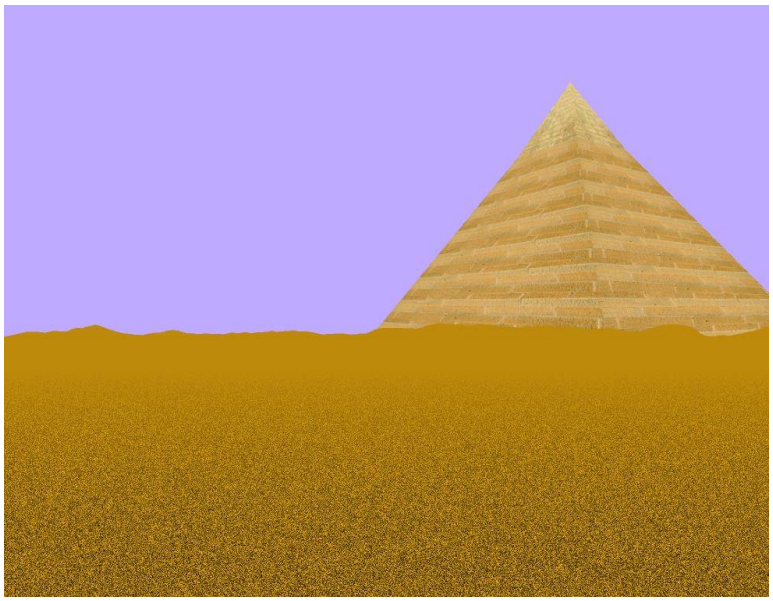
université
PARIS-SACLAY

Université Paris Saclay

Faculté des Sciences d'Orsay

Des momies et des pyramides

Informatique graphique pour la science des données



Binôme :

- ZIMOUCHI Assim
- HIDOUCHE Ouarda

L2 Informatique – Groupe 01

Julien BERRY

université
PARIS-SACLAY

Table des matières

Julien BERRY

université
PARIS-SACLAY

Introduction :

Le projet « Des momies et des pyramides » consiste à concevoir deux modèles 3D interconnectés, en s'inspirant des concepts développés lors du dernier TP sur les labyrinthes. D'une part, l'objectif est de modéliser une momie en faisant tourner un quad-strip dont le rayon varie afin de simuler les différentes parties du corps , du ventre au cou en passant par la tête et en y ajoutant des éléments décoratifs comme les yeux et, éventuellement, les bras. D'autre part, le projet vise à transformer un labyrinthe en une pyramide en empilant plusieurs niveaux de labyrinthes de taille décroissante, tout en intégrant un extérieur désertique et un sol dont la topographie est modulée par un bruit de Perlin.

Objectifs du projet :

- **Modélisation 3D :**
  • Créer une momie à l'aide d'un quad-strip à rayon variable.
  • Transformer un labyrinthe en pyramide en empilant des niveaux de taille décroissante.
- **Texturation et Éclairage :**
  • Appliquer des textures adaptées.
  • Utiliser des shaders GLSL pour un rendu lumineux et réaliste.
- **Interactivité :**
  • Mettre en place des animations fluides.
  • Intégrer une minimap pour la navigation dans le labyrinthe.
  • Gérer le passage de niveau à l'intérieur de la pyramide.

Julien BERRY

université PARIS-SACLAY

Construction de la pyramide :

La construction de la pyramide a été réalisée en adoptant une approche modulaire, basée sur la création d'étages superposés pour former la structure pyramidale. Chaque étage est généré de manière procédurale et correspond à une instance d'une **DonneesEtage** qui stocke notamment la grille du labyrinthe de cet étage ainsi que des décalages pour l'aligner dans l'espace.

Étapes Clés de la Construction

- **Détermination de la Réduction Progressive :** Dans le constructeur de la classe **Pyramide**, le nombre d'étages et la taille de base sont définis. Pour chaque étage, la taille du labyrinthe est diminuée de manière régulière (réduction de 2 unités par étage) afin de simuler l'effet de rétrécissement typique d'une pyramide. Cela permet d'obtenir des niveaux de plus en plus petits au fur et à mesure que l'on monte.

- **Création et Positionnement des Étages :** Pour chaque niveau, la grille correspondante est générée en utilisant la méthode de génération de labyrinthe (via la classe GenerateurLabyrinthe)

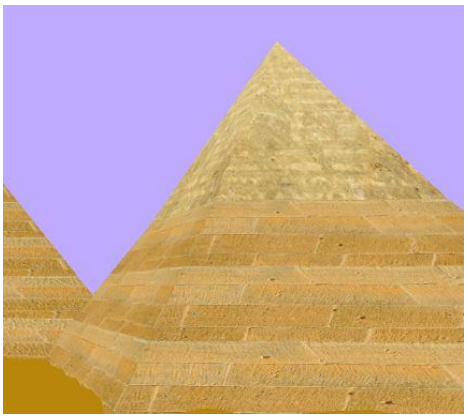
- **Positionnement 3D :** Des décalages sont calculés pour centrer chaque étage sur l'axe central de la pyramide. Les décalages en X et Y permettent de centrer la structure, tandis que le décalage en Z (calculé en fonction de la hauteur entre niveaux) positionne verticalement chaque étage.

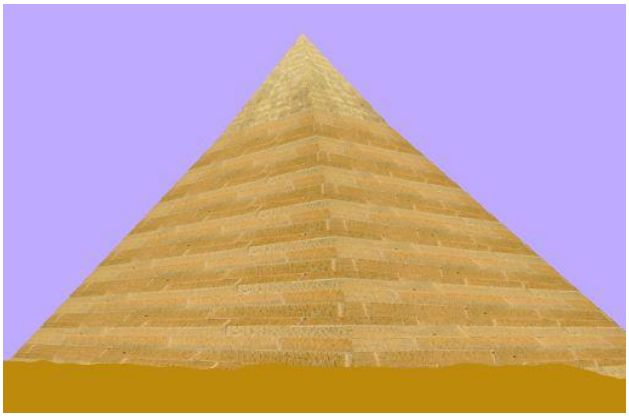
- **Génération de la Coque de la Pyramide :** Une fois les étages générés, la méthode **creerCoquePyramide()** assemble la “coque” extérieure de la pyramide. Pour chaque paire d'étages consécutifs, le code calcule les coins (les bords inférieurs de l'étage de base et les bords supérieurs de l'étage adjacent) et dessine des faces (des quads) qui relient ces deux niveaux. Ces faces sont texturées avec une image de pierre, ce qui donne à la pyramide une apparence authentique et solide

- **Détails et Finitions :** La méthode **dessinerSommetPyramide()** est utilisée pour créer le sommet de la pyramide. En se servant d'un maillage triangulaire et d'une texture spécifique on obtient un “chapeau” lisse qui vient compléter l'aspect global.

Julien BERRY

université
PARIS-SACLAY





Construction du Sable et du Désert

Le terrain sablonneux dans ce projet a été conçu en deux couches complémentaires : **Une surface principale animée** et détaillée, générée par la classe **Sable** à l'aide d'un shader, Et **un terrain périphérique statique**, rendu grâce à la méthode **drawDesert()**, servant de sol autour des pyramides.

Génération du Terrain Dynamique (Classe Sable) : Cette classe est utilisée pour générer une surface de sable dynamique, texturée et éclairée, qui peut représenter une zone de dunes en mouvement ou en surélévation.

Étapes Clés de la Construction :

• **Initialisation Paramétrée :** Dans le constructeur, on définit la taille de la grille, l'espacement entre les points, l'amplitude des hauteurs et la fréquence du bruit (pour simuler les vagues du sable).

• **Relief par Bruit de Perlin :** Chaque point de la grille est assigné à une hauteur générée via le bruit de **Perlin,** créant des dunes aléatoires réalistes.

• **Application d'un Shader Sable :**

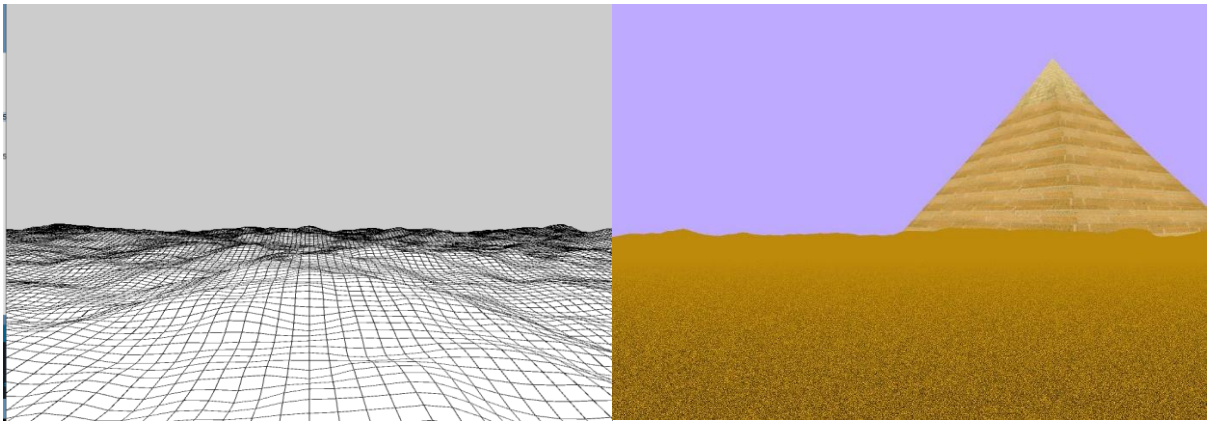
Un shader GLSL (sable_frag.glsl) est utilisé pour ajouter :

- du grain de sable (via un bruit pseudo-aléatoire),
- une lumière directionnelle pour renforcer les ombres,
- et une couleur naturelle de sable.

Julien BERRY

université
PARIS-SACLAY

• **Affichage 3D avec Quads :** Les points sont connectés via des **QUAD_STRIP**, formant une surface continue qui donne une impression de terrain fluide et réaliste.



Génération du Sol Environnant : Méthode drawDesert ()

Cette méthode a pour rôle de générer le désert statique qui entoure les pyramides, formant une base cohérente sur laquelle repose toute la scène 3D.

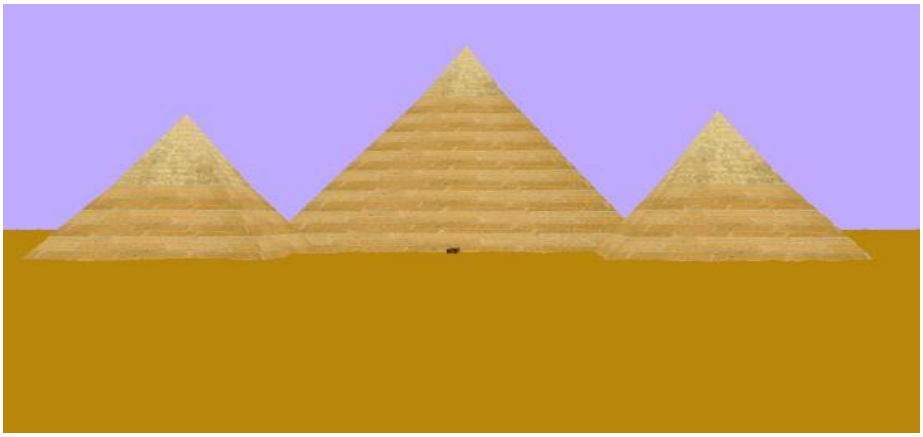
Étapes Clés de la Construction :

• **Définition de la Zone :** La méthode reçoit les paramètres de taille, de résolution, d'échelle du bruit et de hauteur maximale. Cela permet de définir un vaste terrain plat ou légèrement ondulé.

• **Relief Sablonneux Basé sur le Bruit :** Comme pour la classe Sable, le relief est basé sur du bruit de **Perlin,** mais ici utilisé de façon plus discrète (plus étalé) pour créer un terrain relativement plat avec quelques ondulations.

• **Rendu du Terrain :** Le sol est dessiné en utilisant des **TRIANGLE_STRIP**, assurant une continuité fluide entre les lignes. Il est rempli avec une couleur sable uniforme pour s'accorder avec l'univers visuel de la scène.

université
PARIS-SACLAY



Construction des Labyrinthes Intérieurs

Les labyrinthes intérieurs de la pyramide sont conçus pour offrir une expérience immersive en termes de navigation et de visuel. Ils sont générés de manière procédurale grâce à une combinaison de classes et de méthodes, permettant de créer un labyrinthe dynamique et interactif. Le processus de génération repose sur la logique de creuser un espace en suivant des règles de labyrinthes classiques, tandis que l'affichage est rendu avec une texture réaliste des murs et du sol.

Génération du Labyrinthe : Classe GenerateurLabyrinthe

La classe GenerateurLabyrinthe est responsable de la création et de la gestion du labyrinthe. Elle fonctionne de manière procédurale pour générer une grille de taille définie, remplir la grille avec des murs et des chemins.

Étapes de la Génération :

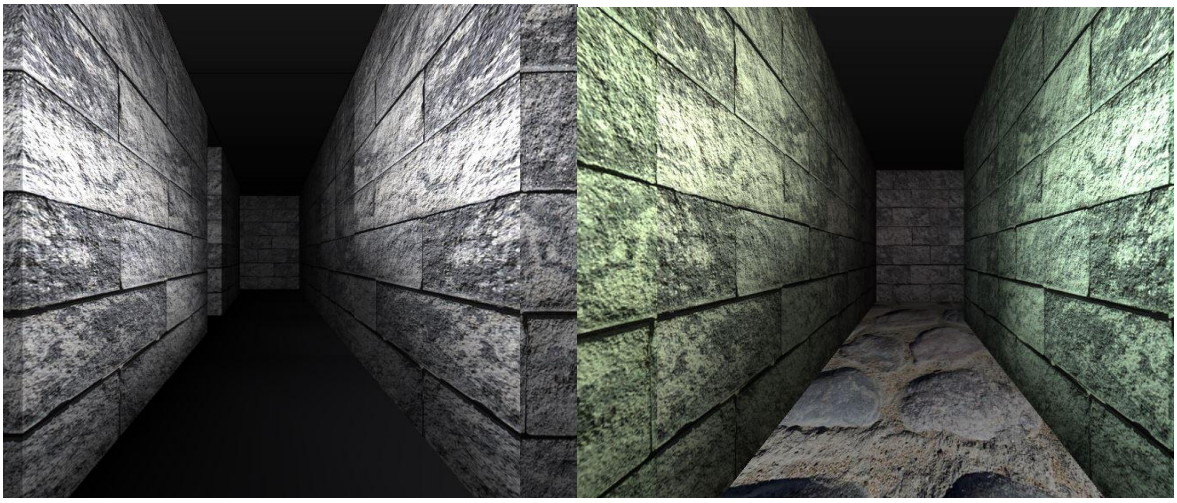
- **Initialisation des Murs et Chemins** : La grille est remplie avec des murs et des chemins
- **Comptage des Zones à Creuser** : L'algorithme identifie les zones ouvertes et calcule combien de murs doivent être creusés.
- **Création des Chemins :** En utilisant un algorithme aléatoire, des chemins sont creusés pour créer un labyrinthe.
- **Définition des Entrées et Sorties :** L'entrée est située en haut à gauche et la sortie en bas à droite.
- **Définition des Côtés des Cellules :** Cette étape permet de savoir où se trouvent les murs et d'ajouter des détails sur chaque côté de la cellule (haut, bas, gauche, droite).

Affichage 3D du Labyrinthe : Classe AffichageLabyrinthe

L'affichage du labyrinthe dans un environnement 3D est géré par la classe AffichageLabyrinthe. Elle est responsable de la création des formes géométriques pour chaque mur et chaque partie du sol du labyrinthe, ainsi que de l'application des textures.

Julien BERRY

université
PARIS-SACLAY



Coloration dynamique des murs du labyrinthe :

Les murs du labyrinthe sont colorés en fonction de leur position sur la grille. Chaque mur dans le labyrinthe est défini par des coordonnées (i, j), et ces coordonnées influencent directement la couleur du mur. Ce système permet de générer des murs avec des couleurs variées en fonction de leur emplacement dans la grille, créant ainsi un effet visuel intéressant et non uniforme.

Méthode de calcul des couleurs : La couleur des murs est calculée à partir des coordonnées des murs sur la grille du labyrinthe. Plus précisément :

- Composante Rouge (R) : Elle est déterminée en fonction de l'index de ligne `j`. L'intensité du rouge varie linéairement en fonction de la position verticale du mur dans la grille.

- Composante Verte (G) : Elle est déterminée par l'index de colonne `i`. Comme pour le rouge, l'intensité de la couleur verte varie en fonction de la position horizontale du mur.

- Composante Bleue (B): Elle est calculée comme la différence entre 255 et la somme des valeurs de rouge et de vert, assurant ainsi que la couleur reste dans les limites de l'espace RGB. Cela crée une dynamique intéressante où les couleurs des murs varient en fonction de la somme des valeurs des composantes RGB, tout en garantissant un contraste et une visibilité optimisés.

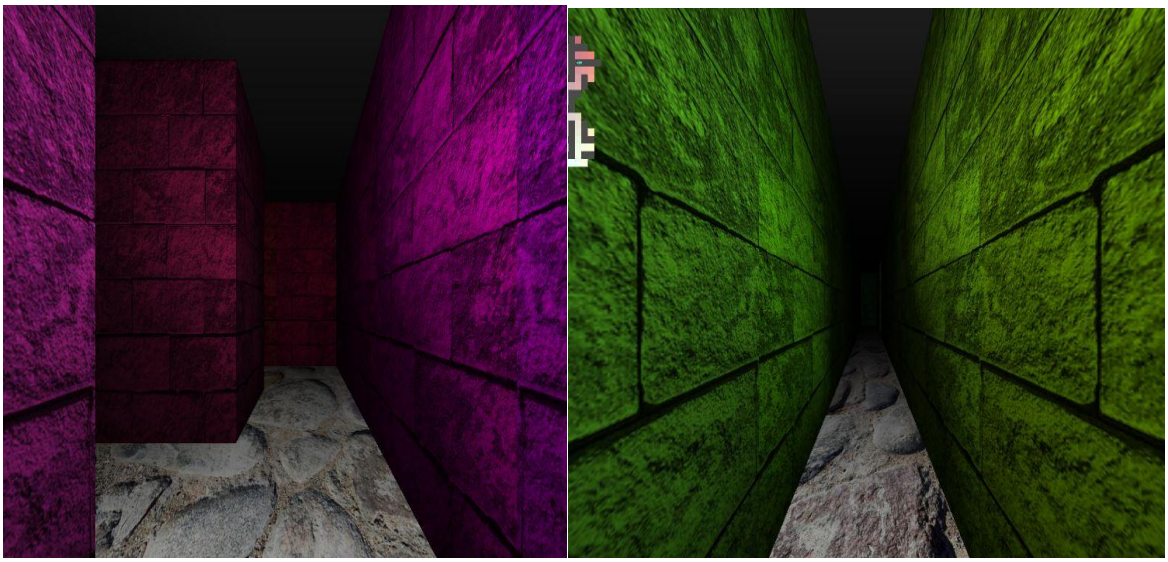
Application de la couleur aux murs : Une fois les composantes RGB calculées, la couleur est appliquée à chaque mur individuellement grâce à la méthode setWallColor(int x, int y, color c). Cette méthode prend en entrée les coordonnées du mur (x, y) et la couleur calculée, puis applique cette couleur au mur correspondant.

Extrait du code de la méthode d'application des couleurs :

Julien BERRY

université
PARIS-SACLAY

```
  public void applyColorToWalls() {
    for (int j = 0; j < labyrinthe.getSize(); j++) {
        for (int i = 0; i < labyrinthe.getSize(); i++) {
            if (labyrinthe.getLabyrinthe()[j][i] == '#') {
                // Calcul des composantes RGB basées sur la position
                float r = map(j, 0, labyrinthe.getSize()-1, 0, 255);
                float g = map(i, 0, labyrinthe.getSize()-1, 0, 255);
                float b = abs(255 - r - g);
                color wallColor = color(r, g, b);

                // Appliquer la couleur
                setWallColor(i, j, wallColor);
            }
        }
    }
}
```



Gestion de la Transition entre les Étages

La transition entre les différents étages du labyrinthe dans le jeu est gérée par une série d'animations et de contrôles de visibilité afin de garantir une expérience immersive et fluide pour le joueur. Lorsqu'un joueur termine un étage, un message de fin de niveau est affiché, accompagné d'une animation de transition qui dure 3 secondes. Ce processus utilise une animation de fondu d'une couleur ambre-or,

créant un effet visuel qui masque progressivement l'interface de jeu. Ce processus est déclenché par la pression d'une touche (la barre d'espace), permettant au joueur de passer à l'étage suivant.

Étapes Clés :

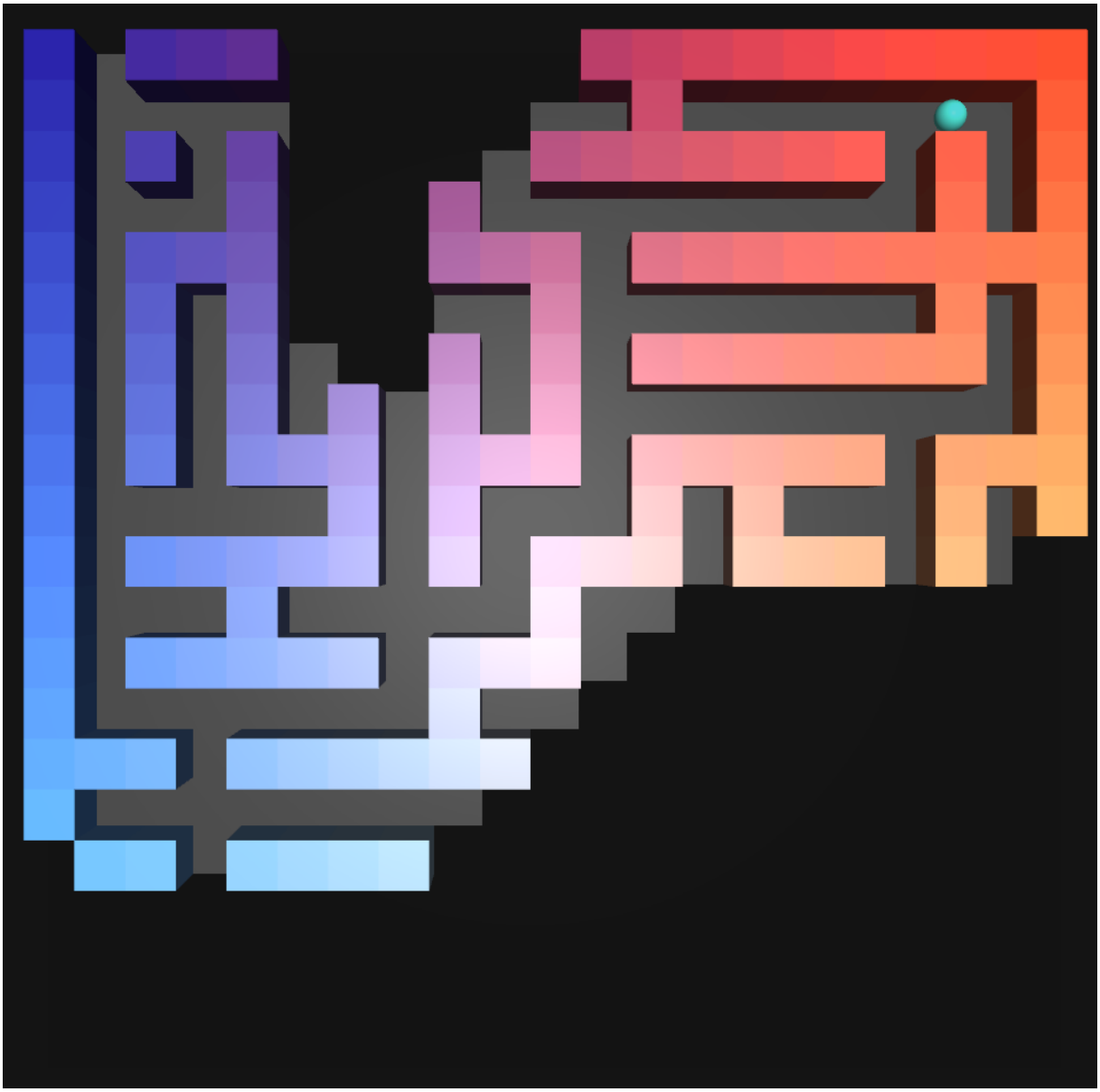
- **Affichage du Message de Fin de Niveau :** Lorsque le joueur termine un étage, un message de fin de niveau est affiché. Ce message reste visible pendant 20 secondes pour permettre au joueur de prendre connaissance de son accomplissement avant de continuer. Ce délai est géré par une temporisation dans le code. Le message est stylisé avec un fond dégradé, un cadre et un texte animé pour rendre l'expérience visuellement attrayante.

- **Animation de Transition :** L'animation de transition est réalisée en modifiant progressivement la couleur d'arrière-plan de l'écran vers une teinte ambre-or. Cette animation utilise un fondu de couleur qui dure 3 secondes. Pendant cette période, l'interface de jeu devient progressivement opaque, masquant ainsi l'écran et offrant un effet visuel agréable pour la transition.

- **Réinitialisation de la Position du Joueur et de la Caméra :**Une fois la transition terminée, la position du joueur et la direction de la caméra sont réinitialisées pour l'étage suivant. Ce processus est géré par des variables qui stockent la position actuelle et la direction du joueur. Lors du passage à un nouvel étage, la position du joueur est ajustée en fonction de la configuration spécifique de ce dernier.

- **Mise à Jour de la Minimap et du Labyrinthe :**Après la transition, le jeu recharge la minimap avec les nouvelles dimensions et configurations du labyrinthe, adaptées au nouvel étage.

Mini-Map Interactive du Labyrinthe



Julien BERRY

université
PARIS-SACLAY

Fonctionnement de la Mini-Map

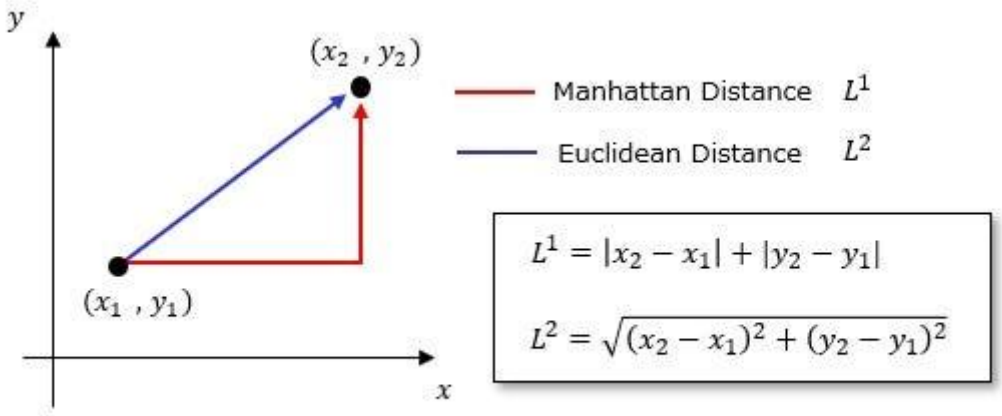
La mini-map est une représentation 2D simplifiée du labyrinthe, affichée en superposition dans l'interface. Elle permet au joueur de s'orienter en temps réel en visualisant :

- Les murs et le sol : Par rapport à la correction du TP 6, les murs sont dessinés via une fonction auxiliaire qui dessine un cube en 3D et supprime les traits entre chaque cube, rendant le résultat plus élégant que l'utilisation de 'box'.
- La position du joueur : la représentation du joueur est désormais une flèche, permettant de mieux se repérer qu'une sphère affichant également vers quelle direction le joueur se dirige.
- Les momies : la représentation des momies se fait par un triangle jaune.
- Découverte progressive des zones : Seules les zones proches du joueur (dans un rayon de 3 cases) sont révélées, créant un effet de brouillard. La détection utilise la **distance de Manhattan**

```
public void updateDiscoveredArea() {
  // Parcourir la zone autour du joueur selon la distance visible
  for (int y = max(0, playerY - VIEW_DISTANCE);
       y <= min(labyrinthe.getSize() - 1, playerY + VIEW_DISTANCE);
       y++) {

    for (int x = max(0, playerX - VIEW_DISTANCE);
         x <= min(labyrinthe.getSize() - 1, playerX + VIEW_DISTANCE);
         x++) {

      // Calcul de la distance de Manhattan
      int distance = abs(x - playerX) + abs(y - playerY);

      // Marquer comme découvert si dans le rayon visible
      if (distance <= VIEW_DISTANCE) {
        discovered[y][x] = true;
      }
    }
  }
}
```

Mesure la distance entre deux points sur une grille, en suivant uniquement les axes horizontaux et verticaux. Calculée par : $|x_1 - x_2| + |y_1 - y_2|$.

Julien BERRY



Exemple : Entre (1,1) et (4,3) → **3** (différence en x) + **2** (différence en y) = **5**.

Gestion des Performances

Seules les cases découvertes sont rendues, réduisant le nombre d'objets à dessiner.

Utilisation des Shaders pour la Minimap

Les shaders sont utilisés pour simuler un éclairage réaliste sur la minimap, même en 2D. Le Shader LabyTexture prépare les données géométriques et d'éclairage pour chaque sommet et s'occupe de la transmission des données.

Le shader LabyColor calcule la couleur finale de chaque pixel en combinant les éléments suivants :

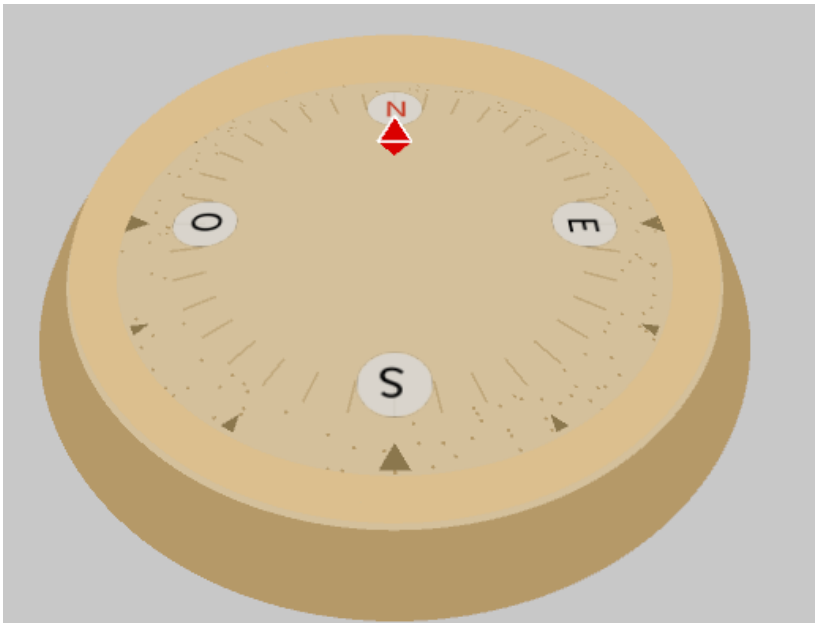
```
// Lumière de base constante.
float ambientStrength = 0.3;
vec3 ambient = ambientStrength * vec3(1.0, 1.0, 1.0);

// Intensité selon l'angle entre normale et lumière.
float diff = max(dot(normal, vertLightDir), 0.0);
vec3 diffuse = diff * vec3(1.0, 1.0, 1.0);

// Reflets brillants.
float specularStrength = 0.5;
vec3 viewDir = normalize(-vertPosition);
vec3 reflectDir = reflect(-vertLightDir, normal);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
vec3 specular = specularStrength * spec * vec3(1.0, 1.0, 1.0);

// Combinaison des effets.
vec3 result = (ambient + diffuse + specular) * vertColor.rgb;
gl_FragColor = vec4(result, vertColor.a);
```

université
PARIS-SACLAY

Impact sur la Minimap

On retrouve une profondeur visuelle, les murs et sols paraissent en relief grâce aux ombres et reflets. Donne une cohérence avec l'environnement 3D. C'est la même logique d'éclairage que la pyramide, appliquée à la minimap. L'impact sur les sont aussi notable. Les calculs sont exécutés sur le GPU, libérant le CPU pour d'autres tâches.

La boussole :



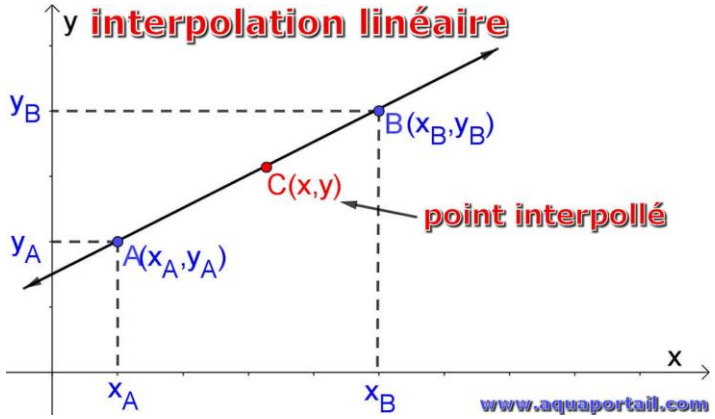
Fonctionnalités Principales

La boussole est un élément clé de l'interface utilisateur qui indique au joueur les directions cardinales (Nord, Est, Sud, Ouest) et s'aligne dynamiquement avec l'orientation de la caméra. Elle se compose des éléments suivants :

- Un cadran circulaire avec des marqueurs de direction stylisés
- Une flèche rouge animée indiquant le Nord
- Des points cardinaux (N, E, S, O) positionnés autour du cadran
- Un système d'animation fluide lors des changements de direction

Optimisations Techniques

- Hiérarchie de PShape : Permet de pré-calculer les éléments statiques
- PGraphics séparé : Pour le texte des points cardinaux
- Système d'animation : Interpolation linéaire pour des rotations fluides



- Paramètre scaleFactor : Adapte la taille à l'interface. Évite le déplacement des éléments lors de l'utilisation de la fonction scale

Cette implémentation combine esthétique de boussole ancienne et animations fluides, tout en restant performante grâce à l'utilisation optimale des PShape.

Modélisation de la Momie

La modélisation de la momie repose sur une approche modulaire organisée autour d'un objet PShape principal de type GROUP. Ce conteneur hiérarchique agit comme une structure maîtresse dans laquelle sont intégrées les différentes parties anatomiques : **le corps**, **la tête**, **les yeux**, **les bras** et **les mains**. Chaque élément est généré séparément, puis ajouté à l'ensemble via la méthode addChild(). Cette architecture permet de manipuler la momie dans son ensemble tout en gardant une flexibilité sur ses composants.

```
momie = createShape(GROUP);
PShape corps = creerCorpsMomie();
PShape tete = creerTeteMomie();
PShape yeux = creerYeuxMomie();
PShape bras = creerBrasMomie();
PShape mains = creerMainsMomie();

momie.addChild(corps);
momie.addChild(tete);
momie.addChild(yeux);
momie.addChild(bras);
momie.addChild(mains);
```

Corps : Le corps de la momie est une structure cylindrique irrégulière, construite dynamiquement à l'aide de la primitive QUAD_STRIP. Cette forme permet de relier deux bandes de

sommets pour créer une surface continue. Des irrégularités sont introduites dans le rayon et dans la couleur pour simuler l'effet de bandelettes enroulées et usées. Le rayon est modulé par une fonction cosinus, et les couleurs sont influencées par un bruit de Perlin.L'itération se fait sur un axe vertical, chaque bande étant légèrement décalée en hauteur et en angle (angleA + da), ce qui crée un effet torsadé et non uniforme, renforçant l'illusion de bandages enroulés.

Tête : La tête est générée avec la même technique que le corps, mais avec des dimensions plus réduites et un rayon conique. Le code utilise un QUAD_STRIP similaire, avec une hauteur plus faible et une modulation du rayon pour évoquer la forme d'un crâne :

```
float radius2 = 16 + 34 * cos(i * PI / 195) + cos(angle2);

head.vertex(radius2 * cos(angle1), radius2 * sin(angle1), depth + i);
```

La couleur continue d'être influencée par un bruit, assurant une cohérence visuelle entre le tronc et la tête.

Yeux : Les yeux sont composés de deux sphères superposées : une grande sphère pour le globe oculaire et une plus petite, noire, pour la pupille. Cette construction permet une représentation stylisée mais expressive :

```
PShape creerYeuxMomie() {
  PShape eyes = createShape(GROUP);

  for (int i = -1; i <= 1; i += 2) {
    float xPosition = i * 16;

    PShape eye = createShape(SPHERE, 10);
    eye.scale(1.7, 1., 1.);
    eye.translate(xPosition, -280, 35);
    eye.setStroke(false);

    PShape eyeball = createShape(SPHERE, 3);
    eyeball.scale(1., 1.7, 1.);
    eyeball.translate(xPosition + i * 7, -280, 42);

    eyeball.setFill(color(0));

    eyes.addChild(eye);
    eyes.addChild(eyeball);
  }

  return eyes;
}
```

Les deux yeux sont positionnés de manière symétrique à l'aide d'une boucle avec i = -1 et i = 1, centrée autour de l'axe vertical de la tête.

Bras : Les bras sont également générés avec un QUAD_STRIP. Chaque bras présente des variations de rayon pour simuler différentes sections : épaules plus larges, coudes plus fins et avant-bras renforcés. Ce détail anatomique est obtenu en appliquant des multiplicateurs sur le rayon en fonction de la position i .

université
PARIS-SACLAY

Mains : Les mains ne sont pas créées par génération procédurale, mais importées à partir de fichiers **.obj** externes, ce qui permet un niveau de détail plus élevé sans complexité excessive dans le code.

Une fois importées, les mains sont orientées et redimensionnées pour correspondre à la position et à la rotation des bras. La coloration est harmonisée avec celle du corps pour une cohérence visuelle.

Déplacement de la momie :

Positionnement sur la Grille

Les momies sont positionnées sur la grille du labyrinthe à l'aide de coordonnées (x, y), où chaque paire représente une cellule spécifique. Il y a deux momies par niveau. La première est initialement positionnée à la même position que le joueur. Les secondes coordonnées sont initialisées aléatoirement. Elles sont mises à jour dynamiquement lors du déplacement. La momie occupe une cellule à la fois et se déplace d'une cellule adjacente (haut, bas, gauche, droite) en suivant un chemin calculé.

Algorithme A* pour le Déplacement

L'algorithme A* est utilisé pour calculer le chemin optimal entre la position actuelle de la momie et une destination aléatoire dans le labyrinthe. Cet algorithme combine :

- Une heuristique (distance de Manhattan) pour estimer le coût restant jusqu'à la destination.
- Un coût réel (nombre de cases parcourues depuis le départ) pour évaluer les chemins explorés.

Fonctionnement

- Initialisation : La momie sélectionne une destination valide dans le labyrinthe.
- Calcul du chemin : L'algorithme explore les cellules voisines, évalue leurs coûts (F = G + H, où G est le coût réel et H l'heuristique), et retient les chemins les plus prometteurs.

- Suivi du chemin : Une fois le chemin calculé, la momie se déplace case par case en alternant rotations (pour changer de direction) et translations (pour avancer). Les animations de rotation et de déplacement sont gérées de manière fluide pour un rendu réaliste.

Cet algorithme est choisi pour sa large documentation et son efficacité dans les grilles, garantissant un chemin optimal tout en évitant les murs. Il permet à la momie de naviguer de manière autonome et intelligente.

Conclusion :

Le projet *Des momies et des pyramides* a constitué une mise en application concrète des notions d'informatique graphique abordées au cours du semestre. Il a permis la conception d'un environnement 3D interactif combinant modélisation procédurale, gestion de textures et lumières, ainsi que diverses mécaniques d'animation et d'interaction.

La pyramide a été générée par empilement de labyrinthes de tailles décroissantes, avec une gestion spatiale rigoureuse pour l'alignement des niveaux et la création d'une coque extérieure réaliste. Un environnement désertique a été modélisé à l'aide du bruit de Perlin, apportant un relief crédible au sol. Une atmosphère différenciée a été créée entre l'intérieur sombre de la pyramide et l'extérieur plus lumineux, en fonction de la position de la caméra.

La momie a été modélisée par l'utilisation de primitives QUAD_STRIP avec variation de rayon, bruit procédural et composition hiérarchique (PShape(GROUP)), permettant un rendu détaillé et une animation plausible. Les différentes parties du corps ont été coordonnées pour maintenir la cohérence visuelle. La momie est également dotée d'un comportement de déplacement autonome dans le labyrinthe.

Divers aspects d'interaction et d'animation ont été intégrés : transitions entre étages, affichage dynamique de messages, minimap, coloration contextuelle des murs. Ces éléments renforcent l'immersion de la scène et enrichissent l'expérience utilisateur.

Compétences acquises

Le projet a permis l'acquisition ou le renforcement des compétences suivantes :

- **Modélisation 3D procédurale** : génération de formes complexes (pyramide, momie) à l'aide d'algorithmes et de fonctions mathématiques (bruit de Perlin, modulation angulaire).
- **Programmation orientée objet** : structuration du projet autour de classes fonctionnelles, facilitant la modularité et la réutilisabilité du code.
- **GLSL et shaders** : conception et intégration de shaders personnalisés pour simuler le grain du sable, l'éclairage directionnel ou les effets de profondeur.
- **Texturation et éclairage** : application de textures adaptées à chaque élément (pierre, sable, bandelettes), gestion de lumières dynamiques selon la position du joueur.
- **Animation et interactivité** : mise en place de mouvements fluides, d'animations de transition, de déplacements contextuels, et d'une interface utilisateur visuelle (minimap).
- **Utilisation de Git** : gestion de version du projet, coordination du travail en binôme, suivi rigoureux des modifications et résolution de conflits.

- **Méthodologie de travail en équipe** : répartition fonctionnelle des tâches (pyramide / momie), intégration des contributions individuelles dans un projet cohérent.

En synthèse, le projet a permis d'allier créativité graphique, rigueur algorithmique et gestion de projet, tout en mobilisant des outils modernes de développement logiciel. Il s'inscrit comme une étape significative dans l'apprentissage des techniques avancées de modélisation 3D et de visualisation interactive.