Milestones

Deadline
Samedi 21/12/2019 – 18h00

Sujet

Il est impératif que votre code soit :

- **le mieux architecturé 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)*
- **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)***.**
- **Vous n'avez pas le droit à GLM (la lib mathématique) ou aux fonctions OpenGL autres que celles pour remplir la texture cible.**
- **Vous êtes bien entendu libres d'implémenter d'autres fonctionnalités que le minimum requis par le sujet.**
- Le projet devra se trouver sur l'espace GIT suivant :
  ssh://git@git.isartintra.com:2424/2019/MTL_Rasterizer/{nom_du_groupe}.git
- **Il y a une section Liens utiles en dernière page.**

Version 0 :

- 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 vecteur 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 vecteur par un facteur)*

      Mat4 : représente une matrice 4 x 4, la classe doit avoir un double tableau public pour représenter les éléments de la matrice.
      Elle doit surcharger les opérateurs suivants :
        - * (Mat4, Mat4) -> Mat4
        - * (Mat4, Vec4) -> Vec4

      Elle doit également contenir la fonction statique suivante :
        - 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

- Vertex : contient les informations suivantes :
  - Vec3 position

- Mesh : contient les informations membres suivantes :
  - std::vector<Vertex> 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 designe 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)
    (crée une sphère de rayon 1)

- Color : contient les 4 membres r, g, b, a pour la couleur, chaque composante étant representer par un unsigned char, valant 0 pour aucune intensité à 255 pour l'intensité maximale. *(Ex : (r, g, b, a) = (0, 0, 255, 255) représente la couleur bleue)*

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

  La classe ne doit pas avoir de constructeur par défaut, et doit avoir un constructeur qui prend en paramètre la largeur et la hauteur, et remplit les pixels en noir.
  Elle doit contenir la méthode SetPixelColor(unsigned int x, unsigned int y, const Color& c)

- Entity : représente une instance de mesh dessinée à l'écran *(on peut donc avoir le même mesh plusieurs fois dessiné à l'écran sans dupliquer les vertex et index buffer du mesh)*, l'entité contient donc les membres suivants :
  - Mesh* mesh : pointeur sur le mesh
  - Mat4 : transformation *(rotation, position, scale)* de l'entité dans le monde

- Scène : représente la scène dessinée à l'écran, contient donc les membres suivants :
  - std::vector<Entity> entities

  La scène doit détruire tous les éléments qu'elle représente *(meshs en l'occurrence)* lorsqu'elle est elle-même détruite.

- Une fois ces structures créées, il faut créer une classe Rasterizer et coder la fonction suivante : void RenderScene(Scene* pScene, Texture* pTarget) qui rend la scène en 3D dans la texture target.
  - Cette fonction doit d'abord remplir la texture pTarget de noir
  - Elle doit ensuite rendre chaque entité dans la texture en rasterisant chaque triangle **transformé** *(cad : les vertex sont transformés par la matrice de transformation de l'entité)* un par un
  - Rasteriser un triangle consiste à colorer les pixels correspondants de l'ecran *(http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html, https://en.wikipedia.org/wiki/Line_drawing_algorithm)*
  - Vous veillerez bien à ne pas traiter les pixels en dehors de l'écran en utilisant l'algorithme de *Sutherland-Hodgeman*
  - Pour l'instant, la couleur de chaque triangle est blanche
    (donc les pixels issus de la rasterization sont blancs)
  - Pour la projection, nous utiliserons pour le moment une projection orthographique codée en dur, telle que la projection d'un point de l'espace *(x, y, z)* sur l'écran donne le pixel *(p_x, p_y)* avec :
    - p_x = ((x / 5) + 1) * 0.5 * width
    - p_y = height - ((y / 5) + 1) * 0.5 * height (à caster en entiers)

    Par exemple, si width = 800 et height = 600
    - le point (-5, 5, 0) donne le pixel (0, 0) cad en haut à gauche
    - le point (0, 0, 0) donne le pixel (400, 300) cad au centre de l'écran
  - Pour l'instant la caméra est située à la position (0,0,0) et à la rotation (0,0,0)

- Créer un main qui fait les choses suivantes
  - Initialise une fenêtre GLUT de 1024 x 768
  - Créer une scène contenant contenant un cube de côté 1 un situé en (-2.5, 0, 2) et une sphère de rayon 1 située en (2.5, 0, 2)
  - Créer une texture de la taille de la fenêtre
  - Créer une boucle qui fait les choses suivantes
    - Rend la scène dans la texture
    - Affiche la texture à l'écran *(Transmet les pixels de votre texture à OpenGL pour que les pixels que vous avez calculé soient affichés à l'écran.)*

- L'application doit quitter lorsqu'on appuie sur la touche Echap ou lorsque l'on ferme la fenêtre

Version 1 : Couleur

- Ajouter le membre Color color à la structure Vertex

- Rendre à l'écran un mesh d'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

- Ajouter le membre Vec3 normal à la structure Vertex, cette normale représente la normale du mesh au vertex

- 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 matériau *(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, 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

- Jusqu'ici, seule une projection orthographique en dur a été utilisée. Modifier la fonction RenderScene de Rasterizer de la façon suivante : RenderScene(Scene* pScene, Texture* pTarget, const Mat4& projectionMatrix). 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

- 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)

- Créer une fonction statique dans Rasterizer Mat4 CreatePerspectiveProjectionMatrix(int width, int height, float near, float far, float fov)

- Rendre la scène précédente avec la matrice de projection perspective

Version 6 : texturing

- Ajouter les variables suivantes à votre classe Vertex : float u, float v, les coordonnées de texture

- Ajouter à la classe Texture un constructeur qui prend en paramètre une chaîne de caractère : l'adresse de l'image PNG à charger dans la texture *(vous pouvez vous servir de la librairie STB_image pour charger le fichier image)*

- Ajouter un membre Texture* pTexture à la classe Mesh *(nullptr indique que le mesh n'est pas texturé)*

- 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 permette 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

- Modifier la fonction RenderScene de Rasterizer en ayant les paramètres suivants : RenderScene(Scene* pScene, Texture* pTarget, const Mat4& projectionMatrix, const Mat4& inverseCameraMatrix)

- Par conséquent 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

- 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

-

Liens utiles

Drawing with Bressenham algorithm

Graphics Programming Black Book *(Attention ce qui est dit dans ce livre n'est plus forcément vrai)*

Rasterisation lien 1

Rasterisation lien 2

Rasterisation lien 3

Graphics Pipeline lesson from Ohio University

The graphics pipeline explained

http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/lectures/02_pipeline.pdf

http://web.cs.ucdavis.edu/~amenta/s12/perspectiveCorrect.pdf

Scratch A Pixel

Intel 64 and IA-32 architectures software developer's manual: Basic architecture

Intel 64 and IA-32 architectures software developer's manual: Instruction set reference, A-Z

Intel 64 and IA-32 architectures sofware developer's manual: System programming guide