Moteur graphique

Victor LASSERRE & Valentin SERVIERES

5 mai 2023

1 Introduction

Engin Pas Tangible est un moteur graphique reposant sur le principe de Ray Marching : un système de 3D similaire au Ray Tracing, mais beaucoup plus rarement utilisé. Ce système a certains avantages par rapport au Ray Tracing, comme par exemple de permettre une implémetation peu coûteuse de fractales, ou autres figures se reproduisant à l'identique.
Le Ray Marching repose sur la projection de rayons depuis une camera vers la scene. Pour projeter ces rayons, on les fait avancer pas à pas (*Marching*), avec la distance d'un pas dependant de la distance à la scene 3D.

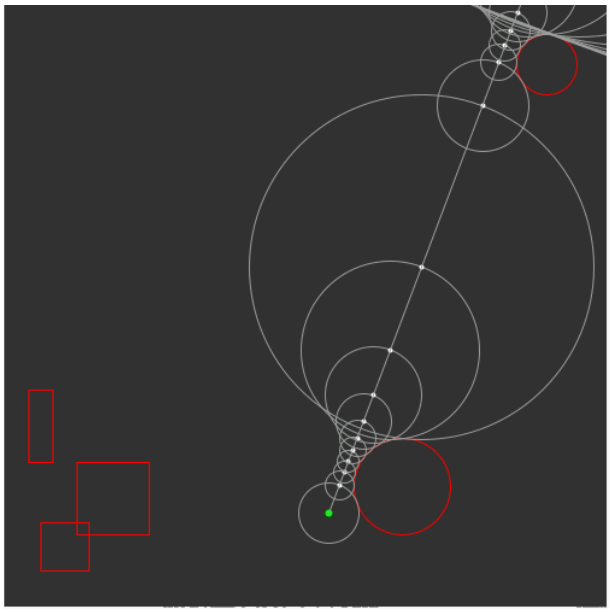

FIGURE 1 – Illustration 2D de la marche d'un rayon (on expliquera plus en detail dans 2.4 Projection des rayons)

2 Fonctionnement

2.1 Prérequis : Signed Distance Function

Le Ray Marching repose sur une fonction appelée SDF : Signed Distance Function. Ainsi, nous avons :

$$SDF : \mathbb{R}^4 \to \mathbb{R}$$
$$(x, y, z, t) \mapsto SDF(x, y, z, t)$$

Cette fonction doit retourner pour tout point M(t) la distance entre ce point et l'objet le plus proche. Cette distance peut et doit être négative si le point est à l'interieur d'une figure. Le temps permet de créer des objets dont la position, la forme, etc. dépendent du temps.

Ainsi, voici la SDF pour une scene 3D ne contenant qu'une sphere de centre o(t) :

$$SDF\_Sphere : \mathbb{R}^4 \to \mathbb{R}$$
$$P(x, y, z, t) \mapsto length(P - o(t)) - r$$

Avec r le rayon de la sphere

2.2 Methode de calcul : les Shaders

La méthode utilisée pour l'affichage des éléments à l'écran est un Shader. Il existe deux types complémentaires de Shaders : les Vertex Shaders qui "projettent" des Vertex (triangles) positionnés en 3D sur l'écran (mais seulement un nombre très limité) et les Fragments Shaders qui déterminent la couleur de chacuns des pixels des Vertex projetés sur l'écran.

Dans notre cas, nous n'utilisons que deux triangles qui forment le rectangle de l'écran. Notre moteur 3D repose principalement sur les Fragments Shaders.

On peut décrire un Fragment Shader comme ceci :

$$Frag : \mathbb{R}^2 \to \mathbb{R}^4$$
$$Pixel\_Position(x, y) \mapsto (Red, Green, Blue, Alpha)$$

On pourra également avoir besoin de rajouter des entrées "input" comme le temps, la position du curseur, le FOV, ect...

Il faut donc créer cette fonction !

2.3 Création des rayons projetés

2.3.1 Création d'une base orthonormée pour la caméra

On suppose que l'on connait *tilt* et *pan* qui sont les angles de la caméra donnés par le curseur.

$$e_z = [cos(tilt) \times sin(pan), sin(tilt), cos(tilt) \times cos(pan)]$$
$$e_x = normalize(e_z \wedge (0, 1, 0))$$
$$e_y = e_x \wedge e_z$$

Remarque : tous ces vecteurs sont orthonormés.

2.3.2 Création des vecteurs directeur pour chaque pixel

Avec x et y les coordonées du pixel et FOV le champ de vision ($x, y, FOV \in \mathbb{R}$) :

$$Direction = x \times e_x + y \times e_y + FOV \times e_z$$
$$Direction_{Normalized} = \frac{Direction}{\|Direction\|}$$

On obtient ainsi un vecteur directeur pour chaque pixel de l'écran.

2.4 Projection des rayons

Pour projeter les rayons partant du point $Origine \in \mathbb{R}^3$ et de direction $Direction \in \mathbb{R}^3$, on utilise la suite suivante :

$$(Pos_n)_{n\in\mathbb{N}} = \begin{cases} Pos_0 & = Origine \\ Pos_{n+1} & = Pos_n + SDF\_Scene(Pos_n) \times Direction \quad \forall \mathbf{n} \in \mathbb{N} \end{cases}$$

Remarque : Il faut que $Direction$ soit normalisée.

Si la suite **diverge**, alors le rayon n'a rencontré aucun obstacle. On affichera donc la couleur du ciel.
Si la suite **converge** vers une limite, alors cette limite est le point d'intersection entre la scene 3D et le rayon. On appelera ce point d'intersection le point **P**.

La section suivante explique comment determiner la couleur à afficher dans ce cas là.

2.5 Calcul de la couleur à afficher

2.5.1 Calcul de normale

Pour les calculs de couleurs, on va avoir besoin de la normale de la surface de l'objet "rencontré" par le rayon.

$$\vec{N} = Normalize(\vec{\nabla} SDF\_Scene(P))$$

avec $Normalize(u) = \frac{u}{\|u\|}$

Remarque : on utilise le fait que SDF_Scene est $\mathcal{C}^1(\mathbb{R}^3, \mathbb{R})$ sauf en certains points particuliers, comme par exemple à l'interieur d'un plan sans volume. On evitera donc d'utiliser certaines figures, au profit d'autres ayant des propriétés plus satisfaisantes.

2.5.2 Calcul de l'éclairage

On pose $SunDir \in \mathbb{R}^3$ la direction **normalisée** du soleil.

3 Fonctionnalités

3.1 Volumes élémentaires

3.1.1 Cube

3.1.2 Sphere

3.1.3 Cadre de cube

3.2 Transformations

Pour toutes les transformations sur les points des volumes, il suffit d'appliquer la transformation inverse sur la position de ce point.
Par exemple, si l'on souhaite translater un point vers les x croissants, il faut réduire d'autant la position en x.

3.2.1 Rotation autour des axes XYZ

$$
\begin{aligned}
R(x,y,z) =& R_{ex}(x)R_{ey}(y)R_{ez}(z) \\
=& \begin{bmatrix} cos(x) & -sin(x) & 0 \\ sin(x) & cos(x) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(y) & 0 & sin(y) \\ 0 & 1 & 0 \\ -sin(y) & 0 & cos(y) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(z) & -sin(z) \\ 0 & sin(z) & cos(z) \end{bmatrix} \\
=& \begin{bmatrix} cos(y)cos(z) & sin(x)sin(y)cos(z) & cos(x)sin(y)cos(z) + sin(x)sin(z) \\ cos(y)sin(z) & sin(x)sin(y)sin(z) + cos(x)cos(z) & cos(x)sin(y)sin(z) - sin(x)cos(z) \\ -sin(y) & sin(x)cos(y) & cos(x)cos(y) \end{bmatrix}
\end{aligned}
$$

3.2.2 Duplication des volumes

4 Historique

4.1 Visualisation 2D

4.2 Première version 3D

4.3 Ajout d'une source lumineuse

4.4 Ajout des ombres

4.5 Création d'un ciel

5 Conclusion