LABYRINTHES 3D

YANN PONTY

`http://www.lix.polytechnique.fr/ ponty/index.php?page=lix431-2010`

1. Introduction

Ce projet consiste en l'implémentation complète d'un jeu vidéo 3D de labyrinthe de type Marbles, dans lequel le joueur doit faire atteindre une sortie à une bille en un temps limité. La bille sera mue par l'inertie, c'est à dire que le joueur ne manipulera pas directement la bille, mais penchera le plateau de jeu par un mouvement de souris, et la bille subira une accéleration dans la direction de plus forte inclinaison. Elle obéira en outre à des règles physiques de base : Détection de collisions, frottements limitant la prise de vitesse de la bille, chocs élastiques lors des collisions ... Les labyrinthes seront engendrés aléatoirement à partir d'arbres couvrants aléatoires construits à partir de pavages, et visualisés grâce à l'extension Java3D. A chaque niveau, le joueur devra faire rallier une position d'arrivée à la bille dans un temps initialement proportionnel à la distance entre la position de départ et d'arrivée, connue du programme au cours de la génération du labyrinthe.

2. Description du projet

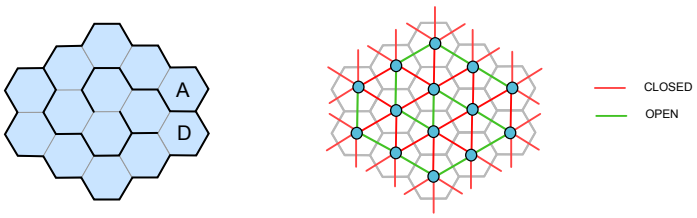
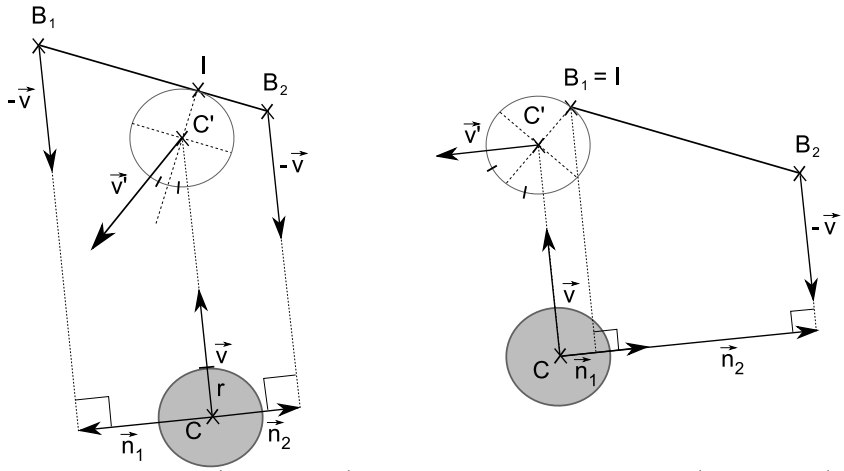


Figure 1. Labyrinthe sur la grille hexagonale et graphe d'adjacence correspondant.

2.1. **Labyrinthes.** On souhaite adopter une vision *assez générale* de ce que peut être un labyrinthe en 2D. On abstrait donc la notion de labyrinthe en un **graphe non-orienté**, dont les sommets sont des **cellules** et l'ensemble des arêtes matérialise l'adjacence des cellules. Chaque arête se verra associer les coordonnées des extrémités de la *frontière* (unique et supposé linéaire) qui sépare les deux cellules, et sera étiquetée par `CLOSED` si celle-ci est infranchissable (Mur), ou `OPEN` sinon. Afin de matérialiser la *frontière extérieure* du labyrinthe, on utilisera un noeud distingué.

En tournant autour des frontières, on pourra ainsi reconstruire le polygone associé à la surface d'une cellule. De tels polygones permettront de faire l'interface entre la représentation logique du labyrinthe (Graphe ou carte) et la représentation 3D. On pourra ainsi localiser la bille au sein du labyrinthe pour déterminer sa cellule actuelle, construire une liste obstacles avoisinant pour des tests de collision, dessiner le labyrinthe ...



Collision *centrale* ($\vec{n_1} \cdot \vec{n_2} < 0$) Collision *de coin* ($\vec{n_1} \cdot \vec{n_2} \geq 0$)

FIGURE 2. Deux types de collisions de la bille sur un mur supposé sans épaisseur.

2.2. **Manipulation de la bille et simulation physique.** La bille sera soumise aux principes physiques de la mécanique de base. On ne considèrera les mouvements et forces que dans leur composantes tangentielles au support (tableau de jeu), et les interactions seront donc essentiellement bi-dimensionnelles, simplifiant grandement les calculs.

2.2.1. *Calcul du vecteur vitesse.* Plus précisément, une bille de masse m, sera dirigée par le joueur à travers l'orientation du plateau de jeu. La composante normale au support du vecteur poids $\vec{P} = m \cdot \vec{g}$ sera annulée par la réaction du plateau et seule subsistera une composante tangentielle $\vec{P_t}$ dont l'expression exacte dépendra du système de coordonnées choisi. Cette composante sera partiellement contrebalancée par un vecteur frottement $\vec{F}$, de direction opposée au vecteur vitesse $\vec{v}$, et de norme proportionnelle à celui-ci, tel que $\vec{F} = -k_f \cdot \vec{v}$. Le vecteur accélération $\vec{a} = \left(\vec{P_t} + \vec{F}\right)/m$ découle de la combinaison de ces deux forces.

En **temps discret**, on simplifie la simulation du déplacement de la bille en *considérant l'accélération comme constante entre deux instants successifs* t et t'. On commence donc par effectuer le bilan des forces, obtenant un vecteur accélération, qu'on multiplie par le temps écoulé $\Delta = t' - t$. On met alors à jour un vecteur vitesse $\vec{v}$, qu'on utilise pour déplacer le centre de masse de la bille, en tenant compte d'éventuelles collisions.

Au alentours du point d'équilibre entre accélération et frottement, la nature discrète du temps risque d'engendrer des à-coups, correspondant à des oscillations autour de la vitesse limite. On pourra résoudre ce problème en précalculant la vitesse limite v^* telle que $\vec{a} = 0$, qu'on utilisera comme norme maximale pour la vitesse.

2.2.2. *Collisions.* Les **collisions** seront traitées itérativement sur des murs initialement supposés sans épaisseur (segments). A chaque collision sur un mur, la bille effectue un **rebond** et on obtient un vecteur vitesse résultant $\vec{v'}$, d'angle explicité par la Figure 2 et de norme atténuée par un choc non-nécessairement élastique. Une fois le choc simulé, on utilise le nouveau vecteur vitesse pour simuler de nouvelles collisions,

jusqu'à ce que celui-ci soit trop faible pour provoquer un nouveau rebond, auquel cas on met à jour la position de la bille et passe au rendu de la scène. On tient ainsi compte des cas où la bille connaît plusieurs collisions entre t et t' (Collision dans un coin exigu, par exemple).

Outre les rebonds sur les murs du labyrinthe, on considérera des **évènements de changement de cellule**, au cours desquels la bille passe d'une cellule à une autre. La prise en compte de ces évènements permettra une localisation de la bille sur le plateau de jeu, ce qui évite d'avoir à tester systématiquement tous les murs pour des collisions potentielles.

Chaque évènement potentiel sera caractérisé par un quadruplet $(T, C', \vec{v'}, N')$ où T est le temps (exprimé en multiple de $\vec{v}$) où un évènement se produit, C la nouvelle position du centre de la bille, $\vec{v'}$ est le nouveau vecteur vitesse et N' la nouvelle cellule. On dispose de $\vec{v}$ le vecteur vitesse et de C le centre de la bille, qu'on utilise pour localiser la cellule courante N.

(1) **Rebonds :** Pour chaque arête c de type `CLOSED` ayant **au moins un point en commun** avec le polygone de la cellule courante, on calcule le temps de collision $T_c = CC'/\|\vec{v}\|$ et le vecteur résultant $\vec{v'}$. Plus précisément, on introduira un **coefficient de restitution** $e \in \mathbb{R}$ qui influera sur l'élasticité des rebonds, modifiant la norme du vecteur vitesse résultant $\vec{v'}$ tel que
$$\left\|\vec{v'}\right\| = e \cdot (\|\vec{v}\| - CC') .$$
On vérifiera en outre que la collision peut effectivement avoir lieu, c'est à dire, entre autres, que le vecteur $\vec{v}$ rapproche initialement C d'au moins une des extrémités B_1 ou B_2 de c et que, dans le cas d'une collision de coin, $\|\vec{n_1}\| < r$ ou $\|\vec{n_2}\| < r$. Sous ces conditions, on ajoute $(T_c, C', \vec{v'}, N)$ à la liste des événements futurs.

(2) **Changement de cellule :** Pour chaque arête c de type `OPEN` adjacente à la cellule, on calcule le temps T'_c auquel le centre C de la bille est compris dans l'intervalle $[B_1, B_2]$, temps auquel la bille entre dans une nouvelle cellule N'. On remarquera que les conditions à tester et les cas à prendre en considérations sont ceux des rebonds (Voir Figure 2) pour une sphère de rayon nul ($r = 0$), et le vecteur résultant est simplement
$$\vec{v'} = \vec{v} - \vec{C'C}.$$
On ajoute alors $(T'_c, C', \vec{v'}, N')$ à la liste des évènements futurs.

(3) Soit $(T^*, C^*, \vec{v^*}, N^*)$ l'événement de temps minimal. Si $T^* < 1$ alors on met à jour la position $C \leftarrow C^*$ de la bille, le vecteur $\vec{v^*} \leftarrow \vec{v^*}$, la cellule courante $N^* \leftarrow N$ et on retourne au point (1). Sinon, on met juste à jour $C \leftarrow C + \vec{v}$ et on passe au rendu de la scène.

On essaiera autant que possible d'éviter les opérations trigonométriques explicites, assez consommatrices et sources d'imprécisions, là où des projections (produits scalaires) suffiront souvent.

2.3. **Visualisation 3D.** La philosophie générale d'un jeu en 3D consiste à afficher de façon répétée une scène, c'est à d'un ensemble d'objets géométriques simples (Sphère et polygones ...) positionnés relativement les uns aux autres par des compositions de transformations géométriques.

La partie logique de la visualisation consistera à construire la scène. Plus précisément, les éléments de jeu suivants devront être présents :

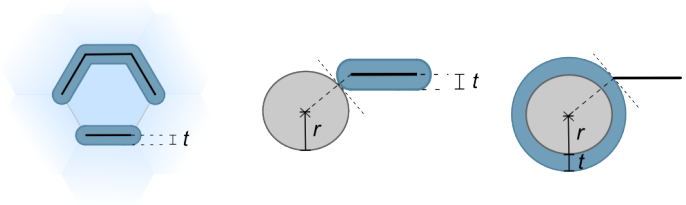


FIGURE 3. Surfaces au sol (En bleu) couvertes par une séquence de murs ou mur d'épaisseur nulle (En noir). Équivalence des collisions d'une bille de rayon r avec un mur d'épaisseur t, et d'une bille de rayon $r + t$ sur un mur d'épaisseur nulle.

- **Plateau de jeu :** Polygone couvrant l'ensemble des cellules, n'ayant pas nécessairement d'épaisseur.
- **Murs :** Afin de faciliter l'algorithmique de la détection des collisions, on supposera que les murs sont d'épaisseur uniforme t et à bord arrondi comme le montre la Figure 3. En effet, les temps de collision et vecteur de réflexion d'une bille de rayon r contre un mur d'épaisseur t sont les mêmes que pour une bille de rayon $r + t$ contre un mur sans épaisseur, et l'on peut alors utiliser l'algorithme de détection brièvement décrit ci-dessus, à des murs
- **Bille :** Une sphère, éventuellement texturée afin de pouvoir en constater les déplacements.

La philosophie générale des moteurs 3D est celle d'un **rendu continu**, dans laquelle le moteur cherche à dessiner autant d'images (Frames) que possible. En Java3D, on pourra étendre la classe abstraite `Interpolateur` (Ou surcharger une de ces classes filles) pour réceptionner les évènements de dessin, et effectuer une mise à jour de la position de la bille selon les principes décrits au chapitre précédent. On pourra aussi s'inspirer de la classe `MouseRotate` pour capturer les mouvements de la souris et orienter le plateau en conséquence.

Cette partie est peut être la plus technique du projet, et il est facile d'y perdre un temps précieux. En particulier, les scènes dessinées par le moteur seront *bloquées* par défaut dans un soucis d'optimisation, et toute modification de la scène (Déplacement de la bille, par exemple) devra déclarée au moment de la création de la scène [1]. Il est donc indispensable de commencer par des exemples *jouet*, et vous trouverez des pointeurs, tutoriels et scénarios d'installation de Java3D sur la page du projet. En particulier, on pourra se référer à l'excellent tutoriel (en français).

`http://rvirtual.free.fr/programmation/java3d/intro.html`

qui date un peu mais dont les exemples illustratifs constituent une excellente entrée en matière.

2.4. **Génération de labyrinthes aléatoires.** A partir du pavage d'une surface quelconque, il existe de nombreux algorithmes de génération aléatoire de labyrinthes, empiriques ou garantissant une propriété précise (Uniformité). Une approche générale pour la création de labyrinthe parfaits, illustrée par la Figure 4, consiste à engendrer un **arbre couvrant du graphe**, i.e. un arbre passant exactement une fois par chaque noeuds du graphe associé au plateau.

1. Voir la méthode `setCapability` de la classe `javax.media.j3d.SceneGraphObject`, primitive de tous les objets d'une scène.

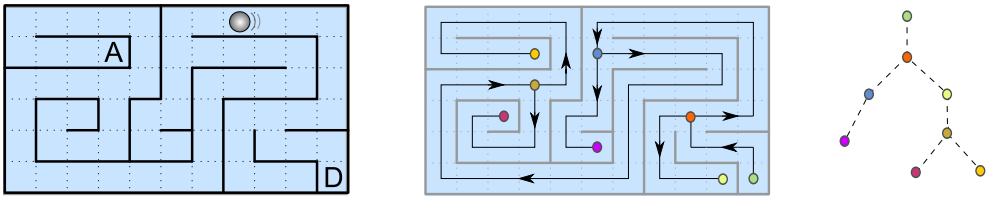


FIGURE 4. Arbre couvrant (spanning tree) sous-jacent à un labyrinthe. Les noeuds unaires de l'arbre couvrant (à droite) sont omis dans un soucis de lisibilité.

On en trouvera quelques stratégies pour engendrer un tel arbre sur la page wikipedia dédiée au sujet :

`http://en.wikipedia.org/wiki/Maze_generation_algorithm`

ou sur la page *think labyrinth*, véritablement encyclopédique sur le sujet :

`http://www.astrolog.org/labyrnth/algrithm.htm`

Plus particulièrement, on engendrera de tels arbres avec l'algorithme de Wilson [1], qui effectue une marche aléatoire dans le graphe en effaçant les cycles. Une fois un tel arbre engendré et son graphe correspondant reconstruit, on place l'entrée et la sortie sur les deux points les plus éloignés dans l'arbre.

On pourra aussi, à faible coût, faire varier les plaisirs en modifiant d'un niveau à l'autre les *constantes* physiques : Masse de la bille m (Accélération), frottement k_f (Vitesse max.), restitution e (Rebonds, effet *flipper* pour $e > 1$).

3. Travail demandé

On devra implémenter les éléments suivants :

- Une bibliothèque de graphe modélisant un labyrinthe.
- Le moteur de simulation physique, calculant à partir d'une position C pour la bille et d'un vecteur vitesse, la prochaine position de la bille.
- Le moteur de rendu, basé sur Java3D.
- La génération aléatoire de labyrinthes par l'algorithme de Wilson, général à tout pavage initial et qui, partant d'un graphe quelconque de labyrinthe (Toutes les arêtes à `OPEN` sauf les murs extérieurs), engendrera un labyrinthe aléatoirement uniformément. On fournira en outre une spécialisation de cet algorithme pour engendrer des labyrinthes sur la grille carrée de dimension $w \times h$.
- Limiter le temps accordé au joueur pour sortir du labyrinthe.

On assemblera tous ces composants en un petit jeu qui pourra, de façon optionnelle :

- Ajouter des bonus (Temps supplémentaire, variation des constantes physiques).
- Implémenter d'autres stratégies de génération de labyrinthes.
- Partir de pavages plus complexes (Hexagonaux, non-réguliers).

Références

1. James Gary Propp and David Bruce Wilson, *How to get a perfectly random sample from a generic markov chain and generate a random spanning tree of a directed graph*, J. Algorithms **27** (1998), no. 2, 170–217.

E-mail address: `yann.ponty@polytechnique.fr`