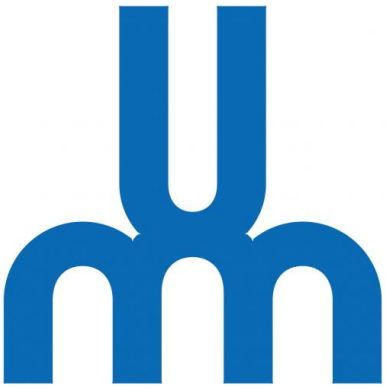
Pathtracing & Filtre SVGF

Samuel Fournier
20218212

Dans le cadre du cours
IFT 6150



Département d'informatique et de recherche opérationnelle
Université de Montréal
Canada
29 octobre 2025

Introduction

Lors de ce projet, j'ai eu la chance d'implémenter un algorithme de débruitage nommé Sptiotemporal Variance-Guided Filter (SVGF) proposé par Christoph Schied (NVIDIA) et al. dans leur article *Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination* [1]. L'algorithme SVGF est un filtre adaptatif qui utilise des informations spatio-temporelles pour réduire le bruit dans les images générées par le pathtracing en temps réel. Dans ce rapport je vais en premier temps expliquer les bases du pathtracing ainsi que les défis associés au rendu en temps réel. Ensuite, je vais détailler le fonctionnement de l'algorithme SVGF et comment il permet d'améliorer la qualité des images rendus par le pathtracing. Finalement, je vais discuter des résultats obtenus et des perspectives futures pour l'amélioration de cet algorithme ainsi que des difficultés rencontrées lors du projet.

1 Pathtracing et rendu en temps réel

Le pathtracing est une technique de rendu basée sur la simulation de la manière dont la lumière interagit avec les surfaces dans une scène 3D. Contrairement aux méthodes de rendu traditionnelles qui utilisent des approximations pour simuler la lumière, le pathtracing suit le trajet des rayons lumineux depuis la caméra jusqu'à la source de lumière, en tenant compte des interactions avec les surfaces rencontrées en cours de route. Cette approche permet de produire des images très réalistes, mais elle est également très coûteuse en termes de calcul, ce qui rend difficile son utilisation pour le rendu en temps réel.

1.1 Survol de l'algorithme

L'algorithme du pathtracing peut être résumé en plusieurs étapes clés :

- **Génération de rayons :** Pour chaque pixel de l'image, un ou plusieurs rayons sont générés à partir de la caméra.
- **Traçage des rayons :** Chaque rayon est tracé à travers la scène, en suivant son trajet jusqu'à ce qu'il atteigne une source de lumière ou qu'il soit absorbé par une surface.
- **Calcul de l'éclairage :** À chaque intersection avec une surface, l'algorithme calcule la contribution de la lumière en fonction des propriétés de la surface (réflexion, réfraction, absorption) et des sources de lumière présentes dans la scène.
- **Accumulation des contributions :** Les contributions lumineuses calculées pour chaque rayon sont accumulées pour déterminer la couleur finale du pixel.

1.2 La génération des rayons

Cette étape est de loin la plus cruciale de l'algorithme de pathtracing. La qualité de l'image finale dépend fortement du nombre de rayons générés par pixel. Malgré son importance, elle est assez simple à implémenter. Pour chaque pixel de l'image, on détermine la position du pixel dans l'espace de la caméra. Pour se faire, on utilise un système de coordonnées normalisées où le coin supérieur gauche de l'image correspond à (0,0) et le coin inférieur droit à (1,1).

```
    ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy); // Position du pixel actuel
    ivec2 imageSize = imageSize(outputImage); // Taille de l'image (1920x1080 par exemple)
    vec2 uv = (vec2(pixelCoords) + vec2(0.5)) / vec2(imageSize);
```

Ensuite, on transforme cette position 2D en une position 3D dans le *Clip Space*.

```
    vec4 clipPos = vec4(screenPos01 * 2 - 1, 1, 1);
```

Ensuite, on utilise la matrice inverse de la projection de la caméra pour transformer cette position en une position dans l'espace de la caméra. Finalement, on calcule la direction du rayon en multipliant la position 3D obtenue par la matrice inverse de vue de la caméra (ce rayon est bien évidemment normalisé).

```
    vec4 viewPos = InverseProjection * vec4(clipPos.xy, -1, 1);
    viewPos.xyz /= viewPos.w;
    vec3 rayDir = normalize(vec3(CameraToWorld * vec4(viewPos.xyz, 0.f)));
```

Une fois que l'on a la direction du rayon, on peut le créer à partir de la position de la caméra (dans l'espace monde) et de la direction calculée précédemment. Il est possible d'ajouter un peu de bruit aléatoire à la position du pixel pour simuler l'anti-aliasage, mais le filtre SVGF n'en a pas besoin pour fonctionner correctement.

```
    float randomX = RandomFloat01(rngState);
    float randomY = RandomFloat01(rngState);
    vec2 screenPos01 = (vec2(pixelCoords) + vec2(randomX, randomY)) / vec2(image_size);
```

1.3 Traçage des rayons

Une fois les rayons générés, ils sont tracés à travers l'entièreté de la scène 3D. Pour chaque rayon, on vérifie s'il intersecte un objet de la scène en utilisant des algorithmes d'intersection géométrique avec les primitives 3D (triangles et sphères dans le cas de ce projet).
Les tests d'intersections sont assez simples et se font en temps constant pour les primitives. Cependant, les maillages 3D (objets composés de plusieurs triangles) vont ralentir le rendu. Dans mon implémentation, j'ai utilisé une structure accélérative nommée *Axis Aligned Bounding Box* (AABB) pour réduire le nombre de tests d'intersections nécessaires. En effet, cette optimisation est essentielle, car sans elle, l'algorithme doit tester chaque rayon contre chaque triangle de tous les maillages de la scène, ce qui est beaucoup de calculs. L'AABB créer une boîte englobante (hit box) autour des maillages. Lors du test d'intersection, on vérifie d'abord si le rayon intersecte la boîte englobante. Si ce n'est pas le cas, on a pas besoin de tester tous les triangles du maillage, on peut simplement l'ignorer. Dans le cas contraire, on doit effectuer les tests d'intersections avec chaque triangle du maillage pour déterminer le point d'intersection exact.
Il existe une autre structure accélérative plus performante nommée *Bounding Volume Hierarchy* (BVH), cependant, je ne l'ai pas encore implémenter, mais c'est une amélioration future que je compte faire.

1.4 Calcul de l'éclairage

Une fois que le rayon intersecte une surface, on doit calculer la contribution de la lumière à ce point. Chaque surface possède un matériel qui définit certains attribus comme son albedo, sa spécularité et sa force et sa couleur d'émission lumineuse (si l'objet est une source de lumière). En utilisant ces informations, on peut déterminer la couleur finale du pixel en accumulant les contributions lumineuses de chaque intersection. Dans mon implémentation, je ne fait que calculer l'illumination

indirecte, c'est-à-dire la contribution provenant des réflexions et réfractions de la lumière sur les autres objets de la scène. Évidemment, une amélioration future serait d'jouter l'illumination directe. Une fois que l'on obtient une intersection, on procède au calcul de l'éclairage indirecte comme suit. En premier lieu, on calcule la couleur émise par la surface (si elle en émet).

```
    vec3 emittedLight = primaryHit.mat.emissionColor.xyz * primaryHit.mat.emissionStrength;
```

Ensuite, on additionne à la couleur finale le produit entre la couleur émise par l'objet et l'énergie du rayon. Ensuite, on multiple l'énergie du rayon par l'albedo de l'objet pour simuler l'absorption de la lumière par la surface.

```
    finalColor += emittedLight * rayColor;
    rayColor *= primaryHit.mat.color.xyz;
```

Ici, la variable `finalColor` est initialisée à (0.f, 0.f, 0.f) et la variable `rayColor` est initialisée à (1.f, 1.f, 1.f). Finalement, pour simuler le chemin que la lumière prend (on fait du PATHtracing après tout), on génère un nouveau rayon à partir du point d'intersection. Pour ce faire, on a besoin de deux directions. La première est une direction diffuse qui est générée aléatoirement dans l'hémisphère au-dessus de l'objet. La seconde est une direction spéculaire qui représente une réflexion parfaite. La direction finale est une interpolation entre ces deux directions en fonction de la spécularité du matériel.

```
    vec3 newPos = (ray.origin + ray.direction * primaryHit.distance) + primaryHit.normal *
     0.000001f;
    ray.origin = newPos;
    uint seed = rngState + frameCnt;
    vec3 diffuseDir = normalize(primaryHit.normal + RandomUnitVector(seed));
    vec3 specularDir = normalize(reflect(ray.direction, primaryHit.normal));
    vec3 newDir = normalize(mix(diffuseDir, specularDir, primaryHit.mat.specular));
    ray.direction = newDir;
```

On met à jour la nouvelle origine du rayon en la décalant légèrement le long de la normale de l'intersection pour éviter les problèmes d'auto-intersection.
On répète ce processus jusqu'à atteindre une de deux conditions d'arrêt: soit le rayon n'intersecte aucun objet (il est parti dans le vide) ou on a atteint le nombre maximum de rebonds (10 dans mon implémentation).

1.5 Accumulation

Cette étape n'est que réalisé que si le nombres de rayons par pixel est supérieur à 1. En effet, pour chaque pixel, on va accumuler la couleur finale obtenue par chaque rayon et le résultat finale qui sera écrit dans l'image de sortie sera la moyenne des couleurs.

```
    averageColor += finalColor;
    ...
    return averageColor / float(RayPerPixel);
```

1.6 En résumé

Le pathtracing est un algorithme de rendu très puissant (surtout lorsque l'on fait de l'illumination globale (illumination indirecte + directe)). Cette technique est fréquemment utilisée dans les films d'animation (Toy Story de Pixar par exemple) pour produire des images les plus photoréalistes possibles. L'algorithme commence à être utilisé dans certains jeux vidéos récents (Cyberpunk 2077 et Indiana Jones and the great circle). Cependant, les résultats peuvent laisser à désirer.

Il est assez évident que le pathtracing est un algorithme de rendu très coûteux. En effet, pour chaque pixel de l'image, on génère plusieurs rayons qui peuvent rebondir plusieurs fois et intersecter plusieurs primitives 3D. La complexité de l'algorithme est donc très élevée, ce qui le rend peu désirable pour le rendu en temps réel. Lorsque l'on l'utilise, il faut souvent compromettre entre la qualité de l'image et la performance. Moins de rayons par pixel et moins de rebonds par rayons vont augmenter le nombre de FPS, mais les rendus seront très bruités. C'est ici que l'algorithme SVGF de NVIDIA entre en jeu.

2 SVGF : Spatiotemporal Variance-Guided Filter

L'objectif principal de l'algorithme SVGF n'est pas de réduire la complexité algorithmique du pathtracing, mais plutôt de permettre l'utilisation d'un nombre réduit de rayons par pixels tout en gardant une qualité d'image satisfaisante. NVIDIA réussit une telle chose en utilisant deux techinques de filtrage. La première est un filtrage temporel qui utilise les informations des images précédentes pour réduire le bruit (similaire à l'accumulation mentionnée précédemment, mais avec les images précédentes). La seconde est un filtrage spatial qui est accomplie grâce à un filtre À-trous.

2.1 Filtrage Temporel

Une méthode d'optimisation assez fréquent dans les pathtracers en temps réel est l'accumulation temporelle. Comme chaque rayon est généré de manière aléatoire, les pixels d'une image à l'autre vont varier légèrement. En accumulant les couleurs des pixels d'une image à l'autre, on peut réduire le bruit et améliorer la qualité de l'image. Cependant, cette méthode possède un inconvénient majeur. Dans le rendu en temps réel, les scènes sont dynamiques ou la caméra (controler par le joueur) peut bouger. Dans ces situations, l'accumulation temporelle naïve va créer des artéfacts visuels (ghosting). La solution de NVIDIA pour contrer ce problème est assez ingénieuse. Lorsque l'on trouve notre première intersection, on va stocker plusieurs informations pertinentes dans des textures (façon de transférer des données sur le GPU avec GLSL). Les informations stockées sont les suivantes:

- La couleur finale du pixel (après tous les rebonds)
- La profondeur de l'intersection
- La normale de l'intersection
- Le meshID de l'objet intersecté
- Le motion vector du pixel

Chacun de ces items sont assez simples à comprendre, sauf peut-être le motion vector.

```
1    HitInfo primaryHit = createHitInfo();
2    primaryHit = intersect(ray);
3    if(primaryHit.hasHit) {
4        imageStore(depthImage, pixelCoords, vec4(primaryHit.distance, 0, 0, 1));
5        imageStore(normalImage, pixelCoords, vec4(primaryHit.normal, 1.f));
6        imageStore(meshIDImage, pixelCoords, uvec4(primaryHit.objectID, 0, 0, 0));
7        vec3 hitPosition = ray.origin + ray.direction * primaryHit.distance;
8        vec3 hitInLocal = vec3(primaryHit.inverseModelMatrix * vec4(hitPosition, 1.f));
```

```
        vec3 previousPositionOfHit = vec3(primaryHit.previousModel * vec4(hitInLocal, 1.f));
        vec4 prevClip = PrevVP * vec4(previousPositionOfHit, 1.f);
        vec2 prevUV = (prevClip.xy / prevClip.w) * 0.5 + 0.5;
        vec4 currClip = CurrentVP * vec4(hitPosition, 1.f);
        vec2 currUV = (currClip.xy / currClip.w) * 0.5 + 0.5;
        vec2 motionVector;
        motionVector = currUV - prevUV;
        imageStore(motionVectorImage, pixelCoords, vec4(motionVector, 0, 1));
    }
    else {
        imageStore(depthImage, pixelCoords, vec4(100000.f, 0, 0, 1));
        imageStore(normalImage, pixelCoords, vec4(0, 0, 0, 1));
        imageStore(meshIDImage, pixelCoords, uvec4(BACKGROUND_ID, 0, 0, 0));
        imageStore(motionVectorImage, pixelCoords, vec4(0, 0, 0, 1));
    }
```

2.1.1 Motion Vector

Le `motionVector` est un vecteur 2D qui représente le déplacement du pixel entre l'image actuelle et l'image précédente. Il est obtenu en retroprojetant la position 3D de l'intersection dans l'espace écran de la caméra précédente. Par exemple, si un objet se déplace vers la droite dans la scène et qu'il intersecte le pixel (100, 100) dans l'image actuelle, alors dans l'image précédente, cet objet aurait intersecté le pixel (95, 100) (par exemple). Le motion vector représenterait donc le vecteur (5, 0) indiquant que le pixel s'est déplacé de 5 unités vers la droite.
Ce vecteur est essentiel pour le filtrage temporel, car il permet de retrouver les informations de l'image précédent.

2.1.2 Le filtrage

Une fois que nous avons stocké les informations nécéssaires, on procède au filtrage temporel. Pour chaque pixel de l'image actuelle, on utilise le motion vector pour retrouver la position du pixel dans l'image précédente.

```
    ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
    ivec2 screenDim = imageSize(denoisedOutput);
    vec2 currentUV = (vec2(pixelCoords) + 0.5f) / vec2(screenDim);
    vec4 motionTexel = texture(MotionVectorTexture, currentUV);
    vec2 motionVector = motionTexel.rg;
    vec2 historyUV = currentUV - motionVector;

    historyColor = texture(HistoryColorTexture, historyUV);
    historyDepth = texture(HistoryDepthTexture, historyUV).r;
    historyNormal = normalize(texture(HistoryNormalTexture, historyUV).rgb);
    historyMeshID = int(texture(HistoryMeshIDTexture, historyUV).r);
```

Une fois la position retrouvée, on procède à un série de comparaisons pour déterminer si l'objet intersecté se trouve encore dans le même pixel.
Le premier test est de vérifier la pronfondeur de l'intersection. Si la différence absolue entre la profondeur actuelle et la pronfondeur précédente est supérieure à un certain seuil arbitraire (0.05 dans mon implémentation), alors on considère que l'objet n'est plus le même.

```
    if(abs(currentDepth - historyDepth) > DEPTH_THRESHOLD) {
        isValid = false;
    }
```

Le second test de vérifier si la normale de l'intersection à suffisement changé. Si le cosinus de l'angle entre la normale actuelle et la normale précédente est supérieure à un certain seuil arbitraire (0.9 dans mon implémentation), alors on considère que l'objet n'est plus le même.

```
    vec3 currentNormal = normalize(texture(CurrentNormalTexture, currentUV).rgb);
    if(length(currentNormal) < EPSILON && length(historyNormal) < EPSILON) {
        isValid = true;
    }
```

Le troisième et dernier test est de vérifier si le meshID de l'objet est le même. Si ce n'est pas le cas, alors on considère que l'objet n'est plus le même.

```
    int currentMeshID = int(texture(CurrentMeshIDTexture, currentUV).r);
    if(currentMeshID != historyMeshID) {
        isValid = false;
    }
```

Si tous les tests passent avec succès, alors on peut utiliser la couleur précédente pour l'accumulation temporelle. L'accumulation se fait en utilisant un *Exponential Moving Average* (EMA) qui permet de donner plus de poids à l'historique débruitée qu'à l'image actuelle bruitée. Si un des tests échoue, alors on n'utilise pas l'historique et on garde la couleur actuelle bruitée.

```
    vec3 finalColor = noisyColor.rgb;
    float currentAccumulationCount = 1.f;
    if(isValid) {
        float oldAccumulationCount = historyColor.a;
        float maxCount = 1024.f;
        float clampedOldCount = min(oldAccumulationCount, maxCount);
        float alpha = 1.f / (clampedOldCount + 1);
        //finalColor = vec3(1, 1, 1);
        finalColor = mix(historyColor.rgb, noisyColor.rgb, alpha);

        currentAccumulationCount = oldAccumulationCount + 1;
    }
    imageStore(denoisedOutput, pixelCoords, vec4(finalColor.rgb, currentAccumulationCount));
```

En plus de tout cela, on stocke aussi les premiers et seconds moment de l'image débruitée pour le filtrage spatial. Le premier moment est simplement la couleur débruitée, tandis que le second est le carré de la couleur débruitée.

```
    vec3 colorSquared = finalColor * finalColor;
    imageStore(firstMomentOutput, pixelCoords, vec4(finalColor, 1.f));
    imageStore(secondMomentOutput, pixelCoords, vec4(colorSquared, 1.f));
```

2.2 Avant le filtrage spatial

Avant d'entammer le filtrage spatial, il est nécessaire de calculer la variance de chaque pixel. Premièrement, on récupère la normale et la profondeur de chaque pixel. Ensuite, on calcule le gradient de la profondeur. Ce gradient est utilisé pour "normaliser" la profondeur en fonction de la géométrie. On commence par initialiser trois variables: `sumWeights` $= 0.f$, `sumMoments1` $=$ `sumMoments2` $= (0.f, 0.f, 0.f)$. Ensuite, on itère sur un voisinage de taille arbitraire (7x7 dans mon cas). Pour chaque pixel du voisinage, on calcule un poids en fonction de la normale et de la profondeur.
Le poids de la normale est calculé de la façon suivante:

```
    float weightNormal = pow(max(0.f, dot(centerNormal, sampleNormal)), PHI_NORMAL);
```

Ici `PHI_NORMAL` est une constante arbitraire (128.f dans mon implémentation) qui contrôle l'importance de la normale. Si les normales sont similaires, le poids sera proche de 1, sinon il sera proche de 0. On utilise la puissance pour accentuer la différence entre les normales similaires et dissimilaires. Par exemple, s'il y a un *sharp edge* entre deux surfaces (comme dans un cube par exemple), le calcul du poids s'assure d'être bas pour que le filtre spatial ne tente pas de les mélanger ensemble,

ce qui créérait des artéfacts visuels comme un *edge* flou.
Le poids de la profondeur est calculé de la façon suivante:

```
    float weightDepth = (abs(centerDepth - sampleDepth) / centerDepthGradient);
    weightDepth = exp(-weightDepth * PHI_DEPTH);
```

Ici `PHI_DEPTH` est une constante arbitraire (1.f dans mon implémentation) qui contrôle l'importance de la profondeur. On divise la différence absolue des profondeurs par le gradient de la profondeur pour normaliser la différence en fonction de la géométrie. En effet, si l'objet est incliné (par exemple), la profondeur peut varier rapidement. Par conséquent, la différence absolue serait trop élevée et le poids serait trop faible, ce qui indiquerait incorrectement au filtre spatial de ne pas mélanger ces pixels ensemble.
Finalement, on calcule le poids total en multipliant les deux poids ensemble. On utilise ce poids pour accumuler les moments.

```
    float w = weightNormal * weightDepth;
    if(w > 1e-4) {
        vec3 m1 = imageLoad(firstMoments, samplePos).rgb;
        vec3 m2 = imageLoad(secondMoments, samplePos).rgb;

        sumMoments1 += m1 * w;
        sumMoments2 += m2 * w;
        sumWeight += w;
    }
```

Une fois cette étape complétée, on divise la sommes des moments accumulées par la sommes des poids accumulées pour obtenir les moments finaux.

```
    sumMoments1 /= sumWeight;
    sumMoments2 /= sumWeight;
```

On calcule ensuite la variance en utilisant la formule suivante:

$$Var(X) = E[X^2] - E[X]^2$$

Et on stocke la variance maximale en `rgb` pour chaque pixel.

```
    vec3 varianceRGB = (sumMoments2 - sumMoments1 * sumMoments1);
    float varianceFinal = max(varianceRGB.r, max(varianceRGB.g, varianceRGB.b));
    imageStore(varianceTexture, pixelPos, vec4(varianceFinal, 0.f, 0.f, 0.f));
```

2.3 Filtre Spatial

Maintenant que nous avons accomplie l'accumulation temporelle et le calcul des variances, nous pouvons procéder à la dernière étape de cet algorithme. Cette étape est un filtrage spatial utilisant le filtre À-trous. Le principe de se filtre est d'appliquer un flouttage sur l'image. Cependant, ce filtre est adaptif. Plutôt que d'appliquer un filtre 10x10 sur chaque pixel, on applique un filtre 3x3 troué. À chaque itération de cette passe de filtrage, on double l'espacement entre les échantillons. Pour la première itération, notre espacement (`stride`) est de 1. Par conséquent, on échantillone les voisins adjacents de chaque pixel. Pour la seconde itération, notre espacement est de 2. Par conséquent, on échantillone les pixels à une distance de 2. On répète ce processus pour un nombre d'itérations arbitraire (5 dans mon implémentation). Cette méthode nous permet de couvrir un très grand voisinage sans nécessairement utilisé un grand filtre. Par exemple, après 5 itérations, on couvre un voisinage de 33x33 pixels avec un simple filtre 3x3. De plus, comme ce filtre est un filtre de type *Wavelet*, on décompose l'image en différents bandes de fréquences, ce qui nous permet

d'atténuer/supprimer le bruit présent dans l'image tout en préservant les détails importants. De plus, ce filtre ne fait pas qu'appliquer un flouttage uniforme. En effet, ce filtre utilise des fonctions de *edge-stopping*. Ces fonctions permettent au filtre de s'adapter en fonction des caractéristiques locales de l'image et de préserver les bords et les détails important des objets présent dans la scène. Le filtrage spatial se fait de la façon suivante. Pour chaque pixel de l'image, on initialise 3 variables: `sumColor` $= (0.f, 0.f, 0.f)$, `sumVariance` = `sumWeight` $= 0.f$. Ensuite, on itère sur un voisinage 5x5 avec un espacement (`stride`).

```
    for(int y = -2; y <= 2; y++) {
        for(int x = -2; x <= 2; x++) {
            ivec2 offset = ivec2(x, y) * stepSize;
            ivec2 samplePos = pixelPos + offset;

            // Boundary check to stay within image
            if(samplePos.x < 0 || samplePos.y < 0 || samplePos.x >= size.x || samplePos.y >=
     size.y) {
                continue;
            }
            ...
        }
    }
```

Pour chaque pixel du voisinage, on calcule trois poids en fonction de la normale, de la profondeur et de la couleur. Les poids de la normale et de la profondeur sont calculés de la même manière qu'à l'étape précédente. Le poids de la couleur est calculé de la façon suivante:

```
    float weightLuminance = abs(centerLum - sampleLuminance) / (PHI_COLOR * sqrt(max(0.f,
     centerVariance)) + 1e-6);
    weightLuminance = exp(-weightLuminance);
```

Ici, `PHI_COLOR` est une constante arbitraire (4.f dans mon implémentation) qui va contrôler l'importance de la couleur. La division par la racine carrée de la variance permet d'adapter le poids en fonction du niveau de bruit local. Sans cette division, les pixels dans les zones bruitées auraient des poids qui risqueraient d'être trop faibles, ce qui empêcherait le filtre de mélanger les pixels ensemble pour réduire le bruit. La division par la racine carrée de la variance permet de normaliser le poids en fonction du niveau de bruit, ce qui nous permet de mieux gérer les zones bruitées. Si on utilise pas cette normalisation, on devrait utiliser un seuil arbitrairement choisi. Cependant, ce seuil ne serait pas idéal dans plusieurs situations. Un seuil trop faible nous permettrait de préserver les détails dans les zones peu bruitées, mais ne réduirait pas suffisament le bruit dans les zones très bruitées. À l'inverse, un seuil trop élevé aurait l'effet inverse. Les zones très bruitées seront atténuer ou complètement éliminer, mais on risque de perdre les détails dans les zones peu bruitées.
Une fois que nous avons les trois poids de calculés, on calcule le poids total en les multipliants ensemble.

```
    float w = weightLuminance * weightDepth * weightNormal;
```

Ensuite, on mutiplie ce poids par le poids dans le noyau du filtre 5x5 pour obtenir le poids final.

```
    float finalWeight = w * kernelWeight;
```

Cette étape est très importante. Le noyau du filtre 5x5 est un noyau presque gaussien. En effet, c'est un filtre B3-Spline qui est une approxmation du filtre gaussien.

```
const float kernel[25] = float[](
    1.0/256.0, 1.0/64.0, 3.0/128.0, 1.0/64.0, 1.0/256.0,
    1.0/64.0,  1.0/16.0, 3.0/32.0,  1.0/16.0, 1.0/64.0,
    3.0/128.0, 3.0/32.0, 9.0/64.0,  3.0/32.0, 3.0/128.0,
    1.0/64.0,  1.0/16.0, 3.0/32.0,  1.0/16.0, 1.0/64.0,
    1.0/256.0, 1.0/64.0, 3.0/128.0, 1.0/64.0, 1.0/256.0
);
```

Multiplier le poids obtenu avec ce noyau permet de réduire l'impact des pixels éloignés du centre filtre. Cette particularité est essentielle pour avoir un flouttage lisse plutôt qu'en bloc. Bref, le poids w représente la mesure de similarité entre le pixel central et le pixel échantillonné. Ensuite, la multiplication par le noyau permet de réduire l'impact qu'un pixel éloigné aura sur le résultat final et favorise les pixels proches du centre du filtre. Bien évidemment, les pixels plus proche du centre on plus de chance d'être similaires au pixel central que les pixels éloignés.
Ensuite, on accumule dans `sumColor` la couleur du pixel pondérer par le poids final obtenue. On accumule aussi dans `sumVariance` la variance du pixel pondérer par le poids final au carré. Finalement, on accumule dans `sumWeight` le poids final.

```
    sumColor += sampleColor * finalWeight;
    sumVariance += sampleVariance * (finalWeight * finalWeight);
    sumWeight += finalWeight;
```

Finalement, une fois que tout le voisinage a été échantilloné, on divise la somme des couleurs accumulées par la somme des poids. Cette division permet de normaliser la couleur finale, en exécutant une moyenne pondérer des couleurs échantillonnées. Sans cette pondération, les couleurs avec des poids élevés domineraient la couleur finale et risqueraient d'exploser et de devenir plus lumineuses que prévu. À l'inverse les couleurs avec des poids faibles n'auraient aucun impact sur la couleur finale (les poids élevées dominent la somme), ce qui n'est pas idéale non plus.

```
    vec3 finalColor = sumColor / sumWeight;
```

On fait la même chose pour la variance.

```
    float finalVariance = sumVariance / (sumWeight * sumWeight);
```

On stocke la couleur finale dans l'image de sortie et on sotcke la nouvelle variance dans la texture de variance. Il est important de mettre à jour la variance. En effet, la variance est initialement calculée à partir d'une image plus bruitée que celle obtenue après le filtrage spatial (même après l'accumulation temporelle il y a encore du bruit). Par conséquent, les variances initiales sont plus élevées que la variance réelle (que l'on ne connaît pas, mais que l'on estime). Par conséquent, chaque itération du filtre À-trous va réduire le bruit et améliorer les mesures de variances des pixels pour les prochaines itérations. Essentiellement, on applique un débruitage sur le guide du débruiteur pour que chaque itération soit plus éfficace que la précédente. Un léger problème que j'ai remarqué, mais auquel je n'ai toujours pas trouvé la solution est les matériaux réfléchissants. Dans les images à la fin du rapport, le pathtracer pur gère ces matériaux corréctement, mais le débruiteur rend le matériel flou (ce qui est attendu étant donné ce que l'algorithme fait).

2.4 En résumé

L'algorithme SVGF de NVIDIA est composée de deux phases de filtrage. La première est un filtrage temporel qui va prendre en compte les informations des images précédentes (normales, profondeurs, couleurs et meshIDs) pour accumuler les pixels d'une image à l'autre. Cette étape nous permet de rapidement réduire le bruite dans les zones statiques de la scène. Cependant, cette étape n'est pas suffisante pour éliminer tout le bruit, surtout dans les zones dynamiques. C'est ici que la seconde phase de filtrage entre en jeu. Cette phase utilise un filtre À-trous de taille adaptive pour appliquer un flouttage spatial sur l'image. Ce filtre utilise des poids basés sur la similarités des normales, des profondeurs et des couleurs du voisinage pour préserver les bords et les détails importants de la scène. Ces poids sont aussi pondérés par un noyau B3-Spline qui approxime un filtre gaussien pour

réduire l'impact que les pixels éloignés auront sur le résultat final. Finalement, la couleur de sortie est obtenue en effectuant une moyenne pondérée des couleurs échantillonnés du voisinage avec leur poids respectifs pour normaliser la couleur finale et s'assurer que la couleur finale ne dépassent pas $1.f$ (ou 255) en intensité et permettre aux couleurs avec des poids plus faibles d'avoir un impact sur le résultat final. Finalement, on met à jour la variance des pixels pour améliorer la précision de la priochaine itération.

3 Conclusion

Dans ce rapport, j'ai présenté l'algorithme de pathtracing ainsi que l'algorithme de SVGF de NVIDIA. J'ai expliqué en détail chaque étapes de ces algorithmes et comment on peut les combiner pour obtenir des images de hautes qualités. Avant de conclure, j'aimerais discuter des difficultés du pathtracing en temps réels.
Comme il l'a été mentionné précédemment, la pathtracing est un algorithme de rendu très coûteux. Si le pathtracer est implémenter sur le CPU d'un ordinateur, il est presque impossible d'atteindre des performances en temps réel. En effet, la complexité théorique de cet algorithme est $\mathcal{O}(P \cdot R \cdot B \cdot I)$ où P est le nombre de pixels de l'image ($2,073,600$ pixels pour une image HD 1920x1080), R est le nombre de rayons par pixels, B est le nombre de rebonds par rayons et I est le nombre de tests d'intersections par rebonds. Cette complexité empire lorsqu'on ajoute l'illumination directe, car à chaque intersection, on doit lancer des rayons supplémentaires vers chaque source de lumière pour calculer leut contribution. Par exemple, pour une image HD avec 10 rayons par pixels, 10 rebonds maximals et 100 tests d'intersections par rebonds (100 primitives), il faut calculer $2,073,600 \cdot 10 \cdot 10 \cdot 100 = 20,736,000,000$ tests d'intersections pour chaque image. Si on désire atteindre 30 images par secondes (30 FPS), cela voudrait dire que le CPU devrait être capable d'effectuer $622,080,000,000$ tests d'intersections par secondes. Même avec le meilleur CPU sur le marché, c'est un objectif impossible.
Par contre, on peut prendre avantage d'une astuce particulière. En effet, la couleur attribué à chaque pixel est indépendante des autres. Par conséquent, on peut paralléliser le processus de rendu. Même avec du multithreading sur le CPU, on ne peut pas atteindre des performances en temps réel. Cependant, l'utilisation du GPU change complètement la donne. En effet, le GPU est conçu pour ce genre de calculs massivement parallèles. C'est pour cela que dans ce projet, j'utilise des *Compute Shaders* en GLSL pour implémenter le pathtracer. Par contre, même avec le GPU, une scène moindrement complexe (comme une salle dans un jeux vidéos modernes) est trop demandante pour être rendue en temps réel avec du pathtracing pur.
C'est ici que l'algorithme SVGF de NVIDIA est entré en jeu. En utilisant des filtres de débruitage spatial et temporel, on est en mesure de drastiquement réduire le nombre de rayons par pixels nécessaires pour obtenir une image de haute qualité. Par exemple, dans mon implémentation, j'utilise qu'un seul rayon par pixel et l'image obtenu est de meilleur qualité que le pathtracer pur avec 1000 rayons par pixels (sans SVGF) avec un taux d'images par secondes environ 30 fois plus élevé.
Une autre amélioration au pathtracer est l'utilisation d'un BVH comme mentionné précédemment. Sans trop entrer dans les détails, car ce n'est pas le sujet du rapport, un BVH partitionne l'espace en une hiéarchie d'objets englobants. Lors des tests d'intersections, plutôt que de tester chaque objet, on fait un parcours de la hiéarchie (un arbre binaire), ce qui nous permet de réduire le nombre de tests à faire.
En conclusion, le pathtracing en temps réel est une tâche difficile à accomplir. Cependant, les ren-

dus obtenus sont d'une qualité incomparable aux autres techniques de rendu. Mais, comme je viens de le mentionner, c'est une tâche incroyablement coûteuse pour un ordinateur même si on utilise le GPU. C'est pour ça qu'il existe un grand domaine de recherche qui tente d'optimiser les temps de rendus avec différentes méthodes. Une de ces méthodes est le débruitage et l'algorithme de SVGF de NVIDIA accompli cette tâche intelligement en prenant en considération les informations obtenus lors des tests d'intersections. Les résultats obtenus par cet algorithme sont très impréssionnant. Comme je l'ai mentionné plus haut, j'ai été en mesure d'obtenir des rendus de meilleures qualités que la pathtracer pur tout en gardant un taux d'images par secondes que je considerais très bon pour du rendu en temps réel.

4 Notes sur le projet

Ce projet de pathtracer est un projet personnel que j'ai entâmé vers la fin de l'été et je l'utilise comme un opportunité d'approfondir mes connaissances sur *OpenGl* et le pathtracer. Par conséquent, il est évident que plusieurs endroits dans mon code pourraient être optimiser ou améliorer (malheureusement ma compréhension de *OpenGl* n'est pas parfaite). Le code CPU est écrit en C++ et le code des *Compute Shaders* est écrit dans le langage de *shading* GLSL. Le programme est compilé avec un CMake qui inclut les librairies nécessaires pour l'exécution du code. Les librairies utilisées sont: glm (une librairie d'algèbre linéaire), GLFW (nécessaire pour intéragir avec le GPU et l'api *OpenGl*), GLAD, stb (pour sauvegarder les images jpeg/png) et assimp (une librairie pour importer des maillages). J'ai utilisé l'éditeur de code *CLion* qui m'a permis d'installer les librairies grâce à *VCPKG* qui est inclus par défaut dans le logiciel. Finalement, pour donner un *baseline* pour les performances, mon ordinateur personnel possède un CPU `AMD Ryzen 5 7600 6-Core Processor` et un GPU `AMD Radeon RX 6800`.

Images produites

Dans cette section, je vais mettre quelques images générées par le pathtracer. En plus, sur le Github du projet, je vais mettre des vidéos dans le dossier `Results` pour montrer la différence de performance. La sphère jaune bouge en orbite autour de la boule verte.

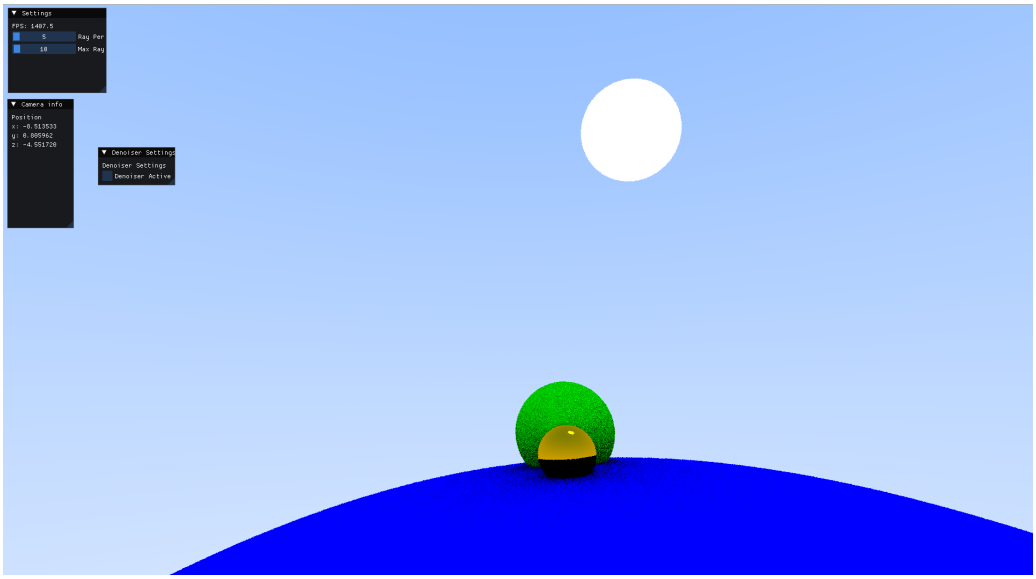


Figure 1: Rendu bruité avec 98 rayons par pixels. FPS = 1407.5, 4 sphère et 0 triangles

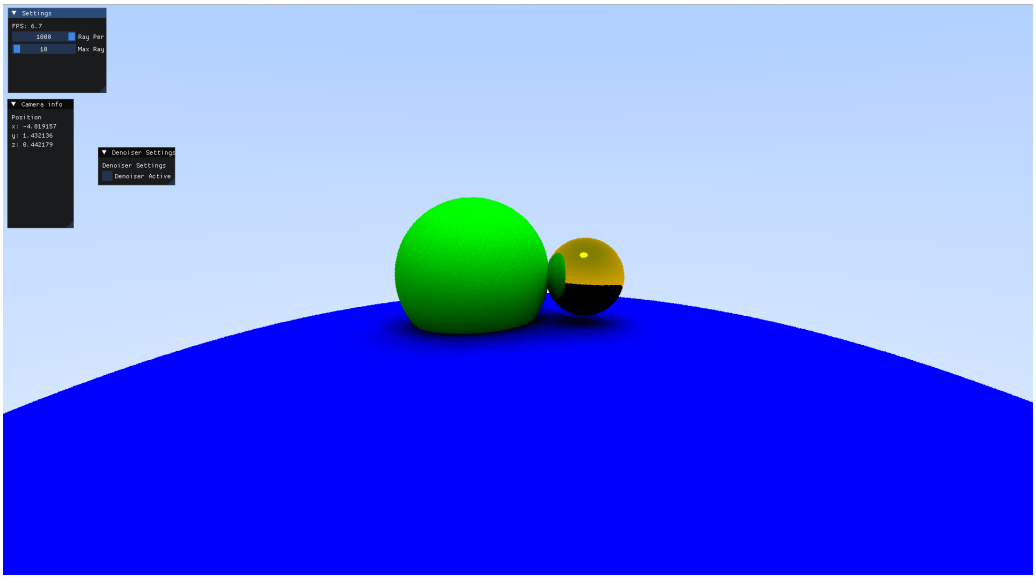


Figure 2: Rendu bruité avec 1000 rayons par pixels. FPS = 6.7, 4 sphère et 0 triangles

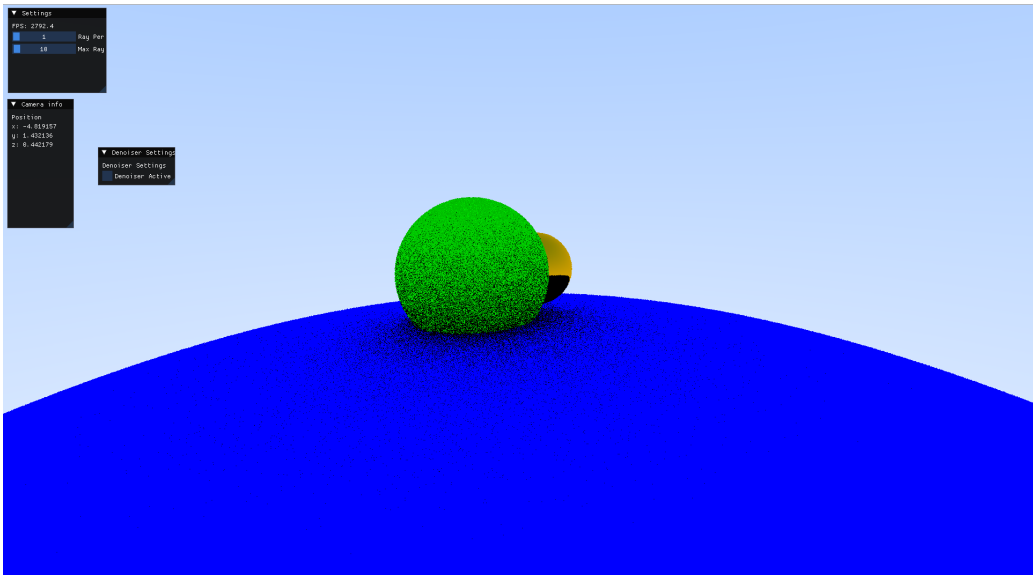


Figure 3: Rendu bruité avec 1 rayons par pixels. FPS = 2792.4, 4 sphère et 0 triangles

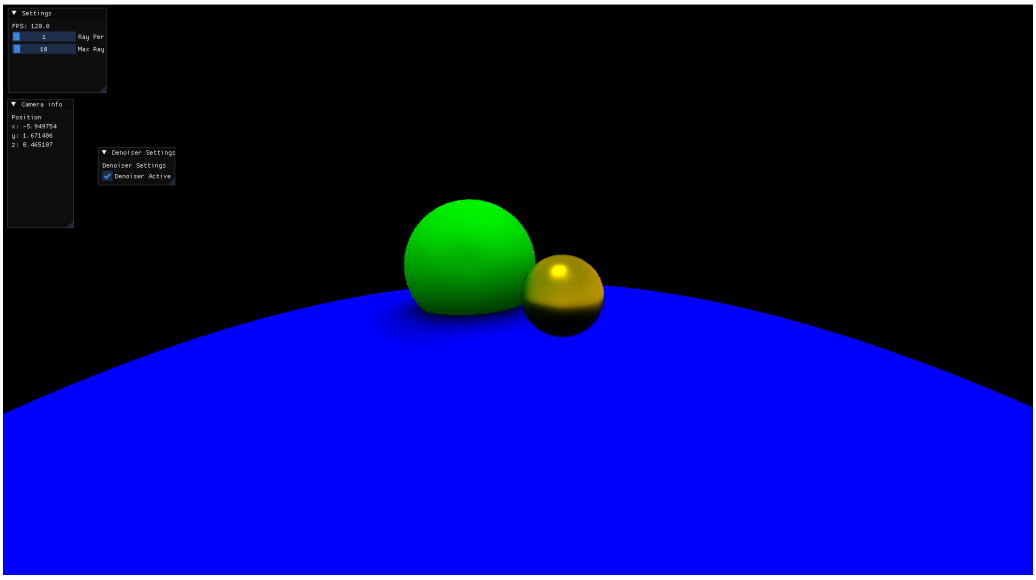


Figure 4: Rendu débruité avec 1 rayons par pixels. FPS = 120.0, 4 sphère et 0 triangles

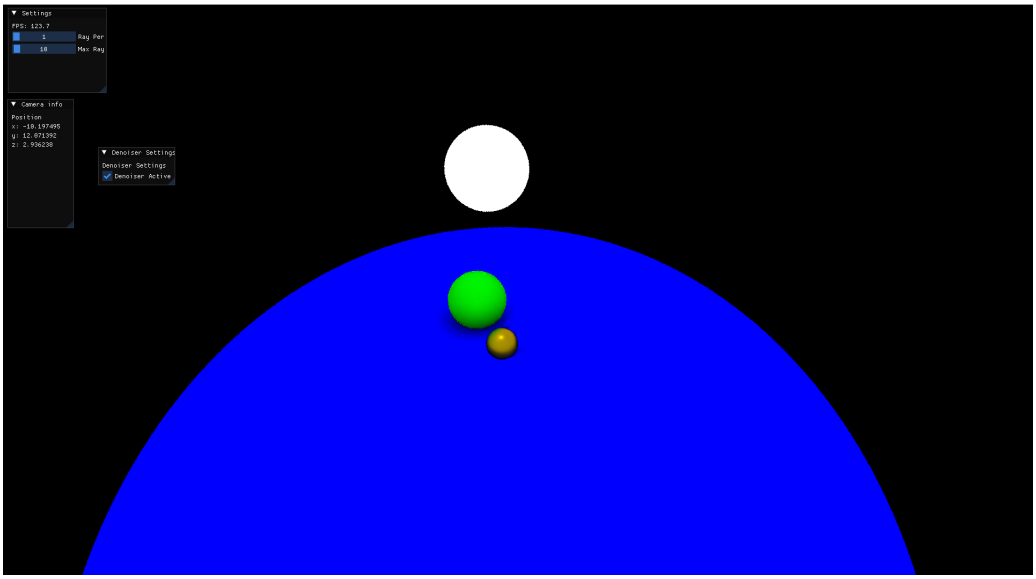


Figure 5: Rendu débruité avec 1 rayons par pixels. FPS = 123.7, 4 sphère et 0 triangles

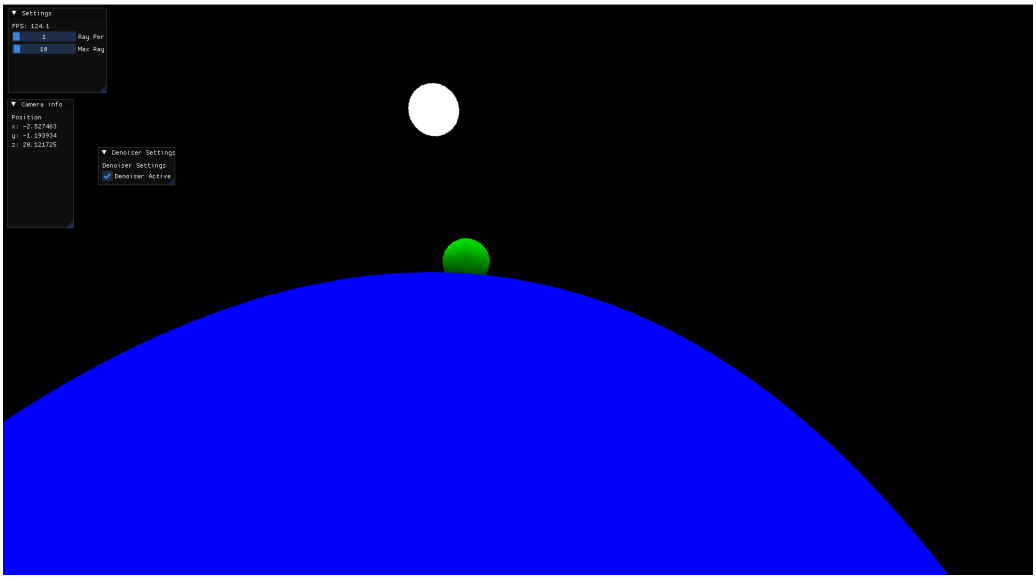


Figure 6: Rendu débruité avec 1 rayons par pixels. FPS = 124.1, 4 sphère et 0 triangles

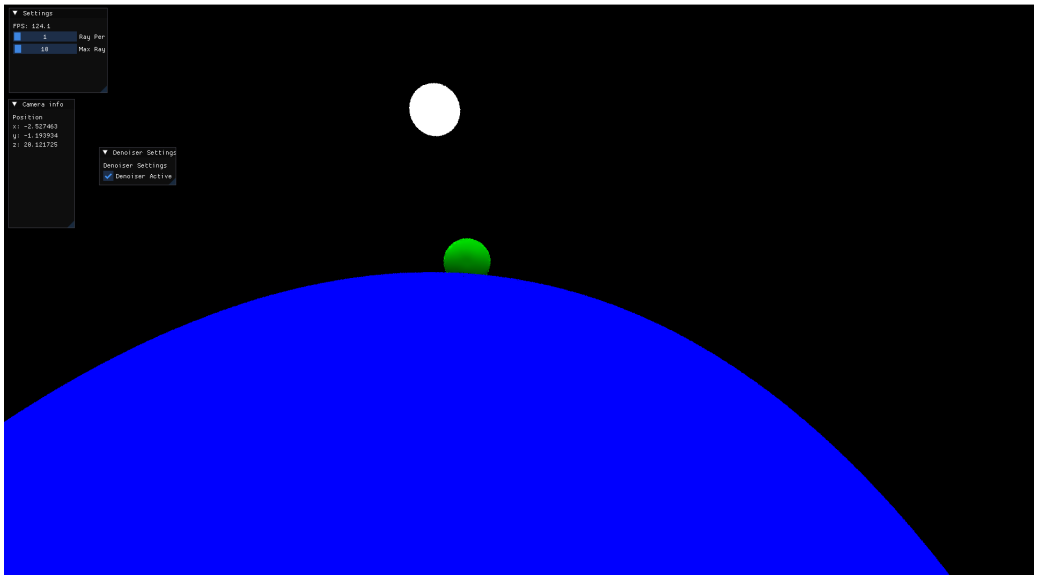


Figure 7: Rendu débruité avec 1 rayons par pixels. FPS = 124.1, 4 sphère et 0 triangles

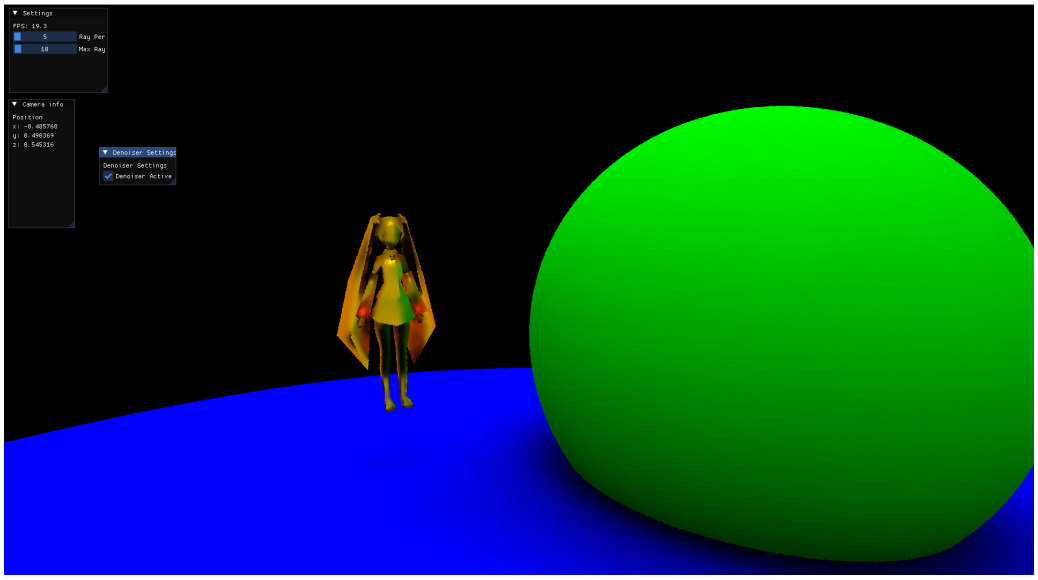


Figure 8: Rendu débruité avec 5 rayons par pixels. FPS = 1184.1, 3 sphère et 595 triangles (maillage low poly Hatsune Miku)

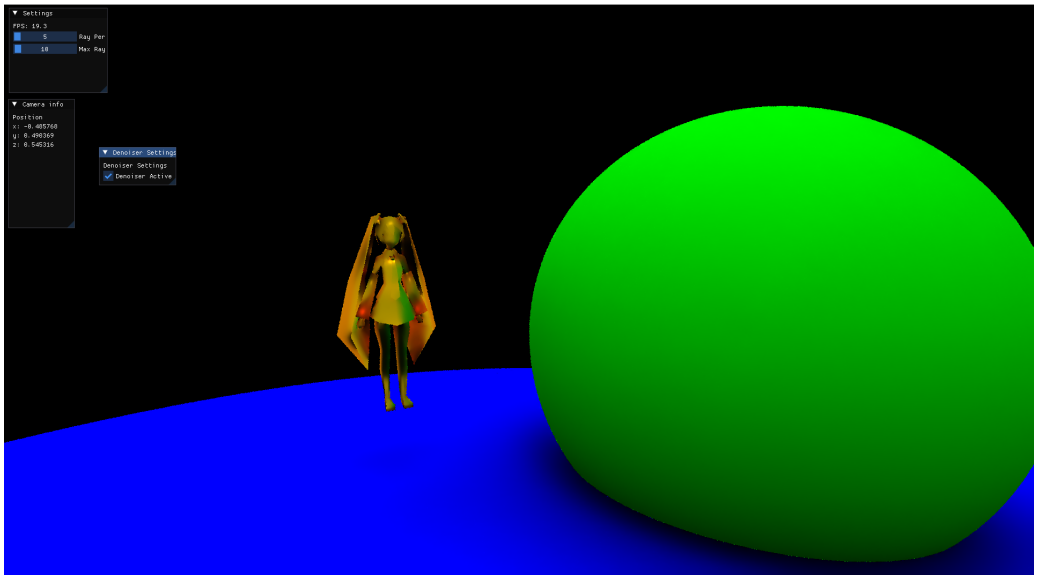


Figure 9: Rendu débruité avec 5 rayons par pixels, angle différent. FPS = 1184.1, 3 sphère et 595 triangles (maillage low poly Hatsune Miku)

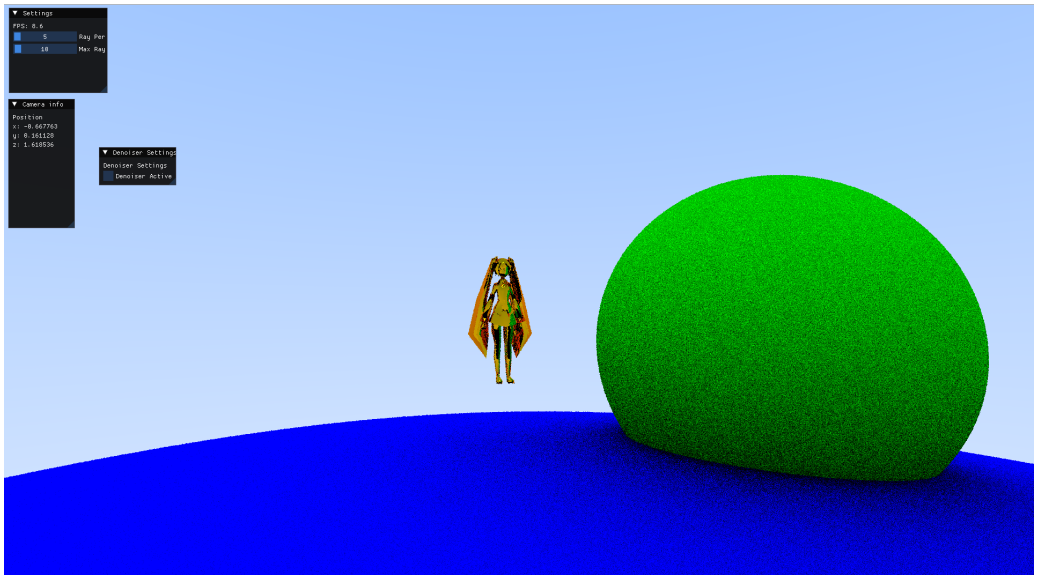


Figure 10: Rendu bruité avec 5 rayons par pixels, angle différent. FPS = 8.6, 3 sphère et 595 triangles (maillage low poly Hatsune Miku)

References

[1] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA, 2017. Association for Computing Machinery.