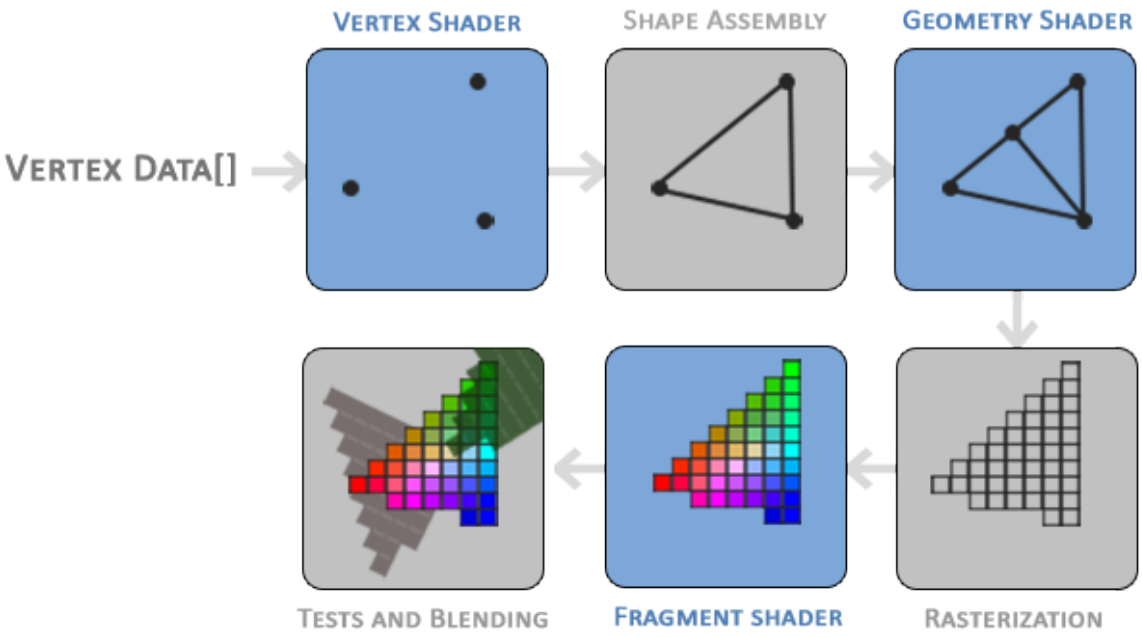
# Rasterization et rendu projectif

**5 ETI – CPE Lyon**

Majeure Image, modélisation et informatique

**Noms, Prénoms : Simon Jeffrey, Rakotoarivony Aurore**

**Date : 10/10/2023**




©https ://learnopengl.com/Getting-started/Hello-Triangle

# 1 Introduction

L'objectif fondamental du rendu graphique est de créer des images à partir de données tridimensionnelles, en simulant la façon dont la lumière interagit avec les objets dans un environnement virtuel. La méthode du rendu projectif consiste à projeter les objets 3D continues sur un écran 2D discrets et à les rendre réalistes en utilisant des techniques d'illumination et d'ombrage.
Au fil du temps, d'autres méthodes de rendu, telles que le raytracing, ont été développées. Le raytracing est une technique plus réaliste qui simule le trajet de rayons dans la scène, ce qui permet d'obtenir des rendus plus précis et esthétiquement plaisants. Cependant, le raytracing nécessite des calculs intensifs et prend considérablement plus de temps pour générer une image. Cela peut poser un problème, en particulier lorsque l'on vise des taux de rafraîchissement élevés, comme 60 images par seconde, sur des écrans haute résolution.
Malgré l'émergence du raytracing et de ses avantages en termes de réalisme, le rendu projectif reste une méthode extrêmement utilisée dans l'industrie du jeu vidéo et de la conception 3D en raison de sa rapidité et de sa capacité à produire des images en temps réel. En effet, la carte graphique a grandement accéléré les calculs parallèles de type matricielle dans la pipeline graphique. En utilisant des shaders, programme dans la carte graphique, les projections et les calculs d'illumination se retrouvent grandement accélérés.
Dans ce travail, nous nous concentrons sur la mise en œuvre d'un rendu projectif de triangles illuminés, en suivant une approche similaire à celle des cartes graphiques traditionnelles, afin de mieux comprendre et maîtriser les fondamentaux du rendu 3D.

# 2 Objectifs

Les objectifs du TP sont les suivants :

1. Implémentation du tracé de segments discrets avec interpolation de couleurs.
2. Implémentation du remplissage de triangles délimités par 3 segments discrets ainsi que l'interpolation de couleurs.
3. Implémentation de la profondeur.
4. Dessiner un ensemble de triangles appelé maillage et associer à celui-ci une texture positionnée aux bonnes coordonnées.
5. Mise en place d'un vertex et d'un fragment shader pour la projection et calcul de l'illumination dans le cas d'un triangle 3D en vue de son rendu sur une image 2D.

# 3 Réalisation

## 3.1 Tracer un Segment avec Interpolation de Couleurs

On souhaite tracer un segment à partir de deux points $p_0$ et $p_1$. Pour obtenir un vecteur discrétisé en pixels du segment, on utilise l'algorithme de Bresenham.

— **Configuration de l'Algorithme de Bresenham :**

L'algorithme de Bresenham trace une ligne en calculant les pixels à activer de manière incrémentielle. On considère uniquement les segments du premier octant. On se déplace donc positivement selon l'axes des abscisses et des ordonnées. Il décide du pixel suivant en fonction de l'erreur accumulée et ajuste cette erreur en fonction de la pente de la ligne. Ce processus se répète pour chaque pixel jusqu'à atteindre le dernier point du segment
Si les segments ne sont pas dans le premier octant on modifie les axes pour se retrouver dans le premier octant puis une fois la ligne discrétiser on fait l'operation inversion sur les axes.

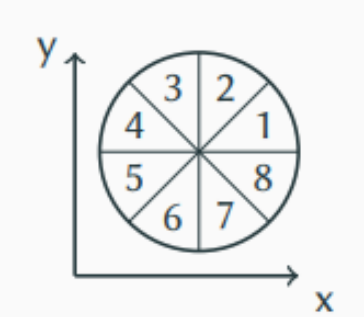



FIGURE 1 – Schéma des huits octants

Sur notre image le y est vers le bas donc le premier octant est situé en bas à droite.

— **Tracé du Premier Octant :**
Pour l'instant l'algorithme de Bresenham est configuré pour le premier octant. Pour tracer un segment du premier octant, il suffit donc de discrétiser le segment a l'aide de l'algorithme de Bresenham puis d'appeler la fonction `draw_line`.

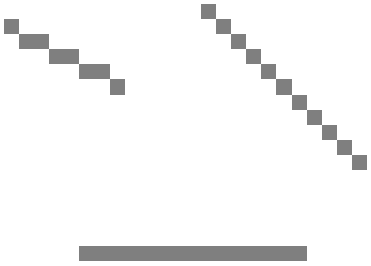


FIGURE 2 – Tracé de segment du premier octant.

— **Tracé des Autres Octants :**
Pour obtenir les autres octants on transforme les coordonnées x et y du segments pour se ramener dans le premier octant, soit en inversant les axes, soit par symétrie. Puis une fois la discrétisation effectuée on se ramène dans l'octant originel par symétrie inverse.

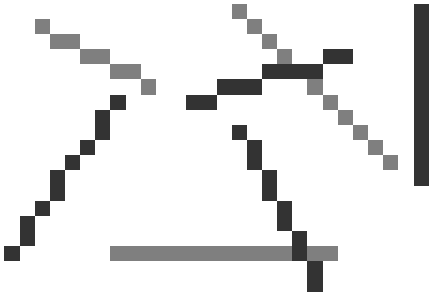


FIGURE 3 – Tracé des autres octants.

— **Interpolation des couleurs :**
On cherche désormais a obtenir un vecteur avec des couleurs interpolées en fonction des couleurs des points $p_0$ et $p_1$, que nous appellerons respectivement $c_0$ et $c_1$. La classe `line_interpolation_parameter` permet d'obtenir la valeur d'interpolation, $\alpha$ , pour chaque pixel, a partir des coordonnées du pixels et de $p_0$ et $p_1$. Cela permet une interpolation fluide des couleurs.
Une fois la valeur d'interpolation obtenue il suffit pour chaque pixel $p_i$ d'interpoler sa couleur en fonction de $c_0$ et $c_1$.

$$\text{couleur}(p_i) = (1 - \alpha) \cdot c_0 + \alpha \cdot c_1 \tag{1}$$

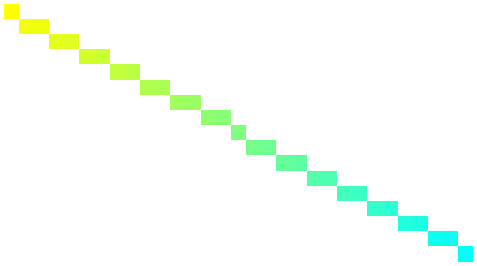


FIGURE 4 – Tracé des segments avec les couleurs interpolées.

## 3.2 Remplissage de Triangles et Interpolation de Couleurs

Le but est désormais de colorier un triangle.
Pour cela on utilise l'algorithme de scanline. L'algorithme de scanline est le suivant : on subdivise le triangle selon ses ligne $y$ puis pour chaque $y$ on vient interpoler les couleurs entre $x_{min}$ et $x_{max}$. Ce processus permet de remplir le triangle avec des couleurs interpolées en fonction de $x_{\min}$ et $x_{\max}$ pour chaque ligne $y$.
Autrement dit, pour chaque ligne, on remplit le pixel de gauche à droite avec la technique d'interpolation de ligne décrit ci dessus.

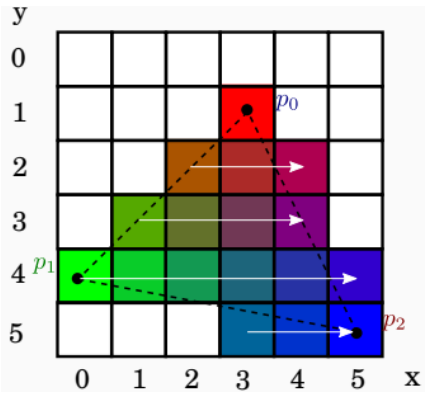



FIGURE 5 – Exemple de scanline pour le remplissage d'un triangle.

La structure `scanline` créées a partir trois points du triangles $p_0$, $p_1$ et $p_2$ et des couleurs respectivement $c_0$, $c_1$ et $c_2$ permet d'accéder aux valeurs des abscisses $x_{\min}$ et $x_{\max}$ de chaque ordonnées $y$ du triangle et de la couleur des pixels de $c_{\min}(x_{\min}, y)$ et $c_{\max}(x_{\max}, y)$.
On obtient le triangle suivant :

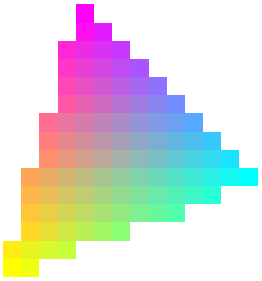


FIGURE 6 – Exemple de rendu de triangle avec interpolation des couleurs.

L'algorithme de scanline étant générique il peut être utilisé pour divers paramètres tels que la profondeur, les normales, les textures, la position des fragments et d'autres paramètres que nous jugeront utiles.

### 3.3 Gestion de la profondeur

On souhaite désormais gérer la profondeur des triangles pour savoir, lorsque deux triangles se superposent, lequel sera affiché. Pour cela on utilise une carte de profondeur ou depth buffer ou encore z-Buffer, initialisée a 1 par défaut, donc lors de la comparaison entre un objet et la carte par défaut l'objet sera forcément derrière.
Cette fois lors du parcours d'une ligne $y$ entre le $x_{\min}$ et $x_{\max}$ correspondant on vérifie a chaque pixel dans la fonction `draw_point`, a l'aide de la carte des profondeurs, si le pixel en question est plus proche que l'objet précédent. Si c'est le cas on actualise la couleur du pixel et la carte de profondeur. Si ce n'est pas le cas on passe au pixel suivant sans rien modifier.
En dessinant deux triangles l'un sur l'autre on obtient ce résultat :

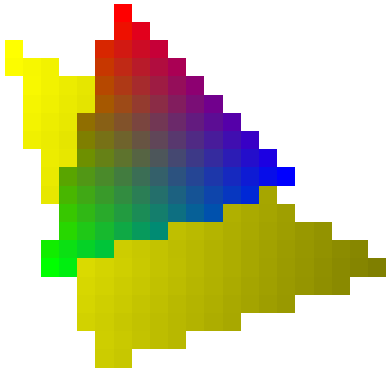


FIGURE 7 – Exemple de deux triangles dessinés en prenant en compte la profondeur.

Le triangle jaune a été dessiné en deuxième, mais comme il est derrière le triangle multicolore la partie du triangle jaune recouverte par l'autre n'a pas été dessiné.

## 3.4 Projection

Pour afficher les objets, nous devons travailler dans plusieurs repères et effectuer les transformations voulues dans les repères qui nous conviennent. Au départ, nous travaillons dans le repère objet. Il s'agit des coordonnées des différents points d'un maillage lorsque nous le chargeons. Nous devons placer cet objet dans un environnement. Pour cela, nous allons créer la matrice modèle. Cette matrice est constituée de translations et de rotations dans l'espace 3D. Pour exprimer la translation, cependant, nous devons ajouter une dimension. Nous obtenons ainsi le repère monde, car l'objet est positionné dans le monde.

Ensuite, nous devons exprimer le fait que la caméra est située à certains endroits du monde. Au lieu de déplacer la caméra, ce qui est compliqué, nous allons déplacer le monde autour de la caméra. La direction de la caméra est obtenue grâce à la fonction lookAt. Elle est également constituée de rotations et de translations, car nous faisons bouger le monde autour de la caméra. Si nous voulons avancer, par exemple, nous devons faire reculer le monde. Nous obtenons ainsi le repère caméra, qui est relatif à la caméra.

Pour exprimer la perspective et passer de l'espace 3D à l'espace 2D tout en conservant l'impression de profondeur, nous ajoutons une matrice de projection. Cette matrice divise non seulement les axes x et y par les coordonnées en z (un objet éloigné en z sera plus petit), mais permet également de définir un espace de clipping, qui est l'espace d'affichage. Si un objet n'est pas dans cet espace, il ne sera pas affiché. Par convention, OpenGL affiche les objets dans un cube compris entre -1 et 1. Cela correspond à une projection orthographique. Nous devons transformer cette projection orthographique, qui est un cube dans l'espace monde, en un frustrum dans l'espace projectif. Ce frustrum aura une forme définie par un near plane (znear) et un far plane (zfar), qui sont les deux bornes de notre espace le long de l'axe z, ainsi que par la fov (field of view), qui est l'angle qui étend l'espace de vue en profondeur.

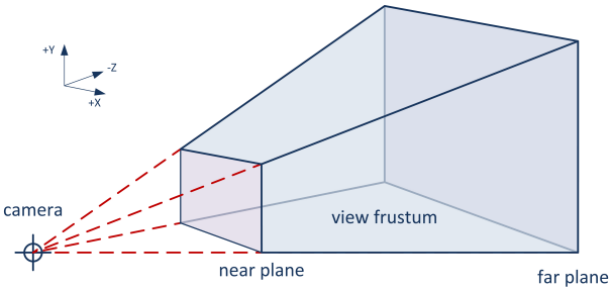



FIGURE 8 – Schéma explicatif du frustrum

Finalement, on obtient les nouvelles coordonnées dans l'espace projectif :

$$p' == model.view.projection.x$$

Les points sont projetés de cette manière. Les matrices vont s'appliquer différemment pour les normales. En effet, en prenant un point tangent $T$ au mesh au point de la normale $N$.

$$N.T = N^T T = 0$$

Soit la matrice $M$ à appliquer pour transformer la position afin de changer de repère. La matrice $G$ de changement de repère pour la normale sera différente. On veut :

$$(GN)^T(MT) = N^T G^T MT = 0$$

Donc $G^T M = I$ convient donc

$$G = (M^{-1})^T$$

Ainsi, avec cette connaissance de la matrice $G$, on a prouvé que pour exprimer les normales dans l'espace caméra, on a :

$$n' = (modelView^{-1})^T n$$

Ceci est la notation utilisée dans le TP, mais elle ne nous arrange pas, car nos futurs calculs d'illumination se feront dans l'espace monde afin d'obtenir une position absolue de la lumière dans le monde. Nous avons donc dans la suite du TP :

$$n' = (model^{-1})^T n$$

Sur OpenGL, les calculs matriciels se font en parallèle sur la carte graphique par l'intermédiaire du vertex shader. Ici, nous n'utilisons pas la carte graphique, mais nous allons simuler la fonction vertexShader en effectuant les calculs matriciels pour chacun des trois sommets du triangle. Dans un second temps, la fonction *render* permet de passer des coordonnées normalisées aux nombres de pixels correspondants.

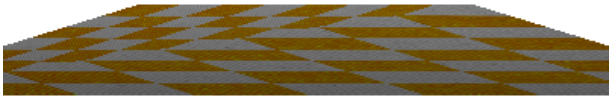


FIGURE 9 – Projection du plan

Le plan permet de montrer les résultats de la projection. La projection introduit néanmoins des composantes non linéaires. C'est un problème lorsque nous voulons interpoler des textures. Nous voyons ici que nous avons perdu la régularité du motif à cause de la projection des sommets. Nous voyons d'ailleurs combien de triangles composent le plan, car nous y observons les changements de régularité de la texture de manière discontinue.
Nous allons montrer le rôle de la matrice view et de la matrice model. Nous avons placé sur un maillage une source lumineuse qui sera fixe dans le monde.

FIGURE 10 – Image du dinosaure obtenu grace au fragment shader

Nous voyons sur la figure l'illumination du maillage. Nous mettons en place un changement de la matrice model en y insérant une rotation.

FIGURE 11 – Image du dinosaure après rotation avec la matrice model

Nous voyons que tout le maillage a subi la rotation. Nous observons aussi que la luminosité a changé car le dinosaure s'est déplacé relativement à la source de lumière fixe. Nous allons ensuite revenir à la matrice model sous la forme de l'identité mais en insérant la même rotation dans la matrice view.

FIGURE 12 – Image du dinosaure dans la rotation view

Nous voyons que la rotation du maillage est la même. Mais les coordonnées lumineuses sont les mêmes que sur la première figure. Car le dinosaure n'a pas bougé par rapport à la source lumineuse, mais c'est la caméra qui s'est déplacée. Le dinosaure se trouve au même cordonnées dans l'espace monde.

## 3.5 Maillage

Un maillage est un ensemble de point régit par des connectivités sous forme de triangle. Pour avoir un maillage, nous dessinons un ensemble de triangle. Les maillages utilisées contiennent des points, des normales, des coordonnées de textures expliqué ultérieurement. Nous parcourons chaque triangle pour obtenir un maillage.

## 3.6 Illumination

Le modèle d'illumination utilisé sera l'illumination de Phong. Il possède trois composantes physique à l'illumination. On réalise le calcul de l'illumination à la fin de la première étape de la projection. Elle dépend de la position de la lumière et des normales.
L'illumination globale que l'on nomme $c_{\text{shading}}$ correspond a la somme de l'illumination ambiante, de l'illumination diffuse et de l'illumination spéculaire. Il s'agit en quelque sorte du pourcentage de couleur affiché. Si on affiche 100% de la couleur, cela veut dire que nous avons une luminosité maximale.

- Illumination ambiante : il s'agit d'une lumière homogène, $I_a = K_a$. En effet, on supposera qu'il existera toujours une lumière ambiante et que le noir absolu n'existe pas
- Illumination diffuse : Il s'agit du produit scalaire entre le rayon émanant de la lumière et la normale de l'objet, multiplié par le coefficient de diffusion. $I_d = K_d\langle \mathbf{n}, \mathbf{u}_L \rangle[0,1]$. Autrement dit, nous calculons le cosinus entre le rayon lumineux et la normale. Lorsque la lumière est en face de

l'objet, le produit scalaire sera à 1 d'où une l'illumination maximale. Plus la lumière aura un angle important avec la normale, moins l'objet sera éclairé. L'illumination dépend donc de la normale, ce qui permet de donner l'impression de profondeur car la surface de l'objet ne sera plus homogène. Si la lumière est derrière l'objet, le cosinus sera négatif donc nous prendrons une illumination à 0.

— Illumination spéculaire : cette illumination donne un effet de brillance à l'objet. Il s'agit de la lumière réfléchit sur la caméra. On calcul donc, comme précédemment, un produit scalaire entre la direction de la caméra et la direction symétrique de la direction de la lumière par rapport à la normale. En effet, cela veut dire que nous prenons le rayon réfléchit (obtenue grâce à la fonction reflected). Plus ce rayon réflechit va en direction de la caméra, plus l'intensité spéculaire sera importante. En mettant un exposant $e_s$, nous pouvons mettre un pic sur la valeur. Un exposant élevé va laisser les valeurs proche de 1 mais va mettre les autres à 0. Le rond lumineux sera donc moins étalé dans l'espace. $I_s = K_s\langle \mathbf{s}, \mathbf{t}\rangle e_s[0, 1]$.

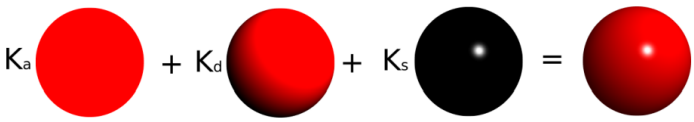



FIGURE 13 – Principe de l'illumination de Phong

Au final, la couleur du sommet est donnée par :

$$c_{\text{shading}} = (I_a + I_d)\mathbf{c} + I_s\text{blanc}.$$

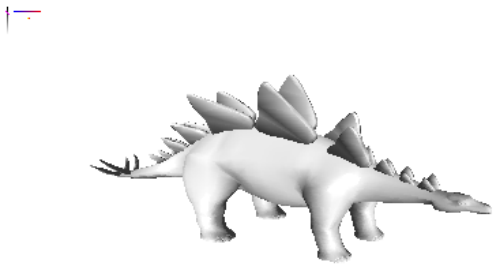


FIGURE 14 – Image du dinosaure sans texture

Nous obtenons le résultat suivant pour le dinosaure. Le dinosaure ressemble davantage à quelque chose qu'auparavant.
Faute d'avoir un fragment shader, nous effectuons les calculs dans le vertex shader. Cela signifie que les calculs d'illumination sont réalisés sur les 3 sommets du triangle et que les couleurs calculées seront ensuite interpolées entre les fragments.
Cette illumination a pour défaut de ne pas prendre en compte les positions relatives des objets. Nous n'avons pas d'objet masqué et par conséquent pas d'ombre. Certaines techniques comme les shadow maps peuvent résoudre ce problème. Cependant, elles nécessitent de rendre la scène deux fois car le principe des shadow maps est de rendre la scène du point de vue de la lumière afin de déterminer quels objets sont visibles par la lumière.
Le lancer de rayon pourra répondre à nos attentes

## 3.7 Texture

Nous souhaitons appliquer une texture à ce dinosaure. Une texture est une image qui va donc donner les couleurs du dinosaure. Pour cela, nous utilisons des coordonnées de texture. La texture est une image carrée, donc les coordonnées vont de 0 à un des deux côtés. Nous associons à chaque sommet du maillage la coordonnée de la texture. Ainsi, sur la texture, nous pouvons prélever un triangle et l'appliquer sur le triangle à dessiner.
Pour cela, nous allons interpoler les coordonnées des textures.

Une erreur commise en TP a été de traduire les coordonnées de texture des 3 sommets du triangle en couleurs et d'interpoler les couleurs pour dessiner le triangle. Cela ne s'est pas vu sur le maillage du dinosaure qui avait une structure assez fine, mais cela s'est vu sur le plan qui ne possédait que quelques points. L'effet a été de perdre les effets de la texture. Si nous avions un motif au milieu du triangle prélevé, le motif n'apparaissait pas. Nous avions donc des couleurs homogènes.

FIGURE 15 – Image du plan avec la texture interpolée par les coordonnées de la texture

Ici nous voyons le résultat lorsque nous interpolons les coordonnées texture. Nous conservons le motif. Néanmoins, nous avons un problème à cause de la projection comme décrit précédemment.

FIGURE 16 – Image du plan avec la texture interpolée par les couleurs et non les coordonnées

Ici nous voyons ce qui se passe quand on interpole la couleur de la texture des 3 sommets. Le motif est perdu.

FIGURE 17 – Image du dinosaure avec de la texture

Voici le résultat pour la texture du dinosaure.

### 3.8 Fragment shader

Nous mettons désormais en place la fonction fragment shader définie dans la fonction draw. En effet, notre gestion de l'illumination est grossière et ne se fait pas par pixel, mais par sommet de triangles. En introduisant le calcul de la lumière par fragment, nous obtenons une meilleure précision. Pour avoir un fragment shader, nous devons interpoler les normales et les positions des fragments à l'aide de l'algorithme scanline. Ensuite, nous effectuons les calculs comme précédemment.

FIGURE 18 – Image du dinosaure obtenu avec les fragments shader

Nous observons un rendu plus réaliste sur ce dinosaure. Regardons le plan plus en détail.Le maillage du triangle n'est pas fin. Il ne contient que 2 triangles, ce qui correspond à 6 calculs d'illumination. Observons :

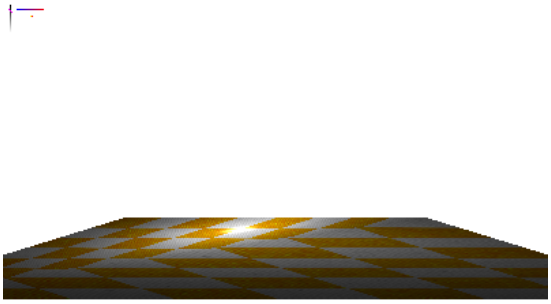


FIGURE 19 – Image du plan obtenu avec les fragment shader

Nous avons désormais l'illumination spéculaire. En effet, l'illumination spéculaire, si elle est au centre du triangle, sera oubliée si on ne calcule pas l'illumination par fragment. Nous avons également une couleur beaucoup moins homogène. Néanmoins, nous pouvons toujours observer les différences entre les deux triangles. De plus, nous avons évidemment plus de temps de calcul car nous effectuons des calculs par pixels au lieu de par sommets.

## 4 Conclusion

Le TP nous a permis de comprendre les bases du rendu projectif tel qu'il est utilisé par OpenGL. Nous avons vu comment, à partir du tracé de lignes, puis de triangles, nous pouvions charger un maillage complexe. Nous avons également abordé les bases des shaders pour projeter nos points dans l'espace et pour le calcul de l'illumination. Néanmoins, le manque de parallélisme des CPU s'est fait sentir, car les algorithmes sont beaucoup plus lents que sur GPU.