Rapport de Stage

Visualisation Interactive de Nuages de Points LiDAR sous Unreal Engine

BUT Informatique - 2e année

Stagiaire : **Lohan Boeglin**

Entreprise d'accueil : Kadran Ingénierie
Maître de stage : Frédéric Jallon
Tuteur universitaire : Sébastien Cazalas

Réalisé du 21 avril au 27 juin 2025
Année universitaire 2024–2025

IUT de Nantes — Département Informatique

Nantes — 27 juin 2025

Remerciements

J'aimerais tout d'abord remercier l'entreprise **Kadran**, et tout particulièrement **M. Frédéric Jallon**, mon maître de stage, pour m'avoir accueilli au sein de son équipe et confié ce projet. Je lui suis reconnaissant pour sa confiance, la clarté de ses orientations et la qualité de son accompagnement tout au long du stage.

Je prie également l'ensemble des collaborateurs de **Kadran** de bien vouloir recevoir mes remerciements pour leur accueil, leur disponibilité et la qualité de nos échanges. Leurs partages d'expérience et leur bienveillance ont grandement contribué à faire de ce stage une expérience formatrice et enrichissante.

Je remercie aussi **M. Sébastien Cazalas**, mon tuteur universitaire, pour son suivi régulier, ses remarques constructives et ses conseils tout au long de la rédaction de ce rapport. Je remercie également **M. Nassim Hadj Rabia** pour son implication dans l'évaluation de ce travail et de la soutenance.

Je n'oublie pas de remercier **Matthias Loste**, camarade de promotion, pour m'avoir fait découvrir cette opportunité de stage. Son aide lors des démarches préparatoires m'a été précieuse.

Enfin, j'adresse mes remerciements à l'ensemble des enseignants de l'IUT de Nantes pour leurs enseignements et leur accompagnement depuis le début de ma formation.

Résumé

Ce rapport présente le travail réalisé lors de mon stage de dix semaines, effectué en deuxième année du Bachelor Universitaire de Technologie (BUT) Informatique à l'IUT de Nantes. Cette expérience s'est déroulée au sein du pôle innovation de l'entreprise Kadran, implantée à Nantes, et s'inscrit dans le cadre d'un projet exploratoire mêlant visualisation 3D et traitement de données géospatiales LiDAR.

Le stage portait sur une étude de faisabilité de l'utilisation du moteur Unreal Engine pour la visualisation interactive de très grands nuages de points. Le travail a débuté par une phase de recherche et d'analyse documentaire sur les formats de données LiDAR et les outils existants, avant de se poursuivre par des expérimentations techniques approfondies. L'une de mes principales missions a ensuite été la conception et le développement d'une application fonctionnelle sous Unreal Engine, accompagnée de la rédaction de sa documentation technique.

This report presents the work carried out during my ten-week internship, completed in the second year of the Bachelor's degree in Computer Science (BUT Informatique) at the IUT of Nantes. The internship took place within the innovation division of Kadran, a company based in Nantes, and focused on an exploratory project combining 3D visualization and the processing of geospatial LiDAR data.

The project consisted in a feasibility study on the use of Unreal Engine for the interactive visualization of very large LiDAR point clouds. The work began with a research and documentation phase focused on LiDAR data formats and existing tools, followed by extensive technical experimentation. One of my main tasks was also to design and develop a functional application using Unreal Engine, along with writing its technical documentation.

Tous les termes suivis d'une astérisque sont définis dans le glossaire. Les acronymes y sont également listés automatiquement.

Table des matières

Introduction

Dans le cadre de ma deuxième année de BUT Informatique à l'IUT de Nantes, j'ai effectué un stage de dix semaines, du 21 avril au 27 juin. Ce stage devait permettre de mettre en pratique les compétences acquises durant la formation, tout en s'inscrivant dans une problématique en lien avec les domaines abordés en cours. J'ai choisi de réaliser ce stage chez Kadran, une entreprise spécialisée dans le traitement et la valorisation de données géospatiales, où j'ai pu travailler sur un sujet mêlant programmation, visualisation 3D et technologies avancées.

La mission qui m'a été confiée consistait à étudier la faisabilité d'un visualiser 3D* de nuage de points* volumineux en utilisant Unreal Engine*, un moteur de rendu graphique* développé initialement pour le jeu vidéo. En plus de cette étude exploratoire, j'ai eu pour objectif de mettre en œuvre un prototype fonctionnel selon les résultats obtenus. Ce sujet m'a particulièrement attiré par son aspect technique et innovant, et bien qu'il ne s'agisse pas à proprement parler de développement de jeu vidéo, le choix d'Unreal Engine ajoutait un intérêt supplémentaire pour moi, tant sur le plan personnel que professionnel.

Tout au long de cette expérience, j'ai été confronté à plusieurs difficultés liées à la complexité du sujet et aux spécificités techniques du moteur Unreal Engine. Cependant, j'ai réussi à les surmonter en m'appuyant sur les connaissances acquises durant ma formation à l'IUT de Nantes, complétées par des recherches personnelles et une démarche d'expérimentation continue.

Dans une première partie, je présenterai le contexte de mon stage et son environnement technique. Dans une seconde partie, je détaillerai les recherches, expérimentations et développements réalisés tout au long du stage. Enfin, une dernière partie proposera un bilan sur le projet, les solutions mises en place et les compétences acquises. Une conclusion viendra clore ce rapport.

Chapitre 1

Contexte et présentation des missions

1.1 Kadran

1.1.1 Constitution et fonctionnement

Le groupe Géoliance

Kadran est une entreprise intégrée au groupe **Géoliance**, un acteur structurant dans le domaine de l'aménagement du territoire, de la connaissance de l'environnement bâti et naturel, et de l'exploitation de la donnée géospatiale.

Géoliance a été fondé en 2020 avec l'objectif de regrouper sous une même bannière des compétences complémentaires, allant de la production de données (relevés terrain, photogrammétrie*, LiDAR*, etc.) à leur valorisation à travers des outils cartographiques, des systèmes d'information et des solutions métiers sur mesure.

Le groupe est structuré autour de plusieurs entités spécialisées :
— **Datageo**, spécialisée dans la collecte, le traitement et la diffusion de données géographiques ;
— **Kadran**, qui développe des outils logiciels de visualisation, de traitement et d'analyse des données ;
— **Inframind / Datakad**, basée à *Rabat* (Maroc), spécialisée dans le support à la production et le développement applicatif.

L'interaction avec les parties prenantes, essentielle à la stratégie du groupe, s'adapte en fonction des projets, des contextes techniques ou des objectifs métiers. Un schéma récapitulatif des principales sphères d'interaction est présenté en annexe (voir annexe 1).

L'entreprise Kadran

Kadran Ingénierie est une Société par Actions Simplifiée Unipersonnelle **(SASU)** fondée en décembre 2015, immatriculée au Registre du Commerce et des Sociétés de Nantes. Son siège social est basé dans l'agglomération Nantes, et l'entreprise est également implantée à **Saint-Nazaire**, **Orange** et **Rouen**, ce qui lui permet de couvrir un large territoire pour ses interventions et partenariats.

Une carte détaillée de ces implantations est présentée en annexe (voir annexe 2)

Elle dispose d'un capital social de **106,410 euros** et emploie entre **11 et 50 collaborateurs**, aux profils variés (ingénieurs, techniciens, développeurs), spécialisés dans les domaines de la géomatique*, du traitement de données géospatiales et du développement de solutions logicielles.

Activités et expertise Kadran est spécialisée dans le traitement, la valorisation et la restitution de données géospatiales, en particulier issues de technologies de télédétection* telles que le LiDAR. L'entreprise intervient sur l'ensemble de la chaîne de production : acquisition de données (captation LiDAR terrestre ou aérienne via drones, véhicules ou avions) ; traitement et classification (nettoyage, segmentation, modélisation MNT/MNS, extraction de structures) ; visualisation et exploitation (développement de solutions logicielles ou d'interfaces de visualisation 3D pour le rendu temps réel et l'analyse) ; assistance à maîtrise d'ouvrage (accompagnement des clients publics ou industriels dans la définition et la conduite de projets cartographiques ou d'infrastructure).

Clientèle et projets Kadran intervient pour le compte de nombreux donneurs d'ordre publics (collectivités territoriales, établissements publics) et privés (entreprises de l'énergie, du BTP, du transport). Les projets concernent notamment la modélisation de réseaux urbains ou ferroviaires, l'inspection et la maintenance d'infrastructures (ponts, routes, lignes électriques), la cartographie haute précision ainsi que la simulation urbaine en environnement 3D.

1.1.2 Infrastructure et environnement informatique

Les postes de travail sont récents et performants, adaptés au rendu 3D, au traitement et à la visualisation de nuage de points volumineux. Une migration progressive des données depuis des supports externes vers un système de stockage centralisé et sécurisé est en cours.

L'entreprise dispose également d'un **intranet** pour centraliser les ressources internes, les documents techniques et les outils partagés. La communication et la coordination au sein des équipes se font principalement via **Microsoft Teams**, utilisé à la fois pour les échanges quotidiens, les réunions à distance et la gestion des projets.

Parmi les outils logiciels utilisés figurent notamment : **MicroStation** et **TerraScan**, pour la classification automatique de nuage de points ; **Vision LiDAR**, pour la visualisation rapide de grands jeux de données ; **CloudCompare**, pour le nettoyage, la segmentation manuelle et la comparaison de relevés ; **AutoCAD**, pour la production de plans techniques à partir de données 3D ; et divers **environnements de développement**, les projets étant réalisés avec des IDE* différents selon les besoins ou les préférences des développeurs.

L'informatique occupe une place centrale dans l'activité de l'entreprise : traitements massifs de données LiDAR, développement de logiciels internes pour la visualisation 3D temps réel, automatisation des chaînes de traitement, gestion de bases de données métiers, etc.

1.2 Le sujet de stage

1.2.1 Contexte technique et problématique

Dans le cadre de ses activités, Kadran manipule régulièrement des **nuages de points très volumineux** issus de relevés LiDAR ou photogrammétrie. Ces fichiers peuvent contenir plusieurs dizaines, voire centaines de millions de points, ce qui rend leur visualisation difficile sans outils adaptés. Les logiciels existants comme CloudCompare ou TerraScan offrent des fonctionnalités puissantes, mais peuvent devenir complexes à utiliser, peu personnalisables, ou gourmands en ressources.

L'entreprise souhaitait donc disposer d'une solution interne, plus fluide et accessible, capable d'afficher ces données en 3D dans un environnement interactif. L'un des enjeux majeurs résidait dans la capacité à **rendre ces nuage de points de manière fluide, même sur des machines aux performances limitées**.

Le sujet de stage s'inscrivait dans cette réflexion. Il s'agissait d'une **étude exploratoire** visant à évaluer la pertinence de **l'utilisation d'Unreal Engine** pour la visualisation de nuage de points. D'autres pistes avaient été envisagées par le passé, notamment des expérimentations sous Unity*, mais jugées peu convaincantes. Unreal Engine, de son côté, avait été identifié comme une technologie prometteuse, notamment grâce à son moteur graphique avancé et à ses performances sur les gros volumes de données.

1.2.2 Objectifs du stage

Le stage avait pour objectif principal d'évaluer la faisabilité de la visualisation de nuage de points massifs dans **Unreal Engine**, en utilisant les formats standards `.las` / `.laz`*. Le sujet était volontairement ouvert et exploratoire, avec une forte dimension de recherche technique.

Les objectifs définis, et précisés au fil de l'avancement, étaient les suivants :
- Comprendre la structure des formats LiDAR et expérimenter leur import dans Unreal Engine ;
- Tester la possibilité d'un **chargement dynamique*** des nuage de points à l'exécution (runtime*) ;
- Étudier différents modes d'**intégration d'Unreal Engine dans une application** existante ou future (sous forme d'application autonome ou de composant) ;
- Vérifier l'interopérabilité* entre **Unreal Engine** et des `.dll*` externes en C# ou C++ ;
- Reproduire des contrôles de navigation inspirés d'outils professionnels comme TerraScan ;
- Permettre la personnalisation de l'affichage (classification, taille des points) lors de l'import par l'utilisateur ;
- Documenter l'architecture du projet et son code afin d'en faciliter la maintenance.

Ces objectifs ont été définis de manière progressive en fonction des possibilités offertes par Unreal Engine et des besoins identifiés au fur et à mesure du stage.

1.2.3 Méthodologie de travail

Le stage s'est déroulé selon une démarche exploratoire et itérative, caractéristique des projets de recherche et développement. Sans cahier des charges figé, les objectifs ont été définis puis ajustés au fil des tests, dans un mode de fonctionnement proche d'une méthode agile. Cette approche a permis une progression par itérations successives, tenant compte des limites techniques rencontrées et des orientations données par le tuteur en entreprise.

Chaque étape technique, qu'il s'agisse de l'importation de fichiers, de l'affichage 3D, ou de la communication entre langages, a été précédée d'une phase de documentation, suivie de prototypage, puis de validation. Cette approche a permis une grande flexibilité, en s'adaptant aux contraintes spécifiques d'Unreal Engine et aux exigences liées au traitement de données LiDAR volumineuses.

1.3 Présentation de l'environnement de travail

1.3.1 Intégration au pôle innovation

Durant mon stage, j'ai été intégré au **pôle innovation** de l'entreprise Kadran, un espace dédié à l'exploration de solutions technologiques, notamment dans les domaines de la visualisation 3D, du rendu temps réel et de l'exploitation avancée des données LiDAR. Ce pôle joue également un rôle central dans le développement de logiciels internes, en réponse aux besoins opérationnels ou en soutien aux autres équipes techniques.

FIGURE 1.1 – Poste de travail occupé durant le stage chez Kadran

Le pôle innovation est situé au deuxième étage, dans les locaux de **Géoliance**, la société holding à laquelle appartient Kadran. Les deux entités partagent cet espace, ce qui favorise les échanges transverses entre les équipes. Le bureau que j'occupais faisait partie d'un open space regroupant six personnes : mon maître de stage, deux autres développeurs confirmés, un stagiaire de l'INSA, un camarade de l'IUT également en stage, et moi-même.

J'ai travaillé en relative autonomie sur le projet qui m'était confié, tout en bénéficiant d'un encadrement régulier de mon maître de stage, avec des points hebdomadaires permettant de suivre l'avancement, d'ajuster les orientations techniques et de valider les choix réalisés. J'ai également pu m'appuyer ponctuellement sur les conseils d'autres membres du pôle innovation, en particulier pour les aspects techniques liées aux besoins métier.

Par ailleurs, plusieurs collaborateurs de Kadran ont été sollicités à différentes étapes pour tester le logiciel en développement. Leurs retours ont permis d'identifier des besoins fonctionnels, d'améliorer l'ergonomie et d'ajuster certaines fonctionnalités afin de répondre au mieux aux attentes du terrain.

1.3.2 Infrastructure matérielle et logicielle

Au sein du pôle innovation, les membres de l'équipe disposaient de configurations matérielles variables selon leurs besoins, allant de simples ordinateurs portables à des postes fixes plus puissants. Mon poste principal était un ordinateur fixe sous Windows, spécialement configuré pour le rendu 3D en temps réel et la manipulation de nuage de points volumineux. J'utilisais également mon ordinateur portable en complément, principalement pour des tâches secondaires, la consultation de documentation ou comme second écran. Cette puissance était nécessaire pour tester les limites du moteur Unreal Engine dans le cadre du projet.

Je disposais d'un compte personnel sur le réseau local de l'entreprise, me permettant d'installer librement les outils nécessaires au développement. Un disque dur externe contenant des jeux de données LiDAR m'a également été attribué afin de travailler localement, sans dépendre d'un accès distant ou d'un stockage en ligne.

Même si je n'avais pas accès aux outils collaboratifs comme **Microsoft Teams** ou à l'intranet de l'entreprise, cela n'a pas représenté de frein à ma progression. Les échanges avec mon maître de stage et les autres membres du pôle se faisaient directement sur place ou par email, de manière fluide et régulière.

Pour le développement, j'ai principalement utilisé les outils suivants : **Unreal Engine 5.5.4** [1] pour la visualisation 3D, **Visual Studio** [2] et **Visual Studio Code** [3] pour le codage et la gestion du projet, **CloudCompare** [4] et **LasTools** [5] pour le traitement des données LiDAR, **Git** [6] pour le suivi des versions, ainsi que **GIMP** [7] pour la création graphique.

1. https://www.unrealengine.com/
2. https://visualstudio.microsoft.com/
3. https://code.visualstudio.com/
4. https://www.danielgm.net/cc/
5. https://lastools.github.io/
6. https://git-scm.com/
7. https://www.gimp.org/

Chapitre 2

Travaux réalisés : recherches, expérimentations et développement

Ce chapitre retrace les différentes étapes de mon travail au cours du stage, depuis les premières phases de recherche jusqu'aux développements applicatifs concrets.

2.1 Phase de recherche et d'exploration

Dès le début du stage, l'objectif initial fixé par mon tuteur portait sur la possibilité de créer un *visualiser 3D* autonome basé sur Unreal Engine. L'idée était de recompiler le moteur Unreal à partir de son code source afin d'en dériver une version personnalisée, compilée sous forme de bibliothèque dynamique (DLL), pouvant être intégrée dans une interface WPF* en C#.

Pour évaluer la faisabilité et la pertinence de cette approche, j'ai mené une phase de recherche approfondie en m'appuyant sur plusieurs sources : la documentation officielle des outils à utiliser, des articles spécialisés, ainsi que de nombreux fils de discussion issus de forums techniques (tels que Stack Overflow, Unreal Engine Forums, ou encore les issues des dépôts GitHub). Cette démarche m'a permis de croiser différentes expériences et retours d'utilisateurs confrontés à des problématiques similaires.

2.1.1 Présentation d'Unreal Engine

Unreal Engine est un moteur de jeu et de rendu 3D développé par Epic Games*. Il permet de créer des environnements virtuels interactifs en temps réel, utilisés notamment dans les jeux vidéo, la simulation, ou encore la visualisation architecturale.

Le moteur repose sur une architecture centrée autour d'*Acteur** : ce sont des objets placés dans la scène, comme des modèles 3D, des caméras, ou des lumières. Chaque acteur peut avoir des propriétés et des comportements spécifiques, définis par du code ou des scripts.

Unreal Engine offre une interface graphique riche et des outils puissants pour gérer la scène, la physique, les animations, et les interactions utilisateur. Il est également open source, ce qui permet d'accéder à son code et de le modifier pour des besoins spécifiques.

Les développements dans Unreal Engine s'effectuent principalement en C++, un langage performant et flexible, ou via Blueprint*, un système de programmation visuelle facilitant le prototypage et la conception d'interactions sans code.

2.1.2 Étude des données LiDAR et analyse d'Unreal Engine

J'ai commencé par étudier les formats LiDAR `.las` et `.laz`, essentiels pour l'entreprise, afin de comprendre leur structure et leurs métadonnées (hauteur, intensité, classification). Cette compréhension était cruciale pour anticiper leur intégration et les contraintes liées aux volumes de données.

Unreal Engine utilise des structures d'*octree** pour organiser l'espace 3D, ce qui permet d'adapter dynamiquement le niveau de détail selon la position et l'échelle de la caméra. Cette hiérarchie spatiale optimise le rendu, notamment dans les scènes massives comme les jeux ou les visualisations géospatiales. Comprendre ce mécanisme a orienté certaines décisions sur la structuration des points à l'importation et les attentes en performance.

Parallèlement, je me suis documenté sur Unreal Engine 5, son accès via GitHub sous licence Epic Games, et les modalités de recompilation avec Visual Studio. L'objectif était d'évaluer la possibilité de générer une DLL personnalisée à partir du moteur. Bien que techniquement réalisable, cette approche s'est heurtée à des difficultés liées à la complexité et aux dépendances internes du moteur.

2.1.3 Interopérabilité Unreal / .NET et alternatives

L'interfaçage entre Unreal (C++) et une application C#/.NET* a été un autre axe majeur de recherche. J'ai exploré différentes méthodes : appels de bibliothèque dynamique (DLL), communication inter-processus*, wrapper* en m'appuyant sur des projets open-source pour en mesurer la faisabilité et la robustesse.

Enfin, j'ai envisagé d'autres moteurs ou bibliothèques 3D : Unity, rapidement écarté pour ses performances insuffisantes avec de très gros nuages de points ; Open3D, plus adapté à l'analyse scientifique ; et OpenGL, flexible mais demandant un développement graphique approfondi. Cette exploration a confirmé Unreal comme choix le plus pertinent pour le projet.

Cette phase de recherche m'a permis d'obtenir une compréhension approfondie des différents aspects techniques du projet, en particulier la structure des données LiDAR et les possibilités offertes par Unreal Engine en matière de visualisation 3D. Elle a également posé les bases techniques nécessaires pour entamer les expérimentations, en évaluant concrètement les capacités d'Unreal Engine à gérer de très grands ensembles de points et en testant différentes architectures d'intégration.

2.2 Expérimentations techniques

2.2.1 Installation et premiers tests avec Unreal Engine

Avant d'envisager des intégrations complexes ou la création d'une application personnalisée, il était nécessaire de vérifier la capacité native d'Unreal Engine à manipuler des données LiDAR. J'ai donc commencé par installer la dernière version stable d'Unreal Engine 5 via l'Epic Games Launcher, puis créé un projet vierge avec les paramètres par défaut.

Unreal Engine propose depuis la version 4.24 un plugin* officiel nommé `LiDAR Point Cloud Plugin`, développé par Epic Games. Ce plugin, une fois activé dans l'éditeur, permet d'importer et de visualiser des fichiers `.las` et `.laz` directement dans l'environnement de développement.

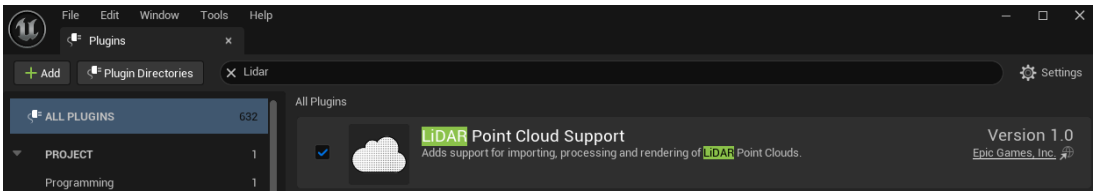


FIGURE 2.1 – Capture d'écran de l'éditeur Unreal Engine montrant l'activation du plugin « LiDAR Point Cloud » dans la section Plugins.

J'ai donc effectué plusieurs tests d'importation en utilisant des jeux de données fournis par l'entreprise, dans les formats `.las` et `.laz`, afin de valider leur compatibilité avec le plugin. Une fois les fichiers importés, ils ont été affichés sous forme de nuages de points dans la vue 3D de l'éditeur Unreal, avec la possibilité de naviguer librement dans la scène et de modifier certains paramètres d'affichage (densité, couleur, classification, etc.).

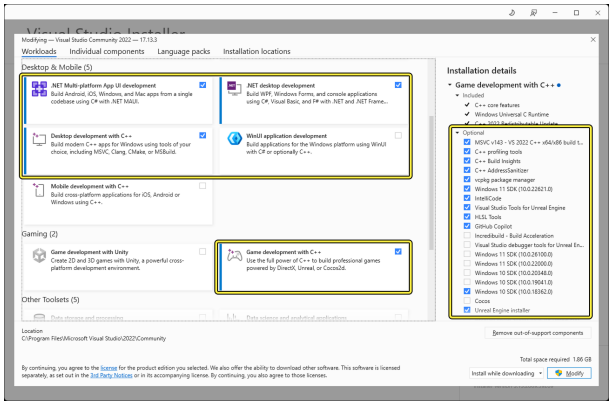
Ces premiers tests ont permis de valider deux points importants :
— les formats de fichiers LiDAR utilisés par l'entreprise sont bien pris en charge par Unreal Engine, sans nécessiter de conversion préalable ;
— le moteur est capable d'afficher des nuages de points de grande taille de manière fluide dans l'éditeur, ce qui confirme son potentiel pour une application de visualisation.

Cette première étape a permis de valider les capacités du moteur et d'aborder les expérimentations techniques suivantes dans de bonnes conditions.

2.2.2 Configuration de l'environnement Unreal Engine / Visual Studio

Pour pouvoir travailler en C++ avec Unreal Engine, il a été nécessaire de mettre en place un environnement de développement adapté avec Visual Studio.

Depuis les versions récentes, Visual Studio prend en charge Unreal Engine de manière native. L'intégration se fait automatiquement si l'on sélectionne les bons composants lors de l'installation, ce qui simplifie la configuration par rapport aux anciennes pratiques où des plugins externes étaient requis.



Capture d'écran de l'installateur Visual Studio avec les options Unreal sélectionnées.

Cette configuration est décrite en détail dans la documentation officielle d'Epic Games ainsi que celle de Microsoft.

Une autre étape importante a concerné l'ajout du plugin LiDAR dans le projet. Pour cela, j'ai dû modifier le fichier `.Build.cs` afin d'y inclure le bon module*. Cette tâche a nécessité un certain temps, car le nom attendu n'était pas directement intuitif : au lieu de `LidarPointCloud`, le module à spécifier était `LidarPointCloudRuntime`. J'ai trouvé cette information en inspectant le fichier `LidarPointCloud.uplugin` situé dans `UE5/Plugins/Enterprise/`, et grâce à des discussions similaires sur les forums Unreal Engine.

Cette phase m'a permis de mieux comprendre le système de modules d'Unreal Engine et les interactions entre le moteur, Visual Studio et les plugins.

2.2.3 Import à l'exécution dans l'éditeur Unreal

Afin de tester la capacité du moteur Unreal à charger dynamiquement des fichiers LiDAR au moment de l'exécution, j'ai développé un script C++ intégré à un acteur personnalisé. Ce script a pour but de détecter automatiquement les fichiers `.las` et `.laz` dans un dossier prédéfini, puis de les charger dans l'environnement 3D à l'aide du plugin `LidarPointCloud`.

L'objectif principal était de valider que le chargement des nuages pouvait se faire directement pendant l'exécution, sans nécessiter d'action depuis l'éditeur, ce qui serait indispensable pour une application autonome.

L'extrait suivant illustre le cœur de ce processus, implémenté dans la méthode `BeginPlay()` de l'acteur :

```
FString Folder = FPaths::Combine(FPaths::ProjectDir(), TEXT("PointCloudsRuntime"));
LoadAllPointCloudsFromFolder(Folder);

// Chargement de chaque fichier .las/.laz detecte dans le dossier
ULidarPointCloud* PointCloud = ULidarPointCloud::CreateFromFile(FullPath, true);
ALidarPointCloudActor* Actor = GetWorld()->SpawnActor<ALidarPointCloudActor>();
Actor->SetPointCloud(PointCloud);
```

Listing 2.1 – Chargement automatique des fichiers LiDAR au démarrage du jeu

Le script parcourt le répertoire `PointCloudsRuntime`, identifie les fichiers LiDAR, et les instancie à l'écran via le système d'acteurs. Cette expérimentation m'a permis de confirmer que le plugin Unreal dédié aux nuages de points peut être invoqué dynamiquement, en dehors d'un import manuel via l'interface graphique de l'éditeur.

2.2.4 Compilation et intégration DLL avec Unreal Engine

Recompilation personnalisée du moteur Unreal en DLL

Dans cette phase, j'ai récupéré le code source d'Unreal Engine 5.5.4 sur GitHub afin d'étudier la possibilité de recompiler le moteur sous forme de bibliothèque dynamique (DLL). L'objectif initial était de dériver une version personnalisée du moteur, allégée et optimisée pour la visualisation de nuages de points, et pouvant être intégrée dans une application tierce, notamment en C# via une interface WPF.

Après une analyse approfondie de la documentation et du code, il est rapidement apparu que cette approche était particulièrement complexe et demandait une charge de travail très importante. Les exemples officiels mentionnant cette possibilité concernaient uniquement la version 4.27 d'Unreal Engine, ce qui posait problème pour conserver les fonctionnalités récentes, notamment le plugin LiDAR indispensable au projet.

De plus, modifier et compiler Unreal Engine en supprimant des modules pour ne garder qu'un viewer 3D risquait d'affecter gravement la stabilité du moteur, compte tenu de l'interdépendance importante entre ses composants. Enfin, des problèmes de compilation sont survenus lors des premières tentatives, ce qui a confirmé la lourdeur et les risques liés à cette démarche.

Au vu de ces éléments, et pour ne pas compromettre l'avancement du projet, j'ai décidé de ne pas poursuivre cette approche, et de chercher des alternatives plus réalistes.

Étude de faisabilité d'une application Unreal sous forme de DLL

En parallèle, j'ai exploré l'option consistant à créer non pas une version modifiée du moteur, mais une application Unreal compilée sous forme de DLL, afin d'être invoquée depuis une autre application hôte (notamment en C#).

Cette approche, quoique plus simple en théorie, nécessite également de modifier certains paramètres de compilation du projet Unreal. Le moteur Unreal est conçu principalement pour fonctionner en tant qu'application exécutable autonome, et sa structure interne fait que faire tourner le moteur en mode DLL, avec un contrôle externe, est complexe.

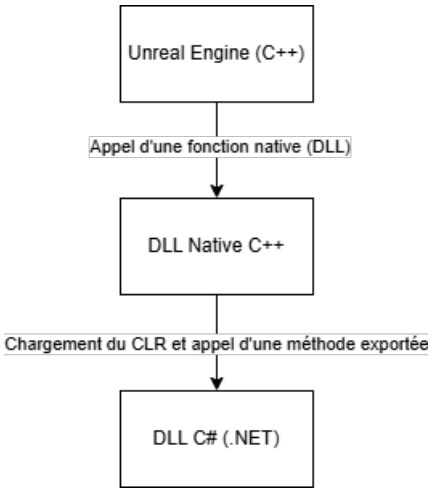
Les recherches documentaires et les expérimentations ont montré que cette solution impliquait également des complications techniques importantes, notamment sur la gestion de la boucle principale d'exécution et l'intégration du moteur dans un environnement tiers.

Au final, cette solution s'est révélée peu adaptée aux objectifs du projet, notamment en raison des contraintes liées au fonctionnement interne d'Unreal Engine, qui est conçu pour être maître de la boucle d'exécution.

Ces études m'ont donc conduit à la conclusion qu'il serait plus efficace de développer directement une application Unreal autonome, intégrant toutes les fonctionnalités nécessaires à la visualisation et à l'interaction avec les nuages de points LiDAR, plutôt que de chercher à faire du moteur une bibliothèque intégrée à un autre environnement.

2.2.5 Expérimentations sur l'interopérabilité Unreal Engine / C# via DLL

Après avoir conclu qu'une application Unreal autonome serait plus adaptée qu'une version du moteur compilée en DLL, j'ai discuté de cette orientation avec mon maître de stage. Il m'a alors demandé de tester la faisabilité d'une communication entre Unreal Engine et une bibliothèque C#, afin d'évaluer la viabilité d'un environnement de développement hybride.



Interopérabilité Unreal/C#
L'objectif était de permettre à Unreal Engine d'interagir avec une DLL C# contenant de la logique métier, par exemple déclenchée par des événements dans l'interface utilisateur. Plusieurs approches techniques ont été explorées pour permettre ce dialogue entre les deux environnements.

FIGURE 2.2 – Principe général de l'interopérabilité entre Unreal Engine et une bibliothèque C#

Méthodes envisagées Après une phase de recherche, plusieurs techniques d'interopérabilité ont été identifiées :

- **.NET Runtime Hosting** : le code natif initialise manuellement le CLR et appelle des méthodes C# via des pointeurs ou des interfaces.
- **Unmanaged Exports (DllExport)** : des méthodes C# sont exportées comme fonctions natives, accessibles via `GetProcAddress`.
- **[UnmanagedCallersOnly]** : introduit avec .NET 5, permet de déclarer une méthode statique comme directement appelable depuis du code natif.
- **Interop COM** : expose des objets C# sous forme de serveurs COM* accessibles depuis C++.
- **Plugin UnrealCLR*** : solution clé-en-main pour exécuter du C# dans Unreal, mais non compatible avec les versions récentes du moteur.
- **C++/CLI Wrapper** : méthode classique d'interopérabilité dans les projets C++, mais incompatible avec Unreal Engine, qui ne supporte pas le C++/CLI.

Tests réalisés J'ai d'abord essayé une DLL C++ exposant des fonctions `extern "C"` appelant une DLL C# qui ouvrait une fenêtre de dialogue, pour créer un pont simple entre Unreal Engine et C#. Cette méthode a échoué, probablement à cause de la gestion du runtime .NET dans Unreal.

Après avoir consulté une ancienne discussion sur le forum d'Unreal Engine, j'ai testé un plugin C# exposant des fonctions appelables en C++. Pour isoler d'éventuels problèmes liés au moteur, j'ai d'abord vérifié son fonctionnement dans une application C++ autonome.

Le premier plugin testé, **RGiesecke's Unmanaged Exports** [1], est une solution populaire reconnue pour sa simplicité. J'ai choisi de commencer par celui-ci en raison de sa large adoption dans la communauté. Dans une application C++ simple, la DLL fonctionnait correctement. En revanche, dans Unreal Engine, son chargement provoquait des plantages, car le Runtime .NET n'était pas initialisé comme attendu.

1. https://www.nuget.org/packages/UnmanagedExports
2. https://github.com/3F/DllExport

Face à ces limites, je me suis tourné vers une alternative plus récente : **DllExport par 3F**. Malgré une configuration plus complexe (documentation partielle, dépendance à des options spécifiques de compilation), cette solution s'est révélée stable et fonctionnelle, aussi bien dans une application C++ que dans Unreal Engine. Elle permet l'appel direct de fonctions C# sans wrapper ni manipulation particulière.

Sous Unreal, j'ai utilisé un acteur C++ personnalisé associé à des Blueprints pour créer une interface simple avec des boutons appelant les méthodes exportées.

Parmi toutes les méthodes testées, seule celle utilisant **DllExport (3F)** a fonctionné correctement avec Unreal Engine. Elle permet d'appeler des fonctions C# comme si elles étaient natives, sans utiliser de wrapper C++/CLI, et le Runtime .NET se charge automatiquement.

Cela confirme qu'il est possible de faire communiquer Unreal Engine avec du code C#, et d'imaginer une architecture où Unreal gère l'affichage et le C# la logique métier.

2.2.6 Conclusion des expérimentations techniques

Ces différentes expérimentations ont permis d'explorer plusieurs pistes techniques autour de la visualisation et de l'exploitation de données LiDAR dans Unreal Engine, en vue d'une application plus complète.

La première série de tests a confirmé la compatibilité du moteur avec les formats `.las` et `.laz` via le plugin officiel, ainsi que sa capacité à gérer de larges nuages de points de manière fluide. L'environnement de développement Unreal/Visual Studio a pu être mis en place sans difficulté majeure, malgré quelques ajustements liés à l'intégration des modules C++ externes.

Les tentatives de recompilation du moteur en tant que DLL, ou d'extraction d'une application Unreal minimaliste sous forme de bibliothèque, se sont révélées peu concluantes. Cela s'explique à la fois par la complexité de la base de code Unreal, les problèmes de compatibilité avec les versions récentes, et le manque de documentation à jour sur ces aspects. Ces pistes ont donc été écartées.

Enfin, les tests d'interopérabilité entre Unreal Engine (C++) et des bibliothèques externes en C# ont montré qu'il était possible d'établir une communication efficace via des DLL exportées, à condition d'utiliser les bons outils (comme le plugin `DllExport` de 3F) et de respecter certaines contraintes de compilation (architecture x64, mode `Release`, etc.).

Ces résultats ont permis de valider la faisabilité d'un prototype Unreal Engine intégré dans un environnement applicatif plus large, tout en identifiant clairement les limites techniques à anticiper pour un développement plus abouti.

2.3 Développement de l'application

À partir des résultats obtenus lors des expérimentations, le développement de l'application a pu débuter. L'objectif principal était de reproduire la navigation de type TerraScan, à laquelle se sont ajoutées plusieurs fonctionnalités complémentaires demandées au fur et à mesure.

2.3.1 Navigation utilisateur

Pour permettre une navigation fluide dans la scène 3D, j'ai développé un *Pawn Actor** personnalisé dans Unreal Engine, en codant manuellement les différents mouvements.

Dans le cadre du projet, un employé m'a présenté TerraScan, logiciel utilisé par l'entreprise pour visualiser et analyser les données LiDAR. Après une brève prise en main, j'ai pu identifier les éléments clés de la navigation, qui ont servi de référence pour le développement de l'application.

J'ai commencé par implémenter les fonctions de zoom avant et zoom arrière, puis le déplacement horizontal et vertical (panning) contrôlé par le clic molette. Enfin, j'ai intégré l'orbitage autour d'un point précis du nuage de points, qui s'est avéré être la partie la plus complexe à réaliser. Cette fonctionnalité a nécessité de nombreux ajustements pour assurer une rotation intuitive et précise, mais a pu être menée à bien avec succès.

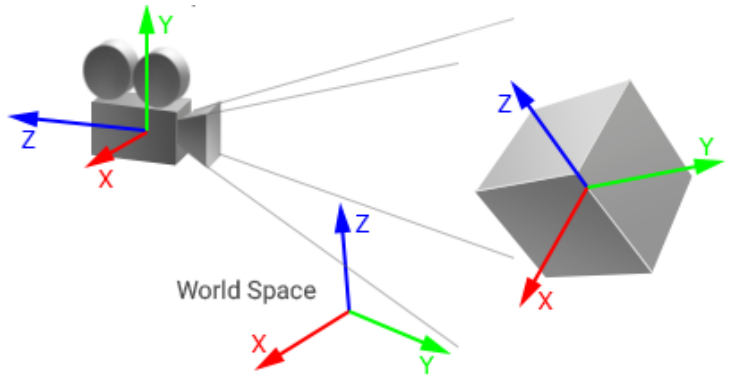


FIGURE 2.3 – Schéma de navigation 3D : la caméra se déplace selon XYZ.

$$\vec{P} = \begin{bmatrix} x & y & z \end{bmatrix} \quad \text{ou} \quad \vec{P} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Vecteur représentant la position d'un point dans l'espace 3D, tel que la position de la caméra ou du centre d'orbite.

L'image ci-dessus illustre le fonctionnement de la caméra dans le repère global, et les axes sur lesquels s'appuient les mouvements programmés (zoom, pan, orbitage). Ces déplacements reposent sur des transformations géométriques calculées à partir de la position de la caméra, de sa cible, et des vecteurs locaux du repère caméra.

2.3.2 Fonctionnalité d'importation de fichiers

Pour permettre à l'utilisateur d'importer des nuages de points dans l'application, j'ai implémenté une fonctionnalité d'ouverture de fichiers. Initialement, j'ai utilisé la méthode `IDesktopPlatform::OpenFileDialog` fournie par Unreal Engine, qui propose une interface native de sélection de fichiers. Cette solution, simple à intégrer et bien documentée, était adaptée pour les tests réalisés directement dans l'éditeur.

Lors de la préparation d'une version packagée, j'ai constaté que cette méthode ne fonctionnait pas hors de l'éditeur, la boîte de dialogue native n'étant pas disponible dans les builds standalone*. J'ai alors étudié plusieurs alternatives, notamment des plugins Unreal dédiés ou la création d'une interface personnalisée. Ces options présentaient soit une complexité importante, soit une surcharge inutile pour une fonctionnalité relativement simple.

J'ai finalement opté pour la bibliothèque externe multiplateforme *Native File Dialog (NFD)*[2], qui fournit une interface légère, fonctionnelle dans toutes les configurations (éditeur et builds standalone), sans nécessiter de développement graphique supplémentaire. Son intégration a permis d'assurer une expérience utilisateur cohérente et fiable.

Par ailleurs, j'ai ajouté la possibilité d'importer plusieurs fichiers simultanément, en restreignant la sélection aux formats `.las` et `.laz`, ceux utilisés par l'entreprise pour les données LiDAR. Cette gestion spécifique des extensions garantit que seuls les fichiers compatibles sont sélectionnés, évitant ainsi les erreurs en amont du traitement.

L'importation des fichiers s'effectue de manière asynchrone*, afin d'éviter de bloquer le fil principal de l'application pendant le chargement potentiellement volumineux des nuages de points.

2.3.3 Gestion du géoréférencement

Un enjeu central de l'application concernait le positionnement spatial correct des nuages de points importés. Les fichiers LiDAR contiennent généralement des coordonnées géographiques absolues, exprimées dans des systèmes de référence comme Lambert 93 (annexe 3.3). À l'inverse, Unreal Engine place tous les acteurs à l'origine de la scène, en $(0, 0, 0)$ dans son espace local.

Durant cette phase, j'ai utilisé des fichiers au format `.xyz`, plus simples à lire et manipuler que les formats `.las` ou `.laz` natifs des données LiDAR, ces derniers sont des formats binaires complexes, difficiles à ouvrir et à analyser directement dans un éditeur de texte. Pour cela, j'ai converti les fichiers originaux à l'aide de CloudCompare afin d'obtenir un format texte exploitable. Cette approche m'a permis de visualiser directement les coordonnées dans un éditeur de texte, facilitant ainsi les tests et la validation des comportements du système d'importation. Les fichiers `.xyz` contiennent des coordonnées géographiques ou projetées dans des plages numériques raisonnables, comme par exemple un point dont les valeurs sont :

X	Y	Z
727 542.50000000	6 375 853.50000000	751.45300293

Toutefois, lors de l'importation directe dans Unreal Engine, ces coordonnées peuvent se transformer en valeurs beaucoup plus grandes dans le système local, comme illustré ci-dessous :

X	Y	Z
72 754 250.0	-637 585 350.0	75 145.300293

Cette différence d'échelle peut poser problème, car des coordonnées très éloignées de l'origine `(0,0,0)` peuvent entraîner des effets indésirables dans Unreal Engine, comme des imprécisions visuelles, des erreurs de collisions ou des comportements physiques inattendus, en raison des limites de précision en virgule flottante.

2. https://github.com/mlabbe/nativefiledialog

Le plugin de visualisation de nuages de points utilisé permet néanmoins de préserver les coordonnées d'origine, mais une réflexion a été menée pour choisir une méthode adaptée aux différentes contraintes.

2.3.4 Améliorations visuelles et contrôle utilisateur

Gestion de la taille des points Initialement, les points étaient affichés avec une taille fixe à l'écran (*fixed screen sized*), garantissant une visibilité constante quel que soit l'éloignement de la caméra. Après une démonstration, il a été constaté que cette méthode pouvait rendre les points trop gros ou trop petits dans certains cas. Pour pallier ce problème, j'ai ajouté des boutons `+` et `-` permettant à l'utilisateur de modifier manuellement la taille des points selon ses besoins.

Zoom adaptatif J'ai également implémenté un système de zoom adaptatif où la vitesse de zoom varie en fonction de la distance aux points situés devant la caméra. Pour cela, une collision est détectée avec le nuage de points, et la vitesse de zoom est ajustée entre une vitesse minimale et maximale prédéfinie. Si aucun point n'est détecté devant la caméra, une vitesse par défaut est utilisée. Ce système rend la navigation plus fluide et intuitive, surtout lors de l'exploration de zones denses ou très éloignées.

Implémentation du marqueur de centre d'orbitage Afin d'aider l'utilisateur à repérer le centre de rotation pendant l'orbitage, un marqueur visible en permanence a été créé. Ce dernier devait rester orienté face à la caméra, être visible même à travers les points du nuage, et s'adapter en taille en fonction de la distance.

Plusieurs techniques ont été testées, l'utilisation d'un *billboard*, qui posait des problèmes de mise à l'échelle. Puis un *widget* Unreal qui n'etait pas visible en permanence, et finalement un `UStaticMesh` avec un matériau spécifique pour assurer visibilité et adaptation.

La texture du marqueur (une croix) a été créée sous GIMP et ajoutée aux ressources du projet afin qu'elle soit disponible en mode *release*. Au démarrage de l'orbitage, le marqueur est automatiquement positionné au point de rotation choisi.

Coloration par classification Dans le contexte LiDAR, la classification des points est essentielle : chaque point est associé à une classe suivant une norme (par exemple ASPRS), identifiant le type d'objet ou de surface (sol, végétation, bâtiments, etc.).

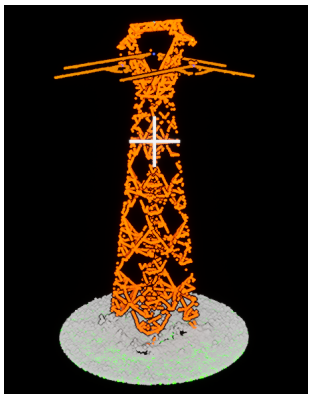


FIGURE 2.4 – Exemple de nuage de points coloré par classification, avec le marqueur d'orbitage visible.

Cette classification permet d'afficher le nuage de points en couleurs distinctes pour chaque classe, facilitant l'interprétation visuelle. J'ai intégré cette fonctionnalité, et pour illustration, des captures d'écran montrent la coloration des points selon leur classe, ainsi que le marqueur en action lors de l'orbitage.

Des informations complémentaires sur les fichiers `.las` et `.laz`, notamment un extrait de leur en-tête généré avec `lasinfo`[3], sont disponibles en annexe 3.3. Ces données permettent de visualiser la structure du fichier ainsi que la répartition des classes utilisées pour la classification des points.

2.3.5 Documentation technique et refactorisation

Cette phase a constitué une étape essentielle du projet. En effet, bien documenter et structurer le code est fondamental dans une optique de pérennité, notamment dans un contexte professionnel où la personne amenée à reprendre le développement pourrait ne pas avoir d'expérience préalable avec Unreal Engine.

Refactorisation du code. Avant de rédiger la documentation, j'ai procédé à une refonte partielle de la structure du code afin de le rendre plus clair, modulaire et évolutif. Par exemple, la logique d'importation de fichiers, initialement intégrée à l'acteur principal chargé de l'affichage des nuages de points, a été extraite dans une classe dédiée (un *sub-process Unreal*) pour mieux séparer les responsabilités. De même, la gestion des couleurs a été isolée dans une logique distincte, ce qui permet à chaque composant de faire appel aux fonctionnalités de l'autre lorsque nécessaire. Cette approche rend le système plus souple, facilite la maintenance, et permet de réutiliser ces briques dans d'autres contextes au sein du projet.

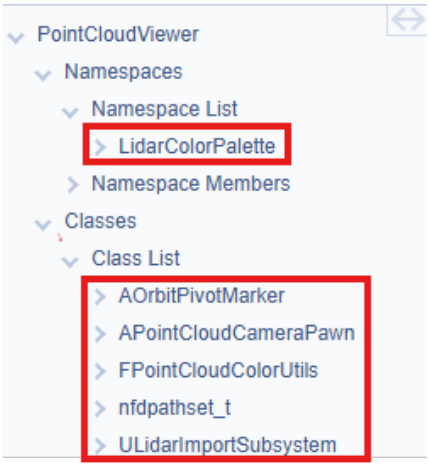


FIGURE 2.5 – Exemple d'arborescence des classes du projet

Cette représentation montre que les logiques du projet ont été bien séparées, ce qui rend l'organisation du code plus claire et structurée.

Gestion flexible de la classification. Les fichiers LiDAR comportent souvent une classification des points (sol, végétation, bâtiments, etc.), mais celle-ci varie selon les jeux de données. Les classes et couleurs associées ne sont pas normalisées entre fournisseurs ou clients.

3. `lasinfo` est un outil de la suite `LAStools` permettant d'inspecter le contenu des fichiers `.las` ou `.laz`.

J'ai donc mis en place une gestion personnalisable de la classification :
— un jeu de couleurs par défaut a été défini pour les classes les plus courantes (basé sur la spécification ASPRS [4]) ;
— il est possible de fournir un fichier `.csv` externe listant les classes et les couleurs à utiliser ;
— les couleurs sont définies dans un espace de noms dédié (namespace*) afin que l'utilisateur puisse simplement renseigner un nom de couleur dans le fichier ;
— une gestion de secours est prévue : si une couleur n'existe pas, une teinte grise est utilisée par défaut.

Comme la bibliothèque standard du C++ ne propose pas une palette de couleurs très fournie, j'ai dû enrichir manuellement cette liste de couleurs disponibles.

L'exemple simplifié de fichier `.csv` est disponible en annexe 3.3.

Documentation et outils. L'ensemble des fonctions et classes principales a été documenté directement dans le code à l'aide de commentaires compatibles avec `Doxygen`*. Cela a permis de générer automatiquement une documentation technique, disponible en HTML (navigable) et en PDF. La version PDF est fournie dans un document séparé, mais la version web est à privilégier pour la consultation.

`Doxygen` repose sur un paquet `LaTeX*` obsolète pour la génération du PDF, ce qui a nécessité une correction manuelle à l'aide d'un Patch*.

En complément, un fichier `README` présente le projet, ses fonctionnalités, les éléments implémentés et les limitations connues. Un fichier PDF séparé fournit des instructions détaillées pour la mise en place de l'environnement de développement (installation d'Unreal Engine, des dépendances, et des plugins nécessaires).

2.3.6 Difficultés rencontrées et solutions apportées

Les deux principaux défis rencontrés durant ce projet ont été la gestion de l'orbitage et le géoréférencement. Ces sujets ont pris le plus de temps, tandis que les autres difficultés, plus mineures, ont été traitées rapidement dans leurs sections respectives.

Avant d'exposer en détail les solutions mises en place, il est utile de préciser les outils de développement utilisés, en particulier la distinction entre Blueprint et C++, afin de mieux comprendre les choix techniques réalisés au cours du projet.

Unreal Engine : Blueprint et C++. Unreal Engine permet de développer des applications en utilisant :
— **Blueprint**, un langage visuel sous forme de graphes de nœuds connectés. Il est principalement destiné à des utilisateurs non développeurs ou à des cas de prototypage rapide, mais il reste limité en termes de performance, de lisibilité sur des projets complexes et d'intégration fine avec certains outils.
— **C++**, un langage de programmation plus classique, qui offre des performances supérieures, un meilleur contrôle sur l'architecture du projet, et une plus grande souplesse pour l'intégration avec des bibliothèques externes ou des plugins.

J'ai choisi de développer en **C++** afin de garantir de meilleures performances, d'assurer une architecture plus robuste et évolutive, et de faciliter l'intégration avec le plugin de gestion des nuages de points.

4. https://www.asprs.org/wp-content/uploads/2019/07/LAS_1_4_r15.pdf

Gestion de l'orbitage

Le premier problème majeur rencontré concernait la définition d'un point d'orbitage pour permettre une navigation fluide et intuitive autour des nuages de points. Le plugin utilisé ne proposait pas de fonction de *point picking* permettant de sélectionner un point cible pour l'orbiteur. J'ai donc dû concevoir une solution sur mesure en m'appuyant sur un système de raycast.

Principe du raycast Le **raycast** est une technique classique en 3D qui consiste à projeter un rayon invisible depuis un point d'origine (par exemple, la caméra ou le curseur) dans une direction donnée. Ce rayon détecte les collisions avec des objets de la scène et renvoie des informations utiles telles que le point d'impact ou l'objet touché (voir schéma 2.6). Cette méthode est largement utilisée pour les sélections d'objets, les clics ou la détection d'interactions.

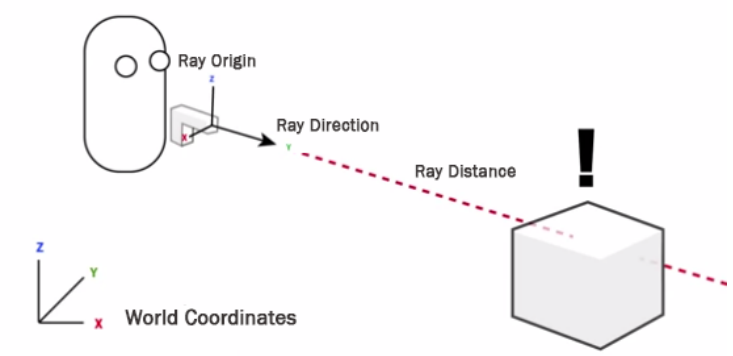


FIGURE 2.6 – Principe du raycast : un rayon est projeté depuis la caméra ou le curseur et retourne un point de collision s'il rencontre un objet

Premiers essais et difficultés J'ai d'abord tenté d'utiliser directement les fonctions de raycast fournies par le plugin. Le choix d'implémenter la logique en C++ dès le début répondait au besoin de performances et de contrôle précis sur les interactions, d'autant plus que l'ensemble du projet était déjà principalement développé dans ce langage. Je n'ai donc pas exploré immédiatement les pistes proposées par les Blueprints.

Cependant, ces fonctions ne renvoyaient aucun point d'impact exploitable sur le nuage de points : le rayon semblait traverser la scène sans détecter d'intersection. Pour expliquer ce comportement, j'ai formulé plusieurs hypothèses pouvant être à l'origine du problème :

- Le rayon ne partait peut-être pas du bon point dans l'espace (caméra, curseur, etc.).
- Sa direction ou sa longueur pouvait être incorrecte, ne traversant pas efficacement la scène.
- Il était possible que les points du nuage ne soient pas pris en compte dans les tests de collision.
- Les coordonnées utilisées dans le traitement n'étaient peut-être pas dans le bon espace (local vs monde).
- Enfin, une mauvaise configuration ou un usage inadapté des fonctions de raycast du plugin restait une possibilité.

Débogage et expérimentations Pour analyser le problème, j'ai mis en place plusieurs outils de débogage :

Affichage visuel des rayons projetés dans la scène ;

Placement de sphères au point d'impact supposé ;

Ajout de logs détaillant l'origine, la direction et les résultats des rayons.

Des captures d'écran illustrant ces tests (rayons projetés, sphères de collision, résultats partiels) sont présentées en annexe 3.3.

J'ai ensuite essayé d'améliorer la détection en plaçant une sphère de collision au point d'impact estimé, mais sans obtenir de résultat fiable. La documentation partielle du plugin et la présence de plusieurs fonctions aux noms très proches ont compliqué les choses.

Approche expérimentale en Blueprint Lors de mes recherches, je n'ai trouvé que très peu de ressources pertinentes sur ce sujet précis : uniquement trois discussions de forum et n'utilisant que des Blueprint, dont aucune ne proposait une solution directe pour récupérer un point précis d'un nuage via raycast avec le plugin.

L'une de ces discussions évoquait l'utilisation d'un raycast classique d'Unreal (et non celui du plugin) dans un contexte d'*Acteur First Person*. Cette discussion incluait un lien vers un tutoriel sur les raycasts avec ce type d'acteur. Pour avancer, j'ai donc créé un projet de test basé sur la *Template First Person* d'Unreal Engine.

> Pour mieux comprendre le test réalisé, il est utile de préciser qu'un acteur *First Person* correspond à un personnage avec caméra intégrée au niveau des yeux, contrôlé comme dans un jeu vidéo classique à la première personne.
>
> En revanche, mon projet utilise un *Pawn Actor*, une classe plus générique offrant un contrôle libre de la caméra, mieux adaptée à une visualisation 3D interactive. Cette différence implique que les méthodes basées sur le *First Person* doivent être adaptées pour fonctionner dans mon contexte.

J'y ai mis en place un raycast standard : il fonctionnait parfaitement sur les objets natifs (cubes, sphères) de la scène. J'ai ensuite tenté d'appliquer ce raycast sur un nuage de points importé : les résultats étaient positifs, mais la précision laissait à désirer. Certains points étaient bien détectés, d'autres généraient un léger décalage ou n'étaient pas interceptés du tout.

Pour analyser ces résultats, j'ai ajouté un mécanisme visuel : à chaque point d'impact détecté, une petite sphère était créée autour de la position du hit, et tous les points situés dans son rayon étaient colorés pour marquer la détection. Ces visualisations (exemples en annexe 3.3) ont confirmé que les collisions fonctionnaient en grande partie, mais avec un manque de fiabilité sur certains points.

J'ai ensuite modifié le blueprint pour que le raycast parte non plus de la caméra *First Person*, mais directement du curseur de la souris. Cette version s'est montrée fonctionnelle, et j'ai pu l'intégrer dans mon projet principal pour confirmer qu'elle donnait les mêmes résultats. Un extrait du Blueprint correspondant est présenté figure 2.7, illustrant la logique mise en œuvre pour lancer le raycast depuis la position du curseur et générer les visualisations associées.

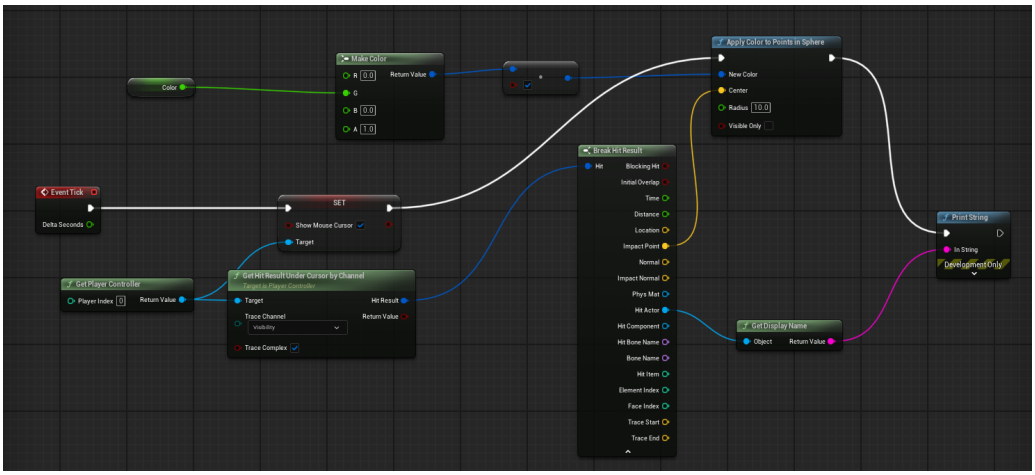


FIGURE 2.7 – Exemple de Blueprint mis en place pour lancer un raycast depuis la position du curseur et visualiser les collisions

Enfin, en poursuivant l'exploration des discussions restantes, j'ai identifié une méthode qui semblait exploiter les fonctions du plugin dans les Blueprints. J'ai adapté mon blueprint en conséquence, et cette approche a apporté un gain notable de précision.

Ce projet m'a permis de valider la bonne exécution du raycast sur des objets standards (cubes, sphères) et sur un nuage de points importé. J'ai confirmé que partir de la position du curseur pour générer le rayon offrait de meilleurs résultats.

Finalisation en C++ Après avoir validé la logique dans les Blueprints, j'ai transcrit et adapté cette solution en C++. Contrairement aux premières tentatives, cette phase s'est appuyée uniquement sur les fonctions de raycast fournies par le plugin, sans recourir aux méthodes de raycast natives d'Unreal Engine.

Cette base solide m'a permis de finaliser un système robuste, capable de récupérer un point précis sous le curseur et de l'utiliser comme centre d'orbite. Cette logique a en outre servi à la mise en place d'un zoom adaptatif, le raycast mesurant la distance au point détecté devant la caméra afin d'ajuster dynamiquement le facteur de zoom.

Géoréférencement

L'objectif principal était que le nuage de points soit positionné dans la scène selon ses coordonnées de scan originales. Plusieurs méthodes étaient envisageables.

Alignement via la méthode *Align Cloud* La méthode *Align Cloud* du plugin permet d'aligner un nuage de points par rapport à un tableau de référence, facilitant ainsi la visualisation relative des nuages. Cependant, cette méthode ne place pas le nuage à ses coordonnées géographiques originales : l'acteur contenant le nuage reste à la position `(0,0,0)` dans le monde. Cela complique l'obtention de la position réelle du nuage, car il faut distinguer les coordonnées *locales* (par rapport à l'acteur) des coordonnées *mondiales* (dans la scène). Cette différence est cruciale, notamment pour le positionnement de l'orbitage, qui nécessite une transformation des vecteurs de collision afin de fonctionner correctement avec ce décalage.

Méthode de réinitialisation aux coordonnées originales J'ai ensuite essayé une autre méthode proposée par le plugin, qui réinitialise directement la position du nuage à ses coordonnées originales dans le système mondial. Cependant, lors de mes essais, le nuage restait systématiquement à l'origine `(0,0,0)`.

Après plusieurs recherches sans solution claire, j'ai envisagé le plugin de géoréférencement officiel d'Unreal Engine, mais il me semblait trop complexe et inadapté aux besoins spécifiques de mon projet.

Investigation et diagnostic Pour comprendre ce dysfonctionnement, j'ai mis en place des outils de débogage : logs, impressions dans Blueprint, etc. Sur les conseils d'un membre du pôle innovation, j'ai vérifié le format `.xyz` des fichiers, afin de m'assurer que les coordonnées géographiques étaient bien présentes dans les données sources.

Dans l'éditeur Unreal, il est possible de réinitialiser manuellement la position du nuage à ses coordonnées via un menu, ce qui fonctionne bien. Mais dans mon cas, je souhaitais automatiser ce positionnement au runtime, sans intervention manuelle.

En comparant les coordonnées importées par Unreal (qui utilise l'unité centimètre comme unité de base), j'ai constaté que les données originales étaient bien importées, mais la variable `OriginalCoordinate` du nuage restait vide après import au runtime. J'ai testé avec plusieurs outils (CloudCompare, LasUtils) pour vérifier l'intégrité des fichiers, sans anomalie détectée.

Hypothèse de la race condition J'ai suspecté que l'importation asynchrone que j'avais mise en place provoquait une condition de concurrence : la variable des coordonnées originales était accédée trop tôt, avant que les données ne soient totalement chargées.

Pour tester cela, j'ai temporairement modifié la fonction d'importation pour la rendre synchrone, ce qui a effectivement résolu le problème. Toutefois, cette solution engendrait un ralentissement significatif, car les imports bloquaient l'interface et ralentissaient la gestion de plusieurs nuages simultanés.

Solution par gestion des callbacks En observant la fonction d'importation disponible en Blueprint, j'ai remarqué qu'il était possible de spécifier des actions à exécuter à différents moments du processus : à chaque pourcentage de progression, et surtout à la fin de l'import. Cela m'a conduit à me demander si cette même logique de callbacks était accessible en C++.

Après vérification dans la documentation du plugin, j'ai constaté que la méthode d'importation existe effectivement sous deux formes : une version simple sans retour, et une version acceptant deux fonctions de rappel (callbacks), l'une pour suivre la progression et l'autre pour agir une fois l'import terminé.

J'ai donc réécrit ma logique d'importation en utilisant cette version asynchrone avec callbacks. Plutôt que de tenter de positionner le nuage immédiatement après l'appel d'import (ce qui entraînait une condition de concurrence), j'attends désormais explicitement le signal de fin d'importation avant d'agir. Ce n'est qu'à ce moment-là que je place l'acteur du nuage à ses coordonnées géographiques originales, extraites depuis ses métadonnées internes.

Ce changement garantit que l'acteur est bien positionné dans le monde, à ses vraies coordonnées, et non plus au point `(0,0,0)`. En conséquence, les calculs de position pour le système d'orbite ne nécessitent plus de transformation de vecteurs entre repères locaux et mondiaux, ce qui simplifie grandement la logique tout en assurant une meilleure précision.

Chapitre 3

Bilan

3.1 Résultats obtenus

3.1.1 Travail réalisé

Durant le stage, plusieurs fonctionnalités essentielles ont été implémentées avec succès, constituant un socle interactif robuste pour la visualisation de nuages de points dans Unreal Engine :

- **Importation dynamique** de fichiers `.las`, `.laz` et `.xyz` pendant l'exécution, avec traitement asynchrone et géoréférencement automatique à partir des métadonnées.
- **Navigation 3D fluide** incluant :
  - Zoom avant/arrière à la molette.
  - Zoom adaptatif selon la distance au point sélectionné.
  - Orbitation autour d'un point sélectionné via raycast.
  - Panoramique (panning) avec le clic molette maintenu.
  - Réinitialisation de la vue par un double clic molette.
- **Personnalisation de l'affichage** :
  - Modification de la taille des points avec les touches `+` et `-`.
  - Taille dynamique des points en fonction de leur distance à la caméra.
- **Affichage par classification** :
  - Coloration des points en fonction de leur classe.
  - Chargement d'une correspondance classe-couleur via un fichier `.csv`.

L'ensemble a été intégré dans un prototype cohérent, offrant une première base utilisable pour un outil de visualisation ou d'analyse géospatiale personnalisée.

Un second projet a également été réalisé pour tester la communication entre Unreal Engine et une bibliothèque externe en C#, via l'utilisation d'une DLL.

Enfin, une documentation technique a été rédigée en parallèle afin de faciliter la prise en main du projet par de futurs développeurs. Elle couvre à la fois l'architecture générale, l'utilisation des fonctionnalités principales et les points spécifiques à la mise en œuvre dans Unreal Engine.

3.1.2 Travail non finalisé

Par manque de temps ou par choix de priorisation, certaines fonctionnalités envisagées n'ont pas été finalisées ou seulement esquissées :

- **Amélioration de la logique d'orbite** : la caméra orbite encore en gardant le point en focus, ce qui diffère du comportement attendu (inspiré de Terrascan).
- **Affichage durant l'importation** : aucune indication visuelle n'est encore fournie à l'utilisateur pendant le chargement d'un nuage.
- **Boîte d'information sur les points** : l'affichage d'attributs détaillés (position, classification, intensité...) sous forme de panneau contextuel reste à développer.
- **Personnalisation de l'environnement** : le changement de la couleur de fond n'est pas encore intégré.
- **Multi-vues** : la possibilité d'afficher plusieurs points de vue simultanés (split screen ou fenêtres secondaires) est une piste à explorer.
- **Vérification fine des interactions souris** : notamment sur le comportement du maintien de la molette, qui nécessite d'être ajusté pour correspondre à l'ergonomie de logiciels existants comme Terrascan.

Ce projet pose ainsi les bases d'un visualiseur 3D interactif adaptable à différents besoins. Pour la suite, plusieurs axes d'amélioration pourraient être envisagés : affiner le comportement de navigation pour se rapprocher d'outils professionnels existants, améliorer les performances de rendu pour de très grands nuages de points, ou encore enrichir l'interface utilisateur.

Par ailleurs, la sortie récente d'Unreal Engine 5.6, intervenue dans les dernières semaines du stage, ouvre de nouvelles perspectives techniques. Cette version apporte des optimisations de performance. Migrer vers cette version pourrait constituer un premier chantier pertinent pour la continuité du projet.

3.2 Rétrospection sur le travail réalisé

Ce projet de visualisation 3D m'a permis de découvrir des problématiques techniques peu abordées en formation, comme la manipulation de données massives ou l'intégration dans un moteur graphique tel qu'Unreal Engine. Malgré la complexité du sujet, cette expérience m'a offert un bon aperçu des enjeux concrets du développement logiciel en contexte métier.

J'ai notamment constaté l'importance :

- d'une bonne structuration du code, indispensable pour assurer sa maintenabilité et faciliter les évolutions futures ;
- d'une compréhension approfondie des besoins utilisateurs, pour concevoir une interface adaptée et ergonomique ;
- d'adopter une approche expérimentale et itérative, en particulier lorsqu'il n'existe pas de solution toute faite ;
- de l'adaptation face aux défis techniques imprévus, tels que la gestion des calculs liés à de très grandes coordonnées ou la navigation fluide dans un espace virtuel.

Travailler sur ce projet de visualisation 3D a été une expérience particulièrement enrichissante, me permettant d'aborder un sujet technique peu traité durant ma formation. J'ai pu développer des compétences spécifiques liées à la manipulation de données massives, à l'intégration de bibliothèques externes, ainsi qu'à la mise en œuvre de fonctionnalités interactives en 3D à l'aide d'un moteur graphique avancé comme *Unreal Engine*.

La nature exploratoire du stage m'a confronté à des problématiques concrètes et parfois imprévues, telles que le géoréférencement ou la gestion fine des interactions utilisateur, enrichissant ainsi ma compréhension globale du développement logiciel.

3.3 Bilan personnel

Ce projet a représenté un défi formateur, aussi bien sur le plan technique que personnel. Il m'a permis de consolider et d'approfondir plusieurs compétences essentielles à un développeur, tout en développant ma capacité à travailler de manière autonome et rigoureuse.

Sur le plan des **savoirs et savoir-faire techniques**, j'ai pu me familiariser en profondeur avec Unreal Engine, en particulier avec son API C++ et son système de Blueprint. J'ai appris à naviguer dans une architecture logicielle complexe, à tirer parti de plugins externes, et à combiner plusieurs paradigmes pour construire une application interactive cohérente. J'ai également développé une meilleure compréhension des systèmes de coordonnées, du géoréférencement, et des problématiques liées à la représentation de données 3D massives.

En parallèle, j'ai acquis des compétences concrètes en débogage, en organisation du code, en utilisation d'outils tiers (Visual Studio, Git, documentation technique), ainsi qu'en gestion d'un projet logiciel sur plusieurs semaines. La partie du stage sur la communication entre Unreal Engine et une bibliothèque C#, m'a permis d'explorer un autre type de problématique technique : l'interfaçage entre deux technologies différentes.

Sur le plan **méthodologique**, ce stage m'a permis de structurer ma démarche face à un problème complexe : poser des hypothèses, expérimenter, tester, corriger, documenter. La difficulté de certaines tâches (comme la détection de points par raycast) m'a appris à adopter une approche systématique et patiente, à ne pas céder à la frustration, et à progresser par itérations.

Enfin, en termes de **savoir-être**, cette expérience m'a aidé à mieux gérer mon autonomie, à organiser mon temps efficacement, et à prendre des décisions techniques de manière justifiée. J'ai également pris conscience de l'importance de la communication avec le tuteur et l'équipe, notamment pour exposer les avancées, signaler les blocages, ou discuter des solutions envisagées.

Ce stage m'a permis de prendre confiance dans mes capacités à mener un projet technique de bout en bout. Il m'a conforté dans mon intérêt pour les technologies 3D et les problématiques liées à l'interaction homme-machine, et m'a donné envie de poursuivre dans cette voie à travers des projets plus ambitieux ou collaboratifs.

Conclusion

Ce stage de dix semaines effectué dans le cadre de ma deuxième année de BUT Informatique à l'IUT de Nantes a constitué une première immersion concrète dans le développement logiciel en environnement professionnel.

L'objectif principal, qui était de concevoir un viewer 3D de nuages de points interactif sous Unreal Engine, a été globalement atteint. Les fonctionnalités essentielles de visualisation, navigation et classification ont été développées, et le prototype final constitue une base solide pour des usages plus poussés dans le domaine de la visualisation géospatiale. D'autres aspects secondaires, moins prioritaires, n'ont pas pu être finalisés dans le temps imparti, mais des pistes claires d'amélioration ont été identifiées.

Ce projet m'a permis de mettre en œuvre les compétences acquises au cours de ma formation : en programmation, algorithmique, conception logicielle. Tout en en développant de nouvelles : gestion d'un moteur 3D complexe comme Unreal Engine, compréhension des systèmes de coordonnées géographiques, intégration de plugins, communication entre langages, etc.

Mais au-delà des compétences techniques, ce stage m'a également apporté sur le plan personnel. J'ai appris à gérer mon autonomie, à structurer mon raisonnement face à des problèmes non triviaux, et à progresser par essais et erreurs. J'ai découvert un environnement de travail stimulant et un domaine applicatif riche, mêlant informatique, 3D et données géographiques.

Cette expérience m'a conforté dans mon envie de poursuivre dans le développement logiciel, en particulier dans des projets à forte composante graphique ou interactive. Si c'était à refaire, je prendrais encore plus de temps pour poser un cadre méthodologique solide dès les premières semaines, afin de maximiser l'efficacité et la clarté dans les phases de développement.

En somme, ce stage a été pour moi une expérience formatrice, enrichissante, et motivante, qui marque une étape importante dans mon parcours d'informaticien.

Glossaire

.NET Plateforme de développement créée par Microsoft, permettant de créer et exécuter des applications sur différents systèmes d'exploitation, notamment utilisée avec le langage C#.

.dll Fichier de bibliothèque dynamique (Dynamic Link Library) utilisé principalement sous Windows pour regrouper du code ou des fonctions pouvant être partagés entre plusieurs programmes.

.las / .laz Formats standards utilisés pour le stockage compressé (.laz) ou non compressé (.las) de données LiDAR.

Acteur Objet élémentaire dans un moteur 3D (comme Unreal Engine), représentant une entité dans la scène, pouvant avoir des propriétés et des comportements spécifiques.

Acteur First Person Acteur dans Unreal Engine représentant le joueur en vue subjective, avec caméra embarquée et contrôles de mouvement typiques des jeux à la première personne.

Asynchrone Mode d'exécution où une opération est lancée sans bloquer le programme principal, permettant ainsi de continuer d'autres tâches pendant que l'opération se termine en arrière-plan.

Billboard Technique d'affichage 2D dans un environnement 3D, où une image ou un objet reste toujours orienté face à la caméra.

Blueprint Système de programmation visuelle d'Unreal Engine permettant de concevoir des logiques sans écrire de code.

Build Standalone Version autonome d'un logiciel ou d'une application, compilée pour fonctionner indépendamment sans nécessiter d'environnement ou de dépendances externes à l'exécution.

Chargement dynamique Technique consistant à charger des données ou ressources pendant l'exécution d'un programme.

CloudCompare Logiciel libre spécialisé dans le traitement et la visualisation de nuages de points 3D.

Common Language Infrastructure (CLI) Norme définissant un environnement d'exécution portable pour le code managé, qui permet à différents langages de compiler en un bytecode commun (le CIL) et d'être exécutés par un runtime compatible, comme le CLR de .NET.

Common Language Runtime (CLR) Composant principal de la plateforme .NET, responsable de l'exécution du code managé. Le CLR gère la compilation Just-In-Time (JIT), la gestion de la mémoire (ramasse-miettes), la sécurité et l'interopérabilité avec du code natif.

Communication inter-processus Mécanisme permettant à différents processus ou programmes s'exécutant sur un même ordinateur ou sur des machines différentes d'échanger des données ou des messages.

Component Object Model (COM) Technologie Microsoft pour l'interopérabilité entre composants logiciels, permettant à des objets logiciels d'être utilisés à travers différents langages et processus via une interface binaire standardisée.

DLL Dynamic Link Library (bibliothèque de liens dynamiques).

Doxygen Outil de génération automatique de documentation technique à partir de commentaires dans le code source (principalement en C++, C, Java, etc.), produisant des sorties en HTML, PDF ou d'autres formats.

Epic Games Entreprise américaine spécialisée dans le développement de jeux vidéo et de technologies, notamment connue pour le moteur de jeu Unreal Engine.

Git Système de contrôle de version décentralisé utilisé pour suivre les modifications dans les fichiers de code source.

Géomatique Discipline combinant géographie et informatique pour la collecte, le traitement, l'analyse et la visualisation de données géospatiales.

IDE Integrated Development Environment (environnement de développement intégré), un logiciel qui centralise les outils nécessaires au développement.

Interopérabilité Capacité de différents systèmes ou logiciels à échanger des informations et à fonctionner ensemble.

Intranet Réseau informatique interne à une organisation, utilisé pour partager des informations, des ressources et des outils de travail.

LasTools Ensemble d'outils spécialisés dans le traitement et la conversion de fichiers LiDAR (.las, .laz).

LaTeX Système de composition de documents basé sur le langage TeX, largement utilisé pour la production de documents scientifiques et techniques.

LiDAR Light Detection and Ranging, une méthode de télédétection par laser utilisée pour obtenir des données géospatiales précises.

MicroStation Logiciel de CAO utilisé pour la cartographie et la modélisation d'infrastructures.

MNS Modèle Numérique de Surface, représentant la surface terrestre incluant les objets présents (arbres, bâtiments, etc.).

MNT Modèle Numérique de Terrain, représentant la surface du sol sans les éléments surfaciques (végétation, bâtiments, etc.).

Module Composant logiciel autonome qui regroupe un ensemble de fonctionnalités, pouvant être intégré dans un projet pour étendre ses capacités.

Moteur de rendu graphique Logiciel ou composant permettant de générer des images à partir de données 3D pour les afficher à l'écran.

Namespace Espace de noms utilisé en programmation pour organiser logiquement les éléments (fonctions, classes, constantes, etc.) afin d'éviter les conflits de nommage entre différentes parties d'un programme ou entre bibliothèques.

Nuage de points Ensemble de points dans un espace 3D, généralement issu d'un scanner LiDAR ou de photogrammétrie, représentant la surface d'un objet ou d'un terrain.

Octree Structure de données arborescente utilisée pour partitionner l'espace 3D en sous-volumes, facilitant l'optimisation du rendu et la gestion efficace des données spatiales.

Open3D Bibliothèque open-source spécialisée dans le traitement et l'analyse de données 3D, particulièrement utilisée en recherche scientifique.

OpenGL API graphique multiplateforme pour le rendu 2D et 3D, offrant une grande flexibilité mais nécessitant un développement graphique approfondi.

Patch Modification apportée à un fichier ou un programme existant pour corriger un bug, améliorer une fonctionnalité ou résoudre un problème de compatibilité.

Pawn Actor Dans Unreal Engine, un Pawn est un acteur représentant une entité contrôlable par l'utilisateur ou l'intelligence artificielle, utilisé pour gérer la navigation et les interactions dans la scène 3D.

Photogrammétrie Technique qui permet d'obtenir des mesures précises à partir de photographies, souvent utilisée pour modéliser des objets ou terrains en 3D.

Plugin Module logiciel additionnel qui étend les fonctionnalités d'un logiciel principal, souvent utilisé dans Unreal Engine pour ajouter des fonctionnalités spécifiques.

Runtime Phase d'exécution d'un programme, durant laquelle le code compilé est lancé et fonctionne en temps réel.

Runtime .NET Environnement d'exécution qui permet d'exécuter des applications développées avec la plateforme .NET. Il comprend le Common Language Runtime (CLR), qui gère la compilation Just-In-Time, la gestion de la mémoire, la sécurité, et l'interopérabilité avec du code natif.

SASU Société par Actions Simplifiée Unipersonnelle, forme juridique d'entreprise en France à associé unique.

Template First Person Gabarit de projet proposé par Unreal Engine contenant les éléments de base pour créer une application ou un jeu en vue subjective (personnage, contrôles, caméra).

TerraScan Extension de MicroStation dédiée à la classification automatique de nuages de points.

Télédétection Ensemble des techniques permettant d'obtenir des informations à distance sur des objets ou phénomènes, généralement via des capteurs embarqués (satellites, drones, avions).

Unity Moteur de jeu multiplateforme développé par Unity Technologies, utilisé pour créer des applications interactives en 2D et 3D.

Unreal Engine Moteur de jeu vidéo développé par Epic Games, utilisé ici pour la visualisation interactive 3D.

UStaticMesh Type d'objet 3D statique dans Unreal Engine, utilisé pour représenter des formes solides dans la scène sans comportement dynamique.

Vision LiDAR Logiciel permettant la visualisation et l'exploration de grands ensembles de données LiDAR.

Visualiseur 3D Application permettant d'afficher et manipuler des objets ou environnements en trois dimensions.

Widget Unreal Élément d'interface utilisateur dans Unreal Engine, utilisé pour afficher des menus, du texte ou des icônes à l'écran.

WPF Windows Presentation Foundation, un framework de Microsoft utilisé pour créer des interfaces graphiques riches sous Windows.

Wrapper Programme ou couche logicielle qui encapsule une autre bibliothèque ou API, facilitant son utilisation ou son intégration dans un autre langage ou environnement.

Sources

Toutes les sources ont été consultées entre avril et juin 2025.

Sources publiées

[2] Sébastien Cazalas. *R406 - Communication interne*. Support de cours, IUT de Nantes, BUT Informatique. 2024.

[3] Sébastien Cazalas. *R407 - Projet personnel et professionnel*. Support de cours, IUT de Nantes, BUT Informatique. 2024.

[35] Loïg Jezequel. *R1.01 - Initiation au développement*. Support de cours, IUT de Nantes, BUT Informatique. 2023.

[36] Loig Jezequel. *R305 - Programmation système*. Support de cours, IUT de Nantes, BUT Informatique. 2024.

[38] Arnaux Lanoix. *R2.01 – Développement orienté objet*. Support de cours, IUT de Nantes, BUT Informatique. 2023.

[48] François Simonneau. *R1.07 – Outils mathématiques fondamentaux*. Support de cours, IUT de Nantes, BUT Informatique. 2023.

Sources en ligne

[1] ASPRSorg / GitHub. *LAS GitHub Repository*. https://github.com/ASPRSorg/LAS.

[4] CloudCompare / GitHub. *CloudCompare GitHub Repository*. https://github.com/CloudCompare/CloudCompare.

[5] Doxygen. *Doxygen Official Website*. https://www.doxygen.nl/.

[6] Epic Games. *BlueprintAPI Button - Unreal Engine Documentation*. https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/Button.

[7] Epic Games. *BlueprintAPI Input Mouse Events*. https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/Input/MouseEvents.

[8] Epic Games. *Building Unreal Engine as a Library*. https://dev.epicgames.com/documentation/en-us/unreal-engine/building-unreal-engine-as-a-library?application_version=4.27.

[9] Epic Games. *Creating UMG Widget Templates in Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/creating-umg-widget-templates-in-unreal-engine.

[10] Epic Games. *Downloading Unreal Engine Source Code*. https://dev.epicgames.com/documentation/en-us/unreal-engine/downloading-unreal-engine-source-code?application_version=4.27.

[11] Epic Games. *ELidarPointCloudColorationMode - Unreal Engine API*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/ELidarPointCloudColorationMode.

[12] Epic Games. *ELidarPointCloudScalingMethod - Unreal Engine API*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/ELidarPointCloudScalingMethod.

[13] Epic Games. *ELidarPointCloudSpriteShape - Unreal Engine API*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/ELidarPointCloudSprite

[14] Epic Games. *FLidarPointCloudAsyncParameters - Unreal Engine API*. 2025. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/FLidarPointCloudAsyncParameters.

[15] Epic Games. *FLidarPointCloudComponentRenderP- - Unreal Engine API*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/FLidarPointCloudComponentRenderP-.

[16] Epic Games. *Georeferencing a Level in Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/georeferencing-a-level-in-unreal-engine#settingupthegeoreferencingsystem.

[17] Epic Games. *IDesktopPlatform OpenFileDialog API - Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Developer/DesktopPlatform/IDesktopPlatform/OpenFileDialog/1?application_version=5.5.

[18] Epic Games. *Lidar Point Cloud API - Unreal Engine Documentation*. https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/LidarPointCloud.

[19] Epic Games. *Lidar Point Cloud Plugin for Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/lidar-point-cloud-plugin-for-unreal-engine.

[20] Epic Games. *Lidar Point Cloud Runtime API - Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime.

[21] Epic Games. *Packaging Unreal Engine Projects*. https://dev.epicgames.com/documentation/en-us/unreal-engine/packaging-unreal-engine-projects?application_version=5.6.

[22] Epic Games. *Reflection System in Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/reflection-system-in-unreal-engine?application_version=5.5.

[23] Epic Games. *Setting Up User Inputs in Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/setting-up-user-inputs-in-unreal-engine?application_version=5.4.

[24] Epic Games. *Slate Overview for Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/slate-overview-for-unreal-engine.

[25] Epic Games. *ULidarPointCloud - Unreal Engine API*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/ULidarPointCloud.

[26] Epic Games. *ULidarPointCloud AlignClouds - Unreal Engine API*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/ULidarPointCloud/AlignClouds.

[27] Epic Games. *ULidarPointCloud CreateFromFile - Unreal Engine API*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/ULidarPointCloud/CreateFromFile/2.

[28] Epic Games. *ULidarPointCloudBlueprintLibrary - Unreal Engine API*. https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/LidarPointCloudRuntime/ULidarPointCloudBlueprintLibrary.

[29] Epic Games. *Unreal Engine Official Website (FR)*. https://www.unrealengine.com/fr.

[30] Epic Games. *Using a Single Line Trace (Raycast by Channel) in Unreal Engine*. https://dev.epicgames.com/documentation/en-us/unreal-engine/using-a-single-line-trace-raycast-by-channel-in-unreal-engine.

[31] Epic Games. *Using the UnrealVS Extension for Unreal Engine C++ Projects*. https://dev.epicgames.com/documentation/en-us/unreal-engine/using-the-unrealvs-extension-for-unreal-engine-cplusplus-projects?application_version=5.6.

[32] Epic Games / GitHub. *Unreal Engine GitHub Repository*. https://github.com/EpicGames/UnrealEngine.

[33] Esri ArcGIS. *What is a LAS Dataset ? - ArcGIS Desktop*. https://desktop.arcgis.com/fr/arcmap/latest/manage-data/las-dataset/what-is-a-las-dataset-.htm.

[34] *Integrating Third-Party Libraries into Unreal Engine — Epic Games Documentation*. https://dev.epicgames.com/documentation/en-us/unreal-engine/integrating-third-party-libraries-into-unreal-engine.

[37] *Kadran — Pappers*. https://www.pappers.fr/entreprise/kadran-815119383.

[39] *Matrices — Unigine Developer Documentation*. https://developer.unigine.com/en/docs/latest/code/fundamentals/matrices/index?rlang=cpp.

[40] Microsoft. *Command Prompt and PowerShell Reference - Visual Studio*. https://learn.microsoft.com/en-us/visualstudio/ide/reference/command-prompt-powershell?view=vs-2022.

[41] Microsoft. *DLLs in Visual C++*. https://learn.microsoft.com/en-us/cpp/build/dlls-in-visual-cpp?view=msvc-170.

[42] Microsoft. *Dynamic Link Library - Windows Client Troubleshooting*. https://learn.microsoft.com/en-us/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library.

[43] Microsoft. *Walkthrough : Compiling a C++ Program that Targets the CLR in Visual Studio*. https://learn.microsoft.com/en-us/cpp/dotnet/walkthrough-compiling-a-cpp-program-that-targets-the-clr-in-visual-studio?view=msvc-170.

[44] Microsoft. *Walkthrough : Hosting a Windows Forms Control in WPF*. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/walkthrough-hosting-a-windows-forms-control-in-wpf?view=netframeworkdesktop-4.8.

[45] Microsoft. *WPF Overview - .NET Desktop*. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/.

[46] Microsoft / GitHub. *VC UE Extensions - Optional Enabling the Plugin*. https://github.com/microsoft/vc-ue-extensions/blob/main/README.md#optional-enabling-the-plugin.

[47] NuGet. *DllExport Package on NuGet*. https://www.nuget.org/packages/DllExport/.

[49] tabu-issues-for-future-maintainer / GitHub. *tabu.sty - GitHub Repository*. https://github.com/tabu-issues-for-future-maintainer/tabu/blob/main/tabu.sty.

[50] TeX Users Group. *TeX Live - Official Website*. https://www.tug.org/texlive/.

[51] Unreal Engine Forums. *Change Color of Points in Point Cloud*. https://forums.unrealengine.com/t/change-color-of-points-in-point-cloud/1534100.

[52] Unreal Engine Forums. *Generate 3D Point Cloud*. https://forums.unrealengine.com/t/generate-3d-point-cloud/661312/21.

[53] Unreal Engine Forums. *Is it Possible to Call a Managed .NET DLL from UE4 C++ Project ? (Post 4)*. https://forums.unrealengine.com/t/is-it-possible-to-call-a-managed-net-dll-from-ue4-c-project/281570/4.

[54] Unreal Engine Forums. *Is it Possible to Call a Managed .NET DLL from UE4 C++ Project ? (Post 6)*. https://forums.unrealengine.com/t/is-it-possible-to-call-a-managed-net-dll-from-ue4-c-project/281570/6.

[55] Unreal Engine Forums. *Lidar Point Cloud Line Trace Issue*. https://forums.unrealengine.com/t/lidar-point-cloud-line-trace-issue/237002.

[56] Unreal Engine Forums. *Packaged Game Starts Fullscreen*. https://forums.unrealengine.com/t/packaged-game-starts-fullscreen/399257.

[57] Unreal Engine Forums. *Point Cloud Plugin*. https://forums.unrealengine.com/t/point-cloud-plugin/106645.

[58] Unreal Engine Forums. *Populating Empty Lidar Point Cloud*. https://forums.unrealengine.com/t/populating-empty-lidar-point-cloud/658944.

[59] Wikipedia. *LAS File Format - Wikipedia*. https://en.wikipedia.org/wiki/LAS_file_format.

[60] YellowScan. *Lidar Point Cloud Basics*. https://www.yellowscan.com/knowledge/lidar-point-cloud-basics/.

Annexes

Journal de stage

Date	Ce que j'ai fait aujourd'hui	Ce qui me reste à faire
22 avril 2025	Installation du poste, prise de contact avec l'environnement de travail. Quelques recherches. Pas de session ce jour-là.	Obtenir la session pour pouvoir travailler.
23 avril 2025	Réception de la session en début d'après-midi. Installation des logiciels et mise en place de l'environnement. Recherches complémentaires.	Approfondir les outils et démarrer des tests pratiques.
24 avril 2025	Recherches et premiers tests sur Unreal Engine. Importation de nuages de points dans l'éditeur et à l'exécution.	Explorer plus de fonctionnalités d'Unreal Engine.
25 avril 2025	Expérimentations techniques. Recherches sur la compilation d'Unreal en tant que DLL et intégration dans une application WPF.	Comprendre les appels C# vers Unreal.
28 avril – 2 mai 2025	Recherches et tests sur l'interfaçage de DLL C# dans Unreal. Fonctionnement atteint le 2 mai.	Préparer une base stable pour commencer la navigation.
5 mai 2025	Échange avec des collègues de Kadran dans leurs bureaux. Découverte de TerraScan, prise de notes et premiers essais de prise en main.	Comprendre les fonctionnalités clés de TerraScan.
6–9 mai 2025	Début de la navigation : zoom in/out, déplacement panoramique (panning).	Ajouter l'orbitage pour une navigation complète.
12–23 mai 2025	Développement de l'orbitage. D'abord sur objet Unreal, puis sur point précis du nuage. Importation de fichiers .las/.laz. Test prévu repoussé.	Finaliser l'import et tester la stabilité.

Date	**Ce que j'ai fait aujourd'hui**	**Ce qui me reste à faire**
26–28 mai 2025	Amélioration de la navigation : vitesse de zoom, zoom adaptatif. Coloration des points selon leur classification.	Ajouter des fonctions avancées (géoréférencement).
2–6 juin 2025	Ajout du géoréférencement. Début de documentation et refactorisation du code.	Continuer la documentation. Ajout d'options utilisateur.
6 juin 2025	Ajout de la fonctionnalité pour changer la taille des points manuellement.	Tester l'ergonomie de l'interface.
10 juin 2025	Documentation, ajout d'un indicateur visuel sur le point d'orbitage.	Préparer tests utilisateurs.
11–13 juin 2025	Test utilisateur par un membre de l'équipe. Corrections sur les points faisables dans le temps imparti.	Finaliser les modifications mineures.
16–20 juin 2025	Rédaction de la documentation technique. Début de la documentation générale du projet. Création d'un document de setup Unreal/VS.	Terminer toute la documentation utilisateur.
23–27 juin 2025	Finalisation du document de setup. Nettoyage du poste, vérifications globales (code, documentation). Nettoyage du rapport.	L'entreprise devra idéalement retester la dernière version du logiciel, car je n'ai pas eu le temps d'intégrer toutes les modifications demandées lors du dernier test.

Éléments complémentaires sur l'entreprise

Parties intéressées

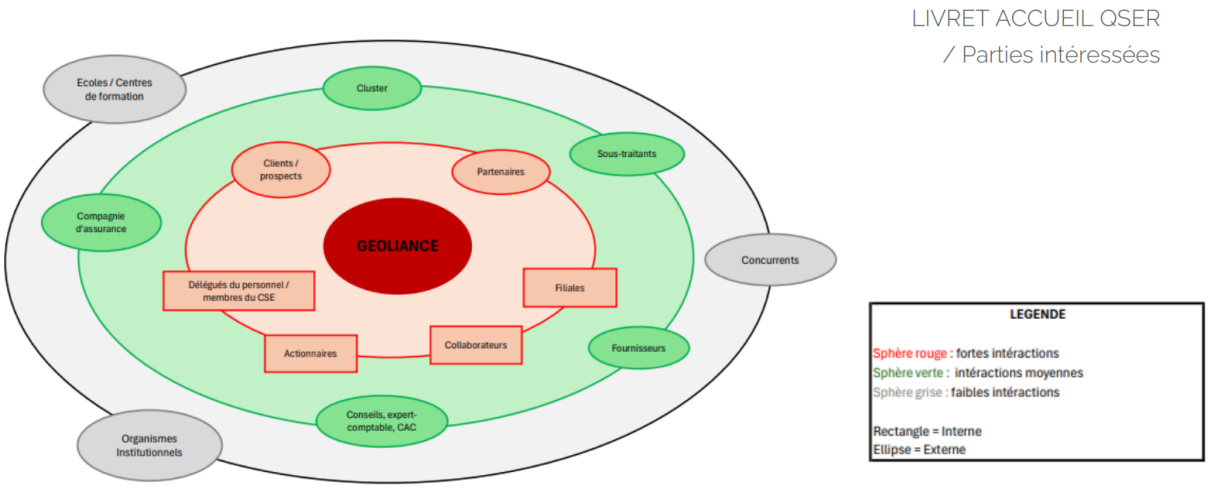


FIGURE 1 – Parties intéressées du groupe Géoliance

Carte des implantations

FIGURE 2 – Implantation géographique des bureaux de Kadran

Informations sur les fichiers LiDAR

Ce fichier `.laz` contient environ 2 millions de points et a été généré à l'aide du logiciel `TerraScan`. Voici les principales informations extraites à l'aide de l'outil `lasinfo` de

LASTools.

Extrait du header LAS

```
lasinfo (250517) report for 'pylones_classif_m3d\0392-6584_20240808_BXLIEL41MARA7_1.la
reporting all LAS header entries:
  file signature:             'LASF'
  file source ID:             0
  global_encoding:            17
  project ID GUID data 1-4:   00000000-0000-0000-0000-000000000000
  version major.minor:        1.4
  system identifier:          'PDAL'
  generating software:        'PDAL 2.6.3 (d12128)'
  file creation day/year:     140/2025
  header size:                375
  offset to point data:       3053
  number var. length records: 3
  point data format:          8
  point data record length:   79
  number of point records:    0
  number of points by return: 0 0 0 0 0
  scale factor x y z:         0.001 0.001 0.001
  offset x y z:               0 6000000 0
  min x y z:                  392531.211 6583518.709 2.237
  max x y z:                  392551.205 6583538.689 200.193
  start of waveform data packet record: 0
  start of first extended variable length record: 0
  number of extended_variable length records: 0
  extended number of point records: 66260
  extended number of points by return: 61549 4257 411 26 1 0 0 0 0 0 0 0 0 0 0
```

Informations géospatiales (système de coordonnées)

Le fichier est référencé selon le système **RGF93 / Lambert-93 (EPSG :2154)** :

```
PROJCS["RGF93 v1 / Lambert-93",
  GEOGCS["RGF93 v1",
    DATUM["Reseau_Geodesique_Francais_1993_v1",
      SPHEROID["GRS 1980",6378137,298.257222101,AUTHORITY["EPSG","7019"]],
      AUTHORITY["EPSG","6171"]],
    PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4171"]],
  PROJECTION["Lambert_Conformal_Conic_2SP"],
  PARAMETER["latitude_of_origin",46.5],
  PARAMETER["central_meridian",3],
  PARAMETER["standard_parallel_1",49],
  PARAMETER["standard_parallel_2",44],
```

```
  PARAMETER["false_easting",700000],
  PARAMETER["false_northing",6600000],
  UNIT["metre",1,AUTHORITY["EPSG","9001"]],
  AXIS["Easting",EAST],
  AXIS["Northing",NORTH],
  AUTHORITY["EPSG","2154"]]
```

Résumé sur les retours de points et classifications

```
number of first returns:        61565
number of intermediate returns: 438
number of last returns:         61535
number of single returns:       57278
histogram of classification of points:
           52755  never classified (0)
            5806  ground (2)
            1938  wire conductor (14)
            5745  tower (15)
histogram of extended classification of points:
               1  extended classification (230)
               1  extended classification (231)
               6  extended classification (232)
               6  extended classification (233)
               1  extended classification (234)
               1  extended classification (235)
```

Extrait de données .xyz (1 point)

```
727542.50000000 6375853.50000000 751.45300293 255 255 255 40762.000000 1.000000 1.0000
```

Exemple complet du fichier CSV de classification

Le fichier `.csv` ci-dessous illustre la façon dont sont définies les classes de points et leurs couleurs associées. Chaque ligne contient un identifiant de classe et un nom de couleur correspondant, permettant à l'application de personnaliser l'affichage selon les données importées.

```
0,White
1,Gray
2,Green
```

Système de coordonnées Lambert 93

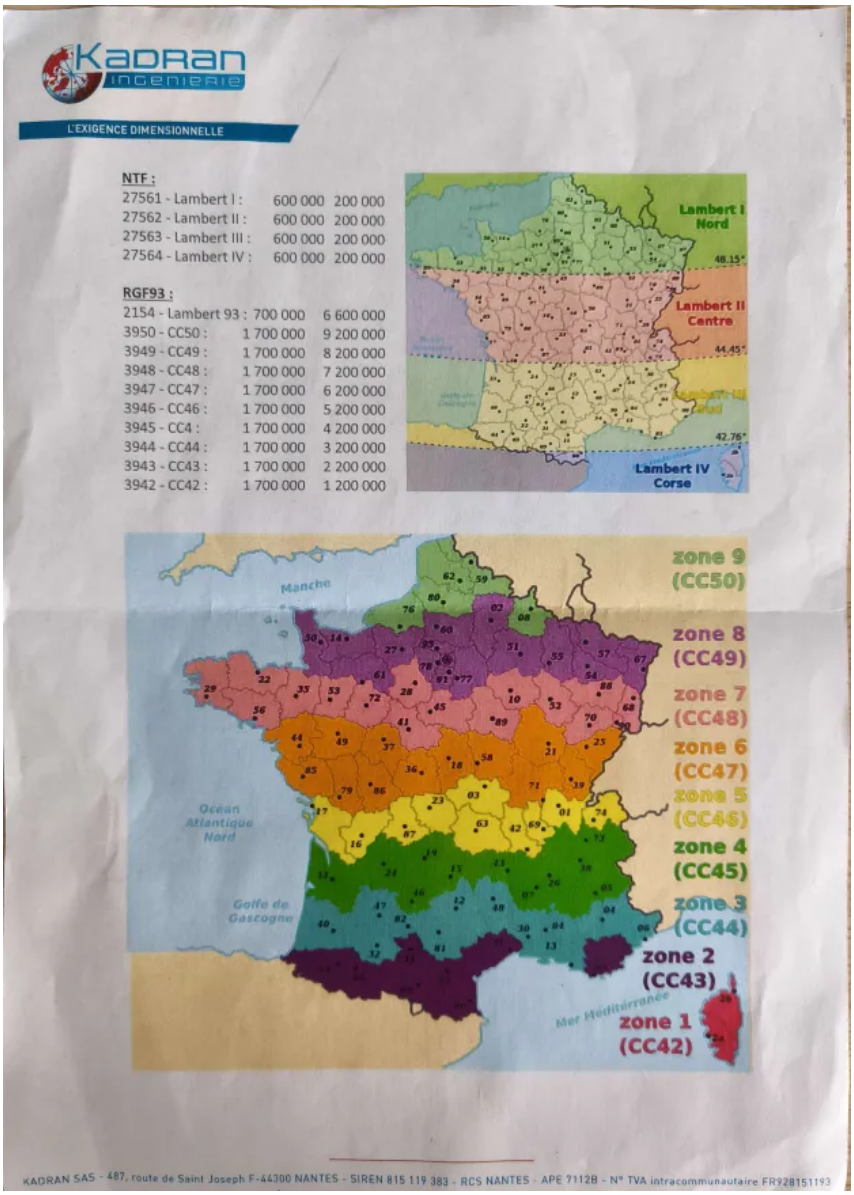


FIGURE 3 – Représentation du système de coordonnées Lambert 93

Eléments de débogage visuel

Cette annexe présente des captures d'écran réalisées lors des phases de développement et de test. Elles illustrent les outils de débogage visuel mis en place pour analyser le comportement des raycasts et des collisions sur les nuages de points.

Raycast Unreal standard

Les sphères rouges représentent les points de collision, c'est-à-dire les points du nuage autour desquels la caméra doit orbiter. Les rayons illustrent les raycasts, depuis leur point de départ jusqu'à leur point d'impact.

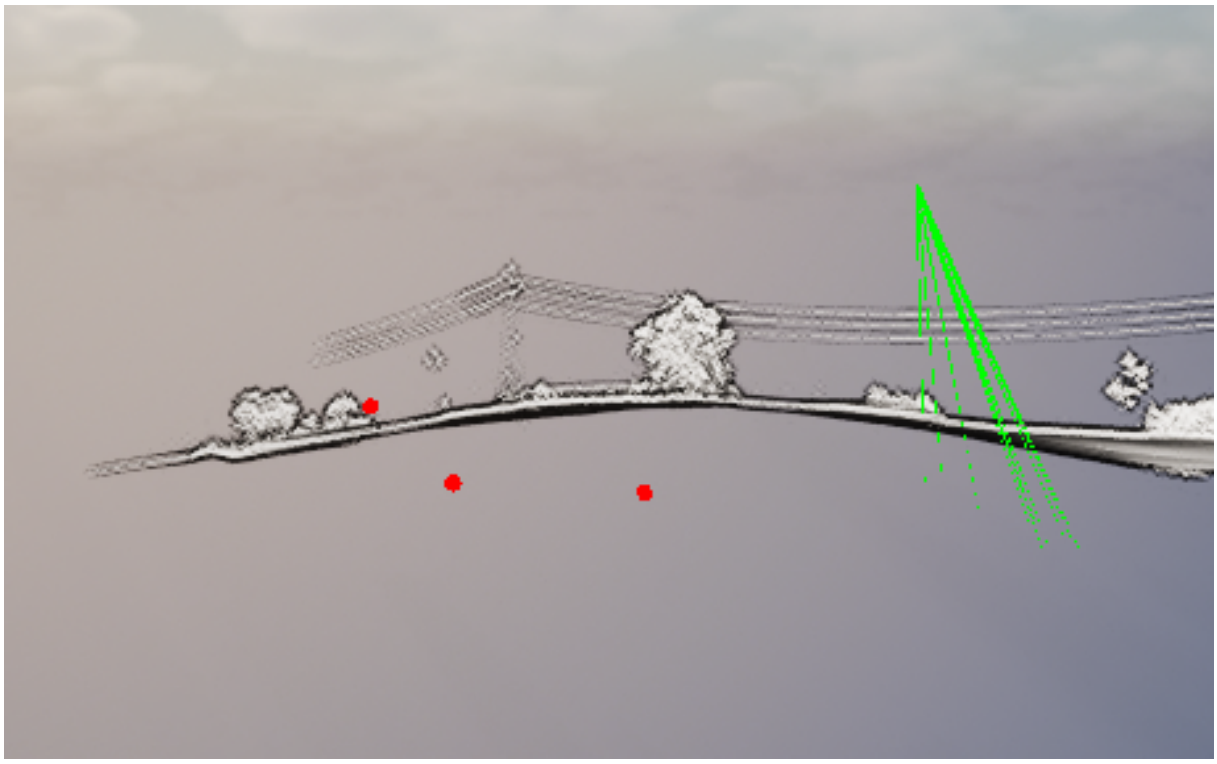


FIGURE 4 – Essai de raycast depuis le code C++ sur un nuage de points : visualisation des sphères positionnées aux points d'impact.

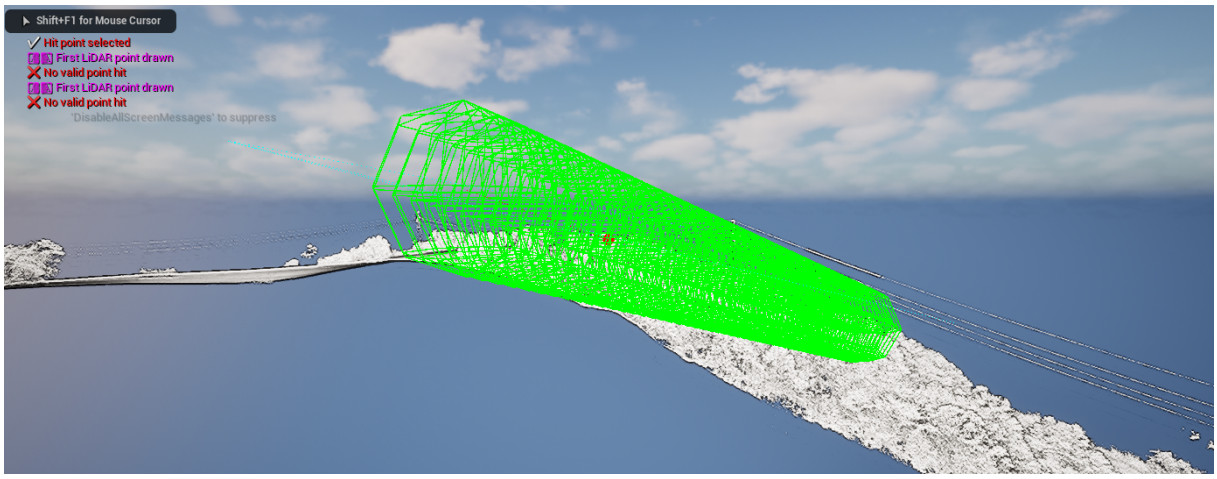


FIGURE 5 – Essai de raycast depuis le code C++ sur un nuage de points : variation de l'angle et vérification des impacts.

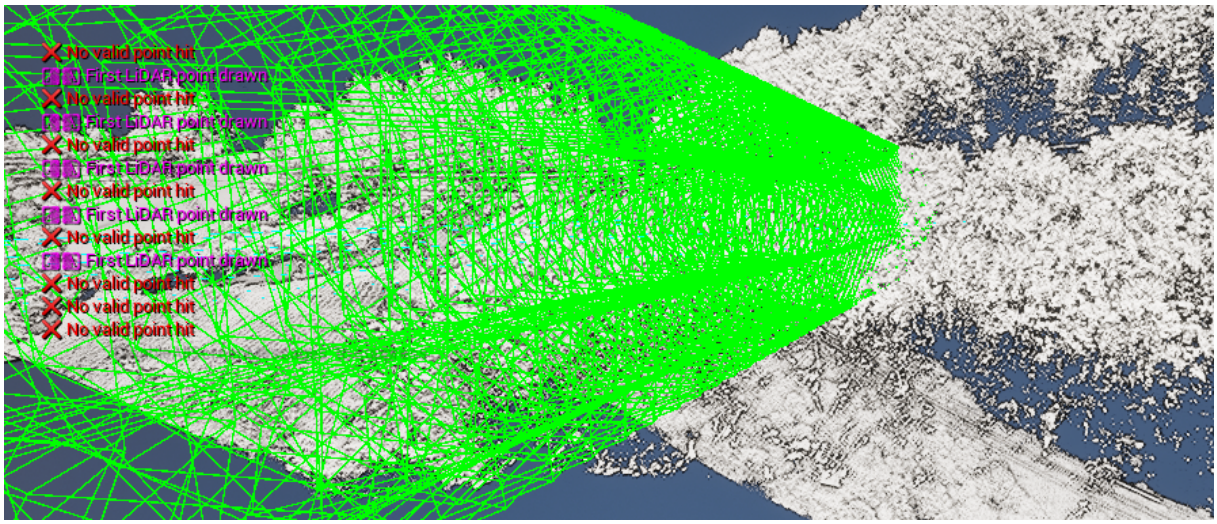


FIGURE 6 – Essai de raycast depuis le code C++ : tests sur des zones éloignées du nuage de points.

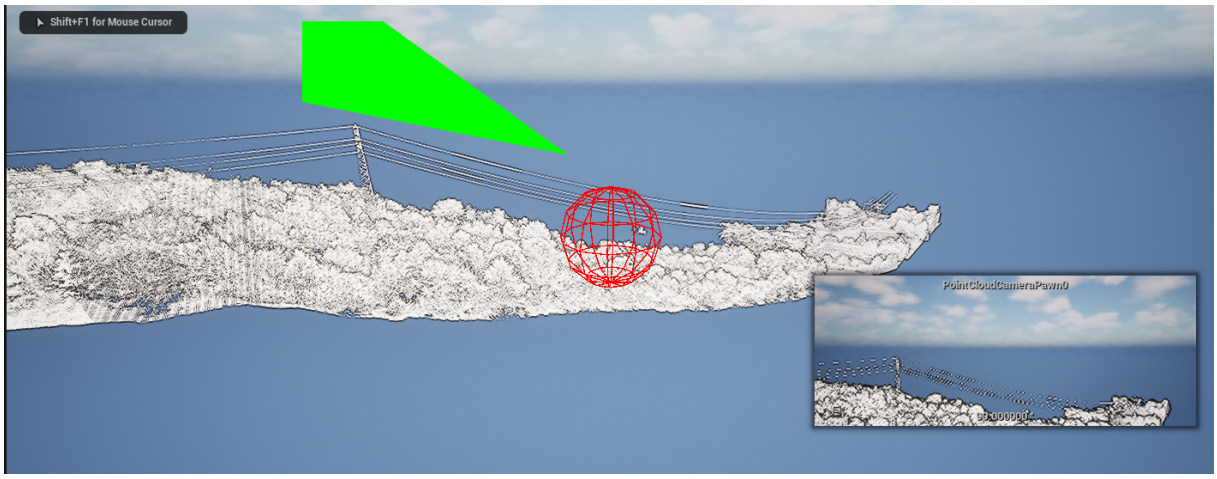


FIGURE 7 – Essai de raycast depuis le code C++ : visualisation des trajectoires de rayons et détection de collisions.

Raycast avec le projet test

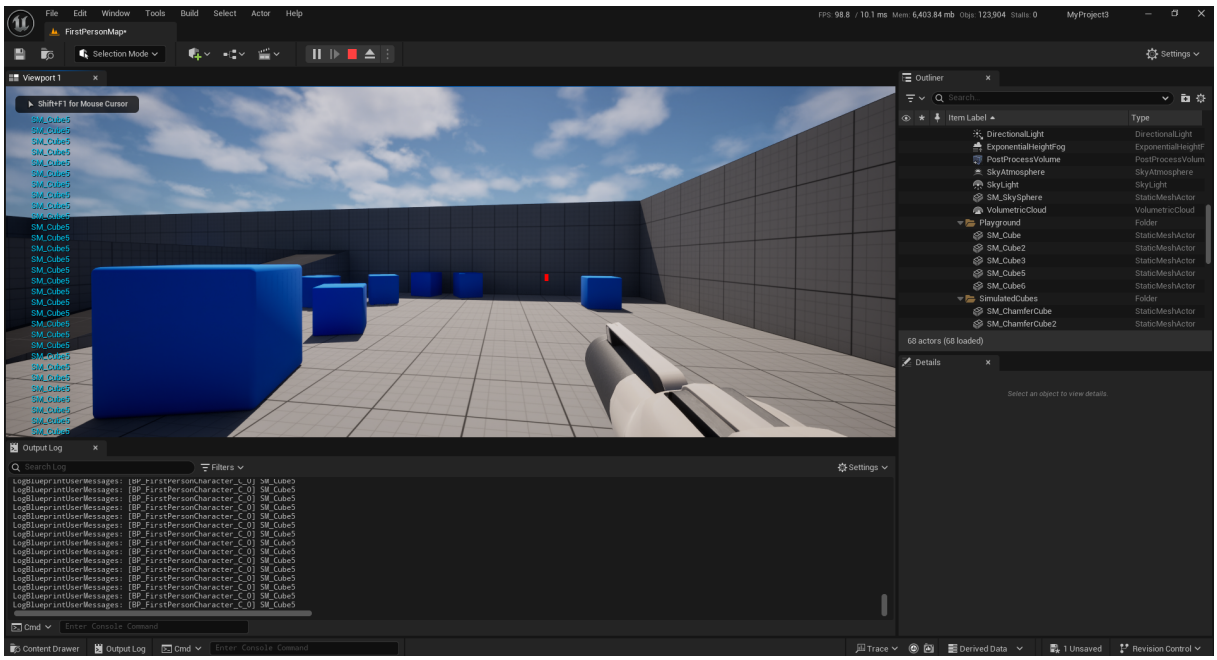


FIGURE 8 – Image de debug montrant les impacts de raycast dans le projet de test. Les sphères rouges indiquent les collisions détectées.

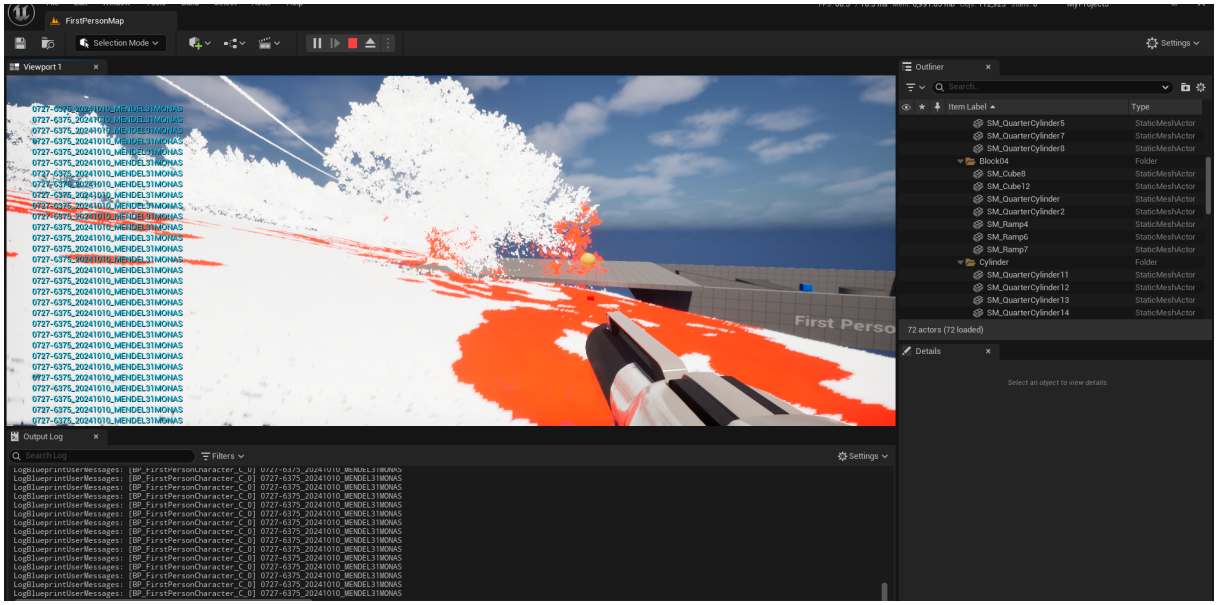


FIGURE 9 – Image de debug montrant les impacts de raycast dans le projet de test. Les sphères rouges indiquent les collisions détectées.

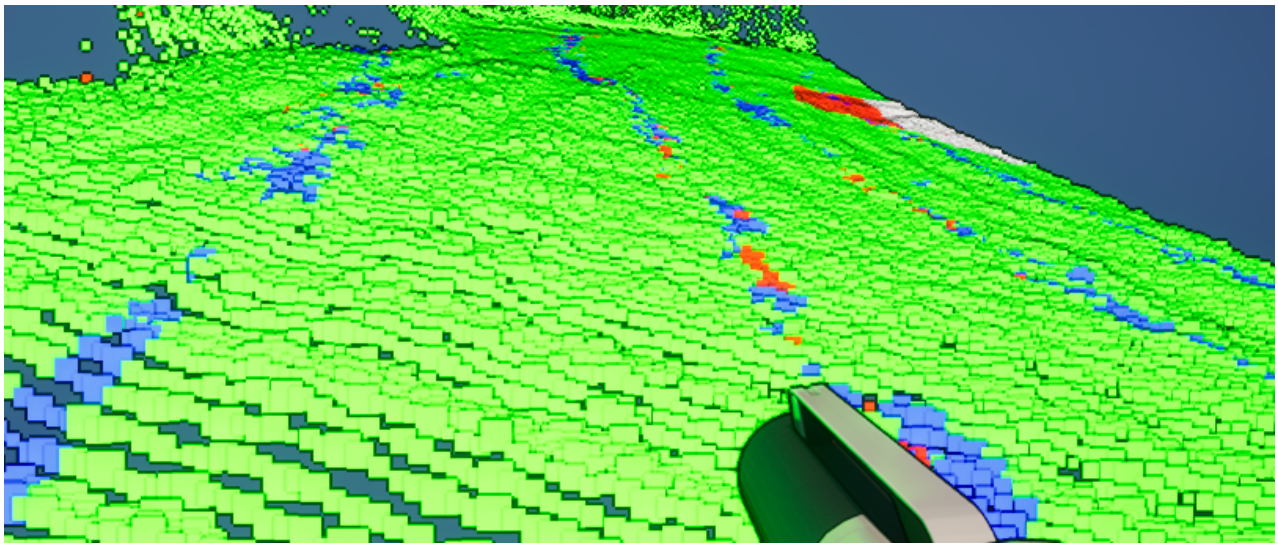


FIGURE 10 – Visualisation des raycasts avec affichage de leurs trajectoires dans le projet de test.

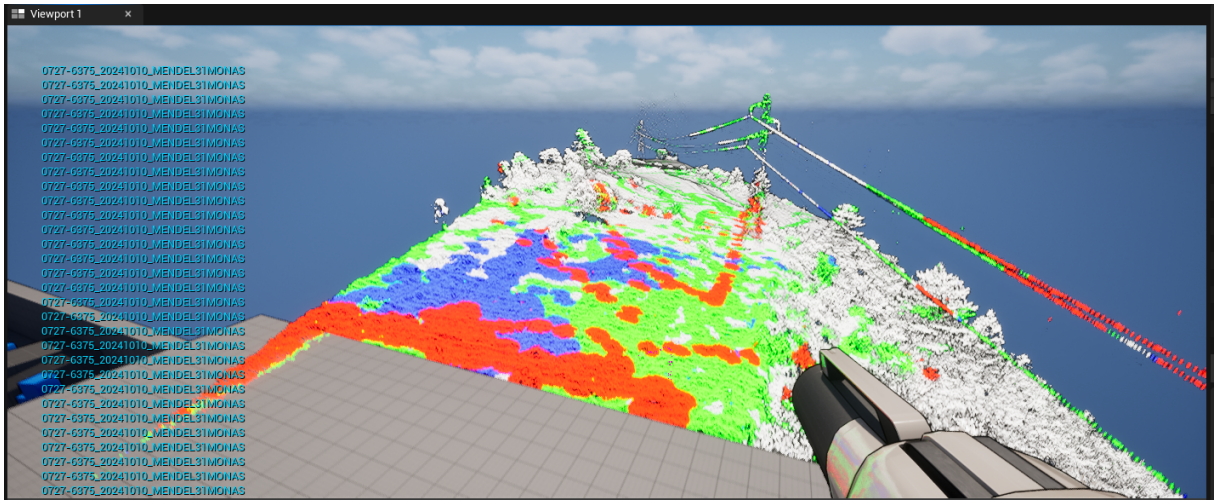


FIGURE 11 – Essai de raycast sur différentes zones du nuage de points, utilisé pour valider le fonctionnement dans un contexte de test.

Blueprints de test

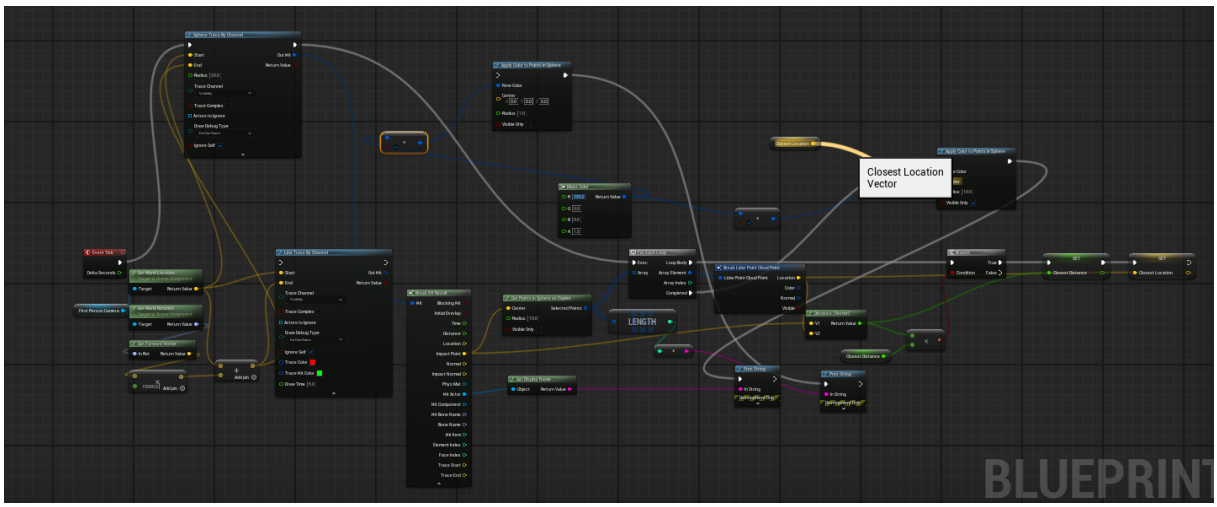


FIGURE 12 – Blueprint de test d'un raycast en vue à la première personne

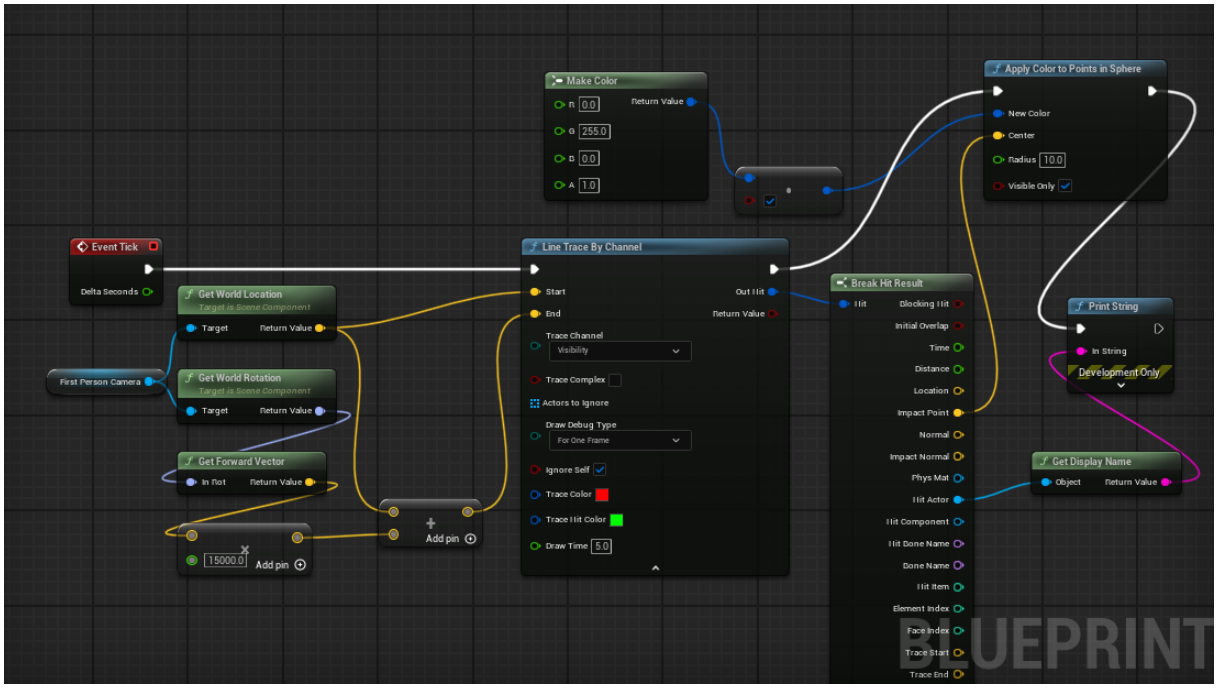


FIGURE 13 – Blueprint de test d'un raycast en vue à la première personne

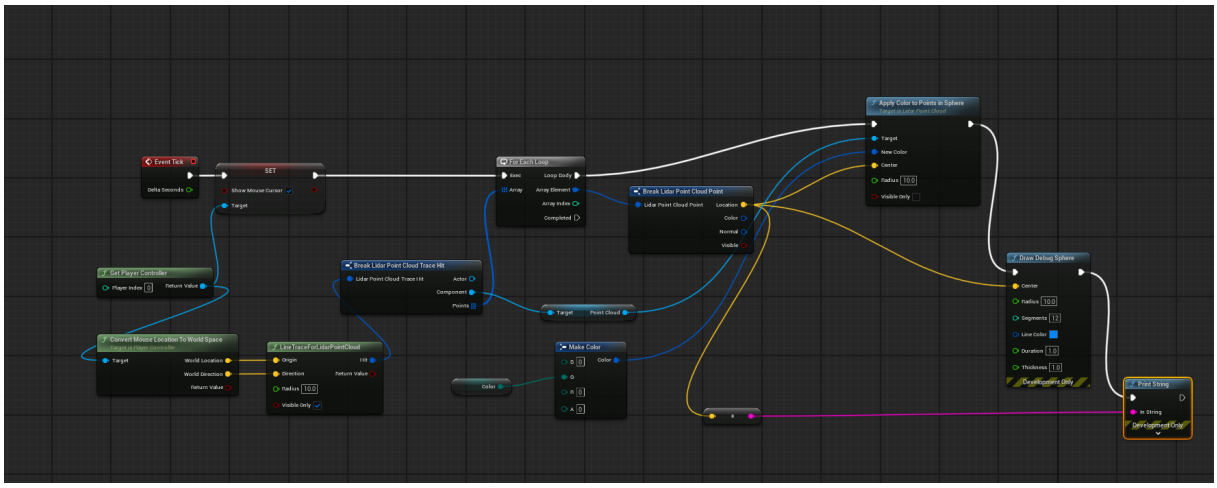


FIGURE 14 – Blueprint illustrant un raycast avec la fonction du plugin.

Code source et documentation technique

Les extraits de code source ainsi que la documentation technique ne sont pas inclus dans ce document, mais sont disponibles dans le second PDF fourni en annexe. Ce fichier regroupe plusieurs éléments habituellement séparés, tels que le ***README*** *du projet, la documentation technique de configuration (Unreal Engine / Visual Studio), ainsi que la documentation générée par Doxygen. Afin de centraliser l'ensemble des ressources liées au développement.*