



**Matthieu / Pereira – Ingénieur
Informatique - Imagerie Numérique
2023-2024**

**X / X
Professeure associée**

**EFICAD
ADRESSE
ADRESSE
TELEPHONE**

**X / X
Ingénieur Recherche et Développement**

Création d'un moteur de rendu pour visualiser des données de CFAO

1. Résumé

Sous la tutelle de M. X X, 6 mois de stage ont été passés à développer un moteur de rendu 3D utilisant des technologies plus récentes que celles du moteur actuel d'EFICAD. J'ai pu poser les bases de l'architecture, développer des fonctionnalités diverses et travailler en collaboration avec un autre stagiaire. Ce stage m'a permis de gagner de l'expérience dans le domaine du rendu 3D. J'ai dû établir l'état de l'art sur différents sujets, intégrer système d'entrée utilisateur inhabituel et présenter mon travail.

2. Remerciements

Je tiens à remercier M. X X, mon maître de stage et ingénieur en recherche et développement pour m'avoir encadré et avoir rendu ce stage possible.

Je suis également reconnaissant envers M. X X et M. X X, qui m'ont accueilli dans leur entreprise et m'ont présenté leur vision en tant que directeurs.

Je souhaite aussi remercier X X, ingénieur recherche et développement à EFICAD, qui m'a fourni de précieux conseils.

Mes remerciements vont également aux différents professeurs qui m'ont formé au cours de mes études, à l'ESIR, à l'UFR de mathématiques et d'informatique de Strasbourg ou à l'IUT Robert Schuman.

Finalement, je remercie l'équipe de X et son président, X X, pour avoir hébergé notre équipe pendant la durée de ce stage.

Sommaire

1 Résumé	2
2 Remerciements	3
Sommaire	4
3 Introduction	6
4 Présentation de l'entreprise et du logiciel	7
4.1 SWOOD	7
5 Objectifs du stage	8
6 Présentation des outils utilisés	9
7 Présentation d'OpenGL et de la 3D	11
7.1 OpenGL	11
7.2 Principes de la 3D	12
7.2.1 Espaces de coordonnées	12
7.2.2 Autres éléments	12
7.3 Principes d'OpenGL	13
7.3.1 Quelques éléments importants	13
7.3.2 Pipeline d'OpenGL	14
8 Planification et déroulement du projet	17
9 Travaux réalisés	18
9.1 Bases du moteur	18
9.2 Les lignes	18
9.2.1 Les lignes épaisses	18
9.2.2 Les pointillés	19
9.3 Boundary representation	20
9.4 Allocation mémoire	21
9.5 Texte en OpenGL	21
9.5.1 Caractères et formats de fichiers	21
9.5.2 Dessiner un caractère	21
9.5.3 Placement d'un caractère	23
9.5.4 Dessin d'une chaîne de caractères	24
9.6 Transparence en OpenGL	25
9.6.1 Pourquoi la transparence	25
9.6.2 Difficultés de la transparence	25
9.6.3 Méthodes de dessin	25
9.6.4 Autres techniques	26
9.7 Organisation des données	27

9.7.1	Structure des classes	28
9.7.2	Résultats	29
9.8	Intégration d'une souris 3d	29
9.8.1	Souris 3D	29
9.8.2	Intégration à l'application	29
9.9	Rendu basé physique	30
9.9.1	Principe du rendu basé physique	30
9.9.2	Modèle <i>PBR</i> utilisé	30
9.10	Autres détails	31
10	Evaluation des résultats, retour d'expérience	32
11	Conclusion	33
A	Annexes	34
A.1	Bibliographie	34
A.2	Table des figures	37
A.3	Glossaire	38
A.4	Acronymes	39
A.5	Bibliothèques utilisées	40
A.6	Choix de l'outil et alternatives	40
A.6.1	<i>API</i> graphiques	40
A.6.2	Moteurs graphiques et de jeu	41
A.7	Figures variées	41

3. Introduction

Un rendu lisible, personnalisable, conservant une fréquence de rendu suffisamment élevée et proposant des pièces agréables à regarder dans une application 3D contribue à réduire la fatigue visuelle [1], augmente le confort d'utilisation et améliore la perception de qualité du logiciel. Le monde du rendu 3D évolue vite, et a subit de nombreux changements majeurs depuis le développement du moteur de rendu des logiciels d'EFICAD, qui n'a que peu changé depuis sa création malgré quelques efforts de modernisation. J'ai donc été chargé d'explorer et de poser les bases d'un nouveau moteur de rendu basé sur des technologies plus modernes, dans le but d'accéder à des améliorations en termes de fréquence de rendu et de qualité visuelle.

4. Présentation de l'entreprise et du logiciel

EFICAD est une entreprise concevant des logiciels de *Conception Assistée par Ordinateur (CAO)/Fabrication Assistée par Ordinateur (FAO)*, spécialisée dans les métiers du bois. Fondée en 1989, et actuellement dirigée par M. X X et M. X X, son siège social se situe à La Grande Motte. Le produit phare d'EFICAD est SWOOD, une suite logicielle destinée à la conception et la fabrication de meubles en bois. Depuis 2019, une filiale d'EFICAD existe aux Etats-Unis : EFICAD America.

Distribuant des produits dépendant de SOLIDWORKS, un logiciel de CAO, EFICAD est en étroite collaboration avec Dassault Systems, l'éditeur de SOLIDWORKS. Cette relation se matérialise par la certification "SOLIDWORKS Gold Product" donnée à SWOOD en 2021.

4.1 SWOOD

La suite logicielle SWOOD propose différents produits, pouvant être utilisés indépendamment ou en coordination.

SWOOD Design SWOOD Design est une application destinée à la conception de meubles en bois. Sa spécialisation autour du bois permet à SWOOD Design d'automatiser des tâches fréquentes liées aux métiers du bois, comme le placement des panneaux et de leurs connecteurs, de choisir les matériaux jusqu'à leurs détails, et d'obtenir un rapport offrant de nombreuses informations comme le coût de fabrication.

SWOOD Design permet de concevoir des meubles dont les dimensions sont complètement paramétriques, ce qui facilite leur modification et la création de sous-composants réutilisables d'un projet à l'autre.

SWOOD CAM SWOOD CAM(*Computer-Aided Manufacturing*) permet de générer des programmes d'usinage ou de découpe sur une grande variété de machines. SWOOD CAM propose également de simuler les opérations à réaliser afin de s'assurer de la cohérence entre le programme et le produit final attendu.

SWOOD Nesting SWOOD Nesting propose d'optimiser le placement de différentes pièces dans un seul et même panneau afin de produire le maximum de pièce avec le moins de matière première possible, le tout en réduisant le temps nécessaire aux découpes.

SWOOD Center SWOOD Center est une application servant à interfaçer la conception paramétrique de SWOOD Design avec d'autres applications, afin de faciliter et d'automatiser la production de fournitures personnalisables.

SWOOD BW SWOOD BW(Beam and Wall) facilite la conception de charpentes et de maisons en bois, en offrant une conception paramétrique et simple d'utilisation.

5. Objectifs du stage

Le moteur de rendu utilisé par EFICAD a été développé au début des années 2000, et depuis les méthodes et paradigmes de rendu 3D ont énormément changé. Des sections spécifiques ont été recodées avec une version plus moderne, et certaines évolutions ont été tentées. Cependant, il était trop difficile de réaliser des changements aussi importants. J'ai donc été pris comme stagiaire pour tester l'intérêt d'une évolution vers un moteur plus moderne et pour poser les bases de cette évolution.

Cette évolution permettrait d'apporter de nombreuses améliorations au rendu 3D actuellement offert par SWOOD.

- Un rendu plus moderne, plus joli, et plus modulable. Le moteur actuel de SWOOD date d'une époque où le rendu 3D via *API* pour le rendu graphique n'était pas programmable, juste configurable. Cela a pour cause de considérablement restreindre les possibilités de dessin. Passer à une technologie plus moderne donne accès à un rendu 3D programmable, ce qui permet de réaliser des effets supplémentaires.
- Un rendu plus performant. A l'époque où le moteur a été écrit, le matériel était très différent, ce qui a mené à la création d'une *API* dite immédiate, qui a comme conséquence de réaliser un très grand nombre d'interactions avec le *driver*⁽¹⁾. De nos jours, ces interactions avec le *driver* sont ce que l'on tente de minimiser[3].

⁽¹⁾Un *driver* est un programme permettant la communication entre une application et du matériel[2]. Dans ce rapport, on parle du programme permettant l'interaction avec la carte graphique via OpenGL

6. Présentation des outils utilisés

Afin de mener ma tâche à bien j'ai été mené à utiliser de nombreux outils :

Langage de programmation Le projet a écrit en C++20 et en GLSL.

Visual studio 2022 La majorité du développement a été réalisé avec l'environnement de développement intégré Visual Studio 2022, sous Windows 10. J'ai eu à prendre en main le logiciel et configurer la chaîne de compilation, générer une bibliothèque statique, ajouter des dépendances, changer des paramètres selon l'activation ou non du mode débogage. Le compilateur utilisé était *Microsoft Visual C++ (MSVC)*. Visual studio propose aussi de nombreux outils intégrés que j'ai eu l'occasion d'utiliser. Par exemple :

1. Un outil de déboggage : cet outil a été utilisé de manière systématique au cours du stage afin de m'aider à identifier et corriger les bugs[4].
2. Un outil de Profiling du code : grâce à cet outil, j'ai pu analyser certaines parties de mon code qui me semblaient particulièrement lentes. J'ai pu facilement trouver les causes et les corriger, ce qui m'a permis par exemple de diviser par 3 le temps de chargement d'un type de modèle[5].
3. Un outil de Profiling du temps de compilation : craignant que les temps de compilation ne deviennent trop élevés, j'ai utilisé un outil afin d'analyser la compilation du programme et trouver ce qui prenait du temps[6].

Doxygen Doxygen est un outil de génération de documentation permettant de standardiser son écriture et de générer facilement des fichiers la contenant dans un format agréable à naviguer. [7] Grâce à cet outil, je suis beaucoup plus confiant du fait que les prochaines personnes touchant au projet seront capables de le comprendre et de le faire évoluer.

Outils de prise de notes Afin de m'aider à réfléchir tout en laissant une trace de mes réflexions "non structurées", que ce soit pour l'écriture de ce rapport de stage ou pour les prochaines personnes menées à utiliser mon projet, j'ai pris des notes sur des applications en ligne. Tout d'abord Notion [8], puis Microsoft Loop [9].

Outils de communication Afin de communiquer avec d'autres employés, j'ai utilisé Microsoft Teams, une application de chat [10], et Microsoft Outlook, qui m'a servi à recevoir et envoyer des e-mails, ainsi que d'agenda [9]

Outils de versioning Git a été l'outil utilisé lors de mon stage afin de garder un historique des différentes versions. L'outil a aussi été utilisé afin de collaborer avec le second stagiaire.

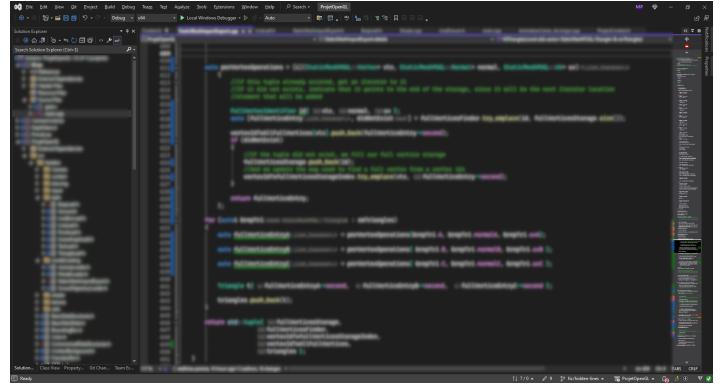
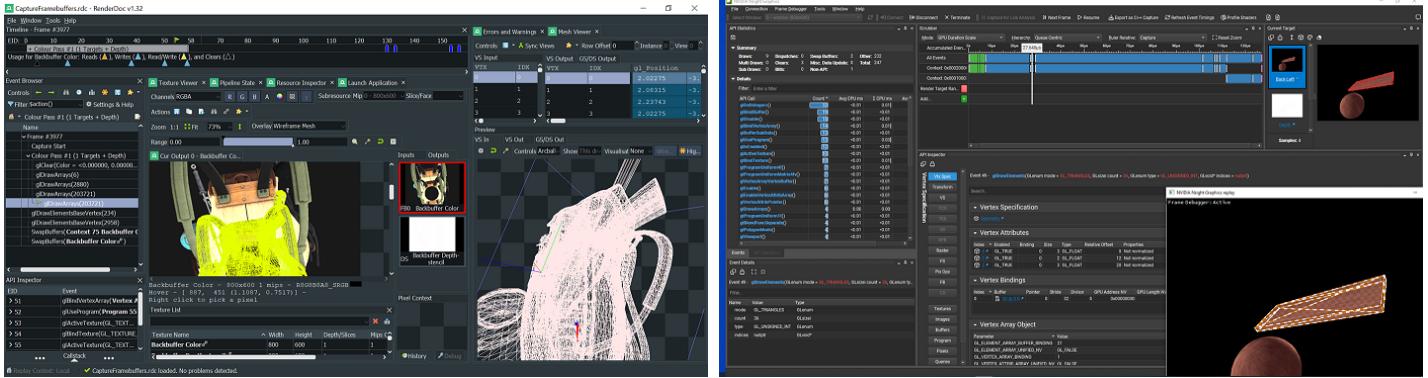


Figure 6.1: Interface de visual studio 2022.



(a) RenderDoc

(b) Nvidia Nsight

Figure 6.2: Outils de débogage et de profiling graphique

Outils de débogage et de profiling graphique Afin de débugger le rendu d'une *trame* et de repérer les sections ralentissant le rendu, Renderdoc et Nvidia Nsight ont été utilisés.

Renderdoc propose de nombreuses fonctionnalités pour décomposer les instructions qui ont mené au dessin d'une *trame*, comme voir l'état de la *pipeline*⁽¹⁾ à chaque appel de dessin ou lister les appels à OpenGL. Cet outil a été d'une aide précieuse pour trouver l'origine de différents problèmes rencontrés.

Nvidia Nsight a permis d'identifier les facteurs majeurs impactant le temps de rendu dans le but de les corriger. Ce logiciel propose des informations très détaillées sur chaque information mesurable. Un point intéressant est qu'il détecte les facteurs limitants dans le rendu et suggère des améliorations ainsi que leur gain possible.

Ces logiciels, visibles en figure 6.2 proposent tout deux des fonctionnalités de débogage ou de *profiling*, mais je les ai utilisés conjointement car ils présentent des avantages et des inconvénients, notamment la quantité d'information présentée par Nsight est tellement grande qu'il m'a été difficile d'en tirer des informations utiles.

Visual studio code Visual studio code a été utilisé pour l'écriture du code *GLSL* (*OpenGL Shading Language*).

⁽¹⁾La pipeline de dessin désigne l'intégralité des opérations, programmables ou non, par lesquelles passent les données lors d'un appel de dessin.

7. Présentation d'OpenGL et de la 3D

OpenGL a été au cœur de ce projet. D'autres alternatives, plus ou moins proches, existent, comme Vulkan, Direct3D, Metal, Unity, OpenSceneGraph. Une recherche des alternative à été réalisée en annexe, section A.6.

Au final, nous sommes restés sur la proposition initiale de développer en OpenGL. C'est la version la plus récente, OpenGL 4.6, qui a été choisie. C'est aussi celle utilisée par SOLIDWORKS, bien qu'elle soit utilisée en mode compatibilité.

7.1 OpenGL

Une grande partie du travail réalisé a été intimement lié à OpenGL et son architecture.

OpenGL est une *API* pour le rendu graphique (que j'abrégerai en *API* graphique) dont la première version remonte à 1992, bien qu'elle ait été formée à partir d'une bibliothèque propriétaire : IRIS GL. C'est actuellement Khronos, un consortium, qui supervise OpenGL.

OpenGL n'est pas une bibliothèque, mais une spécification : l'interface est définie, les comportements sont standardisés, mais en dehors de cela, c'est à chaque créateur d'unité de calcul graphique de proposer des *driver* implémentant cette spécification, peu importe la méthode. Cela laisse une grande liberté d'implémentation. [11]

Le standard OpenGL évolue via des extensions, évaluées par l'OpenGL Architecture Review Board (désormais Khronos). A chaque nouvelle version d'OpenGL, certaines de ces extensions deviennent standards. Ce système d'extension fait qu'il existe un certain nombre de fonctionnalités optionnelles dont il est possible de profiter si l'ordinateur hôte les supporte.[11]

OpenGL a subit d'importantes évolutions, reflétant les changements de paradigmes dans le monde de la programmation graphique. En effet, les *API* graphiques ont initialement été conçues pour réaliser des fonctions fixées sur des données. Cependant, l'arrivée de cartes spécialisées dans ces calculs et le besoin de réaliser des dessins plus complexes et/ou de les personnaliser étaient peu compatibles avec OpenGL.

Ces grands changements ont causé une quasi-duplication des fonctionnalités de l'*API*, et une suppression d'une partie des anciennes fonctionnalités dans les versions récentes.

En-dehors de ces changements, de nombreuses autres sections d'OpenGL ont subi des modifications majeures. La dernière version d'OpenGL est la 4.6, sortie en 2017. Depuis, OpenGL n'a plus subit d'évolution du standard. C'est désormais Vulkan qui représente la nouvelle génération d'*API* graphique de Khronos.

Cependant, d'après Richard S.Wright Jr.[12], employé de LunarG⁽¹⁾, OpenGL est loin d'être obsolète. En effet, bien que certaines fonctionnalités majeures, comme l'accès aux extensions de lancer de rayon, ne soient pas disponibles, l'évolution entre OpenGL et Vulkan se situe surtout au niveau du contrôle qu'a le développeur sur le processus de rendu : Vulkan propose un contrôle beaucoup plus manuel du rendu, ce qui permet d'utiliser la connaissance du programme développé pour optimiser. Cette augmentation en complexité fait que développer un programme sous Vulkan sera beaucoup plus long et complexe, pour un gain de performance marginal. Richard S.Wright Jr. pense donc que OpenGL restera l'*API* de choix pour les applications ne nécessitant pas de pousser la machine à ses limites.

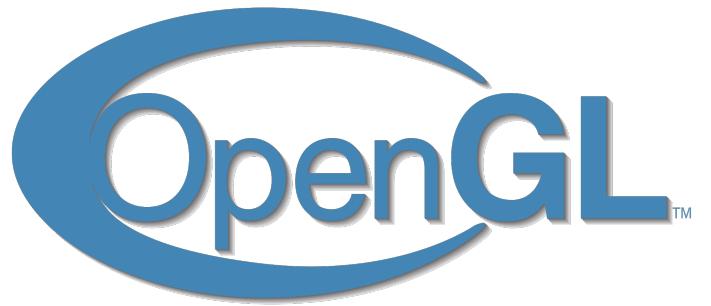


Figure 7.1: Logo d'OpenGL

⁽¹⁾LunarG propose le Vulkan *Source Development Kit (SDK)*, soutenu par Khronos

Une recherche des différentes alternatives à OpenGL a été réalisée. Elle est disponible en annexe, section A.6.

7.2 Principes de la 3D

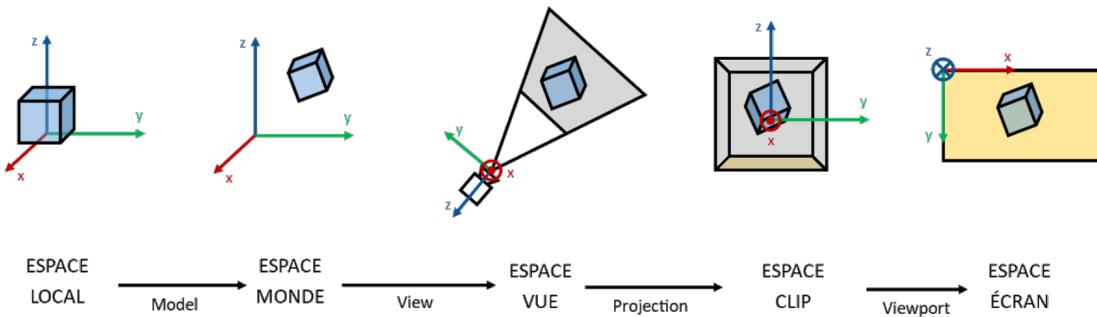
Afin de dessiner un objet en 3 dimensions sur un écran, les données changent plusieurs fois de repère. On peut voir un schéma représentant les différents espaces en figure 7.2. Les coordonnées homogènes sont un outil utile pour aider à réaliser ces changements de repère.

Coordonnées Homogènes Les coordonnées homogènes sont un système de coordonnées qui définit une équivalence telle que, dans le cas de la 3D,

$$[x_1, y_1, z_1] \Leftrightarrow [x_1, y_1, z_1, 1] \Leftrightarrow [w * x_1, w * y_1, w * z_1, w]$$

Ce système de coordonnées permet d'appliquer des translations à un vecteur via une multiplication de matrice.

7.2.1 Espaces de coordonnées



Espace Local Les coordonnées en 3 dimensions commencent dans un espace propre à l'objet. C'est l'espace local.

Espace Monde Via une matrice nommée matrice de modèle, les coordonnées sont passées de l'espace local à l'espace monde, qui est l'espace global de la scène.

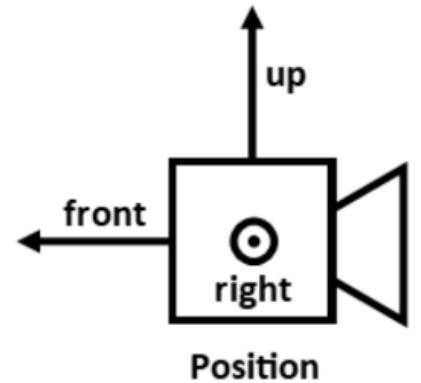
Espace Caméra/œil/vue Via la matrice caméra, les données sont transformées de l'espace monde à l'espace caméra (ou bien œil, vue), visible en figure 7.3.

Espace Clip et NDC Via une matrice de projection, les données sont passées dans un espace représenté par des coordonnées homogènes nommées espace clip, dans lequel la coordonnée z correspond à la profondeur, et w est différent de 1. Les coordonnées seront ensuite passées en *Normalized Device Coordinates (NDC)* via la division perspective, c'est-à-dire la division par la coordonnée w.

Espace Ecran Via une mise à l'échelle et une translation, les données passent vers l'espace écran, qui correspond à l'espace de l'image dans laquelle le dessin se réalise.

7.2.2 Autres éléments

Graphe de scène En 3D, il est utile de représenter des relations de positions simples entre des objets : lorsqu'on déplace une table, il est préférable que les objets posés dessus suivent cette table. Pour cela, on a recours à un graphe de scène. Dans ce graphe, un nœud peut avoir plusieurs fils, qui héritent de la transformation



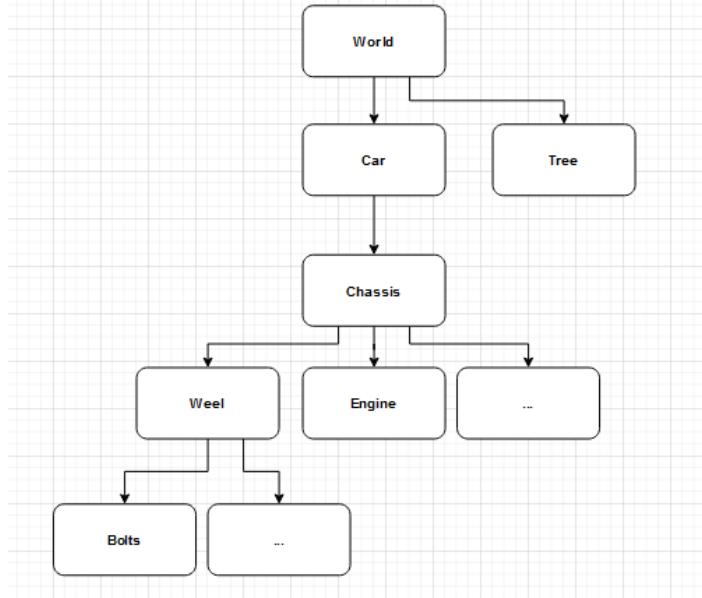


Figure 7.4: Exemple de graphe de scène [13]

du parent et rajoutent la leur. C'est cette transformation cumulée qui sera utilisée comme matrice de modèle. On peut voir un exemple de graphe de scène en figure 7.4

Projections On retrouve 2 catégories de projections : la projection perspective, qui correspond à une projection donnant un sens de profondeur, et la projection orthographique, qui conserve les lignes parallèles. La projection orthographique donne un résultat ne correspondant pas à ce qu'un humain voit normalement, mais le fait qu'elle ne déforme pas les objets est avantageux pour la CAO. On peut voir une comparaison des 2 types de projection en figure 7.5. Il existe d'autres types de projections qui ne seront pas abordées.

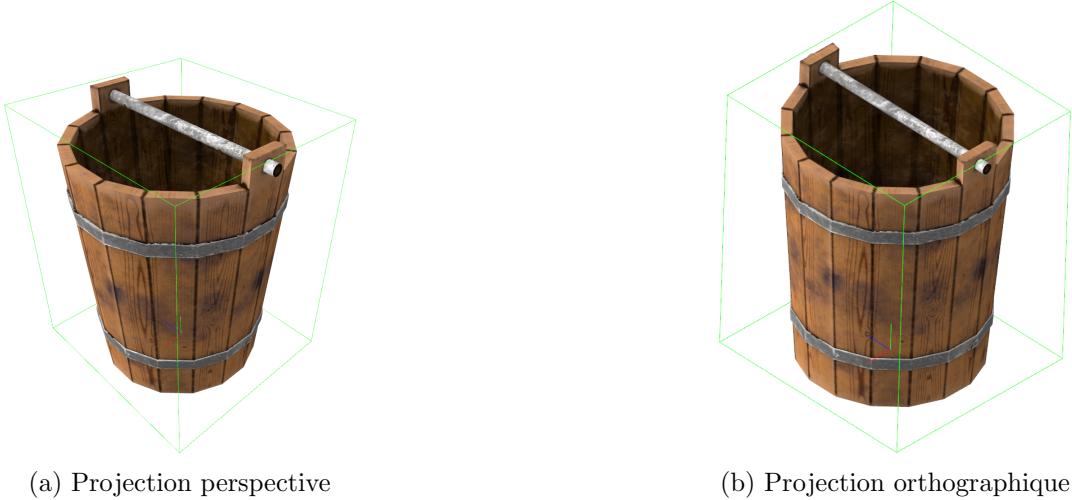


Figure 7.5: Comparaison de la projection orthographique et perspective.

7.3 Principes d'OpenGL

7.3.1 Quelques éléments importants

Shaders Les shaders sont des bouts de code servant à réaliser une partie programmable du processus de rendu. Auparavant, ces shaders étaient écrits exclusivement en *OpenGL Shading Language (GLSL)*, un langage

haut niveau (dans le contexte de la 3D) similaire au C. Depuis OpenGL 4.6, il est possible de fournir les shaders grâce au SPIR-V, un format intermédiaire que l'on peut obtenir en compilant du GLSL, du HLSL (l'équivalent Direct3D), ou bien d'autres langages.

Flux de travail Pour dessiner (naïvement) un objet en OpenGL, on doit tout d'abord préparer et charger sur la carte graphique les données à l'avance, comme les textures ou les données par vertex. Une fois que ces données sont présentes, il suffit de

1. Préparer l'état de la pipeline : par exemple, choisir le shader program, les textures, les données des vertex ou bien la fonction utilisée pour différents tests.
2. Lancer l'appel de dessin.

Pixel et fragment Dans la pipeline de rendu, l'étape du "rasterizer" produit des fragments à partir de primitives[14]. Ils seront utilisés pour calculer la valeur des pixels. Le nombre de fragments par pixel est configurable, et sert notamment à réduire l'aliasing(le crênelage). On peut voir les conséquences de l'utilisation d'un système de réduction d'aliasing (anti aliasing) via l'augmentation du nombre de fragments par pixel en figure 7.6.

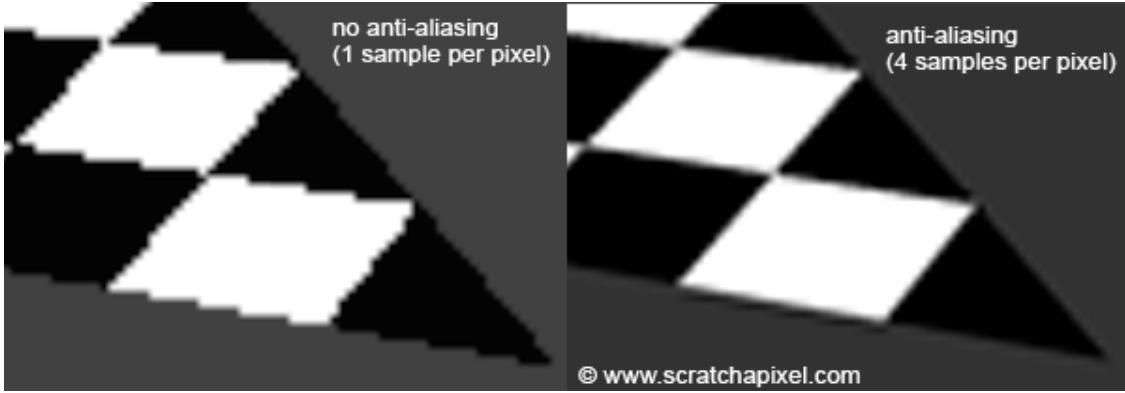


Figure 7.6: Anti-aliasing utilisant plus d'un fragment par pixel [15].

Lors du dessin d'un objet, les données passent par un ensemble d'étapes séquentielles, certaines configurables, d'autres programmables. L'ensemble de ces étapes se nomme la pipeline de rendu, qui va être rapidement présentée.

7.3.2 Pipeline d'OpenGL

L'idée de cette pipeline est la suivante :

1. Prendre des points (vertices) représentant des positions dans un maillage/trajectoire/... en entrée.
2. Leur appliquer des transformations.
3. Les transformer en primitives (lignes, triangles, ...).
4. Produire des fragments pour chaque pixel de l'écran sur lequel cette primitive se trouve.
5. Appliquer différentes opérations sur ces fragments.

Les différentes sections de la pipeline permettant de réaliser ces opérations vont être rapidement présentées.

Vertex Specification[16] Cette étape consiste à lire dans la mémoire et à préparer une liste de données à envoyer à l'étape suivante.

Vertex Shader[17]. Le vertex shader est la première partie programmable du pipeline de rendu. Cette étape permet d'appliquer des transformations à un vertex indépendamment des autres. Généralement, on y retrouve le passage des coordonnées locales aux coordonnées clip ainsi que la préparation de différentes données pour les étapes suivantes

Tesselation[18] Il s'agit d'un ensemble d'étapes programmables (le Tesselation Control Shader et le Tesselation Evaluation Shader) qui servent à subdiviser des surfaces en plusieurs primitives. La tesselation modifie et transmet les données passées par le vertex shader.

Geometry shader[19] Le Geometry shader est une étape programmable permettant de transformer une primitive en 0 ou plusieurs primitives différentes. Le Geometry shader modifie et transmet ou stoppe les données passées par l'étape précédente.

Post traitement des vertices[21] C'est un ensemble d'étapes réalisées une fois que les données des vertices sont fixées.

1. le Transform Feedback[22], permettant de sauvegarder le résultat des calculs réalisés jusque là.
2. L'assemblage des primitives[23], qui assemble les vertices en primitives (par exemple un triangle ou une ligne) et les transmets à l'étape suivante.
3. Le Clipping[24], qui découpe les primitives de manière à ce qu'elles soient exclusivement à l'intérieur ou à l'extérieur du cube de clipping présenté précédemment et stoppe celles qui sont en dehors. Cette étape réalise la division perspective et transforme les coordonnées clip en coordonnées écran.
4. Le face culling[25], qui stoppe les primitives selon leur orientation (font elles face à l'écran ou non).

Rasterisation La rasterisation transforme chaque primitive en un ensemble de fragments et les transmets au fragment shader.

Fragment shader[14] Le fragment shader est l'étape programmable permettant de définir la couleur d'un fragment. Généralement, on utilise pour cela les informations interpolées ayant voyagé dans toute la pipeline, comme les coordonnées de texture lues dans la mémoire à la toute première étape et différentes variables globales passées par l'utilisateur pour calculer la couleur, l'éclairage, la transparence d'un fragment. Cette étape produit des couleurs à écrire dans plusieurs images différentes. Il est possible de stopper le dessin à cette étape.

Opération par échantillon[26] Finalement, on réalise quelques étapes permettant de stopper le fragment selon certaines conditions (Depth test, stencil test, Scissor test ou Ownership test), puis on mélange ou érase la couleur déjà existante dans l'image avec la couleur du fragment généré.

Bien que déjà complexe, cette suite d'opérations est simplifiée et ignore un grand nombre de détails.

Differences entre GPU et CPU

Bien que réalisant tous deux des calculs, un GPU et un CPU ont une architecture radicalement différente, ce qui les rends efficaces dans différents cas. Un GPU est construit autour de l'idée d'appliquer la même opération sur des données différentes en parallèle, alors qu'un CPU ne possède que peu de coeurs, destinés à faire fonctionner des programmes différents. Le GPU possède un grand nombre de cœur et est spécialisé dans les opérations *Single Instruction Multiple Data (SIMD)*⁽¹⁾ sur nombre flottant. Dans le cas du rendu 3D, cela permet à une carte graphique de traiter un grand nombre de vertices ou de fragments en parallèle.

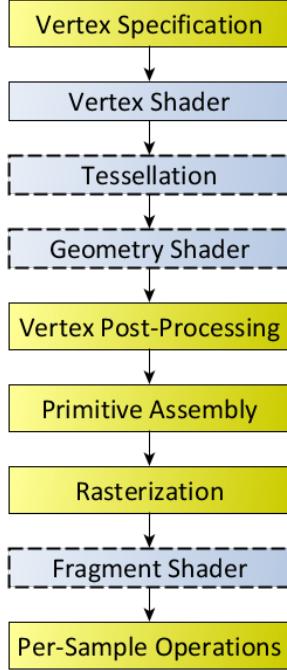


Figure 7.7: Diagramme de la Pipeline de Rendu. Le jaune correspond à une section configurable, le bleu à une section programmable et les pointillés à une section optionnelle[20].

⁽¹⁾ Le SIMD est un type d'architecture permettant d'appliquer la même opération sur plusieurs données en même temps.[27]

GPGPU Malgré son nom, la puissance de calcul d'une carte graphique la rend performante pour réaliser des calculs hors du domaine de l'image : c'est le *General Purpose Computing on Graphics Processing Units (GPGPU)*.

Cette étape ne fait pas partie de la pipeline de rendu, et n'a pas été utilisée lors de ce stage[28], cependant plusieurs utilisations du *GPGPU* pour des opérations réalisées par EFICAD ont déjà été évoquées.

8. Planification et déroulement du projet

La planification des tâches était très libre, mon maître de stage ayant préparé à l'avance une liste de tâches "génériques" donnant une idée de ce qui devait être réalisé avec un temps de réalisation estimé. Ces indications n'ont pas été suivies strictement, et les tâches réalisées étaient plutôt dictées par mes idées actuelles et les problèmes que je rencontrais. Les tâches à réaliser étaient nombreuses, et un grand nombre d'évolutions avaient été proposées. Plusieurs d'entre elles n'ont pas été réalisées, notamment une partie consistant à fournir un équivalent simplifié du moteur sur navigateur. Cependant, plusieurs propositions d'extensions ont été réalisées, et plusieurs tâches non prévues initialement ont été faites.

A mon arrivée, j'ai pu rapidement prendre en main la suite logicielle SWOOD afin de comprendre un peu les différents logiciels qui pourraient être affectés par mon projet. J'ai ensuite pu commencer à poser des bases, que j'ai plus tard affiné en une interface agréable à utiliser. J'ai ensuite rajouté différentes fonctionnalités tout en corrigeant et améliorant ce qui avait déjà été fait.

A 2 reprises, j'ai eu à réaliser une présentation devant les directeurs ainsi que d'autres employés afin de montrer le résultat de mon travail.

9. Travaux réalisés

Des tâches variées ont été réalisées. Je ne m'attarderai pas sur les détails de chaque tâche, mais détaillerai celles qui m'ont nécessité le plus de temps et de recherche.

9.1 Bases du moteur

Un arbre de scène, capable de représenter une hiérarchie d'objet a été créé. Ce moteur peut dessiner les primitives de bases, et propose à l'utilisateur de choisir son matériau. Il est possible de créer de nouveaux matériaux avec des shaders personnalisés, ce qui laisse une grande liberté.

9.2 Les lignes

Une fonctionnalité particulièrement importante dans les applications de *CAO/FAO* est le dessin de points et de lignes, que l'on retrouve bien moins fréquemment dans d'autres types d'applications. Cependant, avec l'arrivée d'OpenGL 3.0, les lignes épaisses et les pointillés ont été dépréciés [29]. Il s'agissait de 2 fonctionnalités importantes, qui ont du être recodées. Les lignes sont souvent représentées sous forme de polylinéaires, qui sont des suites de segments réutilisant le vertex précédent. On peut en voir un exemple en figure 9.1

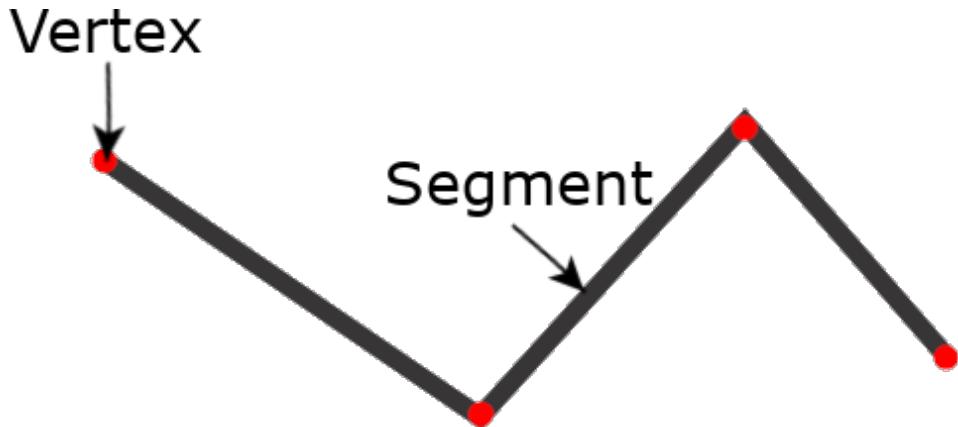


Figure 9.1: Elements d'une polyligne.

9.2.1 Les lignes épaisses

Plusieurs méthodes existent pour dessiner des lignes épaisses, [30] en présente 6 différentes. La méthode choisie a été d'utiliser le Geometry Shader pour transformer des lignes en *quads*⁽¹⁾. Cette transformation étant réalisée dans l'espace écran, il est facile de contrôler l'épaisseur en termes de pixel.

Bien que fonctionnelle, cette méthode n'est probablement pas la meilleure. En effet, compte tenu de la nature même des Geometry shaders, ceux-ci ont des performances assez mauvaises sur la majorité des cartes graphiques[31]. Il pourrait donc être intéressant de tester d'autres méthodes, étant donné que le dessin de trajectoire est un point critique de SWOOD.

Il est intéressant de noter que la modification de l'épaisseur des lignes est revenue dans Vulkan.

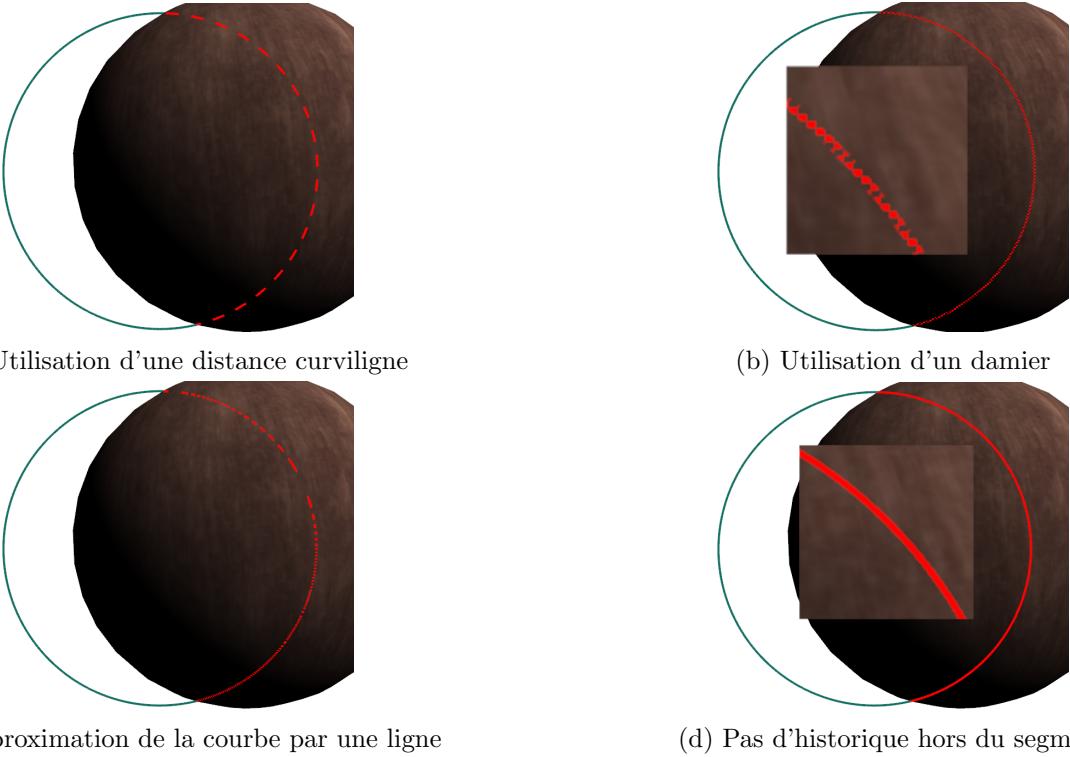


Figure 9.2: Comparaison des méthodes de pointillés sur une polygone très détaillée (100 000 segments)

9.2.2 Les pointillés

Les pointillés dans les vieilles versions d'OpenGL étaient construits à partir de 2 paramètres : leur motif et leur facteur [32]. Le motif est composé de 16 bits. Lorsqu'un bit est à 1, cela signifie qu'il faut dessiner, lorsqu'il est à 0, il ne faut pas dessiner. Ce motif est généralement représenté sous la forme d'un nombre en hexadécimal. Le facteur consiste simplement à étendre ou à réduire la longueur sur laquelle le motif s'applique. Grâce à ce système, il est possible de facilement obtenir différentes représentations de pointillés. On peut voir des exemples de représentations en figure 9.3.

Les pointillés en pratique Initialement, la transformation des lignes en pointillés a été réalisée rapidement, et les résultats obtenus étaient satisfaisants. La transformation avait été réalisée en utilisant la distance en coordonnée écran entre le vertex précédent et la position du fragment. Il s'est avéré que lorsqu'un segment occupait moins d'un pixel, cette formule ne fonctionnait plus : les lignes se transformaient soit en lignes pleines, soit en lignes invisibles. Il a donc fallu chercher des alternatives.

- Utilisation d'une distance curviligne locale. Grâce à cela, il est possible de garder un "historique" des pointillés, qui peuvent donc continuer à travers plusieurs segments. Cette méthode a le défaut de ne pas s'adapter à l'écran. Les pointillés changent donc d'apparence selon la perspective ou le niveau de zoom.
- Utilisation d'une distance curviligne sur l'écran. En projetant les points hors du Fragment shader, il serait possible de récupérer la distance curviligne perçue par l'utilisateur, et donc d'obtenir un résultat probablement "parfait". Cependant, cela nécessiterait de recalculer les distances à chaque nouvelle image, ce qui pourrait entraîner de lourds calculs ainsi qu'un transfert de données important entre le CPU et le GPU. Des méthodes de "sommes suffixe" hautement parallélisables existent^[33], et il serait donc possible de les utiliser afin de réaliser le calcul en GPGPU. Cela aurait l'avantage de profiter de la puissance de calcul de la carte graphique ainsi que de ne pas avoir à transférer les données entre le CPU et le GPU.
- La méthode échiquier. En abandonnant les fragments selon leur position sur l'écran, il est possible de faire apparaître des pointillés cohérents peu importe la taille d'un segment. Cependant, cette méthode

⁽¹⁾ Un quad est un ensemble de 2 triangles formant un carré.

donne des résultats assez peu lisibles.

- La méthode approximation locale par des lignes. Nous avons utilisé les 2 points connus du segment pour trouver le point le plus proche sur la droite quittant l'écran. Les pointillés ont été dessinés à partir de ce point. Cette méthode donnait des résultats corrects, mais avait quelques artefacts et restait mauvaise pour afficher des courbes composées de nombreux segments.

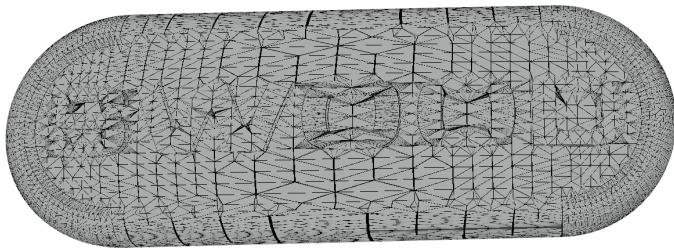
Au final, il n'y a que les méthodes basées distance curviligne qui donnent des résultats graphiques acceptables. Les résultats sont comparés en figure 9.2

PATTERN	FACTOR			
0x00FF	1	_____	_____	_____
0x00FF	2	_____	_____	_____
0x0C0F	1	—	—	—
0x0C0F	3	_____	_____	_____
0xAAAA	1	- - - - -	- - - - -	- - - - -
0xAAAA	2	— — — — —	— — — — —	— — — — —
0xAAAA	3	— — — — —	— — — — —	— — — — —
0xAAAA	4	— — — — —	— — — — —	— — — — —

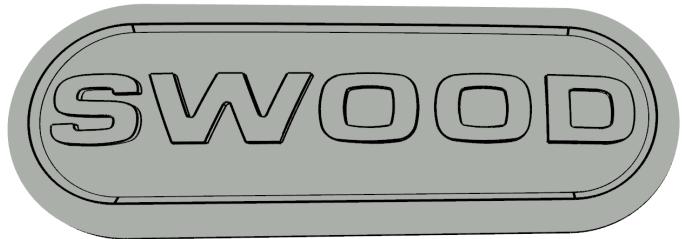
Figure 9.3: Exemple de liens entre le motif et les pointillés[34].

9.3 Boundary representation

Le *Boundary Representation (BREP)* est une manière de représenter un objet via ses contours[35]. Cette représentation est pratique dans le contexte de la *CAO* car elle garantit l'*étanchéité* du modèle. Le *BREP* est utilisé par SOLIDWORKS et SWOOD. J'ai donc dû créer un système pour les afficher.



(a) Modèle composé de triangles



(b) Représentation d'un modèle type BREP

Figure 9.4: Modèle Maillé et modèle BREP

Le *BREP* pour le dessin En réalité, ce sont des primitives qui sont passées au moteur. On retrouve différentes catégories d'éléments dans ces primitives.

- Faces : Ce sont des ensembles de triangles.
- Fins : Ce sont les côtés des triangles.

- Edge : Ce sont des ensembles de Fins.
- Node : Ce sont des points correspondant aux extrémités des Edges.

Afin d'afficher les modèles BREP, il a fallu convertir les données depuis un autre format puis créer différentes briques de base (points, lignes triangles) pour afficher les différents éléments.

9.4 Allocation mémoire

Afin de simplifier la manipulation de la mémoire sur GPU et de définir différentes stratégies de stockage des données, des classes d'allocation mémoire pour OpenGL ont été développées. Au moment où ce rapport est écrit, il ne s'agit que d'une base pouvant être facilement étendue, associé à un système pour traquer les fuites mémoire. Une fois complètement développé, ce système permettra de facilement allouer, modifier et libérer de la mémoire statique ou dynamique, et devra proposer un système d'allocation mémoire efficace comme le buddy allocator, qui permettra d'allouer une grande quantité de mémoire au démarrage du programme et de redistribuer des sections de cette mémoire au besoin.

9.5 Texte en OpenGL

L'affichage de texte est présent dans presque toutes les applications 3D. Dans le cas de SWOOD, on peut par exemple retrouver un besoin lorsqu'on affiche des côtes de mesure. Malheureusement, OpenGL ne propose aucun outil pour afficher facilement du texte. De plus, il est important que les caractères soient facilement lisibles, ils doivent donc avoir une qualité suffisante. J'ai dû faire un état de l'art des méthodes d'affichage de texte sur OpenGL et en implémenter une.

9.5.1 Caractères et formats de fichiers

Avant d'afficher un caractère ou un texte, il faut déjà récupérer la police, que l'on doit charger. On trouve historiquement 2 familles principales de format de polices : les formats rastérisés, et les formats vectoriels [36]

Formats rastérisés Il s'agit d'un système archaïque consistant à stocker une police sous forme d'image rastérisée. Ils ont l'avantage d'être basiques et simple à utiliser, mais c'est bien leur seul avantage. La taille d'un caractère est fixée en nombre de pixels, et tenter d'afficher le caractère en plus gros nécessitera d'interpoler les valeurs, ce qui mène à des caractères flous ou difformes.[37] On les trouve sur les vieux systèmes (des années 70-80) [38] ou bien sur des systèmes destinés à être le plus basique possible, comme un *BIOS*[37].

Formats vectoriels Ces formats consistent à stocker les caractères sous forme de fonctions mathématiques, comme dans les images vectorielles type *Scalable Vector Graphics (SVG)*. On peut donc agrandir ces caractères à volonté sans perdre de détails. [39]

Un format de police vectorielle est dominant : OpenType, mais on retrouve d'autres formats plus anciens, tels que TrueType ou PostScript. Cette catégorie se sépare elle-même en 2 classes de polices : les formats basés pinceau, et les formats basés contour. Le premier consiste à définir les "coups de pinceau" qu'il faut réaliser afin de dessiner le caractère, tandis que le second, que l'on retrouve plus fréquemment, définit ses contours[37]. Utilisant une bibliothèque pour lire ces polices, on ne s'intéressera pas aux détails.

9.5.2 Dessiner un caractère

On trouve de nombreuses méthodes pour afficher un caractère, mais elles consistent presque toutes à rasteriser le caractère/la police en avance, puis l'utiliser comme texture que l'on applique sur un *quad*, c'est-à-dire un ensemble de 2 triangles formant un carré. Cela ne veut pas pour autant dire que les formats vectoriels et rastérisés sont équivalents, puisque la rastérisation du format vectoriel s'adapte aux besoins et aux algorithmes de l'application, tandis qu'un format pré-rasterisé a déjà perdu son information.

Intuitivement, on rasteriserait en une forme binaire, qu'on affichera en utilisant la valeur échantillonée dans le canal alpha avec une fonction similaire à celle en figure 9.5a . Cependant, à moins de prendre une résolution très élevée par caractère, les résultats sont désastreux, comme visible en figure 9.6a.

Il est possible d'améliorer les résultats, en utilisant une texture anti-aliasée ou bien en binarisant le résultat de l'échantillonnage de la texture. On peut voir les différentes combinaisons en figures 9.6b, 9.6c et 9.6d.

Cependant, [40] propose une méthode reposant sur les champs de distance signés, un élément bien connu dans le monde de la génération procédurale [41].

Leur méthode affirme offrir des résultats de meilleure qualité et la possibilité de rajouter plusieurs post traitements comme visible en figure 9.8 sans aucun coût supplémentaire en temps de rendu.

Cependant, cette méthode souffre d'un défaut important : les angles fins ont tendance à se faire aplatis. Également, on retrouve des artefacts sur certains caractères comme visible en figure 9.9. Ce problème affecte principalement les caractères complexes comme les idéogrammes qu'on retrouve dans les langues asiatiques. Pour le premier problème, [40] et [42] proposent des méthodes basées sur des textures multidimensionnelles utilisant plusieurs champs de distance générés différemment afin d'effacer ces problèmes. Pour le second problème, une solution est d'augmenter la résolution. Heureusement, il est tout à fait possible d'allouer une résolution variable par caractère, basé sur sa complexité.

Bien qu'offrant de très bon résultats, la méthode des champs de distance signée n'est pas la seule méthode existante pour le rendu de police.

On trouve des tentatives de dessin de caractères via des mailages, cependant cette méthode s'avère lente compte tenu de la quantité de triangles nécessaire à la formation d'une seule lettre de bonne qualité.

On trouve également des méthodes à base de rendu exact vectoriel : la lettre n'est plus rastérisée dans une texture, mais est à la place rastérisée dans le fragment shader afin d'offrir la meilleure qualité possible [43] [42]. Les résultats visuels sont parfaits, et peu importe le zoom, la lettre reste toujours exacte. En contrepartie, le coût du rendu est plus élevé.

Compte tenu du rapport performance/simplicité/qualité, j'ai choisi d'utiliser une méthode basée champ de distance. J'ai pour cela eu recours à msdf-atlas-gen [44], une application open source permettant de transformer des polices vectorielles en différents formats rastérisés, dont des champs de distance. L'application permet également de générer un atlas complet en optimisant l'agencement. Le programme produit alors une image ainsi qu'un csv contenant différentes informations pour retrouver le caractère et le placer.

```
Color.rgb = vec3(1.0f);
Color.a = V;
```

(a) Valeur de la texture en tant qu'alpha

```
Color.rgb = vec3(1.0f);
Color.a = float(V >= 0.5);
Color.a = smoothstep(0.45, 0.55, V);
```

(c) Valeur de la texture binarisée puis lissée en tant qu'alpha

```
Color.rgb = vec3(1.0f);
Color.a = float(V >= 0.5);
```

(b) Valeur de la texture binarisée en tant qu'alpha



Figure 9.8: Exemple de post traitement sur du texte



Figure 9.9: Exemples d'artefacts liés à la SDF

Figure 9.5: Différentes fonctions traitant les valeurs V lues dans la texture.

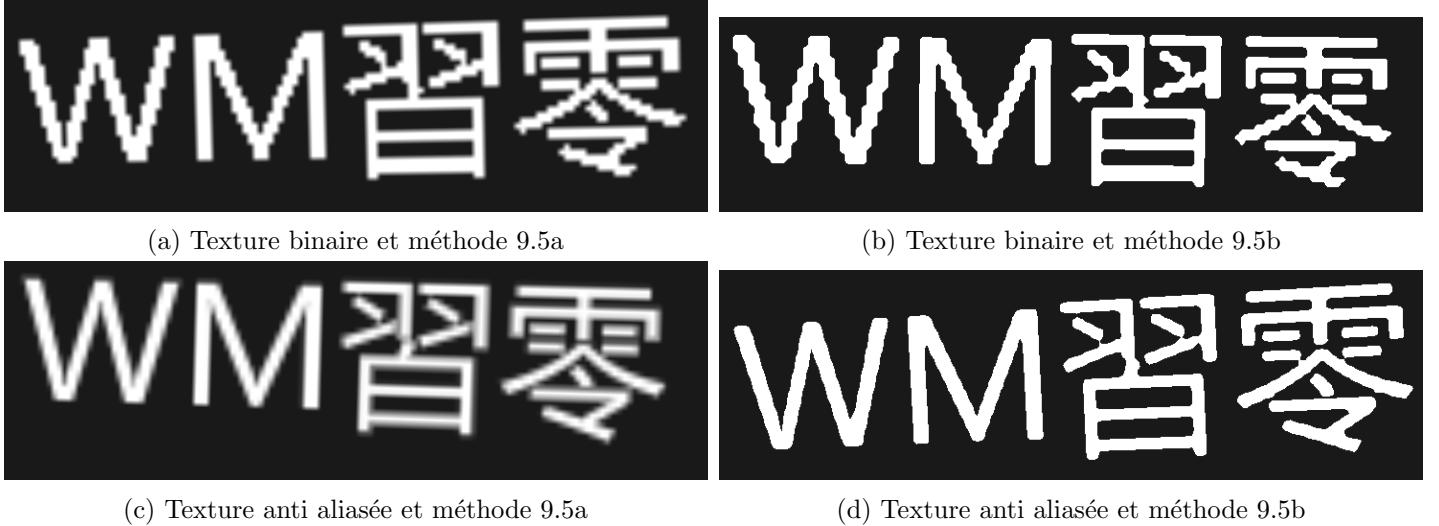


Figure 9.6: Exemple de différentes méthodes pour afficher un caractère

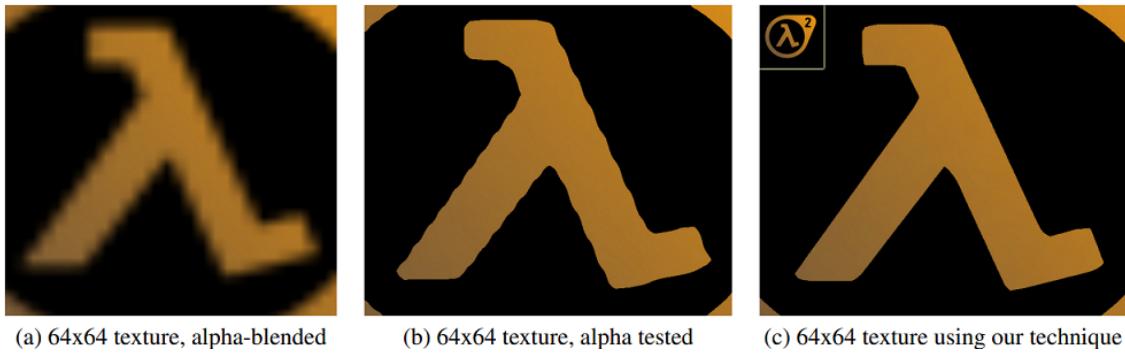


Figure 9.7: Comparaison entre les méthodes de dessin de caractère [40]

9.5.3 Placement d'un caractère

Une fois que l'on est capable de dessiner un caractère, il faut pouvoir le placer correctement par rapport aux autres. On ne peut pas simplement dessiner quad par quad en se déplaçant de manière fixe.

Il existe toute une panoplie de mesures d'une lettre, qu'on peut par exemple retrouver ici [45], mais msdf-atlas-gen ne donne qu'une petite partie des informations sur les caractères qu'il a généré, qu'on peut visualiser sur la figure 9.10.

Tout d'abord, chaque caractère, en lecture horizontale, se trouve sur une ligne imaginaire, nommée baseline. Mais chaque caractère n'est pas posé sur cette ligne, et chaque caractère ne mesure pas la même taille. Pour placer un caractère, on a donc accès à une boîte englobante donnant la position du caractère par rapport à la baseline (par exemple, un p ou un g seront dessinés sous la baseline), ainsi que la dimension attendue du caractère. On peut donc utiliser ces valeurs pour déplacer et agrandir le quad. Ensuite, pour passer d'un caractère au suivant, msdf-gen donne l'"advance", qui permet de calculer le déplacement à réaliser pour trouver l'origine du prochain caractère.

Un caractère est rarement seul, et il peut être utile de considérer le dessin d'un texte comme quelque chose de plus avancé que le simple dessin d'un grand nombre de caractères à des positions arbitraires.

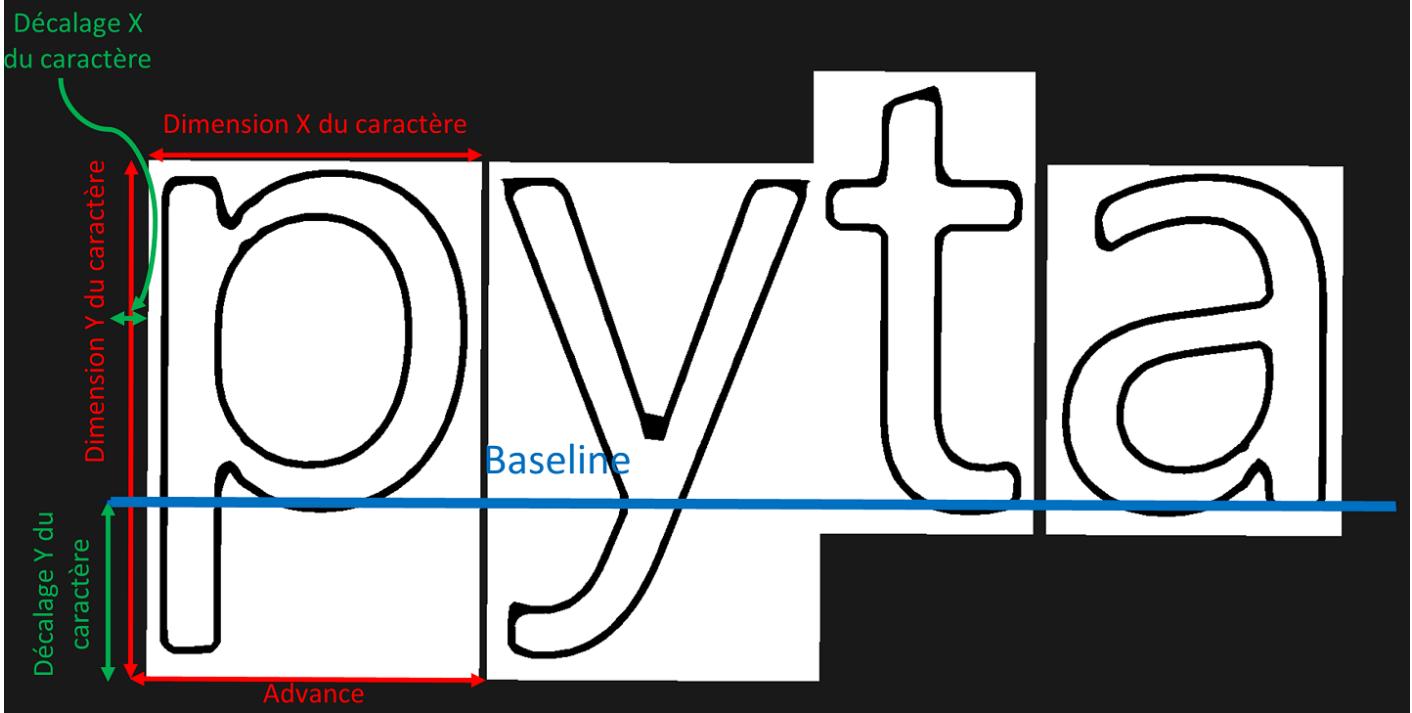


Figure 9.10: Placement d'un caractère sur la Baseline

9.5.4 Dessin d'une chaîne de caractères

Une fois un caractère dessiné, il faut dessiner la chaîne complète. Une méthode naïve consisterait à dessiner caractère par caractère, en calculant la nouvelle matrice de transformation pour chaque caractère, et en adaptant les coordonnées de texture. Cependant, cette méthode produit un grand nombre d'appels de dessin, qui sont considérés comme un des coûts majeurs du rendu d'une scène [46]. Il faut donc chercher à les minimiser.

La méthode naïve de dessin a été testée sur le texte d'introduction de Star Wars épisode 1, mesurant environ 500 caractères. Le temps estimé de rendu du texte était d'environ 2 ms⁽¹⁾, pour bien plus de texte que ce que l'on est censé rendre dans l'application finale. Compte tenu du fait que l'application n'a pas vocation à dessiner beaucoup de texte, j'ai jugé qu'il n'était pas nécessaire de passer du temps à optimiser le dessin.

J'ai cependant relevé d'autres méthodes pouvant être utilisées afin d'accélérer le rendu. Le besoin est le suivant :

1. Dessiner un ensemble de quads, non connectés.
2. Ces quads ont chacun leur propre transformation et coordonnées de texture
3. La position d'un quad ne peut être calculée qu'à partir de la position des quads précédents
4. Le texte peut être dynamique (comme les côtes d'un objet en cours de modification), ou statiques (comme le nom d'un objet)

Au final, pour le dessin d'un grand nombre de caractères, le problème s'apparente au dessin de particules.

Plusieurs améliorations ont donc été relevées :

1. La méthode Multidraw : OpenGL propose une catégorie de fonctions permettant de lancer plusieurs appels de dessin de manière optimisée. En préparant correctement les données, il serait possible de dessiner toute une string (voir même toutes les strings) en un seul appel. Cette méthode correspond aux recommandations d'utilisation d'un OpenGL moderne [3].

⁽¹⁾Pour donner un ordre de grandeur, pour une application tournant à 60 images par seconde, rajouter 2 ms au temps de rendu d'une image fait passer à environ 55 images par secondes

- La méthode draw instanced : cela consisterait à dessiner le même quad autant de fois que l'on souhaite. On aurais alors la possibilité d'associer un second buffer qui contiendrait la matrice de transformation et le vecteur pour retrouver les textures. Cependant, d'après [47], l'instanced rendering est lent pour le dessin d'objets avec un nombre de vertices bas (l'overhead du dessin d'une instance est élevé), ce qui est le cas de nos quads.

Il est très probable que n'importe laquelle de ces méthodes offrirait des résultats meilleurs que les résultats actuels, mais il est également peu probable qu'elles offrent des résultats significativement meilleur compte tenu du fait que l'application n'a pas vocation à écrire beaucoup de texte.

9.6 Transparence en OpenGL

9.6.1 Pourquoi la transparence

Dans Swood, on retrouve de la transparence, ou du moins un besoin de transparence, à plusieurs endroits. Tout d'abord, il est fréquent dans les applications de CAO d'avoir besoin de rendre un objet partiellement transparent, afin de pouvoir observer ce qui se cache derrière sans faire complètement disparaître la surface. On retrouve également un second besoin, celui de pouvoir afficher des objets censés être transparents, comme les vitres qu'on retrouve sur de nombreux meubles. Cependant, en OpenGL, le dessin d'objet transparent n'est pas un thème trivial, il y a besoin de définir une stratégie de dessin afin que les objets non opaques se superposant donne un résultat acceptable.

9.6.2 Difficultés de la transparence

Pour réaliser le mélange de couleurs, nous avons besoin d'un opérateur mélangeant les différentes couleurs avec leur transparence afin d'afficher la couleur finale.

Un opérateur fréquemment utilisé est l'opérateur OVER, décrit dans [48]. Cet opérateur donne de bons résultats, et bien que ceux-ci ne soient pas physiquement exacts, ils sont utilisés comme vérité terrain dans [49]. Cependant, cet opérateur ne fonctionne qu'en l'appliquant d'arrière en avant, et simplement dessiner en indiquant cet opérateur pour le mélange ne suffit pas.

Une solution est de s'assurer que chaque appel dessinera un objet devant le précédent. Sur une scène n'étant pas connue du programmeur, il y a donc besoin d'ordonner le dessin d'arrière en avant. Il n'existe pas toujours de relation d'ordre entre les différents modèles, et même dans un seul modèle, il est donc nécessaire de découper en sous-modèles avant de trier. Ce tri, qui doit être réalisé régulièrement (à chaque fois qu'un objet bouge, apparaît ou disparaît), représente un calcul supplémentaire et augmente le nombre d'appels de dessin et les changements d'état de la pipeline, 2 éléments que l'on tente généralement de minimiser[3]. D'autres méthodes ont donc été trouvées.

9.6.3 Méthodes de dessin

Depth peeling Présenté dans [50], cette méthode consiste à redessiner plusieurs fois les objets transparents de la scène, en ne conservant à chaque fois que le fragment ayant réussi le test de profondeur le plus éloigné, puis en stockant sa profondeur dans le depth buffer, jusqu'à ce qu'il n'y aie plus de fragment réussissant un test de profondeur. Cette méthode permet de dessiner les objets transparents que l'on souhaite dans n'importe quel ordre, et n'a pas de limite "dure" du nombre de fragments pouvant être dessinés au même endroit. Cependant, elle peut nécessiter de réaliser un grand nombre de passes de dessin, en devant à chaque fois reprojeter tous les points des objets transparents. Elle est donc loin d'être idéale en termes de performance. On retrouve dans [51] une amélioration permettant de diviser par deux le nombre de passes.

Record & sort Cette méthode consiste à réaliser un tri au niveau des fragments directement. Cela nécessite donc de stocker chaque "couche" de fragment sur la carte graphique et de les trier avant de réaliser l'opération de mélange. [52] présente différentes méthodes pour répondre à ce problème. Ces méthodes fonctionnent bien, mais restent coûteuses en temps de rendu.

Weighted Blended Order-Independant Transparency Contrairement aux autres méthodes qui cherchent à retrouver un ordre sur les fragments, [49] propose de remplacer l'opérateur OVER par une approximation commutative, et donc indépendante de l'ordre de dessin. C'est cette méthode qui a été choisie. Elle se déroule en plusieurs passes :

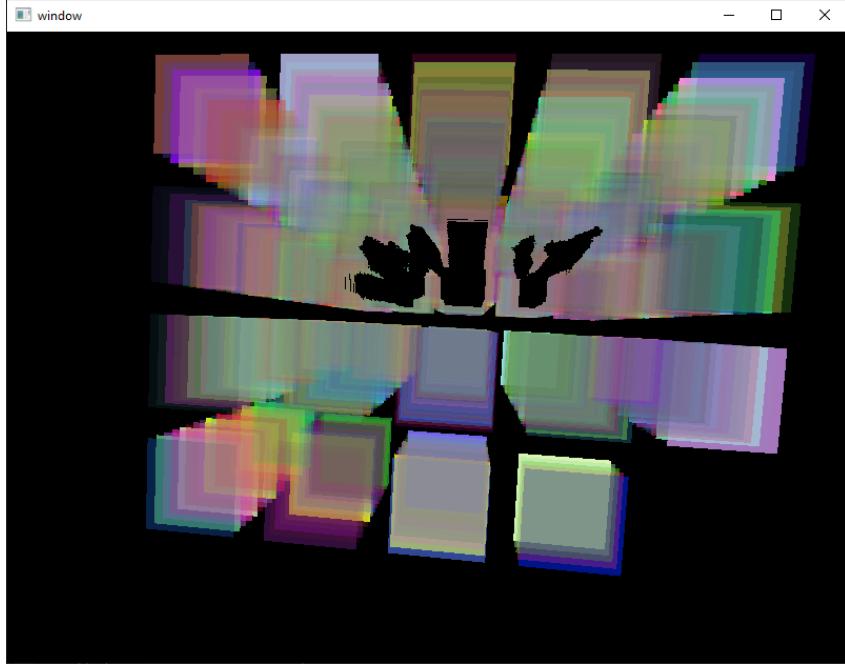


Figure 9.11: Exemple de transparence avec Weighted Blended. On remarque qu'avec l'heuristique choisie, lorsque trop de couches sont superposées, l'algorithme échoue. Ce nombre se situe à environ 45 couches avec un buffer d'accumulation ayant 16 bits par composante.

1. Le dessin ”normal” des objets opaques.
2. Le dessin dans 2 buffers, un buffer d’accumulation et un buffer de révélation
3. La composition de ces 2 buffers sur un buffer de sortie

Cette méthode a l'avantage de ne pas rajouter beaucoup de calcul, de bien supporter un très grand nombre d'objets transparents et d'accepter un large nombre d'objets superposés. Cependant, elle repose sur une heuristique afin de s'approcher au maximum des résultats de l'opérateur OVER, et donc être adaptée à chaque cas, pour n'offrir au final qu'une approximation.

La méthode a été implémentée à l'aide de [53]

Différents essais ont été réalisés afin d'ajuster les paramètres de l'algorithme, comme tester plusieurs heuristiques et changer la précision des buffers, cependant rien de notable n'est sorti de ces essais.

Cette méthode a été retenue car ses résultats visuels sont satisfaisants pour un coût en temps de calcul très faible. Les résultats restent cependant assez différents de méthodes plus exactes, comme les Record&Sort.

9.6.4 Autres techniques

Nvidia propose une application permettant de tester et de comparer différentes méthodes de transparence : `vk_order_independent_transparency`[54]. Fonctionnant avec Vulkan, il faut s'attendre à ce que les résultats visuels ainsi que le temps de calcul sur GPU restent identiques, mais que le temps passé sur CPU puisse changer par rapport à une application OpenGL. [55] L'application propose quelques méthodes, notamment un dessin basique, sans tri, la méthode Weighted Blended, et différentes méthodes de type Record&Sort. On peut voir ces résultats en figure 9.12 ainsi que des tests dans des cas extrêmes en figure A.7.2

Screen-Door transparency et dérivés La Screen-Door transparency simule de la transparence en arrêtant le dessin d'une partie des fragments d'un objet selon un motif spécifique. Grâce à cela, on peut dessiner des objets faussement transparents sans avoir à les gérer différemment, pour un coût faible. De nos jours, cette technique est généralement utilisée pour faire partiellement disparaître un objet qui cache la ligne de vue.[56] On peut voir cette méthode en annexe, figure A.7.3. On trouve plusieurs modifications de cet algorithme :

- Cheesy translucency : Utilisation d'un motif aléatoire.

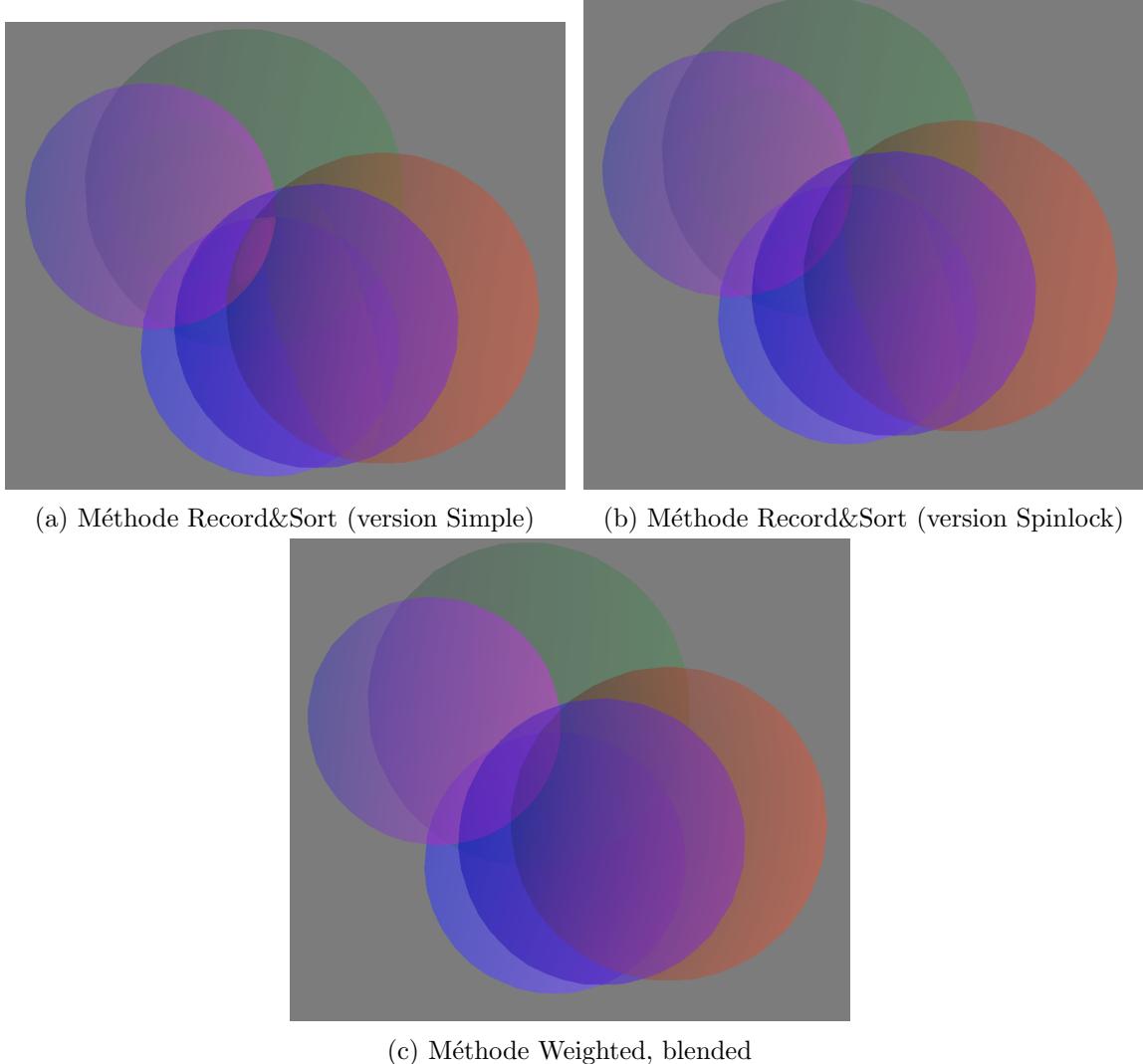


Figure 9.12: Exemple des résultats en fonction des méthodes. On peut voir que la méthode simple cause un artefact.

- Alpha to coverage : L'algorithme travaille sur les échantillons du MSAA.
- Transparence stochastique : Mélange et amélioration des 2 techniques précédentes.[57]

Il existe un certain nombre d'autres méthodes qui n'ont pas été comparées, comme les méthodes basées "moment"[58], qui sont une évolution de Weighted Blended, le lancer de rayon, la transparence phénoménologique[59].

9.7 Organisation des données

La pipeline d'OpenGL en *mode conservé* laisse le contrôle à l'utilisateur sur l'agencement de ses données. C'est à lui de choisir si les données sont entrelacées, si elles sont dans le même buffer, si plusieurs objets cohabitent dans le même buffer... Il n'existe pas de solution constamment meilleure que les autres, et il est donc intéressant de laisser la possibilité à l'utilisateur de choisir son format.

Dans les premiers mois du stage, ces formats étaient écrits plus ou moins en "dur", avec la création d'un nouveau format pour chaque "brique de dessin", et ce format était défini via des fonctions OpenGL directement. Cela a plusieurs défauts :

1. Cela génère de la duplication de code, et il faut s'assurer de la cohérence d'une brique à l'autre.
2. Changer de format de données n'est pas une opération gratuite, et il est impossible de détecter une duplication du format.

3. La validité et cohérence du format (les bonnes données sont chargées au bon endroit, il n'y a pas de données qui se superposent, le format correspond au bon buffer, ...) est vérifiée par OpenGL, qui va bien souvent accepter un format "bancal".
4. Il n'est pas possible d'optimiser l'agencement des données (par exemple placer une donnée dans le padding d'une autre) via le programme : le travail doit être réalisé à la main
5. Il n'est pas possible de vérifier la cohérence entre le format déclaré par un VAO et un shader program. À nouveau, OpenGL va généralement accepter cela sans signaler une erreur.
6. Partager les données entre plusieurs éléments nécessite de réécrire le format et le chargement des données.

Il y avait donc un besoin de représenter ces formats sous forme d'une structure de données, et de contrôler le chargement des données avec un format.

Également, cela a permis de simplifier la mise à jour des données sur la carte graphique, qui était un point sur lequel le second stagiaire avait eu du mal, ce qui l'a mené à juste recréer l'objet à chaque *trame*.

Cette partie était très ambitieuse, et peut être même trop. En plus de demander de nombreuses fonctionnalités, j'ai fait le choix de prendre cette occasion pour monter en compétences en métaprogrammation C++.

Compte tenu du temps passé sur cette partie, il est intéressant de survoler le travail réalisé.

9.7.1 Structure des classes

Une partie de la structure des classes ainsi que les évolutions proposées s'inspirent de ce qui se fait dans des API plus modernes, comme Vulkan ou Direct3D12. Voici les briques majeures du système créé.

InputElementDesc Il s'agit d'une structure représentant le placement d'un "élément de donnée" (par exemple, 3 flottants pour une position ou 2 flottants pour une coordonnée de texture) dans un groupe de *buffers*. Elle ne dépend pas d'un buffer spécifique, le moins possible des autres InputElementDesc et est réutilisée peu importe le nombre d'éléments dans un buffer. Elle contient également une étiquette afin de qualifier le type de donnée désigné (position, normal, ...). Il en existe plusieurs préconstruites dans le programme. Les InputElementDesc composent l'élément suivant

InputLayout Cette structure est une agrégation d'InputElementDesc, censée représenter l'intégralité des entrées d'un programme. À nouveau, il en existe plusieurs préconstruites. Un ou plusieurs de ces formats sont utilisés pour former les BufferRequirements.

BufferRequirements Cette structure sert à définir le format des buffers qui devront être créés : stride⁽¹⁾, taille totale, validité. On peut le construire en mélangeant plusieurs bufferRequirements, et il se chargera de vérifier leur compatibilité. L'intérêt de cette structure est qu'elle permet de fusionner différentes "vues" sur des données similaires. Par exemple, on peut vouloir utiliser les mêmes positions pour dessiner à la fois des lignes et des points, cette structure permet de s'assurer que les formats demandés sont compatibles, et de donner l'agencement des données. Cet agencement de données sera utilisé par les CPUBufferCollection.

CPUBufferCollection Cette structure contient les différents buffers devant être envoyés à la carte graphique : il s'agit donc d'un ensemble de données brutes associées à un format, et contenant les informations nécessaires à l'envoi et la mise à jour des données sur GPU, en créant un GPUSideBuffers.

GPUSideBuffers Il s'agit d'un regroupement d'informations permettant de retrouver les différents buffers sur la carte graphique ainsi que les informations pour les attacher au prochain appel de dessin. C'est la structure utilisée par la classe suivante pour mettre en place la pipeline de dessin.

GLData Finalement, on retrouve une dernière structure contenant les informations nécessaires au dessin d'un objet. C'est cette structure qui est chargée de lancer l'appel de dessin.

Synchronisable Il s'agit d'une classe n'intervenant pas dans le dessin, mais servant à simplifier la gestion de la mise à jour des données.

⁽¹⁾ La stride correspond au nombre de bits qui sépare 2 "données" dans un buffer ou un tableau.

```

1 Synchronisable> sync;
2 ...
3 sync->size(); // OK, modifiable
4 //Ne compile pas, la valeur synchronisée est constante
5 //sync->clear();
6
7 {
8 //Recuperation d'une version modifiable des donnees
9 auto editablePositions = sync.setEditable();
10
11 for (auto& point : editablePositions)
12 point += {0., 0., 0.};
13 }//Destruction de editablePositions, resynchronisation des donnees

```

Figure 9.13: Utilisation de Synchronisable

Une des difficultés lors de l'utilisation de la bibliothèque était la présence de données sur CPU et sur GPU, sans garantie que les 2 soient équivalents. L'objectif était donc de créer une structure imposant la synchronisation des données CPU-GPU. Cependant, l'écriture d'une structure imposant cette compatibilité sans impacter son utilisation n'ayant pas été réussie, sa création reste optionnelle. On peut voir un exemple de manipulation en figure 9.13

9.7.2 Résultats

Ayant eu à modifier les entrées de certains de mes programmes après la création de ce système, je peux affirmer qu'il m'a considérablement simplifié la tâche. J'ai pu très rapidement rajouter des données à transférer à mes programmes, les remplir et les changer d'emplacement.

9.8 Intégration d'une souris 3d

Une des tâches de ce stage a été l'intégration de contrôles via une souris 3D. Cette tâche a été réalisée conjointement avec l'autre stagiaire.

9.8.1 Souris 3D

Une souris 3D est une souris offrant un contrôle à 6 degrés de liberté. La souris utilisée était la SpaceMouse Wireless[60], de 3DConnexion. Elle se présente sous la forme d'un cylindre que l'on saisit à une main et que l'on peut pousser ou faire pencher sur 2 axes, soulever ou enfoncez ou bien faire tourner. Toutes ces actions réunies permettent de contrôler 6 degrés de liberté de manière efficace, intuitive et ergonomique. La souris utilisée possédait également 2 touches de contrôle.

9.8.2 Intégration à l'application

Le kit de développement de la souris contient de la documentation, un court tutoriel pour commencer à utiliser la souris, des exemples ainsi que les bibliothèques en C++ et C# pour intégrer la souris.

L'API pour interagir avec la souris est haut niveau : il suffit de définir une classe équipée de getters et de setters pour les différentes propriétés de la souris, puis de démarrer la navigation 3D. Lors de la lecture des entrées utilisateur, le driver de la souris se chargera tout seul d'appeler ces fonctions afin de réaliser ses actions. Elle a donc été intégrée très facilement au système d'entrée utilisateur.

La souris ne propose pas directement de récupérer les entrées utilisateur. Elle lit la matrice de vue qu'on lui fournit, et la retourne modifiée. Elle demande donc également des informations variées, telles que la composition de la matrice de projection, la boîte englobante de la scène, le centre de rotation... Grâce à cela, contrôler la caméra uniquement avec



Figure 9.14: La SpaceMouse Wireless de 3DConnexion [60]

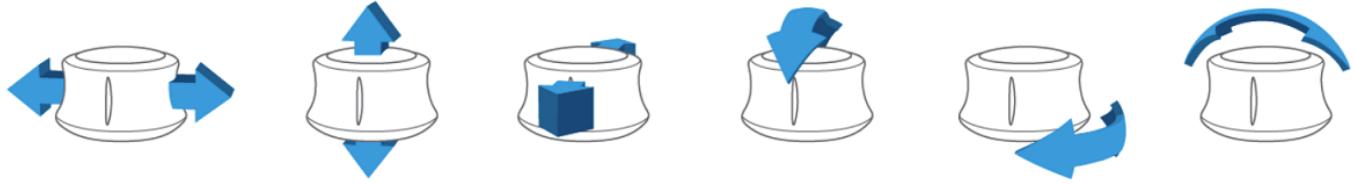


Figure 9.15: Les contrôles de la SpaceMouse [61]

la souris 3D est très simple, et le contrôle reste identique d'une application à l'autre. Il y a même un système de prise de vue permettant de prendre une position définie (haut, face, coté...) avec une animation pour rejoindre la vue.

Cependant, cette interface haut niveau a posé quelques difficultés à plusieurs reprises, notamment lorsqu'il a fallu la faire cohabiter avec le contrôle souris de la caméra, qui régénérait une nouvelle matrice de vue à partir de différents paramètres. Il a donc fallu extraire ces données de la matrice renvoyée par le driver. Cette interface a aussi compliqué la tâche de comprendre le fonctionnement du système.

9.9 Rendu basé physique

Afin de pouvoir proposer un rendu de meilleure qualité, un système d'éclairage basé physique a été développé. Compte tenu qu'il a simplement fallu récupérer les formules déjà écrites, l'implémentation a été très rapide, mais le sujet reste intéressant à présenter.

9.9.1 Principe du rendu basé physique

Contrairement à des modèles d'éclairage tels que Blinn-Phong[62], les modèles basés physiques essayent de donner un rendu s'approchant de la réalité en s'inspirant de la physique, bien que passant par de nombreuses approximations. Ces modèles respectent trois règles :

1. Être basé sur le modèle microfacette.
2. Respecter la loi de conservation de l'énergie
3. Utiliser une *Bidirectional reflective distribution function (BRDF)* basée sur la physique.

Le modèle microfacette Ce modèle suppose qu'une surface peut être considérée comme un ensemble de petites facettes dont l'orientation varie. Lorsqu'un rayon percute la surface, on considère donc qu'il a une chance de percuter une microfacette n'ayant pas la même orientation que la surface qu'elle compose[63].

La loi de conservation de l'énergie Il s'agit simplement du fait qu'un objet **non émissif** ne change pas la quantité d'énergie dans la scène.

La Bidirectional reflective distribution function La *Bidirectional reflective distribution function (BRDF)*, ou fonction de distribution de la réflectance bidirectionnelle est une fonction qui donne la réflectance du matériau en fonction de la direction d'origine de l'illumination et la direction d'observation[64].

9.9.2 Modèle *PBR* utilisé

Il existe une grande variété de modèles répondant aux contraintes énoncées précédemment. Le modèle implémenté est celui présenté dans [13], qui est basé sur le modèle publié par Disney [65], et dépend de la *BRDF* de Cook-Torrance [66].

La formule générale se trouve en figure 9.16a, cependant, notre application n'utilisant pas de *lumière surfacique*⁽¹⁾, on peut simplifier la formule en transformant l'intégrale en somme. La formule a également été adaptée au contexte de notre application. Elle est notée en figure 9.16b. On note p le point sur lequel on

⁽¹⁾ Une lumière surfacique est une lumière qui a un nombre infini de points émettant de la lumière.

On note $R = direction(p, light)$, $I = intensity(p, light)$

$$L_o(p, \omega_o) = \int_{\Omega} f(p, \omega_i, \omega_o) L_i(p, \omega_i) n \cdot \omega_i d\omega_i$$

$$L_o(p, \omega_o) = \sum_{light}^{AllLights} f(p, R, \omega_o) n \cdot R * I$$

(a) Equation de la réflectance "générale"

(b) Equation de la réflectance adaptée

$$f(p, R, \omega_o) = k_d \frac{c}{\pi} + \frac{DFG}{4(\omega_0 \cdot n)(R \cdot n)}$$

(c) BRDF de Cook-Terrance. D, F et G représentent chacun une partie des propriétés de la surface.

Figure 9.16: Equations utilisées

calcule l'*irradiance*⁽¹⁾, ω_o la direction de vue, n la normale de la surface en p , *direction* la fonction calculant la direction entre p et la lumière, et *intensity* la fonction calculant l'intensité de la lumière au point p . *direction* et *intensity* dépendent du type de lumière.

Dans l'équation 9.16c,

- D est la distribution des normales, et modélise l'irrégularité de la surface. Nous utilisons Trowbridge-Reitz GGX[67] pour cette composante.
- G est la fonction de géométrie, et modélise le fait que certaines microfacettes peuvent en cacher d'autres. Nous utilisons Smith's Schlick-GGX.
- F est l'équation de Fresnel, elle donne le ratio de réflexion en fonction de l'angle d'observation. Nous utilisons l'approximation de Fresnel-Schlick[68].

Les fonctions utilisées sont celles utilisées sur Unreal Engine 4, et présentées dans [13]. Aucune test avec des alternatives n'a été réalisé, mais les résultats obtenus sont déjà satisfaisants, en termes d'apparence et de performance. Des alternatives sont présentées dans [69].

9.10 Autres détails

D'autres fonctionnalités annexes ont été développées.

- Différent types de lampes (ponctuelle, directionnelle, spot).
- Un système de changement de vue avec animation.
- Le dessin d'une grille pour le sol.
- Un repère composé de flèches épaisses et de traits, qui affiche uniquement les lignes lorsque les flèches épaisses sont cachées.
- Des fonctionnalités pour mieux visualiser les trajectoires, notamment en n'affichant qu'une section selon un volume de découpe ou bien en utilisant la distance sur la courbe.
- différents arrière-plans, avec gradients, couleur unie, texture proposant différents paramètres, *skybox*.
- Du normal mapping, pour donner l'impression qu'il y a plus de détails que ce que le maillage permet.
- Un calcul de boîte englobante alignée sur les axes de la scène, en utilisant un calcul sur plusieurs threads.

⁽¹⁾Dans ce rapport, l'irradiance est la quantité de lumière reçue sur une surface.

10. Evaluation des résultats, retour d'expérience

Le travail réalisé correspond aux attentes. Il propose de nombreuses améliorations par rapport à l'existant. Notamment, celui-ci propose des performances nettement supérieures. Également, de nombreuses fonctionnalités non-disponibles dans l'ancien moteur sont disponibles, comme l'utilisation de plusieurs polices différentes, l'affichage d'objets transparents, et bien évidemment un rendu programmable via des shaders. Il est aussi possible de prendre des captures d'écran et d'utiliser un rendu ayant une meilleure qualité afin de fournir des images pour les rapports produits par SWOOD Design.

Le développement d'un code prenant compte de la maintenabilité et la production de documentation me rassurent sur le fait que ce code pourra être réutilisé, amélioré et intégré.

Tout le travail prévu n'a pas été réalisé, cependant plusieurs propositions d'extension ont été faites, ainsi que d'autres éléments prévus initialement.

J'aurais souhaité de mieux me renseigner sur l'état de l'art au début du stage et de directement évaluer la possibilité d'utiliser les dernières versions d'OpenGL. Cela m'aurait permis de profiter au maximum des capacités de l'ordinateur. Cependant, un moteur respectant les recommandations les plus récentes risquerait de rendre sa maintenance plus compliquée.

Compte tenu qu'OpenGL était très peu compatible avec le calcul multi-thread, celui-ci n'a pas été activement pris en compte lors du développement. Cependant, il s'avère que les versions récentes d'OpenGL ont certaines fonctionnalités permettant l'utilisation de plusieurs threads et certains autres calculs ne dépendant pas d'OpenGL pourraient également être passés en calcul multi thread. Si on souhaite accélérer le rendu en utilisant plusieurs threads, il faudra s'assurer que cela ne cause pas de problème.

Une partie censée être conséquente qui n'a pas été réalisée est la partie WebGL. Six semaines avaient été prévues pour la réaliser, mais nous n'y avons pas touché. L'objectif était de pouvoir afficher les modèles en 3D dans un navigateur et de pouvoir interagir avec. J'aurais souhaité tester l'utilisation de WebAssembly pour réutiliser au maximum le code C++ dans des clients légers, et combiner cela avec OpenGL ES ou WebGPU pour le rendu, dans l'idée de ne pas avoir à dupliquer le code.

Plusieurs difficultés majeures ont été rencontrées dans ce stage. Tout d'abord, différentes versions d'OpenGL existent, et chaque nouvelle version apporte une couche de complexité supplémentaire. A cause de cela, on retrouve un grand nombre d'articles et de messages sur des forums, même récents, faisant référence à des méthodologies obsolètes. Également, n'étant pas suffisamment expérimenté avec OpenGL, il m'a été difficile de faire des choix de structure de code, et nombreux sont ceux qui se sont avérés être mauvais.

11. Conclusion

Lors de ce stage, j'ai pu démontrer l'utilité, en termes de performance et de personnalisation, de la création d'un nouveau moteur 3D pour EFICAD. J'ai pu poser les bases de ce moteur et rajouter de nombreuses fonctionnalités répondant à des problématiques réelles. J'ai eu l'occasion d'apprendre sur un domaine qui me passionne, et de découvrir les nombreuses manières de faire. Mon stage à EFICAD m'a également permis d'observer la vie d'une PME, et de découvrir les métiers du bois.

Je suis satisfait du travail produit, bien que celui-ci soit perfectible, que ce soit en terme de facilité d'utilisation, de stabilité ou de performance. Les notes prises ainsi que ce rapport pourront servir lorsqu'EFICAD souhaitera améliorer ce moteur ou en créer un de "qualité professionnelle".

Ce stage m'a également convaincu de mon choix d'orientation, et j'ai bien l'intention de continuer à travailler dans ce domaine.

A. Annexes

A.1 Bibliographie

- [1] Hongjin Fang et al. “Frame Loss Effects on Visual Fatigue in Super Multi-View 3D Display Technology”. en. In: *Electronics* 13.8 (Jan. 2024). Number: 8 Publisher: Multidisciplinary Digital Publishing Institute, p. 1461. ISSN: 2079-9292. DOI: 10.3390/electronics13081461. URL: <https://www.mdpi.com/2079-9292/13/8/1461> (visited on 08/12/2024).
- [2] aviviano. *What is a Driver? - Windows drivers.* en-us. Dec. 2023. URL: <https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-> (visited on 08/21/2024).
- [3] Everitt Cass et al. *Approaching zero driver overhead.* fr. Mar. 2014. URL: <https://www.slideshare.net/slideshow/approaching-zero-driver-overhead/32554457> (visited on 08/14/2024).
- [4] *First look at the debugger - Visual Studio (Windows) — Microsoft Learn.* URL: <https://learn.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour?view=vs-2022> (visited on 08/13/2024).
- [5] *First look at profiling tools - Visual Studio (Windows) — Microsoft Learn.* URL: <https://learn.microsoft.com/en-us/visualstudio/profiling/profiling-feature-tour?view=vs-2022> (visited on 08/13/2024).
- [6] *Build Insights Now Available in Visual Studio 2022 - C++ Team Blog.* URL: <https://devblogs.microsoft.com/cppblog/build-insights-now-available-in-visual-studio-2022/> (visited on 08/13/2024).
- [7] *Doxygen homepage.* URL: <https://www.doxygen.nl/> (visited on 08/13/2024).
- [8] *Notion.* URL: <https://www.notion.so> (visited on 08/13/2024).
- [9] *Microsoft Loop: Collaborative App — Microsoft 365.* URL: <https://www.microsoft.com/en-us/microsoft-loop> (visited on 08/13/2024).
- [10] *Video Conferencing, Meetings, Calling — Microsoft Teams.* en-US. URL: <https://www.microsoft.com/en-us/microsoft-teams/group-chat-software> (visited on 08/13/2024).
- [11] *Preface: What is OpenGL? — OpenGLBook.com.* URL: <https://openglbook.com/chapter-0-preface-what-is-opengl.html> (visited on 08/14/2024).
- [12] Richard Jr. Wright S. *OpenGL is not dead, long live Vulkan.* en-US. Apr. 2023. URL: <https://accidentalastro.com/2023/04/opengl-is-not-dead-long-live-vulkan/> (visited on 08/14/2024).
- [13] *LearnOpenGL - Theory.* URL: <https://learnopengl.com/PBR/Theory> (visited on 08/21/2024).
- [14] *Fragment Shader.* Nov. 2020. URL: https://www.khronos.org/opengl/wiki/Fragment_Shader (visited on 08/19/2024).
- [15] *Rasterization.* URL: <https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-practical-implementation.html> (visited on 08/30/2024).
- [16] *Vertex shader.* Nov. 2017. URL: https://www.khronos.org/opengl/wiki/Vertex_Shader (visited on 08/19/2024).
- [17] *Vertex Specification.* July 2024. URL: https://www.khronos.org/opengl/wiki/Vertex_Specification (visited on 08/19/2024).

- [18] *Tessellation*. Oct. 2020. URL: <https://www.khronos.org/opengl/wiki/Tessellation> (visited on 08/19/2024).
- [19] *Geometry Shader*. Nov. 2022. URL: https://www.khronos.org/opengl/wiki/Geometry_Shader (visited on 08/19/2024).
- [20] *Rendering Pipeline Overview*. Nov. 2022. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (visited on 08/19/2024).
- [21] *Vertex Post-Processing*. Feb. 2021. URL: https://www.khronos.org/opengl/wiki/Vertex_Post-Processing (visited on 08/19/2024).
- [22] *Transform Feedback*. July 2024. URL: https://www.khronos.org/opengl/wiki/Transform_Feedback (visited on 08/19/2024).
- [23] *Primitive Assembly*. Sept. 2022. URL: https://www.khronos.org/opengl/wiki/Primitive_Assembly (visited on 08/19/2024).
- [24] *Clipping*. Feb. 2021. URL: https://www.khronos.org/opengl/wiki/Vertex_Post-Processing#Clipping (visited on 08/19/2024).
- [25] *Face Culling*. Oct. 2019. URL: https://www.khronos.org/opengl/wiki/Face_Culling (visited on 08/19/2024).
- [26] *Per-Sample Processing*. Apr. 2019. URL: https://www.khronos.org/opengl/wiki/Per-Sample_Processing (visited on 08/19/2024).
- [27] *SIMD - MDN Web Docs Glossary: Definitions of Web-related terms — MDN*. en-US. July 2024. URL: <https://developer.mozilla.org/en-US/docs/Glossary/SIMD> (visited on 09/06/2024).
- [28] *Compute Shader*. Wiki. Apr. 2019. URL: https://www.khronos.org/opengl/wiki/Compute_Shader.
- [29] *History of OpenGL*. Wiki. Feb. 2022. URL: https://www.khronos.org/opengl/wiki/History_of_OpenGL#Deprecation_Model.
- [30] mhalber. *GitHub - mhalber/Lines: 6 different implementations of doing wide line rendering in OpenGL*. URL: <https://github.com/mhalber/Lines> (visited on 08/20/2024).
- [31] Joshua. *Why Geometry Shaders Are Slow (Unless you're Intel) – The Burning Basis Vector*. en-US. Mar. 2015. URL: <http://www.joshbarczak.com/blog/?p=667> (visited on 08/14/2024).
- [32] *glLineStipple*. URL: <https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/glLineStipple.xml> (visited on 08/14/2024).
- [33] *Chapter 39. Parallel Prefix Sum (Scan) with CUDA*. en-US. URL: <https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda> (visited on 09/02/2024).
- [34] T.Mariah Khayat. *Drawing lines using OpenGL*. URL: https://drive.uqu.edu.sa/_/mskhayat/files/MySubjects/2019SumS_ComputerGraphics/2019SSCGLABLectures/CGLABLectureThree.pdf.
- [35] Spatial Team. *The Main Benefits and Disadvantages of BRep Modeling*. en-us. URL: <https://blog.spatial.com/the-main-benefits-and-disadvantages-of-brep-modeling> (visited on 08/20/2024).
- [36] archwiki. *Fonts - ArchWiki*. URL: <https://wiki.archlinux.org/title/fonts> (visited on 06/10/2024).
- [37] Prof. Girish Dalvi. *Digital Type: Raster Fonts*. en. Text. Last Modified: 2015-08-03T22:26+05:30. Aug. 2015. URL: <https://www.dsource.in/course/digital-typography-1/digital-type-raster-fonts> (visited on 07/31/2024).
- [38] Roger D. Hersch. “Font Rasterization: The State of the Art”. en. In: *From Object Modelling to Advanced Visual Communication*. Ed. by Sabine Coquillart, Wolfgang Straßer, and Peter Stucki. Berlin, Heidelberg: Springer, 1994, pp. 274–296. ISBN: 978-3-642-78291-6. DOI: [10.1007/978-3-642-78291-6_9](https://doi.org/10.1007/978-3-642-78291-6_9).
- [39] Laura Keung. *Different Font File Types Explained (OTF, TTF, WOFF) — Envato Tuts+*. Nov. 2023. URL: <https://design.tutsplus.com/articles/different-font-file-types-explained-ott-ttf-woff--cms-39047> (visited on 06/10/2024).

- [40] Chris Green. “Improved alpha-tested magnification for vector textures and special effects”. en. In: *ACM SIGGRAPH 2007 courses*. San Diego California: ACM, Aug. 2007, pp. 9–18. ISBN: 978-1-4503-1823-5. DOI: 10.1145/1281500.1281665. URL: <https://dl.acm.org/doi/10.1145/1281500.1281665> (visited on 06/07/2024).
- [41] Inigo Quilez. *Inigo Quilez*. en. URL: <https://iquilezles.org> (visited on 06/07/2024).
- [42] V. Chlumský, J. Sloup, and I. Šimeček. “Improved Corners with Multi-Channel Signed Distance Fields”. en. In: *Computer Graphics Forum* 37.1 (2018). eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13265>, pp. 273–287. ISSN: 1467-8659. DOI: 10.1111/cgf.13265. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13265> (visited on 06/07/2024).
- [43] Will Dobbie. *GPU text rendering with vector textures* · Will Dobbie. URL: <https://wdobbie.com/post/gpu-text-rendering-with-vector-textures/> (visited on 06/10/2024).
- [44] Viktor Chlumský. *Chlumsky/msdf-atlas-gen*. original-date: 2020-03-07T10:53:56Z. June 2024. URL: <https://github.com/Chlumsky/msdf-atlas-gen> (visited on 06/10/2024).
- [45] *FreeType Glyph Conventions — Glyph Metrics*. URL: <https://freetype.org/freetype2/docs/glyphs/glyphs-3.html> (visited on 06/10/2024).
- [46] CocosMarketing. *An Introduction To Draw Call Performance Optimization - Knowledge base*. en. Apr. 2022. URL: <https://discuss.cocos2d-x.org/t/an-introduction-to-draw-call-performance-optimization/55852> (visited on 06/10/2024).
- [47] ALSTRÖM MARCUS. *Comparative study of Batch and Instance rendering for static geometry in OpenGL*. English. June 2023. URL: <https://www.diva-portal.org/smash/get/diva2:1779215/FULLTEXT01.pdf>.
- [48] Thomas Porter and Tom Duff. “Compositing digital images”. en. In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. ACM, Jan. 1984, pp. 253–259. ISBN: 978-0-89791-138-2. DOI: 10.1145/800031.808606. URL: <https://dl.acm.org/doi/10.1145/800031.808606> (visited on 07/15/2024).
- [49] Morgan McGuire and Louis Bavoil. “Weighted Blended Order-Independent Transparency”. en. In: 2.2 (2013).
- [50] A. Mammen. “Transparency and antialiasing algorithms implemented with the virtual pixel maps technique”. In: *IEEE Computer Graphics and Applications* 9.4 (July 1989). Conference Name: IEEE Computer Graphics and Applications, pp. 43–55. ISSN: 1558-1756. DOI: 10.1109/38.31463. URL: <https://ieeexplore.ieee.org.passerelle.univ-rennes1.fr/document/31463/?arnumber=31463> (visited on 07/17/2024).
- [51] Louis Bavoil and Kevin Myers. “DualDepthPeeling”. en. In: (2008).
- [52] Christoph Kubisch. *Order Independent Transparency In OpenGL 4.x*. Apr. 2023. URL: <https://web.archive.org/web/20230430190951/https://on-demand.gputechconf.com/gtc/2014/presentations/S4385-order-independent-transparency-opengl.pdf> (visited on 08/02/2024).
- [53] Moghaddam Mahan Heshmat. *LearnOpenGL - Weighted Blended*. URL: <https://learnopengl.com/Guest-Articles/2020/OIT/Weighted-Blended> (visited on 08/02/2024).
- [54] *nvpro-samples/vk_order_independent_transparency*. original-date: 2020-06-01T20:47:13Z. July 2024. URL: https://github.com/nvpro-samples/vk_order_independent_transparency (visited on 07/18/2024).
- [55] *Transitioning from OpenGL to Vulkan*. en-US. URL: <https://developer.nvidia.com/transitioning-opengl-vulkan> (visited on 08/02/2024).
- [56] *Screen-Door Transparency*. URL: <https://digitalrune.github.io/DigitalRune-Documentation/html/fa431d48-b457-4c70-a590-d44b0840ab1e.htm> (visited on 08/19/2024).
- [57] Eric Enderton et al. “Stochastic Transparency”. In: (2010).
- [58] Cedrick Müntermann et al. “Moment-Based Order-Independent Transparency”. en. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1.1 (July 2018), pp. 1–20. ISSN: 2577-6193. DOI: 10.1145/3203206. URL: <https://dl.acm.org/doi/10.1145/3203206> (visited on 08/02/2024).

- [59] Morgan McGuire and Michael Mara. “Phenomenological Transparency”. en. In: *IEEE Transactions on Visualization and Computer Graphics* 23.5 (May 2017), pp. 1465–1478. ISSN: 1077-2626, 1941-0506, 2160-9306. DOI: 10.1109/TVCG.2017.2656082. URL: <https://ieeexplore.ieee.org/document/7828148/> (visited on 08/20/2024).
- [60] *SpaceMouse Wireless - Bluetooth Edition*. en-US. URL: <https://3dconnexion.com/dk/product/spacemouse-wireless/> (visited on 07/24/2024).
- [61] “Test Report: Ergonomic Evaluation of 3D Mice”. en. In: () .
- [62] James F. Blinn. “Models of light reflection for computer synthesized pictures”. en. In: *ACM SIGGRAPH Computer Graphics* 11.2 (Aug. 1977), pp. 192–198. ISSN: 0097-8930. DOI: 10.1145/965141.563893. URL: <https://dl.acm.org/doi/10.1145/965141.563893> (visited on 08/20/2024).
- [63] Matt Pharr, Jakob Wenzel, and Greg Humphreys. *Microfacet Models*. Livre. URL: https://www.pbr-book.org/3ed-2018/Reflection_Models/Microfacet_Models (visited on 08/20/2024).
- [64] UMass Boston. *Modis - UMass Boston*. en. URL: <https://www.umb.edu/spectralmass/terra-aqua-modis/modis/> (visited on 08/20/2024).
- [65] Brent Burley. “Physically Based Shading at Disney”. en. In: () .
- [66] Robert L Cook. “A Reflectance Model for Computer Graphics”. en. In: *ACM Transactions on Graphics* 1.1 () .
- [67] Bruce Walter et al. “Microfacet Models for Refraction through Rough Surfaces”. en. In: () .
- [68] *An Inexpensive BRDF Model for Physically-based Rendering - Schlick - 1994 - Computer Graphics Forum - Wiley Online Library*. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1330233> (visited on 08/21/2024).
- [69] Brian Karis. *Graphic Rants: Specular BRDF Reference*. Aug. 2013. URL: <https://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html> (visited on 08/21/2024).
- [70] *COMP3421 Immediate Mode vs Retained Mode*. URL: https://www.cse.unsw.edu.au/~cs3421/14s2/lectures/11_ImmediateMode.pdf
- [71] David Stutz. *A Formal Definition of Watertight Meshes* • David Stutz. en-US. Jan. 2018. URL: <https://davidstutz.de/a-formal-definition-of-watertight-meshes/> (visited on 08/20/2024).
- [72] *Features*. en-US. Feb. 2009. URL: <https://www.ogre3d.org/about/features> (visited on 08/20/2024).
- [73] *OpenSceneGraph*. en. URL: <https://openscenegraph.github.io/openscenegraph.io/openscenegraph.io/> (visited on 08/20/2024).

A.2 Table des figures

6.1	Interface de visual studio 2022.	9
6.2	Outils de debuggage et de profiling graphique	10
7.1	Logo d'OpenGL	11
7.2	Les différents espaces utilisés pour projeter un objet 3D sur un écran. (X X)	12
7.3	Repère caméra (X X)	12
7.4	Exemple de graphe de scène [13]	13
7.5	Comparaison de la projection orthographique et perspective.	13
7.6	Anti-aliasing utilisant plus d'un fragment par pixel [15].	14
7.7	Diagramme de la Pipeline de Rendu. Le jaune correspond à une section configurable, le bleu à une section programmable et les pointillés à une section optionnelle[20].	15
9.1	Elements d'une polyligne.	18
9.2	Comparaison des méthodes de pointillés sur une polyligne très détaillée (100 000 segments)	19
9.3	Exemple de liens entre le motif et les pointillés[34].	20

9.4	Modèle Maillé et modèle BREP	20
9.8	Exemple de post traitement sur du texte	22
9.9	Exemples d'artefacts liés à la SDF	22
9.5	Différentes fonctions traitant les valeurs V lues dans la texture.	22
9.6	Exemple de différentes méthodes pour afficher un caractère	23
9.7	Comparaison entre les méthodes de dessin de caractère [40]	23
9.10	Placement d'un caractère sur la Baseline	24
9.11	Exemple de transparence avec Weighted Blended. On remarque qu'avec l'heuristique choisie, lorsque trop de couches sont superposées, l'algorithme échoue. Ce nombre se situe à environ 45 couches avec un buffer d'accumulation ayant 16 bits par composante.	26
9.12	Exemple des résultats en fonction des méthodes. On peut voir que la méthode simple cause un artefact.	27
9.13	Utilisation de Synchronisable	29
9.14	La SpaceMouse Wireless de 3DConnexion [60]	29
9.15	Les contrôles de la SpaceMouse [61]	30
9.16	Equations utilisées	31
A.7.1	Comparaison de différentes méthodes de pointillés, avec profondeur.	41
A.7.2	Test des méthodes de transparence dans des cas extrêmes (100 000 sphères transparentes)	42
A.7.3	Exemple de Screen-Door transparency	43

A.3 Glossaire

- buffer** Tampon : zone dans laquelle on stocke des données afin qu'un autre élément puisse y accéder.. 28
- driver** Un driver est un programme permettant la communication entre une application et du matériel[2]. Dans ce rapport, on parle du programme permettant l'interaction avec la carte graphique via OpenGL. 8, 11
- irradiance** Dans ce rapport, l'irradiance est la quantité de lumière reçue sur une surface.. 31
- lumière surfacique** Une lumière surfacique est une lumière qui a un nombre infini de points émettant de la lumière.. 30
- mode conservé** Mode d'OpenGL dans lequel les données des primitives sont envoyées au driver avant le dessin afin de pouvoir les dessiner sans les recharger [70]. 27
- pipeline** La pipeline de dessin désigne l'intégralité des opérations, programmables ou non, par lesquelles passent les données lors d'un appel de dessin.. 10
- profiling** Le fait d'analyser le système pendant son fonctionnement afin d'étudier ses performances.. 10
- quad** Un quad est un ensemble de 2 triangles formant un carré.. 18, 19, 21
- SIMD** Le SIMD est un type d'architecture permettant d'appliquer la même opération sur plusieurs données en même temps.[27]. 15
- skybox** Une skybox est une boîte qui englobe le monde afin de générer un arrière plan "à l'infini".. 31
- stride** La stride correspond au nombre de bits qui sépare 2 "données" dans un buffer ou un tableau. . 28
- trame** Une Trame(ou Frame en anglais) est l'image finale produite pour être affichée.. 10, 28
- étanchéité** Un modèle étanche (watertight) est un modèle n'ayant pas de "trou", et qui possède un intérieur clairement défini [71]. 20

A.4 Acronymes

API Application Program Interface. 5, 8, 11, 40, 41

BIOS Basic Input Output System. 21

BRDF Bidirectional reflective distribution function. 30

BREP Boundary Representation. 20

CAM Computer-Aided Manufacturing. 7

CAO Conception Assistée par Ordinateur. 7, 18, 20

FAO Fabrication Assistée par Ordinateur. 7, 18

GLSL OpenGL Shading Language. 10, 13, 40

GPGPU General Purpose Computing on Graphics Processing Units. 16

MSVC Microsoft Visual C++. 9

NDC Normalized Device Coordinates. 12

PBR Physically-based rendering. 5, 30

PME Petite et moyenne entreprise. 33

SDK Source Development Kit. 11

SIMD Single Instruction Multiple Data. 15

SVG Scalable Vector Graphics. 21

VAO Vertex Array Object. 28

A.5 Bibliothèques utilisées

Le code écrit dépend d'un certain nombre de bibliothèques/*API*

- OpenGL : Il s'agit de la base du stage.
- Dear ImGui : Cette bibliothèque sert à générer des interfaces utilisateur en mode "immédiat", et a été utilisée pour instrumenter le code afin d'aider au déboggage par exemple.
- GLFW : GLFW est utilisée pour initialiser le contexte OpenGL et gérer les interactions avec Windows (gestion de la fenêtre, des entrées et événements utilisateur, ...)
- GLAD : GLAD a servi à charger les fonctions OpenGL, dont l'implémentation doit être récupérée au lancement du programme
- GLM : OpenGL Mathematics propose des fonctions mathématiques dans une interface similaire au *OpenGL Shading Language (GLSL)*. Elle a été utilisée pour réaliser des opérations mathématiques avec des vecteurs et des matrices.
- STB : Cette bibliothèque a été utilisée pour charger et enregistrer des images.
- Assimp : Pour charger les fichiers de modèles ou scènes 3D, Assimp a été utilisée. Sa principale utilité a été de pouvoir charger des modèles complexes et texturés pour tester le programme.

Un certain nombre des ces bibliothèques n'ont été utilisée que pour aider au développement, et n'ont pas vocation à terminer dans le programme final car les fonctionnalités proposées sont déjà disponibles dans SWOOD.

A.6 Choix de l'outil et alternatives

Bien que OpenGL ait été la technologie initialement proposée sur l'offre de stage, il existe des alternatives, plus ou moins haut niveau, qu'il est intéressant de présenter.

A.6.1 *API* graphiques

Tout d'abord, en termes d'*API* graphique, OpenGL n'est pas seul, et je vais présenter les alternatives majeures.

- Vulkan : Vulkan est l'*API* destinée à succéder à OpenGL. L'idée de l'utiliser a été évoquée. En effet, utiliser Vulkan permettrait d'accéder à des performances et à une personnalisation encore meilleures, tout en profitant d'un grand nombre de fonctionnalités supplémentaires. Cependant, ces gains étaient marginaux par rapport à la difficulté de développement et la quantité de travail à réaliser en utilisant Vulkan.
- Direct3D : Direct3D est l'*API* 3D de Microsoft. Le passage de la version 11 à la version 12 correspond au même changement de paradigme qu'entre OpenGL et Vulkan. Direct3D n'a pas été utilisé car il était peu probable qu'il soit utilisable avec SOLIDWORKS, qui fonctionne à l'aide d'OpenGL. De plus, Direct3D possède une communauté bien plus petite qu'OpenGL (ou Vulkan), ce qui rend l'apprentissage et recherche de ressources plus difficile.
- Metal : Metal est l'*API* graphique d'Apple. La cible de Swood étant Windows, il n'est pas possible de l'utiliser (facilement).

Par rapport à ces technologies, OpenGL possède de nombreux avantages, car il s'agit:

- De l'*API* utilisée par SOLIDWORKS, et utilisée par la version précédente de SWOOD : la compatibilité était donc garantie, et les développeurs d'EFICAD n'étaient pas sans expérience sur le sujet.
- D'une *API* mature, ayant une grande communauté, ce qui facilite la recherche de ressources pour réaliser différentes tâches.

- D'une *API* qui reste facile d'accès : comparé aux *API* les plus récentes, OpenGL est moins verbeux et plus haut niveau : le développement est donc accéléré et simplifié.

Ces *API* graphiques ne sont que des briques de base, et il existe donc différentes bibliothèques construites sur ces technologies :

A.6.2 Moteurs graphiques et de jeu

Des alternatives plus haut niveau aux *API* graphiques, comme des "moteurs de jeu" existent. Unity ou Godot en sont des exemples.

Cependant, l'intégration dans SOLIDWORKS de ce genre de système, bien que non testée, est probablement hasardeuse. Également, l'utilisation de certains de ces moteurs n'est pas gratuite.

Des alternatives situées entre L'*API* graphique et le moteur de jeu existent : des moteurs graphiques comme Ogre3D[72] ou OpenSceneGraph[73]. Ces bibliothèques proposent une surcouche d'OpenGL plus facile à manipuler et permettant de continuer à mixer des appels en OpenGL. Ces méthodes auraient pu être une alternative légère ayant plus de chances de permettre l'intégration à SOLIDWORKS, mais la possibilité de les utiliser n'a pas été étudiée au début du stage.

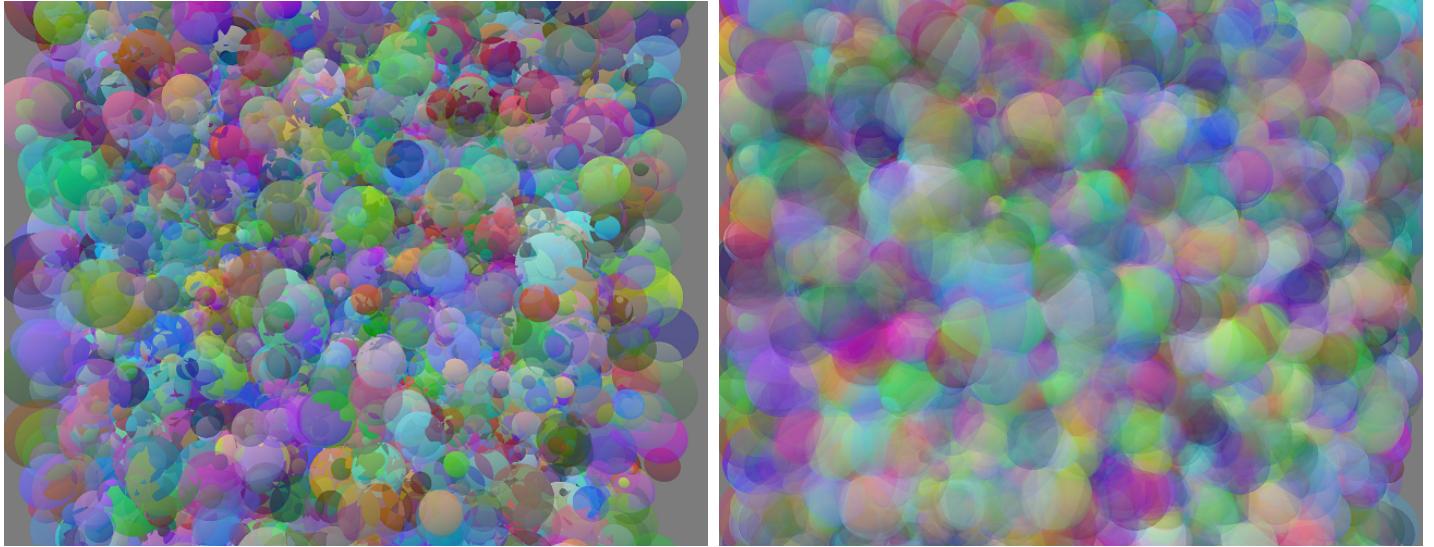
Au final, nous sommes donc restés sur la proposition initiale de développer en OpenGL. C'est la version la plus récente, OpenGL 4.6, qui a été choisie. C'est aussi celle utilisée par SOLIDWORKS, bien qu'elle soit utilisée en mode compatibilité.

A.7 Figures variées



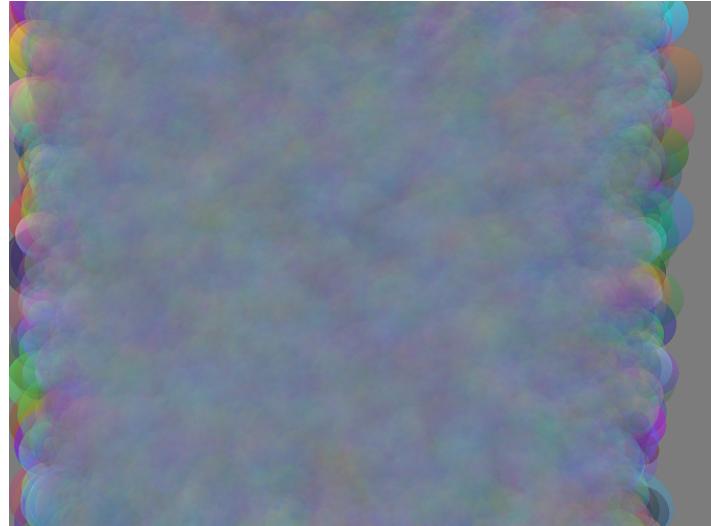
(a) Utilisation d'une distance curviligne (b) Suppression des fragments à l'aide d'un damier (c) Approximation de la courbe par une ligne (d) Pas d'historique hors du segment

Figure A.7.1: Comparaison de différentes méthodes de pointillés, avec profondeur.



(a) La méthode Record&Sort (Simple) échoue

(b) La méthode Record&Sort (Spinlock) fonctionne



(c) La méthode Weighted, Blended échoue

Figure A.7.2: Test des méthodes de transparence dans des cas extrêmes (100 000 sphères transparentes)

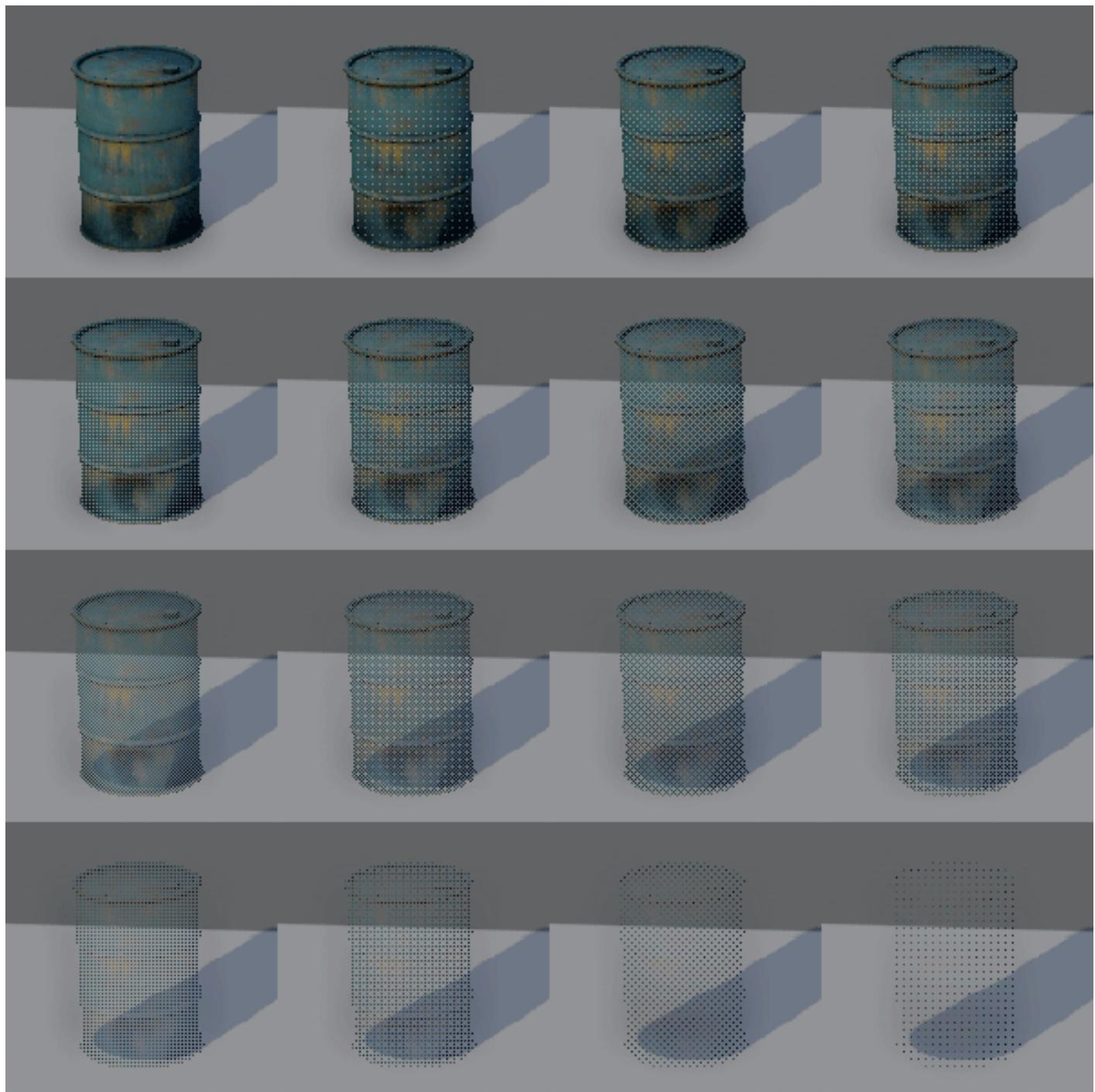


Figure A.7.3: Exemple de Screen-Door transparency