Ray tracing et snake

Louis Cohen & Anthony Strock

30 novembre 2014

Première partie

Principe du ray tracing

1 principe général

définition Le ray tracing est une méthode de génération d'images de synthèse en 3 dimensions dont le principe se base sur le parcours inverse de la lumière.

intérêt Bien que coûteuse en complexité, du fait du calcul du lancer du rayon de chaque pixel, qui développe un certain nombre de calculs (comme nous le verrons par la suite). La méthode de ray tracing permet un rendu particulièrement réaliste avec assez peu d'effort d'implémentation.

2 parcours inverse de la lumiere

utilisation de rayons Le principe du ray tracing (en français lancer de rayons), est basé sur l'utilisation d'objets informatiques appelés rayons (un point d'origine et un vecteur de direction) pour déterminer les points d'intersection avec les objets de l'environnement et les couleurs qui résultent de ces collisions.

parcours inverse de la lumière Le rayon ainsi lancé va suivre le parcours de la lumière dans le sens inverse (voir figure 1) : en effet il va partir de l'œil (eye), frapper l'écran de la focale en un point O, puis l'objet le plus proche en un point I, et enfin un rayon sera relancé du point I vers la source de lumière (light)

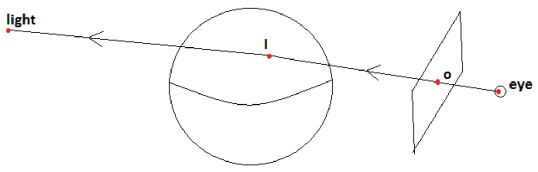

FIGURE 1 – illustration du parcours inverse de la lumière

3 formalisation mathématique des objets

intérêt Pour déterminer le point d'intersection du rayon avec l'objet le plus proche il faut d'abord pouvoir calculer le point d'intersection d'un rayon avec un objet. Pour cela il va falloir avant formaliser les objets d'un point de vue mathématique. Ici nous n'utiliserons que des formes simples, des plans et des sphères, donc les formalisations mathématiques seront relativement simples, dans des cas plus compliqués ont pourrait opter pour une approximation de la forme de l'objet.

cas des sphères Une sphère sera stockée en mémoire par :
- sont centre C
- sont rayon R
- sa couleur Col

cas des plans un plan sera stocké en mémoire par :
- Point d'origine P
- deux vecteurs v_1 et v_2 (de préférence orthogonaux) donnant les deux directions du plan avec respectivement $||v_1||$ et $||v_2||$ pour longueur suivant chaque direction (comme illustré dans la figure 2)

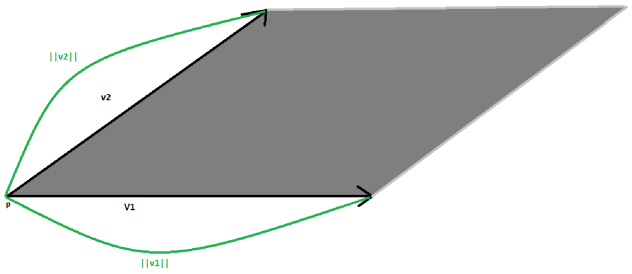

FIGURE 2 – exemple de définition de plan

On peut déduire de ces informations :

- le vecteur normal n=(a,b,c)
- l'équation de plan $ax + by + cz = k$ (avec $k = a.x_P + b.y_P + c.z_P$)

Maintenant que nous avons formalisé nos objets nous allons pouvoir calculer des distances d'un point à ces objets.

4 calcul de distances

calcul d'un point à une sphère Nous allons calculer le point d'intersection du rayon porté par u et partant de O avec la sphère S et en déduire donc la distance voulue. Le point d'intersection (si il existe) appartient à la sphère, donc il a une distance de R au centre C. De plus ce point d'intersection est sur le parcours du rayon si il existe t tel que OI=O+t.u. En combinant ces deux conditions on obtient la condition de la présence d'un point d'intersection avec

la sphère : si il existe t tel que $||OC-(O+t.u)|| = R$. Ce qui donne un polynôme du second degré en t :
$||u||^2.t^2 - 2.t.(OC|u) + ||OC||^2 = 0$
Les différents cas pour la résolution de ce polynôme correspondent aux différents cas d'intersection, nous ferons la distinction suivant le nombre de solutions en t et de leur signe :

- si il n'y a pas de solution : pas de point d'intersection avec la sphère.
- si une solution : un point d'intersection, et le rayon est tangent à la sphère.
- si deux solutions
  - si deux solutions positives : la sphère est devant l'écran dans l'espace que l'on peut bel est bien voir, on choisit la plus petite valeur de t car c'est le point d'intersection le plus proche du point d'origine O du rayon.
  - si deux solutions négatives : la sphère est derrière l'écran elle n'est donc en réalité pas visible
  - si une solution positive et une solution négative : ce cas est particulier, le point O est en réalité a l'intérieur de la sphère, on peut donc conserver la solution positive pour gérer le cas où l'utilisateur est dans une sphère.

calcul d'un point à un plan De la même manière, pour que le point d'intersection soit bien un point issu du rayon R partant de O il faut qu'il existe t tel que OI=O+t.u.
Et il faut que le point I appartiennent au plan donc :

- a fortiori il appartient au plan infini défini par l'équation du plan P : $a.x_i + b.y_i + c.z_i = k$.
  On vérifie que u n'est pas colinéaire au plan (sinon pas de point d'intersection) et si ce n'est pas le cas on obtient le point d'intersection (qui existe dans tous les cas puisque l'on a une intersection entre une droite et un plan infini, qui ne sont pas parallèles) en isolant t :
  $t = \frac{a.x_O+b.y_O+c.z_O-k}{a.x_u+b.y_u+c.z_u}$
- Mais une fois qu'on a le point d'intersection I du rayon avec le plan infini il faut encore vérifier que ce point d'intersection appartient au plan fini, *id est* :
  - $0 < (PI|v_1) < ||v_1||$
  - $0 < (PI|v_2) < ||v_2||$

gestion de plusieurs éléments nous fonctionnons dans un univers à plusieurs éléments. Pour déterminer le point d'intersection du rayon avec tout ces éléments il suffit de calculer le point d'intersection du rayon avec chacun de ces éléments et choisir la plus petite distance. (dans le cas de la figure 3 le point I est plus proche que le point I_2)

un mot de complexité Nous voyons donc que en posant n le nombre de pixels qu'on gère dans la fenêtre graphique (et donc le nombre de rayon à envoyer) et m le nombre d'objets la complexité est en $O(n \times m)$.

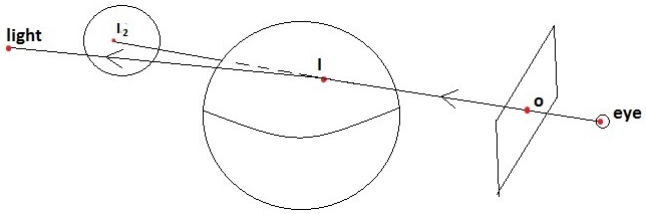

FIGURE 3 – cas de plusieurs objets

5 rendu lumière

Maintenant que nous avons obtenus le point d'intersection grâce à notre formalisation mathématique nous allons pouvoir déterminer la couleur que nous allons renvoyer sur le pixel dû a ce point d'intersection.
Pour cela nous remarquerons que dans la vraie vie l'éclairage d'un objet dépend de sa position par rapport à la source lumineuse. Nous allons donc étudier le produit scalaire entre la normale à la surface touchée et le vecteur issu de la source lumineuse et dirigée vers le point I sur l'objet (voir figure 4).

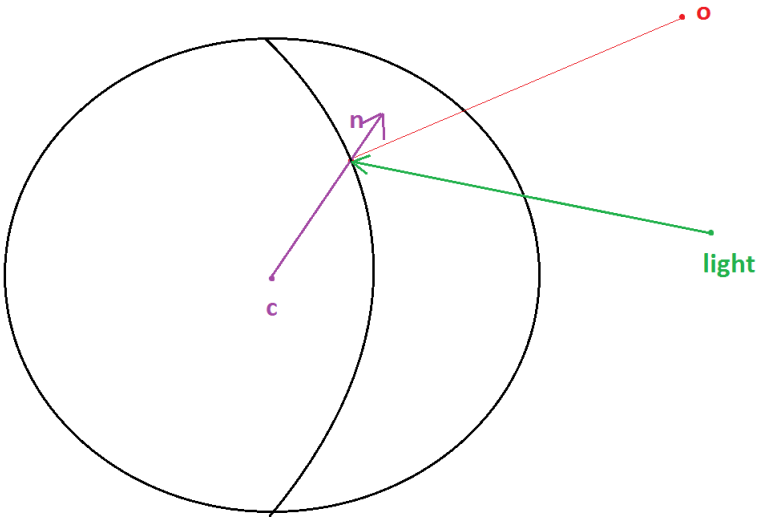

FIGURE 4 – illustration du produit scalaire

On fait donc le produit scalaire $k = -(n|ray_light)$, en normant les deux rayons $k \in [-1, 1]$. Si le produit scalaire $(n|ray_light)>0$ on renvoie la couleur noire car la lumière vient de derrière la surface de l'objet donc objet n'est pas éclairé en ce point, on ne traite donc ensuite que le cas $k \in [0, 1]$. On a donc l'apparition d'un coefficient que l'on va appliquer a la couleur de la sphère, en l'appliquant a chaque composante RGB de la couleur :
$(k.couleur)_{blue} = k.(couleur_{blue})$
$(k.couleur)_{red} = k.(couleur_{red})$

$(k.couleur)_{green} = k.(couleur_{green})$

Et si il y a plusieurs lumières on effectue le même procédé pour chaque lumière, cela nous produit un ensemble de couleurs dont on fait la moyenne où la moyenne de couleurs est une moyenne composante par composante.

Deuxième partie
snake

1 Généralisation et simplification

Notre objectif dans cette partie du projet était d'arriver à faire de l'animation avec un moteur de lancer de rayon. Pour cela la première étape a été d'essayer d'en simplifier son utilisation tout en essayant de laisser la possibilité d'en généraliser son cadre d'utilisation.

Nous avons donc choisi d'implémenter trois classes qui vont nous permettre de gérer facilement le rendu : SceneObject, Camera, Scene.

1.1 SceneObject

Nous avons choisi d'essayer d'avoir la possibilité de manipuler des objets différents, pas uniquement des sphères... Donc pour cela on a créé une classe virtuelle SceneObject qui va permettre de regrouper tous les types d'objets que l'on aimerait rajouter à notre scène. Pour cela on s'est posé une question : Qu'est-ce que des objets on en commun dans leur utilisation dans un moteur de lancer de rayon ?

Dans un moteur de lancer de rayon on a principalement besoin de savoir si lorsqu'on lance un rayon il intersecte des objets, mais pour le traitement des lumières on a aussi besoin d'arriver à récupérer la normale à l'objet en le point d'intersection avec le rayon.

On a donc créer la classe virtuelle SceneObject en utilisant la réponse à cette question, un SceneObject aura une méthode send qui permettra de savoir si le rayon a intersecté le SceneObject en question et qui renverra la distance au point d'envoie du rayon, et une méthode intersect qui permettra de générer un objet hit qui contiendra les informations nécessaires au traitement à savoir un rayon, un point d'intersection, une normale et une couleur. On a aussi choisi de rajouter un attribut type à un SceneObject pour avoir le choix à l'avenir de traîter différement les objets sur la scène s'ils ont des types différents.

1.2 Camera

On a choisi de modéliser une caméra par un point et un plan de projection. Ensuite la seule chose qu'on demande à une caméra c'est, étant donné un screen et une liste d'objets et de lumières afficher sur le screen la scène correspondante. Ceci est la méthode principale d'une Camera. En pratique le traitement se fait pixel par pixel, et donc on a une méthode qui permet le traitement précis d'un pixel.

1.3 Scene

Quelle donnée essentielle contienne une scène ? Finalement à cette question on a répondu :

- Une liste d'objets implémentée par une liste de pointeur vers des SceneObject
- Une liste de lumières à appliquer à la scène
- Une liste de caméras
- Une caméra courante
- Un écran où l'on doit afficher la visualisation de la scène

Et donc la seule chose qui reste à faire à partir de là c'est d'implémenter une méthode display qui va afficher la visualisation de la scène par la caméra courante sur l'écran.

2 Choix du Snake

Pourquoi avoir choisi d'implémenter un snake ? Comme nous l'avons dit plus haut, notre objectif était d'essayer de faire de l'animation avec un moteur de lancer de rayon. Et en s'étant aperçu que avec une configuration avec peu d'objets le moteur de lancer de rayon n'était pas tellement long nous nous sommes proposé d'essayer d'atteindre les limites de l'animation et d'essaye de faire une animation en temps réel avec une gestion de jeu. Et pour cela le snake était un bon choix pour plusieurs raisons :

- Les objets de la scène vont n'être que des objets simples nécessitant peu de calculs : des sphères ou des plans
- Le jeu en lui même n'est pas d'une très grande complexité et ne demande pas beaucoup de calculs

3 Approche globale du Snake

Pour commencer le Snake que l'on a implémenté est une classe fille de Scene à laquelle on va rajouter des méthodes pour gérer l'animation et les gestions d'entrées au clavier.

3.1 Objets de Snake

Nous avons modélisé le serpent qui se déplace par une tête et des morceaux de queues qui se déplacent en suivant leur prédécesseur pour cela on modélise ses deux types d'objets par des classes toutes deux héritant de sphère. Comme elles héritent de sphère leur affichage est déjà géré par la classe sphère, il nous reste donc plus qu'à implémenter un déplacement. Pour cela en ce qui concerne la tête on rajoute une méthode move qui prend en argument un vecteur de déplacement et qui déplace le centre de la sphère en direction, sens et norme du vecteur de déplacement donné en argument. En ce qui concerne les morceaux de queue, elles ont en attribut un pointeur vers le morceau du serpent qui les précede, cela permet par exemple de déplacer le morceau de queue en direction de son prédecesseur en restant à une distance fixe. Ensuite pour gérer plus facilement le déplacement nous avons créé une classe Body qui s'occupe principalement de gérer les déplacements dans le bon ordre des éléments du serpent, d'abord la

tête bouge, ensuite le premier morceau de queue et ainsi desuite... Cette classe Body nous permettra aussi de gérer les ajouts de morceaux de queue au serpent.

Dans le Snake il y aussi le morceau de nourriture qui permet de faire grandir le serpent quand celui-ci le mange, quand la tête rentre en collision avec lui, celui-ci n'ayant pas besoin de méthode spéciale il ne sera rien plus qu'une sphère, nous avons ajouté aux sphères elle-même une méthode collision avec d'autres sphères.

3.2 Initialisation du jeu et boucle principale

Le jeu débute avec l'ajout à la scène d'un objet tête de serpent(Head) et d'un objet queue de serpent(Tail) qui pointe vers le centre le la tête du serpent. Par peur de la lenteur du rendu nous avons placé le premier morceau de nourriture dans la trajectoire direct du serpent. Nous gardons en attribut un vecteur _direction qui nous permet de savoir dans quelle direction se déplace le serpent et à quelle v̈itesse: On va laisser le serpent bouger uniquement dans une zone réduite de l'espace, déjà il ne pourra bouger que dans une plan, et ce plan sera fini, dès qu'il sortira de ce plan le jeu prendra fin.

Le jeu en lui même s'organise autour d'une boucle de traitement simple, on met en attribut de snake un booléen _continue qui nous permettra de savoir si le jeu a pris fin ou pas. Tant que ce booléen est vrai on suit le schéma de traitement suivant :

- Tant que le jeu continue :
  - Si jamais il y a une entrée clavier alors
    - Si on a appuyé sur la flèche gauche on fait rotationer le vecteur _direction dans le sens direct sur le plan
    - Si on a appuyé sur la flèche droite on fait rotationer le vecteur _direction dans le sens horaire sur le plan
  - On fait bouger le serpent en suivant la direction du vecteur _direction
  - On regarde d'abord si le serpent est sorti en dehors de sa zone, si jamais c'est le cas le jeu prend fin sinon on regarde si le serpent a collisioné autre chose sur le plateau
  - Si la tête du serpent a collisionné la nourriture, on rajoute un morceau de queue au serpent et on replace autre part le morceau de nouriture, la nouvelle position est choisie aléatoirement
  - Si la tête du serpent a collisionné un morceau de queue alors le jeu prend fin
  - On affiche la scène

4 Résultats

Le jeu en lui même fonctionne bien et est jouable sur un ordinateur décent, de plus si l'on doit retenir une chose c'est que même s'il apparait quelque peu lent, il n'a été utilisé aucune méthode d'optimisation ou d'accélération. Nous sommes restés simples et n'avons ni utilisé la carte graphique, et nous n'utilisons qu'un seul coeur du processeur. On peut imaginer pouvoir avoir des résulats bien meilleurs cependant il ne faut pas oublier non plus qu'on a choisi d'implémenter un jeu simple qui ne nécessite pas de gros efforts de rendu graphique et dans lequel on ne manipule que des objets simples.

A Annexe : auto-évaluation

Points forts

— Fort engagement du début a la fin du projet malgré les difficultés rencontrées
— Tentative d'originalité
— Rendu finalement utilisable

Points faibles

— Mauvaise gestion du temps : projet trop ambitieux
— Application ne marchant pas à la date de rendu, projet presque invalide
— Idée du temps réél considérée par beaucoup comme très peu intéressante

Améliorations possibles

— Parallélisation sur carte graphique
— Insertion d'un moteur 3D plus intéressant (d'un autre groupe ?)
— Calcul des pixels a chaque étape plus ciblés (distance d'une boule du snake

Difficultés à anticiper

— Plus de communication dans le groupe
— Meilleure anticipation des problèmes possibles
— Problème de gestion de pointeur et destruction d'objets