INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet scientifique

Moteur graphique 3D

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

LASSERRE Victor
SERVIERES Valentin

2MIC A
2022/2023

Soutenance le 6 juin 2023

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet scientifique

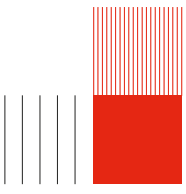
Moteur graphique 3D

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

LASSERRE Victor
SERVIERES Valentin

2MIC A
2022/2023

Soutenance le 6 juin 2023



Remerciements

Nous tenons à remercier notre tuteur de projet M. Sanchez, notamment pour son aide précieuse et ses conseils.

Nous remercions également Inigo Quilez pour la riche documentation sur le Ray Marching qu'il fournit librement sur son site web.

Table des matières

Introduction

EnginPasTangible est un moteur graphique reposant sur le principe de Ray Marching. Un moteur graphique a pour but de fournir des rendus (images) d'une scène 3D.

Le Ray Marching est un système de 3D similaire au Ray Tracing, mais beaucoup moins fréquemment utilisé. Ce système possède certains avantages par rapport au Ray Tracing. Il permet par exemple une implémentation peu coûteuse de fractales ou autres figures se reproduisant à l'identique.
Le Ray Marching repose sur la projection pas à pas (*Marching*) de rayons depuis une caméra vers la scène.

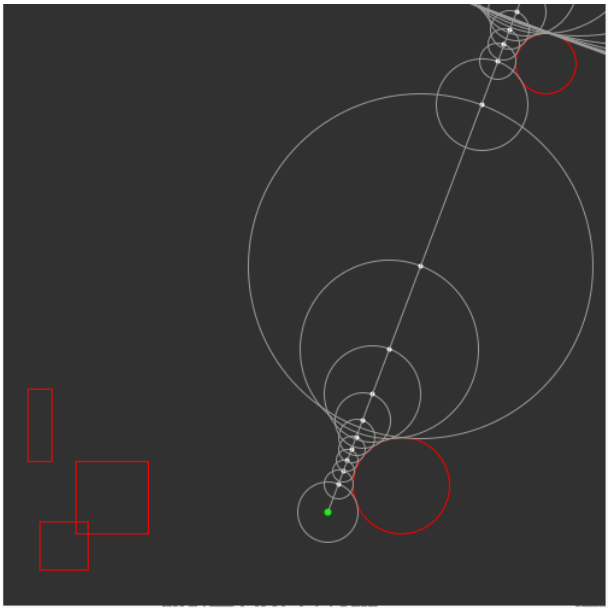


FIGURE 1 – Illustration 2D de la marche point par point d'un rayon (plus de détails dans 1.4 Projection des rayons)

Après plusieurs essais sur différentes bibliothèques de code, nous avons choisi d'utiliser le framework [1] open source GLFW. Celui-ci permet l'implémentation de OpenGL Shading Language (GLSL), un langage de shading. Les langages de shading permettent une implémentation rapide des vecteurs et matrices ainsi que leurs opérations associées.

GLFW étant fait pour le langage C, il permet une portabilité du code sur de nombreuses plateformes telles que Windows, macOS et Linux (X11 et Wayland).

Nos principales attentes du projet moteur graphique étaient d'apprendre à utiliser un langage de shading, et de créer un programme suffisamment performant pour obtenir des résultats en temps réel. Un autre objectif était de pouvoir se déplacer de façon fluide dans la scène 3D avec des commandes intuitives.

1. Infrastructure logicielle : bibliothèque de code générique

1 Fonctionnement

1.1 Prérequis : Signed Distance Function

Le Ray Marching repose sur une fonction appelée SDF : Signed Distance Function.
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'intérieur d'une figure. L'ajout du temps permet de créer des objets dont la position, la forme, etc. dépendent du temps.
Ainsi, nous avons :

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

Par exemple, voici la SDF pour une scène 3D ne contenant qu'une sphère de centre o(t) :

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

Avec r le rayon de la sphère et $length(\cdot) = \| \cdot \|$ la norme euclidienne.

1.2 Méthode 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 triangles positionnés en 3D sur l'écran (mais seulement en nombre très limité) et les Fragments Shaders qui déterminent la couleur de chacun des pixels des précédents triangles.
Dans notre cas, nous n'utilisons que deux triangles qui forment le rectangle de l'écran. Notre moteur 3D repose principalement sur les Fragment Shaders.
On peut décrire un Fragment Shader comme ceci :

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

On pourra également avoir besoin de rajouter des entrées "input" comme le temps, la position du curseur, la longueur focale (qui détermine le champ de vision), etc.
Il faut donc créer cette fonction !
Remaque : Cette méthode permet de paralléliser les calculs pour chaque pixel.

1.3 Création des rayons projetés

1.3.1 Création d'une base orthonormée pour la caméra

Pour projeter les rayons vers la scène 3D, on va d'abord avoir besoin de donner une direction à chacun de ces rayons. Pour cela, il va nous falloir une base orthonormée de vecteurs pour définir le plan de l'écran.
On suppose que l'on connait *tilt* et *pan* [2] qui sont les angles de la caméra donnés par le curseur.

$$\begin{aligned} e_z =& (cos(tilt) \times sin(pan),\ sin(tilt),\ cos(tilt) \times cos(pan)) \\ e_x =& normalize(e_z \wedge (0,1,0)) \\ e_y =& e_x \wedge e_z \end{aligned}$$

Avec $normalize(\cdot) = \frac{\cdot}{\|\cdot\|}$

Ainsi, on a e_x et e_y qui forment le plan de l'écran, et e_z qui est dans le sens opposé à la direction de la caméra.

Remarque : Cette base de vecteurs est orthonormée.

1.3.2 Création des vecteurs directeurs pour chaque pixel

Pour chaque pixel de l'écran, on va associer un vecteur directeur. Ce vecteur est calculé avec x et y les coordonnées du pixel et FL la longueur focale ($x, y, FL \in \mathbb{R}$) :

$$Direction = normalize(x \times e_x + y \times e_y - FL \times e_z)$$

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

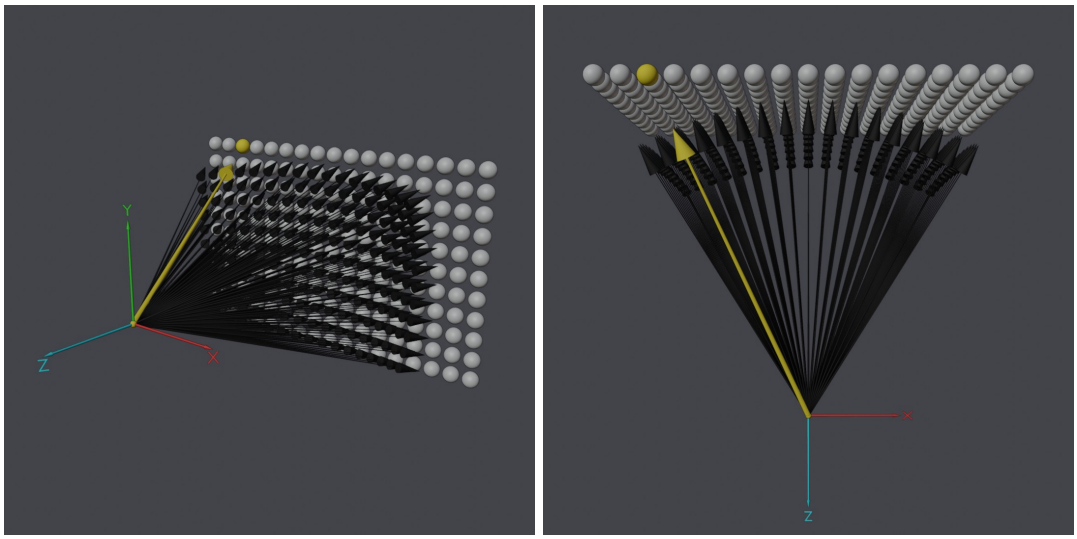


FIGURE 2 – Visualisation des vecteurs associés aux pixels de l'écran, chaque sphère représentant un pixel

2. Inclinaison et Panoramique de la caméra

1.4 Projection des rayons

Pour projeter un rayon, on le fait avancer pas à pas. La distance des pas doit être la plus grande possible, mais sans que le rayon ne traverse d'objet de la scène 3D.

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

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

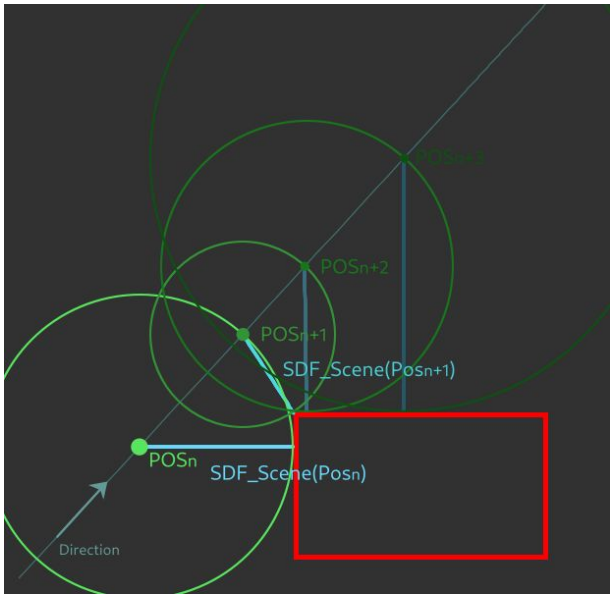


FIGURE 3 – Illustration 2D de la marche d'un rayon à la position Pos_n. Le rectangle rouge est un objet de la scène.

Remarque : La suite "avance" dans le sens de *Direction* avec un pas de $SDF\_Scene(Pos_n)$

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 scène 3D et le rayon. On appellera ce point d'intersection le point **P**.

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

1.5 Calcul de la couleur à afficher

1.5.1 Calcul de normale

Pour les calculs de couleurs, on va avoir besoin de la normale $\vec{N}$ de la surface de l'objet "rencontré" par le rayon.

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

avec $normalize(\cdot) = \frac{\cdot}{\|\cdot\|}$

Remarque : On utilise le fait que SDF_Scene est $\mathcal{C}^1(\mathbb{R}^3, \mathbb{R})$ sauf en certains points particuliers, comme à l'intérieur d'un plan sans volume. Cependant, ces points critiques sont négligeables par rapport au volume total des objets. On évitera d'utiliser certaines figures, au profit d'autres ayant moins de discontinuités. Par exemple, on préférera utiliser un pavé très fin plutôt qu'un plan en deux dimensions.

1.5.2 Calcul des ombres

Pour déterminer s'il y a une ombre à un point P, il suffit de projeter un rayon d'origine P et de direction $Sun\_Direction$ de la même manière que dans la partie 1.4 Projection des rayons. Si le rayon projeté converge, alors il a rencontré un obstacle et il y a une ombre. Sinon, il n'y a pas d'ombre.

Remarque : Le calcul des ombres est couteux en ressources.

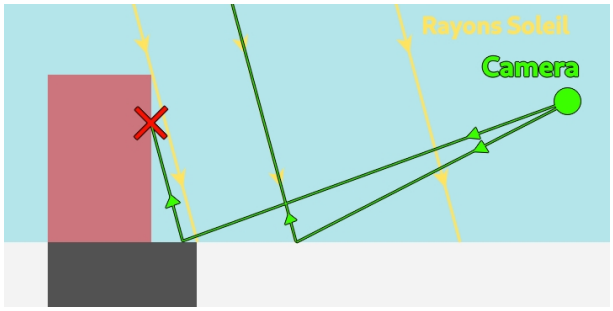


FIGURE 4 – Illustration du fonctionnement du calcul des ombres

1.5.3 Calcul de l'éclairage

On pose $LightPos \in \mathbb{R}^3$ la position d'une source de lumière et $LightColor$ sa couleur.
La contribution de cette source de lumière à l'éclairement d'un objet blanc de normale $\vec{N}$ au point **P** est donnée par ce calcul :

$$Contribution = \begin{cases} 0 & \textbf{si P est dans l'ombre} \\ max(\left\langle \vec{N},\ normalize(Lightpos - P) \right\rangle,\ 0) \times LightColor & \textbf{sinon} \end{cases}$$

Remarque : S'il y a plusieurs sources de lumière, il suffit de sommer la contribution de chacune de ces sources. Il n'est pas nécessaire de gérer d'éventuels problèmes de dépassement de couleur car le shader le fait lui-même.

1.5.4 Calcul du rouge, vert, bleu à partir d'une teinte

La couleur des objets est donnée par une valeur comprise entre 0 et 359 degrés. La valeur correspond à la teinte de l'objet. Comme montré sur le schéma ci-dessous, une valeur à 180 degrés (donc au milieu) correspondrait à du cyan.

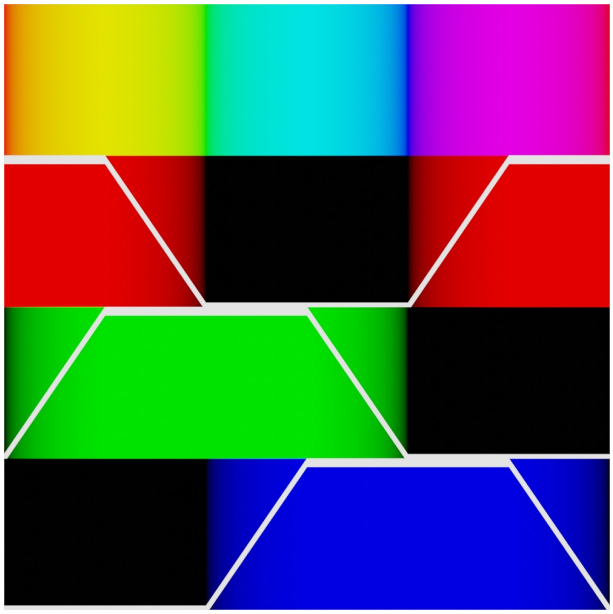


FIGURE 5 – Décomposition de la couleur

On remarque que l'on peut facilement calculer les valeurs de rouge, de vert et de bleu (comprises entre 0 et 1) nécessaires à l'obtention de la couleur souhaitée.
En effet, il suffit de découper la courbe tous les 60 degrés et on obtient des courbes constantes ou linéaires.
Pour reprendre l'exemple du cyan, en se plaçant à 180 degrés on remarque que le rouge vaut 0 et que le bleu et le vert valent 1.

1.5.5 Calcul final de la couleur d'un pixel

Il faut finalement combiner tous ces calculs de manière à obtenir la couleur finale à afficher. Voici le calcul de la couleur apparente d'un point P d'un objet de couleur $CouleurObjet$:

$$Couleur\_Pixel = min(CouleurObjet, Couleur\_Eclairage)$$

avec $min(\vec{u}, \vec{v}) = (min(u_x, v_x), min(u_y, v_y), min(u_z, v_z))$

2 Fonctionnalités

Une des contraintes du Ray Marching est que l'on ne peut pas mettre n'importe quel objet 3D dans la scène.
En effet, nous sommes limités aux objets 3D pour lesquels il existe une Signed Distance Function (SDF). C'est ce que l'on appellera volumes élémentaires. On peut ensuite combiner ces objets, soustraire leurs volumes, appliquer des transformations pour obtenir de nouveaux objets.

2.1 Volumes élémentaires

Dans cette partie, on s'intéresse aux fonctions Signed Distance Function (SDF) de différentes formes géométriques. On remarque que toutes ces fonctions supposent que l'objet géométrique est centré en O. Cela simplifie beaucoup les fonctions et il est très facile de leur appliquer une translation ou autre transformation, comme indiqué dans 2.2 Transformations.

2.1.1 Sphère

La sphère a la plus simple des SDF. En effet, la distance entre un point P et une sphère de rayon r est donnée par

$$distance = length(P) - r$$

Avec $length(x) = \|x\|_2$.
Remarque : On rappelle que le volume est centré en O.

2.1.2 Plan

La distance entre un point P et un plan de normale N (normalisée) et de hauteur h est donnée par

$$distance = \langle P, N \rangle - h$$

2.1.3 Et bien d'autres...

Il existe de nombreuses autres fonctions SDF. Cependant, elles deviennent très vite complexes à comprendre.
Voici par exemple la fonction SDF pour un pavé avec $taille = (largeur, hauteur, profondeur) \in \mathbb{R}^3$ Cette fonction utilise le fait que la fonction |.| sur un vecteur équivaut à une symétrie par rapport aux plans xOy, xOz et yOz. Or, un pavé centré en O possède ces trois plans de symétries. Ainsi, la distance d'un point P à ce pavé est donnée par le code suivant :

```
float SDF_Box(vec3 p, vec3 taille) {
    vec3 q=abs(p)-taille;
    return length(max(q,0.0))+ min(max(q.x,max(q.y,q.z)),0.0);
}
```

2.2 Transformations

Les transformations sont une des spécificités les plus utiles et uniques du *Ray Marching*. En effet, elles sont extrêmement peu coûteuses en ressources et elles permettent de créer des formes uniques. Pour toute transformation, il suffit, lorsqu'on veut obtenir la distance entre un point P de l'espace et un objet transformé, de prendre la distance entre la **transformation inverse** du point P et l'objet non transformé.

$$SDF\_Objet\_Tranforme(P) = SDF\_Objet(Transformation\_Inverse(P))$$

Nous allons maintenant voir quelques exemples.

2.2.1 Translation

Pour obtenir une translation de vecteur *translation* d'un objet, on peut utiliser le code suivant :

```
SDL_Objet(p-translation);
```

2.2.2 Rotation autour des axes XYZ

Pour une rotation autour des axes X, Y et Z, d'angles respectivement *alpha*, *beta* et *gamma* d'un objet, on utilise le code suivant :

```
SDL_Objet(Rotation(p,-alpha,-beta,-gamma))
```

Avec *Rotation* une application linéaire dont la matrice dans la base canonique est donnée ici :

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

2.2.3 Duplication des volumes

Il est également possible d'appliquer des transformations plus complexes, permettant par exemple la duplication d'objets. Pour dupliquer un nombre infini de fois un volume, il suffit d'appliquer un modulo (reste de la division euclidienne) sur la position de l'objet.
Le code pour dupliquer un objet à l'infini sur l'axe Ox, toutes les N unités, est celui-ci :

```
SDF_Objet( vec3( mod(p.x - N/2, N) + N/2, p.y, p.z))
```

Remarque : *vec*3 est un constructeur de vecteur 3D

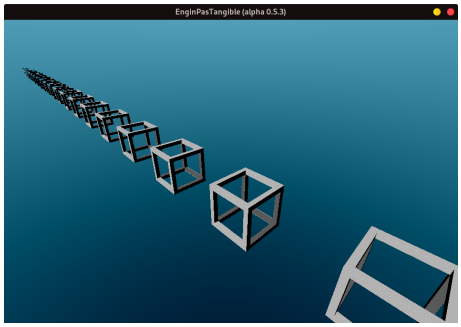


FIGURE 6 – Duplication d'un objet à l'infini selon l'axe Ox

2.2.4 Symétrie

Il est également possible de faire des symétries par rapport à un plan de normale N et de décalage $offset$. Le code est le suivant :

```
SDF_Objet(symetrie(p,-normale,-offset));
```

Avec le code de *symetrie* :

```
vec3 symetrie(vec3 position, vec3 normale,float offset){
    float d=dot(position,normale) + offset;
    if (d>0) return position;
    else return position-2.0*d*normale;
}
```

Avec *dot* le produit scalaire.
Remarque : Cette symétrie ne fonctionne que dans un seul sens. C'est-à-dire que l'un des côtés du plan "disparaît" pour laisser place au symétrique de l'autre côté du plan.

2.3 Vue orthogonale et perspective

Le moteur graphique permet de changer de mode de vue et permet de passer d'une vue en perspective (active par défaut) à une vue orthogonale.
Cette vue est très utilisée notamment dans les logiciels de modélisation 3D faits pour la mécanique.

Pour obtenir cette vue, il suffit d'assigner la position de la caméra comme étant la position de l'écran et d'affecter e_z (la direction de la caméra) aux vecteurs directeurs des pixels de l'écran.

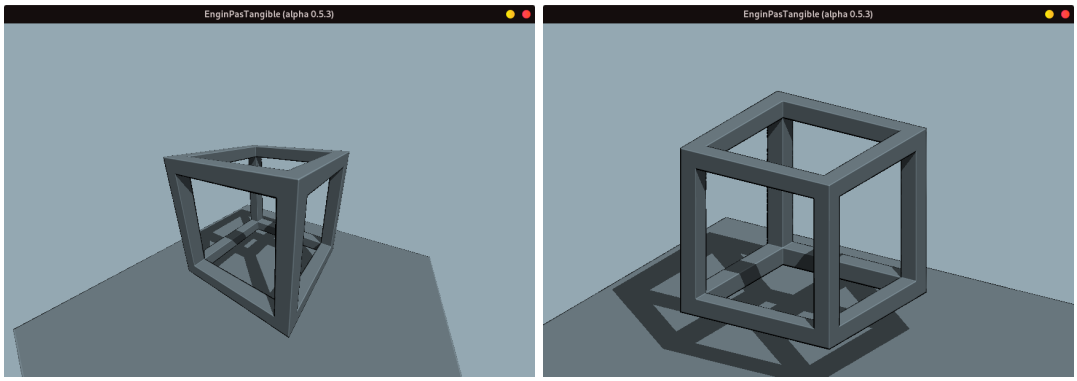


FIGURE 7 – Différence entre perspective et vue orthogonale

3 Historique

3.1 Première version 3D

La première version de l'affichage était simple. Si le vecteur de l'écran rencontrait un obstacle, alors il retournait la couleur RVB avec les composantes égales aux composantes XYZ du vecteur de la normale de l'objet au point de l'intersection. Sinon, cela signifie que l'on regarde le ciel, auquel cas on retourne une couleur bleue cyan.

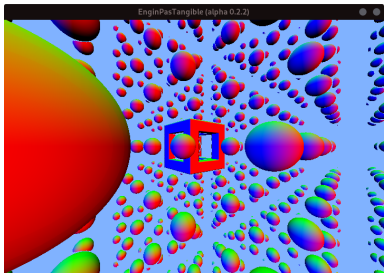


FIGURE 8 – La couleur des objets est calculée grâce à la direction de la normale

3.2 Ajout d'une source lumineuse et des ombres

Lors de la première version contenant une source lumineuse, il n'y avait que l'éclairement de l'objet. Puis, dans une version suivante, nous avons ajouté les ombres portées.

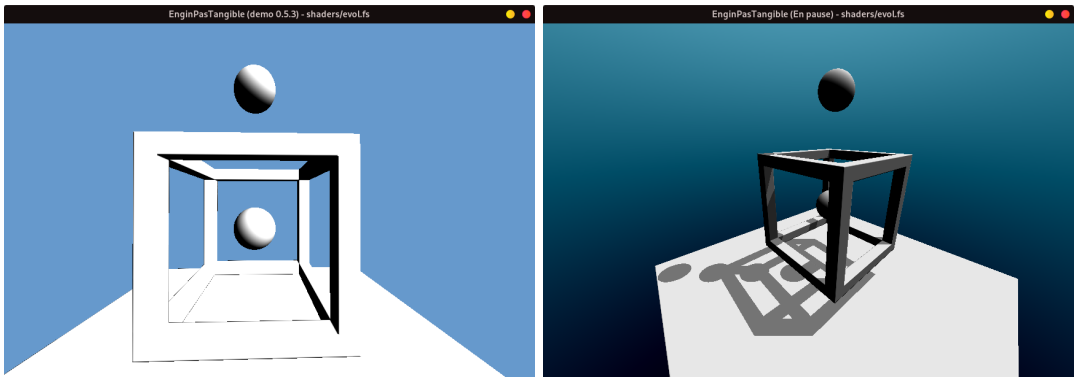


FIGURE 9 – Calculs des ombres sur les objets

3.3 Utilisations des transformations

Voici quelques exemples de transformation d'objets. La pyramide de Sierpiński est uniquement réalisée à partir de symétries et d'homothéties.

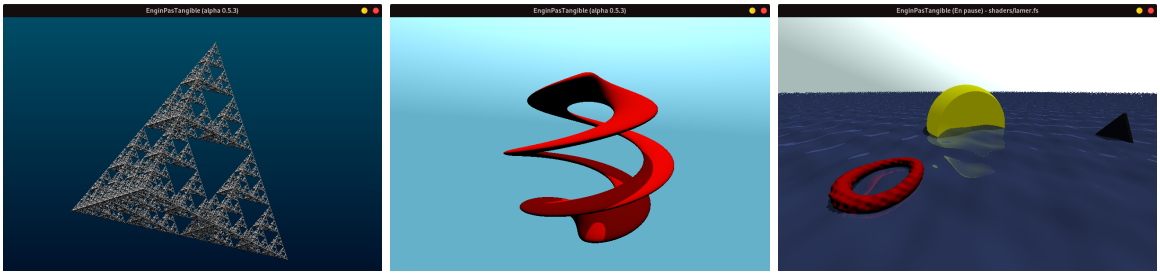


FIGURE 10 – Transformations d'objets à partir de fonctions

3.4 Création de scènes plus complexes

Nous avons commencé à faire des scènes plus complexes, avec différentes géométries, différentes transformations et plusieurs couleurs afin de tester les capacités du moteur. Le but était également de créer une scène qui puisse être agréable à regarder et qui permette de montrer ce que l'on peut faire avec ce moteur.

Nous avons également ajouté un ciel plus réaliste, qui rougit lorsque le soleil est bas sur l'horizon. Les composantes vertes et bleues de la couleur du ciel sont multipliées par le produit scalaire avec la direction des rayons du soleil.
Ainsi, il ne reste plus que le rouge lorsque le soleil s'approche de l'horizon.

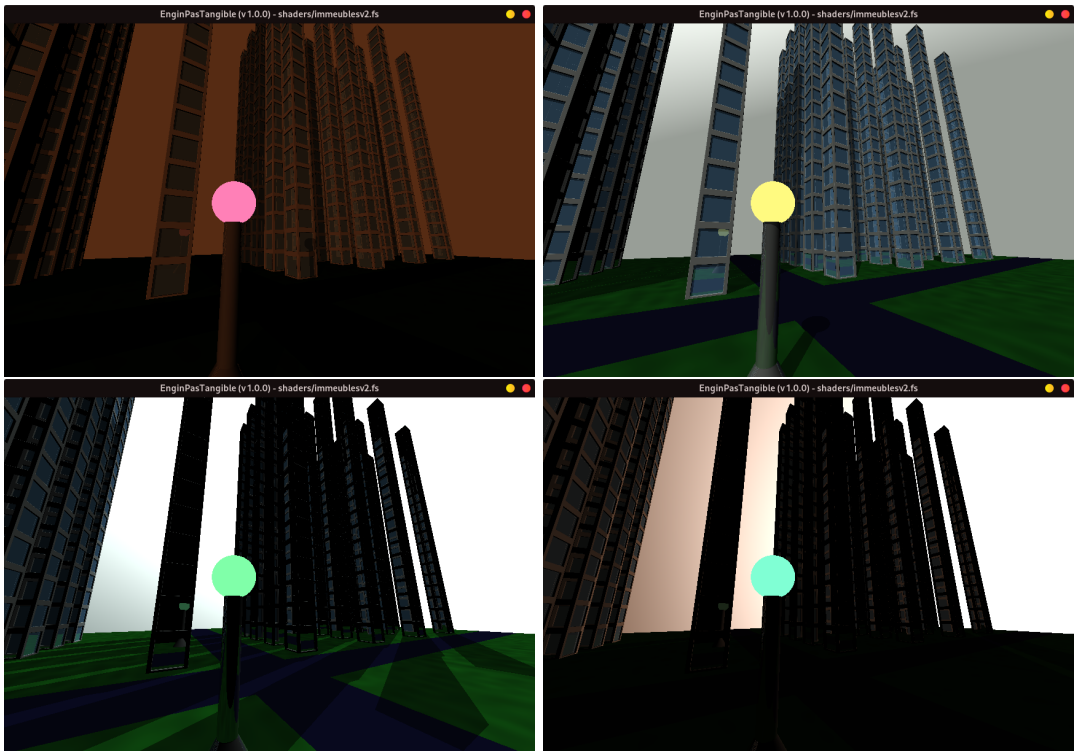


FIGURE 11 – Scène avec des immeubles et un cycle jour/nuit

Conclusion

Ce projet a été captivant et très motivant. Le fait de pouvoir observer rapidement les résultats de notre code était très agréable.
Nous avons rempli et même dépassé les objectifs que nous nous étions fixés.
Il était également intéressant de retrouver des notions vues en maths à l'INSA dans une application concrète.

L'expérience fut très enrichissante car nous avons dû nous organiser pour pouvoir travailler en binôme sur le même code avec l'utilisation d'un gestionnaire de version (git) auquel nous nous sommes formés.
Nous avons pu explorer le principe du Ray Marching, découvrir un nouveau langage, le GLSL, ainsi que les bases de développement logiciel.

Ce rapport était également l'occasion d'apprendre à maîtriser le LaTeX, mais c'était surtout l'occasion d'expliquer de façon claire et concise un concept technique.

Finalement, ce projet de moteur graphique nous a beaucoup apporté et nous pensons continuer à le développer dans le futur.

Informations complémentaires

Toutes les illustrations et captures d'écran sont réalisées par nos soins.
Le projet est téléchargeable ici : github.com/MagicTINTIN/EnginPasTangible

Bibliographie

[1] I. Quilez. Inigo quilez : : computer graphics, mathematics, shaders, fractals, demoscene and more. URL `https://iquilezles.org/`.

Table des Annexes

A1 Scène simple

Voici à quoi ressemble le code d'un fragment shader en entier.
Cette scène contient uniquement une sphère dans une maille de cube qui tourne sur elle même.
Code à mettre dans un fichier, nommé *example.fs* par exemple.

Example de fragment shader

```
#version 330 core
in vec2 FragCoord; //FragCoord = position pixel
in float Time; //parametre du shader

out vec4 FragColor; //FragColor = couleur sortie par le programme (r,g,b,a)

float SDF_Box_Frame( vec3 p, vec3 b, float e ){
    p = abs(p  )-b;
    vec3 q = abs(p+e)-e;
    return min(min(
        length(max(vec3(p.x,q.y,q.z),0.0))+min(max(p.x,max(q.y,q.z)),0.0),
        length(max(vec3(q.x,p.y,q.z),0.0))+min(max(q.x,max(p.y,q.z)),0.0)),
        length(max(vec3(q.x,q.y,p.z),0.0))+min(max(q.x,max(q.y,p.z)),0.0));
}

float SDF_Sphere(vec3 p,float r){
    return length(p)-r;
}

float SDF_Scene(vec3 p){
    return min(
        SDF_Box_Frame(p,vec3(0.5,0.5,0.5),0.1),
        SDF_Sphere(p,0.25));
}

vec4 Get_Impact(vec3 origin,vec3 direction){ //must have length(dir)==1
    vec3 pos=origin;
    float dist;
    int maxStep=60;
    for(int i=0;i<maxStep;i++){
        dist=SDF_Scene(pos);
        pos+=dist*direction;
        if(dist<=0.01) return vec4(pos,1.0);
        if(dist>=20.0) return vec4(pos,-1.0); //-1.0 => rayon a l'infini
    }
    return vec4(pos,-1.0);
}

vec3 grad(vec3 p){
    vec3 dx = vec3(0.01,0.0,0.0);
    vec3 dy = vec3(0.0,0.01,0.0);
    vec3 dz = vec3(0.0,0.0,0.01);
    return normalize(vec3(SDF_Scene(p+dx)-SDF_Scene(p-dx),
                          SDF_Scene(p+dy)-SDF_Scene(p-dy),
                          SDF_Scene(p+dz)-SDF_Scene(p-dz)));
}

vec3 Get_Color(vec3 origin,vec3 direction){
    vec4 impact = Get_Impact(origin,direction);
    if(impact.w<0.) return vec3(.5,.7,1.); //couleur du ciel
    vec3 normale = grad(impact.xyz);
    vec3 SunDirection = normalize(vec3(1.0,2.0,3.0));
    vec3 SunColor = vec3(1.0,0.9,0.5);
    vec3 CouleurObjet = vec3(1.0,1.0,1.0);
```

```
    return min(CouleurObjet,
               SunColor*max(dot(normale,SunDirection),0.0));
}

void main(){
    vec3 lookingAt = vec3(0.0,0.0,0.0);
    vec3 posCamera =
       vec3(2.0*sin(Time*0.5),1.0,2.0*cos(Time*0.5));//rotating camera

    vec3 ez = normalize(lookingAt - posCamera); //base orthonormee
    vec3 ex = normalize(cross(ez,vec3(0.,1.,0.)));
    vec3 ey = cross(ex,ez);

    vec3 direction = normalize(FragCoord.x * ex + FragCoord.y*ey + 1.0*ez);
    FragColor=vec4(Get_Color(posCamera,direction),1.0);
}
```