

## Motivations

Les images illustrent un vaisseau abandonné dans une vaste grotte aux parois fractales, éclairé par des rayons de lumière filtrant à travers des ouvertures naturelles. Ce concept visuel a été choisi pour plusieurs raisons, à la fois thématiques, techniques et symboliques.

L'image incarne la fusion entre nature et technologie. Le contraste entre la grotte organique évoquant la lente érosion naturelle et le vaisseau mécanique, symbole d'un passé technologique révolu, crée une tension poétique entre le vivant et l'artificiel, le temps et la mémoire.

Notre intention est de concevoir un batch renderer capable de simuler l'évolution de cette scène au fil du temps. Les sources lumineuses (soleil, lune, torches, phares, etc.) permettent de visualiser des variations d'éclairage en intensité, angle et teinte, traduisant le cycle jour-nuit ou les saisons.

## Résumé des fonctionnalités mise en oeuvre

Tableau des fonctionnalités			
Thibaut		Wylliam	
Disney BSDF	8	Batch Renderer	1
Spotlight	1	Sphere Walking	4
Denoiser	1	Milieu homogène	4
		Design scène Blender	
10		9	
Bonus devoirs 6			
Total des points: 25			

## Images d'inspirations

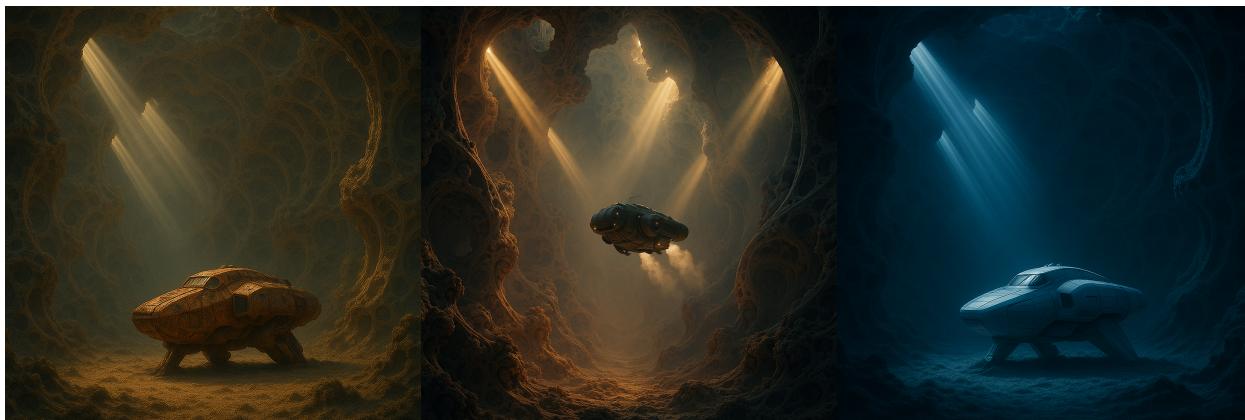


Fig. 1. – Images d'inspirations générées via diffusion

## Image finale

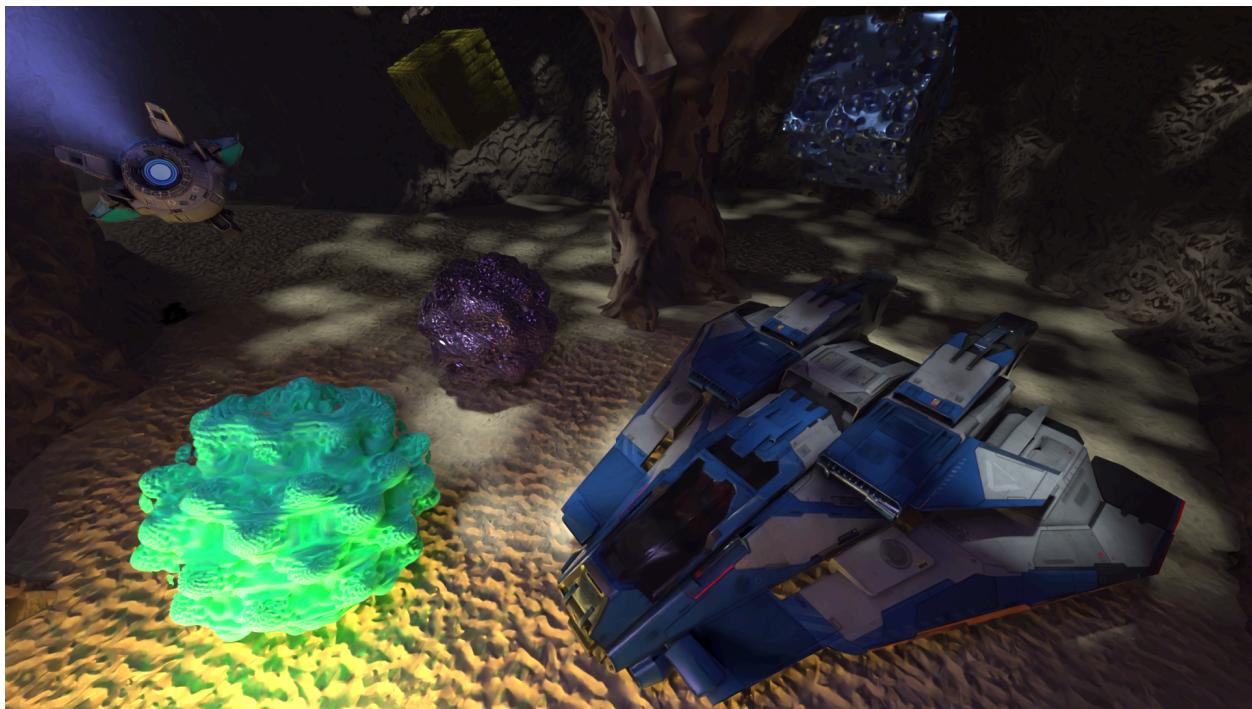


Fig. 2. – Frame 133 de l'animation finale

## Validation des fonctionnalités

### Sphere Walking - Wylliam

Cette section décrit le cœur de la partie « rendu implicite » du projet, c'est-à-dire la manière de représenter et intersector les objets définis par des fonctions de distance signée (SDF) plutôt que par des maillages explicites. L'objectif était de pouvoir manipuler des formes très complexes, en particulier des fractales 3D et des bruits volumétriques, tout en gardant un contrôle raisonnable sur les coûts de rendu. Après avoir étudié différentes stratégies d'intersection (ray marching classique, sphere tracing/sphere walking), nous avons opté pour un algorithme de sphere walking, mieux adapté à la nature des SDF utilisées dans ce projet. Cette section présente d'abord le cadre théorique des SDF, puis discute du ray marching classique, avant de détailler l'algorithme de sphere walking, son implémentation et les difficultés rencontrées.

#### Fonction de distance signée (SDF)

Une fonction de distance signée (Signed Distance Function) associe à chaque point de l'espace une distance à une surface implicite, avec un signe indiquant de quel côté du volume on se trouve.

Une SDF  $f : \mathbb{R}^3 \rightarrow \mathbb{R}$  vérifie que  $f(\mathbf{p}) < 0$  à l'intérieur de l'objet,  $f(\mathbf{p}) = 0$  sur la surface et  $f(\mathbf{p}) > 0$  à l'extérieur.

Dans les cas simples, la SDF s'écrit de manière analytique. Par exemple, pour une sphère de centre  $\mathbf{c}$  et de rayon  $r$ ,

$$f_{\text{sphère}}(\mathbf{p}) = |\mathbf{p} - \mathbf{c}| - r$$

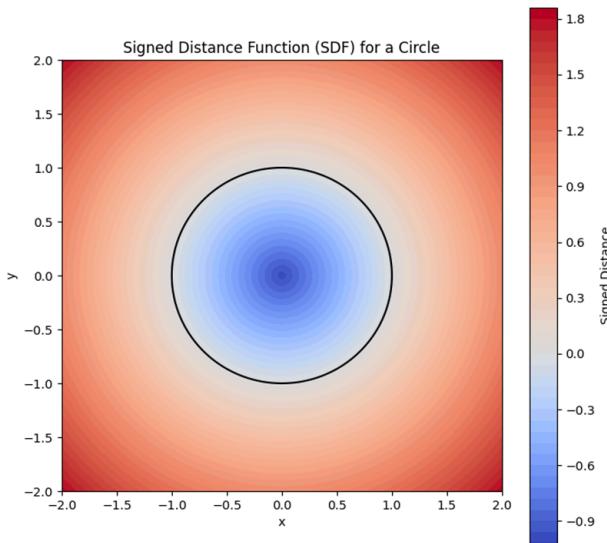


Fig. 3. – Visualisation de la SDF d'un cercle de rayon 1, centré en [0;0]

et pour une boîte demi-étendue alignée sur l'axe  $\mathbf{h} = (h_x, h_y, h_z)$ ,

$$f_{\text{boîte}}(\mathbf{p}) = |\max(|\mathbf{p}| - \mathbf{h}, 0)| + \min(\max(|\mathbf{p}| - \mathbf{h}), 0)$$

où les opérations max/min sont appliquées composante par composante.

L'un des grands avantages des SDF est la possibilité de combiner ces primitives à l'aide d'opérateurs booléens.

$$f_{\cup(\mathbf{p})} = \min(f_A(\mathbf{p}), f_B(\mathbf{p}))$$

$$f_{\cap(\mathbf{p})} = \max(f_A(\mathbf{p}), f_B(\mathbf{p}))$$

$$f_{-(\mathbf{p})} = \max(f_A(\mathbf{p}), -f_B(\mathbf{p}))$$

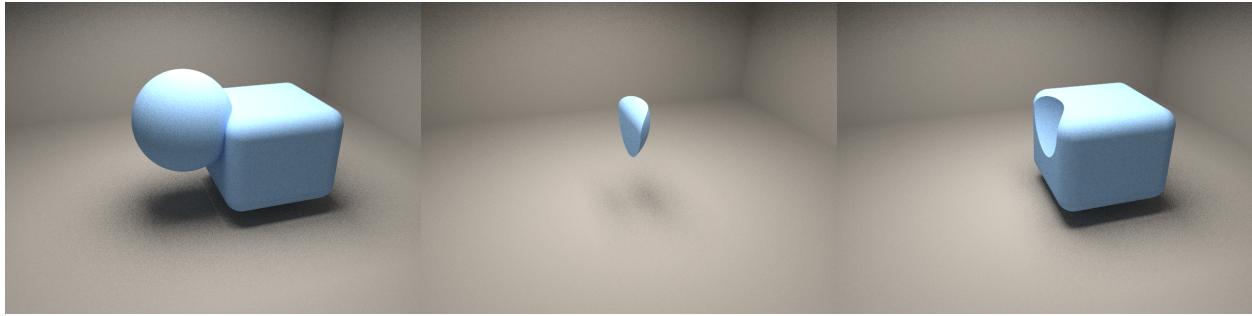


Fig. 4. – Visualisation des opérations d’union, intersection et différence sur une sphère SDF et une boîte SDF

Auxquels des versions « smooth » peuvent être ajouté pour adoucir les transitions entre volumes. À partir de là, il devient possible de construire des structures extrêmement riches : cavités bruitées, tunnels, cristaux, etc.

Enfin, les SDF sont particulièrement adaptées pour représenter des fractales et objets récursifs. Par exemple des évaluateurs comme `sdf_mandelbulb`, `sdf_julia`, ou encore des variantes bruitées (`sdf_fbm_noise`, `sdf_fbm_noise_sphere`) qui appliquent un bruit de type FBM par dessus une primitive de base ont été implémentés. Ces SDF fractales sont coûteuses à évaluer (boucles d’itération, puissances, trigonométrie), mais elles offrent des détails infinis et des silhouettes impossibles à obtenir avec des géométries classiques à résolution raisonnable.

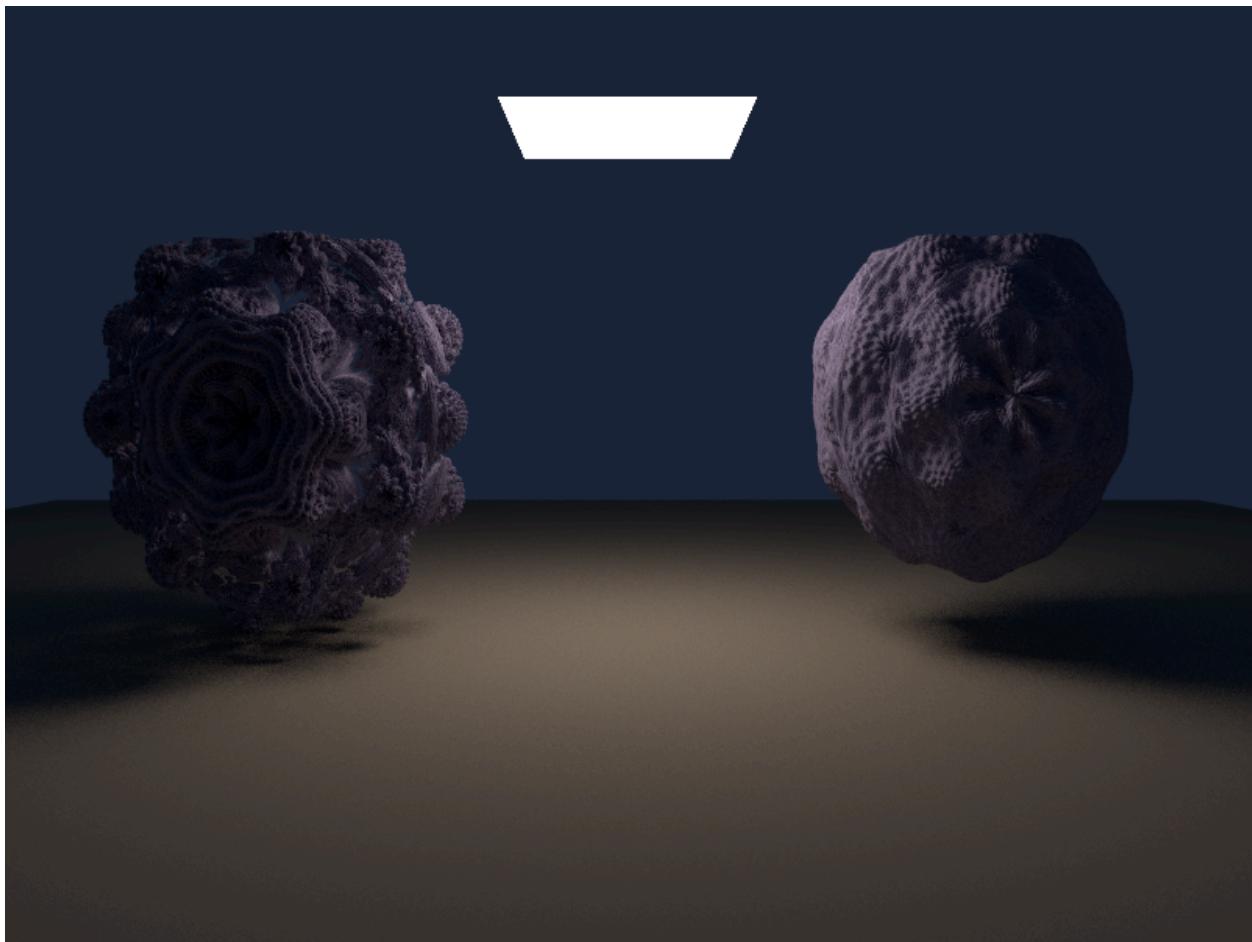


Fig. 5. – Visualisation d’un Mandelbulb à gauche et d’un Juliaset à droite

## Ray Marching

Le ray marching classique consiste à avancer un rayon en ajoutant un pas fixe (ou presque fixe) jusqu'à rencontrer la surface ou dépasser une distance maximale. À chaque itération, la SDF est évaluée au point courant  $p(t) = o + td$  (origine  $o$ , direction  $d$ , paramètre  $t$ ) et si la valeur testée est suffisamment proche de zéro, il y a intersection.

```
float ray_march(vec3 o, vec3 d) {
    float t = t_min;
    for (int i = 0; i < max_steps && t < t_max; ++i) {
        float dist = sdf(o + t * d);
        if (dist < hit_epsilon) return t;
        t += fixed_step; // ou dist_clampé
    }
    return NO_HIT;
}
```

Cette approche est conceptuellement simple, mais elle pose deux problèmes majeurs dans le contexte. Si le pas est trop grand, il y a un risque de « sauter par dessus » des structures fines, en particulier dans les fractales et la caverne. Si le pas est trop petit, le coût d'intersection explose (des centaines voire des milliers d'itérations par rayon), ce qui est incompatible avec des hautes résolutions. En pratique, il faut continuellement ajuster le pas pour chaque scène, sans garantie solide de convergence ni de performance. Compte tenu de la complexité des SDF utilisées, cette approche s'est révélée trop fragile et difficile à calibrer. Il est donc préférable de s'appuyer sur une variante plus robuste, le sphere walking.

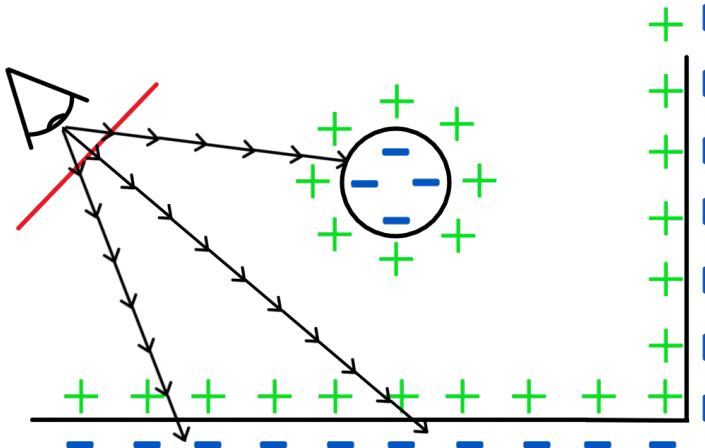


Fig. 6. – Visualisation du ray marching (Garcia, M., n.d.)

## Sphere walking

Le *sphere walking* exploite directement l'information fournie par la SDF : la valeur absolue  $|f(p)|$  donne une borne inférieure sur la distance à la surface la plus proche, tant que la SDF est 1-Lipschitz ou que l'on contrôle la constante de Lipschitz.

Au lieu de marcher avec un pas arbitraire, on avance le rayon par bonds de longueur égale à la distance retournée par la SDF (éventuellement *clampée*), ce qui évite de pénétrer dans l'objet si la SDF respecte ses propriétés.

```
float sphere_walk(vec3 o, vec3 d) {
    float t = t_min;
    for (int i = 0; i < max_steps && t < t_max; ++i) {
        vec3 p = o + t * d;
        float dist = sdf(p);
        if (dist < hit_epsilon) return t;
        float step = clamp(dist, 0.0, step_clamp);
        t += step;
    }
    return NO_HIT;
}
```

Dans le projet, cette logique est paramétrée par des hyper paramètres exposés dans des structures comme `DEFAULT_SDF_SETTINGS` (`hit_epsilon`, `normal_epsilon`, `step_clamp`, `max_steps`, voir par exemple `scripts/convert_blender_sdf.py`). Le champ de distance est évalué via des fonctions spécialisées (`sdf_mandelbulb`, `sdf_julia`, `sdf_fbm_noise`, etc.) et combiné avec des opérateurs booléens pour former des scènes SDF complexes, chaque objet étant de plus limité par une boîte englobante pour réduire le nombre de rayons à tester.

Un autre point important de l'implémentation concerne le calcul des normales. Comme on ne dispose pas toujours d'expression analytique pour le gradient de la SDF (notamment pour les fractales avec FBM), une approximation par différences finies est utilisée :

$$\begin{aligned}\nabla f(\mathbf{p}) \approx & (f(\mathbf{p} + \varepsilon \mathbf{e}_x) - f(\mathbf{p} - \varepsilon \mathbf{e}_x), \\ & f(\mathbf{p} + \varepsilon \mathbf{e}_y) - f(\mathbf{p} - \varepsilon \mathbf{e}_y), \\ & f(\mathbf{p} + \varepsilon \mathbf{e}_z) - f(\mathbf{p} - \varepsilon \mathbf{e}_z))\end{aligned}$$

où  $\varepsilon$  est `normal_epsilon`. Ce gradient normalisé sert ensuite à l'éclairage (BRDF/BSDF Disney, effets de réflexion, etc.) et au calcul de la lumière dans le milieu participatif.

La principale difficulté rencontrée avec le *sphere walking* vient du choix des hyperparamètres `hit_epsilon`, `normal_epsilon`, `step_clamp` et `max_steps`, surtout pour des géométries fines et récursives comme les fractales et les parois très bruitées de la caverne.

**hit\_epsilon** S'il est trop grand, les surfaces apparaissent « épisses » ou floues, et certains détails disparaissent. S'il est trop petit, il faut beaucoup plus d'itérations pour converger, ce qui augmente drastiquement le temps de rendu.

**normal\_epsilon** Contrôle la précision du gradient. Une valeur trop grande introduit du bruit dans les normales (*banding*, facetisation), tandis qu'une valeur trop petite amplifie les erreurs numériques et augmente encore le nombre d'évaluations SDF par *hit*.

**step\_clamp** Nécessaire pour éviter des bonds gigantesques dans les parties très lisses du champ de distance (par exemple loin du fractal), qui peuvent conduire à rater des structures secondaires ajoutées par union/différence. Cependant, le brider trop agressivement revient à se rapprocher d'un *ray marching* à pas presque fixe, donc à perdre l'intérêt du *sphere walking*.

**max\_steps** Doit être suffisamment élevé pour permettre à un rayon de traverser des structures profondes sans interrompre prématurément l'algorithme, mais pas trop élevé pour éviter des explosions de temps de calcul sur les rayons qui, de toute façon, ne touchent rien.

Dans les scènes finales, ces paramètres ont dû être réglés empiriquement, en équilibre entre robustesse, précision géométrique et budget de calcul. Il a fallu accepter que certaines configurations (parois extrêmement fines ou détails fractals à très petite échelle) soient partiellement filtrées pour rester dans un temps de rendu raisonnable.

Cette calibration des  $\epsilon$ s et du pas de marche a constitué une part importante du travail d'implémentation du *sphere walking* dans le projet.

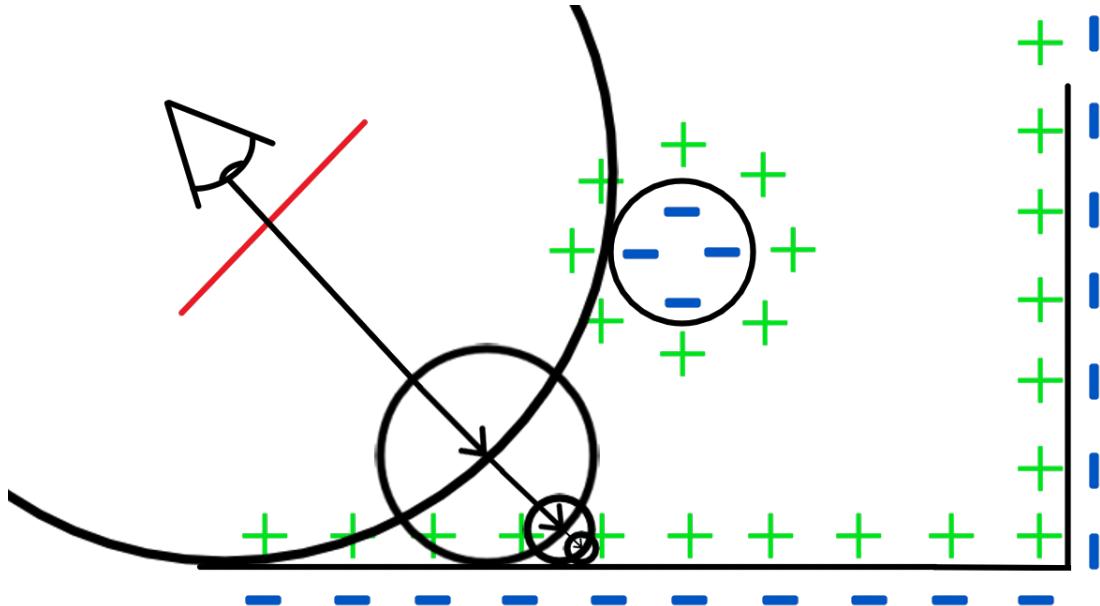


Fig. 7. – Visualisation du sphere walking (Garcia, M., n.d.)

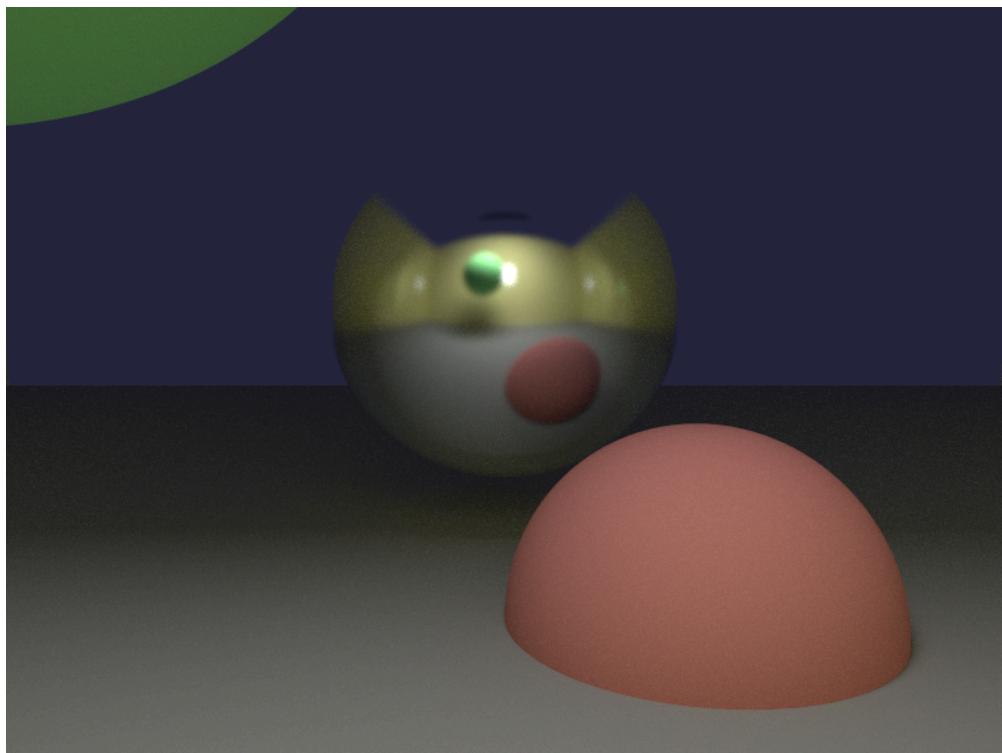


Fig. 8. – Visualisation d'une scène complètement faites avec des SDF

## Milieu homogène - Wylliam

Le traitement du milieu participatif dans ce projet s'inspire directement du chapitre **Volume Scattering** de *PBRT 3e édition* (2018) et en reprend les grandes lignes théoriques et les modèles d'échantillonnage. L'intérieur de la caverne est modélisé comme un milieu homogène : les coefficients d'absorption ( $\sigma_a$ ) et de diffusion ( $\sigma_s$ ) sont constants dans l'espace, et la densité est décrite par un simple scalaire.

Le comportement du milieu est alors entièrement piloté par les coefficients spectraux ( $\sigma_a$ ) et ( $\sigma_s$ ), la densité globale, ainsi qu'une fonction de phase qui décrit la distribution angulaire de la diffusion simple. Dans l'engin, une fonction de phase de Henyey–Greenstein avec paramètre d'asymétrie ( $g$ ) est utilisée, ce qui permet de contrôler la proportion de diffusion avant ( $g > 0$ ) et arrière ( $g < 0$ ), conformément au modèle présenté dans PBRT.

La transmittance le long d'un segment de longueur ( $d$ ) est donnée par la loi de Beer–Lambert :

$$T(d) = \exp(-\sigma_t d)$$

où  $\sigma_t = \sigma_a + \sigma_s$ , ce qui régit à la fois l'extinction de la lumière directe et la pondération des contributions volumétriques.

Côté implémentation, le milieu est décrit dans les fichiers de scène via un bloc `medium` de type homogène, avec des paramètres par défaut définis dans `scripts/convert_blender_sdf.py`. Lors du chargement de la scène (`json_to_medium` dans `src/json.rs`), ces paramètres sont convertis en une instance de `HomogeneousMedium` (`src/medium.rs`) qui implémente le trait `Medium`.

La transmittance est calculée par `transmittance_from_sigma_t`, qui applique la loi exponentielle composante par composante sur les canaux RGB. L'échantillonnage des événements de diffusion dans le volume suit également la formulation de PBRT : le coefficient d'extinction  $\sigma_t$  est considéré et une distance libre est tirée selon une loi exponentielle  $-\frac{\ln(u)}{\sigma_t}$  le long du rayon, avec une sélection de canal pour la version spectrale. Si la distance échantillonnée est inférieure à la distance de la prochaine interaction géométrique, il y a un scatter volumique, sinon un simple terme de transmittance est retourné.

L'intégrateur `hybrid_vol_path_mis` exploite ensuite cette interface. Il intercale les événements de diffusion volumique entre les interactions de surface, évalue la fonction de phase (`PhaseFunction::HenyeyGreenstein::phase_func` et `sample_p`) et combine les contributions de lumière directe et indirecte dans le volume via MIS, aussi bien pour les points de surface que pour les points de scatter dans le milieu.

Sur le plan pratique, la principale difficulté a été de régler des valeurs de  $\sigma_a$ ,  $\sigma_s$ , de densité et de  $g$  qui produisent une atmosphère crédible dans la caverne. Des paramètres trop diffusants ou une densité trop élevée peuvent saturer la scène, détruire le contraste et multiplier les rebonds volumétriques, ce qui augmente fortement la variance et les temps de convergence. À l'inverse, un milieu trop peu dense donne une image presque « vide » où les effets de *god rays* et de volumétrie deviennent à peine perceptibles, ce qui justifie à peine le coût de l'intégrateur volumique.

Un autre point délicat a été la gestion de la transmittance dans le cadre de l'intégrateur hybride : comme les rayons traversent à la fois des surfaces SDF et un milieu homogène, il a fallu vérifier que la transmittance était appliquée de manière cohérente le long des segments (*shadow rays, next event estimation* depuis le volume, etc.), conformément à la théorie de PBRT. En pratique, plusieurs itérations ont été nécessaires pour ajuster ces paramètres, comparer les résultats visuels et s'assurer que le rendu volumique restait lisible, convergent et esthétiquement satisfaisant.



Fig. 9. – Différentes combinaisons de paramètres d'absorption et diffusion de la lumière dans un milieu homogène

## Rendu par lots & Design scène Blender - Wylliam

Le rendu d'une seule image en  $1920 \times 1080$  à 512 échantillons par pixel (SPP) prenait environ 1 h 30 sur une machine de développement, et ce malgré l'utilisation de structures accélératrices pour les maillages et les SDF. À ce rythme, produire une séquence de 250 images représentait environ 375 heures de calcul continu, ce qui est incompatible avec un cycle d'itération raisonnable et rend la fonctionnalité d'animation pratiquement inutilisable en local. Pour conserver cette fonctionnalité tout en respectant les contraintes de temps, une stratégie de rendu par lots a été conçue pour être capable d'exploiter du calcul parallèle massif. Cette section décrit la mise en place de ce pipeline sur superordinateur, ainsi que les compromis effectués entre coût de calcul, qualité d'image et temps de rendu.

### Rendu par lots

En tant qu'étudiant à la maîtrise, Wylliam a accès aux superordinateurs de Compute Canada. Le pipeline de rendu a donc été adapté pour fonctionner sur cette infrastructure en s'appuyant sur le gestionnaire de tâches Slurm. Concrètement, chaque image de la séquence est décrite par un fichier de scène individuel (par exemple `ctestXXXXX.json` exporté depuis Blender), et un script de soumission construit un « job array » Slurm (`SBATCH --array=1-250`) dans lequel chaque tâche rend une image distincte.

Lors de l'exécution sur Narval, le script `render_array.sh` charge les modules nécessaires, configure l'environnement pour la bibliothèque de débruitage OIDN, se place dans le répertoire du projet et appelle le renderer Rust (`cargo run --release --example=render --features oidn`) avec les paramètres appropriés (nombre de SPP, fichier d'accélération BVH, graine, chemins d'entrée et de sortie). Ce découpage image-par-image permet à Slurm de distribuer la charge sur de multiples nœuds en parallèle et de réduire le temps de rendu global de plusieurs centaines d'heures à quelques heures.

La mise en place de ce pipeline a nécessité d'apprendre à utiliser Slurm au-delà des commandes de base : gestion des tableaux de tâches, choix du nombre de coeurs par tâche (`--cpus-per-task`), configuration de la mémoire, des chemins de caches GPU (OptiX) et de la durée maximale (`--time`) pour minimiser les échecs de jobs. Il a également fallu comprendre et ajuster les compromis entre coût de calcul, temps d'attente en file, temps de calcul réel et qualité des résultats. Par exemple, augmenter fortement le nombre de coeurs par tâche réduit le temps de rendu par image, mais peut diminuer la priorité du job dans la file et retarder son démarrage et inversement, lancer un grand nombre de tâches légères en parallèle augmente le débit global, mais consomme davantage de slots dans la grappe. Ces essais ont d'abord été réalisés sur le superordinateur Nibi de l'Université de Waterloo, où la priorité d'Adrien a temporairement subi les conséquences de nos expérimentations peu optimisées, avant d'aboutir à une configuration plus raisonnable sur Narval.

Pour valider le fonctionnement du pipeline sans mobiliser inutilement des centaines d'heures de calcul, nous avons utilisé les sessions interactives sur la grappe. Wylliam a ainsi pu tester les scripts Slurm et les paramètres de rendu en conditions réelles, mais avec une configuration fortement réduite : résolution à 10 % de la taille finale et seulement 1 SPP. Ces tests interactifs, très rapides, ont permis de vérifier la cohérence des chemins de fichiers (entrées JSON, sorties EXR), la bonne initialisation des modules et des variables d'environnement, ainsi que la stabilité du renderer sur l'infrastructure HPC. Une fois ces validations effectuées, il a été possible d'augmenter progressivement la résolution et le nombre de SPP jusqu'aux valeurs de production, en lançant cette fois le rendu complet de la séquence sous forme de jobs batch parallélisés sur Narval.



Fig. 10. – Example d'un frame généré à 10% de résolution et 1 SPP

## Design scène Blender

### Géométrie de la caverne dans Blender

Les limitations du *sphere walking* dans la caverne (détaillées dans la section sur le *sphere tracing*) ont rapidement montré qu'il serait difficile de construire l'intégralité de l'environnement uniquement à partir de SDF animées, tout en gardant un contrôle artistique précis sur les silhouettes, la navigabilité de la caméra et le temps de rendu.

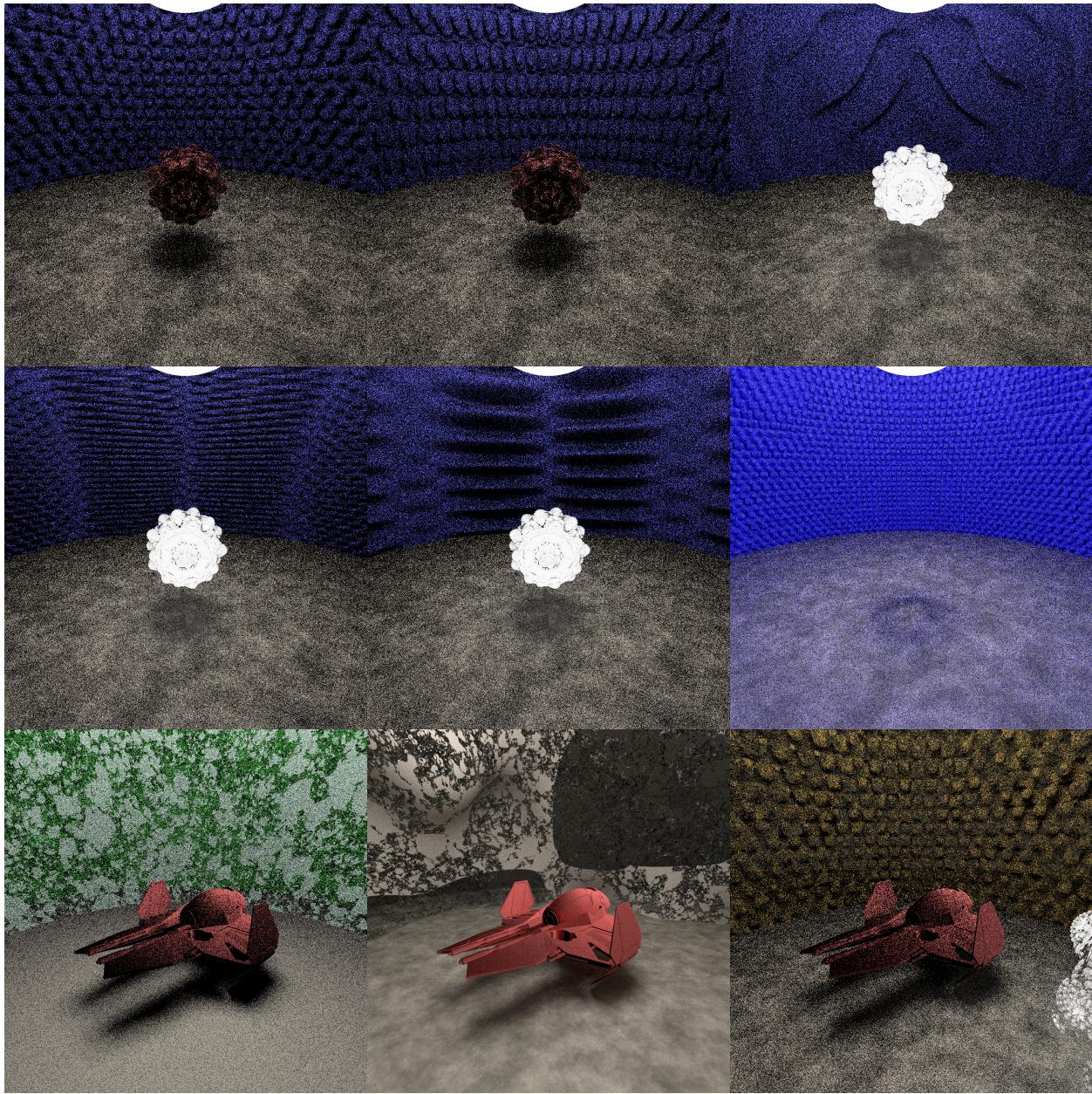


Fig. 11. – Multiples itérations du cavernes de fractales SDF

La solution retenue a donc été de générer la topologie de base de la caverne dans Blender à l'aide d'un script procédural dédié, plutôt que de tout laisser au moteur de rendu. Le script construit d'abord un beigne de sphères, chacune perturbée par un léger *jitter*, puis ajoute un tunnel secondaire à partir d'une des sphères du cercle, ce qui donne une topologie de type anneau avec embranchement.

Toutes ces sphères sont ensuite fusionnées en un seul objet, *voxel-remeshé* et lissé, puis agrémenté de deux modificateurs de *Displacement* avec une texture *Musgrave* (un pour les grandes formes rocheuses, un pour les détails plus « spiky »). Le même script génère un plan de sol subdivisé et bruité pour donner un effet réaliste au niveau du sol.

### Visualisation des objets fractals

Les objets véritablement fractals (Mandelbulb, Julia, FBM bruité, etc.) restent définis analytiquement dans le moteur via des SDF. Cependant, pour les placer et les composer artistiquement dans la scène, il est indispensable de pouvoir les visualiser sous forme de maillage dans l'éditeur.

Pour cela, nous avons mis en place un pipeline de prévisualisation utilisant *marching cubes*. Les scripts `scripts/sdf_extraction.py` et `scripts/batch_sdf_jobs.py` échantillonnent le champ de distance sur une grille 3D, puis appliquent `skimage.measure.marching_cubes` pour extraire un maillage triangulé de l'isosurface SDF.

Le résultat est exporté au format `.obj` (par exemple à partir de presets décrits dans `jobs_preview.json` ou `jobs_extensive.json`), ce qui permet d'importer ces fractales dans Blender comme des maillages standards. L'éditeur de scène peut ainsi afficher les fractales, les manipuler, les dupliquer, les faire interagir avec la géométrie de la grotte, tout en sachant que, au moment du rendu final, ces *proxies* polygonaux seront remplacés par leurs SDF analytiques beaucoup plus précises.

Il est possible de voir dans la figure suivante que l'engine de rendu interne à Blender, Cycles, est incapable de produire un joli rendu à cause de la complexité géométrique des maillages extraits par *marching cubes*.

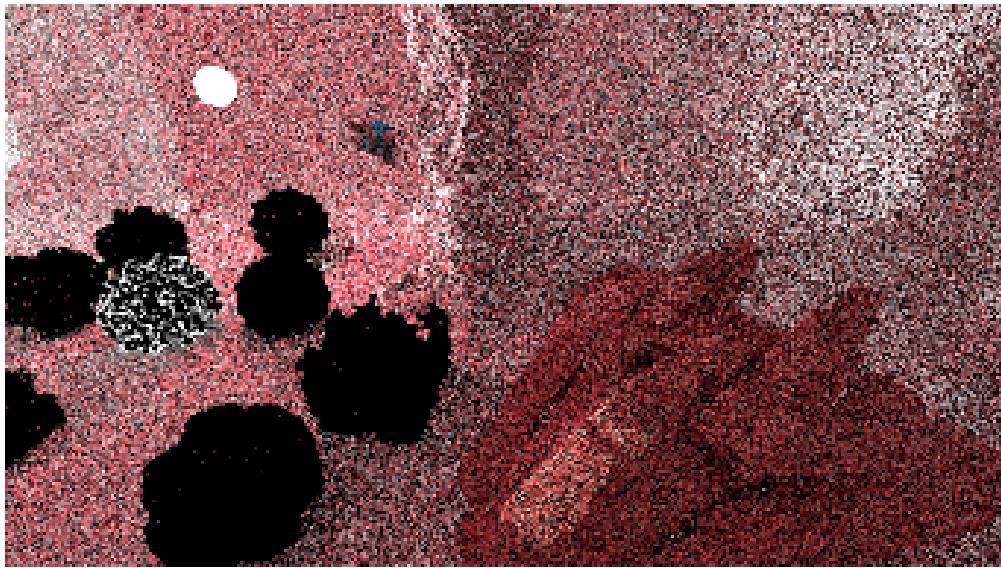


Fig. 12. – Limitations de Blender Cycles face à la complexité géométrique des maillages extraits

### Conversion de l'export Blender vers la scène SDF

L'export JSON brut de Blender tend à dupliquer la géométrie pour chaque *frame* (`meshes` dans `meshes/<frame>/...`), ce qui n'est ni nécessaire ni souhaitable. Pour éviter ces duplications et réinjecter les paramètres physiques, un script de *parsing* a été conçu.

Le script `convert_blender_sdf.py` passe en post-traitement sur les fichiers de scène `.json` exportés. Ce script parcourt toutes les formes de type `mesh`, normalise leurs chemins de fichiers (`flatten_mesh_path` pour retirer le sous-répertoire de `frame`) et, lorsqu'un `mesh` est identifié comme un *proxy* de fractale (via son nom et les jobs décrits dans `jobs_*.json`), il le remplace par un objet SDF analytique dans le champ `sdf_objects`. Pour chaque SDF, le script recalcule une boîte englobante locale et mondiale (`base_bounds`, `transform_bounds`, `world_bounds`) à partir de la matrice  $4 \times 4$  Blender, puis dérive des paramètres de marche adaptés ( $\varepsilon$  de `hit` et de normale) en fonction de l'échelle.

En parallèle, le convertisseur nettoie et enrichit la description de la scène : il simplifie les matériaux imbriqués, dé-duplique les matériaux *diffuse*, crée des *spotlights* physiques à partir de *meshes* « *spot* » (conversion en type: "spotlight" en utilisant la normale du quad comme direction), injecte un milieu participatif homogène, un intégrateur de rendu volumétrique adapté et des matériaux Disney/Principled prédéfinis pour les fractales et le vaisseau (*principled\_bsdf* pour les cristaux, le métal et les vitres). Le script ajuste également certains paramètres de caméra (comme le champ de vue *vfov*) pour rester cohérent entre Blender et notre moteur. En pratique, cela crée un pont propre entre Blender (*meshes*, matériaux *Principled*, animations) et le moteur interne (SDF analytiques, intégrateur volumétrique, pipeline de débruitage).

### Animation et flux artistique

Sur cette base technique, une grande partie du travail a été consacrée à l'apprentissage et au design des animations tel que la trajectoire du droïde à travers la caverne, les mouvements de caméra, la variations de lumière, et la modulation de la densité/du contraste du milieu participatif. Les animations elles-mêmes sont créées dans Blender (courbes de Bézier, *F-curves*, modificateurs de bruit, etc.), puis *baked* dans les fichiers de scène exportés sous forme de matrices de transformation par frame. Le script *convert\_blender\_sdf.py* se contente de relayer fidèlement ces transformations vers les objets SDF et les sources de lumière dans notre format de scène, de manière à ce que le rendu final reproduise exactement le flux artistique conçu dans l'éditeur. Ce découplage a permis de conserver une grande liberté créative sur les mouvements, les rythmes de la lumière et l'ambiance générale de la séquence, tout en bénéficiant des avantages numériques des SDF analytiques et des intégrateurs volumétriques maison.

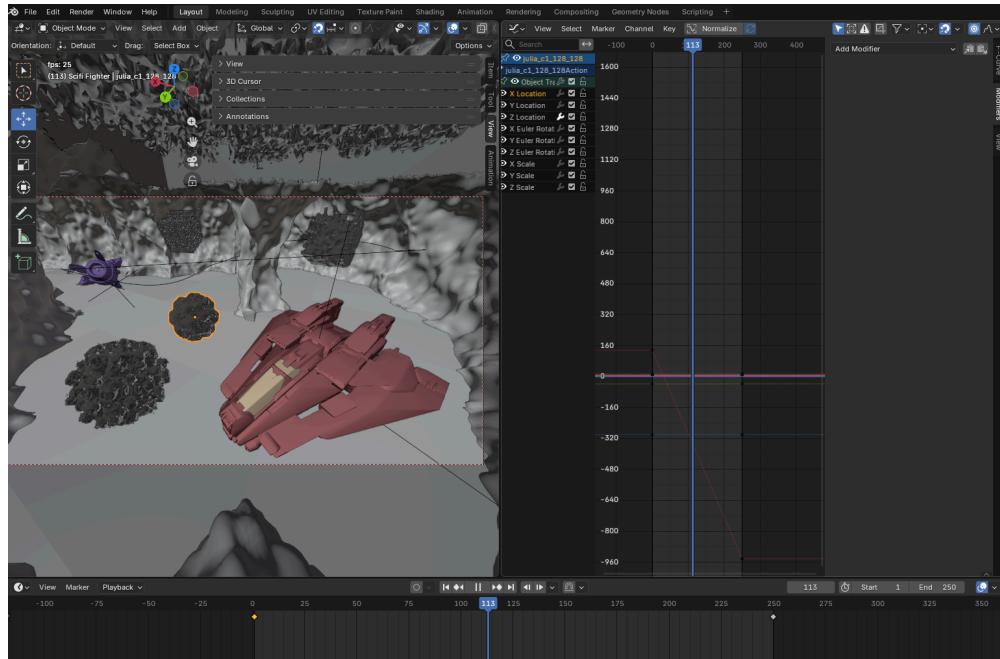


Fig. 13. – Capture d'écran de l'interface d'animation

## BSDF Disney - Thibaut

L'implémentation de la BSDF Disney a été basée sur le document du devoir 1 du cours CSE 272 à l'université de Californie à San Diego, donné par Tzu-Mao Li. Ce document présente une version du matériau que les étudiants du cours doivent implémenter dans leur engin de rendu lajolla. Le matériau est divisé en cinq parties: *diffuse*, *metal*, *clearcoat*, *glass*, et *sheen*. L'implémentation a donc été faite de façon similaire dans notre engin, à l'exception que toutes les parties sont contenues dans le même fichier, `principled_bsdf.rs`.

Pour éviter de faire plusieurs requêtes de textures pour les différents paramètres du matériau, une structure additionnelle est définie: `PBsdSample`, qui contient des valeurs réelles pour chacun des paramètres. Dans les fonctions d'implémentation du trait `Material`, un objet de type `PBsdSample` est créé et passé aux diverses fonctions utilisées dans l'évaluation du matériau.

### **Diffuse (diffuse\_sample, f\_diffuse, et diffuse\_pdf)?**

La composante diffuse du matériau utilise simplement un échantillonnage cosinus-hémisphère avec la PDF associée. L'évaluation est légèrement plus complexe, vu qu'il y a une combinaison d'un matériau diffus de base et d'un faux effet de *subsurface*. Le diffus de base est évalué selon la formule:

$$f_{\text{diffuse,base}} = \frac{\text{base\_color}}{\pi} F_D(\omega_i) F_D(\omega_o) |\omega_i \cdot n|$$

Où

$$\begin{aligned} F_D(\omega) &= 1 + (F_{D90} - 1)(1 - |\omega \cdot n|)^5 \\ F_{D90} &= 2 \cdot \text{roughness} \cdot |h \cdot \omega_i|^2 + \frac{1}{2} \end{aligned}$$

Pour le faux *subsurface*, on a:

$$f_{\text{diffuse,subsurface}} = 1.25 \cdot \frac{\text{base\_color}}{\pi} \left( F_{SS}(\omega_o) F_{SS}(\omega_i) \left( \frac{1}{|\omega_o \cdot n| + |\omega_i \cdot n|} - \frac{1}{2} \right) + \frac{1}{2} \right) |\omega_i \cdot n|$$

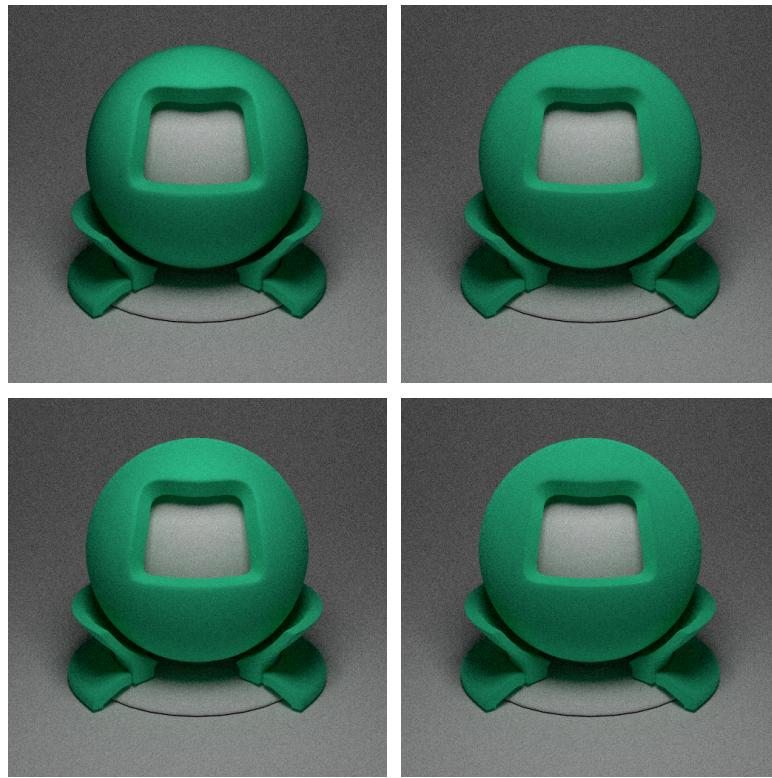
Où

$$\begin{aligned} F_{SS}(\omega) &= 1 + (F_{SS90} - 1)(1 - |\omega \cdot n|)^5 \\ F_{SS90} &= \text{roughness} \cdot |h \cdot \omega_i|^2 \end{aligned}$$

Finalement, on combine les deux:

$$f_{\text{diffuse}} = (1 - \text{subsurface}) \cdot f_{\text{diffuse,base}} + \text{subsurface} \cdot f_{\text{diffuse,subsurface}}$$

Rendus avec (*roughness*, *subsurface*) =  $\begin{pmatrix} (0,0) & (0,1) \\ (1,0) & (1,1) \end{pmatrix}$ :



#### **Metal (metal\_sample, f\_metal, metal\_pdf)**

La composante métallique est plus complexe que le diffus. L'évaluation du matériau en elle-même est plutôt simple, mais l'échantillonnage suggéré suit une distribution GGX par Heitz. L'échantillonnage génère d'abord un *half-vector* perturbé selon la distribution GGX, puis l'utilise pour réfléchir la direction incidente.

L'implémentation de l'échantillonnage GGX, utilisé aussi pour la composante *glass*, est basée sur la version de lajolla<sup>1</sup>, mais adapté en anisotropique.

La PDF du *metal* est définie comme suit:

$$\text{PDF}_{\text{metal}} = \frac{D_m G(\omega_o)}{4 |n \cdot \omega_o|}$$

Et l'évaluation du matériau:

$$f_{\text{metal}} = \frac{F_m D_m G_m}{4 |n \cdot \omega_o|}$$

Où:

$$\begin{aligned} F_m &= C_0 + (1 - C_0)(1 - |h \cdot \omega_i|)^5 \\ C_0 &= \text{specular} \cdot R_0(\eta)(1 - \text{metallic})K_s + \text{metallic} \cdot \text{base_color} \\ K_s &= (1 - \text{specular\_tint}) + \text{specular\_tint} \cdot C_{\text{tint}} \\ R(\eta) &= \frac{(\eta - 1)^2}{(\eta + 1)^2} \end{aligned}$$

---

<sup>1</sup>[https://github.com/BachiLi/lajolla\\_public/blob/main/src/microfacet.h#L85](https://github.com/BachiLi/lajolla_public/blob/main/src/microfacet.h#L85)

$$C_{\text{tint}} = \begin{cases} \text{base\_color} / \text{luminance}(\text{base\_color}) & \text{if } \text{luminance}(\text{base\_color}) > 0 \\ 1 & \text{otherwise} \end{cases}$$

$$D_m = \frac{1}{\pi \alpha_x \alpha_y \left( \frac{h_x^2}{\alpha_x^2} + \frac{h_y^2}{\alpha_y^2} + h_z^2 \right)^2}$$

$$G_m = G(\omega_i)G(\omega_o)$$

$$G(\omega) = \frac{1}{1 + \Lambda(\omega)}$$

$$\Lambda(\omega) = \frac{1}{2} \left( \sqrt{1 + \frac{(\omega_x \cdot \alpha_x)^2 + (\omega_y \cdot \alpha_y)^2}{\omega_z^2}} - 1 \right)$$

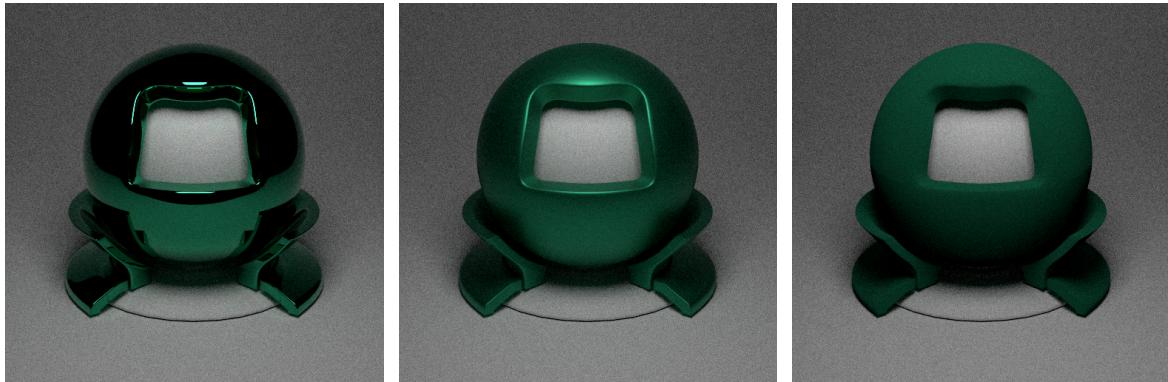
Les paramètres  $\alpha$  anisotropiques sont définis comme:

$$\alpha_x = \max(0.0001, \text{roughness}^2 / \text{aspect})$$

$$\alpha_y = \max(0.0001, \text{roughness}^2 \cdot \text{aspect})$$

$$\text{aspect} = \sqrt{1 - 0.9 \cdot \text{anisotropic}}$$

Ceci produit les résultats suivants, avec metallic = 0.5, specular = 0.5, roughness = (0.0 0.5 1.0):



### **Clearcoat (clearcoat\_sample, f\_clearcoat, clearcoat\_pdf)**

La composante *clearcoat* a aussi beaucoup de paramètres, mais n'est pas si compliquée en soi. L'échantillonnage proposé dans la référence est de choisir un *half-vector* perturbé proportionnel au terme  $D_c$  pour réfléchir la direction incidente. La méthode d'échantillonnage est décrite directement dans la référence comme suit:

$$\cos(h_{\text{elevation}}) = \sqrt{\frac{1 - (\alpha_g^2)^{1-s_x}}{1 - \alpha_g^2}}$$

$$h_{\text{azimuth}} = 2\pi s_y$$

$$h = \begin{pmatrix} \sin(h_{\text{elevation}}) \cos(h_{\text{azimuth}}) \\ \sin(h_{\text{elevation}}) \sin(h_{\text{azimuth}}) \\ \cos(h_{\text{elevation}}) \end{pmatrix}$$

Où

$$\alpha_g = (1 - \text{clearcoat\_gloss}) \cdot 0.1 + \text{clearcoat\_gloss} \cdot 0.001$$

La PDF est alors de:

$$\text{PDF}_{\text{metal}} = \frac{D_c}{4} \frac{|n \cdot h|}{|h \cdot \omega_i|}$$

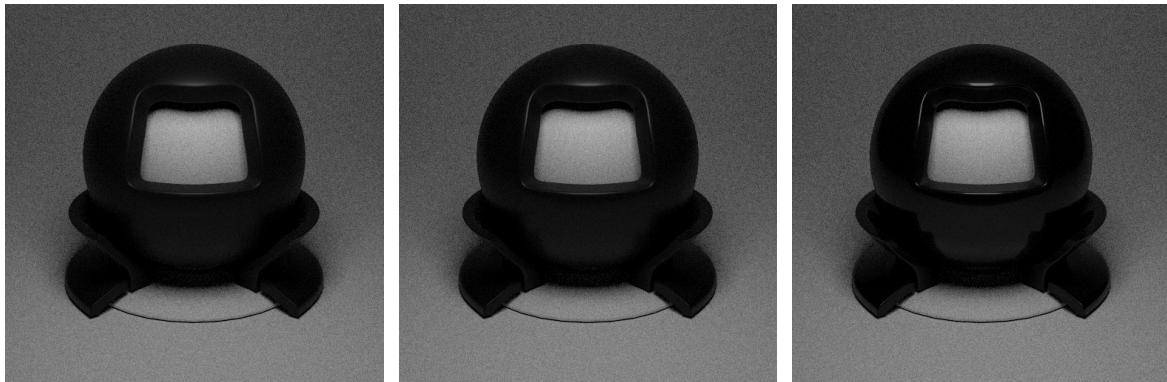
L'évaluation du matériau se fait avec la formule:

$$f_{\text{clearcoat}} = \frac{F_c D_c G_c}{4 |n \cdot \omega_o|}$$

Où

$$\begin{aligned} F_c &= R_0(1.5) + (1 - R_0(1.5))(1 - |h \cdot \omega_i|)^5 \\ D_c &= \frac{\alpha_g^2 - 1}{\pi \ln(\alpha_g^2)(1 + (\alpha_g^2 - 1)h_z^2)} \\ G_c &= G_c(\omega_o)G_c(\omega_i) \\ G_c(\omega) &= \frac{1}{1 + \Lambda_c(\omega)} \\ \Lambda_c(\omega) &= \frac{1}{2} \left( \sqrt{1 + \frac{(\omega_x \cdot 0.25)^2 + (\omega_y \cdot 0.25)^2}{\omega_z^2}} - 1 \right) \end{aligned}$$

Résultats du *clearcoat*, avec clearcoat\_gloss = (0.0 0.5 1.0):



### **Glass (*glass\_sample*, *f\_glass*, *glass\_pdf*)**

La composante diélectrique du matériau est possiblement la plus complexe, vu qu'il faut gérer et le cas de la réflexion, et le cas de la réfraction. Le document de référence mentionne de réutiliser l'échantillonnage du matériau *roughdielectric* de lajolla, en l'adaptant pour l'anisotropie.

L'échantillonnage génère d'abord un *half-vector* perturbé comme dans le *metal*, suivant la distribution GGX. Ensuite, un coefficient de Fresnel est calculé selon la formule:

$$\begin{aligned} F &= \begin{cases} 1 & \text{if } w < 0 \\ F' & \text{otherwise} \end{cases} \\ w &= 1 - \frac{1 - h \cdot \omega_o}{\eta^2} \\ F' &= \frac{1}{2} \left( \left( \frac{|h \cdot \omega_o| - \eta \cdot \sqrt{w}}{|h \cdot \omega_o| + \eta \cdot \sqrt{w}} \right)^2 + \left( \frac{\eta \cdot |h \cdot \omega_o| - \sqrt{w}}{\eta \cdot |h \cdot \omega_o| + \sqrt{w}} \right)^2 \right) \end{aligned}$$

Dans le cas où  $w < 0$ , on a de la réflexion totale interne, d'où le coefficient de Fresnel forcé à 1. Ensuite, on compare un nombre aléatoire avec ce coefficient pour décider entre réflexion et réfraction. Pour la réflexion, on réfléchit simplement la direction incidente autour du *half-vector*. Pour la réfraction, il faut calculer la nouvelle direction à partir de l'indice de réfraction  $\eta$ .

La PDF est définie:

$$\text{PDF}_{\text{glass}} = \begin{cases} \frac{F \cdot D_m \cdot G(\omega_i)}{4 |n \cdot \omega_o|} & \text{if } (n \cdot \omega_o)(n \cdot \omega_i) > 0 \\ (1 - F) \cdot D_m \cdot G(\omega_i) \left| \frac{\eta^2(h \cdot \omega_i)(h \cdot \omega_o)}{(h \cdot \omega_o + \eta(h \cdot \omega_i))^2(n \cdot \omega_o)} \right| & \text{otherwise} \end{cases}$$

Où  $F$  est le coefficient de Fresnel, et  $D_m$  et  $G$  sont les mêmes termes que pour le *metal*.

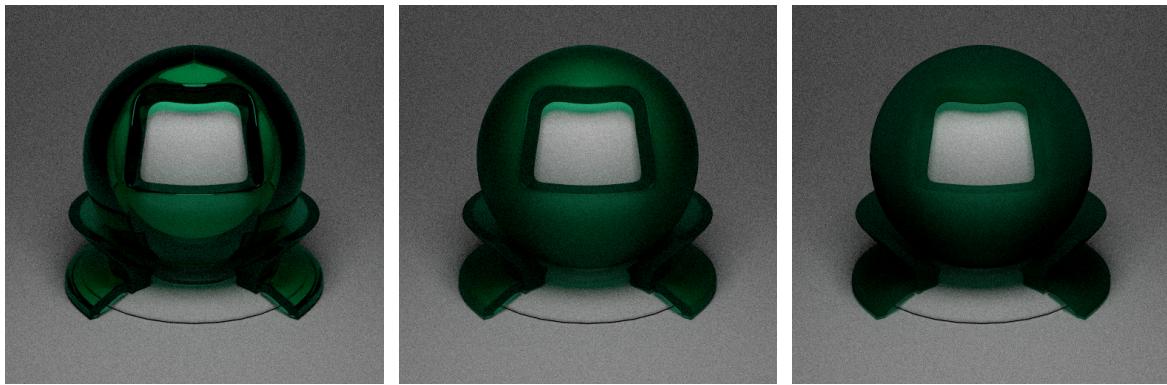
Finalement, l'évaluation du matériau en lui-même est calculée avec:

$$f_{\text{glass}} = \begin{cases} \frac{\text{base_color } F_g D_m G_m}{4 |n \cdot \omega_o|} & \text{if } (n \cdot \omega_o)(n \cdot \omega_i) > 0 \\ \frac{\sqrt{\text{base_color}}(1 - F_g) D_m G_m |(h \cdot \omega_i)(h \cdot \omega_o)|}{|n \cdot \omega_o| (h \cdot \omega_o + \eta(h \cdot \omega_i))^2} & \text{otherwise} \end{cases}$$

Où  $D_m$  et  $G_m$  sont encore une fois les mêmes que pour le métal, et  $F_g$  est un coefficient de Fresnel calculé avec la valeur réelle de  $\omega_i$ , contrairement au  $F'$  de ci-dessus.

$$F_g = \frac{1}{2} \left( \left( \frac{|h \cdot \omega_o| - \eta \cdot |h \cdot \omega_i|}{|h \cdot \omega_o| + \eta \cdot |h \cdot \omega_i|} \right)^2 + \left( \frac{\eta \cdot |h \cdot \omega_o| - |h \cdot \omega_i|}{\eta \cdot |h \cdot \omega_o| + |h \cdot \omega_i|} \right)^2 \right)$$

Résultats du *glass*, avec roughness = (0.0 0.5 1.0):



### ***Sheen (f\_sheen)***

L'échantillonnage du *sheen* et par conséquent, sa PDF, ne sont pas utilisés dans le mélange final du matériau puisque sa contribution est généralement plutôt faible. Pour les tests ci-dessous, un échantillonnage et une PDF cosinus-hémisphère sont utilisés.

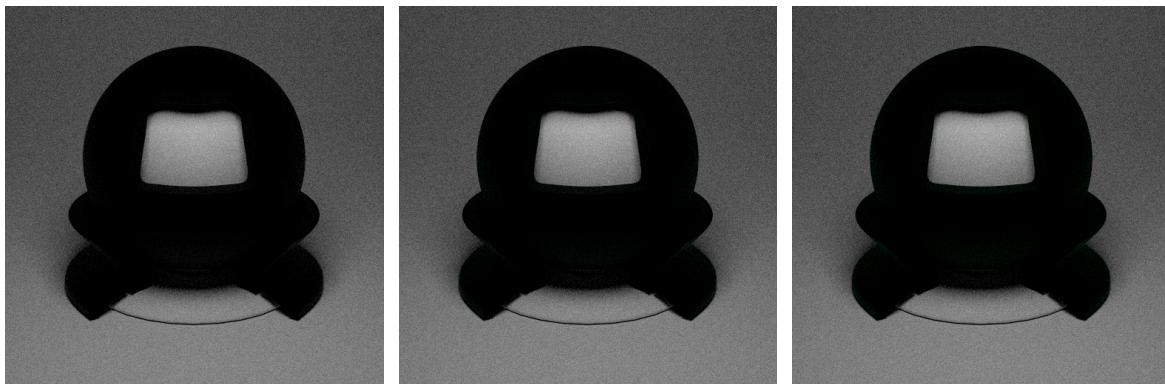
L'évaluation du *sheen* est assez simple:

$$f_{\text{sheen}} = C_{\text{sheen}} (1 - |h \cdot \omega_i|)^5 |n \cdot \omega_i|$$

$$C_{\text{sheen}} = (1 - \text{sheen\_tint}) + \text{sheen\_tint} \cdot C_{\text{tint}}$$

Avec  $C_{\text{tint}}$  défini dans la section *metal*.

Le résultat, avec  $\text{sheen\_tint} = (0.0 \ 0.5 \ 1.0)$ :



### Mélangé

Finalement, les cinq composantes sont mélangées pour obtenir le matériau final. Pour l'échantillonnage et l'évaluation des PDF, les poids suivants sont utilisés:

$$\begin{aligned}\text{diffuse\_weight} &= (1 - \text{metallic})(1 - \text{specular\_transmission}) \\ \text{metal\_weight} &= \text{metallic} \cdot (1 - \text{specular\_transmision}) \\ \text{glass\_weight} &= (1 - \text{metallic}) \cdot \text{specular\_transmission} \\ \text{clearcoat\_weight} &= 0.25 \cdot \text{clearcoat}\end{aligned}$$

Si la direction incidente provient de sous la surface, alors on n'utilise que l'échantillonnage du verre.

Pour l'évaluation du matériau, les mêmes poids sont utilisés, en ajoutant celui du *sheen*:

$$\text{sheen\_weight} = (1 - \text{metallic}) \cdot \text{sheen}$$

Ici aussi, si la direction incidente provient de sous la surface, on ignore les autres composantes que le diélectrique. Par contre, celui-ci est tout de même multiplié par son poids.

### Validation

Pour valider les résultats, la méthode principale était de comparer des rendus avec ceux de Mitsuba en utilisant une définition de matériau similaire. Généralement, c'était une bonne technique, mais cela devenait plus difficile dans certains cas particuliers.

Une autre technique utilisée était d'utiliser l'exécutable test du devoir 2. Ceci a révélé plusieurs problèmes, principalement dans les échantillonnages et PDFs du *metal*, *clearcoat*, et *glass*. En essayant de résoudre ces problèmes, le résultat final s'est empiré : le matériau devenait proportionnellement plus éclairé avec le nombre d'échantillons par pixel. Il est possible que cette version était en réalité plus proche du calcul correct, à l'exception d'un facteur de correction produisant cet effet d'éclaircissement. Tout de même, la version soumise est celle produisant des résultats plus stables, malgré un clair ajout d'énergie aux rayons lumineux avec certaines configurations de paramètres.

## Spotlight - Thibaut

Le spotlight est implémenté en deux parties: une Shape, SpotLight et un Material, DiffuseEmitSpotLight. Le premier contient la position et la direction de la lumière, et le deuxième, la couleur de la radiance ainsi qu'un paramètre focus d'ouverture ou d'angle.

Vu qu'il s'agit d'une source ponctuelle, sans aire ni volume, la forme SpotLight ne peut pas être touchée via hit. Pour l'échantillonnage du direct, il n'y a donc pas non plus de facteur aléatoire, le point échantillonné est toujours équivalent à la position de la lumière, avec la normale égale à sa direction. La PDF est donc de 1.

Pour évaluer l'émission du matériau DiffuseEmitSpotLight, on calcule simplement le produit scalaire entre la normale (donc la direction du spotlight) et la direction incidente à la puissance du carré du paramètre focus, multiplié par la radiance. C'est-à-dire:

$$\text{Emission}_{\text{spotlight}} = \begin{cases} 0 & \text{if } \omega_o \cdot n < 0.0 \\ \text{radiance} \cdot (\omega_o \cdot n)^{\text{focus}^2} & \text{otherwise} \end{cases}$$

Le focus est mis au carré pour donner une réponse plus linéaire entre son changement de valeur et l'angle de l'éclairage résultant.

Dans les images suivantes, les spotlights sont orientés incorrectement, du a des problèmes dans l'export de scène blender. Toutefois, on voit l'effet de l'éclairage conique sur la scène.



Fig. 19. – Orientation défaillante des spotlights (1) et (2), puis correctement aligné en (3)

## Denoiser - Thibaut

Pour débruiter les images, la librairie Intel Open Image Denoise (OIDN) a été intégrée en utilisant la crate `oidn`. Pour le mode de débruitage utilisant l'albedo et les normales, l'intégrateur `NormalIntegrator` a été réutilisé, et un second `AlbedoIntegrator` fut créé. Celui-ci fonctionne sur le même principe que l'intégrateur des normales, mais en utilisant une fonction additionnelle `get_albedo` sur le trait `Material`.

Vu que l'utilisation d'OIDN nécessite une configuration additionnelle sur la machine utilisée (pour la compilation et lors de l'exécution), cette fonctionnalité a été mise derrière un *feature flag* pour pouvoir continuer l'utilisation de l'engin sans OIDN sans devoir faire la configuration OIDN.

### Validation

Pour la validation, la scène `living_room` a été utilisée, rendue avec un faible nombre d'échantillons par pixel, sans débruitage, avec débruitage simple, et avec le débruitage complet (qui utilise les albedos et les normales). Ensuite, les trois images produites ont été comparées côté-à-côte via l'outil ICAT de Nvidia.

Vu que le temps de rendu de notre scène finale est très long sur une machine personnelle, le débruitage a seulement été validé sur des images fixes de celle-ci, et non sur une animation. L'effet de scintillement du à l'incohérence d'une image à l'autre n'a donc pas pu être réglé.

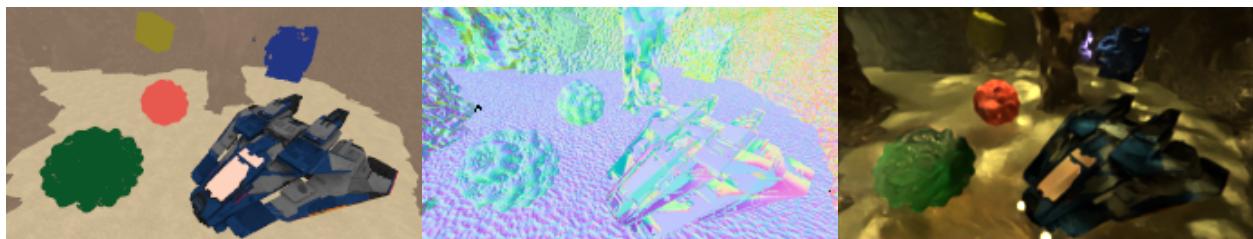


Fig. 20. – Visualisation des cartes de normales, albedos et rendu débruité

## Références

An Introduction to Raymarching. (s.d.). Maxime Garcia. Repéré à [http://typhomnt.github.io/teaching/ray\\_tracing/raymarching\\_intro/](http://typhomnt.github.io/teaching/ray_tracing/raymarching_intro/)

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. Physically Based Rendering: From Theory to Implementation (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Ray Marching and Signed Distance Functions. (s.d.). Zero Wind :: Jamie Wong. Repéré à <http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/>

Tzu-Mao Li. UCSD CSE 272 Assignment 1. <https://sayan1an.github.io/pdfs/references/disneyBrdf.pdf>

GPT for image diffusion of inspired images

Gemini for correction, translation of equations to typst and text syntax