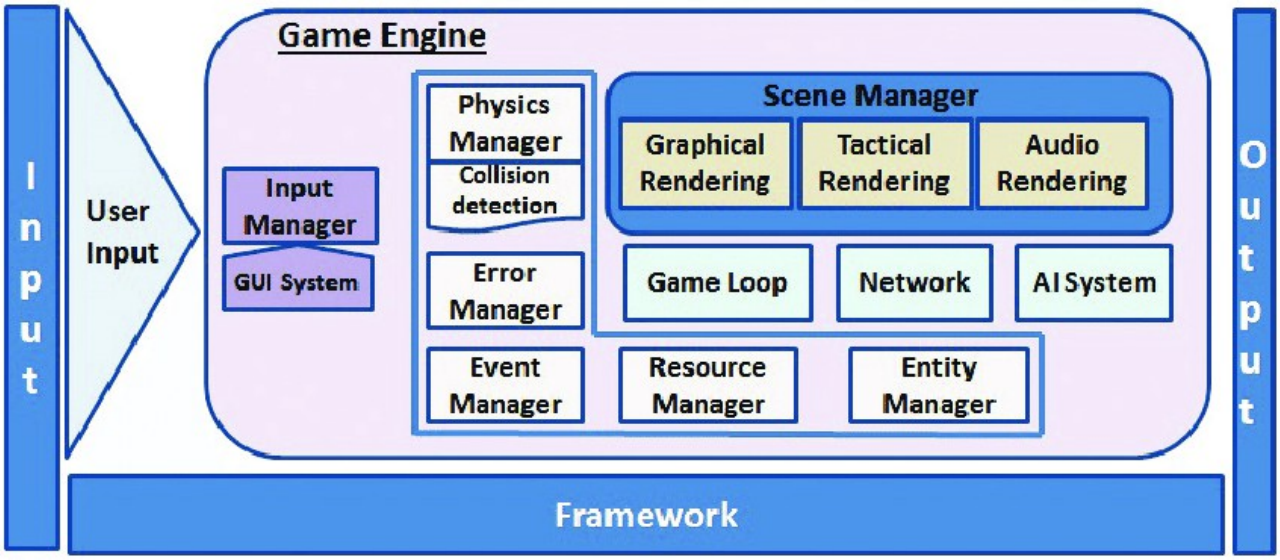
Source :
https://www.youtube.com/@TravisVroman
https://www.youtube.com/@GameEngineSeries
https://www.youtube.com/@tokyospliff
https://learnopengl.com
https://www.youtube.com/watch?v=C8YtdC8mxTU
https://indiegamedev.net/category/game-dev/game-engine-dev/
https://medium.com/@ghedger42/the-structure-of-video-game-engines-e29329f6ba2c
https://www.researchgate.net/publication/323179176_Game_Engine_Solutions#pf6
https://www.gamedeveloper.com/programming/writing-a-game-engine-from-scratch---part-1-messaging

Un moteur de jeu est un framework pour le développement de jeux-vidéo qui gère principalement les aspects suivant :

- partie graphique
- partie audio
- partie logique
- partie physique
- partie réseaux

En revanche, il n'existe pas de définition précise sur ce qu'est un moteur ,pas tout les moteurs ne prennent en charge toutes les parties cité au dessus par exemple.
Certain sont fait pour la création d'un jeu spécifique et certain sont fait pour n'importe quel type de projet. Il s'agit donc d'un ensemble d'outil qui aide à la création d'un jeu-vidéo et sont la pour accéléré le développement.

La structure d'un moteur de jeu :



Exemple de structure d'un moteur^

Un moteur a comme structure différent module indépendants mais qui communiquent entre eux chacun charger d'effectuer une tâche particulière.
Tout les moteurs on comme point d'entrée dans le programme une boucle **main** appelé à chaque frame. C'est cette boucle qui va appelé à chaque itération tout les autres composant du système selon le besoin actuel du jeu qui tourne. Cette boucle doit-être contrôler par l'état global du jeu.

Quelque modules essentiels :
Input manager
Module chargé de contrôler les entrées utilisateur

Logique
Élément qui constitue toutes la logique du jeu cet-a-dire les règles
exemple : le joueur est soumis à la gravité, si le joueur entre en collision avec tel ennemie il prend des dégâts etc...

Renderer
Module chargé d'effectuer l'affichage des éléments présent dans la scène. Cette étapes doit être réalisé en dernier dans la boucle car on affiche le résultat des calculs précédent.

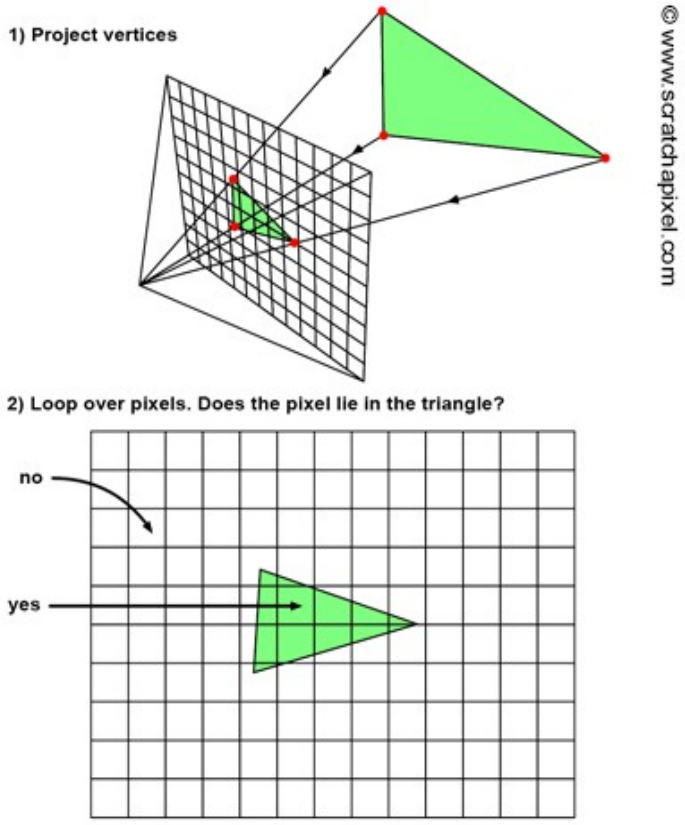
Ressource manager
Ce modules sert à gérer toutes les ressources du jeu cet a dire les modèles,les images, les sons etc. Il les charge dans la RAM quand les ressources sont nécessaires, vérifie qu'une ressource n'est pas chargé plusieurs inutilement, détruit les données de la RAM quand elle ne sont plus nécessaire …
Une bonne gestion des ressources est nécessaire pour que le moteur ai de meilleur performance.

Comment un moteur rend des graphisme en 3 étapes :

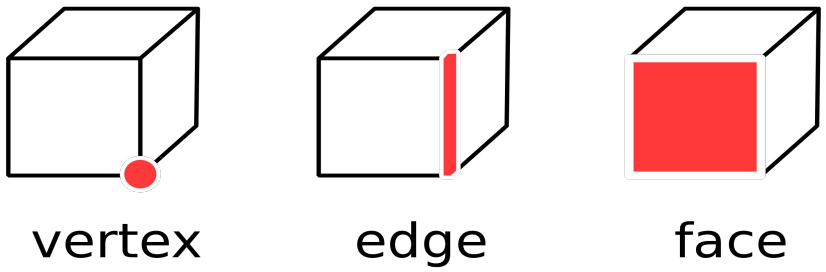
-Vertex shading
-Rasterization
-Fragment shading

Étape 1 vertex shading :

Consiste à prendre les mesh de tout les éléments apparaissant dans le champ de vision de la camera sur le plan 3d et de calculer la position de chaque objet sur un plan 2d qui correspond donc à l'écran ou est affiché l'image



Ce processus est lui même divisé en quatre étape qui sont appliqué pour chaque vertex (collection de point qui ont une position dans l'espace et qui forme l'objet en 3d).



Les quatre étapes dans l'ordre :
-calcul de la position du vecteur(vertex) par rapport à l'objet
coordonnée x,y,z **du vertex par rapport a l'objet**

-calcul de la position du vecteur par rapport à l'espace
coordonnée x,y,z **de l'objet par rapport au monde** et prise en compte de la rotation et de la taille de l'objet

-calcul de la position du vecteur par rapport à la camera (elle même dans l'espace)
coordonnée de la camera dans le monde et son champ de vision.

-calcul de la position du vecteur sur le plan 2d de l'affichage
toutes les valeurs calculé dans les étapes précédentes sont stockée dans des matrices qui sont utilisé ensuite pour calculé la position du vecteur sur l'écran.

Les vertex calculés sont assemblé pour former des triangles. Tout ces calculs sont géré par la carte graphique qui est un processeur optimisé spécialement pour ce genre de calcul de matrice.

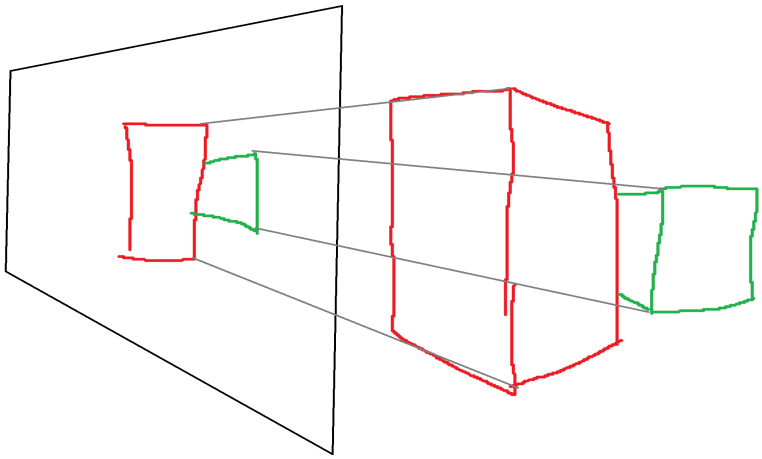
Étape 2 rasterization :

Consiste à prendre chaque triangle calculé précédemment et calculer quel sont les pixels qui sont concerné par ce triangle.
Le GPU prend en compte les coordonnées des trois vertex constituant le triangle pour ensuite le placer sur l'écran **en fonction de la résolution de celui ci**.
Est ensuite appliqué la couleur et l'ombrage en fonction de la texture assigné.

Avec cette étape on passe de triangle (ensemble de trois vertex) à des fragments qui sont des ensembles de pixels qui viennent du même triangle et qui ont la même texture.
Ce processus est appliqué à chaque triangle et on applique ensuite la valeur de rouge, vert et bleu à chaque pixel de l'écran pour obtenir notre image

Comment sont géré les éléments qui sont caché l'un par l'autre ?

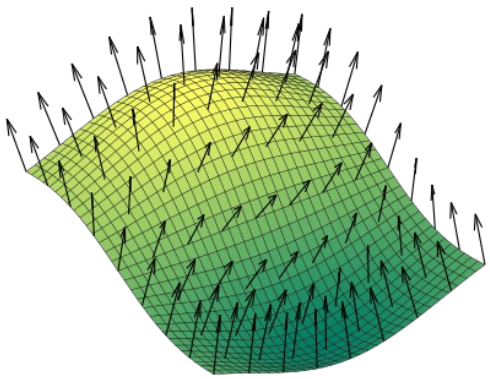
Par exemple ici l'objet vert est caché par l'objet rouge :



Pour calculer quel objet est caché derrière un autre, valeur de coordonnée Z est ajouté pour chaque pixel de l'écran (en plus de la valeur x et y). Cette valeur correspond à la profondeur cet a dire a la **distance par rapport a la camera.**
Quand un triangle et « rasterisé » on compare la valeur Z de chaque pixel du fragment et on l'a compare avec les valeur stockés :
Si la valeur Z du triangle et plus petite que celle dans le buffer (donc que le triangle est plus proche de la camera que celui dans le buffer) alors celui ci est affiché a l'écran et la valeur du z buffer est remplacé par sa valeur z.
Grâce à cette méhode, uniquement les triangle les plus proche de la camera pour chaque pixel seront affiché.

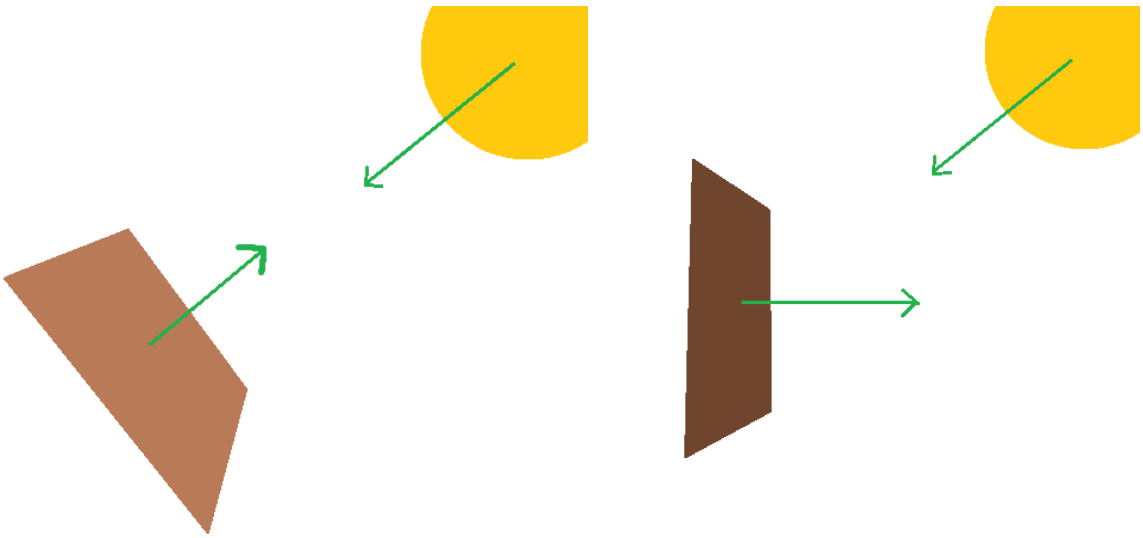
<u>Étape 3 fragment shading :</u>

Cette étape consiste à prendre en compte la direction de la lumières par rapport à la caméra pour calculer les ombres et la réflexion de la lumière sur les fragments précédemment calculé.
Pour cette étape, il faut prendre en compte chaque matériaux assigné aux objets 3D de la scène. Un matériel en 3d contient les informations sur la texture et le comportement de la lumière sur cette objet (par exemple un matériel métallique ou bois).
C'est avec ces informations que l'on calcul pour chaque fragments l'ombrage par rapport aux sources de lumière de la scène afin de la rendre réaliste.



Exemple des « normales » d'un mesh 3D qui correspond à la direction perpendiculaire de chaque face. Cette direction est comparé à celle de la source de lumière pour savoir si la face doit être plus

ou moins illuminé.



Plus la normal de la face est perpendiculaire a la direction de la lumière, moins la surface est éclairé. Une fois qu'on connais cette valeur, il faut prendre en compte l'intensité de cette source de lumière et appliqué la bonne couleur selon la couleur de base du fragment. Du coup on ajuste la valeur rgb des pixels du gragment.

Ce calcul est réalisé pour chaque fragment et sources de lumière de la scène.

Quel librairie utiliser ?

Pour coder un programme capable de rendre des graphisme sur notre écran on utilise une librairie de rendu, il en existe plusieurs dont les trois principales qui sont DirectX, OpenGL et Vulkan

DirectX est une librairie développé par Microsoft uniquement compactible avec les machine Windows et la Xbox.
OpenGL qui la plus connu, compactible avec de nombreuse plateforme donc mac et Linux mais qui en revanche commence à se faire vieille puisqu'elle date de 1992. Il est recommandé de commencé à utilisé celle ci pour les débutant.
Vulkan est la plus recente des trois lancé en 2014, elle est moins accessible au débutant mais plus performante avec l'hardware recent, mais étant donné que cet une librairie récente, les GPU trop vieux ne pourront pas faire tourner un programme écrit à l'aide de Vulkan.

Pour le language, il est préférable de coder en Cpp