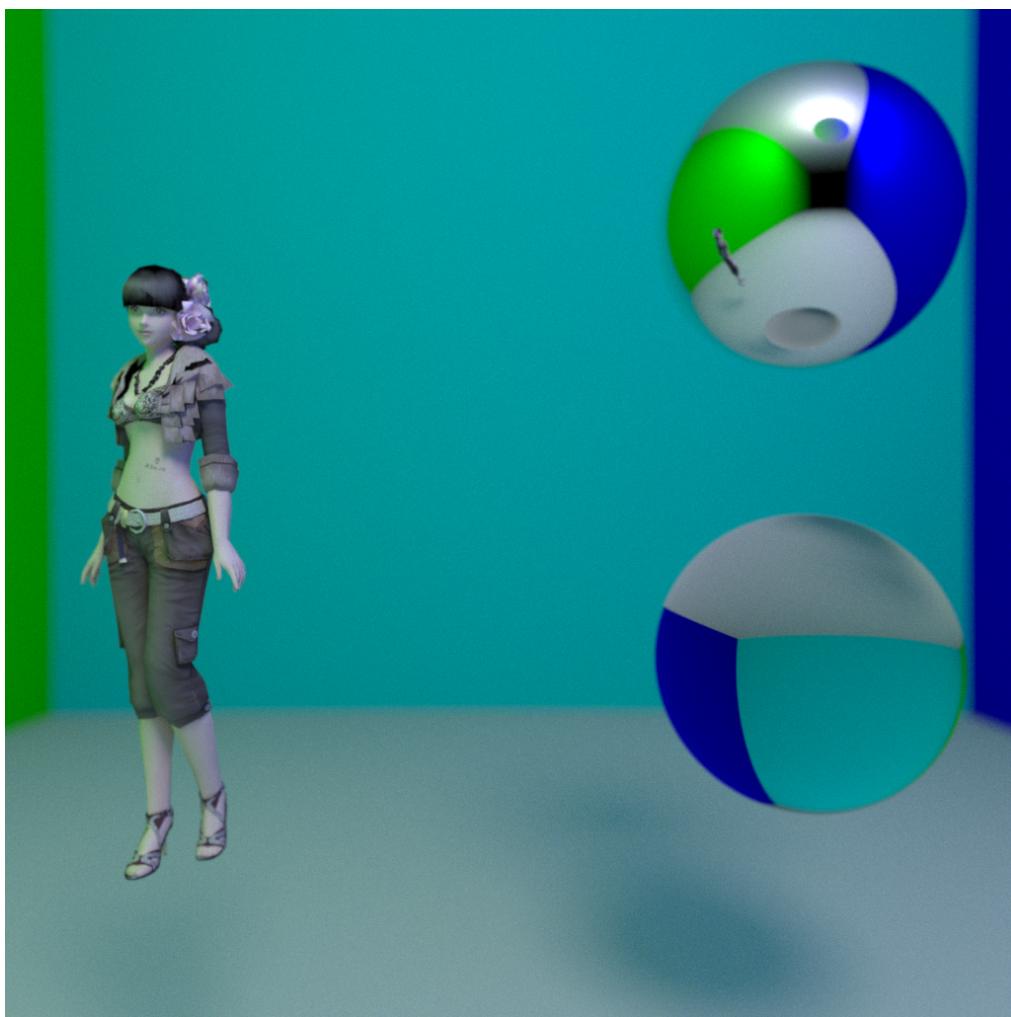


Raytracing

MOS 2-2 : Informatique graphique

LEMEILLE Pauline - 22 mars 2020



Rendu final (500 rayons - Environ 1 heure)

Introduction

Le but de ce projet est de réaliser le rendu d'une scène le plus réaliste possible en C++. Pour cela nous allons utiliser une méthode de Raytracing, c'est à dire de lancer de rayon. Nous allons en particulier travailler en backward-raytracing, nous allons donc suivre le chemin inverse de la lumière (en lançant des rayons depuis la caméra et en voyant si ces derniers interceptent un objet lumineux sur leur chemin et non en les lançant depuis la source de lumière).

L'idée est de prendre en main progressivement le langage et les méthodes, en complexifiant au fur et à mesure :

- Les types d'objet de la scène (sphère, triangle, géométrie complexe)
- Les surfaces des objets (diffus, miroir, transparent, ajout des textures)
- Le réalisme de la scène (ombre, éclairage indirect, source de lumière étendue)

Nous réaliserons aussi des corrections de nos rendus, comme la correction du facteur gamma, de l'anti-aliasing, de l'ouverture de la caméra, de Phong...

Je vais détailler ici les différentes étapes qui me permettent d'aboutir à l'image de la page de garde.

L'intégralité du code et des rendus sont disponibles à [cette adresse](#).

Séance 1

L'objectif de la séance 1 a été dans un premier temps de prendre en main le langage C++ puis de comprendre la méthodologie à mettre en place pour créer ma première scène. J'ai donc suivi à la lettre les informations fournies sur le GGdoc du cours. En particulier, j'ai commencé par mettre en place ma classe **Vecteur** qui me sera utile tout le temps, la plupart des objets manipulés seront, ou auront des attributs qui seront, des Vecteurs. J'ai également importer tout le code nécessaire à l'enregistrement des images .

Premier rendu : un cercle blanc

Le but de ce premier rendu est de placer une caméra et une sphère dans la scène. On lance ensuite un rayon lumineux qui part du centre de la caméra vers chaque pixel de notre scène. Si le rayon intercepte la sphère, alors, la couleur du pixel sera « blanc », sinon, il restera « noir ». A ce stade, ma fonction **main()** fait déjà appel une autre fonction **getColor()** ce qui permet de ne gérer que la génération du rayon dans le **main()** et de faire tous les calculs d'intersection en dehors.

Deuxième rendu : surface diffuse

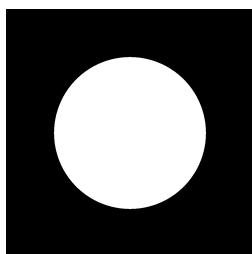
Ce premier rendu n'est pas du tout réaliste, en effet on obtient un cercle blanc et il est impossible pour nous de savoir que notre objet est en réalité une sphère. Ce manque de relief est dû à l'absence d'éclairage. Nous rajoutons donc à notre scène une source de lumière (pas au même endroit que la caméra) et nous considérons à partir de maintenant que la sphère est en matériau diffus.

Troisième rendu : plusieurs sphères

On veut maintenant pouvoir ajouter plusieurs sphères à notre scène (en particulier pour ajouter des murs, un sol, un plafond, etc...). On crée alors un objet **Scene** qui possède un vector de **Sphere**. On peut facilement accéder à chaque sphère grâce à son *sphere_id* dans ce vecteur.

Quatrième rendu : Sphères colorées

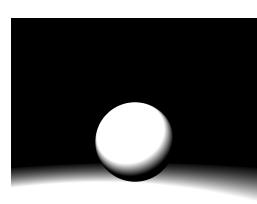
Nous voulons maintenant donner de la couleur à nos sphères. Pour cela, un nouvel attribut *albedo* est ajouté à cet objet et on inclus ce dernier dans la formule de calcul de l'intensité de chaque pixel.



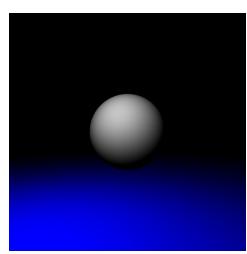
Premier rendu



Deuxième
rendu



Troisième rendu



Quatrième
rendu

Séance 2

L'objectif de la séance 2 est rendre notre scène plus réaliste, premièrement en ajoutant les ombres générées les une sur les autres par nos sphères. Le deuxième but est de rajouter de nouveaux types de surfaces, comme transparente ou miroir.

Premier rendu : les ombres portées

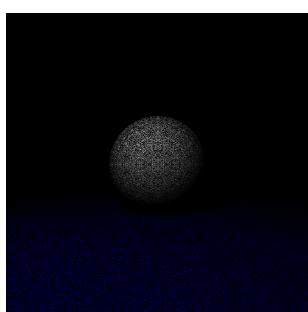
Ici, pour chaque intersection initiale trouvée, on cherche s'il n'y a pas en fait un autre objet entre notre sphère et la source de lumière. Si tel est le cas, il devrait y avoir une ombre et notre pixel est finalement noir. Une version native du code donne une image bruitée. Il faut légèrement décoller le nouveau rayon de la surface pour voir une image propre.

Deuxième rendu : les surfaces spéculaires

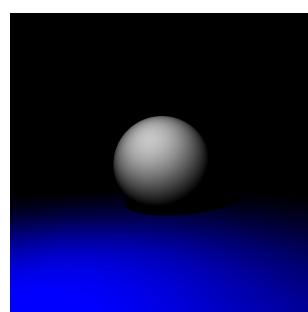
On ajoute dans un premier temps plus de sphères à notre scène afin de mieux voir le résultat (en s'inspirant de la scène fournie dans le GGdoc). Pour les surfaces spéculaires, on réfléchit chaque rayon incident par rapport à la normale et on appelle de nouveau la fonction `getColor` sur ce rayon réfléchi. On a alors une fonction récursive, il faut donc une condition d'arrêt (au cas où deux surfaces miroirs se renvoient les rayons à l'infini), on fixe donc un nombre de rebond maximum.

Troisième rendu: les surfaces transparentes

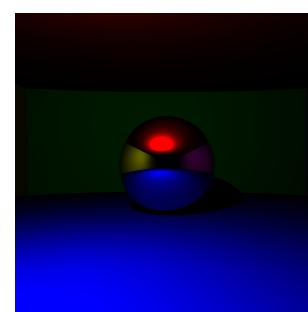
On réalise une opération similaire à l'étape précédente, mais cette fois en appliquant la formule de Fresnel pour générer le rayon réfracté. Il faut bien veiller à inverser le centre de notre normale si notre rayon sort de la sphère.



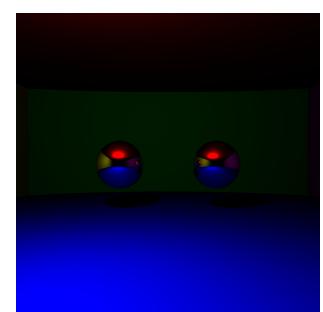
*Ombres portées
bruittées*



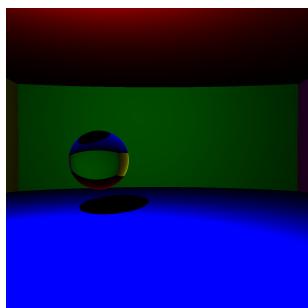
Ombres portées



Surface miroir

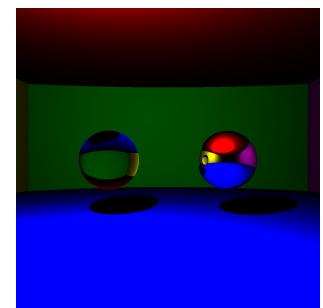


*Deux sphères
miroir*



*Surface
transparente*

*Une sphère
miroir et une
transparente*



Séance 3

L'objectif de la séance 3 est principalement de gérer les éclairages indirects. C'est-à-dire, de considérer que la lumière ne vient pas uniquement de la source de lumière mais que les objets de la scène deviennent aussi des sources de lumière.

Premier rendu : correction Gamma

On corrige notre rendu pour qu'il s'adapte au facteur gamma des écrans.

Deuxième rendu : éclairage indirect

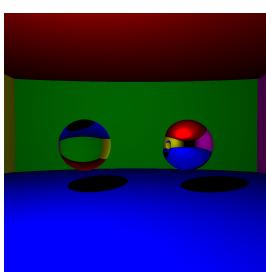
On étudie dans un premier temps la BRDF des surfaces. Ici toutes les surfaces sont diffuses donc cette dernière est donnée par $f(\vec{i}, \vec{o}) = 1/\pi$. Pour connaître la couleur finale de chaque pixel on applique l'équation du rendu. Celle-ci consiste à intégrer sur la totalité des directions sortantes de l'intégrale. En appliquant la méthode de Monte-Carlo, on peut traiter cette intégrale comme une somme. Pour générer ces directions de sortie par rapport à la normale, on définit la fonction `randomcos()` qui va nous générer un vecteur de direction aléatoire autour de la normale.

On utilise également la librairie OpenMP pour cette étape, afin que la génération de nombre aléatoire soit meilleure que celle que l'on peut obtenir avec la librairie random native.

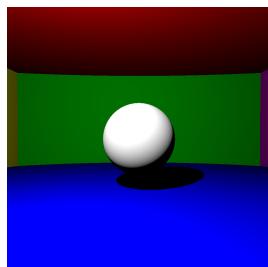
Si on ne tire qu'un rayon par pixel et un rayon par intersection, l'image est très bruitée car cette somme ne contient qu'un terme.

Finalement, pour ne pas faire exploser le nombre de calcul, on ne tire qu'un rayon à chaque intersection, mais en revanche, on tire plusieurs rayons par pixel au début. On obtient alors, en une image beaucoup plus propre et surtout réaliste.

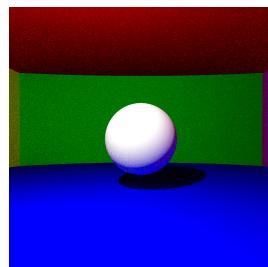
On utilise alors une nouvelle fois OpenMP pour pouvoir paralléliser les calculs des pixels sur les différents thread dont dispose notre machine. Cela accélère grandement le temps de calcul.



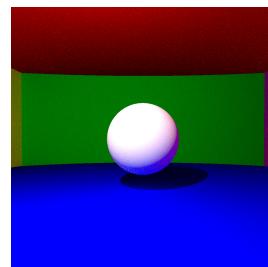
Correction gamma de l'image précédente



Scène sans éclairage indirect

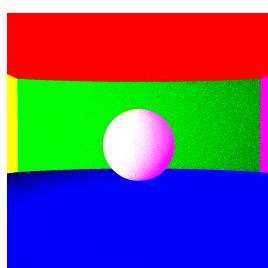


Eclairage indirect - 1 rayon

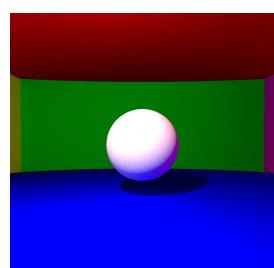


Eclairage indirect - 10 rayons

(J'ai longtemps eu des images peu réalistes comme celle-ci car je donnais aux vecteurs couleur des sphères la valeur « 255 » et non pas « 1 » à la couleur que je voulais activer.)



Eclairage indirect - 80 rayons



Séance 4

L'objectif de la séance 4 est de rendre notre rendu toujours plus réaliste. On travaille ici principalement sur la mise en place de sources de lumières étendues et plus ponctuelles. On améliore aussi la précision en rendant les frontières de nos objets plus propres. Enfin, on améliore le réalisme en donnant la possibilité de jouer sur l'ouverture du « diaphragme » de la caméra et donc sur la profondeur de champs de celle-ci.

Premier rendu : correction de l'anti-Aisling

On corrige notre rendu pour que les frontières de nos objets soient plus douces. L'anti-Aisling est dû au fait que tous les rayons de l'on envoie sur un pixel sont dirigés vers son centre. Or, il faudrait que l'on touche plus largement le pixel. On utilise donc la méthode de Box-Muller pour générer des échantillons gaussiens et décaler de ces échantillons la direction des rayons envoyés depuis la caméra.

Deuxième rendu : lumières étendues - version naïve

On cherche à représenter maintenant des sources de lumière étendues et non ponctuelles. Notre source sera donc ici un objet (une sphère), pour se simplifier la vie, on décide que ce sera toujours le premier. Ainsi, on supprime tout ce qui est lié à l'éclairage direct et on ne donne de la couleur à un pixel que si un des rayons qui passe intercepte la source lumineuse. La probabilité que cela arrive est très faible. On obtient donc une image très bruitée.

Troisième rendu : lumières étendues - version de la vidéo

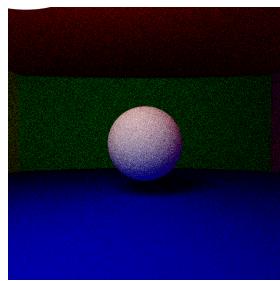
On va donc ici redéfinir notre éclairage direct. Le but est d'envoyer des rayons en direction de la sphère qui émet de la lumière et ainsi avoir le moins de bruit possible. Par un changement de variable dans les intégrales, il est équivalent d'intégrer sur toutes les directions de sorties que d'intégrer sur tous les points des objets. On choisit donc la deuxième option et on génère un point aléatoire sur la sphère de lumière (le plus proche possible de l'axe entre le centre de la sphère de lumière et le point d'intersection P). On vérifie qu'il n'y a pas d'intersection avec un autre objet entre les deux et si ce n'est pas le cas on applique l'équation du rendu (avec les changements de variable).

Quatrième rendu : Distance focale et Ouverture de camera

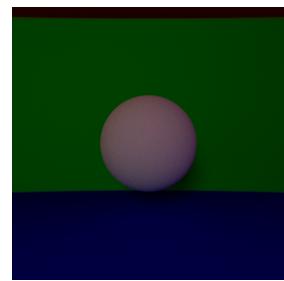
On cherche maintenant à modéliser plus proprement le comportement de la caméra, en particulier le fait qu'un objet sera net ou flou suivant sa position par rapport au plan focal image. Pour faire en sorte que les objets en dehors du plan focal soient tout de même relativement net, on modélise une « ouverture » (carrée ici) plus ou moins grande du diaphragme de la caméra et on décale l'origine du rayon envoyé au pixel au sein de ce carré. Cela permet d'avoir un rendu plus réaliste.



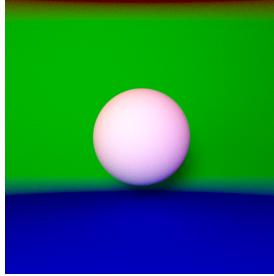
Correction de l'anti-Aisling



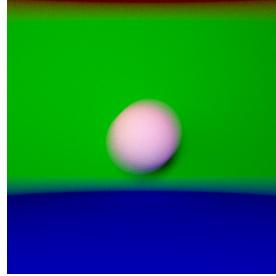
*Lumière étendue
Version naïve
(on voit la source de
lumière en haut à gauche)*



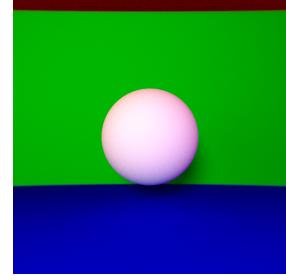
*Lumière étendue
80 rayons*



*Lumière étendue
Sphère dans le plan
(grosse ouverture)
80 rayons*



*Lumière étendue
Sphère hors plan
(grosse ouverture)
80 rayons*



*Lumière étendue
Sphère dans le plan
(petite ouverture)
120 rayons*

Séance 5

L'objectif de la séance 5 est de pouvoir intégrer de nouveaux types d'objets à notre scène. En particulier, on souhaite ajouter des objets quelconques représentés grâce à un maillage triangulaire. On cherche donc dans un premier temps à travailler sur un simple objet **Triangle** puis on élargit à une **Geometry** qui est un ensemble de triangle.

Premier rendu : un triangle

On trace ici un triangle seul pour vérifier que notre définition de la classe et en particulier de la fonction interception est correcte. Pour pouvoir gérer nos triangles comme des objets de la scène au même titre qu'une sphère, on crée une nouvelle classe **Objet** dont héritent les classes **Triangle** et **Sphere**. Les éléments de notre scène seront, à partir de maintenant représentés par un vecteur d'objets et non plus par un vecteur de sphères. On écrit ensuite la routine d'intersection des triangles. Pour savoir s'il y a intersection entre un rayon et un triangle on vérifie d'abord s'il y a intersection entre le plan (défini par le triangle) et le rayon, puis regarde si le point d'intersection appartient au triangle en résolvant un système de Cramer. Ce dernier nous donne les coordonnées barycentriques du point et on sait ainsi s'il appartient ou non au triangle.

Deuxième rendu : Beautiful Girl avec boîte englobante

On veut maintenant ajouter un objet quelconque à notre scène (.obj réalisé par des artistes). On crée donc la classe **Geometry** qui hérite elle aussi de la classe **Objet**. En plus d'une routine d'intersection, une géométrie dispose aussi d'une boîte englobante (**Bbox**). Cette boîte représente la boîte la plus petite dans laquelle on peut placer notre géométrie. Ainsi, si on est hors de cette boîte, la routine d'intersection ne fait rien. En revanche, si on est dans cette boîte on appelle chaque élément de notre maillage (qui est un triangle) et on regarde s'il y a intersection avec ce dit triangle.

NB: Pour lire l'objet en .obj on utilise une fonction qui nous est fournie et qui crée les différents vector dont nous avons besoin pour travailler ensuite sur les géométries.

Troisième rendu : Structure récursive de boîtes englobantes

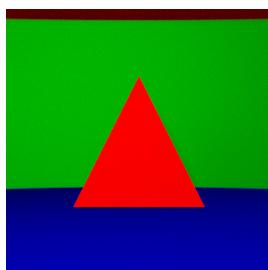
On peut facilement imaginer que, pour optimiser notre temps de calcul très long, chaque objet soit en réalité un ensemble de sous objets, et que chacun d'eux possède donc une boîte englobante. Ainsi, on traiterait beaucoup moins de test d'intersection pour chaque rayon lancé (boîtes de taille beaucoup plus petite que la boîte englobante tout l'objet). Nous définissons donc ici cette représentation récursive des boîtes englobantes : les **BVH**. Une **BVH** est un arbre de **BVH**, chacune d'elle possède une **Bbox**, deux indices de début et de fin de cette boîte englobante, un fils gauche et un fils droit. On construit récursivement, chaque **BVH** en regardant la dimension la plus grande de sa **Bbox**, en trouvant son milieu et en la coupant en deux. On applique ensuite un Quick Sort sur le vecteur *indices* pour

que tous les triangles appartenant à la première sous **Bbox** soient dans l'arbre fils gauche, et tout ceux appartenant à la deuxième dans l'arbre fils droit.

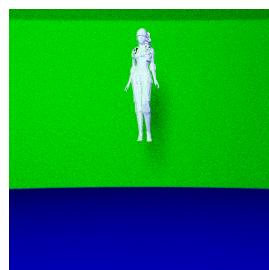
Une fois la **BVH** de la géométrie construite, on modifie la routine d'intersection afin de ne tester les intersections rayon-triangles de notre maillage que sur la **Bbox** d'une feuille de notre **BVH** (si celle-ci existe) ayant une intersection avec le rayon. On gagne énormément en temps de calcul et on peut facilement se permettre d'envoyer 10 fois plus de rayons par pixel.

Quatrième rendu : Correction de Phong

Nous travaillons ici sur une géométrie au maillage triangulaire. On observe donc un effet boule à facettes dû au fait que lorsque l'on trouve une intersection avec notre géométrie, on considère la normale à P comme étant la normale au triangle du maillage qui donne l'intersection. Afin d'avoir un rendu plus réaliste, l'artiste a fourni les « normales » aux sommets de chaque triangle. On revoit donc, à la place de la normale classique, la « moyenne barycentrique » des normales aux sommets.



Triangle



*Beautiful Girl
Mal placée sans
BVH
(très long si beaucoup de
rayons)*



*Beautiful Girl avec
BVH
(Pas de différence au
rendu mais beaucoup
plus rapide)*



*Beautiful Girl avec
BVH et Phong
(Plus de lumière
- 80 rayons)*

Séance 6

L'objectif de la séance 6 est de rajouter les textures à nos objets.

Premier rendu : Beautiful Girl avec ses textures

Maintenant que notre géométrie et son maillage sont maitrisés, on cherche à ajouter des textures à notre géométrie, c'est à dire que chaque face (triangle) de celle-ci puisse avoir une couleur différente.

Dans un premier temps, on utilise le code fourni pour « lire » le fichier de textures fourni par l'artiste et rendre ses données utilisables dans un vecteur *textures*. Pour les textures, on travaillera avec les coordonnées uvs, elles aussi accessibles grâce aux fichiers fournis.

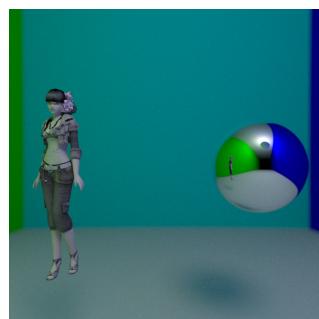
L'idée est qu'à chaque intersection trouvée sur un maillage d'une géométrie, on aille chercher dans *textures* la couleur correspondant au triangle maillage. (Pour ne pas coder pour chaque triangle alors que plusieurs d'entre eux peuvent avoir la même texture, on crée des groupes de faces, et on a une seule texture par groupe de face.)

On modifie donc les routines **intersection** pour qu'elles renvoient la couleur de l'objet intercepté pour tous les objets sauf les géométries. Dans le cas des géométries, on interpole les coordonnées uvs avec une moyenne barycentrique puis on récupère les couleurs correspondant à ces coordonnées dans textures.

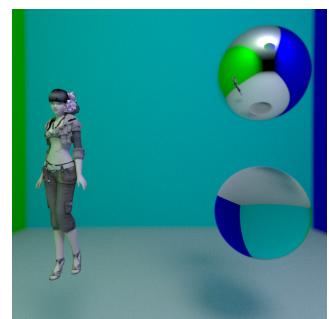
On obtient alors, une couleur pour chaque triangle du maillage, ce qui nous donne un rendu beaucoup plus réaliste.



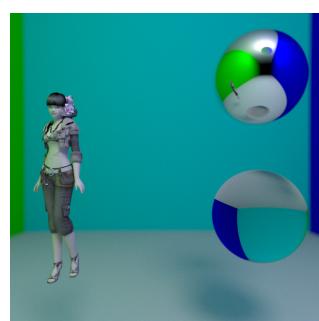
*Beautiful Girl avec
textures
(Plus de lumière
-
120 rayons)*



*Beautiful Girl avec
sphère miroir
(130 rayons)*



*Beautiful Girl avec
sphères miroir &
transparente
(200 rayons)*



*Rendu final
(500 rayons
-
Environ 1 heure)*

Conclusion sur le cours

Comme demandé, je vous fait ici un compte rendu sur le cours.

Tout d'abord, je pense important de signaler que je n'avais jamais fait de simulation de rendu ni de C++ avant de commencer ce module. J'ai donc plutôt eu des difficultés à suivre ce cours. En effet, surtout au début, je perdais beaucoup de temps sur des erreurs de syntaxe et sur la manière de gérer les objets, les pointeurs, etc... Il n'était pas possible pour moi de réaliser pendant la séance l'intégralité des features demandées et je devais reprendre le projet chaque semaine chez moi pendant plusieurs heures. Pour cela, les vidéos en ligne m'ont beaucoup aidé et permis de comprendre des notions que je n'avais pas comprises ou eu le temps de mettre en place en cours.

Malgré ces difficultés, en partie dues à mon manque d'expérience sur le sujet, je suis très contente d'avoir pu suivre ce cours. Premièrement car cela m'a permis de découvrir l'informatique graphique et que j'ai trouvé très intéressant de mêler optique/mathématiques et informatique. De plus, cela m'a permis de découvrir le C++, je suis maintenant beaucoup plus à l'aise avec ce langage.

Enfin, le fait de travailler sur le même projet de bout en bout est très stimulant et satisfaisant lorsque l'on obtient enfin le résultat voulu.

Pour conclure je dirais qu'il me semble judicieux de mentionner dans le descriptif du cours que la connaissance de C++ est quasiment nécessaire pour s'y inscrire. Il est également possible de découvrir le C++ en même temps que le cours mais cela demande un très gros investissement à la maison, en particulier car il n'est pas possible (ou très difficile) de suivre le cours au rythme imposé par les séances.