POC Fonctions raster renderer QGIS

Par Les Pinguouins :

Antoine ANQUETIL

Jean-Baptiste OLVIER

Antonin OLLIER

Qiuling LIN

Commanditaire :

Nicolas DAVID

April 2025

Résumé

Ce rapport présente le projet *Render QGIS*, qui a pour objectif d'étendre les capacités de rendu raster du logiciel QGIS, à travers deux approches complémentaires : la modification du code source en C++, et le développement d'un plugin Python.

Dans un premier temps, une étude approfondie du fonctionnement interne du moteur de rendu de QGIS est menée. Elle détaille les différentes étapes du processus de rendu – de l'appel initial via `QgsMapCanvas::refresh()`, jusqu'à l'exécution de la fonction `doRender()` – en expliquant la structure modulaire basée sur les classes `QgsMapRendererJob`, `QgsRasterRenderer` et le pipeline `QgsRasterPipe`.

Dans un second temps, le rapport explore l'implémentation d'un renderer personnalisé directement dans le code source C++ de QGIS. Cette approche permet une intégration native du renderer à l'interface de QGIS, au système de symbologie, ainsi qu'aux exports. La méthode centrale à implémenter est `block()`, qui produit les blocs d'image affichés. Le processus d'intégration (interface graphique, enregistrement dans le registre, gestion mémoire avec SIP) est également détaillé.

Enfin, une alternative plus accessible est proposée via le développement d'un plugin Python nommé *Render*. Ce plugin permet de réaliser deux traitements classiques en télédétection : le calcul du NDVI et la génération d'un ombrage (hillshade). Basé sur les bibliothèques NumPy et GDAL, il permet une intégration fluide au sein de QGIS, avec une interface utilisateur simple conçue sous Qt.

Ce travail propose ainsi un double interêt : d'une part, une meilleure compréhension de l'architecture de QGIS pour le rendu raster, et d'autre part une preuve de faisabilité de modification du processus de rendu.

Table des matières

Résumé **1**

1 Introduction **3**
1.1 Contexte . . . 3
1.2 Présentation du Projet . . . 3
1.3 Objectifs . . . 3
1.4 Organisation du travail . . . 3

2 Étapes clefs du rendering QGIS **5**
2.1 Démarrage et instanciation du processus . . . 5
2.2 Configuration, orchestration et lancement des tâches de rendu . . . 6
2.3 Implémentation concréte dans le cas du rendu séquentiel . . . 6
2.4 Rendu raster dans QGIS . . . 7
2.4.1 Types de raster renderers . . . 7
2.4.2 Séquence de rendu . . . 8
2.4.3 Pipeline de rendu raster . . . 8
2.4.4 Traitement par bloc . . . 8
2.4.5 Résumé . . . 9

3 Option modification du code source **10**
3.1 Présentation de la solution . . . 10
3.2 De quelle manière éditer le code source C++ de QGIS pour créer un Renderer ? . . . 10
3.2.1 Compréhension des dépendances . . . 10
3.2.2 Ce qu'il faut mettre en place . . . 12
3.2.3 Explication du `SIP_FACTORY` . . . 15

4 Présentation du plugin Render **17**
4.1 Fonctionnalités principales . . . 17
4.2 Traitement raster avec NumPy et QGIS . . . 18
4.3 Interface utilisateur . . . 18
4.4 Dépendances techniques . . . 19
4.5 Performance et limitations . . . 19

5 Conclusion **21**
5.1 Approche 1 : Développement d'un plugin Python . . . 21
5.2 Approche 2 : Modification du code source C++ de QGIS . . . 22
5.3 Synthèse comparative . . . 22

1 Introduction

1.1 Contexte

Le moteur de rendu raster de QGIS repose sur une chaîne de traitement performante, mais difficile à personnaliser sans modifier son code source en C++. Des fonctions comme le hillshade ou les palettes de couleurs sont disponibles, mais peu extensibles dynamiquement.

Inspiré par le modèle des raster functions d'ArcGIS, le projet vise à étudier les possibilités d'implémenter des fonctions similaires dans QGIS, notamment via les points d'extension en Python comme `QgsPyRasterFunctionRenderer`. Le plugin développé à cet effet permet d'exécuter du traitement raster à l'aide de bibliothèques scientifiques comme NumPy ou GDAL, tout en s'intégrant à l'interface utilisateur de QGIS.

Cette approche soulève plusieurs enjeux : l'intégration dans la chaîne de rendu, le contrôle de la performance, et la compatibilité avec différents formats et couches.

1.2 Présentation du Projet

Le projet **Render QGIS** s'inscrit dans une volonté d'étendre les capacités de rendu raster du logiciel QGIS. Il repose sur deux axes complémentaires : d'une part, une étude du moteur de rendu natif en C++ et la possibilité de l'étendre via des renderers personnalisés ; d'autre part, la mise en œuvre d'un plugin Python permettant d'implémenter de nouvelles fonctions raster de manière flexible, sans modifier le cœur du logiciel.

Ce travail a été conduit dans un cadre exploratoire, en lien avec les besoins exprimés par l'Institut national de l'information géographique et forestière (IGN), autour d'un cas d'usage concret : la génération d'ombres portées (hillshade) sur des MNT, et le calcul d'indices spectraux comme le NDVI.

1.3 Objectifs

Le projet poursuit plusieurs objectifs complémentaires :

- **Étudier** le fonctionnement interne du moteur de rendu raster de QGIS (chaîne d'appel, classes clés, structure modulaire) ;
- **Documenter** les points d'extension Python et le cycle de rendu dans le code source (classes `QgsMapRendererJob`, `QgsRasterRenderer`, etc.) ;
- **Développer** une preuve de concept (POC) permettant de calculer et afficher des couches raster générées à la volée (ombrage, NDVI) ;
- **Évaluer** les performances de cette approche Python, notamment les coûts liés aux conversions entre `QgsRasterBlock` et `NumPyArray` ;
- **Proposer** une interface utilisateur simple pour paramétrer les traitements et les exporter comme fichiers raster.

1.4 Organisation du travail

Pour structurer les tâches et planifier les différentes étapes du projet, nous avons utilisé Google Sheets afin de construire :

- Un **diagramme de Gantt**, représentant les phases du projet ;

— Un **tableau de répartition des tâches**, précisant les responsabilités de chaque membre de l'équipe.

Les figures 1 et 2 ci-dessous en donnent un aperçu.

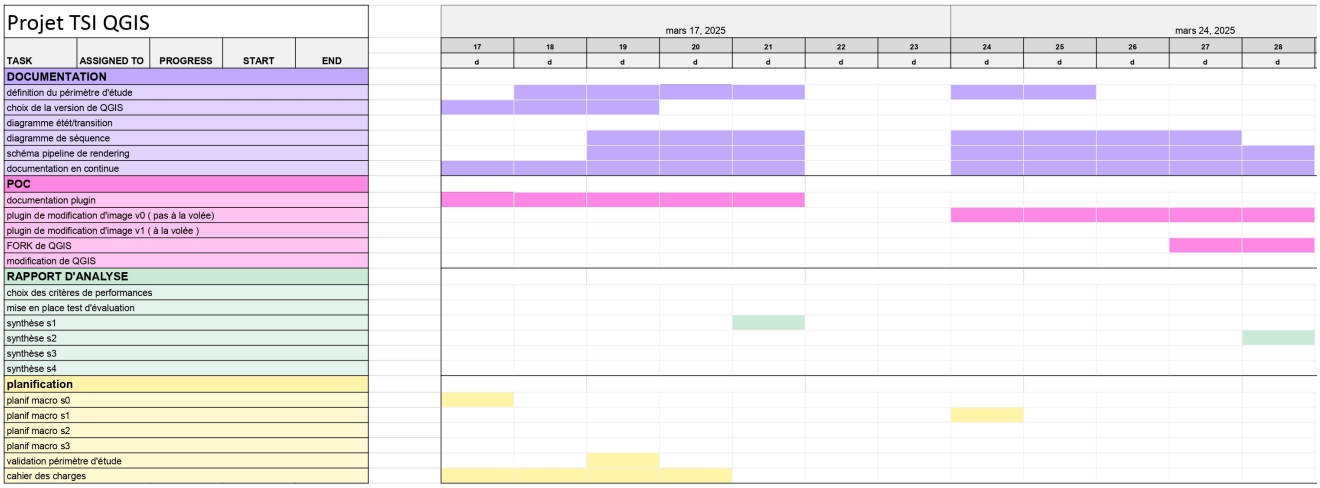

FIGURE 1 – Diagramme de Gantt du projet

Tâche	Priorité	Antonin	Qiuling	JB	Antoine	État	Date de début	Date de fin	Livrable	Notes
plannification macro v0	P0	☐	☐	☑	☐	Terminé	14/03/2025	17/03/2025	Gant Plannification	à retravailler ultérieurement
plannification macro v1	P1	☐	☐	☑	☐	Terminé	24/03/2025	24/03/2025	Gant Plannification	intégrer production qgspluginrenderer
plannin journalier	P0	☑	☑	☑	☑	Terminé	19/03/2025	19/03/2025	Fichier	*Notes*
mise en place doc de suivi	P0	☐	☐	☑	☐	Terminé	13/03/2025	14/04/2025	Fichier	*Notes*
complétion cahier des charges	P3	☐	☐	☑	☐	Terminé	19/03/2025	01/04/2025	Cahier des charges	à préciser compte-tenu de la réunion du 20/03, bien redéfinir le sujet, à relire en groupe et à faire relire par nicolas David
définition Objectifs	P3	☑	☑	☑	☑	Terminé	19/03/2025	19/03/2025	Tâches	*Notes*
applique doxygen aux classes à étudier	P2	☐	☐	☑	☐	Terminé	18/03/2025	18/03/2025	Fichier	peu utile in fine, extraire les résultats pour les classes choisies lors de la réunion du 20/03
définir liste des classes QGIS à étudier	P3	☑	☐	☑	☑	Terminé	17/03/2025	20/03/2025	Fichier	qgsrasterinterface et classes filles
choix de la version de QGIS	P2	☑	☑	☑	☑	Terminé	17/03/2025	19/03/2025	Fichier	3.40, faudrait trouver un moyen de justifier le choix
choix des critères de performances	P0	☐	☐	☐	☐	non assigné	*dd/mm/yyyy*	*dd/mm/yyyy*	Fichier	*Notes*
exploration libre DOC / CodeSource QGIS	P3	☑	☑	☑	☑	Terminé	14/03/2025	17/03/2025	Fichier	*Notes*
doc plugin binding c++		☐	☐	☐	☑	Terminé	17/03/2025	*dd/mm/yyyy*	Fichier	un petit livrable ?
planification semaine 14/03	P0	☐	☐	☑	☐	Terminé	17/03/2025	17/03/2025	Tâches	*Notes*
Trouver/produire UML QGIS	P3	☑	☑	☑	☑	Terminé	17/03/2025	*dd/mm/yyyy*	renderULMdiagram.drawio	*Notes*
diagramme état-transition v0	P2	☐	☐	☐	☐	non assigné	*dd/mm/yyyy*	*dd/mm/yyyy*	Fichier	*Notes*
étude détaillé de rasterpipe	P2	☑	☐	☐	☐	Terminé	21/03/2025	*dd/mm/yyyy*	Fichier	à répartir
étude détaillé de layerrenderer	P3	☑	☐	☐	☐	Terminé	21/03/2025	*dd/mm/yyyy*	Fichier	l'objectif est d'étudier en détail les renderer pour produire un qgspluginrenderer

FIGURE 2 – Répartition des tâches dans l'équipe

Suivi avec le commanditaire

Des réunions ont été organisées tout au long du projet, aux dates suivantes : **14 mars, 21 mars, 28 mars et 9 avril 2025**. Ces échanges ont été essentiels pour valider les orientations techniques et s'assurer de l'adéquation des livrables avec les besoins réels.

2 Étapes clefs du rendering QGIS

Le processus de modification et d'affichage du rendu dans QGIS est un ensemble complexe faisant appel à différents objets et classes de QGIS ; comprendre son fonctionnement nécessite de le découper en étapes. Nous en avons distingué trois : l'appel du processus, son orchestration, et son exécution.

2.1 Démarrage et instanciation du processus

Le rendu (*rendering*) d'une couche dans QGIS peut être déclenché par une gamme de signaux (liés aux actions de l'utilisateur), qui appellent tous la fonction `refresh()` de la classe `QgsMapCanvas`.

Plus précisément, cette fonction modifie un objet `QTimer` nommé `mRefreshTimer`, qui décompte une durée avant d'appeler la fonction `refreshMap()`. Ce système permet de regrouper les modifications de l'utilisateur avant de lancer le rendu.

Cette cascade d'appels est représentée dans la Figure 5.

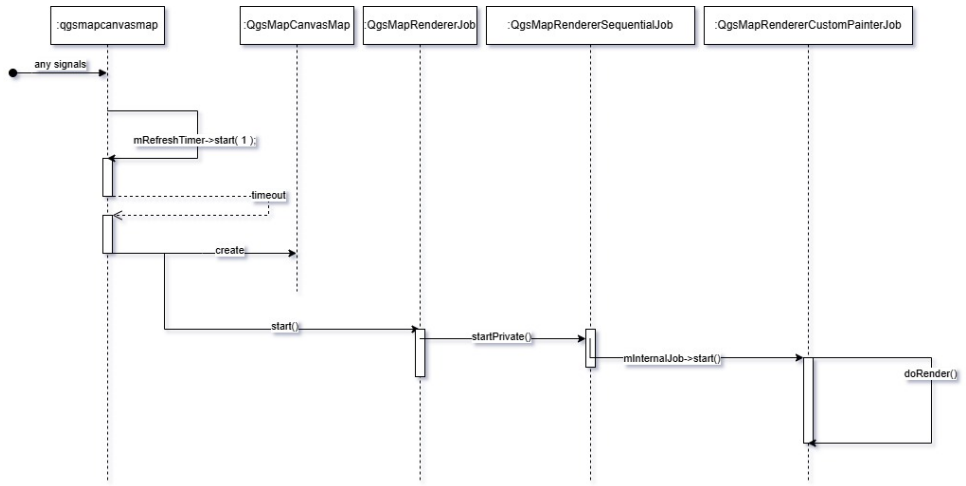

FIGURE 3 – Séquence d'appels pour le rendu d'une couche

Cette séquence se termine par l'exécution de la fonction `doRender()` de la classe `QgsMapRendererCustomPainterJob`.

Pour mieux comprendre ce processus, notons qu'un objet de la classe `QgsMapCanvas` peut contenir plusieurs couches (*layers*). Chacune de ces couches est associée à un objet de la classe `QgsMapRendererJob`, qui représente un processus de rendu. Ce processus peut être implémenté de manière séquentielle (`QgsMapRendererSequentialJob`) ou parallèle (`QgsMapRendererParallelJob`).

La fonction `refreshMap()` initialise deux éléments nécessaires au rendu :

— un objet `renderSettings` de type `QgsMapSettings`, qui contient le contexte et les paramètres du rendu ;

— un objet `QgsMapRendererJob` (concrètement, une des deux classes filles mentionnées ci-dessus).

Elle lance ensuite le processus de rendu via la méthode `start()` de `QgsMapRendererJob`.

2.2 Configuration, orchestration et lancement des tâches de rendu

Il est utile de comprendre la structure des classes qui organisent les tâches de rendu. La Figure 4 ci-dessous présente un diagramme de classes simplifié représentant cette organisation :

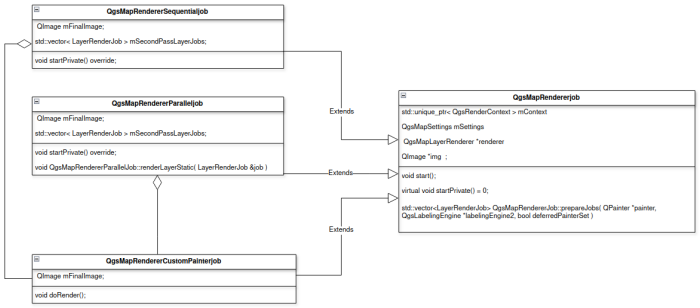

FIGURE 4 – Structure simplifiée des classes de rendu

Le processus de rendu peut être abstaitement découpé en trois grandes étapes :

1. **Contrôle et récupération des informations de contexte** : récupération de la configuration actuelle de l'application via un objet `QgsMapSettings` (zone géographique, échelle, transformation CRS, résolution, liste des couches visibles, etc.).
2. **Calcul et initialisation des tâches de rendu** : cette étape consiste à préparer l'ensemble des sous-tâches nécessaires au rendu (couches raster et vecteur, labels, options de shading ou de masquage), mais sans les exécuter.
3. **Exécution proprement dite du rendu** : lancement des tâches dans un ou plusieurs threads/processus, soit de manière séquentielle soit en parallèle, en fonction du mode de rendu choisi. Toujours à l'aide d'un objet QPainter.

2.3 Implémentation concréte dans le cas du rendu séquentiel

Conformémment aux trois étapes décrites precedemment :

1. Initialisation du rendu et configuration du job La méthode `start()` d'un objet `QgsMapRendererQImageJob` (ou d'un de ses dérivés) constitue le point d'entrée de l'API publique de rendu. Elle vérifie la validité du `QgsMapSettings` fourni, puis délègue l'exécution à la méthode virtuelle pure `startPrivate()` implémentée dans une sous-classe.

Dans le cas du rendu séquentiel, un objet interne de type `QgsMapRendererCustomPainterJob` est instancié. Celui-ci reçoit le `QPainter` cible, utilisé pour dessiner le rendu dans un `QWidget` (Canvas graphique basé sur Qt), ainsi que les paramètres de rendu globaux. Le rendu est ensuite déclenché par l'appel à la méthode `start()` de l'objet interne.

2. Préparation des tâches de rendu (jobs) Cette étape correspond à la phase d'orchestration des sous-jobs :

— Pour chaque couche visible, est génèré un `LayerRenderJob` correspondant. On notera que le LayerRenderJob désigne soit une structure, soit une classe et que le code dans le définissant se situe

— Chaque `LayerRenderJob` encapsule :
  — L'identifiant de la couche, son étendue projetée, ses paramètres de rendu (opacité, mode de fusion, CRS).
  — Un contexte de rendu `QgsRenderContext`, préparé à partir des `QgsMapSettings`.
  — Éventuellement, un résultat en cache si la couche n'a pas changé.
  — Un renderer spécifique généré via `createMapRenderer()`.
— Des tâches supplémentaires sont définis pour :
  — Le rendu des labels (texte), via un moteur dédié (`labelEngine`).
  — Le rendu optionnel de type shading (ombrage par élévation).
  — Une éventuelle seconde passe de rendu pour appliquer des effets comme le masquage sélectif.

L'ensemble des tâches est organisé dans un vecteur de type `std::vector<LayerRenderJob>`.

3. Exécution du rendu L'exécution des jobs est assurée par la méthode `QgsMapRendererCustomPain` appelée de façon synchrone ou asynchrone selon le contexte (affichage GUI ou export). Cette fonction réalise les actions suivantes :

1. Initialise les images de rendu pour chaque couche.
2. Pour chaque `LayerRenderJob`, appelle la méthode `render()`, qui délègue au renderer de la couche (par exemple `QgsRasterRenderer` ou `QgsVectorLayerRenderer`).
3. Applique si besoin le rendu global d'ombrage par élévation.
4. Lance le rendu des labels, s'ils sont activés.
5. Effectue une seconde passe de rendu dans le cas de certaines options (masquage sélectif).

En mode parallèle, plusieurs processus de rendu peuvent être exécutés simultanément. Les résultats intermédiaires (images rasterisées de chaque couche) sont ensuite fusionnés dans le thread principal pour produire l'image finale.

2.4 Rendu raster dans QGIS

Le rendu des couches raster repose sur un système modulaire orchestré par la classe `QgsRasterLayerRenderer`, qui hérite de `QgsMapLayerRenderer`. Cette classe est responsable de piloter le rendu d'une couche raster en s'appuyant sur un pipeline de traitement (`QgsRasterPipe`) composé de multiples modules appelés `QgsRasterInterface`.

2.4.1 Types de raster renderers

Plusieurs types de renderers peuvent être utilisés selon la nature des données et les besoins d'affichage :

— `QgsSingleBandGrayRenderer`
— `QgsSingleBandPseudoColorRenderer`
— `QgsSingleBandColorDataRenderer`
— `QgsMultiBandColorRenderer`
— `QgsPalettedRasterRenderer`
— `QgsRasterContourRenderer`
— `QgsHillshadeRenderer`

Tous ces renderers héritent de la classe de base `QgsRasterRenderer` et sont insérés dans le pipeline de rendu raster comme des modules intermédiaires transformant les données.

2.4.2 Séquence de rendu

Le processus de rendu suit une séquence bien définie :

1. Le rendu est initié depuis le job principal via `job.renderer->render()`.
2. La méthode abstraite `QgsMapLayerRenderer::render()` est implémentée pour les rasters dans `QgsRasterLayerRenderer::render()`.
3. Ce dernier initialise un `QgsRasterIterator` à partir du `QgsRasterPipe` et crée un `QgsRasterDrawer` pour gérer le rendu.
4. `QgsRasterDrawer::draw()` itère ensuite sur des blocs d'image via l'iterator, et les dessine un par un dans un `QPainter`.

Le travail sur blocs (tiling) permet de gérer efficacement de grandes images raster sans consommer trop de mémoire.

2.4.3 Pipeline de rendu raster

Le `QgsRasterPipe` contient une liste chaînée de modules `QgsRasterInterface`. Chaque interface traite un bloc raster en demandant son équivalent à l'interface précédente. Cette chaîne de traitement est dite *lazy* : le calcul est déclenché uniquement lorsque nécessaire.

Voici l'ordre typique des composants du pipeline :

1. **QgsRasterDataProvider** : accède aux données brutes (source).
2. **QgsRasterRenderer** : applique les règles de style (couleurs, échelles...).
3. **QgsBrightnessContrastFilter** (optionnel) : ajuste la luminosité/contraste.
4. **QgsHueSaturationFilter** (optionnel) : ajuste la teinte et la saturation.
5. **QgsRasterResampleFilter** (optionnel) : applique un ré-échantillonnage.
6. **QgsRasterProjector** (optionnel) : effectue une reprojection géographique.

L'enchaînement de ces interfaces se fait de façon via des appels à la méthode `block()` de chaque interface. Il est important de noter que les qgsRasterBlock ont des structures assimilables à des array numpy, ce qui permet leur manipulation simple dans les plugin.

2.4.4 Traitement par bloc

L'objet `QgsRasterIterator` divise l'image à dessiner en blocs (ou tuiles), ce qui permet :

— une meilleure gestion mémoire ;
— une possibilité de rendu partiel ou progressif ;
— une optimisation du calcul parallèle.

Chaque bloc est récupéré via l'appel :

```
mIterator->readNextRasterPart(bandNumber, nCols, nRows, block, topLeftCol, topLeftRow
```

Et ce bloc traverse ensuite le pipeline en cascade, depuis le `QgsRasterDataProvider` jusqu'à l'image affichée.

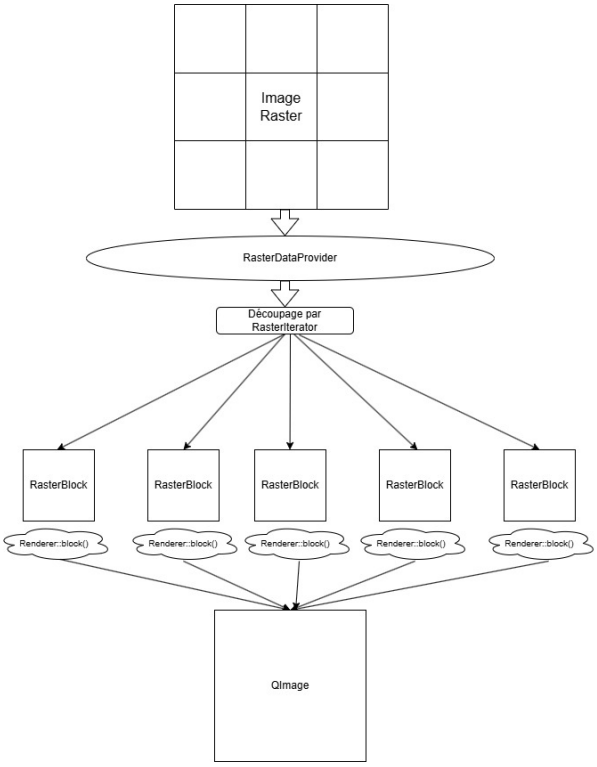

FIGURE 5 – Schéma simplifié du traitement par bloc

2.4.5 Résumé

En résumé, le rendu raster dans QGIS suit une architecture orientée pipeline et modulable.

Pipeline type :

`Layer → Extraction (extent) → Renderer → Filters (Hue/Brightness) → Resample → Reprojection → QImage`

La fonction à modifier pour implementer un Renderer est `block()` :

```
QgsRasterBlock *block(
   int bandNo,
   const QgsRectangle &extent,
   int width,
   int height,
   QgsRasterBlockFeedback *feedback = nullptr ) override = 0 SIP_FACTORY;
```

En effet c'est cette fonction qui fournit la Qimage affichée.

3 Option modification du code source

3.1 Présentation de la solution

La première option consiste à modifier directement le code source de QGIS afin d'implémenter un renderer personnalisé. Cette approche peut sembler lourde de prime abord, mais elle s'avère extrêmement puissante pour ceux qui souhaitent une intégration complète de leur solution au sein du cœur du logiciel.

Dans cette section, nous allons nous concentrer sur les principes fondamentaux à suivre pour appliquer cette méthode de manière structurée et la plus simple possible. Nous ne traiterons pas ici des étapes de contribution formelle au projet (comme la création de branches Git, la rédaction de documentation, les pull requests ou les tests automatisés), qui relèvent d'un autre cadre.

Modifier directement le code source de QGIS offre un certain nombre d'avantages. Le principal réside dans la liberté qu'elle donne : en agissant au niveau du cœur applicatif, il devient possible d'exploiter l'ensemble des mécanismes internes de QGIS sans limitation. Le renderer que l'on implémente peut ainsi bénéficier d'une intégration native dans l'interface graphique, dans les traitements, ou encore dans les exports. C'est également une façon de mieux comprendre le fonctionnement de QGIS, en se confrontant à son architecture modulaire et à ses conventions de développement.

Cependant, cette solution implique un certain nombre de contraintes importantes. Travailler sur le code de QGIS signifie manipuler un environnement complexe, basé sur C++ avec Qt, et s'appuyant sur de nombreuses bibliothèques comme GDAL ou PROJ. Il est donc nécessaire de se familiariser avec l'organisation du projet, la logique des plugins internes, ainsi que le cycle de compilation complet. Cela peut représenter un investissement non négligeable, d'autant plus que chaque modification nécessite souvent de recompiler tout ou partie de QGIS, ce qui peut ralentir la phase de test et d'itération.

En somme, cette approche est exigeante, mais elle permet une maîtrise fine du comportement du renderer et une intégration cohérente avec les autres composants de QGIS. Elle s'adresse avant tout à ceux qui souhaitent étendre les capacités de rendu de QGIS de façon durable, voire contribuer en retour au projet en proposant une fonctionnalité nouvelle.

3.2 De quelle manière éditer le code source C++ de QGIS pour créer un Renderer ?

3.2.1 Compréhension des dépendances

Les interfaces de base sur lesquelles reposent les renderers personnalisés dans QGIS sont principalement les classes `QgsRasterInterface` et `QgsRasterRenderer`. Ces deux éléments jouent un rôle central dans l'architecture du rendu raster et doivent être bien compris avant d'entamer toute modification.

`QgsRasterInterface` : le socle commun

`QgsRasterInterface` est une classe abstraite, ce qui signifie qu'elle ne peut pas être instanciée directement. En C++, une classe abstraite contient au moins une méthode pure virtuelle (notée avec `= 0`) que les classes dérivées devront obligatoirement implémenter. Cette interface pose donc les bases du fonctionnement d'un renderer en définissant un

ensemble de méthodes clés comme `block()` ou encore `dataType()`, qui permettent de récupérer les données raster à un niveau bas.

Elle agit comme un patron générique : tous les types de renderers, qu'ils soient standards ou personnalisés, doivent l'utiliser comme point d'ancrage. Son rôle est fondamental pour structurer la chaîne de traitement des données raster, avant même la phase de visualisation.

`QgsRasterRenderer` : spécialisation pour l'affichage

`QgsRasterRenderer` hérite directement de `QgsRasterInterface`, et constitue une spécialisation dédiée à la visualisation. Elle est elle aussi abstraite, mais elle se situe un cran plus haut dans l'architecture, en se concentrant uniquement sur le rendu à l'écran.

Concrètement, chaque renderer qui apparaît dans la liste de rendu d'une couche raster dans QGIS (comme le renderer "Paletted", "Singleband pseudocolor", ou un renderer NDVI personnalisé) hérite de cette classe. Il devra alors implémenter, entre autres, la méthode `block()` qui est le cœur du processus de génération des tuiles raster à afficher dans l'interface graphique.

Il est recommandé, pour comprendre le comportement attendu, d'examiner les renderers existants dans le code source de QGIS. Ces derniers constituent une excellente base pour construire une nouvelle implémentation, notamment pour gérer les conversions de types, la normalisation des valeurs ou encore la gestion des palettes de couleurs.

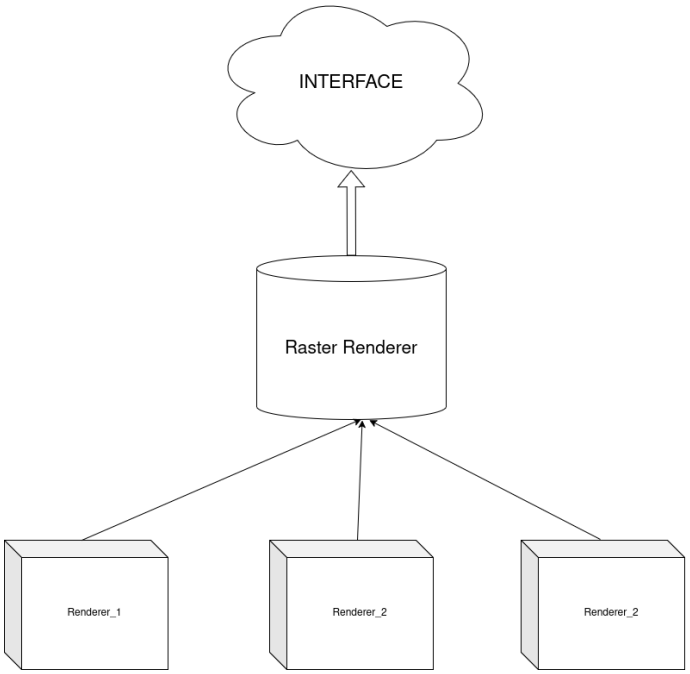

FIGURE 6 – relation de dépendances des renderers

Création d'un widget associé au renderer

Pour permettre à l'utilisateur de configurer un renderer personnalisé depuis l'interface de QGIS, il est nécessaire de créer un widget associé. Dans le cas d'un renderer NDVI, par exemple, on définit une classe héritant de `QgsRasterRendererWidget`, qui correspond à la base des interfaces graphiques de configuration des renderers.

Cette classe personnalisée va être liée à un fichier `.ui` (fichier Qt Designer) contenant la

description de l'interface visuelle du widget : sélection de bandes, paramètres numériques, choix de palette, etc. C'est ce widget qui sera affiché dans le panneau de symbologie d'une couche raster lorsque l'utilisateur sélectionne le type de rendu correspondant.

Le fichier `.ui` devient ainsi le point d'entrée visuel de notre renderer dans l'interface de QGIS. Il permet une configuration intuitive par l'utilisateur, tout en servant de passerelle entre l'interface et la logique C++ du renderer.

Intégration dans le panneau de symbologie

L'un des éléments clés pour faire apparaître notre renderer dans la liste des types disponibles dans la symbologie est la classe `QgsRendererRasterPropertiesWidget`. Cette classe agit comme une fabrique (factory) ou patron de conception de widgets de configuration. C'est dans cette classe que chaque type de renderer est enregistré via une fonction `create()`, propre à son widget. Nous reviendrons plus en détail sur le rôle de cette fabrique dans la sous-partie suivante, mais il est essentiel de savoir qu'elle joue un rôle dans la liaison entre notre renderer C++, son interface graphique, et l'environnement général de QGIS.

3.2.2 Ce qu'il faut mettre en place

L'implémentation d'un nouveau renderer personnalisé dans QGIS s'appuie sur un schéma de classe représentant l'architecture d'un renderer de type `HillshadeRenderer`. Ce schéma met en évidence les composants communs à tous les renderers (surlignés en jaune) ainsi que la méthode `block()`, spécifique à chaque renderer, en vert. Cette dernière constitue le cœur de la logique du renderer : c'est elle qui renvoie les valeurs des pixels affichés à l'écran.

Les méthodes suivantes sont obligatoires pour tout renderer, car elles garantissent une intégration complète avec le système de rendu de QGIS :

clone()

- Crée une copie indépendante du renderer.
- Utilisé lors de l'édition d'un style de rendu pour permettre :
  - l'annulation des modifications,
  - la duplication de styles entre couches,
  - la préservation de l'état original.
- Mécanisme de sécurité évitant de modifier directement le renderer actif tant que les changements ne sont pas confirmés.

flags()

- Méthode virtuelle obligatoire définie dans la classe de base `QgsRasterRenderer`.
- Informe QGIS des capacités spécifiques du renderer (ex. `Support8BitRgbOutput` pour NDVI).
- Permet au moteur de rendu d'optimiser le pipeline en fonction des fonctionnalités offertes.
- Garantit l'intégration correcte du renderer dans le système global de rendu.

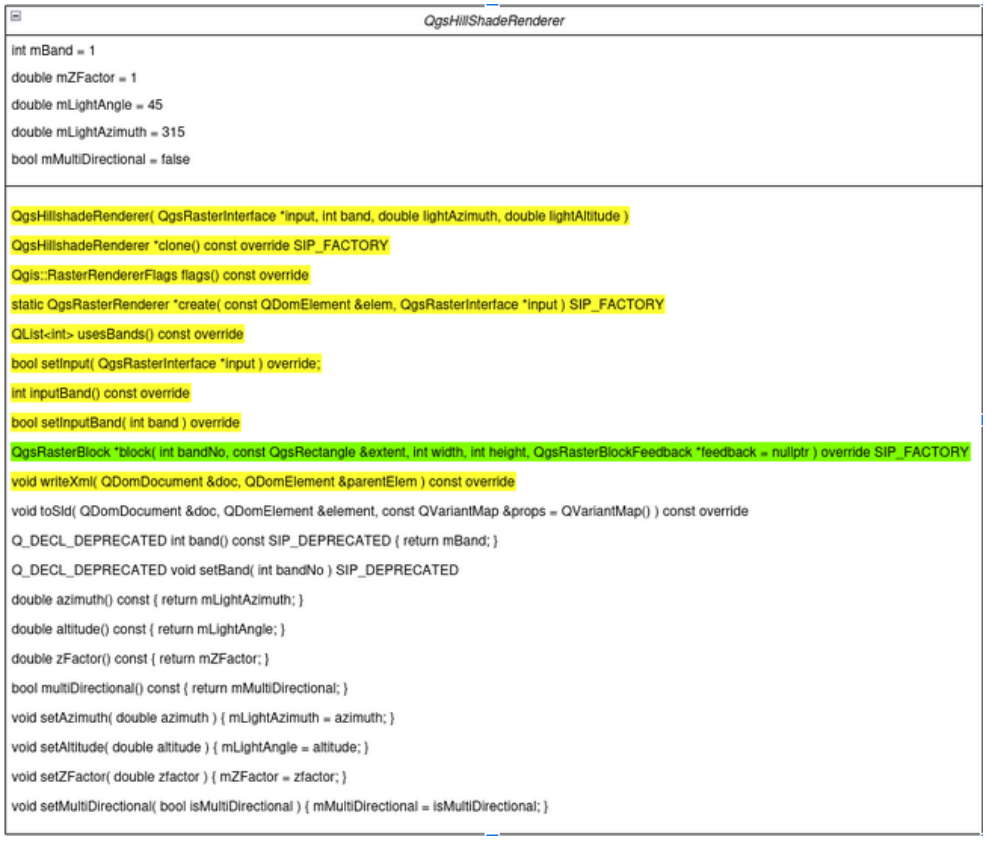

FIGURE 7 – classe Ombrage

create(const QDomElement &elem, QgsRasterInterface *input)

- Implémente le pattern *Factory* pour recréer un renderer à partir de données XML.
- Utilisé lors du chargement d'un projet ou fichier de style.
- Enregistre le renderer dans le `QgsRasterRendererRegistry`.
- Spécifique au NDVI :
  - Extraction des paramètres (bandes NIR et RED),
  - reconstruction éventuelle du shader,
  - configuration des propriétés communes,
  - retour d'une instance configurée.

writeXml(QDomDocument &doc, QDomElement &parentElem)

- Sauvegarde l'état du renderer en XML pour persistance dans un projet QGIS.
- Fonctionne de pair avec `create()` pour former un cycle complet :
  - `writeXml` : objet → XML,
  - `create` : XML → objet.
- Dans le renderer NDVI :
  - Création d'un élément `<rasterrenderer>` avec type = `ndvi`,
  - sauvegarde des bandes NIR et RED comme attributs XML,
  - sérialisation du shader et des min/max si présents,
  - appel à `writeCommonProperties()` pour les champs communs.

usesBands()

— Indique les bandes raster utilisées par le renderer.
— Essentiel pour :
  — l'optimisation (lecture sélective des bandes),
  — la détection d'erreurs (bandes manquantes),
  — l'interface utilisateur (affichage des dépendances),
  — la validation et la compatibilité avec d'autres rasters.
— Exemple pour le NDVI : retourne les indices des bandes NIR et RED.

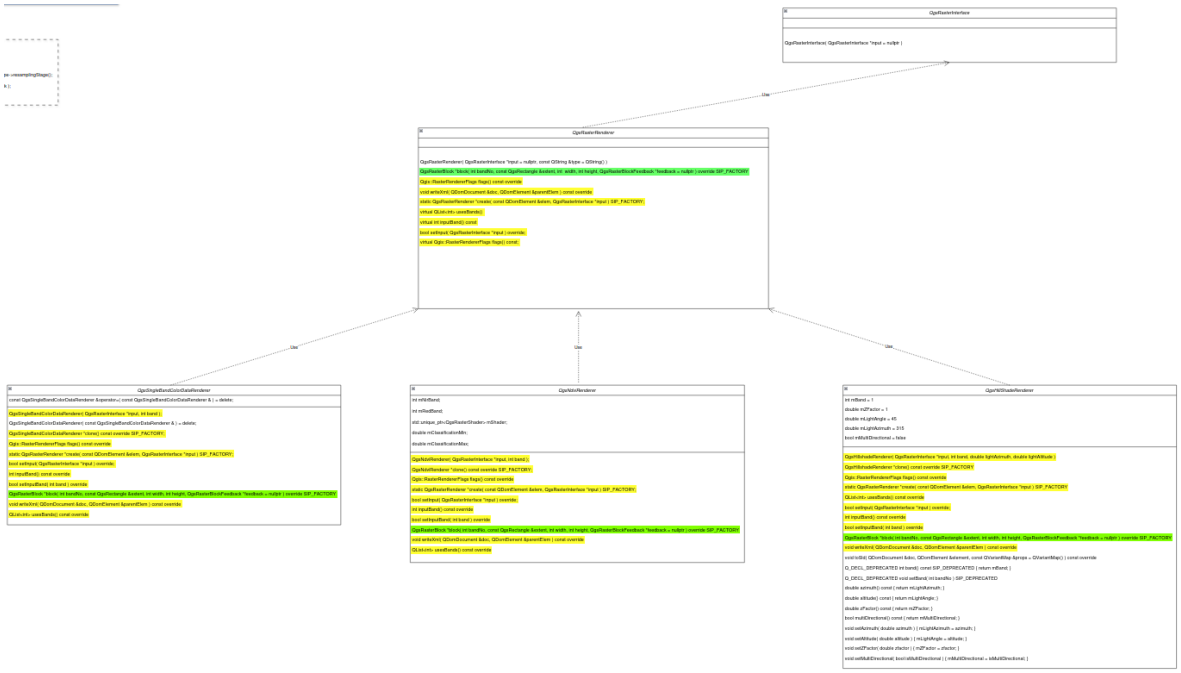

FIGURE 8 – classes renderer completes

Méthode spécifique à chaque renderer – `block()`

`block()` est la méthode cœur du renderer : elle calcule les valeurs des pixels à afficher à l'écran selon l'algorithme spécifique du renderer. Elle traite les blocs de données raster et renvoie un objet de type `QgsRasterBlock` contenant les valeurs d'intensité ou de couleur pour chaque pixel.

Implémentation UI et enregistrement dans QGIS

Une fois la classe `QgsNdviRenderer` complétée, il est nécessaire de l'intégrer à l'interface de QGIS :

— Créer la classe `qgsndvirendererwidget.h` et `qgsndvirendererwidget.cpp` dans `src/gui/raster/`.
— Concevoir l'interface graphique avec QtDesigner : `qgsndvirendererwidget.ui`.
— Modifier `src/core/raster/qgsrasterrendererregistry.cpp` pour enregistrer le renderer.
— Enregistrer le widget dans `src/gui/raster/qgsrendererrasterpropertieswidget.cpp`.
— Ajouter les fichiers au `CMakeLists.txt` concerné.
— Ajouter une icône SVG (ex. `ndvi.svg`) dans `images/styleicons/`.

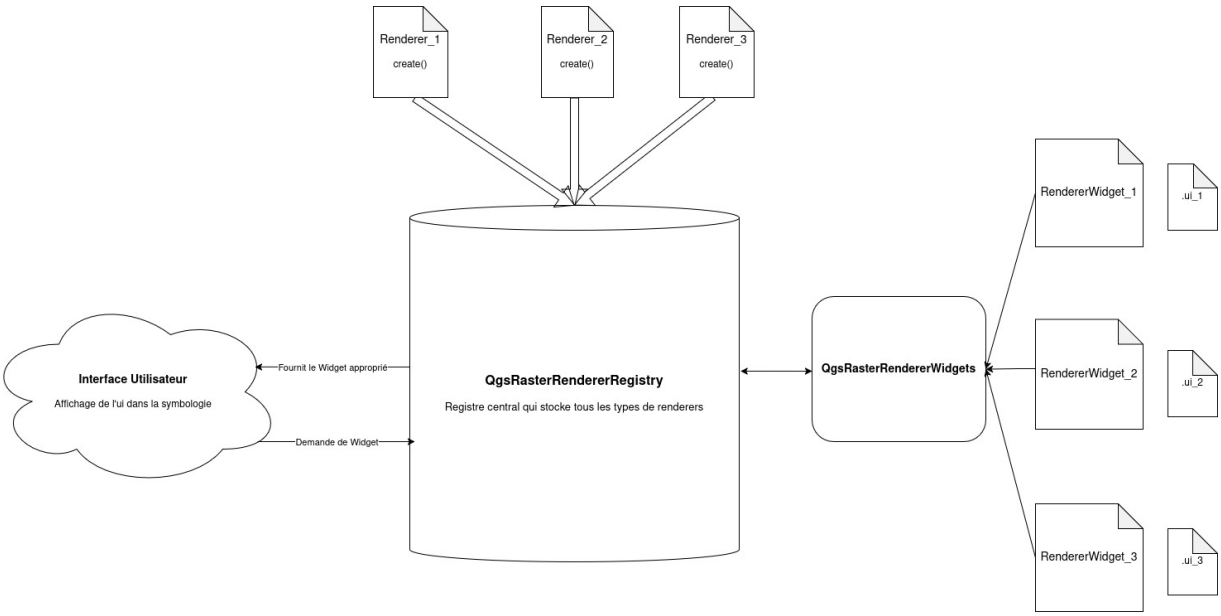

FIGURE 9 – schéma de l'intéraction entre interface et widgets

3.2.3 Explication du `SIP_FACTORY`

`SIP_FACTORY` est une annotation qu'on utilise dans le code C++ de QGIS, en particulier dans les déclarations de méthodes qui retournent des objets. Elle sert à dire au système SIP (celui qui gère les passerelles entre C++ et Python) que l'objet créé doit être pris en charge par Python — autrement dit, que c'est lui qui devra s'occuper de le détruire quand il ne sert plus.

Pourquoi c'est important ? Si on ne précise pas ça, l'objet reste orphelin : ni C++ ni Python ne sait qu'il doit s'en occuper, et on finit avec des fuites mémoire.

Dans le cas d'un renderer perso, comme le `QgsNdviRenderer` par exemple, on a plusieurs méthodes qui créent des objets dynamiquement :

— `clone()` : pour dupliquer proprement le renderer,
— `create()` : méthode statique utilisée pour l'enregistrer dans le registre des renderers,
— `block()` : retourne un `QgsRasterBlock` utilisé pour le rendu.

Toutes ces méthodes doivent être annotées avec `SIP_FACTORY` si on veut que le renderer fonctionne correctement quand il est utilisé depuis un script ou un plugin Python.

Le schéma montre que SIP agit comme un pont entre les mondes C++ et Python. SIP permet de transférer la responsabilité de gestion mémoire au bon endroit — Python — dans les cas où des objets sont instanciés en C++ mais utilisés en Python (comme un renderer ou un raster block). Sans cette annotation, ces objets resteraient "orphelins", et donc non libérés correctement.

En pratique, cette annotation est prise en compte au moment de la génération des bindings Python (via `sipify`). C'est un détail technique, mais essentiel pour que tout ce qui est créé côté C++ soit bien géré côté Python. Sans ça, pas d'intégration propre dans l'écosystème QGIS.

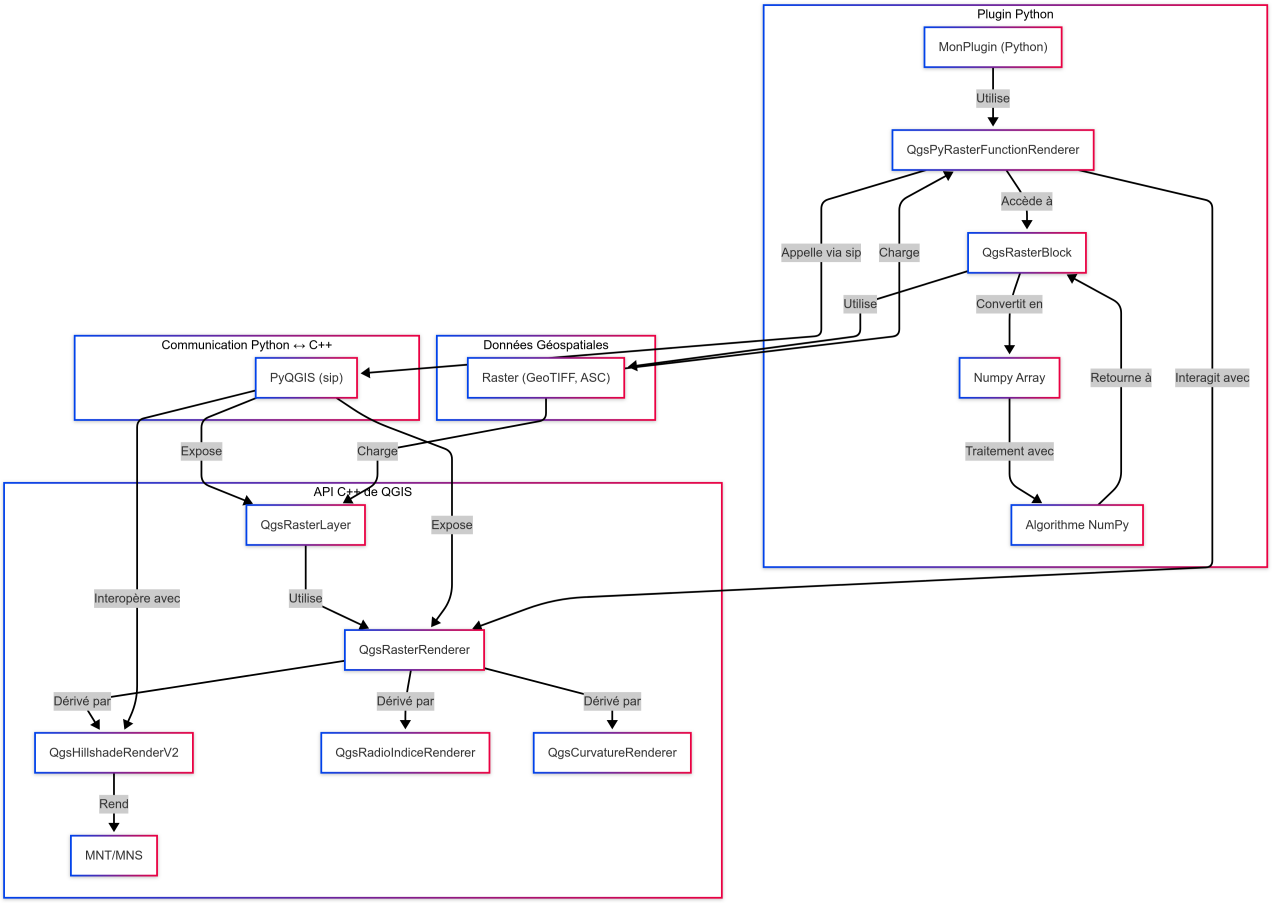

FIGURE 10 – Schéma d'intéraction SIP

4 Présentation du plugin Render

Dans le cadre de ce projet, nous avons développé un plugin QGIS intitulé **Render**, dédié au traitement d'images raster issues de la télédétection. Ce plugin s'inscrit dans une logique d'intégration directe de traitements spatiaux au sein de QGIS, sans passer par des outils externes comme Python en ligne de commande ou des logiciels propriétaires. Il s'adresse à des utilisateurs ayant besoin d'effectuer rapidement des opérations courantes de traitement d'images satellites ou aéroportées, notamment dans les domaines de l'agriculture, de la gestion des ressources naturelles ou encore de la cartographie environnementale.

4.1 Fonctionnalités principales

Le plugin *Render* propose deux fonctionnalités principales :

— **Calcul de l'indice de végétation NDVI (Normalized Difference Vegetation Index)**
Le NDVI est un indicateur classique en télédétection permettant d'évaluer la vigueur de la végétation. Il est calculé à partir de deux bandes spectrales : l'infrarouge proche (PIR) et le rouge (Red), selon la formule :

$$\text{NDVI} = \frac{\text{PIR} - \text{Rouge}}{\text{PIR} + \text{Rouge}}$$

L'utilisateur sélectionne les deux bandes raster depuis l'interface, et le traitement est effectué pixel par pixel grâce à l'utilisation de la bibliothèque `NumPy`, qui permet une vectorisation efficace des calculs. La gestion des erreurs de division (zéros, NaN) est intégrée automatiquement, assurant la robustesse du résultat. Le NDVI est sauvegardé comme fichier `.tif` géoréférencé, puis automatiquement affiché dans QGIS.

— **Génération d'un ombrage (Hillshade)**
Cette fonctionnalité permet de créer une représentation ombrée d'un Modèle Numérique de Terrain (MNT), afin d'améliorer la lisibilité visuelle de la topographie. Le calcul repose sur l'analyse du gradient d'altitude par convolution avec des noyaux de Sobel, pour obtenir la *pente* et l'*orientation* (aspect) de chaque pixel :

$$\text{slope} = \arctan\left(\sqrt{\left(\frac{\partial z}{\partial x}\right)^2 + \left(\frac{\partial z}{\partial y}\right)^2}\right) \quad , \quad \text{aspect} = \arctan 2\left(\frac{\partial z}{\partial y}, \frac{\partial z}{\partial x}\right)$$

L'utilisateur peut spécifier l'altitude et l'azimut du soleil. L'ombrage est alors généré selon la formule suivante :

$$\text{Hillshade} = 255 \times (\cos(\theta)\cos(\text{slope}) + \sin(\theta)\sin(\text{slope})\cos(\phi - \text{aspect}))$$

Où θ est l'angle zénithal (90° moins l'altitude solaire) et ϕ l'azimut du Soleil.

L'**altitude solaire** correspond à la hauteur de la source lumineuse au-dessus de l'horizon, avec une valeur comprise entre 0° (soleil rasant) et 90° (soleil au

zénith). L'**azimut du Soleil** est mesuré en degrés dans le sens horaire à partir du nord géographique, de 0° à 360°.

L'ombrage est un effet visuel simulé basé sur une lumière fictive. Il ne reflète pas la position réelle du soleil au moment de l'acquisition, mais vise à améliorer la perception du relief.

Nous avons choisi une valeur par défaut de **315° pour l'azimut du Soleil**, correspondant à une lumière venant du *nord-ouest*. Ce paramètre est couramment utilisé en cartographie (notamment dans ArcGIS ou QGIS) car il produit une ombre vers le sud-est, ce qui donne une impression de relief tridimensionnel plus naturelle pour les utilisateurs habitués à lire les cartes en éclairage nord-ouest.

De même, une **altitude solaire par défaut de** 45° est un compromis visuel permettant de simuler une lumière suffisamment haute pour éviter des ombres excessives, tout en conservant un effet de relief marqué. Cette configuration est adaptée à la majorité des MNT et assure une lisibilité optimale.

Le résultat est enregistré en tant que raster GeoTIFF avec toutes les métadonnées spatiales (résolution, projection, emprise), et directement ajouté au canevas QGIS.

4.2 Traitement raster avec NumPy et QGIS

Le plugin repose sur l'API de QGIS pour accéder aux données raster via l'objet `QgsRasterDataProvider`. La méthode `block()` permet d'extraire un `QgsRasterBlock`, représentant les valeurs d'une bande raster sur toute l'étendue de l'image. Ce bloc est converti en tableau NumPy pour bénéficier des fonctionnalités de calcul scientifique haute performance :

```
block = provider.block(1, extent, cols, rows)
array = np.array([
    [block.value(i, j) for i in range(block.width())]
    for j in range(block.height())
])
```

Une fois traitées (NDVI, ombrage...), ces matrices NumPy sont réécrites sur disque via la bibliothèque `GDAL`, qui permet de générer des fichiers GeoTIFF valides, géoréférencés et exploitables par QGIS.

4.3 Interface utilisateur

L'interface graphique du plugin (les figures 11 et 12) a été développée avec *Qt Designer* et intégrée à QGIS via `PyQt5`. Elle a été pensée pour être simple et pédagogique. Elle comprend :

- Une combo box permettant de sélectionner une couche raster, afin d'afficher ses dimensions et statistiques (min, max, moyenne)
- Deux menus déroulants pour sélectionner les bandes PIR et Rouge à utiliser dans le calcul du NDVI
- Des champs numériques pour paramétrer l'azimut et l'altitude solaire pour l'ombrage
- Deux champs de sélection de fichiers de sortie pour sauvegarder les résultats NDVI et Hillshade au format `.tif`

— Un bouton de validation pour exécuter les traitements et afficher les résultats automatiquement dans QGIS

4.4 Dépendances techniques

Le plugin a été développé pour **QGIS 3.40 Bratislava**, version Long Term Release, en utilisant les bibliothèques suivantes :

— `NumPy` pour la manipulation efficace des tableaux de données raster
— `SciPy` pour le calcul de gradient (convolution 2D avec noyaux de Sobel)
— `GDAL / osgeo` pour la lecture et l'écriture des fichiers GeoTIFF
— `PyQt5` pour la conception de l'interface graphique du plugin
— `QGIS API` pour l'accès aux couches raster, au canvas, et à la logique de projet

Le plugin est compatible avec les systèmes **Linux, Windows et macOS**, à condition de disposer d'une installation de QGIS correctement configurée.

4.5 Performance et limitations

Le plugin exécute les traitements directement en mémoire via NumPy. Pour cela, les valeurs raster sont extraites sous forme de blocs à l'aide de l'API de QGIS, puis converties en tableaux NumPy. Si cette approche permet une grande souplesse dans le traitement, elle présente également certaines limites en termes de performance.

En effet, les traitements sont réalisés hors du pipeline de rendu natif de QGIS. Cela signifie que les résultats ne sont pas affichés dynamiquement à l'écran (en "online rendering"), mais sauvegardés sur disque avant d'être rechargés. Pour les grandes images raster ou des résolutions élevées, cette étape peut engendrer un temps de calcul notable.

L'utilisation de Python dans la boucle de rendu pose aussi des défis de performance : les conversions entre les structures internes de QGIS (`QgsRasterBlock`) et les objets NumPy peuvent devenir coûteuses si elles sont répétées en temps réel. Une perspective future serait d'explorer l'utilisation de `QgsPyRasterFunctionRenderer`, qui permet d'intégrer le traitement directement dans la chaîne de rendu.

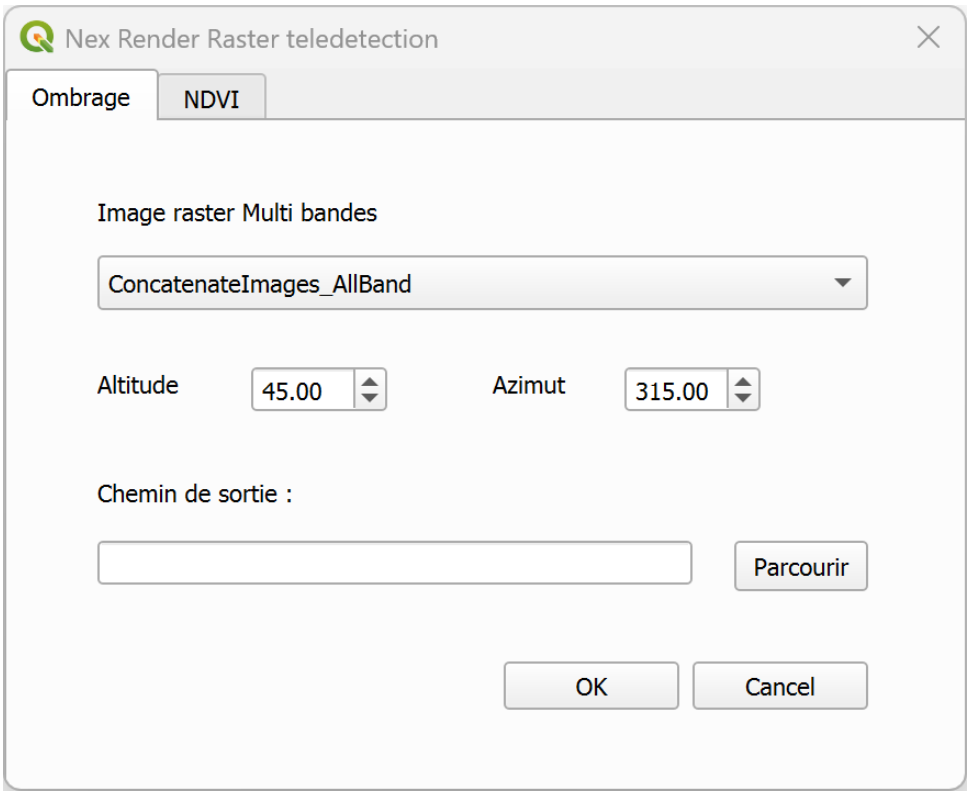

FIGURE 11 – Interface graphique du plugin Render dans QGIS pour l'ombrage

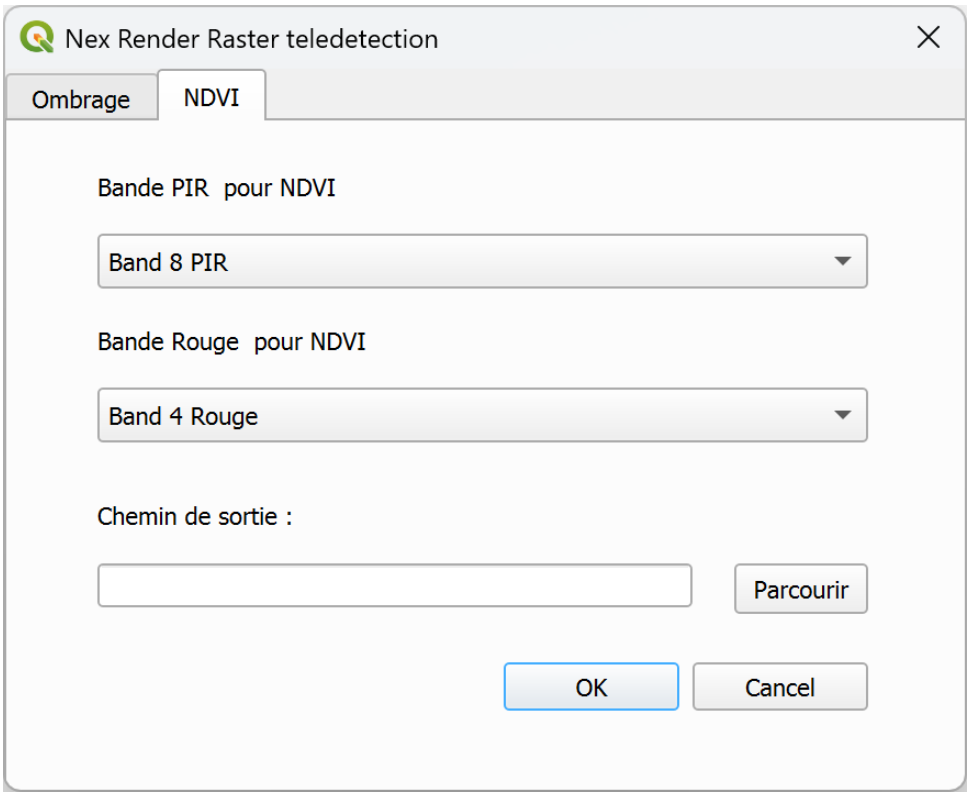

FIGURE 12 – Interface graphique du plugin Render dans QGIS pour le NDVI

5 Conclusion

Ce travail avait pour objectif initial de tester la faisabilité de l'intégration de nouveaux types de rendu raster au sein de QGIS, en particulier pour des traitements tels que le calcul d'indices (par exemple le NDVI, *Normalized Difference Vegetation Index*) ou l'ombrage en relief (*hillshade*). Pour atteindre cet objectif, deux approches distinctes ont été explorées : d'une part le développement d'un plugin Python s'appuyant sur l'API de QGIS et PyQt5, et d'autre part l'intégration directe de nouvelles fonctionnalités dans le code source C++ de QGIS. Nous comparons ci-dessous ces deux approches de manière structurée, en mettant en évidence leurs avantages et inconvénients respectifs.

5.1 Approche 1 : Développement d'un plugin Python

Avantages La solution via un plugin Python présente de nombreux atouts :

- **Simplicité et rapidité de mise en œuvre** – Aucun processus de compilation n'est requis ; le développement s'effectue en Python (langage de haut niveau) avec l'aide de PyQt5 pour l'interface. Ce contexte de travail permet un prototypage rapide et une expérimentation aisée, ce qui en fait une solution idéale pour tester de nouvelles idées ou dans un cadre pédagogique.
- **Accessibilité pour un utilisateur avancé** – La création ou l'installation d'un plugin Python est à la portée d'utilisateurs avancés de QGIS, sans nécessiter une expertise en développement C++ ni la manipulation du code source du logiciel. Le partage du plugin via le dépôt officiel des extensions QGIS facilite en outre sa diffusion et son adoption par la communauté.
- **Maintenabilité et évolutivité** – Le code Python du plugin est relativement facile à lire et à modifier. Il peut être mis à jour rapidement pour s'adapter aux nouvelles versions de QGIS. De plus, la distribution d'une nouvelle version du plugin est simple (il suffit de publier la mise à jour), ce qui garantit une certaine pérennité tant que le plugin est activement maintenu.

Inconvénients En contrepartie, cette approche présente quelques limitations :

- **Performances limitées** – L'exécution d'opérations lourdes (comme le calcul d'un NDVI sur de grandes images) dans l'environnement Python peut être moins efficace que dans du code natif. Le plugin peut donc être plus lent et consommer plus de ressources qu'une implémentation équivalente en C++ intégrée au cœur de QGIS.
- **Dépendance vis-à-vis de QGIS** – Le fonctionnement du plugin repose sur l'API QGIS qui est susceptible d'évoluer. Sans mise à jour appropriée, le plugin risque de ne plus fonctionner correctement avec les futures versions du logiciel. Autrement dit, sa pérennité dépend d'un effort de maintenance continu de la part de son développeur ou de la communauté.
- **Intégration moins transparente** – Étant externe au noyau de QGIS, le plugin ajoute une couche logicielle supplémentaire. L'utilisateur doit installer et activer manuellement le plugin pour en bénéficier, et l'ergonomie peut différer légèrement des outils natifs. De plus, certaines fonctionnalités avancées du rendu raster pourraient ne pas être accessibles via l'API Python, limitant potentiellement ce qu'il est possible de réaliser dans un plugin.

5.2 Approche 2 : Modification du code source C++ de QGIS

Avantages La solution consistant à modifier le code source C++ de QGIS offre, quant à elle, des avantages différents et complémentaires :

- **Performance et efficacité** – Une implémentation en C++ intégrée dans QGIS bénéficie des optimisations du code natif. Les nouveaux renderers raster ainsi ajoutés fonctionnent de manière optimale, ce qui est particulièrement avantageux pour les traitements intensifs (par exemple le calcul d'indices sur de grands rasters ou le rendu en relief d'un modèle d'élévation).
- **Intégration native et ergonomie** – En modifiant directement le code de QGIS, les fonctionnalités ajoutées sont intégrées nativement dans le logiciel. Ainsi, un nouvel indice comme le NDVI ou un mode d'affichage en relief apparaît comme une option de rendu standard dans l'interface de QGIS, sans nécessiter d'installation supplémentaire. L'expérience utilisateur est alors homogène, car ces outils s'utilisent comme n'importe quel autre renderer natif.
- **Pérennité et robustesse** – Une fonctionnalité intégrée au code source a vocation à être maintenue dans le cadre du projet QGIS lui-même. Si le code est accepté et incorporé dans la base officielle, il profitera des tests, des corrections de bogues et des améliorations continues apportées par la communauté des développeurs. Sur le long terme, cela assure une meilleure pérennité de la solution, qui évoluera de concert avec le logiciel.

Inconvénients En revanche, cette approche s'accompagne de plusieurs contraintes :

- **Complexité de développement et de maintenance** – L'ajout de code au cœur de QGIS requiert une expertise technique élevée (maîtrise du C++ et compréhension approfondie de l'architecture du logiciel). Le cycle de développement est plus lourd : il faut recompiler QGIS à chaque modification, ce qui ralentit les phases de test et d'itération. De plus, jusqu'à son éventuelle inclusion dans une version officielle, le code personnalisé doit être ajusté manuellement à chaque nouvelle mise à jour de QGIS, ce qui alourdit la maintenance.
- **Accessibilité limitée** – Contrairement au plugin Python, cette solution n'est pas facilement reproductible par la plupart des utilisateurs. Seuls des développeurs expérimentés peuvent mettre en œuvre et distribuer une version modifiée de QGIS. Pour un utilisateur avancé non développeur, il est pratiquement impossible d'appliquer cette approche de son propre chef, ce qui limite la diffusion immédiate de la fonctionnalité avant son intégration officielle.
- **Déploiement et diffusion plus longs** – Même une fois développée, la solution intégrée au code source ne peut être partagée qu'au travers d'une version compilée de QGIS (ou en attendant une prochaine version officielle incluant la fonctionnalité). Cela rend son déploiement plus laborieux et plus lent par rapport à un plugin, qui peut être diffusé instantanément et installé en quelques clics.

5.3 Synthèse comparative

En synthèse, chacune de ces deux approches répond à des besoins différents. Le plugin Python offre une solution rapide à développer, simple à partager et parfaitement adaptée au prototypage ou à la formation, tandis que l'intégration dans le code source fournit une solution plus robuste, performante et pérenne, quoique plus complexe à réaliser. Il

convient de souligner que ces deux stratégies ne sont pas exclusives l'une de l'autre : au contraire, un prototype développé sous forme de plugin peut servir de base et de banc d'essai pour une intégration native ultérieure dans QGIS. Ce dernier, en tant que logiciel libre en constante évolution, bénéficie ainsi de ces deux voies de développement parallèles qui permettent d'innover rapidement tout en ouvrant la voie à des améliorations durables au sein du projet officiel.

Références

[1] QGIS sur GitHub. *QGIS Project Repository*. Disponible sur : `https://github.com/qgis/QGIS`

[2] Documentation de QGIS. *Développement de plugins Python*. Disponible sur : `https://docs.qgis.org/3.40/fr/docs/pyqgis_developer_cookbook/plugins/index.html`