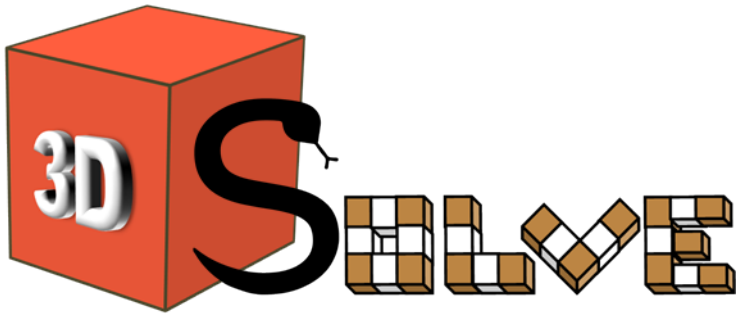
INSA - CVL

Rapport de développement

Projet application - 3DSolve

31 mai 2015



Auteurs :Aubry Lisa, Chazot Alban, Colas Korlan, Gourd Anthony -
Promotion 2017
Tuteur : P.Clemente

Résumé

Compte rendu portant sur le projet application effectué lors de la première année de formation à la Sécurité et aux Technologie de l'Informatique (STI) à l'INSA Centre Val de Loire

Table des matières

Introduction

Première partie

Conception

Chapitre 1

Présentation du casse-tête

Le Snake Cube (ou cube serpent en français) est un casse-tête géométrique à trois dimensions appartenant à la famille des casse-tête mécaniques. Cette famille, dont fait parti le Rubik's Cube, recouvre l'ensemble des jeux de réflexion basés sur la manipulation d'un objet tridimensionnel, afin de lui donner un agencement précis.

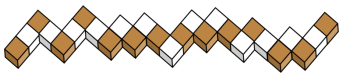


FIGURE 1.1 – Snake Cube déplié

Ce casse-tête se présente généralement comme une succession de 27 petits cubes en bois, reliés les uns aux autres par un fil élastique qui les traverse de part en part. Lorsque tous les cubes sont mis sur le même plan, la forme du puzzle s'apparente à un serpent (Figure 1.1). Pour résoudre le puzzle, il faut manier le serpent afin de le ramener à une forme entièrement cubique telle que présentée ci-dessous (Figure 1.2). Dans la suite de cette partie et pour éviter les ambiguïtés, les petits cubes qui forment le serpent seront appelés **unités** tandis que le mot cube désignera plutôt le volume final visible sur la figure 1.2.

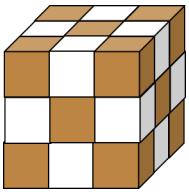


FIGURE 1.2 – Snake Cube résolu

Comme mentionné précédemment, les unités sont reliées entre elles ce qui les rend totalement indissociables. Ainsi, les manipulations pouvant être effectuées sur le puzzle sont réduites à des rotations. La Figure 1.3 illustre une manipulation réalisée sur le serpent présenté figure 1.1. De plus, on remarque que les rotations conduisant à modifier la forme de la figure s'effectuent toujours au niveau des coins du serpent. Cette particularité sera notamment exploitée lorsque l'on traitera de la résolution du casse-tête.

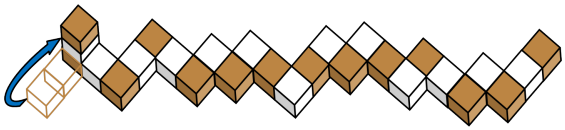


FIGURE 1.3 – Exemple de manipulation du Snake Cube

Notons également que pour une figure finale donnée (ici une cube 3x3x3), il existe une multitude de formes de serpents correspondantes. Les serpents les plus connus seront listés en annexe de ce document, associés à leurs spécificités (difficulté, nombre de solutions, temps de calcul correspondant). De manière moins classique, on trouve aussi dans le commerce des serpents permettant de réaliser des cubes 4x4x4. À notre connaissance il n'existe pas de casse-tête correspondant à des Snake Cubes de tailles supérieures.

Chapitre 2

Description des objectifs

2.1 Résoudre par le calcul un Snake Cube

Dans un premier temps, notre objectif consiste en l'implémentation d'une application capable de résoudre des Snake Cubes. L'utilisateur fournit au résolveur deux paramètres essentiels : la définition d'un serpent initial ainsi qu'un volume final. Ensuite l'application automatise la tâche de recherche pour présenter à l'utilisateur la liste de solutions correspondant au problème. Les solutions devront être présentées de manières claires avec, pour chacune d'entre elles, la possibilité pour l'utilisateur de la visionner pas-à-pas ou bien encore de revenir en arrière. Une première contrainte évidente liée à cet objectif concerne le temps nécessaire au calcul. Ainsi, il va de soi que le résolveur devra à la fois s'employer à utiliser des méthodes de calcul pertinentes et être capable de ne pas rechercher des solutions redondantes. Cette redondance traduit en fait le caractère symétrique de plusieurs solutions.

2.2 Proposer à l'utilisateur de résoudre lui-même le Snake Cube

Dans un second temps, nous proposons à l'utilisateur un mode interactif où celui-ci pourra tenter de résoudre virtuellement le casse-tête. En termes de contraintes, ce versant de l'application nécessite très peu de calcul. Néanmoins, il exige une réflexion accrue concernant la représentation graphique. En effet, il faut ici proposer au joueur un environnement en 3 dimensions à la fois clair et maniable ainsi qu'un panel de fonctionnalités rendant l'application intuitive.

Chapitre 3

Principe de la résolution

Dans cette partie nous allons exposer les principes algorithmiques qui sous-tendent le fonctionnement du résolveur. Cette approche, bien que dénuée de détails techniques et autres soucis d'implémentation, proposera une réponse à nos différents objectifs tout en tenant compte de certaines des contraintes citées précédemment. Nous nous baserons ici sur la résolution d'un cube 3x3x3 à l'aide du serpent présenté précédemment, le Cubra Orange.

3.1 Quelques définitions

Tout d'abord, deux éléments constituent la base de l'algorithme de résolution :

— le serpent, ici le Cubra Orange, composé de 27 unités reliées entre elles
— le volume final, ici un cube 3x3x3

Le volume final est définit par ses coordonnées, fixées, dans l'espace. Le serpent est quant à lui définit par sa forme qui découle de la nature de chacune des unités qui le composent. En effet, il faut distinguer trois types d'unité du serpent (Figure 3.1) :

— les extrémités
— les unités droites
— les coins

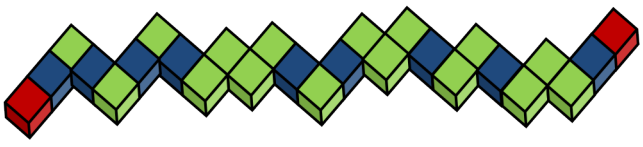


FIGURE 3.1 – Natures des unités du Cubra Orange

3.2 Construction d'un arbre n-aire

La résolution s'effectue de manière linéaire, en considérant tour à tour chacune des unités du serpent. Le principe est de placer temporairement l'unité considérée à l'intérieur du volume final en fonction de l'emplacement et de la nature de l'unité précédente. On construit ainsi une branche d'un arbre n-aire qui est le cœur de l'algorithme de résolution. Chaque nœud correspond à une unité associée à un couple coordonnées-direction.

Ici il est important de comprendre que ce travail s'effectue de telle sorte que chaque nouvelle unité soit nécessairement placée à l'intérieur du volume final. En tenant compte de cette contrainte et en fonction des emplacements choisis précédemment, on peut constater deux comportements pour une branche donnée.

Soit on arrive à une impasse, c'est-à-dire qu'une configuration donnée ne permet pas de placer l'unité suivante sans sortir du volume ou la placer sur une autre unité. Dans ce cas, la branche avorte, il faut alors remonter dans l'arborescence jusqu'au prochain nœud qui a donné plusieurs fils. Ce cas de figure est illustré sur les figures 3.2 et 3.3. Les flèches horizontales représentent les liens de fraternité dans l'arborescence et les flèches verticales représentent les liens de paternité. Ici on a construit six vecteurs initiaux, c'est-à-dire les couples coordonnées-direction possibles pour la première unité. Puis le premier nœud est devenu le nœud courant et a lui-même engendré des fils, en fonction des vecteurs possibles pour la deuxième unité. Et ainsi de suite jusqu'à ce que le nœud représenté en rouge devienne le nœud courant. À ce stade, on admet que ce nœud ne puisse pas engendrer de fils pour une des raisons évoquées précédemment. Cette branche va donc avorter. On va supprimer le nœud rouge et remonter jusqu'à son père (en vert). Si celui-ci a un frère, comme c'est le cas ici avec le nœud bleu, c'est ce frère qui devient le nouveau nœud courant. Il convient de supprimer également le nœud vert qui a généré le sous arbre menant à une impasse. Ensuite la résolution peut suivre son court, et le nœud bleu générera à son tour ses fils (figure 3.3).

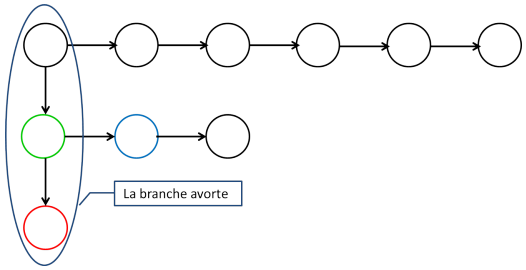


FIGURE 3.2 – Exemple d'une branche qui avorte

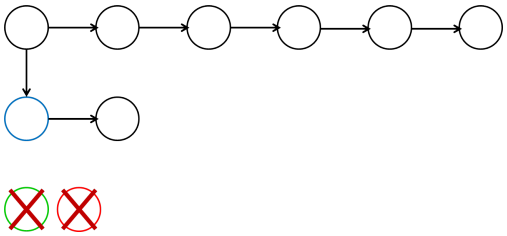


FIGURE 3.3 – Arbre après modifications

Le deuxième cas de figure est celui où l'on réussit à placer toutes les unités du serpent à l'intérieur du volume. Dans ce cas, la branche aboutit et la succession de ses nœuds depuis la racine constitue une solution au problème. Il faut alors garder cette solution en mémoire puis, comme dans le cas précédent, remonter dans l'arborescence pour chercher des solutions supplémentaires.

3.3 Méthode de calcul des couples coordonnées-direction des nœuds fils

Lorsqu'un nœud devient le nœud courant, il convient que ses fils soient générés dans l'arbre. Cela consiste notamment à leur attribuer un couple coordonnées-direction dans l'espace. Ce qu'il faut voir ici, c'est que le résultat de cette opération découle de la nature du nœud fils et du couple coordonnées-direction du père. Dans la suite de cette partie, on se placera dans un repère orthonormé (O ; x, y, z). Pour calculer les coordonnées des fils d'un nœud, il suffit d'ajouter ses coordonnées avec sa direction. La figure suivante illustre ce procédé.

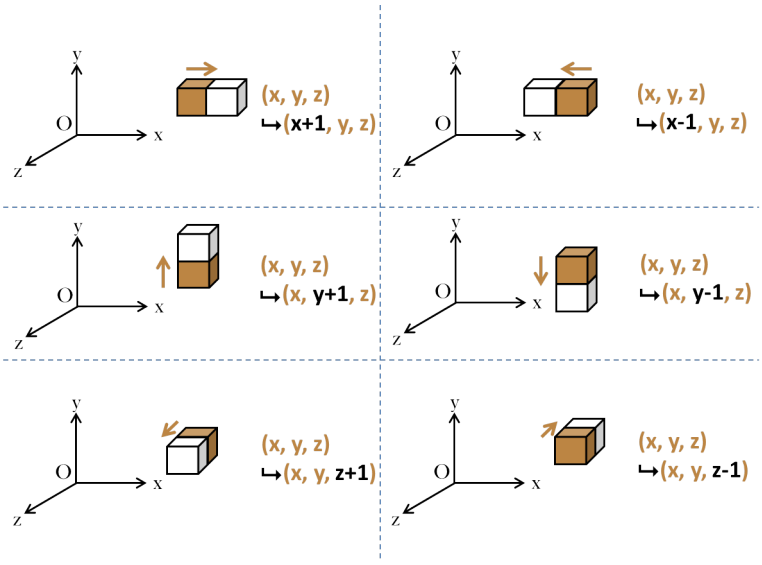


FIGURE 3.4 – Calcul des coordonnées des fils pour six pères différents

Le nœud marron est le nœud courant et le nœud blanc est son fils. Les coordonnées du fils découlent de la position du père ainsi que de sa direction, matérialisée par la flèche marron. On constate alors que, pour un nœud père donné, tous les fils qu'il va engendrer auront le même triplet de coordonnées dans l'espace.

La différence entre les fils d'un nœud intervient lors du calcul de leur direction. Deux cas de figure sont alors à envisager :

— le nœud fils correspond à une unité droite (voir figure 3.1), dans ce cas, un seul nœud fils sera créé et celui-ci héritera de la direction de son père
— le nœud fils correspond à un coin, dans ce cas, plusieurs nœuds fils seront créés avec des directions différentes de celle du père

La figure 3.5 illustre le fonctionnement de l'arbre pour placer la 3e unité du serpent à partir d'un nœud donné. Les flèches indiquent les directions associées au dernier cube placé. À la hauteur n, on a déjà positionné les deux premières unités et la 2e unité a pour direction droite. Étant donné que la 3e unité est un coin (voir figure 3.1), il convient que la direction change entre la 2e et la 3e unité. Ainsi, les directions possibles pour la 3e unité sont : bas, derrière, devant et haut, d'où les quatre nœuds représentés à la hauteur n+1.

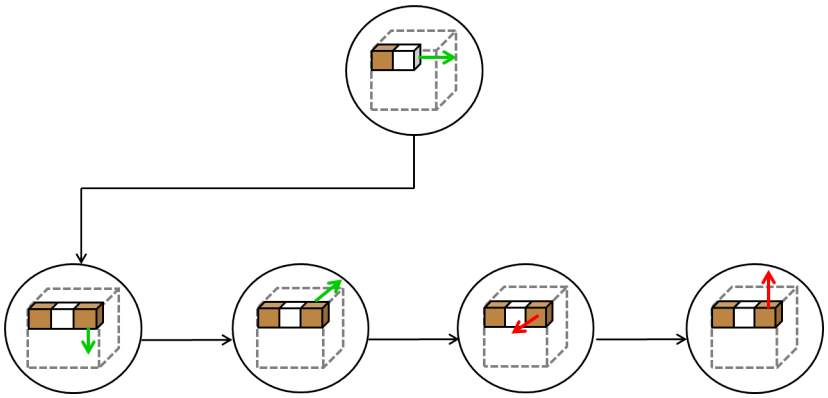


FIGURE 3.5 – Exemple de création des fils pour un coin

NB : on peut d'ores et déjà remarquer que les directions devant et haut vont déboucher sur une configuration interdite car la prochaine unité créée sortira du cube.

3.4 Problème des nœuds initiaux

Dans cette partie nous allons exposer la méthode employée pour placer la première unité dans le volume final. En effet, la démarche exposée jusqu'ici ne s'applique pas pour les premiers nœuds de l'arbre étant donné qu'elle dépend du nœud père. Ici l'enjeu est double puisqu'il faut également déterminer et éliminer les couples coordonnées-direction symétriques. Afin de simplifier la lecture, les couples coordonnées-direction seront appelés vecteurs dans ce qui va suivre. La figure 3.6 illustre en partie le problème des symétries sur la première face du cube. Dans cet exemple, l'unité marron amènera à la création d'un nœud et les trois autres unités devront être reconnues comme étant redondantes et sans intérêt pour le calcul des solutions.

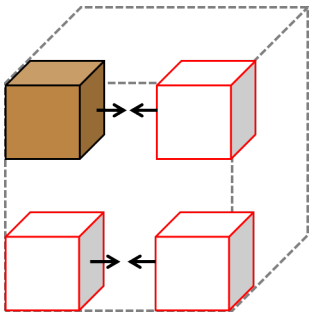


FIGURE 3.6 – Quatre vecteurs initiaux redondants

La recherche des vecteurs initiaux se déroule en plusieurs étapes et se base sur les axes de symétrie horizontale, verticale et diagonale (en slash et antislash). Il faut d'abord déterminer, parmi ces axes, ceux que l'on peut appliquer aux faces du volume final. Cette information sera fournie dans un fichier contenant les caractéristiques du volume. Une fois les axes connus, nous allons traiter des tranches de volume les unes après les autres. Par tranche on entend l'ensemble des unités appartenant au même plan parallèle à (O ; x, y). Ensuite il convient de projeter orthogonalement le centre géométrique de la figure sur la face en cours de traitement. Une fois les coordonnées du projeté connues, on peut calculer les équations cartésiennes des axes de symétries et ainsi éliminer les vecteurs redondants. Aussi notons que les vecteurs avec une direction qui sort du volume ne seront pas retenus. En suivant ce principe, la recherche des vecteurs initiaux pour un volume cubique 3x3x3 amène à la création de six vecteurs au lieu des cent huit présents.

Chapitre 4

Algorithme de résolution

4.1 Structure de donnée

Afin de pouvoir présenter certain des algorithmes que nous avons mis en place dans notre application, nous allons d'abord présenter brièvement les structures de données utilisées par ces algorithmes.

Dir Type énuméré qui définit les six différentes directions dans l'espace, plus une valeur qui signale une erreur

Unit Type énuméré qui définit les trois natures d'unité de cube, coin, unité droite ou extrémité

Coord Structure contenant trois valeurs (x ; y ; z) entières pour des coordonnées dans l'espace

Etape (Step) Structure représentant une étape de la solution c'est-à-dire un champ coord (type Coord) et un champ dir (type Dir).

VolumeState Type énuméré qui permet d'indiquer dans le volume final si une position est libre, occupée ou interdite

Volume Type structuré qui contient un champ max (type Coord) définissant les limites du volume dans l'espace et selon les trois coordonnées, et un champ état (state) (tableau 3D de VolumeState)

Serpent (snake) Type structuré qui contient :
- Un entier, taille (length), indiquant le nombre d'unité du serpent
- volume, un élément de type Volume qui définit le volume final
- units, qui pointe sur les unités du serpent
- currentUnit, pointant sur l'unité en cous de traitement
- solutions, pointeur sur la liste des solutions
- symetries, tableau de quatre entiers correspondant aux différents axes de symétrie

Nœud (TreeNode) Structure de données représentant un nœud dans l'arbre de recherche des solutions. Un nœud représente une étape dans l'enchaînement en cour de recherche. Si cet enchaînement abouti à une solution alors le mouvement (l'étape) porter par le nœud deviendra une étape de la solution. Chaque nœud possède un pointeur vers son père, son frère et son premier fils.

4.2 Parcours en profondeur d'une branche

Cet algorithme permet de créer à la voler et de parcourir en profondeur une branche de l'arbre à partir d'un nœud initial correspond au placement du premier élément du serpent. Cet algorithme fait référence au principe exposé dans la partie 3.2 (page 9).

```
1  Procedure ResoudreNoeud(racine : Noeud, s : Serpent)
2  Variables - noeud : Noeud // Le noeud courant
3            resultatConstruction : entier
4  Début
5  | noeud <- racine
6  | Tant que noeud ≠ racine OU racine->aJoué = 0
7  | | Si noeud->aJoué = 0
8  | | | resultatConstruction <- ConstructionDesFils(noeud, s)
9  | | | Si resultatConstruction = 1 Alors
10 | | | // Construction OK
11 | | | | noeud <- noeud->enfantCourrant
12 | | | Sinon si resultatConstruction = 2 Alors
13 | | | // Construction OK et casse-tête résolu
14 | | | | sauvegardeSolution(s)
15 | | | Fin si
16 | | Sinon // Le noeud n'a pas joué
17 | | | serpent->volume.etat[noeud->étape.coord.x]
18 | | |                     [noeud->étape.coord.y]
19 | | |                     [noeud->étape.coord.z] <- LIBRE
20 | | | Si noeud->frère ≠ NIL
21 | | | | noeud <- noeud->frère
22 | | | Sinon
23 | | | | noeud <- noeud->père
24 | | | | remonterSerpent(s)
25 | | | Fin si
26 | | Fin si
27 | Fin Tant que
28 Fin
```

Listing 4.1 – Algorithme de résolution pour une branche

Explication Tant que l'algorithme tout les chemin possible découlant du nœud initial (`racine`), il fait "jouer" le nœud courant s'il ne l'a pas déjà fait. Faire "jouer" le nœud consiste à tenter de lui créer ses fils grâce à l'algorithme `ContructionDesFils`. Si la construction réussi, cela signifie qu'au moins un fils a été créé, le premier des fils créé devient alors le nœud courant. Si la construction a réussi et qu'en plus, le dernier élément du serpent a été posé, alors le casse-tête est résolu. On enregistre donc la séquence de mouvement aboutissant à cette solution. Si la construction, cela signifie que le chemin emprunté ne peut pas aboutir à une solution et rien de ne passe.

Si le nœud a déjà jouer alors on réinitialise l'état du sous-volume qu'il occupe à Libre. Autrement dit, on enlève cet élément et le mouvement qu'il lui est associé de la suite de mouvement en cour de calcul. Si le nœud possède un frère, alors son frère devient le nœud courant. Sinon, c'est son père qui devient nœud courant, ce qui signifie que l'on "remonte" d'une étape dans la recherche de la solution.

Lorsque le nœud courant reprend pour valeur celle de la racine et la racine a déjà jouer, cela signifie que l'on a explorer tous les chemins possible à partir de la racine. On termine donc l'algorithme.

4.3 Création des fils

```
1  Fonction ConstructionDesFils(noeud : Noeud, s : Serpent) : entier
2  Variables - nCoord : Coordonné
3              prochaineUnité : entier
4              fils : Noeud
5  Début
6  | nCoord <- CalculCoordonné(noeud->étape.coord, noeud->étape.dir)
7  | Si CoordonnéValide(nCoord) ET
8  | serpent->volume.état[nCoord.x][nCoord.y][nCoord.z] = Libre Alors
9  | | SerpentAjouterEtape(s, @(noeud->étape))
10 | | serpent->volume.état[nCoord.x][nCoord.y][nCoord.z] = Occupé
11 | | noeud->aJoué = 1
12 | | prochaineUnité = SerpentProchaineUnité(s)
13 | | Si prochaineUnité = Extrémité Alors
14 | | | fils <- Nouveau(Noued)
15 | | | fils->étape.dir <- noeud->étape.dir
16 | | | file->étape.coord <- nCoord
17 | | | NoeudAjouterFils(noeud, fils)
18 | | | Retourner 2
19 | | Sinon si prochaineUnité = Droit
20 | | | fils->étape.dir <- noeud->étape.dir
21 | | | file->étape.coord <- nCoord
22 | | | NoeudAjouterFils(noeud, fils)
23 | | Sinon // C'est un angle
24 | | | Pour i de 0 à 6
25 | | | | Si TableVéritéAngle[noeud->étape.dir][i] = 1 Alors
26 | | | | | fils <- Nouveau(Noeud)
27 | | | | | fils->étape.dir <- i
28 | | | | | fils->étape.coord <- nCoord
29 | | | | | NoeudAjouterFils(noeud, fils)
30 | | | | Fin si
31 | | | Fin pour
32 | | Fin si
33 | | Retourner 1
34 | Sinon
35 | | Retourner -1
36 | Fin si
37 Fin
```

Listing 4.2 – Algorithme de création des fils

Explications La première étape dans la construction des fils d'un nœud consiste à vérifier la validité des coordonnées des-dit futurs fils. Si les coordonnées sont valides, c'est-à-dire qu'elles sont comprises dans le volume final et que le sous-volume qu'elles repèrent est Libre, alors on ajouter une étape dans la séquence de résolution, on indique que ce nœud a joué et on récupère le type du prochain élément du serpent qu'il faut placer.

Si le type récupéré est "Extrémité", cela signifie que c'est le dernier élément à placer et que donc, le casse-tête est résolue. L'algorithme renvoie donc la valeur 2. Si le type récupéré est "Droit" alors le fils à créer est de type Droit et n'a donc qu'une seul façon d'être orienté. On créer un seul fils que l'on "accroche au nœud étudié et on retourne 1.

Si le type du prochain élément est "Corner" alors il existe au maximum 4 manière de l'orienter parmi les 6 directions possibles. Les orientations possibles sont données par `TableVéritéAngle` dont voici la composition se trouve figure 4.1. Pour chaque direction valide, on créé un fils que l'on accroche au nœud traité et on fini en retournant la valeur 1.

	Haut	Bas	Gauche	Droite	Avant	Arrière
Haut	0	0	1	1	1	1
Bas	0	0	1	1	1	1
Gauche	1	1	0	0	1	1
Droite	1	1	0	0	1	1
Avant	1	1	1	1	0	0
Arrière	1	1	1	1	0	0

FIGURE 4.1 – Table de vérité des angles

Deuxième partie

Développement

Chapitre 5

Organisation

Ce projet d'application s'inscrivant dans un travail de groupe, nous avons, dès la conception de l'application, puis lors de son développement, attaché une grande importance à l'organisation et à la répartition des tâches.

5.1 Gestionnaire de version Git

D'un point de vue technique, cette préoccupation d'organisation s'est traduite par l'utilisation du gestionnaire de version Git afin de pourvoir non seulement avoir des copies de sauvegarde de notre code sources mais également pour pouvoir travailler en parallèle les uns des autres grâce au système de branche.

L'utilisation de cet outils nous grandement facilité la tâche lors de la répartition des tâches en nous permettant de gérer indépendamment les différentes partie de l'application.

5.2 Organisation modulaire

Toujours dans le but de pourvoir nous répartir les tâches le plus facilement possible, nous avons essayé de rendre le développement de l'application le plus modulaire possible. Ainsi, nous avons séparé au maximum les modules liés à la gestion du rendu 3D des modules liés au calcul des solutions.

Enfin, dans le but d'unifier notre code sources, nous avons établies des conventions communes quant à la nomenclature des variables et des fonctions.

Chapitre 6

Core

Dans ce chapitre, nous allons aborder les aspect technique liés au développement de notre application.

6.1 Le format de fichier ".snake"

Les différents snakes proposés par l'application sont stockés dans des fichiers "*.snake" dans le répertoire "Snakes".

```
1  [Volume]
2   3;3;3
3   0;0;0;1
4   0;0;1;1
5   0;0;2;1
6   0;1;0;1
7   0;1;1;1
8   0;1;2;1
9
10  [ ... ]
11
12  2;1;1;1
13  2;1;2;1
14  2;2;0;1
15  2;2;1;1
16  2;2;2;1
17  [Snake]
18  27
19  0;1;2;2;2;1;2;2;1;2;2;2;1;2;1;2;2;2;2;1;2;1;2;1;2;1;0;
20  [Symetry]
21  1;1;1;1
```

Listing 6.1 – Contenu du fichier snake.snake

La section `[Volume]` permet de définir les caractéristiques du volume final, soit :

— sa largeur (x) ;
— sa hauteur (y) ;
— sa profondeur (z) ;
— l'état de chacun des sous-cubes le formant (x ;y ;z ;State).

L'état des sous-cubes formant le volume final permet de définir des Snake Cube dont le volume final est plus complexe qu'un simple cube. La valeur signifie que le sous-cube doit être rempli par un des élément du snake afin de résoudre le casse-tête. La valeur -1 indique que ce sous-cube doit rester inoccupé.

La section `[Snake]` définit le nombre et l'enchaînement des unités formant le serpent. La valeur 0 indique que l'unité est une extrémité du serpent, la valeur 1 indique une unité de type "droite" et la valeur 2 une unité de type "angle".

La section `[Symetry]` permet quant à elle de définir quels sont les axes de symétrie à prendre en compte lors de la résolution du casse-tête.

6.2 L'utilisation des threads

Nous avons dès le départ pris le parti d'utiliser la technique du "multi-threading" afin de séparer l'exécution de la fonction de rendu 3D et le reste de l'application. Nous avons fait ce choix dans le but de faciliter la gestion des animations et de manière générale pour ne pas rendu le rendu 3D trop dépendant des diverses fonctions de calcul.

Plus tard dans le développement de l'application, nous avons également utiliser les "thread" afin de paralléliser le calcul des solutions dans le but d'améliorer le temps de calcul en particulier pour le Snake Cube de taille 4x4x4.

Chapitre 7

Graphique

Chapitre 8

Portabilité

Troisième partie

Résultat

Chapitre 9

Réalisations

9.1 Résolution d'un Snake Cube

L'objectif visant à résoudre un Snake Cube est atteins. Notre application propose divers Snake Cube et les résout en un temps raisonnable (voir Chapitre 10). En plus de résoudre des Snake Cube classique, c'est à dire dont la forme finale est un cube, notre application peut également résoudre des Snake dont la forme finale est plus complexe. Nous avons par exemple créer un Snake dont la forme finale peut être apparenté à un temple maya (`snake_maya_temple.snake`).

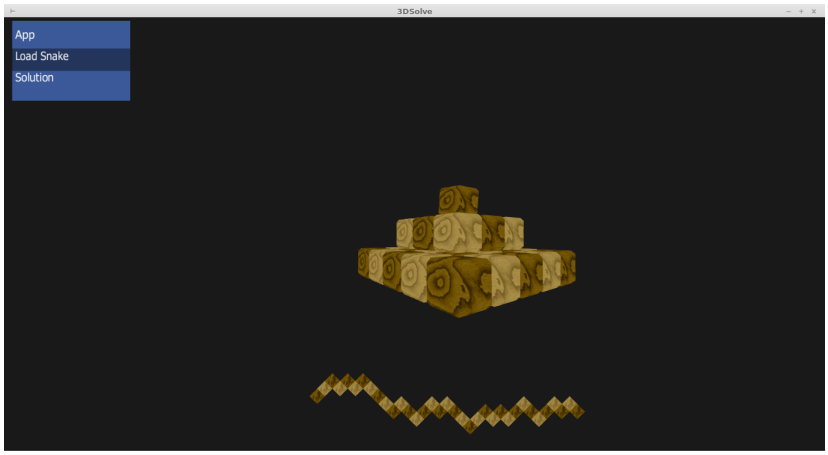


FIGURE 9.1 – "Temple maya" résolu

L'interface de rendu 3D propose à l'utilisateur de visualiser, à son rythme, les diverses solutions menant à la résolution du casse-tête sélectionné.

9.2 Interactivité avec l'utilisateur

L'objectif d'interactivité avec l'utilisateur, autrement dit la possibilité donné à l'utilisateur de résoudre lui-même le casse-tête, est également atteint. En effet, notre application propose une interface 3D permettant à l'utilisateur de manipuler virtuellement le casse-tête et finalement, un pris d'un effort cérébrale non-négligeable, de le résoudre.

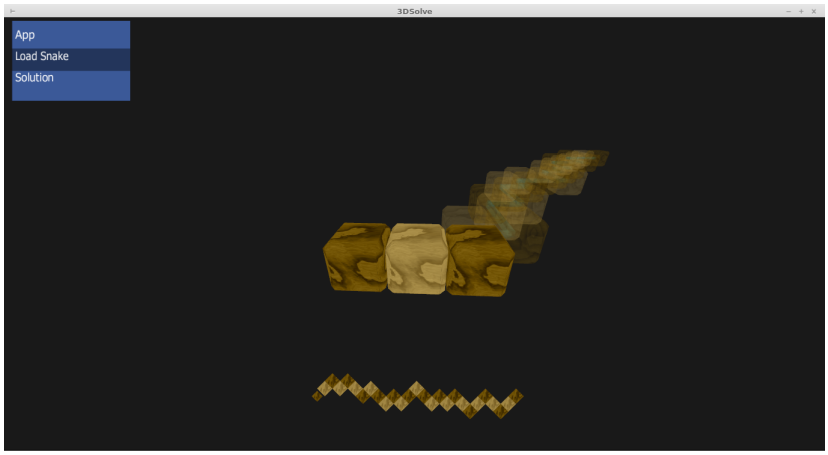


FIGURE 9.2 – Snake Cube déplié en attente de résolution

9.3 Amélioration proposée

Nous avons eu diverses idées que nous aurions aimé implémenté si nous avions eu plus de temps. Nous les avons regroupées ici sous une seul et même proposition d'amélioration.

L'amélioration consiste en un éditeur de snake. Cet éditeur, sous la forme d'une interface dans le contexte de rendu 3D, permettrait à l'utilisateur de créer ses propres casse-tête.

En premier lieu, l'éditeur pourrait proposer à l'utilisateur de définir un volume en 3D puis de passer ce volume à un algorithme qui effectuerait le travail inverse de l'algorithme de résolution. C'est-à-dire qu'à partir du volume qui lui est donné, il créer un ou plusieurs enchaînement d'unité de snake et créé donc un casse-tête. Le snake ainsi créé pourrait ensuite être sauvegardé dans le format définit au chapitre 6 et être ensuite utilisé comme les autres snake dans la partie interactive de l'application.

Ensuite, on pourrait également imaginer que cet éditeur permette à l'utilisateur de définir non seulement un volume mais également l'enchaînement des unités du snake afin de modéliser un snake réel qui n'est pas proposé par l'application.

Chapitre 10

Tests

10.1 Test n°1

Date : 18/05/15
Sur code révisé : commit f7da39f0001d221ccd82f236781b4c97f7f9ca88
OS : Linux Mint 17 Cinnamon 64-bit
Noyau Linux : 3.31.0-24-generic
Processeur : Intel© Core™ i5-3230M CPU @ 2.60GHz * 2
Type d'algorithme : mono-thread
Objectif : Prouvé l'utilité de l'algorithme de réduction des points de départ.

Algorithme	Taille du cube	Temps de calcul (s)	Solutions	Chemins explorés
Actif	2x2x2	0,000429	6	38
	3x3x3	0,047914	32	91 347
	4x4x4	140,208572	11	615 765 110
Inactif	2x2x2	0,005929	144	889
	3x3x3	0,781355	576	1 922 269
	4x4x4	2985,381852	192	11 258 895 649

FIGURE 10.1 – Résultats du test n°1

Interprétation La première interprétation que l'on peut faire de ses résultats, c'est que sans surprise, plus la taille du cube est grande, plus le temps de calcul est important (0,000429 secondes pour un 2x2x2 contre 140,208572 pour un 4x4x4). Ces résultat nous permettent également de prouver l'efficacité de notre algorithme de réduction des points de départ. En effet lorsque l'algorithme est actif on constate que les temps de calcul, tout comme le nombre de chemins explorés, sont plus faibles que lorsque qu'il est inactif.

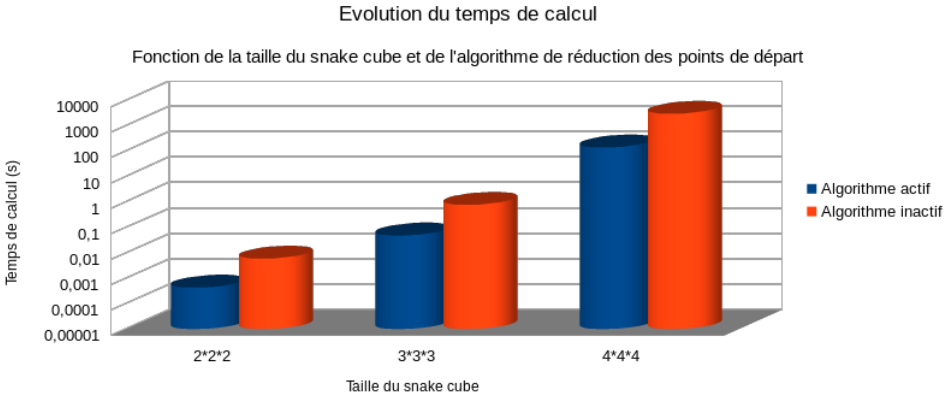


FIGURE 10.2 – Graphique test n°1

10.2 Test n°2

Date : 30/05/15

Sur code révisé : commit c8d1b7bdf5eb1f94255cd66aca9b0b800cba838d

OS : Linux Mint 17 Cinnamon 64-bit

Noyau Linux : 3.31.0-24-generic

Processeur : Intel© Core™ i5-3230M CPU @ 2.60GHz * 2

Type d'algorithme : mono-thread ou multi-thread

Objectif : Évaluer l'efficacité de la parallélisation du calcul des solutions.

Snake 3x3x3	
Nombre de thread	Temps d'exécution (s)
1	0,002462
2	0,001642
3	0,003461
4	0,002232
5	0,002773
6	0,002652
7	0,002292
8	0,001988
9	0,002477
10	0,002387

FIGURE 10.3 – Résultats du test n°2-1

Snake 4x4x4	
Nombre de thread	Temps d'exécution (s)
1	99,067031
2	67,078735
3	65,022525
4	53,198234
5	47,776286
6	71,80401
7	54,967149
8	47,836183
9	45,215047
10	45,38195

FIGURE 10.4 – Résultats du test n°2-2

Interprétation Le temps de résolution du snake 3x3x3 étant très cours, l'efficacité du calcul parallèle est difficile à percevoir. De plus, l'influence des autres processus lancés sur la machine dans l'ordonnancement et donc dans le sur le temps (réel) d'exécution du calcul n'est pas négligeable dans le cas du 3x3x3. Le cas du snake 4x4x4 est plus intéressant. On constate que plus on affecte de thread à la résolution du calcul, moins celui-ci prend de temps jusqu'à ce que l'on arrive à 6 thread. A ce moment là, on observe un "pic" dans le temps de calcul puis il décroit de nouveau. Selon nous, la raison de ce pic est dû à la façon dont nous avons implémenté le système de répartition des points de départ à traiter pour chaque thread. Dans ce cas précis, il y a 10 points de départ à traiter. Si l'on utilise 6 thread de calcul, le programme affecte (10 / 6) = 1 point de départ pour chaque thread, excepté le dernier auquel il affecte (10 - 5) = 5 thread. Finalement, avec ce système, on constate que la parallélisation du calcul n'est vraiment pas optimale, ce qui explique le "pic " dans le graphe.

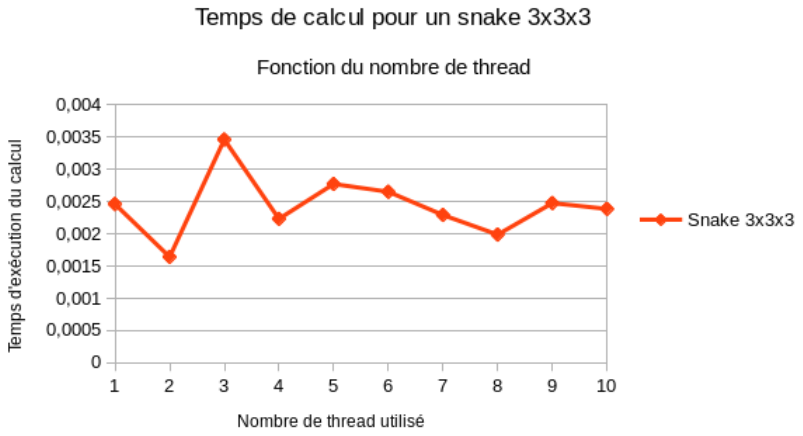


FIGURE 10.5 – Graphique test n°2-1

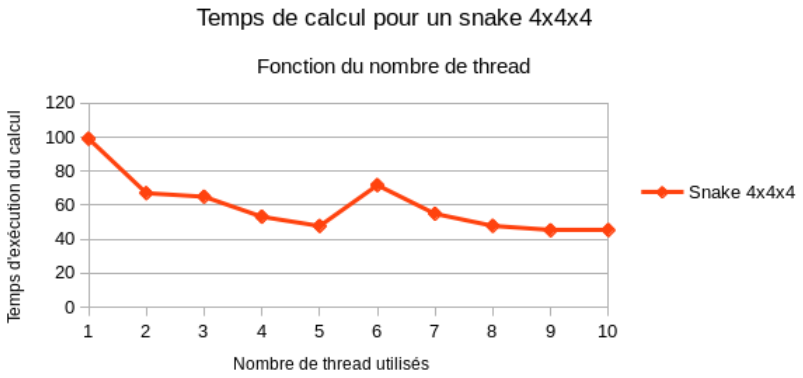


FIGURE 10.6 – Graphique test n°2-2

10.3 Limites

Nos tests nous ont permit de mettre en évidence la limite principale de notre application : la taille du cube à résoudre. En effet, pour un cube 4x4x4 le temps d'attente n'est déjà plus négligeable et sa résolution sollicite grandement le CPU.

Remerciements

Table des figures

Listings