Laffite Axel

21606855

TPA11

Rapport IGTAI

Université Paul Sabatier

Sommaire

Introduction

Changements

Appel du programme

Quelques petits changement au niveau de l'appel du programme. En effet on peut maintenant choisir des options telles que la largeur, la hauteur, le taux d'antialiasing etc.

L'appel se fait comme avant pour la configuration basique. Pour changer les paramètres en revanche, on utilise la syntaxe suivante :

./mrt filename scenenumber [options]

Rentrer un nombre incorrect de paramètres (< 2) ou bien ajouter « help » nous redirige vers le message suivant :

```
usage : ./mrt filename scenenumber [options]
 - filename         : where to save the result, without extention
 - scenenumber      : scene number, optional (default 0)

Available options are:
 - width=integer    : size of the resulting image width in pixels   (default 800)
 - height=integer   : size of the resulting image height in pixels  (default 600)
 - aa=integer       : amount of AA applied to the image             (default 4)
 - aamode=string    : optimisation applied to the AA                (default sobel)
                      should be one of 'none' | 'luma' | 'sobel'
 - focaldist=float  : focal distance from the camera                (defined by the scene)
 - focalrange=float : focal range from the focal distance           (defined by the scene)
```

Celui-ci décrit les paramètres disponibles, ce à quoi ils correspondent et quelles sont les valeurs par défaut.

Il est tout à fait possible d'appeler le programme comme avant, les options étant facultatives.

Affichage lancement du programme

Du code à aussi été déplacé, les fonctions d'intersections notamment, qui sont maintenant dans intersections.h pour plus de facilité d'intégration avec le KDTree (on évite de recopier du code inutilement). La fonction intersectScene a elle aussi été déplacée bien qu'elle aurait pu rester dans le fichier raytracer.cpp.

Une ligne apparaît en plus à la fin du rendu, il s'agit du temps total de rendu qui se présentera sous cette forme :

```
Render time: 29.049857s
```

Dû au type d'implémentation utilisé et la façon dont ont étés parallélisées certaines fonctions, il n'a pas été possible d'afficher une barre de progression pour les deux optimisations de l'antialiasing qui vous seront présentés dans ce rapport.

Un message s'affiche donc lors du début de son calcul, il en est de même pour le *depth of field*.

```
Computing Anti aliasing..
Computing depth of field..
```

Pour finir les changement sur l'affichage du programme, un message comme celui-ci est affiché en début de chaque rendu pour spécifier les options dans la scène courante.

```
-----------------------------------------
Parameters of the scene :
Width      : 800
Height     : 600
AA         : 4
AA mode    : luma
F distance : 2.000000
F range    : 2.000000
-----------------------------------------
```

Temps de travail

Le temps de travail a été d'environ 40 heures.

La partie sur 16 points a duré environ 15h, les bonus 25h.

Cette différence est surtout due au temps de recherche pour les différents algorithmes à implémenter.

Le code complet avec tous les rendus effectués en plus est disponible à cette adresse :

https://github.com/ElZozor/IGTAI-Raytracer

Il ne sera bien sûr rendu public qu'à partir du moment où le projet sera terminé.

1.1 Calcul d’intersection

Difficultés

Pas de difficultés particulières, cependant il fallait repérer le fait de changer la valeur tmax du rayon en cas d’intersection avec un objet. Cette information n’était pas très visible car « cachée » dans l’introduction du document.

1.2 Couleur et shading

Difficultés

Pas de difficultés particulières.

1.3 Ombres

Difficultés

J’ai mis du temps à comprendre comment calculer l’intersection uniquement entre P et L. Cependant le sujet était assez clair, il s’agit juste de lacunes personnelles. Je pensais aussi que le champ ray→invdir correspondait à –ray→dir …

2.1 Distribution de Beckmann et terme de Fresnel

2.1.1 Distribution de Beckmann

Difficultés

Pas de difficultés particulières.

2.1.2 Terme de Fresnel

Difficultés

Pas de difficultés particulières.

2.1.3 Atténuation

Difficultés

J’ai eu du mal à comprendre le x = l ou v. Je n’avais pas bien compris comment les appliquer.

2.2 Implémentation de la BSDF

Difficultés

J’ai eu du mal à comprendre comment associer les fonctions entre elles, du moins où devait aller certains bouts de code.

De plus, j’avais mal calculé le *half-vector*.

2.3 Réflexions

Difficultés

Valeurs parfois trop haute retournées par la fonction shade. J’ai mis du temps à trouver le problème et ai finalement implémenté une solution qui ne me semble pas être la meilleure :

```
return vec3(min(1.f, ret.x), min(1.f, ret.y), min(1.f, ret.z));
```

La valeur *acne_eps* était trop grande et certains artefacts apparaissaient malgré tout, j’ai dû mettre comme tmin $1e^{-3}$ pour ne plus les avoir.

3.1 Antialiasing

Méthode

Je suis parti sur la méthode « naïve » autrement dit, le *superslamping* qui consiste à subdiviser des les pixels en *n* sous-pixels puis à faire la moyenne de toutes les couleurs obtenues.

Ce paramètre est variable via l’option « aa=X » où X est le taux d’antialiasing à appliquer, ajouter ensuite « aamode=none » pour spécifier qu’on ne veut pas d’optimisations.

Difficultés

Pas de difficultés particulières, l’algorithme est assez simple en soit.

Améliorations

Détection de contours : Contraste

Technique

La technique reste toujours du superslamping mais au lieu d’appliquer le traitement sur tous les pixels de l’image, le but ici est de détecter les bords des objets.

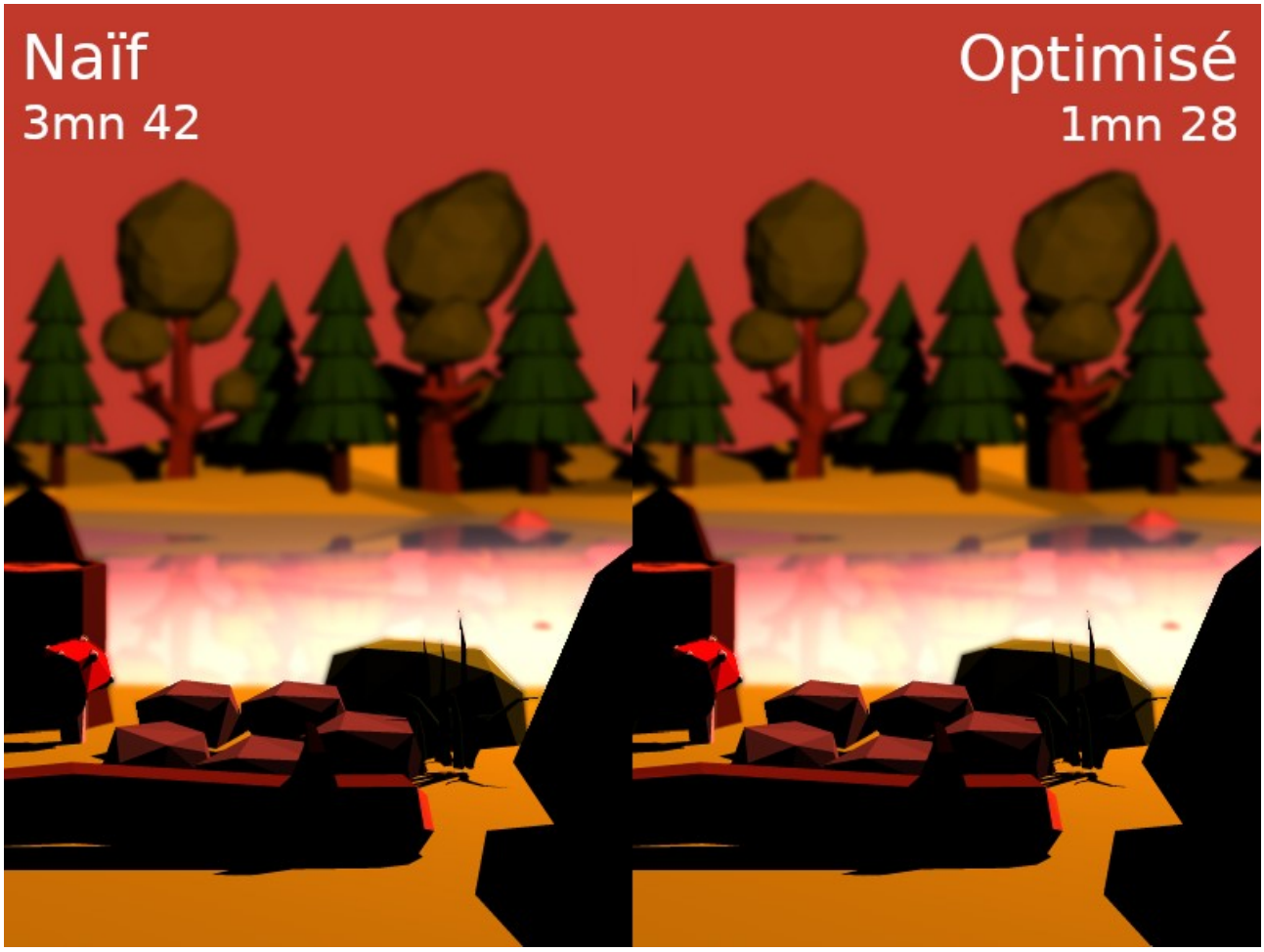
La technique pour la détection de bords est proche de celle utilisée par les appareils photo pour l'auto-focus. On va ici se concentrer sur le contraste de l'image.

Si le contraste calculé entre le pixel courant et les 8 pixels l'avoisinant dépasse un certain seuil, on considère alors qu'on est sur le bord d'un objet.

Cette technique permet donc d'éviter d'effectuer le traitement sur des pixels où ce ne serait pas vraiment visible.

Il y a cependant un défaut qui est que certains bords ne seront pas détectés, on aura donc des endroits non lissés alors qu'ils devraient l'être. La technique peut cependant être améliorée via les normales et un « depth buffer ».

Comparatif



Les gains eux sont particulièrement intéressants !

	Scène 0	Scène 1	Scène 2	Scène 3	Scène 4
Naïf	0.678209s	1.812674s	2.035722s	2.937060s	18.384353s
Optimisé	0.145843s	0.984185s	0.954235s	0.493648s	11.664189s

Temps de rendu en 800x600, depth of field activé, AAx4

La détection de bords étant cependant parfois peu précise, nous allons nous concentrer sur un autre algorithme de détection de bord : Le « sobel opertor ».

**Pour effectuer un rendu avec la détection de contraste, mettez en argument « aamode=luma » lors de l'appel du programme.*

Détection de contours : Sobel

Technique

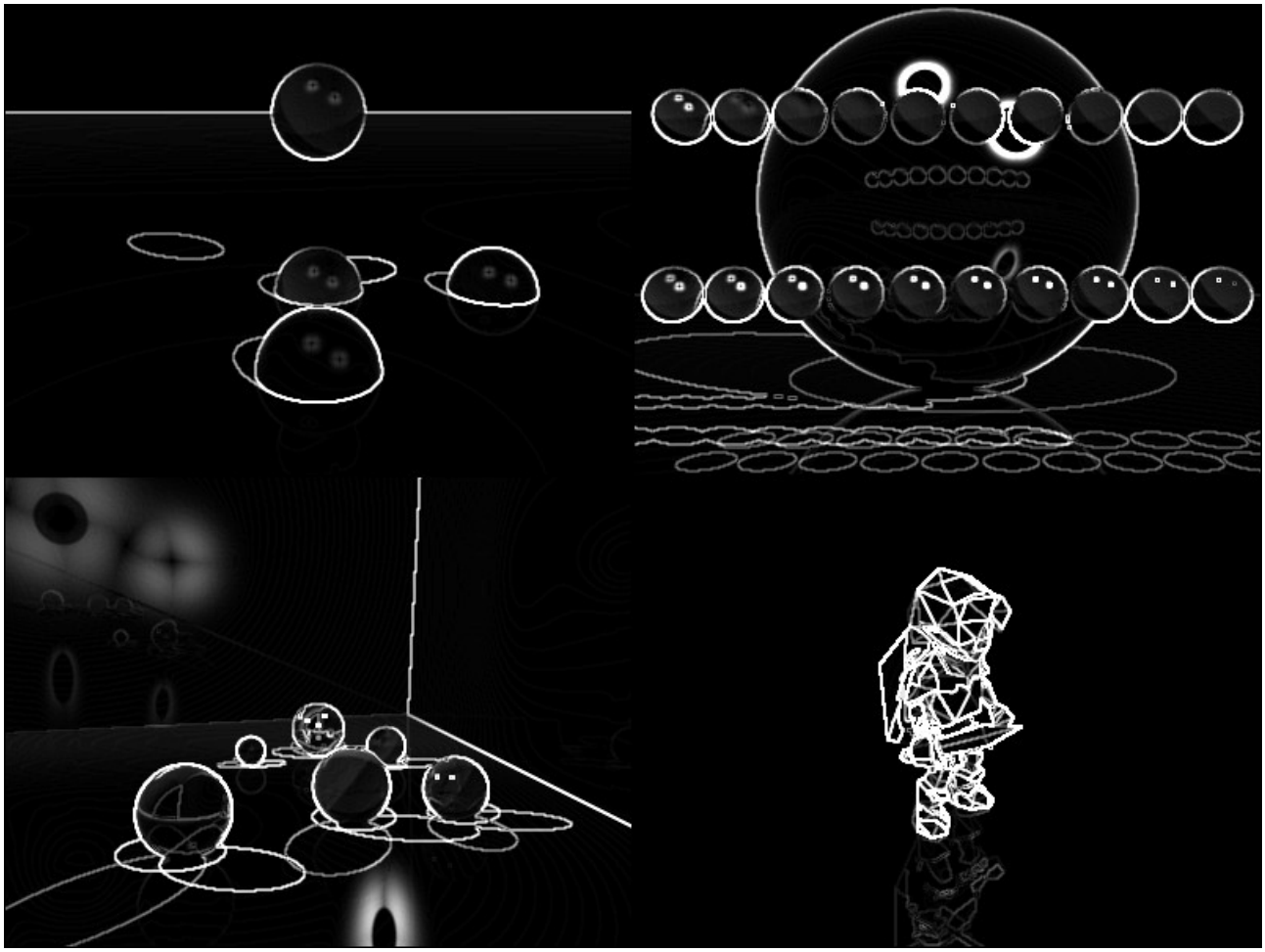
Un algorithme de détection de contours largement répandu est l'opérateur de Sobel.

Il est basé sur deux matrices, qui nous permettent d'effectuer la détection des contours en 2 passes. Une pour l'axe X et une pour l'axe Y (également applicable dans différentes dimensions, notamment en 3D).

Pour effectuer ma détection, je calcule le rendu de base sans antialiasing. Je parcours ensuite mon image et applique mon opérateur de Sobel sur les différents pixels de l'image qui auront étés préalablement convertis en niveau de gris via la formule suivante :

$$greyscale(R,G,B) = (R * 0.07 + G * 0.72 + B * 0.21)$$

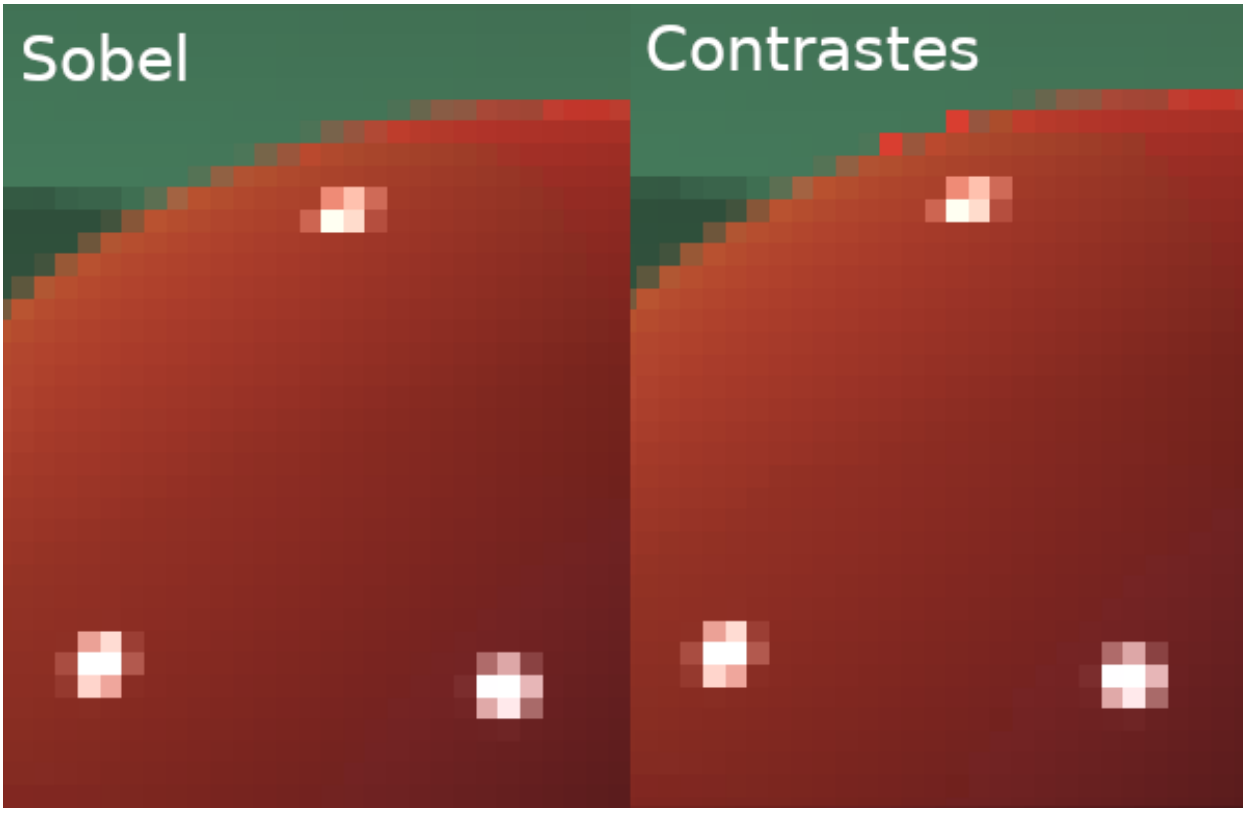
Visualisation des contours



Visualisation des contours avec l'opérateur de Sobel

On peut ici voir que l'opérateur de Sobel s'adapte bien à la plupart des scène et que même sur des contours peut tracés (réflexions par exemple) on obtient un détection assez fidèle pour un coût très faible en temps d'exécution.

Comparatif



Comparaison entre la technique avec la détection via l'opérateur de Sobel (à gauche) et via contrastes (à droite)

On peut voir que certains pixels qui n'étaient pas bien détectés par la technique avec les contrastes le sont via l'opérateur de Sobel. Le temps de rendu est cependant plus long.

	Scène 0	Scène 1	Scène 2	Scène 3	Scène 4
Contrastes	0.145843s	0.984185s	0.954235s	0.493648s	11.664189s
Sobel	0.155728s	0.608734s	0.964977s	1.225555s	9.987367s

Temps de rendu en 800x600, depth of field désactivé, AAx4

On peut constater que bien que certains temps de rendus soient plus court, d'autres sont en revanche plus long.

Ceci s'explique par le fait que selon l'algorithme, on détecte plus ou moins de bords, la rapidité des algorithmes de détection est cependant assez similaire, en effet on n'effectue qu'une seule passe pour chacun d'eux.

J’aurais quand même une préférence pour l’algorithme de Sobel qui en plus de mieux détecter les bords, est aussi plus rapide et facile à mettre en place.

**Pour effectuer un rendu avec l’opérateur de Sobel, mettez en argument « aamode=sobel » lors de l’appel du programme (bien que ce soit le paramètre par défaut).*

3.2 KD-Tree

Méthode

Après m’être renseigné sur quelques documents j’en ai conclu le fonctionnement suivant pour les Kd-Tree :

- On calcule les limites de la scène que l’on veut mettre dans le KD-Tree
- On initialise le KD-Tree avec ces limites
- Pour ajouter un objet au KD-Tree, on effectue de manière récursive les étapes suivantes :
  - Si le nœud courant est une feuille, on ajoute l’objet, si le buffer d’objet à atteint la taille maximale, on *splitte* le nœud courant sur un seul axe. On crée deux fils suivant cette séparation et on ajoute les objets dans les fils.
  - Si le nœud courant n’est pas une feuille, on parcours récursivement jusqu’à une feuille et on effectue l’étape précédemment décrite.
- Si il s’avère que l’on a atteint le nombre maximum en profondeur dans l’arbre, on ne crée plus de feuilles lorsque le buffer est considéré plein, on continue à mettre les objets dans ce buffer.

Mon implémentation est la suivante :

- Je coupe selon la profondeur du nœud (profondeur % 3 où x=0, y=1 et z=3).
- Pour effectuer ma coupe, je trie les objets dans l’axe choisit pour pouvoir sélectionner la médiane, je sépare donc équitablement mes objets.
- La récursion maximum est de 256 (donc 512 feuilles).
- La taille de mes buffers est de 32 objets.

J’ai décidé d’implémenter cette classe en C++ en utilisant les fonctionnalités comme les fonctions dans les structures, les lambdas, vecteurs, algorithmes de tri etc.

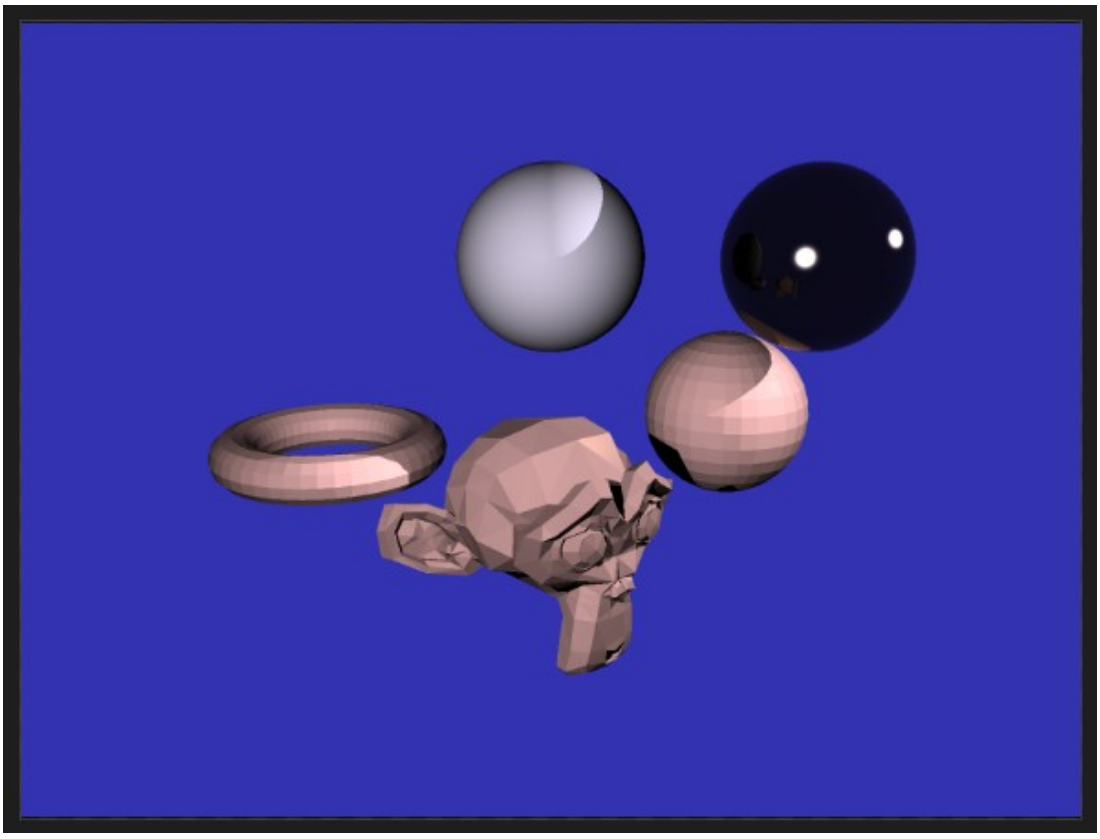
Gains

Les gains ne sont pas négligeables. Avec un i5-8265U mes temps sont les suivants :

	Scène 0	Scène 1	Scène 2	Scène 3	Scène 4
Classique	0.581979s	1.680773s	2.430130s	2.818646s	63.502931s
KD-Tree	0.678209s	1.812674s	2.035722s	2.937060s	18.384353s

*Temps de rendu en 800*600, depth of field activé et AAx2 (naïf)*

On dénote que sur les rendus avec peu d'objets le temps de rendu n'est pas plus rapide, voire plus lent (du fait qu'il faille construire l'arbre), mais dés que l'on passe sur un rendu avec un nombre conséquent d'objets, la différence est clairement notable, avec un *speedup* de 3.5 pour la scène 4 !



Pour le rendu affiché, contenant 3080 triangles, avec la méthode classique, il faut 332.508884s pour effectuer le rendu. En revanche avec le KD-Tree, il faut « seulement » 174.923733s ! Une accélération donc non-négligeable qui permet d'optimiser grandement le temps de rendu.

A noter qu'avec l'opérateur de sobel pour l'anti-aliasing on tombe à 76.759273s, avec luma 29.049857s.

3.3 Autres objets

Triangles

C’était forcément l’objet à implémenter, il est indispensable dans le monde de la 3D et pour charger des objets personnalisés il est obligatoire.

Son implémentation dans le KDTree a été simplifié, en effet, au lieu de réaliser une vraie intersection entre un triangle en 3 dimensions et un AABB, j’ai simplement calculé la « bounding box » du triangle pour effectuer son intersection avec celle du KDTree.

On se retrouve donc ici avec un simple calcul d’intersection entre deux AABB. Cependant cette technique bien que très simple à mettre en place présente des défauts puisqu’en effet on approxime tellement que certains triangles vont s’insérer dans des nœuds du KDTree alors qu’ils ne devraient pas y être.

Cependant pour les scènes que nous calculons actuellement, j’ai quand même constaté un gain conséquent, je n’ai donc pas cherché à trouver une formule plus précise.

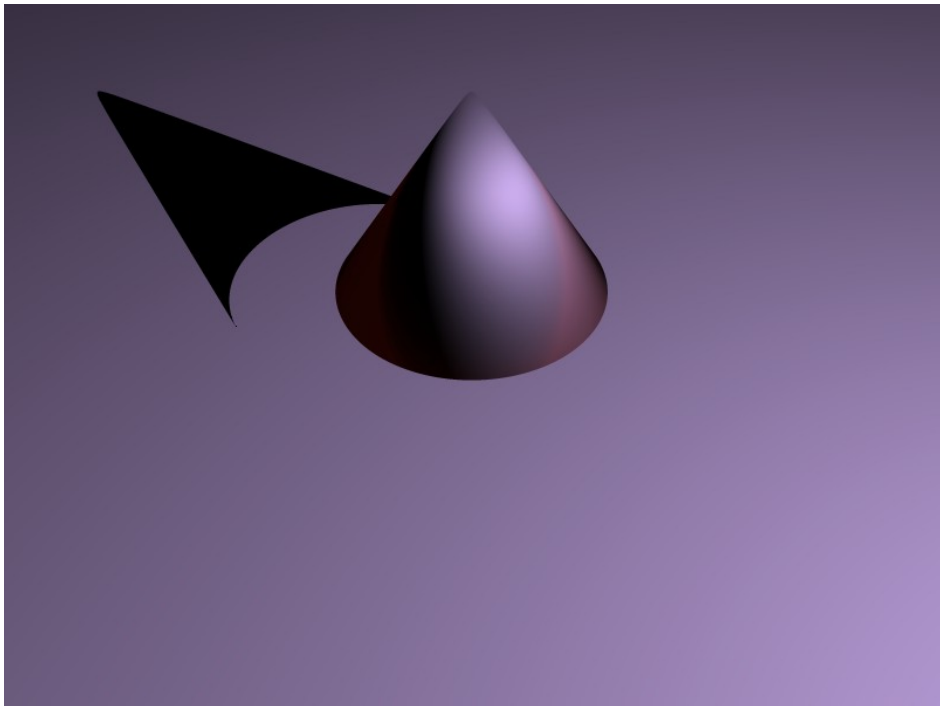
Vous pouvez observer des rendus effectués avec des triangles à la fin de ce rapport.

Cylindre

Non implémenté

Cône

Implémenté mais des erreurs subsistent. Par exemple, le calcul avec l’intersection du cercle en dessous du cône n’est pas effectuée, on ne le voit donc pas lors du rendu. Il n’est pas non plus rendu avec les ombres. Le résultat est le suivant :



3.4 Textures

Les textures ont été implémentées pour les triangles uniquement. Je me suis basé sur les coordonnées barycentriques.

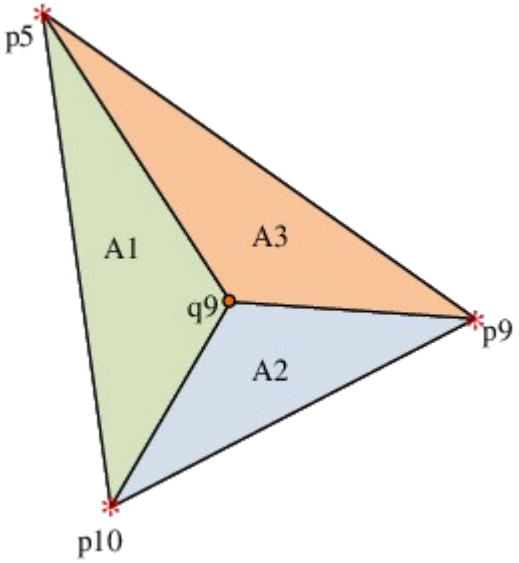
Celles – ci sont calculées par la fonction *intersectTriangle* qui se trouve dans le fichier intersections.h.

Dans Blender, lorsque l'on *wrappe* une texture à un objet, les normales de textures lui sont associées. Lorsque l'on exporte le modèle en format .obj, on se retrouve alors avec les normales de textures inclues à ce fichier. Ces dernières nous permettront alors de *mapper* la texture sur les différents triangles.

L'interpolation est effectuée avec la formule suivante :

```
vec3 uv = (t0*w + t1*u + t2*v)/(u + v + w);
```

où t[0..2] sont les normales des textures et u,v,w les poids des triangles que l'on peut voir sur l'image ci-contre.



On trouve ensuite les coordonnées sur l'image avec la formule suivante :

```
const auto uvmod = clamp(uv, 0.0f, 1.0f);
const auto y = (int((height-1) * uvmod.y+0.5f)) % height;
const auto x = (int((width-1)  * uvmod.x+0.5f)) % width;

color3 result =  image[y * width + x] ;
```

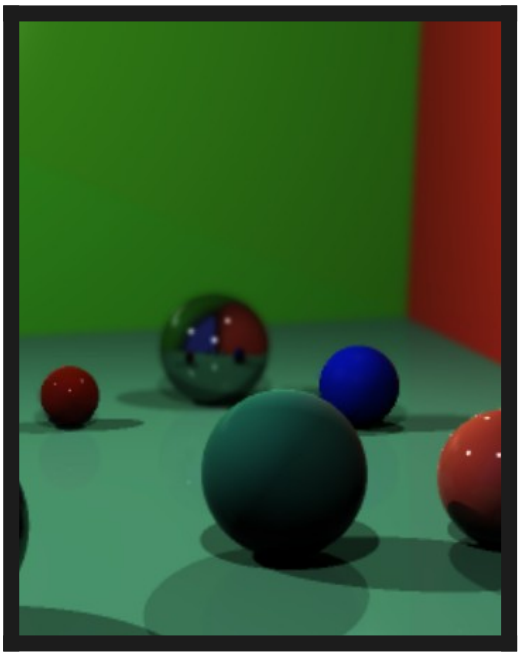
Cette formule permet d'obtenir les coordonnées de l'image par rapport aux valeur *u* et *v* trouvées précédemment. Les valeurs *x* et *y* sont calculées respectivement modulo *width* et *height* pour répéter la texture en cas de dépassement de la taille de l'image.

On peut observer le résultat de ces calculs sur l’image suivante :

Modèle réalisé avec blender.

3.5 Faites nous rêver

Profondeur de champ



Je me suis basé sur un algorithme qui construit une « height map » pour pouvoir déterminer la profondeur de champ (voir section « quelques rendus »).

Je passe ensuite un flou gaussien sur les pixels en fonction de leur éloignement par rapport au point focal.

On peut observer sur l'image ci-contre, que la sphère verte est bien plus nette que la sphère en fond qui elle est floutée par le flou gaussien.

Cette technique n'est pas parfaite et des aspérités peuvent apparaître.

J'ai aussi rajouté un champ pour pouvoir régler la zone de netteté. Cette option peut être observée dans la section « Quelques rendus » sur l'image présentant un paysage.

Ici, la valeur est celle par défaut pour cette scène.
Pour calculer la matrice pour le flou gaussien, j'ai utilisé une approximation grâce au triangle de pascal.
La formule que j'ai utilisé est la suivante :

$$poid = (pascal(m)[x] * pascal(m)[y])$$

Où m est la taille de la matrice souhaité et x et y les décalages par rapport au pixel d'origine, pascal(m) représente donc la ligne m du triangle de pascal.

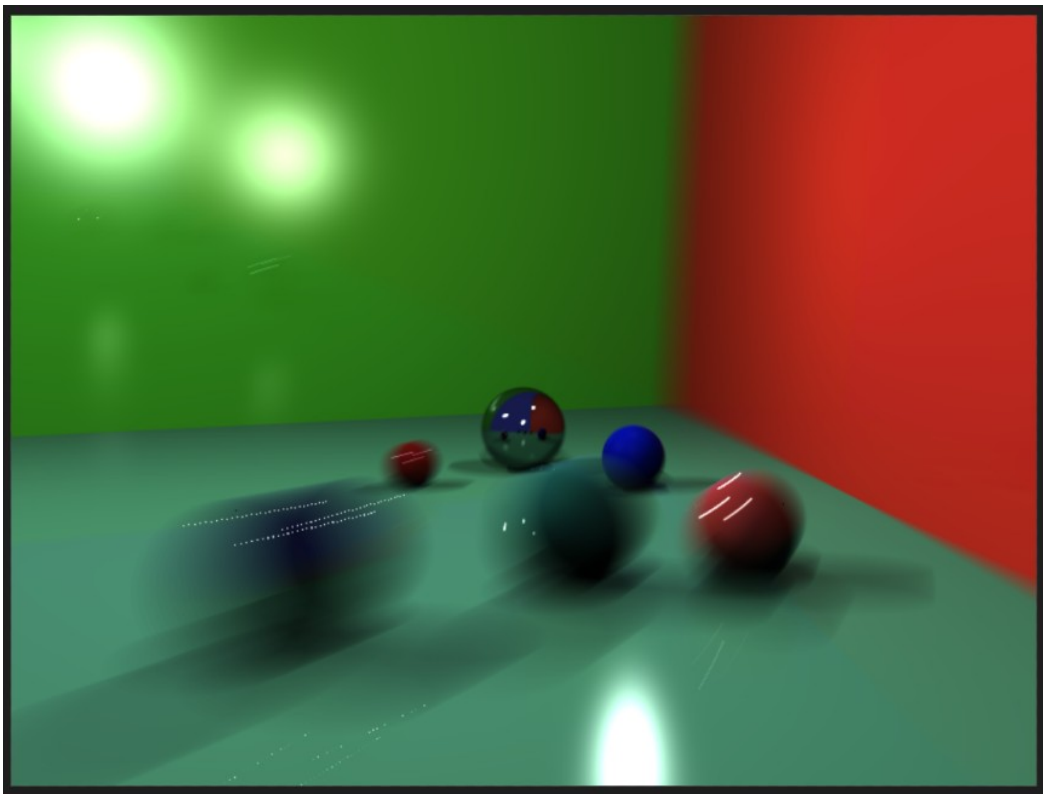
On additionne les pixels environnant en les multipliant par les poids et on divise par la somme des poids.

**Pour régler la profondeur de champs appelez le programme avec les options suivantes :*

« focaldist=X » et « focalrange=Y » où X et Y sont les distances où le focus se fera. Par exemple avec focaldist=3 et focalrange=2, le focus sera net pour les points à une distance comprise entre 1 et 5.

Motion blur

La technique utilisé ici est tout simplement de faire n rendus avec une translation de la caméra (ou du décors) et de faire la moyenne de ces rendus, le résultat est le suivant :



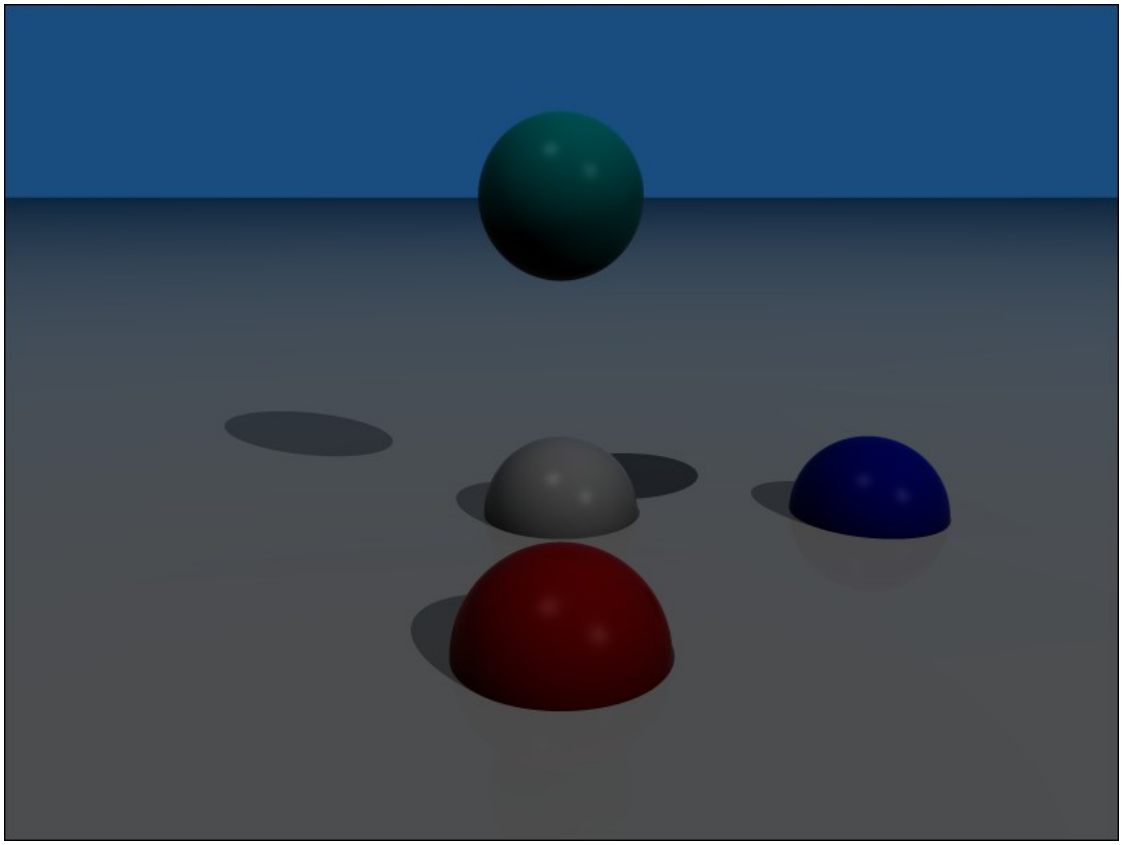
On peut voir quelques problèmes notamment sur les points lumineux qui sont nets (sphère bleu) alors qu'ils devraient être flous.

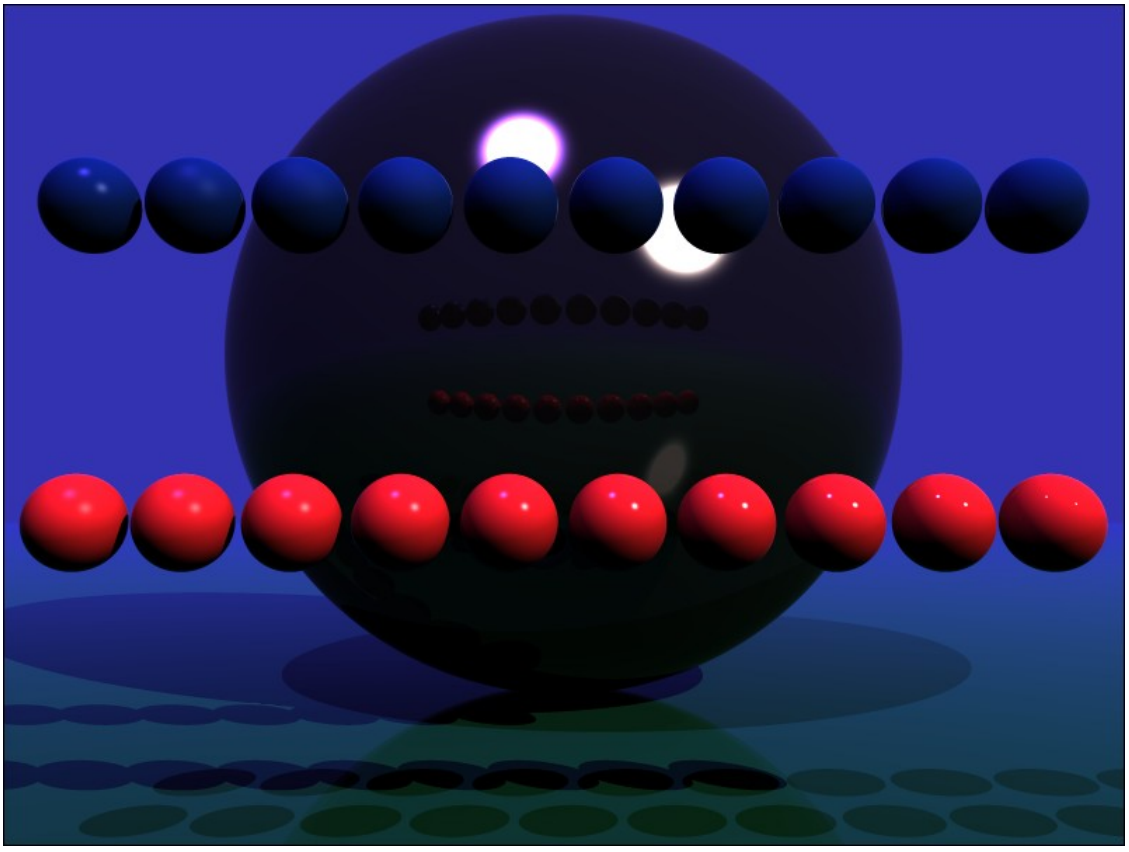
Chargeur de fichiers OBJ

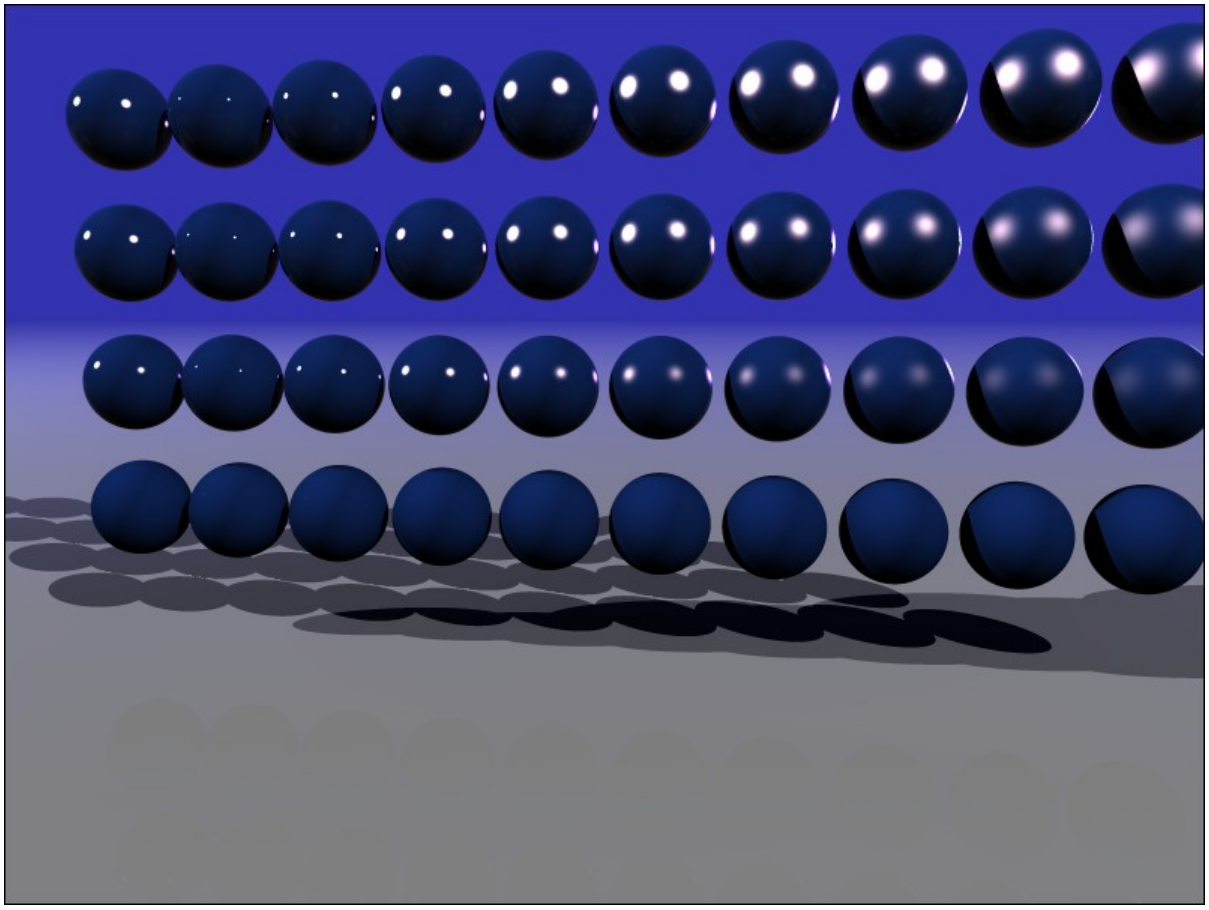
Même s'il est basique, il permet de charger des fichiers générés depuis blender.
Les contraintes sont les suivantes :

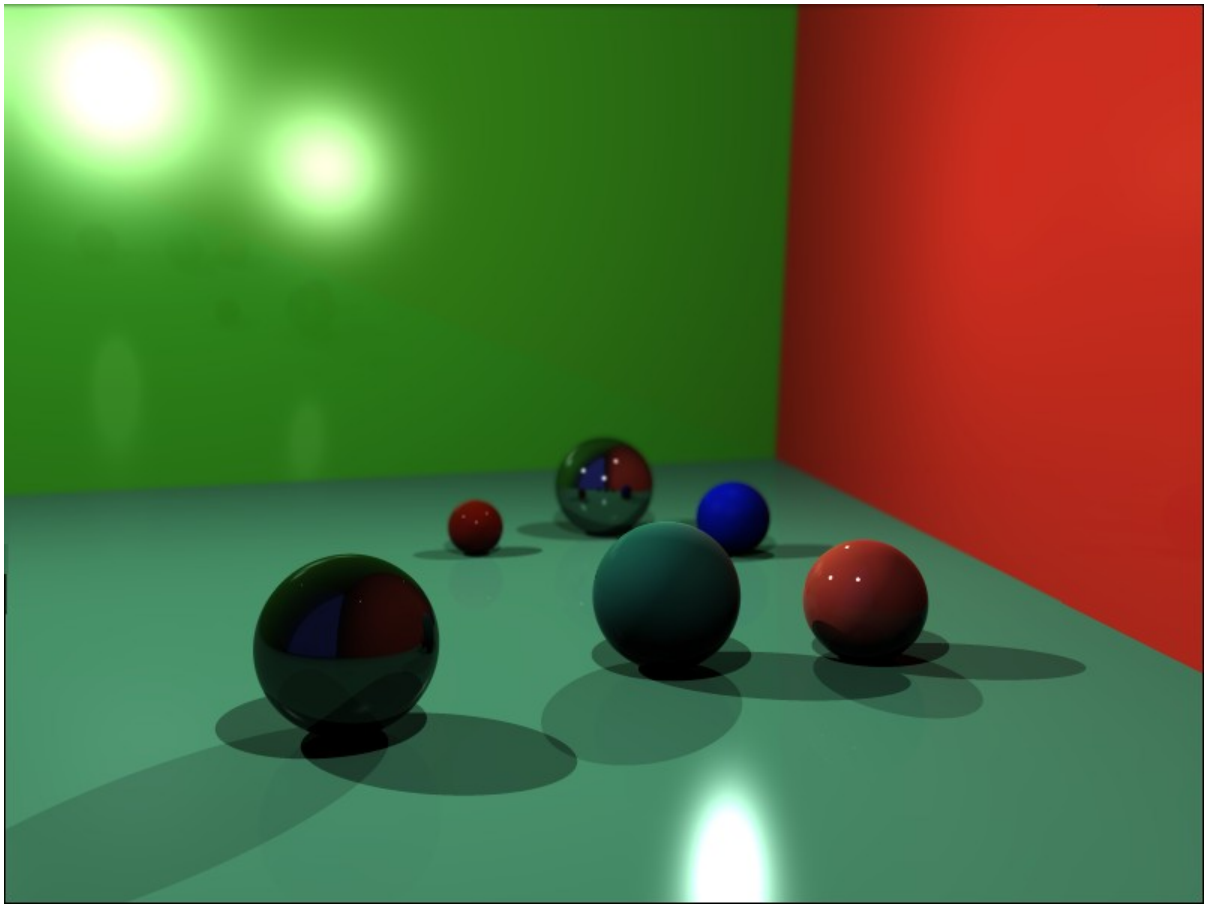
- Pour l'exportation, seules les cases « Write normals », « Write materials », « Include UVs », « Triangulate faces » et « Keep vertex order » **doivent** être cochées.
- Il faut activer la réflexion sur le matériau et la mettre à 0 si on ne souhaite pas de réflexion, en effet le shader utilisé n'est pas testé et par défaut la réflexion est mise à 1 dans Blender.
- Les textures peuvent être chargées avec ce programme mais il faudra tenir compte du path indiqué dans le fichier .mtl.
- Pour finir, les textures sont uniquement chargées en *.png*, je me suis servi de la librairie *lodepng* fournie avec le code du projet.

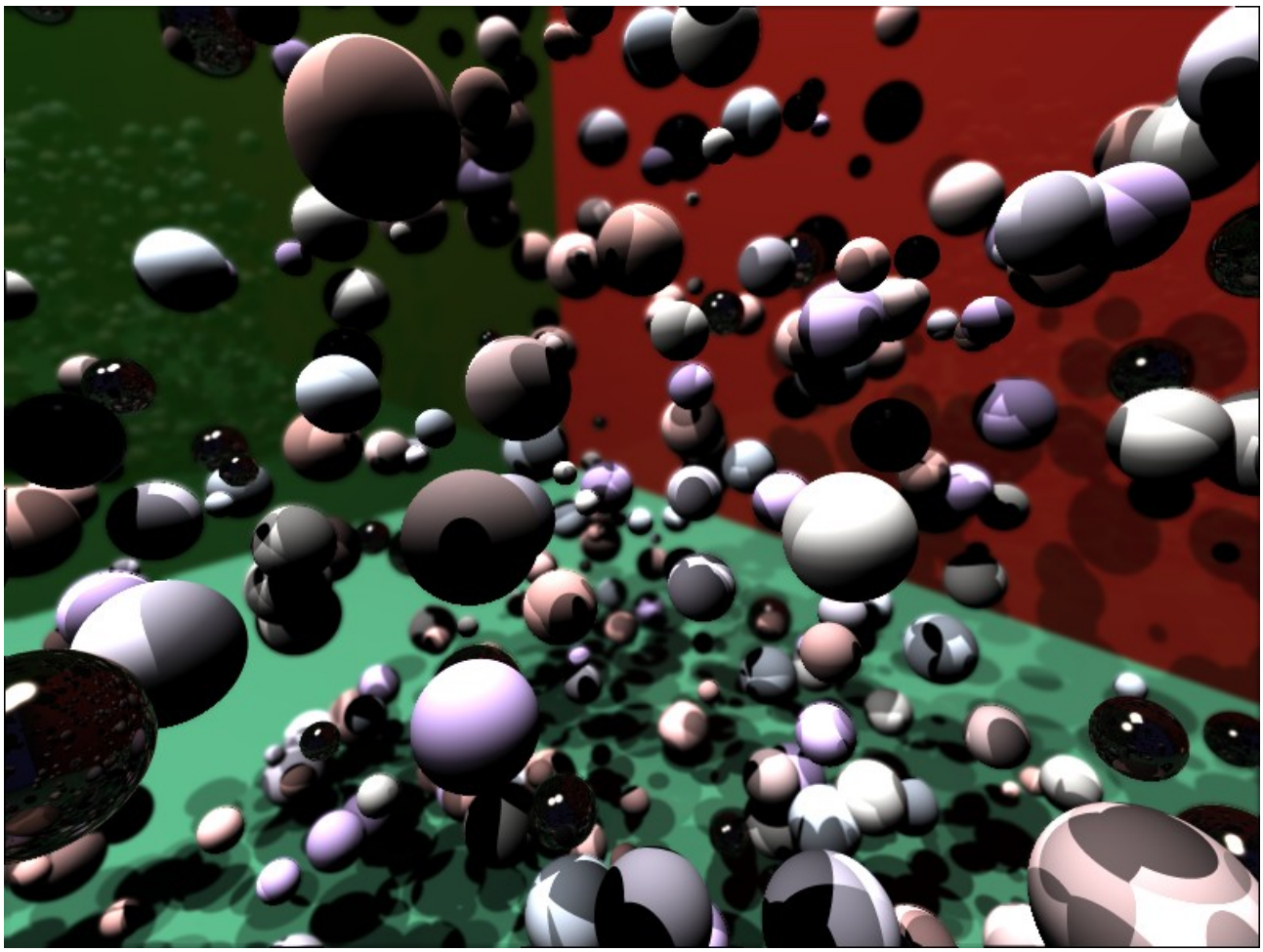
Rendus finaux









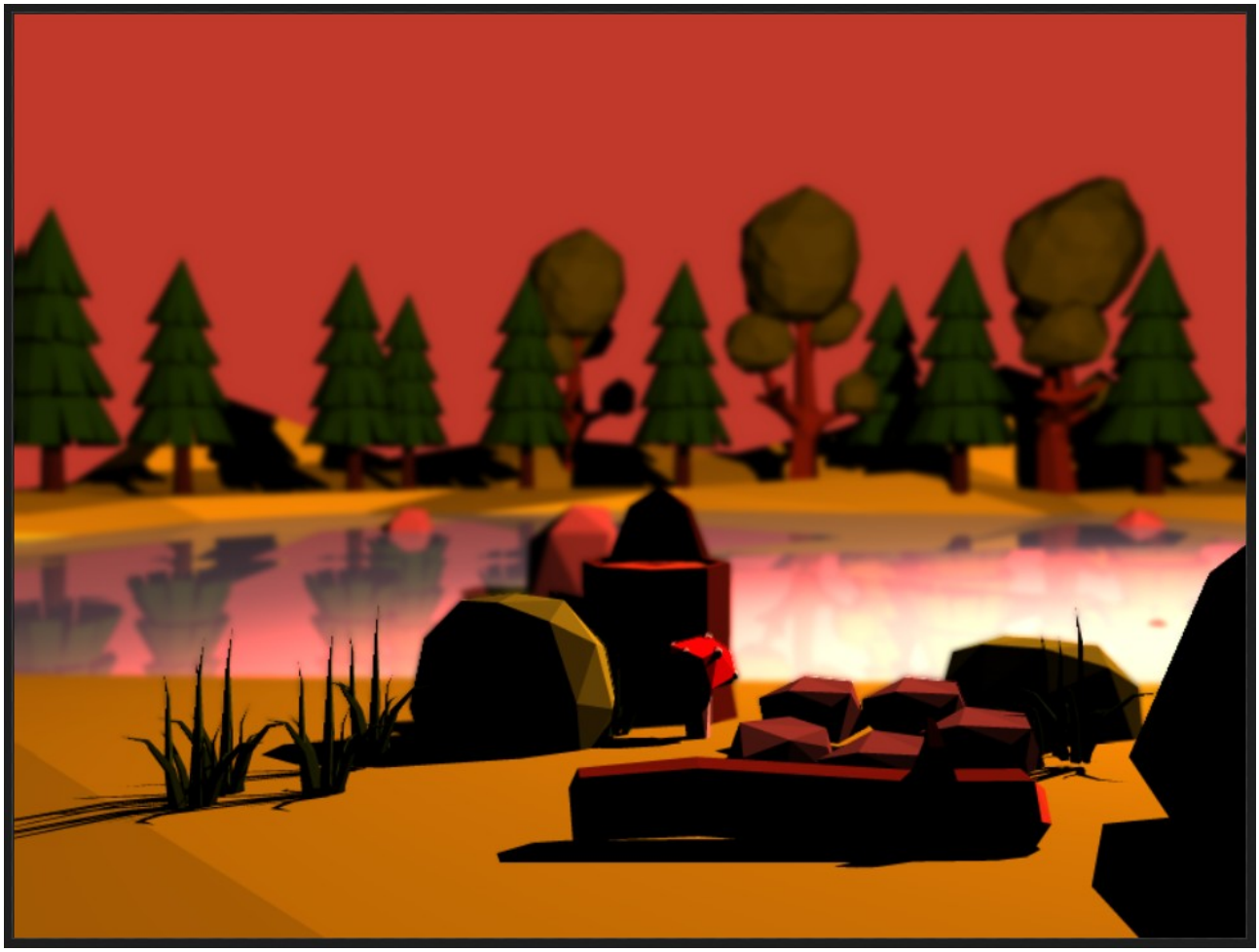


Ces rendus sont disponibles en 800x600 dans le sous-dossier « rendus ».

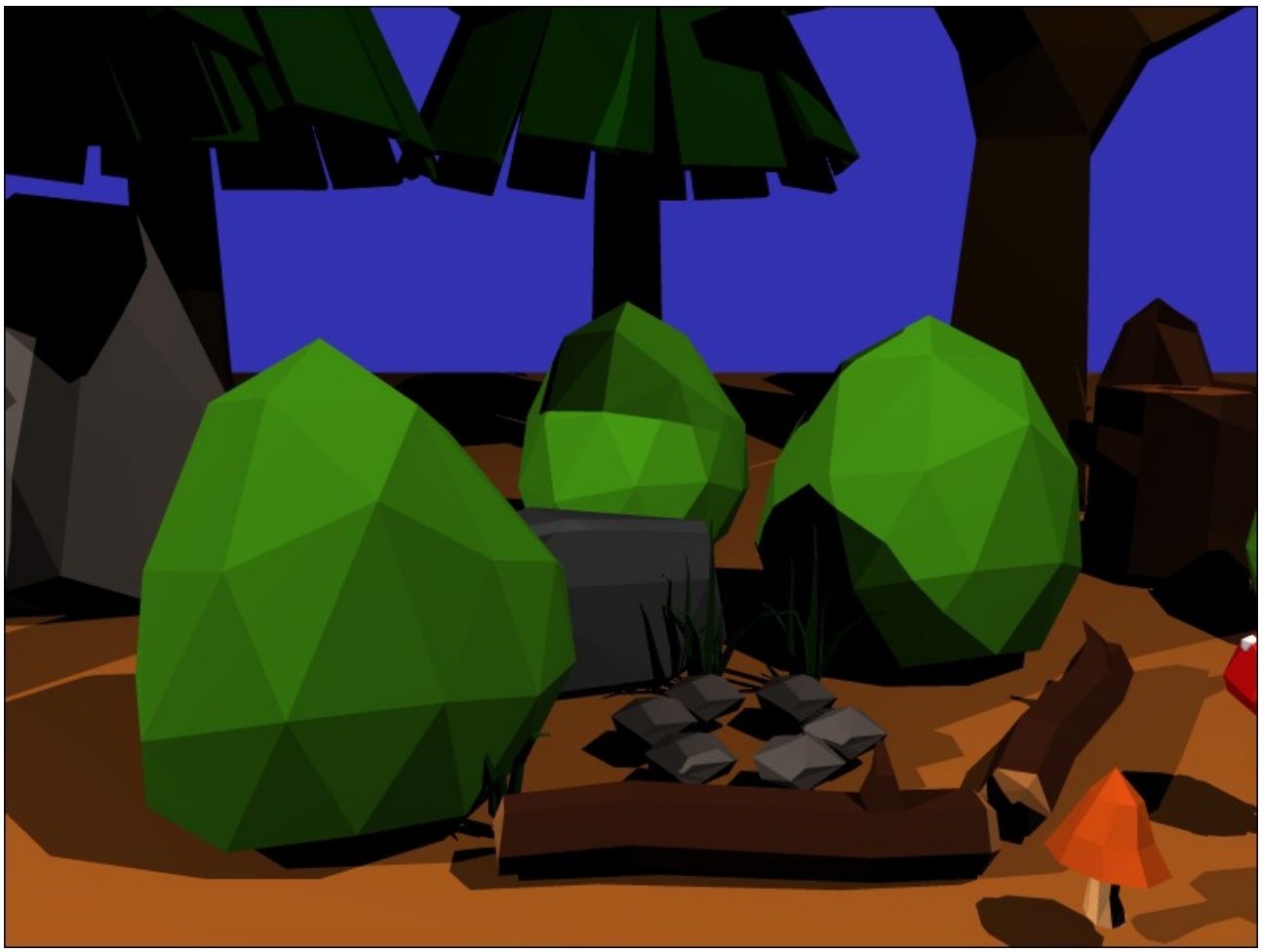
Ils peuvent aussi être générés via la commande :

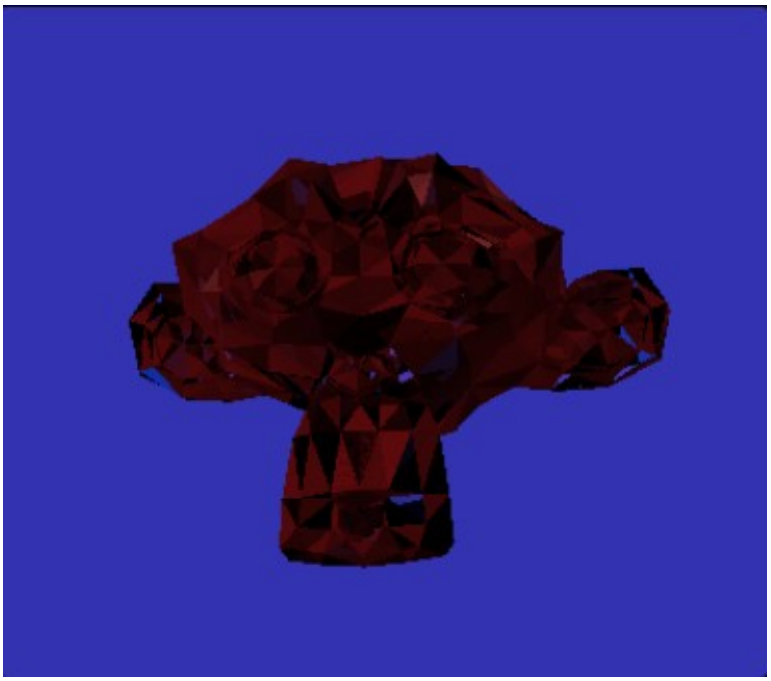
for i in {0..4}; do ./mrt "rendus/scene$i" "$i"; done

Quelques rendus…



On peut même faire du pixel art !





Ça ne fonctionne pas toujours..

Visualisation de la height map pour le depth of field