Rendu

Deadline
Mercredi 23/02/2022 – 18h00

sur Git Tag gold branch master !
git@gitlabstudents.isartintra.com:projets/2021_gp_2026rasterizer_gp_2026rasterizer-{nom_du_groupe}.git

Objectif

Le but de ce projet est de vous faire comprendre comment marche en arrière plan opengl 2.0.
C'est un projet assez dense et qui demande de la curiosité de votre part, bon courage!

Sujet

Etape 0 : *Les maths*

Mettez à jour votre librairie de maths :

- Créer les classes/structures de données suivantes :

- Vec3 : un point dans l'espace 3D, la classe doit avoir les membres x,y,z public et les méthodes suivantes :
  - float GetMagnitude() const *(retourne la norme du vecteur)*
  - void Normalize()

  et également surcharger les opérateurs suivants :
  - + (Vec3, Vec3) -> Vec3
  - * (Vec3, float) -> Vec3 *(multiplie toutes les composantes du vecteurs par un facteur)*

- Vec4 : représente un point dans l'espace 3D en coordonnées homogènes, la classe doit avoir les membres x,y,z et w public et les méthodes suivantes :
  - void Homogenize() : divise toutes les composantes par w *(si w != 0)*
  - float GetMagnitude() const : retourne la norme du vecteur homogénéisé *(en ignorant w)*
  - void Normalize()
  - constructeur Vec4(const Vec3& vec3, float _w = 1.0f)

  et également surcharger les opérateurs suivants :
  - + (Vec4, Vec4) -> Vec4
  - * (Vec4, float) -> Vec4 *(multiplie toutes les composantes du vecteurs par un facteur)*

  Mat4 : représente une matrice 4 x 4, vous pouvez chosir un double tableau public pour représenter les éléments de la matrice ou 16 float.
  Elle doit surcharger les opérateurs suivants :
  - * (Mat4, Mat4) -> Mat4
  - * (Mat4, Vec4) -> Vec4

  Elle doit également contenir la fonction statique suivante :
  - InverceMatrix()->Mat4
  - TransposeMatrix() -> Mat4
  - static Mat4 CreateTransformMatrix(const Vec3& rotation, const Vec3& position, const Vec3& scale)
    (les membres du vecteur rotation représentant les angles d'Euler en degrés)
    Si vous éprouvez des difficultés pour coder cette fonction, vous pouvez coder ces 5 fonctions intermédiaires :
    - static Mat4 CreateTranslationMatrix(const Vec3& translation)
    - static Mat4 CreateScaleMatrix(const Vec3& scale)
    - static Mat4 CreateXRotationMatrix(float angle) (*Matrice de rotation autour de l'axe des X)*
    - static Mat4 CreateYRotationMatrix(float angle) (Matrice de rotation autour de l'axe des Y)
    - static Mat4 CreateZRotationMatrix(float angle) *(Matrice de rotation autour de l'axe des Z)*

Et donc la matrice de transformation peut se calculer comme ca :
transform = matTranslation * matRotY * matRotX * matRotZ * matScale

Version 1 : *Couleur*

- Rendre à l'écran un seul triangle situé en (0,0,2) dont les vertices ont pour coordonnées respectivement (-0.5, -0.5, 0), (0.5, -0.5, 0), (0, 0.5, 0) et pour couleurs respectivement rouge, vert, bleu. La couleur des pixels du triangle doit être interpolée de façon bilinéaire par rapport à la couleur des sommets *(résultat attendu : http://openglsamples.sourceforge.net/image/triangle.png, https://en.wikipedia.org/wiki/Bilinear_interpolation)*

Version 2 : *Z-Buffer*

- Rendre une scène composée d'un cube rouge de côté 1 situé en (-0.5, 0, 2) et d'une sphère bleue de rayon 1 située en (0.5, 0, 2)

- Utiliser l'algorithme du zbuffer de façon à ce que les pixels cachés ne soient pas affichés *(http://www.wikiwand.com/de/Z-Buffer)*

Version 3 : *Eclairage avec modèle de Phong* *(10/20)*

- Modifier les fonctions CreateSphere et CreateCube de façon à calculer les normales des vertices en plus des positions.

- Créer la class Light qui représente une lumière ponctuelle dans l'espace, elle contient les membres suivants :
  - Vec3 position
  - float ambientComponent
  - float diffuseComponent
  - float specularComponent

- Il faut donc rajouter le membre suivant à la classe Scene : std::vector<Light> représentant l'ensemble des lumieres de la scène

- Rendre la même scène que précédemment avec en plus une lumière située en (0, 0, 0) et de composante ambiante 0.2, diffuse 0.4, et spéculaire 0.4

- Il faut utiliser le modèle d'éclairage de Phong *(https://fr.wikipedia.org/wiki/Ombrage_de_Phong)* pour calculer la couleur des vertices éclairés *(couleur qui sera donc automatiquement interpolée pour les pixels du triangle)*, la brillance du matériaux *(le coefficient alpha)* sera à votre convenance afin d'obtenir un résultat qui satisfait vos goût raffinés de graphiste

Version 4 : *Per-pixel lighting & Blinn-Phong*

- Remplacer le modèle d'éclairage de Phong par celui de Blinn-Phong *(http://sunandblackcat.com/tipFullView.php?l=eng&topicid=30&topic=Phong-Lighting (archive), une approximation optimisée du modèle de Phong)*

- Jusqu'ici l'éclairage était calculé en chaque vertex du triangle, et la couleur des pixels du triangle était interpolée parmi ces 3 valeurs *(per-vertex lighting)*. Désormais il faut que l'éclairage soit calculé pour chaque pixel, pour cela vous devrez non pas interpoler la couleur, mais la position et la normale des vertex *(attention à bien formaliser la normale interpolée !)*, puis recalculer l'éclairage pour chaque pixel

- A chaque frame, modifier la rotation du cube et de la sphère pour les faire tourner sur eux même à vitesse constante

Version 5 : *Projection* *(12 /20)*

- Jusqu'ici, seule une projection orthographique en dur a été utilisée. Les points sont donc transformés de la façon suivante : Vec4 projectedVertex = projectionMatrix * entity.transformation * vertex.position
  Ce qui donne un point 4D qu'il faut homogénéiser.
  Une fois homogénéisé,
  - projectedVertex.x représente la position x "viewport" du pixel à l'écran (cad -1=gauche de l'écran, 0=centre, 1=droite de l'écran)

- projectedVertex.y représente la position y “viewport” du pixel à l’écran (cad -1=bas de l’écran, 0=centre, 1=haut de l’écran
- projectedVertex.z représente la profondeur du point, et peut donc être utilisée par le z buffer, si z < -1 le point est derrière la caméra et n’est donc pas affiché. De même si z > 1 le point est trop loin et n’est pas affiche
- projectedVertex.w vaut 1 et ne sert plus à rien

- Il faut ensuite convertir les coordonnées x et y de projectedVertex en coordonnées de pixel dans la texture de rendu (framebuffer)

- Par exemple, pour obtenir le même résultat que précédemment, en prenant near=0, et far=2, il faut utiliser la matrice de projection suivante :
  (2/5 0 0 0)
  (0 2/5 0 0)
  (0 0 1 -1)
  (0 0 0 1)

Version 6 : *Texturing*

- Créer une classe Mesh : contient les informations membres suivantes :
  - std::vector<rdrVertex> vertices : vertex buffer,
    vertices du mesh qui ne doivent jamais changer au cours de la simulation *(pour déplacer un objet on modifie sa matrice de transformation, pas les vertices du mesh)*
  - std::vector<int> indices : index buffer, suites de triplets d’index dans le vertex buffer qui déterminent les triangles du mesh
    (ex : 0,1,2 désigne le triangle vertices[0],vertices[1],vertices[2])

  Le mesh doit aussi contenir les fonctions suivantes
  - static Mesh* CreateCube() : créer un cube de côté 1
    (donc composé de 6 faces donc 12 triangles)
  - static Mesh* CreateSphere(int latitudeCount, int longitudeCount)
    (créer une sphère de rayon 1)

    - Texture : représente une image et a donc les membres suivants **privés**
        - unsigned int width
        - unsigned int height
        - Color* pixels

- Ajouter un membre Texture* pTexture à la classe Mesh *(nullptr indique que le mesh n'est pas texturé)*
- Vous pouvez utiliser STB_Image pour charger vos textures https://github.com/nothings/stb/blob/master/stb_image.h
- Rendre une scène avec juste un cube dont chaque face est texturée avec une texture de votre choix *(par exemple essayez de faire une caisse en bois en utilisant cette texture : crate.png)*
- Pour l'instant, vous devez utiliser le nearest-neighbor interpolation pour le filtrage des textures *(https://en.wikipedia.org/wiki/Texture_filtering)*

Version 7 : *Mode wireframe*

- Implementer le mode de rendu wireframe, c'est à dire qu'au lien de rasterizer tout le triangle, vous ne rasterizer que les lignes reliant les points du vertex : https://i.ytimg.com/vi/amyBTL5z8jg/maxresdefault.jpg

- Faites en sorte qu'appuyer sur la touche F1 permettra de passer au mode wireframe et inversement (switch)

Version 8 : *Alpha-blending & Back-face culling*

- Changer le filtrage des texture pour le bilinear filtering

- Ajouter un membre float alpha à la classe Entity, qui par défaut vaut 1

- Ce membre est à multiplier par la composante alpha de chaque vertex du mesh pour avoir la composante alpha finale du vertex. Par exemple, si alpha vaut 0.5 et que vous êtes en train de rendre à l'écran un vertex de couleur (0, 200, 0, 255), he bien sa couleur sera en fait remplacée par (0, 200, 0, 127). L'intérêt est de pouvoir spécifier une valeur de transparence spécifique à chaque entité

- Rendre un gros cube texture opaque, et un autre cube texture plus petit devant transparent à 50% qui tourne sur lui-même.

- Pour le calcul de la couleur avec transparence, vous devez utiliser l'alpha-blending (https://fr.wikipedia.org/wiki/Alpha_blending). Pour info dans votre cas
    - Ca est la couleur du pixel actuel de l'écran (donc de la texture de rendu)
    - alpha_a vaut 1 (pas de transparence concernant les pixels déjà dessinés)
    - Cb est la couleur du pixel du triangle en cours de rendu
    - alpha_b est la transparence de ce pixel

- Vous ferez donc bien attention à rendre le cube opaque avant le cube transparent

- Faites en sorte qu'uniquement les faces avant soient rendues à l'ecran (back-face culling : https://en.wikipedia.org/wiki/Back-face_culling). Il faudra donc peut-être modifier les fonctions CreateCube et CreateSphere de façon à ce que les triangles soient orientés vers l'exterieur (donc modifier l'ordre des vertex du triangle)

Version 9 : Camera

- Ajouter la fonction Mat4 GetInverse() const à la classe Mat4

- La projection d'un point dans l'espace se fait par l'opération suivante :
  Vec4 projectedVertex = projectionMatrix * inverseCameraMatrix * entity.transformation * vertex.position

- On utilise l'inverse de la matrice de la caméra. En effet, déplacer la caméra de 2 unités vers la gauche revient à déplacer les vertex de 2 unités vers la droite

- Modifier la simulation de façon à avoir les contrôles clavier suivants :
  - Flèches W/S : avance/recule la caméra vers (0,0,0)
  - Flèches gauche/droite : tourne la caméra horizontalement autour de (0,0,0)
  - Flèches haut/bas : tourner la caméra verticalement (0,0,0)

  Donc la caméra tourne et se déplace en orbite autour du point (0,0,0), exactement comme quand vous contrôlez la caméra d'un jeu third-person comme Zelda ou Dark-Souls

- Faire une scène de votre choix qui montre bien toutes les features codées jusqu'à maintenant

Version 10 : *Anti-aliasing* *(20/20)*

- Implementer le Multi-Sample Anti-Aliasing 16x (https://mynameismjp.wordpress.com/2012/10/24/msaa-overview/)
- Ajouter la touche F2 permettant de switcher l'activation du MSAA

Il est impératif que votre code soit le mieux architecture possible *(choix de classes, de variables et de fonctions, respect de la POO, encapsulation et protection au maximum des variables membres, à l'exception des structures mathématiques, pas de memory leak, code le plus optimisé possible)*
et le plus propre possible *(pas de nombres en dur, une classe par fichier avec .h et .cpp, fonctions const chaque fois que c'est possible, taille des fonctions minimale, choix judicieux dans les noms de variables et fonctions, indentation et aération du code)***.**