

## L2 INFORMATIQUE

INFORMATIQUE GRAPHIQUE POUR LA SCIENCE DES DONNÉES

RAPPORT DU PROJET - PRINTEMPS 2025

## Des momies et des pyramides

*Thanh Phat DOAN,  
Naël EL YAZGHI*

## Préface

Ce rapport répond non seulement aux exigences du projet, mais vise également à documenter en détail notre approche, nos choix méthodologiques et les résultats obtenus tout au long du processus. Il constitue ainsi une trace écrite précieuse, reflétant toutes les étapes franchies, consignant les enseignements tirés et garantissant la transparence de notre travail.

## Abstract

# Table des matières

<b>Glossaire</b>	<b>3</b>
<b>A Introduction et Mise en place</b>	<b>4</b>
A.1 Introduction . . . . .	4
A.2 S'éloigner de l'IDE intégré de Processing . . . . .	4
A.3 Maven . . . . .	5
A.3.1 Processing dépendances . . . . .	5
A.3.2 Build commands . . . . .	5
A.4 Structure du projet . . . . .	6
<b>B Système de caméra 3D</b>	<b>7</b>
B.1 Système de coordonnées . . . . .	7
B.2 Déplacement du curseur . . . . .	8
B.3 Rotation de la caméra . . . . .	8
B.4 Mouvements de caméra . . . . .	9
<b>C Pyramide</b>	<b>10</b>
C.1 Cellule . . . . .	10
C.2 Détection efficace des collisions . . . . .	10

---

TABLE DES MATIÈRES

C.3 Labyrinthe . . . . .	11
C.4 Pyramide . . . . .	12
C.5 Rendu de la pyramide extérieure . . . . .	12
C.6 Sol en sable extérieur . . . . .	13
<b>D Momies</b>	<b>15</b>
<b>E Gestionnaire de textures et de PShape</b>	<b>18</b>
E.1 PShape Factory . . . . .	18
E.2 Carte de texture . . . . .	19
<b>F Shaders : éclairage dynamique</b>	<b>21</b>
F.1 Eclairage à l'intérieur de la pyramide . . . . .	21
F.2 Eclairage à l'extérieur de la pyramide . . . . .	22
<b>G HUD : La carte du labyrinthe</b>	<b>23</b>

# Glossaire

**atténuation** Dans le contexte de l'éclairage, l'atténuation fait référence à la diminution de l'intensité lumineuse d'une source lumineuse au fur et à mesure que la distance entre la source et l'objet augmente. Cela est modélisé par une fonction mathématique qui réduit l'intensité lumineuse en fonction de la distance. 2, 17

**blocage de cardan** Le blocage de cardan (ou gimbal lock en anglais) est la perte d'un degré de liberté qui survient quand les axes de deux des trois cardans nécessaires pour appliquer ou compenser les rotations dans l'espace à trois dimensions sont portés par la même direction. 2, 7

**hard-code** hard-code fait référence à la pratique de coder des valeurs directement dans le code source d'un programme, plutôt que de les rendre dynamiques ou configurables. 2, 15

**OOP** Object-Oriented Programming, ou Programmation Orientée Objet en français. 2, 14, 15

**sketch** Dans ce contexte, sketch fait référence à un programme utilisé par Processing (type de fichier .pde) qui consiste en un code écrit en Java pour produire des graphiques 2D et 3D. 2, 3, 5, 15

## CHAPITRE A

# Introduction et Mise en place

## A.1 Introduction

Le projet comprend le rendu 3D d'une pyramide en dessert en utilisant le framework Processing. Voici une liste non exhaustive que nous avons réussi à implémenter pour le projet :

- Système de caméra 3D
- Pyramide à plusieurs niveaux
- Momies
- Gestionnaire de textures et de PShape
- Shaders : éclairage dynamique
- HUD : position du joueur et carte du labyrinthe
- Optimisation : draw calls optimisation avec PShape

Ce rapport tentera de couvrir chacune des fonctionnalités susmentionnées en détaillant notre processus de réflexion, les problèmes rencontrés et les solutions apportées.

## A.2 S'éloigner de l'IDE intégré de Processing

En effet, l'IDE de Processing est pour exécuter des sketch simples sans avoir à se préoccuper de l'importation de bibliothèques ou de la commande de compilation. Cependant, l'inconvénient est qu'il est beaucoup plus restrictif sur la façon dont on organise le projet<sup>1</sup> et qu'il est moins riche en fonctionnalités que d'autres (VSCode, JetBrains, etc).

En s'éloignant de l'« écosystème » Processing, nous pouvons envisager Processing non pas comme un « langage » auquel nous devons nous conformer, mais plutôt simplement comme une bibliothèque Java dédiée au rendu d'images 2D et 3D. Les implications dans le monde réel de cette approche sont importantes, car elle offre une plus grande flexibilité, permettant d'adopter des outils et des technologies plus adaptés à des besoins spécifiques, tout en garantissant une évolutivité de nos projets.

1. Processing permet d'importer plusieurs fichiers .pde afin d'obtenir un code plus modulaire. Cependant, le point reste inchangé.

## A.3 Maven

Pour gérer les dépendances requises par Processing, nous pourrions les spécifier directement dans notre commande de compilation. Toutefois, une approche plus robuste et maintenable consiste à utiliser un gestionnaire de dépendances de Java tel que **Maven**.

**Maven** est un outil d'automatisation de la construction largement adopté dans l'industrie et il est toujours utile de l'apprendre, même si c'est un peu excessif pour ce projet. Nous ne nous intéressons qu'au strict minimum<sup>2</sup> de Maven afin de garder notre système de construction simple et facile à utiliser.

### A.3.1 Processing dépendances

Processing est livré avec sa bibliothèque **core**, qui dépend de deux autres bibliothèques pour les liens Java : **jogl-all** et **gluegen-rt**.

Pour les installer à l'aide de Maven, il suffit de les inclure dans notre fichier pom.xml.

---

```

1 <dependency>
2   <groupId>org.processing</groupId>
3   <artifactId>core</artifactId>
4   <version>3.3.7</version>
5 </dependency>
6
7 <dependency>
8   <groupId>org.jogamp.gluegen</groupId>
9   <artifactId>gluegen-rt-main</artifactId>
10  <version>2.3.2</version>
11 </dependency>
12
13 <dependency>
14   <groupId>org.jogamp.jogl</groupId>
15   <artifactId>jogl-all-main</artifactId>
16   <version>2.3.2</version>
17 </dependency>
```

---

Vous pouvez remarquer que la version de **core** est 3.3.7 et non 4.3.3 (la version 4 de Processing celle que nous utilisons dans le cours). Ceci est dû au fait que pour la dernière version de **core** a besoin des dernières versions de **jogl-all** et **gluegen-rt** (2.5.0), ce qui entraîne de nombreuses complications à cause de l'importation de dépendances supplémentaires, ce qui a entraîné divers bugs sur notre système. Cependant, tant que toutes les fonctions fonctionnent correctement, ce n'est pas notre préoccupation pour l'instant.

Il convient de noter que le système de construction n'a été testé que pour fonctionner sous Linux. Nous ne l'avons pas encore testé sur d'autres systèmes d'exploitation, et le fait d'être multiplateforme n'est pas non plus notre priorité.

### A.3.2 Build commands

Les commandes sont assez simples. Si nous n'avons pas encore installé les dépendances ou si nous venons de mettre à jour pom.xml, dans votre terminal à l'intérieur du dossier du projet (dans le même répertoire

2. Nous avons suivi le guide officiel : <https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>

#### A.4. STRUCTURE DU PROJET

---

que pom.xml), tapez :

---

```
1 mvn clean install -U
```

---

Pour compiler notre projet, il faut :

---

```
1 mvn clean compile
```

---

Puis pour exécuter notre projet :

---

```
1 mvn exec:java
```

---

## A.4 Structure du projet

Tous nos codes sources java se trouvent dans src/main/java/processing/<sup>3</sup> et Main.java contient le point d'entrée du programme (méthode **main**).

Dans Processing, chaque sketch est en fait une classe **PApplet**. Nous laissons donc notre classe Main en hériter (et éventuellement surcharger les méthodes setup() et draw()). Cependant, pour que d'autres classes puissent accéder aux variables de l'instance de PApplet (mouseX, frameRate, etc.), nous devons stocker une référence de cette instance de PApplet dans ces classes. Par exemple :

---

```

1 import processing.core.PApplet;      // Importer la classe PApplet de la bibliothèque Processing
2
3 // Cursor.java
4 public class Cursor {
5     private PApplet context; // Référence à l'instance de PApplet
6     // ... autres attributs et méthodes
7     public Cursor(PApplet context) {
8         this.context = context;
9         // ...
10    }
11    public void func() {
12        System.out.println(context.mouseX); // Accès à mouseX
13    }
14 }
15
16 // Main.java
17 public class Main extends PApplet {
18     public void setup() {
19         Cursor cursor = new Cursor(this); // Passer la référence de l'instance de PApplet
20     }
21 }
```

---

Cette **JavaDoc** couvre toutes les classes et fonctions à de la bibliothèque actuelle **core** de Processing (la moitié d'entre elles ne sont même pas listées dans la documentation officielle!). Cela nous permet d'avoir une contrôle plus précise et donc de réaliser des choses assez cool.

---

3. Cette structure redondante de dossiers imbriqués est due à la convention Maven.

## CHAPITRE B

# Système de caméra 3D

Pour la caméra, nous avons 2 classes dédiées pour la gérer : **Camera.java** et **Cursor.java**. Cursor calcule les distances de rotation de la caméra en fonction du mouvement du curseur (ou de la souris). La classe Camera prendra ces valeurs pour mettre à jour la direction du regard de la caméra et pour mettre à jour la fonction camera() de Processing.

## B.1 Système de coordonnées

Tout d'abord, faisons pivoter le système de coordonnées par défaut de Processing (figure B.1) en un système plus facile à visualiser (figure B.2). Il convient de noter que notre système de coordonnées (ainsi de Processing) est gaucher, c'est-à-dire que d'après la figure B.2, :

- (i) L'axe X est horizontal, les valeurs positives s'étendant vers la droite.
- (ii) L'axe Y est vertical, les valeurs positives s'étendant vers le haut.
- (iii) L'axe Z est perpendiculaire à l'écran, les valeurs positives s'étendant vers l'extérieur du spectateur.

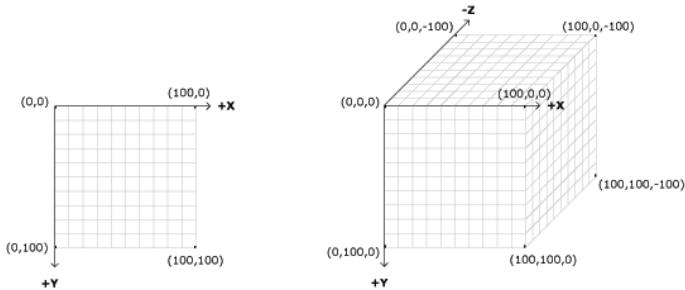


FIGURE B.1 – Système de coordonnées de Processing

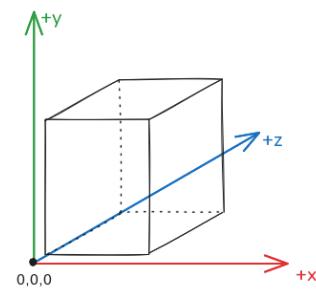


FIGURE B.2 – Rotation de 180° sur l'axe X

Plus précisément, la position des cellules de notre labyrinthe est définie par un système de coordonnées 2D (X, Z), tandis que l'axe Y représente la hauteur de chaque cellule. Ce système rend la caméra plus intuitive : déplacer le curseur sur le plan de l'écran 2D ajuste simultanément la direction du regard en (X, Y) (regarder à gauche et à droite sur l'axe X, en haut et en bas sur l'axe Y).

## B.2 Déplacement du curseur

Lorsque nous déplaçons notre curseur, le point que la caméra vise doit s'ajuster en fonction de la direction du déplacement du curseur. Soit `mouseDelta`, un vecteur 2D représentant la distance parcourue par le curseur par rapport à la frame précédente sur le plan 2D (X, Y) :  $mouseDelta = (mouseX - pmouseX, mouseY - pmouseY)$ . Avec ce delta, nous pouvons effectivement calculer la rotation de notre caméra.

Cependant il y a un problème. Que se passe-t-il si notre curseur atteint le bord de l'écran ? Dans ce cas, disons que nous avons atteint le bord droit de l'écran, nous ne pouvons plus tourner que vers la gauche car le curseur ne peut pas aller au-delà de l'écran. Pour résoudre ce problème, nous devons recentrer le curseur au milieu de l'écran à chaque frame. Pour y parvenir, il n'est pas aussi simple dans Processing de faire  $(mouseX, mouseY) = (width/2, height/2)$  car cela provoque de nombreux comportements anormaux. Après une longue recherche, par chance, nous avons réussi à trouver cet [article de blog](#) qui contient la solution à ce problème.

En outre, pour que le mouvement du curseur soit aussi fluide que dans les jeux FPS, nous effectuons également quelques calculs d'interpolation sur `mouseDelta` avant de le transmettre à la classe Camera. Nous n'entrerons pas dans les détails ici, mais vous pouvez examiner la méthode `updateCursorMovement()` de la classe Cursor.

## B.3 Rotation de la caméra

Pour effectuer une rotation, nous modifions les valeurs (`centerX`, `centerY`, `centerZ`) dans la fonction `camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)` en fonction du déplacement du curseur. Il y a plusieurs façons de procéder, mais prenons le système le plus simple<sup>1</sup>, les angles d'Euler<sup>2</sup>.

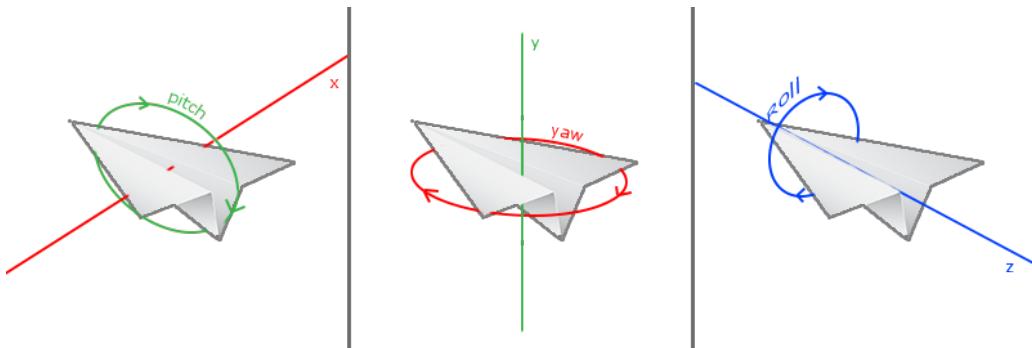


FIGURE B.3 – Angles d'Euler

Soit l'angle thêta (yaw - rotation sur l'axe Y) et l'angle phi (pitch - rotation sur l'axe X). Ces angles sont calculés en fonction du mouvement delta du curseur et de la sensibilité du curseur.

---

<sup>1</sup> `PVector cursorMovement = cursor.setCursorMovement();`  
<sup>2</sup> `this.theta += cursorMovement.x * sensitivity;`  
<sup>3</sup> `this.phi += cursorMovement.y * sensitivity;`  
<sup>4</sup> `this.phi = PApplet.constrain(this.phi, -89.f, 89.f);`

---

1. Nous pouvons également utiliser des quaternions pour obtenir un système de rotation plus robuste.  
2. Ici, nous n'avons pas de problème de blocage de cardan (gimbal lock) car nous n'utilisons que des rotations sur 2 axes (X et Y), et non sur 3.

#### B.4. MOUVEMENTS DE CAMÉRA

Supposons que le vecteur  $v$  indiquant la direction vers l'avant après avoir ramené le curseur au centre de l'écran soit  $(0, 0, 1)$ . Pour trouver le nouveau vecteur  $v'$  après avoir déplacé notre curseur, nous devons appliquer la matrice de transformation  $A$  à ce vecteur. Soient  $R_x(\phi)$  et  $R_y(\theta)$  les matrices de rotation sur les axes X et Y respectivement :

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

$$R_y(\theta) \cdot R_x(\phi) = \begin{bmatrix} \cos(\theta) & \sin(\theta)\sin(\phi) & \sin(\theta)\cos(\phi) \\ 0 & \cos(\phi) & -\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} = A$$

$$v' = A \cdot v = A \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \sin(\theta)\cos(\phi) \\ -\sin(\phi) \\ \cos(\theta)\cos(\phi) \end{bmatrix}$$

Ensuite, nous normalisons  $v'$  et pouvons enfin trouver notre nouveau `(centerX, centerY, centerZ)` pour `camera()`. Supposons que `lookDistance` soit un petit entier positif qui représente la distance entre la caméra et le point que nous regardons. On a donc :

$$\begin{bmatrix} centerX \\ centerY \\ centerZ \end{bmatrix} = \begin{bmatrix} eyeX \\ eyeY \\ eyeZ \end{bmatrix} + lookDistance \cdot v'$$

## B.4 Mouvements de caméra

Grâce à  $v'$ , nous pouvons l'utiliser pour rendre nos mouvements (haut, bas, gauche, droite) plus naturels. C'est-à-dire que notre mouvement suit la direction de la caméra. Nous calculons `forwardVect`, `rightVect` et `upVect`<sup>3</sup> les directions de notre mouvement en fonction de la direction de la caméra.

---

```

1 // Ici lookDir est le vecteur v'
2 forwardVect.set(lookDir.x, 0, lookDir.z);
3 PVector.cross(forwardVect, new PVector(0, -1, 0), rightVect);
4 PVector.cross(forwardVect, rightVect, upVect);

```

---

Lorsque l'utilisateur clique sur une touche de mouvement, un vecteur correspondant est multiplié par la vitesse de mouvement<sup>4</sup> que nous avons définie.

3. En fait, nous n'avons besoin que de `forwardVect` et `rightVect` pour nous déplacer dans le labyrinthe. Cependant, `upVect` nous permet de voler, ce qui est utile pour le débogage.

4. La vitesse (pixels par seconde) est fixe, quelle que soit le `frameRate`. Voir la méthode `updateOnKeyPressed()` de la classe `Camera` pour plus de détails.

# CHAPITRE C

## Pyramide

Pour réaliser une pyramide à plusieurs niveaux dans laquelle un labyrinthe représente un niveau, nous suivons une structure logique d'héritage et de composition entre les classes.

### C.1 Cellule

Le niveau le plus bas, Cell - la cellule d'un labyrinthe, est la classe mère de 2 classes de cellules primaires : PathCell et WallCell. PathCell est également héritée par 2 autres classes de cellules : StartCell et EndCell (qui sont principalement utilisées pour le changement de niveau). Toutes les cellules ont les mêmes attributs, la seule différence est la façon dont elles sont rendues.

Il convient de noter que la classe Cell et Maze héritent de la classe AABB, qui définit la boîte englobante utilisée par notre détecteur de collision.

### C.2 Détection efficace des collisions

En effet, il est très coûteux de vérifier si la caméra est entrée en collision avec chaque cellule du mur du labyrinthe à chaque frame, il y a donc quelques astuces ici.

Soient  $c_{maze}$  l'indice de la cellule du labyrinthe (de  $(0, 0)$  à  $(mazeSize - 1, mazeSize - 1)$ ) et  $c_r$  sa coordonnée réelle dans le monde 3D. Comme chaque cellule a la même taille fixe, étant donné n'importe quelle coordonnée réelle  $c$ , nous pouvons rapidement déterminer son  $c_{maze}$  le plus proche<sup>1</sup> et vice versa.

---

```

1 // returns (i, j, mazeLevel) where mazeLevel = [0..(PYRAMID_SIZE - 1)/2 - 1]
2 public static PVector getCellIndex(PVector coord, int cellSize, int levelHeight) {
3     int mazeLevel = PApplet.floor(coord.y / (cellSize * levelHeight));
4     return new PVector(coord.z / cellSize - mazeLevel,
5                        coord.x / cellSize - mazeLevel,
6                        mazeLevel);
7 }
```

---

Alors le principe est simple : à chaque frame, on convertit le  $c_r$  de la position de caméra en  $c_{maze}$  le plus

1. Nous devons nous assurer que  $c$  est à l'intérieur du labyrinthe, ce qui est facile, avant de faire la conversion

### C.3. LABYRINTHE

proche, puis on vérifie si  $\text{cells}[c_{maze}.x][c_{maze}.y]$  est un WallCell ou non. Cela donne une détection de collision presque  $O(1)$ .

Cependant, il y a un soucis avec cette méthode. Nous vérifions que la position de la caméra n'entre pas en collision avec le mur, mais pas avec le point cible de la caméra (centerX, centerY, centerZ). Cela signifie que, dans certaines situations, si nous serrons trop près du mur, il y a un problème de **clipping** (essentiellement, on voit derrière le mur sans être réellement derrière le mur). Pour résoudre ce problème, au lieu de vérifier la position de la caméra, nous vérifions le carré de délimitation de la caméra dans lequel chaque point est à *lookDistance* de la position réelle de la caméra comme décrit dans la figure C.1.

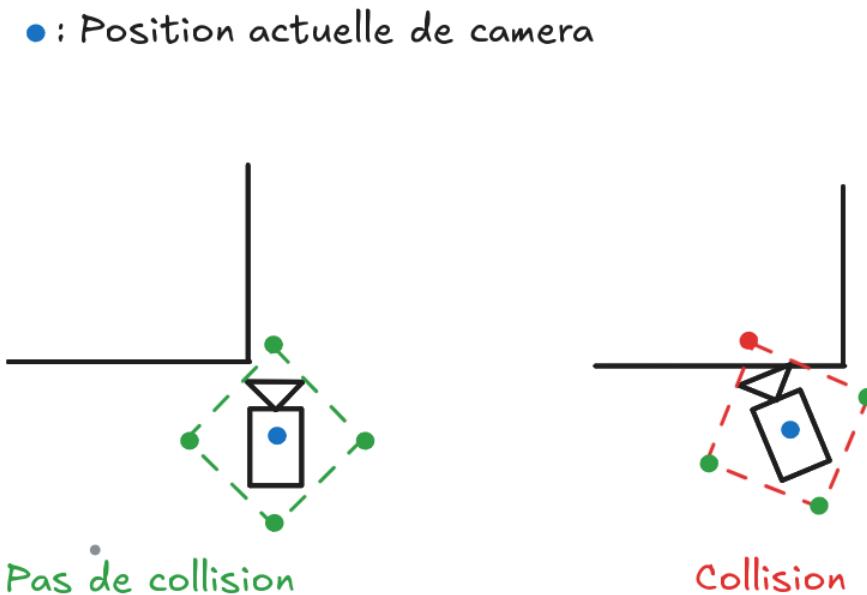


FIGURE C.1 – Clipping

Cette solution n'est certes pas idéale, car elle peut échouer dans certains cas extrêmes. Au lieu d'un carré englobant, il faudrait un cercle englobant, mais cela nécessiterait plus de calculs.

Mais que se passerait-il en cas de collision ? Le plus simple est de ne pas enregistrer la commande de mouvement qui provoque la collision. Notre solution nécessite un peu plus de calculs, mais rend le mouvement plus naturel. C'est-à-dire au lieu de l'interdire complètement, nous « glissons » le long des axes sur lesquels on ne provoque pas de collision avec le mur.

Soit *moveDir* le vecteur pour la prochaine direction de mouvement de la caméra. Si la nouvelle position en collision avec la mur, on décompose *moveDir* en 3 vecteurs représentant 3 axes de mouvement : *horizontalMove* (sur l'axe X), *verticalMove* (sur l'axe Y) et *depthMove* (sur l'axe Z). On vérifie ensuite si chacun de ces nouveaux vecteurs provoquerait ou non la collision avec le mur. Si oui, on le garde sinon on l'ignore<sup>2</sup>.

## C.3 Labyrinthe

La classe Maze contient un tableau 2D pour stocker les cellules du labyrinthe. La majeure partie du code de cette classe est inspirée du TP6, comprenant la fonction de génération du labyrinthe et le calcul du tableau *sides*]. Elle possède un attribut *level* (comme la pyramide comporte plusieurs niveaux) pour faciliter le

2. Pour plus de détails, consultez la méthode *resolveCollision()* dans la classe CollisionDetector

#### C.4. PYRAMIDE

calcul des coordonnées des cellules.

## C.4 Pyramide

La classe Pyramid est également assez simple, principalement un ArrayList pour stocker tous les labyrinthes. Cependant, nous conservons également une référence à Camera car nous devons optimiser le rendu en fonction de la position actuelle de la caméra.

Par exemple, nous n'avons besoin de rendre le labyrinthe que si nous sommes actuellement à ce niveau et non pas de rendre tous les autres labyrinthes. Avec ceci, nous avons juste besoin de changer le rendu des autres labyrinthes seulement si le joueur va à la fin du labyrinthe (EndCell) et monte d'un niveau ou retourne au début du labyrinthe (StartCell) et descend d'un niveau (sauf pour le niveau 0)<sup>3</sup>.

## C.5 Rendu de la pyramide extérieure

L'empilement des labyrinthes les uns sur les autres, chaque labyrinthe étant plus petit que le précédent, permet de former la forme de la pyramide. Cependant, la hauteur de chaque cellule étant relativement importante, la pyramide ne semble pas correspondre à la réalité.

Pour rendre notre pyramide plus belle, nous avons besoin d'une autre classe dédiée **PyramidExterior** pour rendre l'extérieur de la pyramide avec une texture différente de celle que nous utilisons pour l'intérieur. Cependant, il y a quelques défis à relever. Le premier concerne les textures dont nous parlerons au chapitre D. Le second concerne l'optimisation du rendu car notre pyramide est assez grande.

Nous devons d'abord décomposer la pyramide en niveaux encore plus petits (en terme de hauteur). Chaque niveau est décrit comme un carré creux. Pour le rendre, nous pourrions faire de chaque côté de ce carré une boîte rectangulaire. Cependant, à la place, nous devons rendre les faces qui montrent l'extérieur afin de réduire le nombre de vertex qui seront de toute façon cachés (voir figure C.2). Puis, 4 de ces faces<sup>4</sup> forment un étage de la pyramide extérieure (voir figure C.3).

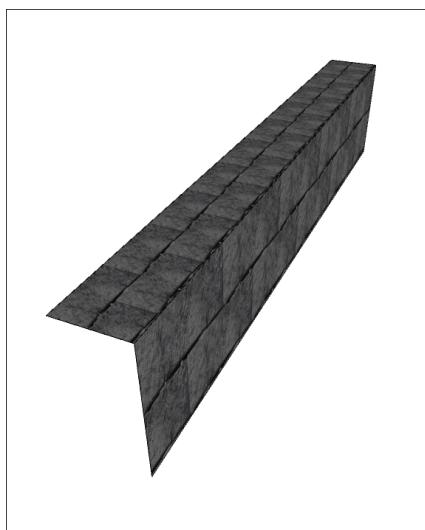


FIGURE C.2 – 1 côté du carré creux

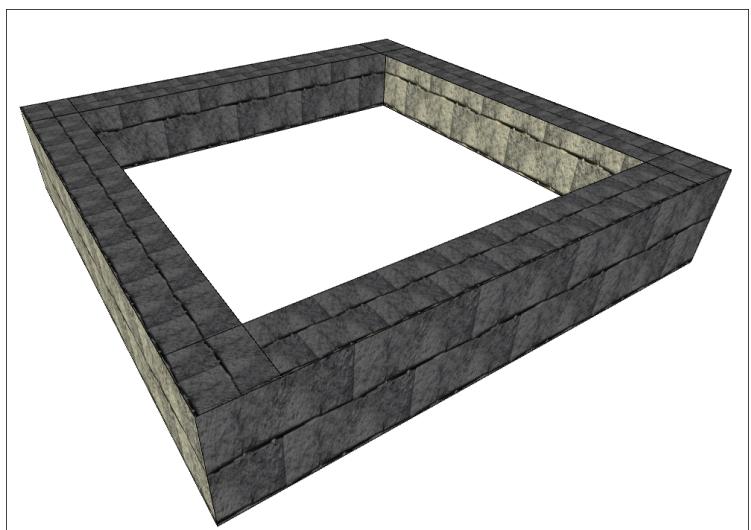


FIGURE C.3 – Carré creux

3. Les détails se trouvent dans la méthode changeLevel() de la classe Pyramide.

4. Il y a un problème de **Z-fighting** dans les coins du carré lorsque les côtés se chevauchent.

### C.6. SOL EN SABLE EXTÉRIEUR

Afin de laisser un espace pour l'entrée sur le niveau le plus bas, 1 côté est combiné par deux barres plus petites (voir figure C.4). Ensuite, on doit également couvrir les côtés de l'entrée en raison de nos barres creuses (voir figure C.5).

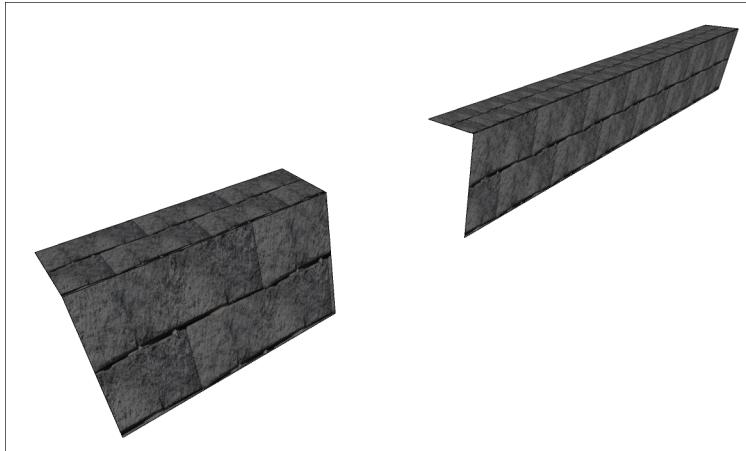


FIGURE C.4 – 1 côté du carré creux avec un espace pour l'entrée

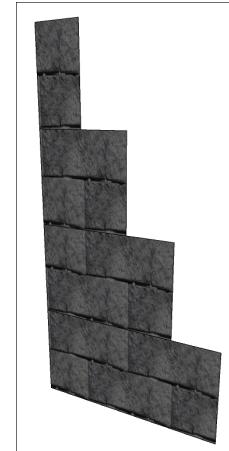


FIGURE C.5 – Le côté de l'entrée

En combinant toutes ces formes, nous obtenons notre belle pyramide avec un minimum de vertex (voir figure C.6).

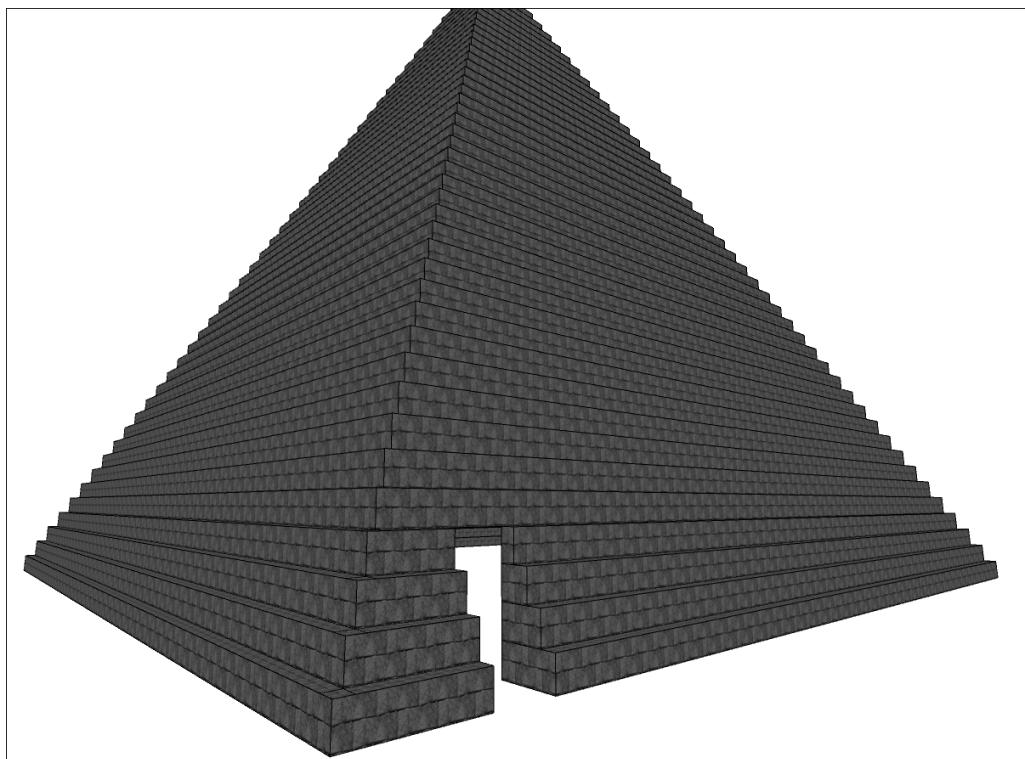


FIGURE C.6 – Pyramide extérieure

## C.6 Sol en sable extérieur

Pour le sol du désert, en suivant les consignes du projet, nous utilisons du bruit de perlin pour rendre le sol un peu rugueux (avec une hauteur variable) au lieu d'un simple rectangle.

## C.6. SOL EN SABLE EXTÉRIEUR

Pour ce faire, nous divisons le sol en cellules (plus les cellules sont nombreuses, plus la surface est « naturelle »). Supposons que  $(x_0, z_0)$  soit le coin inférieur gauche et  $(x_1, z_1)$  le coin supérieur droit d'une cellule. Soient  $y_0, y_1, y_2$  et  $y_3$  les hauteurs des 4 coins de la grille générés par la fonction `noise()`. Nous devons donc créer un carré avec ces 4 coins (4 vertex(s)) :  $(x_0, y_0, z_0), (x_1, y_1, z_0), (x_1, y_2, z_1)$  et  $(x_0, y_3, z_1)$ . Enfin, il faut appliquer la texture de sable (voir figure E.1) à cette cellule.

Cependant, vu de haut, on a l'impression que nous avons appliqué un pattern texture répétitif sur le sol (voir figure C.7). Pour rendre la texture plus variée (comme dans figure C.8), nous avons appliquée une transformation aléatoire (échelle, rotation) pour la texture à chaque cellule.



FIGURE C.7 – Le sol sans transformation



FIGURE C.8 – Le sol avec transformation

Soit  $(0, 0), (1, 0), (1, 1), (0, 1)$  les coordonnées des quatre coins de la texture et  $(c_x, c_y) = (0.5, 0, 5)$  le centre.

Avant d'appliquer la rotation, nous traduisons les coordonnées de sorte que le centre de la texture  $(c_x, c_y)$  devienne l'origine  $(0, 0)$ . Cela nous permet d'effectuer une rotation autour du point central. Soit un point quelconque  $(x, y)$  :

$$d_x = (x - cx) \quad d_y = (y - cy)$$

Pour appliquer l'échelle :

$$d'_x = d_x \cdot scale \quad d'_y = d_y \cdot scale$$

Pour appliquer la rotation, on utilise la matrice de rotation 2D pour l'angle  $\theta$  :

$$A_{2D} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Donc, appliquée à un point  $(d'_x, d'_y)$ , on obtient :

$$\begin{bmatrix} d''_x \\ d''_y \end{bmatrix} = A_{2D} \cdot \begin{bmatrix} d'_x \\ d'_y \end{bmatrix}$$

Les valeurs de `scale` et de  $\theta$  sont aléatoires pour chaque cellule. Pour voir en détails l'implémentation, consultez les méthodes `sandFloor()` et `getRotatedTexCoords()` dans la classe ShapeFactory.

# CHAPITRE D

## Momies

Pour réaliser la momie, nous avons dans un premier temps repris la structure du ressort étudié en TP, en utilisant QUADSTRIP pour constituer le corps de la momie.

Il aurait été plus judicieux d'écrire moins de coordonnées en utilisant simplement un décalage de dx, dy et dz (comme c'était le cas au début). Cependant, il a fallu implémenter toutes ces coordonnées pour effectuer des tests.

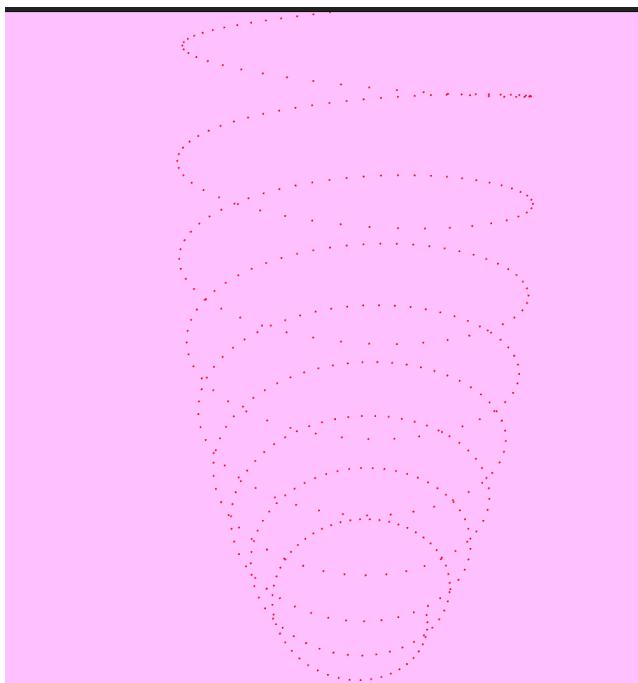


FIGURE D.1

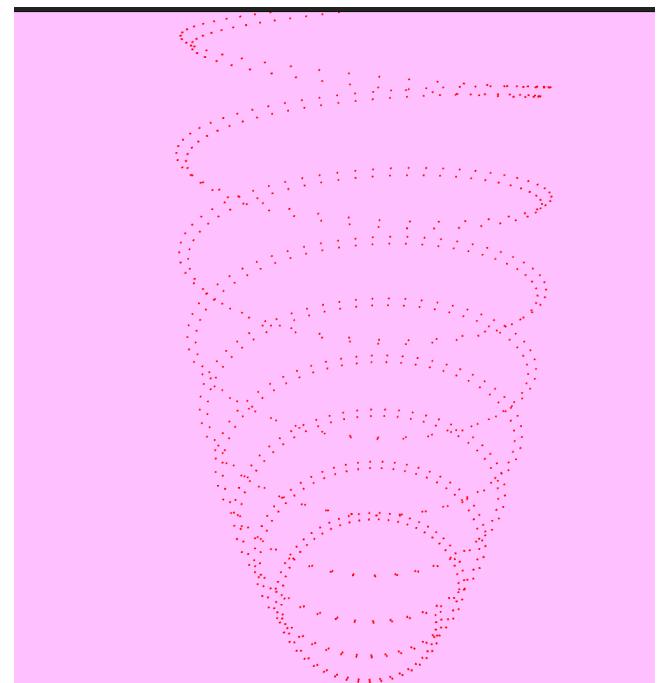


FIGURE D.2

On s'assure que les points sont bien horizontaux et on génère les quad strips. On modifie également le coefficient d'étirement k pour resserrer les bandelettes. Pour pouvoir étirer différentes parties du corps, chacune d'entre elle possède son propre coeff. On applique également une teinte vert foncé pour coller à l'exemple de l'énoncé.

---

```

1 // Corps : le rayon est directement multiplié par bodyThickness
2 void corps() {
3     float kCorps = 13;

```

```

4   float da = 4 * PI / 100;
5   int nb_de_tours = 10;
6   int iter_i = nb_de_tours * 50 / 2;
7   for (int i = -iter_i; i <= iter_i; i++) {
8     float a = i / 50.0 * 2 * PI;
9     float t = map(i, -iter_i, iter_i, -1, 1);
10    float R1 = (150 + 100 * (1 - t * t)) * bodyThickness;
11    float offset = R1 * offsetFactor;
12
13    float x1 = R1 * cos(a) * scaleFactor;
14    float y1 = R1 * sin(a) * scaleFactor;
15    float z1 = a * kCorps * scaleFactor;
16
17    float x2 = R1 * cos(a + da) * scaleFactor;
18    float y2 = R1 * sin(a + da) * scaleFactor;
19    float z2 = (a + da) * kCorps * scaleFactor;
20
21    float x3 = R1 * cos(a) * scaleFactor;
22    float y3 = R1 * sin(a) * scaleFactor;
23    float z3 = z1 + offset * scaleFactor;
24
25    float x4 = R1 * cos(a + da) * scaleFactor;
26    float y4 = R1 * sin(a + da) * scaleFactor;
27    float z4 = z2 + offset * scaleFactor;
28    // ...
29  }

```

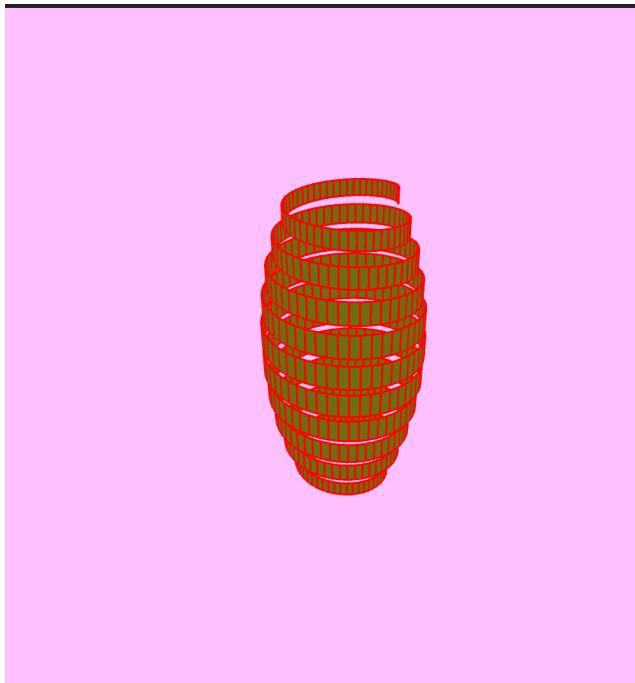


FIGURE D.3

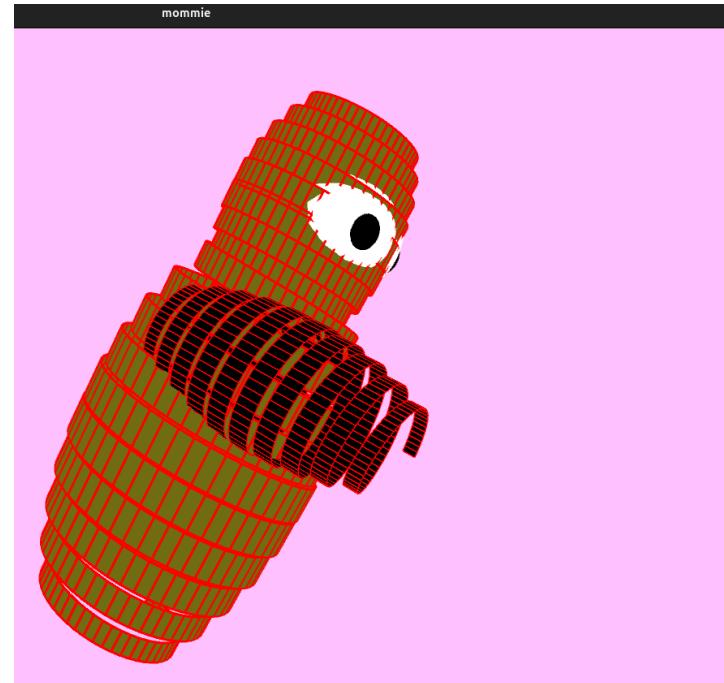


FIGURE D.4

Par la suite, nous avons dupliqué la forme initialement créée afin de générer la tête et les bras. Cette duplication a permis de conserver la cohérence de la structure tout en nous évitant de réécrire du code. Afin d'ajuster la proportion et de jouer sur la dynamique de la momie, nous avons utilisé maintes fois le coefficient d'élargissement K, déjà présent dans l'algorithme du ressort, pour allonger ou compresser les segments comme bon nous semble. Pour apporter du détail au visage, nous avons intégré deux sphères pour représenter les yeux. À l'intérieur de ces sphères, de plus petites sphères de couleur noire constituent les pupilles.

Ensuite, nous avons ajouté les modèles de mains fournis avec l'énoncé, complétant ainsi l'architecture

du personnage. Pour éviter un rendu trop « dessiné », nous avons retiré les strokes sur l'ensemble des formes. Nous avons également étiré les pupilles pour obtenir cette forme d'amande

---

```
1 scale(pupilSizeFactor, pupilSizeFactor * 0.6, pupilSizeFactor);.
```

---

Enfin, dans une optique d'optimisation des performances, nous avons regroupé toutes les composantes de la momie (corps, tête, bras, yeux et mains) au sein d'un unique PShape. Cette consolidation permet non seulement de simplifier le rendu, mais également de réduire le nombre d'appels à la fonction de dessin, ce qui est essentiel pour maintenir un frame rate élevé tout au long du projet.

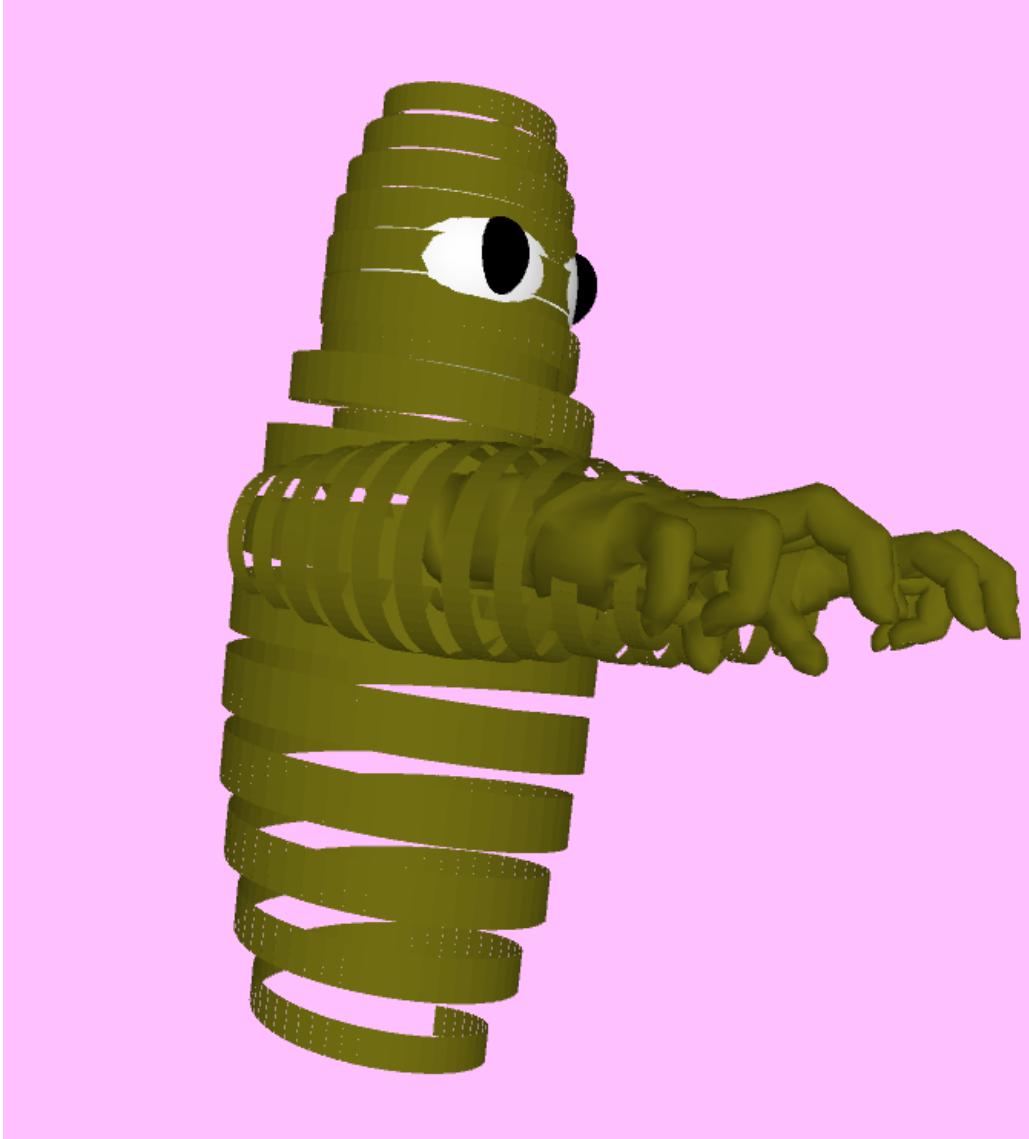


FIGURE D.5 – La momie

# CHAPITRE E

# Gestionnaire de textures et de PShape

## E.1 PShape Factory

La plupart des objets étant formés à l'aide des mêmes formes, donc on doit pouvoir disposer de fonctions renvoyant **PShape** et pouvant être appelées de n'importe où.

Pour y parvenir dans le style de la OOP, on a une classe dédiée ShapeFactory qui contient toutes les méthodes statiques. Nous devons juste nous assurer qu'elle est initialisée au début du programme<sup>1</sup> avec le contexte PApplet et la carte de texture<sup>2</sup>.

---

```

1  public class ShapeFactory {
2      private static PApplet context;
3      private static PImage textureMap;
4      // ...
5      public static void init(PApplet context, PImage textureMap) {
6          ShapeFactory.context = context;
7          ShapeFactory.textureMap = textureMap;
8          // ...
9      }
10
11     // Exemple d'une méthode statique
12     // La méthode renvoie un côté d'une cellule à 6 côtés en utilisant la texture des pierres.
13     public static PShape square0(float width, float height) {
14         setDefault();
15         PShape s = context.createShape();
16         s.beginShape(PApplet.QUADS);
17         s.normal(0f, 0f, -1f);
18         s.texture(textures);
19         s.noStroke();
20         s.vertex(0, 0, 0, 0, 0);
21         s.vertex(width, 0, stones_w, stones_h);
22         s.vertex(width, height, 0, stones_w, stones_h);
23         s.vertex(0, height, 0, 0, stones_h);
24         s.endShape();
25         return s;
26     }
27 }
```

---

1. Par convention, il est préférable de suivre le **singleton pattern**, mais si on se assure que seule notre classe **Main** peut l'initialiser, cela ne pose pas de problème.

2. On va en parler dans la section suivante

## E.2. CARTE DE TEXTURE

Cette classe contient toutes les PShape que nous utiliserons (cellules de mur, pyramide extérieure, sol du désert, etc). Il faut noter que pour les cellules murales, nous ne rendons pas la boîte entière mais nous avons des côtés individuels (square0, square1, ...). Cela permet aux classes d'avoir plus de contrôle sur ce qui est rendu (optimisation des murs inspirée de TP6 par exemple).

## E.2 Carte de texture

Nous avons beaucoup de formes et chacune d'entre elles doit être appliquée avec des textures différentes. On peut charger plusieurs images dans ShapeFactory et chaque méthode peut charger la texture dont elle a besoin. Cependant, l'utilisation abusive de cette méthode peut entraîner des problèmes de performance car le GPU doit constamment changer de texture.

Pour des sketchs simples de Processing, ce n'est pas un problème car on peut organiser à la main l'ordre des PShape afin de pouvoir charger les textures en conséquence (en regroupant les PShape ayant la même texture à rendre ensemble). Par exemple :

---

```

1 // PShapes avec la texture de sable
2 texture(sand);
3 shape(floor);
4
5 // PShapes avec la texture de pierre
6 texture(stones);
7 shape(walls);
8 shape(ceiling);

```

---

Cependant, avec notre structure OOP, il devient très difficile pour nous de garder une trace au fur et à mesure que le projet grandit. La meilleure méthode conventionnelle est donc de combiner toutes les textures dans un seul grand canevas que nous appelons une **carte de texture**<sup>3</sup> (voir figure E.1). En stockant les coordonnées de chaque texture (dans un fichier externe ou le **hard-code**), nous limitons au minimum le nombre de changements de textures.

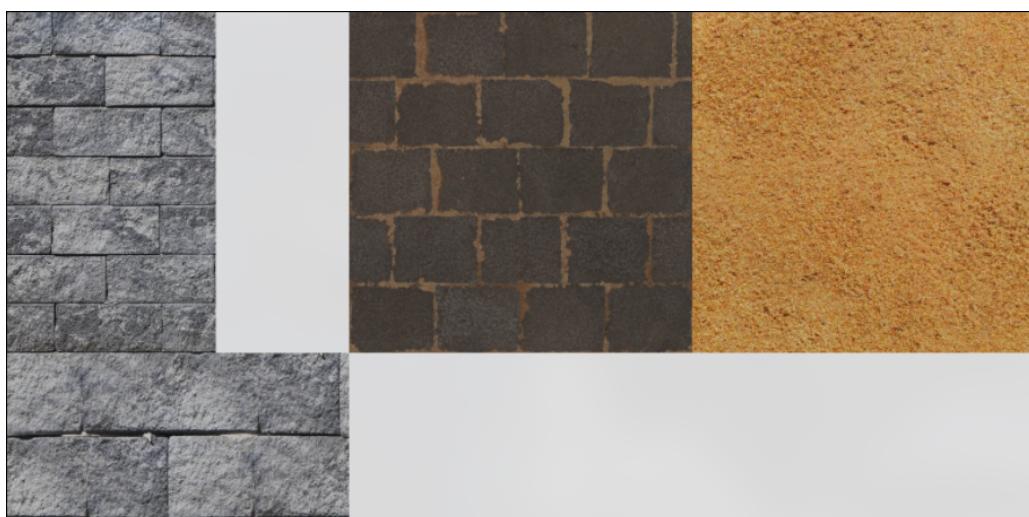


FIGURE E.1 – Carte de texture

<sup>3</sup>. Cette carte de texture est tuilée par 512x512 à l'aide de l'application **Gimp**. Elle se trouve dans /src/main/resources/assets/

## E.2. CARTE DE TEXTURE

On n'a pas complètement éliminé le changement de texture, car dans Processing, si on a besoin d'une texture avec un pattern répété (pour l'extérieur de la pyramide et le sol du désert), on doit extraire la texture que on veux pour ce pattern et la charger séparément.

---

```

1  public static void setRepeat() {
2      context.textureMode(PApplet.NORMAL);
3      context.textureWrap(PApplet.REPEAT);
4  }
5
6  public static void setDefault() {
7      context.textureMode(PApplet.IMAGE);
8      context.textureWrap(PApplet.CLAMP);
9  }
10 // On doit extraire la texture ayant besoin de REPEAT pattern de la carte de texture
11 brickTex = textures.get(brick_x, brick_y, brick_w, brick_h);
12
13 // Si pattern CLAMP, on peut spécifier directement les coordonnées de la texture dans la carte de texture
14 public static PShape square0(float width, float height) {
15     setDefault();
16     //...
17     s.texture(textures)
18     s.vertex(0, 0, 0, 0, 0);
19     s.vertex(width, 0, stones_w, 0);
20     s.vertex(width, height, 0, stones_w, stones_h);
21     s.vertex(0, height, 0, 0, stones_h);
22     // ...
23 }
24
25 // Si pattern REPEAT, on doit spécifier le ratio (le nombre de fois que la texture est répétée)
26 // et cela est seulement possible en Processing si la texture est séparée
27 public static PShape py_box(float l, float h) {
28     setRepeat();
29     // ...
30     s.texture(brickTex);
31     s.tint(238,232,170);
32     s.vertex(0, h, 0, 0, 0);
33     s.vertex(l, h, 0, ratio, 0);
34     s.vertex(l, 0, 0, ratio, 1);
35     s.vertex(0, 0, 0, 0, 1);
36     // ...
37 }
38

```

---

Cela dit, comme notre environnement est statique (c'est-à-dire que la texture de la forme ne change pas du tout au cours du programme), cela ne coûte qu'un peu de performance au démarrage.

# CHAPITRE F

## Shaders : éclairage dynamique

### F.1 Eclairage à l'intérieur de la pyramide

Nous pourrions faire de l'éclairage comme inspiré de TP6, c'est à dire mettre spotlight() sur les murs qui sont considérés comme des culs-de-sac (définis par sides[]). Cependant, ce n'est pas aussi réaliste et nous voulions nous défier en codant le shader pour simuler le caractère tenant une lampe de poche en utilisant spotlight(), c'est-à-dire que la lumière devant la caméra illumine les cellules du labyrinthe (avec également l'atténuation) et la position de la source de lumière est mise à jour en fonction de la position et de la direction de la caméra.

On a d'abord suivi ce [guide](#) recommandé par le cours dans la partie **Texture-light shaders** et puis on ajoute la fonction d'atténuation, qui est de la forme :

$$A = \frac{1}{C_0 + C_1 \cdot d + C_2 \cdot d^2}$$

où  $d$  est la distance entre la source de lumière et le pixel/fragment,  
 $C_0$ ,  $C_1$  et  $C_2$  sont des constantes d'atténuation,  
 $A$  est la valeur d'atténuation de la lumière à ce pixel/fragment.

Ici  $C_0 = 1.0$  et les valeurs  $C_1$ ,  $C_2$  sont ajustées jusqu'à ce que nous nous sentons bien et réalistes (les objets près de la caméra sont brillamment éclairés et ceux qui sont plus éloignés sont atténués).

Maintenant, nous avons 2 choix, soit l'éclairage par vertex et l'éclairage par pixel (plus réaliste mais plus lourd sur GPU). Cependant, j'ai décidé d'aller avec par pixel car il y a un boîtier de bord avec l'éclairage par vertex est que lorsque nous nous rapprocherons trop près, il deviendra noir. Pour expliquer ce phénomène, voyons en détail comment fonctionne l'éclairage diffus (voir figure F.1).

Soit  $\theta$  l'angle entre le rayon lumineux (ligne noire) et le vecteur normal de la surface de notre objet (flèche jaune). Plus  $\theta$  est grand, moins la lumière devrait avoir d'impact sur la couleur du fragment. Mathématiquement, il s'agit d'un produit de points entre ces deux vecteurs. Cela fonctionne bien avec le shader par pixel (fragment shader) car chaque fragment a son propre vecteur normal et calcule sa propre valeur d'intensité lumineuse. Il y a cependant une différence cruciale avec le vertex shader.

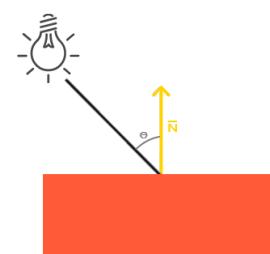


FIGURE F.1 – Eclairage diffus

## F.2. ECLAIRAGE À L'EXTÉRIEUR DE LA PYRAMIDE

Parce qu'un rectangle est formé de 4 vertex(s), chacun situé à un coin, ses vecteurs normales sont donc  $n_1, n_2, n_3, n_4$ , comme le montre la figure F.2. Si la source lumineuse s'approche trop près de la surface du rectangle, nous pouvons voir que l'angle qu'elle forme avec tous les autres vecteurs normaux est de 90 degrés, ce qui fait que la surface apparaît noire même si nous nous trouvons directement en face d'elle.

Nous pourrions utiliser une valeur d'éclairage ambiant pour atténuer ce problème, mais ce n'est toujours pas souhaitable car cela rendrait les éclairs pas du tout réalistes et introduirait des effets secondaires indésirables.

Dans le fragment shader, pour combiner l'intensité de l'atténuation diffuse et de l'atténuation de la lumière en fonction de la distance, on utilise la formule suivante :

$$I = \max(0, v_D \cdot ecNormal) \cdot A$$

- $v_D$  est le vecteur de direction entre la source lumineuse et le fragment,
- $ecNormal$  est le vecteur normal du fragment,
- $A$  est la valeur d'atténuation de la lumière à ce pixel/fragment,
- $I$  est l'intensité lumineuse finale à ce pixel/fragment.

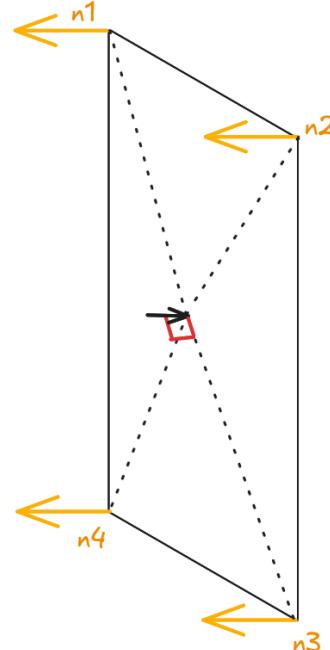


FIGURE F.2 – Cas limite du vertex shader

## F.2 Eclairage à l'extérieur de la pyramide

Nous voulons également que l'éclairage illumine l'environnement extérieur mais n'affecte pas l'intérieur (comme la lumière du soleil). Il y a plusieurs façons de le faire.

Nous pourrions spécifier une autre source de lumière à l'intérieur du shader qui mettrait son éclairage ambiant au maximum, en d'autres termes, nous traiterions cette source de lumière comme un soleil. Cependant, afin de ne pas affecter les objets qui se trouvent à l'intérieur de la pyramide, nous devons effectuer des tests de profondeur (par z-buffer), ce qui est assez difficile.

Une autre solution est d'avoir 2 shaders, un pour la lampe de poche à l'intérieur de la pyramide et un pour l'extérieur avec l'éclairage ambiant au maximum. Il suffit de passer de l'un à l'autre selon que l'on se trouve ou non à l'intérieur de la pyramide.

Pour éviter de changer de shader (pour des raisons de performance), nous introduisons une variable **isSunlit** à la place dans le shader. Si nous sommes à l'extérieur, l'intensité lumineuse est maximale, sinon elle est calculée selon la formule ci-dessus. De même, nous introduisons une variable **hasTexture**<sup>1</sup> afin de pouvoir gérer différemment les fragments qui ont ou n'ont pas de texture<sup>2</sup> sans changer de shader.

<sup>1</sup> `uniform float isSunlit;`  
<sup>2</sup> `uniform float hasTexture;`

Pour voir en détail l'implémentation, consultez les fichiers **lightTextureFragRealistic.glsl** et **lightTextureVertRealistic.glsl** dans le dossier `/src/main/resources/shaders/`.

1. `isSunlit` et `hasTexture` sont tous deux de type `float` et non `boolean` car ils sont plus optimisés pour le GPU.  
2. Sinon toute surface qui n'a pas de texture sera rendue entièrement noire.

# CHAPITRE G

## HUD : La carte du labyrinthe

L'affichage de la carte du labyrinthe est inspiré du TP6, puisqu'il suffit de stocker sides[4] pour chaque cellule afin d'indiquer ou non l'affichage de cette cellule sur la carte. Lorsque le personnage est dans la cellule  $(i, j)$ , il suffit de marquer toutes les cellules environnantes dans un rayon de 1 pour qu'elles soient affichées sur la carte. Nous affichons également une petite boule verte qui se met à jour à chaque frame en fonction de la position du joueur.

Pour le rendre les choses un peu plus raffiné, nous pouvons rendre le cône de vue que la caméra regarde (voir figure G.1) pour une navigation plus facile. Il suffit d'utiliser le vecteur de direction du regard  $v'$  que nous avons calculé dans chapitre B et de rendre le cône dans l'espace 2D.

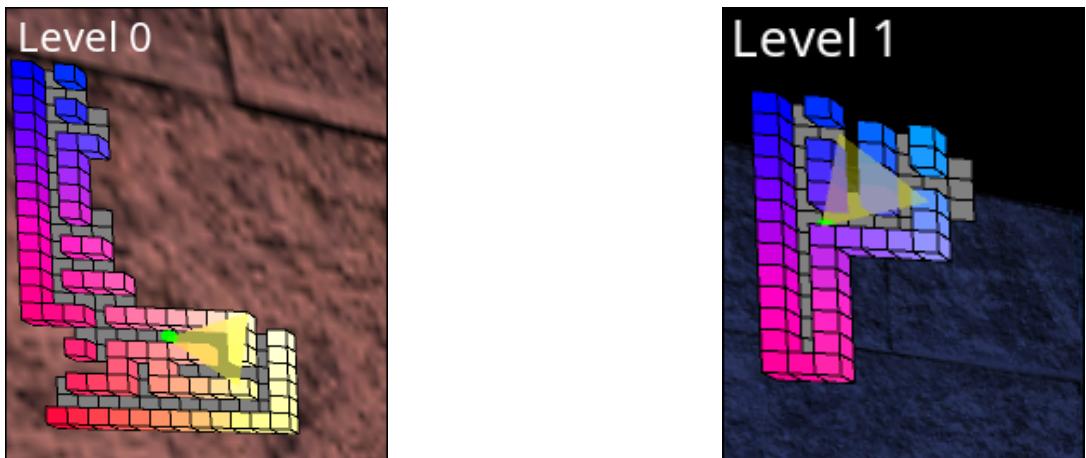


FIGURE G.1 – Carte du labyrinthe

On le map dans l'espace 2D :

$$v'' = (v'_x, v'_z)$$

Normaliser, puis trouver l'angle  $\alpha$  entre l'axe X positif (Est) et  $v''$  :

$$\alpha = \arctan 2(v''_y, v''_x)$$

Suppose l'angle FOV est  $\beta$ , on peut calculer les coordonnées des 2 points de la base du cône :

$$\theta_L = \alpha - \frac{\beta}{2}$$

$$\theta_R = \alpha + \frac{\beta}{2}$$

$$\vec{L} = \vec{P} + l \cdot (\cos(\theta_L), \sin(\theta_L)) \quad \text{où } \vec{P} \text{ la position 2D du personnage sur la carte,}$$

$$\vec{R} = \vec{P} + l \cdot (\cos(\theta_R), \sin(\theta_R)) \quad l \text{ la longueur du cône,}$$

$\vec{L}, \vec{R}$  le côté gauche et le côté droit du cône

L'implémentation concrète est la méthode **createFOVCone()** dans HUD.java.