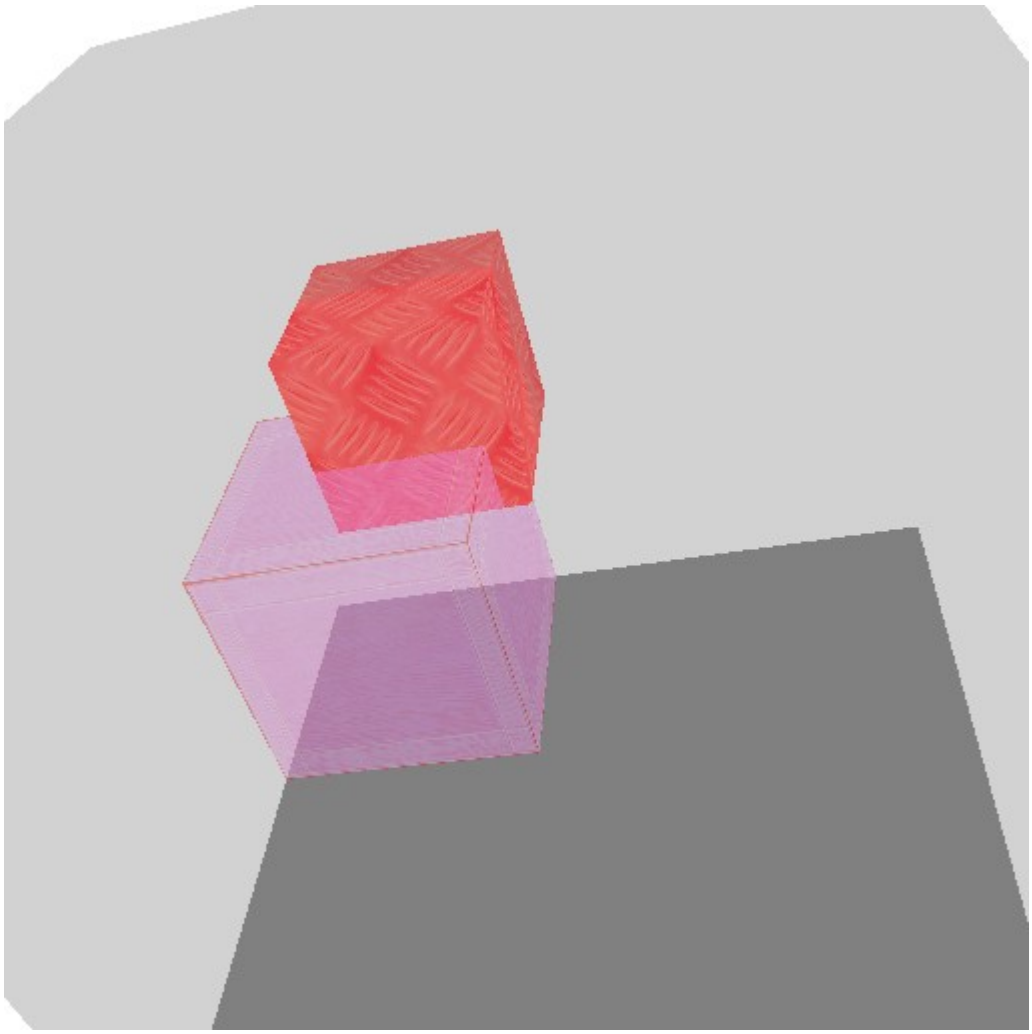


Figure 2-16 : 2 caisses en collision sous ISIBox

2.4. Ouverture et enregistrement

Il faut dans cette partie distinguer deux types d'entrées/sorties : celles créées par et/ou pour notre logiciel ; et celles générées par le projet de Chen DAI et Caroline LEOTOING.

2.4.1 Les fichiers spécifiques

Notre logiciel gère deux extensions qui lui sont propres : les fichiers « cont », et les fichier « box ». Nous avons nous mêmes défini le formalisme de ces fichiers, et ils ne répondent pour le moment à aucun standard. Voyons ces fichiers un peu plus en détail.

2.4.1.a les fichiers « box »

Ces fichiers servent à décrire ce qui est propre à une caisse, à savoir, ses dimensions, sa texture, et d'éventuels commentaires comme le poids ou l'expéditeur par exemple.
Le formalisme est le suivant :

- 1ère ligne : les dimensions de la caisse, séparées par des virgules
- 2ème ligne : la texture de la caisse (attention, la texture doit être le nom d'un fichier image contenu dans le dossier des textures)
- Ensuite : toutes les lignes suivantes seront considérées comme des commentaires.

Voyons un exemple de fichier « box »

```
0.5,0.5,0.5
bois.jpg
tout ceci
sera considéré
comme commentaire
```

Figure 2-17 : exemple de fichier Box

Ce fichier créera donc une caisse cubique de dimension 0,5 ayant une texture de bois avec pour commentaire « tout ceci sera considéré comme commentaire » comme on peut le voir sur la figure 2-18

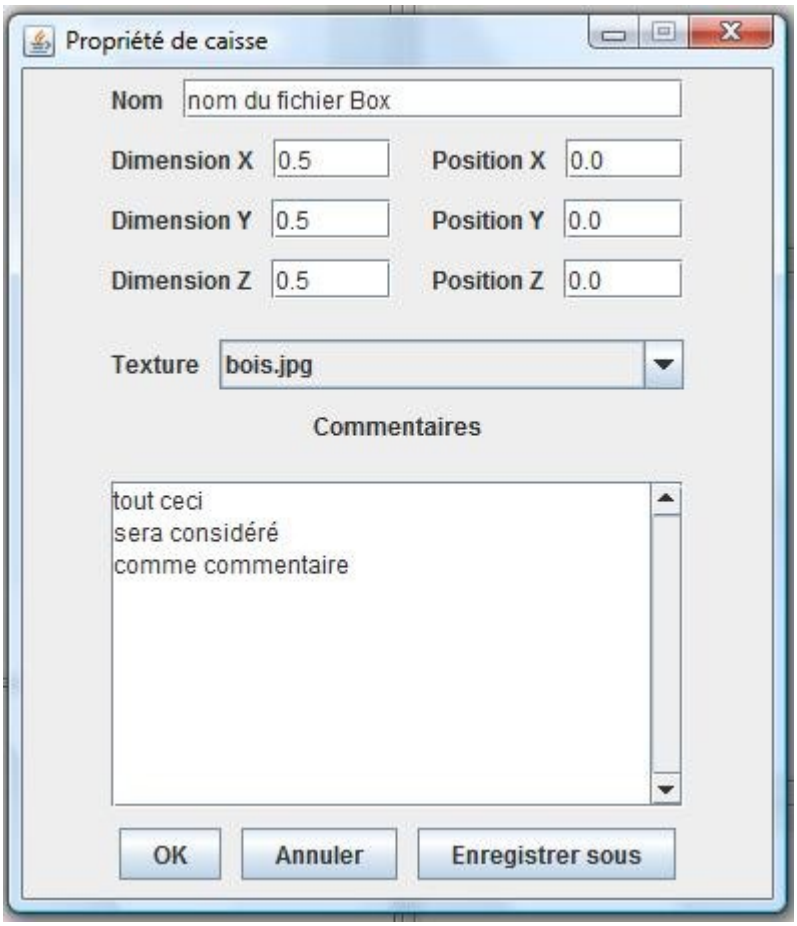


Figure 2-18 : la fenêtre d'ouverture ou de création d'une caisse

L'intérêt de créer un fichier contenant les informations propres de la caisse est de pouvoir ajouter de multiples fois une caisse à un projet à partir d'un fichier, sans avoir à créer cette caisse plusieurs fois.

A noter que le nom du fichier « box » sera utilisé pour définir le nom de la caisse dans le projet courant.

2.4.1.b les fichiers « cont»

Les fichiers « cont » sont des fichiers servant à définir l'ensemble d'un projet. Il ne se suffit pas à lui même, puisque son rôle est de définir la taille du conteneur ainsi que le placement des caisses, mais ils ne contient aucune information sur les caisses en elles-mêmes. Il va pour cela faire référence à des fichiers « box ».

L'intérêt d'avoir 2 types de fichiers différents, l'un faisant référence à l'autre, a été longuement discuté entre nous. L'inconvénient est que pour transmettre un projet, il faut transmettre l'ensemble du dossier qui contient le fichier « cont » et les fichiers « box ». L'avantage principal est qu'il devient très simple d'utiliser dans un projet une caisse créée dans un autre.

Le formalisme est le suivant :

- 1ère ligne : les dimensions du conteneur séparées par des virgules
- Ensuite : chaque ligne va correspondre à une caisse présente dans le conteneur. On donne 4 informations en 1 ligne : le nom du fichier « box » correspondant à la caisse, et le placement de notre caisse en 3D. Ces informations sont séparées par des virgules.

Voici un exemple de fichier « cont »

```
1.0,1.0,1.0
box1,0.0,0.0,0.5
box2,0.0,0.5,0.0
box3,0.5,0.0,0.0
box4,0.0,0.0,0.0
```

Figure 2-19 : exemple de fichier cont

Ce fichier créera donc un conteneur cubique de dimension 1, contenant 4 caisses. Les 4 fichiers « box » doivent être placées dans le même répertoire que le fichier « cont ».

2.4.2 Les fichiers extérieurs

Le but premier du projet est la visualisation du résultats d'algorithmes de résolution du problème du *Bin Packing*. Notre programme doit donc être capable de lire les résultats du projet de Chen DAI et Caroline LEOTOING qui ont le format « slt » qui ne correspond pas non plus à un standard. Leur résultat ne comporte aucune information esthétique, mais simplement les dimensions du conteneur, ainsi que pour chaque caisse présente dans le conteneur, un numéro, et les coordonnées de la grande diagonale desquelles on déduit dimensions et position.

Voici un exemple de fichier « slt »

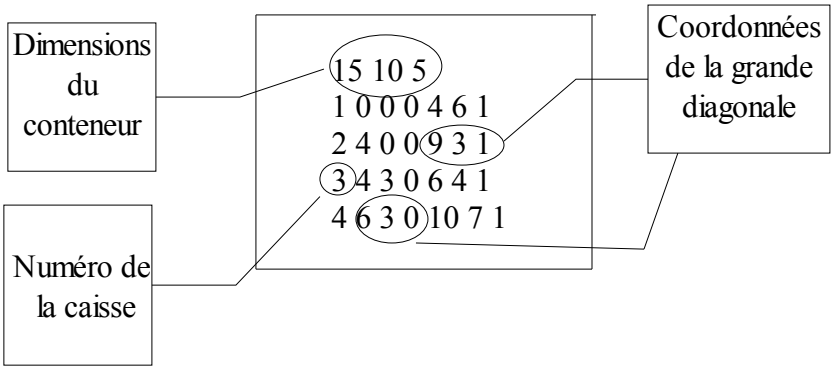


Figure 2-20 : exemple de fichier slt

Le nom de chaque caisse sera le numéro indiqué dans le fichier, le champ des commentaires sera laissé vide, et la texture sera choisie aléatoirement parmi toutes celles possibles.

Les principales difficultés de l'ouverture/sauvegarde étaient liées aux erreurs dans les fichiers. Nous avons autant que faire ce peut essayer de gérer ses erreurs en évitant un plantage du système ou même du logiciel. Des erreurs comme l'absence d'un fichier « box » appelé lors de la lecture d'un fichier « cont » se gère facilement en ignorant cette caisse ; mais le problème devient plus compliqué à gérer si une des dimensions du conteneur est manquante. Nous avons mis deux solutions en place en cas d'erreur : ou alors un traitement secondaire (par exemple ignorer la caisse et passer à la suivante), ou alors interrompre net le traitement (c'est le cas de la coordonnée manquante par exemple).

3. Partie 3 : Les améliorations possibles

Bien que relativement complet, notre projet peut encore s'améliorer sur de nombreux points sur lesquels nous ne nous sommes pas penchés par manque de temps.
Voici certaines pistes de réflexion :

- Parallélisation : nous l'avons vu précédemment, la vue 3D est l'intersection des vues 2D ; ceci explique que certains traitements sont a effectuer de manière totalement similaire sur les vue 2D comme c'est le cas pour les collisions. La parallélisation prend donc tout son sens, surtout grâce à la banalisation des processeurs mutlicoeurs.
- Déplacement du zoom : le zoom actuel mis en place est centré sur la vue, et on n'a pas la possibilité de le décentrer, ce qui est pourtant très utile à l'usage.
- Invisibilité par rapport à la sélection : il pourrait être très intéressant de pouvoir rendre une caisse invisible à la sélection afin de pouvoir sélectionner une caisse placée au centre du conteneur, sans avoir à déplacer toutes les autres placées devant.
- Dock : le dock constituerai une alternative à l'ouverture d'une caisse. Au lieu d'avoir à ouvrir plusieurs fois une caisse, on pourrait l'ouvrir dans le dock, et l'insérer dans la scène courante aussi souvent qu'on le voudrait par le biais d'un glisser/déposer.
- Initialisation de la vue : à l'heure actuelle, la caméra est placé de manière arbitraire par rapport au conteneur, peu importe les dimensions de ce dernier. Il pourrait être très confortable pour l'utilisateur d'avoir un placement automatique de la caméra lors de la création du conteneur, ainsi qu'un bouton qui replacerait toutes les caméras à une distance optimale.

Il subsiste cependant certains défauts que nous n'avons pas eu le temps de corriger.
En voici certains :

- Annuler/rétablir : l'utilisation de cette fonctionnalité pose un certain nombre de problème. Un exemple typique est lors de l'ouverture d'un projet : si on clique sur annuler juste après l'ouverture du projet, la dernière caisse créée disparaitra, mais pas l'ensemble du conteneur.
- Insertion de caisse : le champ des commentaires n'est pas enregistrer lors de l'insertion d'une caisse, mais l'est lorsqu'on modifie ce champ d'une caisse existante.

Ces points sont les principaux qui nécessitent une amélioration avant de rendre ISIBox totalement opérationnel, bien qu'il soit tout à fait utilisable en l'état. D'autres améliorations peuvent être apportées, comme rendre l'interface utilisateur plus ergonomique.

Conclusion

Le logiciel ISIBox est loin d'être totalement abouti à l'heure actuelle ; cependant, notre projet constitue déjà une aide pour Christophe DUHAMEL et Philippe LACOMME. Il répond aux objectifs principaux que nous avions définis ensemble au mois d'Octobre, et comporte de nombreuses fonctionnalités supplémentaires que nos tuteurs ont approuvées au fur et a mesure de l'évolution de notre programme.

De nombres améliorations et évolutions sont possibles, voire nécessaires, comme nous avons pu le voir précédemment. Nous avons offert un certain nombre de pistes de réflexion pour les prochains étudiants qui travailleront sur ce projet.

Ce projet s'est révélé très instructif. Il nous a permis d'avoir une première approche du langage JAVA avant de le voir en cours. La partie la plus intéressante pour nous a été d'être initié au monde des logiciels 3D qui est bien à part. En effet, l'interface utilisateur joue un rôle encore plus prépondérant que pour les autres genres de logiciels. Cela nous a aussi permis de prendre conscience des problèmes mathématiques qui se cache derrière la manipulation de formes 3D basiques comme les pavés.

En conclusion, nous pouvons dire que ce projet nous a apporté énormément en terme de compétences techniques et théoriques, et il s'est révélé d'autant plus motivant car nous étions conscients de l'utilité qu'un tel projet pouvait avoir pour nos tuteurs ou pour des entreprises de transport.