Cahier des Charges
Projet GIMP Remake

Laurent Jiang
Alessandro Tosi

11 février 2026

Table des matières

1 Présentation du Projet

1.1 Contexte

Le projet **GIMP Remake** est une réimplémentation complète ("from scratch") du célèbre logiciel de retouche d'image GIMP (GNU Image Manipulation Program). Contrairement au projet original écrit en C avec la bibliothèque GTK, cette refonte s'appuie sur des technologies modernes : C++20, le framework applicatif Qt6 et le moteur de rendu Skia.

Ce projet ne consiste pas à créer une simple interface ("wrapper") autour du code existant de GIMP, mais bien à reconstruire l'architecture logicielle en utilisant des paradigmes de programmation modernes et orientés objet.

1.2 Nature du Logiciel

Il s'agit d'une application de bureau ("Desktop Application") multiplateforme destinée à la création graphique, la retouche photo et la composition d'images. Le logiciel se veut performant grâce à l'accélération matérielle (GPU) et modulaire.

1.3 Public Cible

— **Utilisateurs finaux :** Graphistes, photographes et amateurs souhaitant un outil de retouche performant et réactif.

— **Développeurs et Étudiants :** Le projet sert également de démonstrateur technique pour l'apprentissage de l'architecture logicielle complexe (C++20, Design Patterns, Gestion mémoire).

2 Objectifs du Projet

2.1 Objectifs Pédagogiques

Le projet vise à servir de cas d'école pour comprendre comment se construit un éditeur d'image complexe. Il met l'accent sur :

— L'application des principes **RAII** (Resource Acquisition Is Initialization) et de la gestion mémoire moderne.

— L'implémentation de **Design Patterns** classiques (Command, Strategy, Observer, Factory).

— La compréhension des pipelines de rendu graphique et des algorithmes de traitement d'image.

2.2 Objectifs Techniques

— **Modernisation :** Utiliser exclusivement C++20 (Ranges, Concepts).
— **Performance :** Utiliser Skia pour un rendu accéléré par le GPU, offrant une fluidité supérieure lors des zooms et des déplacements sur le canevas.
— **Maintenabilité :** Une architecture en couches (Layered Architecture) claire, découplant l'interface utilisateur de la logique métier.
— **Indépendance :** S'affranchir de la complexité de l'écosystème GLib/GTK original pour une intégration native Qt.

3 Fonctionnalités Cœur

3.1 Interface Utilisateur (UI)

L'interface reprend les standards ergonomiques des logiciels de retouche (type Photoshop/GIMP) :

— Système de panneaux ancrables ("Docking Layout") : Boîte à outils, Calques, Historique.
— Thème sombre moderne adapté à la création graphique.
— Palette de commandes ("Command Palette") pour un accès rapide aux fonctions.
— Overlay de débogage (HUD) affichant les performances (FPS, mémoire).

3.2 Outils de Création

— **Dessin :** Pinceau (Brush) avec dynamique de vélocité, Crayon (Pencil) pixel-perfect, Gomme (Eraser).
— **Remplissage :** Outil Pot de peinture (Bucket Fill) et Dégradés (Linéaire, Radial).
— **Sélection :** Rectangle, Ellipse, Lasso (Sélection libre). Support des opérations d'ajout/soustraction.
— **Pipette :** Sélection de couleur sur le canevas.

3.3 Gestion des Calques

Système complet d'empilement de calques ("Layer Stack") supportant :

— Visibilité et opacité par calque.
— Modes de fusion (Normal, Multiply, Screen, Overlay, etc.).
— Réorganisation de la pile de calques.

3.4 Traitement d'Image

Application de filtres destructifs sur les calques :

— Flou Gaussien (Gaussian Blur).
— Accentuation de netteté (Unsharp Masking).

4 Spécifications Techniques

4.1 Stack Technologique

— **Langage :** C++20.
— **UI Framework :** Qt6 (Widgets + SVG).
— **Moteur de Rendu :** Skia (Google).
— **I/O Image :** OpenCV (chargement/sauvegarde de fichiers standards).
— **Logging :** spdlog.
— **Build System :** CMake + Ninja.
— **Gestionnaire de paquets :** vcpkg (Manifest mode).

4.2 Architecture Logicielle

Le projet suit une architecture en couches stricte :

1. **UI Layer :** 'MainWindow', 'SkiaCanvasWidget', Panels. Gère l'affichage et les entrées Qt.
2. **Application Layer :** 'ToolFactory', 'CommandBus', 'EventBus'. Fait le lien entre l'UI et le domaine.
3. **Domain Layer :** 'Document', 'Layer', 'Tool', 'Filter'. Contient la logique métier pure et les données.
4. **Infrastructure Layer :** 'SkiaRenderer', 'IOManager'. Implémentations techniques bas niveau.

4.3 Architecture des Données

— **Format Pixel :** RGBA 32-bits (0xRRGGBBAA).
— **Gestion Mémoire :** Utilisation intensive de 'std : :shared$_ptr$'et'$std :: unique_ptr$'.
— **Historique :** Implémentation via le pattern **Command**. Chaque action modifiant le document stocke l'état avant/après pour permettre l'annulation (Undo/Redo).

4.4 Structure de l'Espace de Travail

4.4.1 Arborescence des Fichiers

L'organisation des fichiers du projet est structurée pour séparer clairement les interfaces (headers) des implémentations, ainsi que les tests et les ressources :

```
gimp-remake/
|-- include/           # Déclarations publiques (.h)
|   |-- core/          # Logique métier (outils, commandes)
|   |-- ui/            # Composants graphiques Qt
|   |-- render/        # Moteur de rendu
|   |-- io/            # Entrées/Sorties fichiers
|-- src/               # Implémentations (.cpp)
|   |-- main.cpp       # Point d'entrée de l'application
|   |-- ...            # Structure miroir de include/
|-- tests/             # Suite de tests
|   |-- unit/          # Tests unitaires purs
|   |-- integration/   # Tests avec I/O ou rendu
|-- scripts/           # Scripts de build et maintenance
|-- resources/         # Assets et configurations
|-- docs/              # Documentation technique
```

4.4.2 Conventions de Nommage

Le projet applique des conventions strictes pour faciliter la navigation et la maintenance :

Type	**Format**	**Exemple**
Fichiers	`snake_case`	`brush_tool.h`, `main_window.cpp`
Classes	`PascalCase`	`BrushTool`, `MainWindow`
Fonctions/Méthodes	`camelCase`	`addLayer()`, `setColor()`
Tests	`test_<nom>`	`test_history.cpp`

4.5 Algorithmes Clés

- **Compositing :** Opérations alpha "Porter-Duff" (Over, Source-Over) pour la gestion de la transparence.
- **Flood Fill :** Algorithme par scanline (lignes de balayage) pour le remplissage de zones contiguës.
- **Flou :** Convolution séparable (passes horizontale puis verticale) pour optimiser la complexité algorithmique en $O(n \cdot k)$ au lieu de $O(n \cdot k^2)$.
- **Interpolation :** Bilinéaire pour la mise à l'échelle et le rendu fluide.
- **Conversion Couleur :** Transformations RGB $\leftrightarrow$ HSV pour le sélecteur de couleurs.

4.6 Tests et Qualité

- **Unit Testing :** Framework Catch2 v3.
- **Intégration Continue (CI) :** Workflows GitHub Actions pour la compilation, le formatage (‘clang-format‘) et l'analyse statique (‘clang-tidy‘).

5 Suivi du Projet et Livrables

5.1 Organisation des Milestones

Le développement est découpé en jalons (milestones) incrémentaux pour assurer une livraison continue de fonctionnalités.

v0.1.0 - Fondations : Mise en place de l'architecture cœur.

- Pipeline de rendu (Core Rendering) et composition de calques.
- Squelette de l'interface graphique (Shell Docking) en Qt6.
- Système d'historique (Undo/Redo) via le pattern Command.
- Gestion des fichiers (Image I/O) et toile interactive (Canvas).
- Architecture de gestion d'erreurs et intégration continue (CI).

v0.2.0 - Outils de Dessin : Premiers outils créatifs.

- Implémentation des outils : Pinceau, Gomme, Pipette, Pot de peinture, Dégradé.
- Panneaux d'options d'outils et sélecteur de couleurs.
- Raccourcis clavier.
- Moteur de filtres de base.

v0.3.0 - Sélection & Transformation : Cœur des outils de manipulation.

- Outils de sélection (Rectangle, Ellipse) et transformations.
- Opérations Presse-papier (Copier/Coller).
- Gestion des fichiers projets (.json) avec support multi-calques.
- HUD de débogage et reporting d'erreurs.

v0.4.0 - Gestion Avancée des Calques : Consolidation du moteur de composition.

- Verrouillage et visibilité des calques.
- Modes de fusion (Blend Modes).
- Fusion de calques (Merge Down / Flatten).
- Sauvegarde automatique et récupération de session.

v0.5.0 - Outils Avancés : Enrichissement fonctionnel.

v1.0.0 - Release Finale : Version stable complète.

5.2 Livrables

1. **Code Source :** Dépôt Git structuré et documenté.
2. **Exécutable :** Binaire autonome ("Standalone") pour Windows x64.
3. **Documentation Technique :**
   - `ARCHITECTURE.md` : Guide d'architecture et de maintenance.
   - Documentation API générée par Doxygen.
4. **Rapport de Tests :** Couverture de code ($> 50\%$) et résultats des tests unitaires/intégration.

5.3 Organisation du Développement et Qualité

5.3.1 Workflow de Collaboration

Le projet suit un flux de travail rigoureux pour garantir la stabilité de la branche principale :

- **Protection de la branche 'main' :** Aucune modification directe n'est autorisée. Tout changement doit passer par une **Pull Request (PR)**.
- **Branche par Issue :** Chaque fonctionnalité ou correctif doit être développé sur une branche dédiée, liée à une issue spécifique (ex : `feature/60-rect-select`).
- **Conventional Commits :** Les messages de commit doivent respecter la convention *Conventional Commits* (ex : `feat: add brush tool`, `fix: crash on undo`) pour faciliter la génération automatique du changelog.

5.3.2 Stratégie QA et CI/CD

L'assurance qualité repose sur une double validation, locale et distante :

1. Validation Locale (Scripts) Avant toute soumission, le développeur doit valider son code via les scripts fournis :

- `scripts/run-format.ps1` : Formatage automatique du code (Clang-Format).
- `scripts/run-lint.ps1` : Analyse statique et détection de bugs potentiels (Clang-Tidy).

2. Pipeline d'Intégration Continue (GitHub Actions) À chaque Pull Request, le pipeline CI exécute automatiquement :

- Compilation sous Windows (plateforme cible principale).
- Vérification du respect du formatage et absence de warnings du linter.
- Exécution complète de la suite de tests (Unitaires et Intégration).
- Génération d'un rapport de couverture de code (Cible : $> 50\%$).