



POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

INF2705 Infographie

Travail pratique 2

Transformation matricielle et textures

Département de génie informatique et génie logiciel
Polytechnique Montréal
Tristan Rioux, Automne 2023

Table des matières

1	Introduction	2
1.1	But	2
1.2	Portée	2
1.3	Remise	2
2	Description globale	3
2.1	But	3
2.2	Travail demandé	3
3	Exigences	11
3.1	Exigences fonctionnelles	11
3.2	Exigences non fonctionnelles	12
A	Liste des commandes	13

1 Introduction

Ce document décrit les exigences du TP2 « *Transformation matricielle et textures* » (Automne 2023) du cours INF2705 Infographie.

1.1 But

Le but des travaux pratiques est de permettre à l'étudiant de directement appliquer les notions vues en classe.

1.2 Portée

Chaque travail pratique permet à l'étudiant d'aborder un sujet spécifique.

1.3 Remise

Faites la commande « `make remise` » ou exécutez/cliquez sur « `remise.bat` » afin de créer l'archive « **INF2705_remise_TPn.zip** » (ou .7z, .rar, .tar) que vous déposerez ensuite dans Moodle. (Moodle ajoute automatiquement vos matricules ou le numéro de votre groupe au nom du fichier remis.)

Ce fichier zip contient tout le code source du TP (`makefile`, `*.h`, `*.cpp`, `*.glsl`, `*.txt`).

2 Description globale

2.1 But

Le but de TP est de permettre à l'étudiant de mettre en pratique les concepts de transformation et d'utiliser des textures. Il sera en mesure d'effectuer diverses transformations afin d'afficher plusieurs objets à l'écran de façon simultanée dans différentes vues et perspectives. Il sera aussi capable d'utiliser des textures afin d'ajouter des détails aux objets. Le travail fera l'utilisation des fonctions d'OpenGL `glUniform3f` et `glUniformMatrix4fv` et les fonctions de `glm` `glm::translate`, `glm::rotate`, `glm::scale`, `glm::lookAt`, `glm::ortho` et `glm::perspective`.

2.2 Travail demandé

Le tp2 est une continuation du tp1, vous pouvez donc reprendre les mêmes fichiers de projet. Cependant, il va falloir retirer quelques bouts de code et rajouter certains fichiers au projet. Vous pouvez retirer les fonctions `changeRGB` et `changePos`, les formes de la partie 1 (on garde le cube qui tourne), le chargement des shaders basic et color (vous pouvez garder les `.glsl`), la selection de la forme avec la touche T. Vous devez aussi retirer toutes les variables qui ne sont plus utilisées (dans `vertices_data.h` notamment). La main loop devrait que contenir l'utilisation du shader de transformation, les assignations des matrices et le dessin du cube.

Dans les fichiers à ajouter, il y a une mise à jour de la classe Window, une classe Model avec une fonction pour charger les `.obj`, une fonction pour le chargement des fichiers d'images, ainsi qu'un fichier de fonctions utilitaires. Ceux-ci utilise les librairies "OBJLoader" et "stb_image" qui sont du format "Header only". Il y a aussi des squelettes de classe de Camera, Model, Texture2D et TextureCubeMap que vous pouvez utiliser. Le fichier `utils.h` contient quelques indications supplémentaires sur la structure de votre main.

Pour être compatible avec les modèles `.obj` qui peuvent avoir beaucoup d'indexes, il faudra faire une modification à la classe `BasicShapeElements`. On changera les types `GLubyte` par des `GLuint`. N'oublier pas la méthode `draw` et de modifier le type des tableaux (celui du cube notamment). Il faudra aussi faire un constructeur par défaut qui ne fait que créer les `vao`, `vbo` et `ebo`. On ajoutera une méthode `setData` pour passer les données comme l'ancien constructeur du tp1.

Partie 1 : création d'une scène graphique

Maintenant que vous êtes à l'aise avec les bases d'OpenGL, nous allons commencer à dessiner une scène 3D. Une fonction vous est donnée pour faire le chargement des modèles 3D.

Avant de peupler la scène, il sera important de créer une caméra pour pouvoir facilement voir les objets. Une caméra en première personne et en troisième personne devront être implémentées. Utiliser le cube qui tourne du tp1 pour vous aider à débugger. Les touches W-A-S-D du clavier seront pour

le déplacement, alors que le mouvement de la souris sera utilisé pour modifier l'orientation en x et en y. Penser à orienter le déplacement dans le référentiel du joueur (donc par rapport à son orientation en Y). Pour le change de position et d'orientation, il va falloir appliquer des multiplicateurs pour réduire la vitesse un peu. Les valeurs sont libres à votre choix et confort (de mon côté, j'ai 0.05 et 0.01 respectivement). Pour la caméra en première personne, vous devez faire les translations et rotation directement, alors que pour la caméra en troisième personne, vous devez utiliser `glm::lookAt`. Lorsqu'on change la caméra en troisième personne, vous devez dessiner le personnage (suzanne le singe). Le modèle doit être descendu à la hauteur du sol ($Y=-1$), avoir une rotation de π en Y et avoir une taille réduire de facteur 2 en plus des rotations et translation pour le déplacer selon l'orientation et position. On utilisera le scroll de la souris pour alténer entre les deux modes de vue.

Pour la caméra en première personne, il vous faudra faire des rotations selon l'orientation du personnage et une translation selon la position du personnage. Pour la caméra en troisième personne, l'utilisation de `glm::lookAt` et des coordonnées sphériques sera utile. On utilisera un rayon de 6 pour la position de la caméra. Puisqu'on voudra initialement avoir la caméra derrière le personnage, un décalage de $\pi/2$ sera appliqué sur l'orientation. On décalera la position de la caméra et du point observé par la position du personnage. La position et orientation du joueur sont passées par référence à la création de la caméra.

Lorsque la classe `Camera` sera complétée, vous pouvez retirer le cube et faire le chargement des modèles 3d en complétant la classe `Model`. Vous pouvez charger les positions et indexes des modèles avec la méthode `loadObj`. Vous allez avoir besoin d'utiliser une instance de `BasicShapeElements` pour contenir les données et les dessiner.

Un nouveau shader devra être fait pour dessiner les modèles. Pour le moment, le seul attribut en entré du vertex shader sera la position. Celui-ci aura un uniform pour la matrice mvp. Pour le fragment shader, on aura un second uniform de type `vec3` pour donner la couleur de sortie. Vous pouvez faire les modifications dans le shader de transformation du tp1 (que vous devriez probablement renommé comme bon vous semble, j'ai choisis de l'appeler `model` dans mon cas).

Dans notre scène 3d, nous allons répartir des groupes d'objets sur un terrain. Un groupe sera composé d'un arbre, d'un roché et d'un champignon. Les groupes ont une position et une rotation en Y aléatoire. Pour le moment, les modèles seront seulement d'une couleur quelconque de votre choix par un uniform pour pouvoir les différencier (dans la partie 2, ils pourront être texturés).

Vous allez devoir :

- Créer un plane carré dans le plan X-Z centré en $(0, -1, 0)$ de taille 60x60 pour faire le sol.
Utiliser la classe `BasicShapeElements` comme au tp1 pour vous faire un modèle.
- Créer un plane rectangulaire sur un des côtés du sol de taille 60x20 pour faire un ruisseau.
Utiliser la classe `BasicShapeElements` comme au tp1 pour vous faire un modèle.
- Répartir 49 groupes d'objets dans la scène (tableau de 7x7). Dans le code, les groupes ne sont représentés que par leur matrice. Vous devrez initialiser ces matrices dans l'initialisation du programme.
- Les groupes possèdent une transformation de translation qui est donnée par `getGroupRandomPos`, de rotation en Y aléatoire entre $[0; 2\pi]$ et mise à l'échelle aléatoire entre $[0.7; 1.3]$. Les groupes doivent être sur le sol.
- Un groupe contient un arbre centré en $(0, 0, 0)$ de la position du groupe. Il doit aussi avoir une

mise à l'échelle d'une valeur aléatoire entre [0.7; 1.3].

- Un groupe contient un roché placé à un angle aléatoire entre $[0; 2\pi]$ sur un cercle de rayon entre $[1; 2]$ centré à la position du groupe. Vous allez probablement vouloir le sortir un peu plus du sol avec une translation de 0.2 en Y.
- Un groupe contient un champignon toujours placé à la base de l'arbre à $(0.3, 0, 0.3)$. Cependant, la translation doit suivre la mise à l'échelle de l'arbre.
- Vous ne devriez pas avoir de trigonométrie dans le code des transformations des objets et des groupes, utilisez les matrices de rotation efficacement.
- Toutes les multiplications de matrices devront être fait sur le cpu. Je vous recommande de calculer les matrices qui risque d'être réutiliser, par exemple la multiplication `matriceProjection * matriceVue` qui peut être fait une seule fois et être réutilisé pour les prochains calculs. C'est un peu l'équivalent de push/pop matrix vu en cours.
- Vous avez accès à la fonction `rand01` pour avoir une valeur aléatoire entre $[0; 1]$.

Pour la matrice de projection, nous allons continuer avec la matrice de projection du tp1 avec un fov de 70, le aspect ratio de la fenêtre, un near plane à 0.1, mais un far plane à 200 cette fois-ci. Il sera aussi possible de modifier cette matrice pour une matrice orthogonal afin de dessiner directement dans l'écran. Pour le moment, on dessinera tous simplement un carré de couleur en coin de l'écran. Il est libre à vous de définir la matrice orthogonal, tant que le carré garde les proportions au changement de la taille de la fenêtre (commenter `SDL_SetRelativeMouseMode(SDL_TRUE)` dans `window.cpp` pour avoir la souris visible). Dans mon cas, il a une taille de 100 et un décalage de position du $3/4$ de sa taille en x et y dans le référentiel de l'écran. D'autres valeurs seront acceptées (puisque cela dépendent de vos définitions), tant que le résultat est semblable. Puisque le carré est dans l'écran, il faudra le dessiner en dernier en faisant toujours passé le test de profondeur pour ne pas qu'il soit impacté par les éléments 3d.

De plus, les modèles 3d ont tendances à cacher certaines de leur faces. Ce phénomène est tout à fait normal, dû à la projection d'une forme 3d dans le plan de l'écran. Cependant, les fragments de ces primitives cachées seront encore calculer, ce qui est dégradant pour les performances. On peut facilement y remédier en activant le face culling sur les faces arrières.

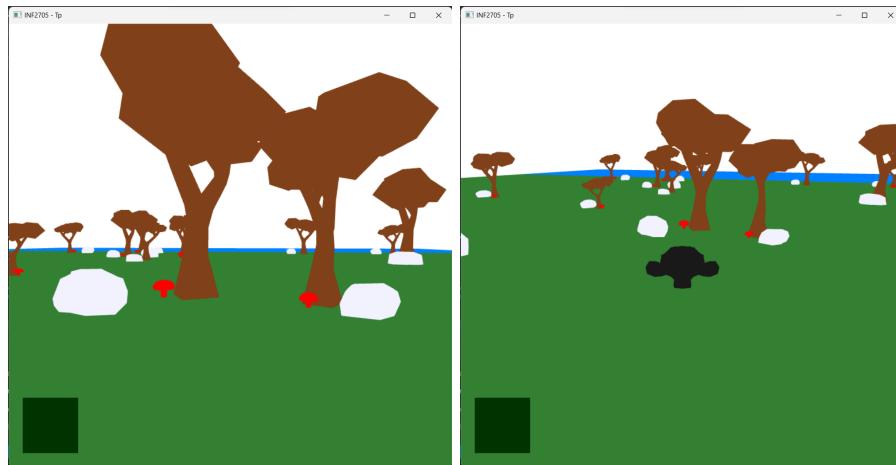


FIGURE 1 – La scène vue en fps et en tps.

Partie 2 : utilisation de texture

Malgré les nouveaux ajouts que nous avons fait à la scène, elle manque clairement de détail. Une technique couramment utilisée pour facilement rajouter du détail est d'utiliser des textures.

Pour se faire, il va falloir charger les textures de chaque modèles et les appliquer sur ceux-ci lorsqu'il sera temps de les dessiner avec la méthode `use`. Une classe `Texture2D` contenant du code pour le chargement des images vous est fournis. Vous devez complété son constructeur pour charger l'image en tant que texture, ainsi que les autres méthodes pour utiliser la texture.

Vous allez devoir modifier votre shader de la partie 1 pour accepter l'attribut des coordonnées de texture dans le vertex shader. Celui-ci devra être passé au fragment shader qui l'utilisera avec la fonction `texture` pour donner la couleur du fragment. Il ne sera nécessaire que d'utiliser la couleur de la texture. Le alpha de la couleur de sortie sera à 1, puisque le blending n'est pas activé. N'oublier pas d'ajouter un `uniform sampler2d` dans le fragment shader. Vous pouvez retirer l'uniform de couleur, puisqu'il ne sera plus utilisé.

Vous allez devoir charger les textures suivantes et les appliquer sur les objets respectifs : "suzanne-Texture.png", "treeTexture.png", "rockTexture.png", "mushroomTexture.png", "groundSeamless.jpg", "waterSeamless.jpg", "heart.png". On ne veut pas que les textures se répètent sur ces objets. Remarquer que les textures des modèles 3d ont de l'illumination de précalculer dedans, ce qui ajoute beaucoup de détails. Pour les modèles 3D, l'attribut de coordonnée de texture est déjà défini dans le .obj. Il faudra charger les coordonnées de texture (en décommentant les lignes de la méthode de chargement) et activer l'attribut dans le `BasicShapeElements`.

Il faudra faire de même pour les autres géométries que vous avez défini en ajoutant les coordonnées de texture et activer l'attribut. Le terrain et la rivière possèdent des textures dites "seamless". Elles sont idéales pour couvrir une très grande région tout en ayant des textures relativement petites en espace mémoire. Pour ce faire, il faut nécessairement charger cette texture dans le mode répétition pour pouvoir couvrir la totalité de la surface. De plus, il faudra activer le mipmap pour ces textures spécifiquement afin de retirer les artéfacts de rendu.

Pour le terrain, la texture couvrira le dixième des arêtes du plane (elle sera répété 10 fois sur le long d'une arête).

Pour la rivière, on créera un nouveau shader program pour faire le déplacement des coordonnées de texture en fonction du temps qui devrait être passé en uniform dans celui-ci. Le temps est obtenable à l'aide de la méthode `getTick` de la classe `Window`, qui retourne le temps en milliseconde depuis le début de l'exécution. Comme le terrain, la texture est répétée et prend la moitié de la largeur et couvre seulement le cinquième de la surface sur la longueur.

Pour simuler l'écoulement de la rivière, on ajoutera un déplacement des coordonnées le long de la longueur. La vitesse devrait être d'un facteur 1/10 du temps. On peut aussi rajouter un effet de vague en ajoutant un autre décalage dans le fragment shader. On utilisera l'équation 1 et 2 pour faire la fonction d'oscillation.

$$x = \cos(\text{time} * \text{timeScale}.x + (\text{texCoords}.x + \text{texCoords}.y) * \text{offsetScale}.x) \quad (1)$$

$$y = \sin(\text{time} * \text{timeScale}.y + (\text{texCoords}.x + \text{texCoords}.y) * \text{offsetScale}.y) \quad (2)$$

L'élément le plus important dans cette formule est le décalage du sinus/cosinus qui dépend de la coordonnée de texture sur le plan. On utilisera un timeScale de $x = 1$ et $y = 2$, un offsetScale de $x = y = 2$. Au final, on pourra multiplier le tout par l'amplitude de $x = 0.025$ et $y = 0.01$.

On peut aussi mettre une texture sur le fond pour remplacer la couleur uni qu'on a défini avec glClearColor. On utilisera ce qu'on appelle un skybox. Un skybox est un cube qui englobe la totalité de la scène du point de vue de l'écran. On utilise une texture qui contient la projection d'un décor lointain et de nuages pour donner l'impression d'être dans un très grand monde.

Il faudra compléter la classe TextureCubeMap. Le mode de texture est plutôt GL_TEXTURE_CUBE_MAP au lieu de GL_TEXTURE_2D. Puisque les coordonnées de texture sont utilisées pour déterminer quelle texture sera utilisée, elles seront données en vec3. Cela implique qu'il faudra changer le paramètre de wrap pour la composante r du vecteur de coordonnée de texture.

Pour charger les 6 textures, la cible sera GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, où i est l'index de la texture. Le tableau des chemins des textures vous a été donné, ainsi que les coordonnées de position.

Il faudra faire un nouveau shader programme. Pour le vertex shader, on a en entrée l'attribut vec3 de position, une sortie vec3 de coordonnée de texture et la matrice mvp en uniform. La sortie de coordonnée de texture est simplement la position. Pour le fragment shader, l'entrée est l'attribut de coordonnée de texture qui sera utilisé pour prendre la texture de type samplerCube.

Puisqu'on voudra seulement remplir le derrière de l'image (les endroits où rien n'a été dessiné), on dessinera le skybox à la toute fin (même après le carré dans l'écran) en prenant soin de changer la fonction du test de profondeur pour qu'il passe au endroit qui n'a pas été dessiné (la valeur de ces endroits sera de 1 dans le depth buffer) (n'oubliez pas de restaurer l'ancienne fonction après avoir dessiné le ciel). Il sera nécessaire de changer la valeur du Z dans la position finale du vertex shader pour permettre ce fonctionnement. De plus, on ne veut pas que le skybox bouge avec la caméra, il faudra donc retirer la translation de la matrice de vue (en la convertissant en matrice 3x3, puis 4x4 de nouveau par exemple).

Essayer de dessiner les objets en changeant le moins souvent l'état d'OpenGL (les shaders et textures notamment).

Finalement, il faut mettre une texture sur le quad du hud. On lui mettra la texture du cœur pour imiter le hud d'une barre de vie. Puisque la texture aura des texels transparents, il faudra ajouter un discard du fragment dans le fragment shader pour éliminer les fragments dont le alpha est en dessous de 0.3.



FIGURE 2 – La scène après avoir été texturée.

Partie 3 : LOD avec billboard et gazon

Maintenant qu'on a plusieurs éléments graphiques à l'écran, il serait pratique de ne pas utiliser toutes les ressources de l'ordinateur pour les éléments lointains, qui ne couvre déjà pas beaucoup d'espace à l'écran. Le niveau de détail n'a pas besoin d'être aussi élevé que pour les éléments près de l'observateur. Pour ce faire, une technique très commune est d'utiliser des LOD (level of details). Souvent, il est question de réduire la résolution de l'image, réduire le nombre de vertices dessinées (ou de ne pas dessiner l'objet dans certains cas), changer pour un shader moins coûteux ou encore en utilisant des billboards. Les billboards sont souvent utilisés pour réduire le détail des arbres. C'est ce que nous allons faire ici.

Lorsqu'un arbre aura une distance absolu supérieur à 25 unités par rapport à la caméra, on ne dessinera pas le modèle 3d, mais plutôt un plan texturé d'une projection du modèle 3d. On pourra utiliser la texture "treeBillboard.png". Le billboard doit avoir sensiblement la même taille que le modèle original pour rendre l'effet convainquant. L'arbre possède une taille de 3.47 en y. Puisqu'on doit respecté l'aspect ratio de l'image, le facteur sera de $0.96 * 3.47$ en x. Comme pour le modèle original de l'arbre, l'origine du plan doit toucher le sol pour être bien positionné.

Le billboard sera toujours orienté vers la caméra. On doit appliquer les matrices de transformation de l'arbre, du group et de la vue, puis retirer la rotation de cette matrice en la remplaçant par la matrice identité. Cependant, cette action retire la matrice de mise à l'échelle. Il est possible de la retrouver en calculant la norme du vec3 de chaque colonne. Par exemple, le facteur de mise à l'échelle en x est la norme des trois premiers nombres de la première colonne. Vous pouvez utiliser `glm::length` pour vous aider. N'oublier pas que les matrices glm sont en espace colonne, ce qui signifie que le premier index sélectionne la colonne, puis le second la ligne. Par exemple, pour avoir l'élément de

la première ligne, à la dernière colonne (la translation en x), on fera $matrix[3][0]$. Ainsi, on pourra restaurer cette matrice pour conserver les proportions. De plus, puisqu'on veut que le billboard soit orienté seulement en y, il faut seulement modifier la première et troisième colonne de la matrice de rotation (la deuxième reste intact). On finira comme à l'habitude avec la multiplication par la matrice de projection.

Parfois, certains engins de rendu utilisent plusieurs textures sous différents angles pour dessiner les billboards, ce qui évite de voir toujours le même côté. Nous allons avoir ce problème, spécifiquement visible lorsqu'on change entre le lod et le modèle, cependant le résultat sera acceptable pour le cadre du tp.

Pour finaliser ce tp, on ajoutera du gazon "3d" au sol. On n'utilisera pas de billboards qui s'orientent vers la caméra, mais plutôt ce que j'aime bien appeler des "x shapes". Ces modèles statiques en forme de croix couvrent plusieurs angles sans avoir à être mettre à jour et utilise un petit nombre de vertices. On utilisera la texture "grassAtlas.png", qui contient trois textures différentes qu'on pourra sélectionner en modifiant les coordonnées de texture. Il ne faut pas que la texture soit en mode répétition.

Les positions et coordonnées de texture d'un modèle de gazon vous sont données. Vous allez devoir les utilisés pour générer plusieurs fois ce modèle à différentes positions et avec une différente texture (en déplaçant les coordonnées de texture). Il devra y avoir 500 fois ce modèle de gazon (ou plus si vous voulez essayer). Il va donc y avoir plusieurs fois le modèle en mémoire, mais à des positions différentes. Vous devez appliquer la différence de position et de coordonnée de texture sur chaque élément avec les fonctions aléatoires fournis. Ils seront statiques dans la mémoire dans un BasicShapeArrays. Au lieu d'utiliser une matrice de transformation à chaque fois pour les dessiner, on dessine la totalité avec la matrice identité comme matrice de modèle.

Cette technique est assez courante pour "fixer" les objets qu'on sait qu'ils ne bougeront pas. Cependant, cela augmente la quantité de mémoire utilisée.

Il sera aussi intéressant de créer un autre shader program avec un vertex shader différents de celui des modèles 3d pour pouvoir permettre l'animation du gazon. Ce shader program peut réutiliser le même fragment shader que celui des modèles 3d. Le vertex shader prends en entré la position et les coordonnées de texture. En sortie, il y a les coordonnées de texture non modifiés. On ajoutera la matrice mvp et le temps en uniform. Pour faire l'animation du gazon, on utilisera la formule suivante pour ajouter un décalage de position en x et en z :

$$inTexCoords.y * \sin(time/2.0 + pos.x + pos.z) * 0.05 \quad (3)$$

On utilise la coordonnée de texture en y pour déterminer si la vertice traité est celle du haut, puisqu'on veut seulement déplacer le haut du gazon. Le sinus du temps décaler par rapport à la position permet d'avoir un peu plus de variance entre nos instances.

On pourrait trouver une meilleure formule, mais celle-ci donne un bon résultat pour le but du tp. Il existe d'autres façons d'avoir de meilleurs animations et résultats (avec l'instanciation ou un geometry shader), mais elles sortent un peu cadre du tp.

Il faudra désactiver le culling des faces arrières le temps de dessiner le gazon.



FIGURE 3 – La scène finale avec lod et gazon.

3 Exigences

3.1 Exigences fonctionnelles

Partie 1 :

- E1. La classe Camera est implémenté correctement. Il est possible de voir en première et troisième personne et de ce déplacer. [3 pts]
- E2. La classe Model est implémenté correctement. Il est possible de dessiner les modèles 3d. [1 pt]
- E3. Les modèles géométriques (ruisseau et sol) sont bien défini. [1 pt]
- E4. Les transformations sont faites correctement sans trigonométrie et ont une bonne réutilisation. [5 pts]
- E5. Le modèle du personnage est seulement dessiné en troisième personne. [1 pt]
- E6. La projection orthogonal et le quad sont bien dessiné dans l'écran. Il ne traverse pas les objets 3d. [1 pt]
- E7. Le face culling est activé. [1 pt]

Partie 2 :

- E8. La classe Texture2D est implémenté correctement. [1 pt]
- E9. La classe TextureCubeMap est implémenté correctement. [1 pt]
- E10. Les textures sont chargées dans le bon mode. [1 pt]
- E11. Les attributs de coordonnée de texture sont défini correctement. Ils sont correctement utilisés dans le shader pour voir les textures sur les objets. [2 pts]
- E12. Les coordonnées de texture du terrain et de la rivière sont bien défini. Il active le mipmap et fait la bonne interpolation de minification. [2 pts]
- E13. Le ruisseau est animé à l'aide du shader. On voit le courant et les vagues. [2 pts]
- E14. Le skybox est visible et ne bouge qu'avec la rotation de la caméra. Il est dessiné en dernier avec le bon depth test. [4 pts]
- E15. Il y a minimisation des changements d'états d'OpenGL. [1 pt]
- E16. On discard les fragments avec un alpha trop bas. [1 pt]

Partie 3 :

- E17. On utilise un billboard avec la bonne texture et les bonnes proportions pour dessiner les arbres plus loin. [1 pt]
- E18. Les billboards des arbres s'orientent vers la caméra correctement (pivot sur le Y). [3 pts]
- E19. Le gazon est bien affiché des deux côtés de la face. Il utilise le texture atlas pour avoir une texture différente et plusieurs instances sont visible à des positions aléatoires. Les données sont statiques dans la mémoire (sans transformation matriciel pour le rendu). [4 pts]
- E20. Un vertex shader permet l'animation du gazon. [2 pts]

3.2 Exigences non fonctionnelles

De façon générale, le code que vous ajouterez sera de bonne qualité. Évitez les énoncés superflus (qui montrent que vous ne comprenez pas bien ce que vous faites !), les commentaires erronés ou simplement absents, les mauvaises indentations, etc. [2 pts]

ANNEXES

A Liste des commandes

Touche	Description
ESC	Quitter l'application
w	Mouvement avant
s	Mouvement arrière
a	Mouvement gauche
d	Mouvement droit
scroll	Alterner entre fps/tps
souris	Changer l'orientation de la camera