EPITA

Rapport de Projet S4

Application de peinture numérique - Rustique 3.0

FIGURE 1 – *
Logo du projet Rustique - Version finale

Réalisé par :

Vincent NEDELIAN

Leonardo MONROC

Nabil BOROUS

Aurélien MAZE

Mai 2025

Table des matières

1

Introduction

1.1 Présentation du projet

Rustique 3.0, c'est l'aboutissement de quatre mois de développement intensif qui nous ont menés bien plus loin que ce qu'on imaginait au départ. Quand on a commencé en février 2025, on se disait qu'on allait faire une petite application de dessin en Rust. Au final, on s'est retrouvés avec un vrai logiciel de peinture numérique qui peut rivaliser avec des solutions commerciales.

Le projet a vraiment pris forme autour de l'idée de créer quelque chose d'utile avec Rust, le langage qu'on devait apprendre dans le cadre de nos études. On voulait pas juste faire un énième clone de Paint, mais plutôt comprendre ce qui rend un logiciel de création à la fois puissant et agréable à utiliser. Rust nous permettait d'avoir les performances nécessaires sans les galères de gestion mémoire qu'on aurait eues avec d'autres langages.

Ce qui nous a motivés dès le début, c'est l'envie de créer quelque chose de concret qu'on pourrait vraiment utiliser après le projet. Trop souvent, les projets étudiants finissent dans un tiroir virtuel. Là, on voulait que Rustique puisse servir à de vrais artistes pour créer de vraies œuvres. Cette ambition nous a poussés à aller au bout des choses même quand c'était compliqué.

Notre équipe de quatre développeurs s'est très vite organisée autour de spécialisations naturelles, tout en gardant une vision commune du produit final. Vincent s'est passionné pour les algorithmes de dessin, Leonardo a plongé dans l'architecture et le rendu, Nabil a masterisé tout ce qui touche aux fichiers et formats, et Aurélien a créé une interface utilisateur vraiment agréable.

L'évolution du projet s'est faite de manière organique. On avait prévu trois phases, mais en réalité chaque phase nous a amenés à repenser nos objectifs pour la suivante. La Phase 1.0 nous a montré qu'on pouvait faire du bon travail ensemble. La Phase 2.0 nous a convaincus qu'on pouvait viser haut avec les calques. Et la Phase 3.0 nous a permis de vraiment fignoler pour avoir quelque chose dont on peut être fiers.

Ce qui rend Rustique intéressant, c'est pas seulement les fonctionnalités qu'on a réussi à implémenter, mais aussi la façon dont on les a pensées. Le format .rustiq, par exemple, ça nous permet de sauvegarder exactement l'état de travail d'un artiste sans rien perdre. C'est le genre de détail qui fait la différence quand on utilise vraiment l'outil au quotidien.

1.2 Contexte et motivation

L'idée de Rustique est née d'un constat simple : les logiciels de peinture numérique sont soit très chers, soit très complexes, soit pas terribles côté performances. GIMP est gratuit mais pas toujours intuitif, Photoshop coûte cher et fait trop de choses, Krita est bien mais parfois lourd. On se disait qu'il y avait de la place pour quelque chose de différent.

Comme on devait apprendre Rust dans le cadre de nos études, autant s'y mettre avec un vrai projet. Le langage promettait des performances de niveau C/C++ sans les problèmes de sécurité mémoire, et ça tombait bien pour une application graphique qui va manipuler beaucoup de données pixel. En plus, l'écosystème Rust commençait à être assez mature pour qu'on puisse s'appuyer sur de bonnes bibliothèques.

Le timing était bon aussi. Les outils de création numérique évoluent vite, et il y a de la place pour de nouveaux acteurs qui repensent l'expérience utilisateur. On voyait bien que les artistes numériques cherchent des outils plus directs, moins surchargés que les mastodontes traditionnels. Rustique pouvait être cette alternative moderne et épurée.

Notre motivation venait aussi de l'envie d'apprendre en profondeur. Faire une application complète, c'est toucher à tout : architecture logicielle, algorithmes, interface utilisateur, gestion de fichiers, optimisation... C'est le genre de projet qui nous force à sortir de notre zone de confort et à vraiment progresser techniquement.

L'aspect créatif nous motivait énormément. Développer des outils de création, c'est en quelque sorte participer à l'art qui sera créé avec. Chaque pinceau qu'on programme, chaque effet qu'on implémente peut servir à créer quelque chose de beau. Cette dimension nous donnait envie de bien faire les choses.

Il y avait aussi un côté défi technique qui nous plaisait. Les applications graphiques, c'est complexe : il faut gérer les performances, la mémoire, l'affichage temps réel, les formats de fichiers... Ça nous permettait de travailler sur des problèmes variés et intéressants. Et Rust nous donnait confiance pour s'attaquer à cette complexité sans se planter.

Enfin, on voulait prouver qu'une équipe d'étudiants pouvait créer quelque chose de

quality professionnelle. Pas juste un prototype qui marche dans des conditions idéales, mais un vrai logiciel qu'on peut utiliser pour de vrais projets. Cette ambition nous a poussés à aller jusqu'au bout des fonctionnalités et à vraiment soigner les détails.

1.3 Cahier des charges

Définir précisément ce qu'on voulait faire avec Rustique, ça nous a pris du temps au début. Pas parce qu'on manquait d'idées, mais plutôt parce qu'on en avait trop ! Il a fallu se fixer des limites réalistes tout en gardant assez d'ambition pour que le projet reste motivant.

Pour les fonctionnalités de base, on était d'accord sur l'essentiel : il nous fallait un canvas sur lequel on peut dessiner avec différents outils. Le pinceau, la gomme, le seau de peinture et la pipette étaient nos incontournables. Après avoir testé plein de logiciels existants, on s'est rendu compte que ces quatre outils couvrent déjà 80% des besoins basiques de dessin numérique.

Le système de calques, au début on hésitait. C'est compliqué à implémenter et ça change complètement l'architecture. Mais quand on a commencé à discuter avec des gens qui font du graphisme, ils nous ont dit que sans calques, on ne pouvait pas vraiment parler d'outil professionnel. Ça nous a convaincus de l'inclure, même si on savait que ça allait être le gros morceau technique du projet.

Pour l'interface, on voulait quelque chose de propre et intuitif. Pas la surcharge de boutons de GIMP, pas la complexité de Photoshop, mais pas non plus la simplicité extrême de Paint qui limite les possibilités. Il fallait trouver le bon équilibre entre puissance et accessibilité. L'idée était qu'un débutant puisse commencer à dessiner tout de suite, mais qu'un utilisateur avancé puisse accéder rapidement aux fonctions complexes.

Côté technique, Rust était non négociable pour nous. On voulait apprendre le langage en profondeur, et surtout bénéficier de ses avantages pour la performance et la sécurité. Pour l'interface utilisateur, egui nous semblait le bon compromis : assez mature pour être stable, assez simple pour qu'on puisse maîtriser rapidement, assez performant pour nos besoins.

La gestion des formats de fichiers était importante dès le départ. On savait que si Rustique ne pouvait pas importer/exporter les formats standards, personne ne l'utiliserait. PNG et JPEG étaient évidemment prioritaires, mais on voulait aussi supporter une gamme plus large pour être vraiment utile dans différents contextes.

L'idée du format natif .rustiq est venue assez tôt dans la réflexion. On se disait que

si on allait jusqu'au bout avec les calques et les fonctions avancées, il nous faudrait un moyen de sauvegarder tout ça sans rien perdre. Les formats standards ne preservent jamais toute l'information de travail, alors qu'un projet créatif, c'est pas juste le résultat final mais aussi tout le processus.

Pour les performances, on s'était fixé comme objectif de rester fluide même sur des canvas de 2000×2000 pixels avec plusieurs calques. C'est la taille typique pour de l'illustration numérique sérieuse, et si on n'arrivait pas à être réactifs là-dessus, Rustique ne servirait que pour du gribouillis.

L'extensibilité était dans un coin de notre tête dès le début. On ne pouvait pas tout faire en un semestre, mais on voulait construire une base qui permettrait d'ajouter facilement de nouvelles fonctionnalités plus tard. Cette approche nous a amenés à bien réfléchir à l'architecture plutôt que de juste empiler du code qui marche.

1.4 Méthodologie de travail

S'organiser à quatre sur un projet technique de cette ampleur, ça s'apprend sur le tas. Au début, on pensait naïvement qu'il suffirait de se répartir les tâches et que chacun coderait dans son coin. On s'est vite rendu compte que c'était plus compliqué que ça.

Notre première approche était très... disons artisanale. Chacun prenait un morceau du projet et on se retrouvait une fois par semaine pour voir où on en était. Ça a marché pendant les deux premières semaines, puis on a commencé à avoir des problèmes d'intégration. Le code de Vincent ne s'interfaçait pas bien avec celui de Leonardo, les conventions de Nabil étaient différentes de celles d'Aurélien... bref, c'était le bazar.

La révélation, ça a été de mettre en place de vraies sessions de travail ensemble. Plutôt que chacun dans sa chambre, on s'est mis à se retrouver en salle machine ou chez l'un de nous pour coder en équipe. Au début, on pensait que ça nous ferait perdre du temps, mais en fait ça en faisait gagner énormément. Les problèmes se résolvaient au fur et à mesure, les idées circulaient mieux, et on évitait les gros bugs d'intégration.

Pour la communication au quotidien, Discord s'est imposé naturellement. On avait un serveur dédié au projet avec des canaux séparés pour les différents aspects : développement, bugs, idées, partage de ressources... Ça nous permettait de rester en contact même quand on travaillait chacun de son côté. Et c'est fou comme on peut résoudre rapidement un problème technique en partageant son écran sur Discord.

Git, au début, c'était notre ennemi. On avait beau connaître les commandes de base, gérer les branches et les merges à quatre, c'est une autre histoire. Les premiers conflits nous ont fait perdre des heures, parfois des journées de travail. Mais on a appris,

et finalement on a développé une stratégie qui nous convenait : chacun sa branche de fonctionnalité, merge réguliers sur main, et surtout, toujours tester avant de pusher.

La revue de code s'est mise en place naturellement. Au début, on se contentait de faire confiance aux autres, mais on s'est rendu compte qu'un regard extérieur permettait d'attraper des bugs qu'on ne voyait plus à force de regarder son propre code. Et puis c'était un bon moyen d'apprendre les techniques des autres. Vincent nous a appris des astuces d'optimisation, Leonardo nous a montré comment bien structurer l'architecture, Nabil nous a sensibilisés à la robustesse, Aurélien nous a fait comprendre l'importance de l'ergonomie.

Pour la planification, on a essayé plusieurs approches. Au début, on voulait faire du Scrum en bonne et due forme avec des sprints de deux semaines et tout. Mais en pratique, on était trop flexibles pour ça. Ce qui a marché pour nous, c'était plutôt de définir des objectifs par phase et de s'adapter en cours de route selon ce qu'on découvrait.

Les moments difficiles, il y en a eu. Surtout vers la fin de la Phase 2.0 quand on s'est retrouvés avec des bugs complexes qui touchaient à l'intégration entre plusieurs modules. On avait l'impression de régresser au lieu d'avancer. C'est là qu'on a appris l'importance de bien tester et de ne pas vouloir aller trop vite.

La documentation, on l'a un peu négligée au début. Chacun se contentait de documenter son code, mais on n'avait pas de vision d'ensemble. Vers la Phase 3.0, on s'est rendu compte qu'on perdait du temps à se redemander comment fonctionnaient certaines parties. Alors on s'est mis à vraiment documenter les interfaces et les choix architecturaux importants.

Ce qui nous a le plus aidés, au final, c'est de rester flexibles tout en gardant une vision commune. On s'est adaptés aux contraintes qu'on découvrait, mais on n'a jamais perdu de vue ce qu'on voulait créer. Et l'ambiance dans l'équipe est restée bonne même dans les moments de stress, ce qui n'est pas donné à toutes les équipes de projet.

2

État des lieux initial et objectifs

2.1 Analyse des besoins

Avant de se lancer tête baissée dans le développement, on a pris le temps d'analyser ce qui existait déjà dans le monde de la peinture numérique. Et franchement, l'offre est à la fois très riche et un peu frustrante. Chaque logiciel a ses forces et ses faiblesses, mais aucun ne nous semblait parfait.

GIMP, par exemple, c'est gratuit et puissant, mais l'interface fait parfois mal aux yeux. On dirait qu'elle a été conçue par des développeurs pour des développeurs, pas pour des artistes. Photoshop, c'est l'étalon-or, mais il coûte cher et fait tellement de choses qu'on se perd facilement. Krita est vraiment bien fait pour la peinture, mais parfois un peu lourd. Paint.NET est simple et efficace, mais limité pour des projets ambitieux.

Ce tour d'horizon nous a aidés à définir ce qu'on voulait vraiment faire avec Rustique. On ne visait pas à remplacer Photoshop - on n'est pas fous - mais plutôt à créer un outil qui prendrait le meilleur de chaque approche. L'intuitivité de Paint.NET, la puissance de GIMP, l'élégance de Krita, mais dans un package plus moderne et plus performant.

Côté fonctionnalités essentielles, on a identifié les incontournables : créer un canvas, dessiner dessus avec différents outils, gérer les couleurs de façon intuitive, sauvegarder son travail. Ça paraît basique dit comme ça, mais bien implementer ces fonctions de base, c'est déjà un gros boulot. Et si ces fondations ne sont pas solides, tout le reste s'effrite.

Pour les fonctionnalités avancées, les calques s'imposaient comme priorité absolue. Tous les logiciels sérieux en ont, et on comprend vite pourquoi quand on commence à faire des compositions complexes. La possibilité de travailler par couches indépendantes change complètement la façon dont on crée. Ça ouvre des possibilités créatives qu'on n'a pas sans calques.

Les utilisateurs qu'on a interrogés (des potes qui font du graphisme, des profs d'art numérique, quelques illustrateurs qu'on connaît) nous ont tous dit la même chose :

l'ergonomie, c'est au moins aussi important que les fonctionnalités. Un outil peut être très puissant, s'il est chiant à utiliser, on finit par le délaisser. L'interface doit être logique, les raccourcis intuitifs, les opérations courantes facilement accessibles.

Une chose qui nous a surpris dans notre analyse, c'est l'importance du workflow complet. Les artistes ne créent pas dans le vide : ils importent des références, exportent pour différents usages, partagent leur travail... Un bon logiciel de peinture, c'est pas juste un endroit où on fait de jolis pixels, c'est un maillon d'une chaîne créative plus large.

Les contraintes techniques qu'on a identifiées étaient principalement liées aux performances. Manipuler des images de plusieurs millions de pixels en temps réel, ça demande des optimisations sérieuses. La mémoire aussi peut vite devenir un problème, surtout avec des projets multi-calques complexes. Et il faut que tout reste fluide même quand l'utilisateur fait des opérations coûteuses.

L'analyse nous a aussi fait réaliser l'importance de bien gérer les cas d'erreur. Quand on travaille sur un projet créatif pendant des heures, la dernière chose qu'on veut c'est perdre son travail à cause d'un bug ou d'un plantage. La robustesse et la capacité de récupération après problème, c'est crucial pour un outil de création.

Une dernière chose qui est ressortie de notre analyse : l'importance de l'évolutivité. Les besoins des créateurs évoluent, les technologies progressent, les standards changent. Un logiciel qui ne peut pas s'adapter finit rapidement obsolète. Il fallait qu'on conçoive Rustique de façon à pouvoir facilement ajouter de nouvelles fonctionnalités sans tout casser.

2.2 Technologies choisies

Rust étant le langage imposé pour ce projet dans le cadre de notre formation, on a dû apprendre à bien l'utiliser pour une application graphique complexe. Au final, ça s'est révélé être un excellent choix : le langage nous donne les performances du C/C++ sans les galères de gestion mémoire.

Le gros avantage qu'on a découvert avec Rust, c'est qu'il nous aide à éviter plein de bugs qu'on aurait eus dans d'autres langages. Pour une application qui va manipuler des millions de pixels et faire des calculs intensifs, c'est crucial. Et puis, le système de types de Rust nous aide énormément : quand ça compile, on a une bonne confiance que ça va marcher comme prévu.

Pour l'interface utilisateur, on a hésité entre plusieurs options. Flutter était tentant pour la beauté des interfaces, mais on avait peur que l'intégration avec Rust soit com-

pliquée. GTK ou Qt auraient été plus traditionnels, mais on voulait quelque chose de plus moderne. Au final, egui nous a séduits par sa simplicité et son intégration native avec Rust.

egui, c'est un framework GUI "immédiat", ce qui change un peu de ce qu'on connaissait. Au lieu de créer des objets interface qui persistent, on redessine tout à chaque frame en fonction de l'état de l'application. Ça paraît bizarre au début, mais en fait c'est très libérateur : on n'a pas à gérer la synchronisation entre l'état de l'interface et l'état de l'application.

Pour la manipulation d'images, la crate 'image' s'imposait naturellement. C'est la référence dans l'écosystème Rust, elle supporte tous les formats qu'on voulait, et elle est bien optimisée. Plutôt que de réinventer la roue pour les algorithmes de base, on pouvait se concentrer sur ce qui fait la spécificité de Rustique.

La sérialisation avec serde et JSON pour notre format .rustiq, c'était un choix de simplicité et de flexibilité. JSON, c'est pas le plus compact, mais c'est lisible, debuggable, et extensible. Pour un projet étudiant, ces avantages compensaient largement le surcoût en taille de fichier. Et serde nous donnait une sérialisation automatique très pratique.

Pour le développement et l'outillage, on s'est appuyés sur l'écosystème standard de Rust. Cargo pour gérer les dépendances et la compilation, c'est vraiment bien fait. rustfmt pour formater le code automatiquement, ça évite les disputes sur le style. clippy pour détecter les problèmes potentiels, ça nous a appris plein de bonnes pratiques Rust.

Le choix de rester sur du rendu CPU plutôt que GPU au début, c'était une décision pragmatique. Bien sûr, le GPU serait plus performant pour certaines opérations, mais ça ajoutait une complexité qu'on préférait éviter pour commencer. L'idée était de faire quelque chose qui marche bien d'abord, et d'optimiser ensuite.

Une des bonnes surprises avec Rust, c'est l'écosystème de tests. Les tests unitaires sont intégrés au langage, et ça nous a encouragés à bien tester notre code. C'est quelque chose qu'on néglige souvent dans les projets étudiants, mais là, c'était tellement facile qu'on s'est pris au jeu.

La documentation générée automatiquement par rustdoc, c'est aussi un point fort qu'on a découvert en cours de route. Ça nous a motivés à bien documenter notre code, parce qu'on voyait immédiatement le résultat sous forme de doc web propre et navigable.

Au final, cet ensemble de technologies nous a donné une base solide et moderne pour développer Rustique. On avait les performances nécessaires, un développement agréable, et une architecture extensible. C'était exactement ce qu'il nous fallait pour mener à bien notre projet.

2.3 Planification du développement

Planifier un projet de développement quand on est étudiants, c'est un art délicat. On a tendance à être soit trop optimistes (« ça va être facile ! »), soit trop pessimistes (« on n'y arrivera jamais »). Pour Rustique, on a essayé de trouver un équilibre réaliste tout en gardant assez d'ambition pour rester motivés.

Notre approche en trois phases, ça nous est venu naturellement. L'idée était d'avoir toujours quelque chose qui marche, même si ce n'est pas complet. Ça évite l'effet tunnel où on développe pendant des mois sans rien montrer, et au final on se rend compte que ça ne marche pas comme prévu.

La Phase 1.0, on l'avait pensée comme notre « proof of concept ». Il fallait qu'on arrive à créer quelque chose qui ressemble à une application de peinture, même basique. Canvas, pinceau, couleurs, sauvegarde : les fondations absolues. Si on n'arrivait pas à faire ça proprement, le reste n'avait aucune chance de marcher.

Pour cette première phase, on s'était donné six semaines. Avec le recul, c'était un peu juste, mais ça nous a forcés à aller à l'essentiel. Pas question de se perdre dans des détails ou des optimisations prématurées. Il fallait que ça marche, point. Et c'est une bonne discipline quand on débute sur un nouveau projet.

La Phase 2.0, c'était le gros morceau. Ajouter les calques, ça changeait tout dans l'architecture. On le savait, mais on sous-estimait à quel point. Du coup, on s'était donné huit semaines pour cette phase, en se disant qu'on aurait sûrement des surprises. Spoiler : on en a eu !

L'idée pour la Phase 2.0 était de transformer notre prototype en vrai outil de création. Les calques, mais aussi le support multi-formats, le format natif, l'historique d'annulation... Tout ce qui fait qu'on passe d'un jouet sympa à quelque chose qu'on peut vraiment utiliser pour créer.

La Phase 3.0, on l'avait prévue pour le polish et les fonctionnalités avancées. Formes géométriques, modes de fusion, optimisations... En théorie, c'était la phase la plus « relaxe » parce qu'on avait déjà un outil fonctionnel. En pratique, c'est là qu'on s'est rendu compte de tous les détails qui clochaient et qu'il fallait corriger.

Notre planning prévisionnel, on l'avait fait sur une feuille de calcul avec des jolies couleurs et des dates précises. Dans la réalité, on l'a adapté au fur et à mesure. Certaines tâches ont pris plus de temps que prévu (l'intégration des calques avec les outils existants, par exemple), d'autres moins (l'outil ligne était plus simple qu'on pensait).

Les jalons intermédiaires, ça nous a sauvé la mise plusieurs fois. Chaque semaine, on faisait le point sur ce qui marchait et ce qui coincait. Ça nous permettait de réajuster

le tir rapidement. Et surtout, ça maintenait la motivation de voir régulièrement qu'on avançait.

Une chose qu'on n'avait pas assez anticipée, c'est le temps de debug et d'intégration. Quand chacun développe son module dans son coin, tout a l'air de marcher. Mais quand on assemble tout, c'est souvent là que les problèmes apparaissent. On a appris à réserver du temps spécialement pour ça.

La gestion des risques, au début on n'y pensait pas trop. Mais après quelques galères (un bug qui nous a fait perdre une semaine, un problème de performance qu'on n'avait pas vu venir), on a commencé à identifier les points critiques et à prévoir des solutions de repli.

Au final, notre planning s'est révélé plutôt bon, mais surtout notre capacité d'adaptation. Le projet était assez bien structuré pour qu'on puisse ajuster les priorités sans tout remettre en question. Et c'est peut-être ça, la vraie leçon : un bon planning, c'est pas celui qui prévoit tout parfaitement, c'est celui qui permet de s'adapter quand la réalité rattrape les prévisions.

2.4 Répartition des rôles dans l'équipe

Se répartir le travail à quatre sur un projet comme Rustique, c'est un peu comme former un groupe de musique : il faut que chacun trouve son instrument et que l'ensemble sonne juste. Au début, on s'est basés sur nos affinités et nos petites expériences, mais on s'est vite rendu compte que les rôles évoluaient naturellement selon les besoins du projet.

Vincent, dès le départ, était attiré par tout ce qui touchait aux algorithmes de dessin. C'est le genre de mec qui peut passer des heures à optimiser un algorithme de tracé de ligne pour gagner quelques millisecondes. Du coup, il était logique qu'il s'occupe des outils de base : pinceau, gomme, seau, pipette. Ces outils, c'est vraiment le cœur de l'expérience utilisateur, et Vincent a cette obsession du détail qui fait la différence.

Son truc, c'est la précision mathématique. Quand il explique comment fonctionne l'antialiasing de son pinceau ou pourquoi il a choisi tel algorithme de flood fill, on voit qu'il a vraiment creusé le sujet. Et ça se sent dans le résultat : ses outils sont fluides, précis, agréables à utiliser. C'est le genre de qualité qu'on ne voit pas forcément au premier coup d'œil, mais qui fait toute la différence à l'usage.

La suite logique pour Vincent, c'était de s'attaquer aux outils plus complexes : l'outil ligne avec son aperçu temps réel, puis les formes géométriques. Là aussi, il y avait des défis intéressants d'algorithmique et d'optimisation. Et Vincent aime bien quand c'est

un peu technique.

Leonardo, lui, c'est notre architecte. Quand on débat d'une solution technique, c'est souvent lui qui pose les bonnes questions : « Et ça scale comment ? », « Qu'est-ce qui se passe si on a 50 calques ? », « Cette approche va nous poser des problèmes plus tard ? ». Il a cette vision systémique qui nous aide à éviter de nous peindre dans un coin.

Le canvas et le système de rendu, c'était fait pour lui. Il fallait quelqu'un qui puisse penser l'architecture de données de façon propre et extensible. Et quand est venu le moment des calques, c'était évident que Leonardo devait s'en occuper. Ce système touche à tout dans l'application, il fallait quelqu'un qui maîtrise bien l'ensemble.

Ce qui impressionne avec Leonardo, c'est sa capacité à refactorer du code complexe sans tout casser. Plusieurs fois, on a eu besoin de revoir l'architecture d'un module pour ajouter une nouvelle fonctionnalité, et il arrivait à faire ça proprement. C'est un talent qu'on ne développe qu'avec l'expérience et la réflexion.

Nabil, c'est notre spécialiste de tout ce qui touche aux données et aux fichiers. Au début, ça pouvait paraître moins glamour que les algorithmes de dessin ou l'architecture, mais on s'est vite rendu compte que c'était crucial. Un logiciel de création qui ne sait pas bien gérer les fichiers, c'est un logiciel qu'on n'utilise pas.

Son travail sur le support multi-formats nous a ouvert beaucoup de portes. Pouvoir importer et exporter vers tous les formats standards, ça change tout pour l'intégration dans des workflows existants. Et le format .rustiq, c'est vraiment son bébé. Il a réussi à créer quelque chose qui préserve toute l'information du projet tout en restant raisonnablement compact.

Ce qui caractérise Nabil, c'est son attention aux détails. Il pense aux cas limites, aux gestions d'erreur, aux problèmes de compatibilité. C'est le genre de travail ingrat mais essentiel. Grâce à lui, on n'a jamais eu de problème de corruption de données ou de fichiers illisibles.

Aurélien, c'est notre expert UX/UI. Et honnêtement, sans lui, Rustique aurait probablement eu l'interface typique d'un projet étudiant : fonctionnelle mais pas folichonne. Aurélien a cette sensibilité pour l'ergonomie et l'esthétique qui fait qu'on a envie d'utiliser l'application.

Son travail sur l'interface avec egui nous a montré que même avec des contraintes techniques (egui n'est pas le framework le plus flexible), on peut créer quelque chose d'agréable et d'intuitif. Il a cette capacité à se mettre à la place de l'utilisateur et à anticiper ses besoins.

L'historique d'annulation et les raccourcis clavier, ça peut paraître des détails, mais c'est exactement le genre de fonctionnalités qui rendent un outil agréable au quotidien. Aurélien a bien compris que l'expérience utilisateur, c'est autant dans ces petits détails

que dans les grosses fonctionnalités.

Ce qui était intéressant dans notre répartition, c'est qu'elle n'était pas figée. Chacun avait sa spécialité, mais on travaillait souvent ensemble sur les problèmes complexes. Vincent nous aidait pour optimiser les performances, Leonardo pour l'architecture, Nabil pour la robustesse, Aurélien pour l'ergonomie. Cette collaboration croisée a enrichi le projet et nos compétences individuelles.

3

Développement par phases - Qui a fait quoi

3.1 Phase 1.0 - Fondations (Février 2025)

3.1.1 Vincent : Pinceau, gomme, seau de remplissage, pipette

Commencer par les outils de dessin, c'était logique mais aussi intimidant. Ces outils, c'est ce que l'utilisateur va manipuler directement, alors si ils sont pas au top, tout le reste n'a plus d'importance. J'avais conscience de porter une grosse responsabilité pour l'expérience utilisateur finale.

Le pinceau, évidemment, c'était la priorité absolue. Mais quand on y réfléchit, qu'est-ce qui fait un bon pinceau numérique ? Il faut qu'il soit réactif, qu'il donne un trait fluide même quand on dessine vite, qu'il gère bien les différentes tailles et opacités... En creusant, j'ai réalisé que c'était beaucoup plus complexe que ce qu'il paraît.

Le problème principal, c'est l'interpolation entre les points de mouvement de la souris. Quand on bouge rapidement, les événements de souris sont espacés, et si on ne fait que dessiner des points isolés, on obtient un trait en pointillés pas terrible. Il faut interpoler intelligemment pour créer un trait continu et naturel.

J'ai commencé par implémenter l'algorithme de Bresenham pour tracer des lignes entre les points consécutifs. C'est un classique de l'infographie, mais il a fallu l'adapter pour gérer l'opacité et l'antialiasing. Puis j'ai ajouté un système de "brush stamps" pour éviter les discontinuités visuelles quand on peint lentement.

La gestion de la taille du pinceau était un autre défi. Pour les petites tailles, pas de problème, mais quand on arrive à des pinceaux de 50-100 pixels de diamètre, les calculs deviennent coûteux. J'ai mis en place un système de précalcul des masques circulaires avec antialiasing, ce qui améliore nettement les performances sur les gros pinceaux.

L'outil gomme partage beaucoup de code avec le pinceau, mais la logique de l'effacement est différente. Au lieu de composer la couleur avec ce qui existe déjà, il faut

remplacer les pixels par de la transparence. Ça paraît simple, mais bien gérer les transitions et l'antialiasing avec la transparence, c'est subtil.

Le seau de remplissage était un bon exercice d'algorithmique. J'ai implémenté un flood fill avec une tolérance de couleur configurable. L'algorithme de base est assez simple, mais l'optimiser pour de grandes zones tout en évitant les débordements de pile, ça demande de la réflexion. J'ai fini par utiliser une approche par pile plutôt que récursive, et des optimisations par scan lines.

La pipette, c'est l'outil le plus simple techniquement, mais il faut qu'elle soit super responsive. L'utilisateur s'attend à voir immédiatement la couleur sous son curseur. J'ai mis en place un système d'aperçu temps réel qui marche bien, avec intégration au système de couleurs primaire/secondaire selon le bouton de souris.

L'intégration de tous ces outils dans l'interface egui était un challenge en soi. Il fallait créer une palette d'outils intuitive, gérer la sélection de l'outil actif, les paramètres de taille et d'opacité... Et tout ça en gardant une interface clean et réactive.

Ce qui m'a le plus appris dans cette phase, c'est l'importance de l'optimisation. Un algorithme qui marche, c'est bien, mais un algorithme qui marche vite et reste fluide même dans les cas limites, c'est mieux. J'ai passé pas mal de temps à profiler mon code et à identifier les goulots d'étranglement.

3.1.2 Leonardo : Structure du canvas et transparence

Concevoir la structure de données qui allait porter tout Rustique, c'était à la fois excitant et stressant. Toutes les fonctionnalités de l'application allaient reposer sur ces fondations, alors il fallait que ce soit solide et bien pensé dès le départ.

J'ai commencé par me documenter sur les différentes approches possibles. Représentation par tiles, stockage compressé, structures hiérarchiques... Il y avait plein d'options sophistiquées, mais j'ai finalement opté pour quelque chose de plus simple et prévisible : un vecteur linéaire de pixels RGBA 32 bits, avec accès indexé bidimensionnel.

Cette approche a plusieurs avantages. D'abord, c'est simple à comprendre et à débugger. Ensuite, l'accès aux pixels est prévisible et optimisé par le cache CPU. Enfin, ça s'interface bien avec les bibliothèques existantes pour l'affichage et l'export. Parfois, la solution la plus directe est la meilleure.

La gestion de la transparence était cruciale dès cette phase. J'ai implémenté l'alpha blending standard avec la formule classique de Porter-Duff. Ça peut paraître technique, mais en gros, ça définit comment on compose deux couleurs dont l'une peut être partiellement transparente. C'est la base de tout compositing en infographie.

Un point important était de préserver la transparence totale. Quand un pixel est

complètement transparent (alpha = 0), il faut que ça reste le cas même après des opérations de composition. Ça paraît évident, mais mal implémenté, ça peut créer des artefacts visuels subtils.

J'ai aussi pensé dès le début au système de coordonnées. Origin en haut à gauche, axe Y vers le bas, coordonnées entières pour les pixels... Il fallait établir des conventions claires et s'y tenir. Ça évite les erreurs de mapping et facilite l'intégration avec les outils développés par les autres.

L'optimisation des accès mémoire était importante. J'ai organisé les données pour favoriser la localité spatiale : les pixels proches dans l'image sont proches en mémoire. Ça améliore les performances du cache CPU lors des opérations de dessin, surtout avec des pinceaux de grande taille.

Même si les calques n'étaient pas prévus pour cette phase, j'ai conçu l'interface du canvas de façon à faciliter l'extension future. Des méthodes abstraites pour l'accès aux pixels, une séparation claire entre les données et les opérations... Cette anticipation s'est révélée précieuse pour la Phase 2.0.

L'intégration avec egui pour l'affichage demandait un pont efficace entre notre représentation interne et les formats attendus par le framework. J'ai optimisé cette conversion pour minimiser les copies de données, crucial pour maintenir la fluidité de l'affichage.

Une des choses les plus intéressantes de cette phase, c'était de voir comment les choix de bas niveau influencent toute l'architecture. Une structure de données bien pensée facilite tout le développement ultérieur, alors qu'une mauvaise décision peut nous handicaper pendant tout le projet.

3.1.3 Nabil : Zoom, navigation et sauvegarde PNG

S'occuper du zoom et de la navigation, au début je pensais que c'était secondaire comparé aux outils de dessin. Mais j'ai vite réalisé que ces fonctionnalités conditionnent complètement l'expérience utilisateur. Un zoom qui lag ou une navigation peu intuitive, et l'application devient inutilisable.

Le système de zoom était plus délicat qu'il n'y paraît. Il ne suffit pas de multiplier la taille d'affichage par un facteur, il faut aussi gérer l'interpolation pour éviter les artefacts visuels. Pour les zooms avant, j'ai utilisé une interpolation bilinéaire qui donne des résultats propres. Pour les zooms arrière, un échantillonnage intelligent évite les effets de moiré.

J'ai implémenté une gamme de zoom de 10

La navigation dans les grands canvas posait des défis intéressants. Il faut pouvoir se déplacer rapidement d'une zone à l'autre, mais aussi naviguer avec précision pour les

ajustements fins. J'ai mis en place un système qui combine défilement avec momentum et maintien-glissement pour le déplacement direct.

L'optimisation du rendu était cruciale. Afficher un canvas de 4000×4000 pixels à chaque frame, même en partie, ça peut vite saturer les performances. J'ai implémenté un système de viewport intelligent qui ne calcule et n'affiche que la portion visible, avec un petit buffer pour les déplacements fluides.

Les contrôles de navigation devaient être intuitifs et suivre les conventions établies. Molette de souris pour le zoom, maintien et glissement pour le déplacement, indicateurs visuels pour s'orienter... J'ai testé différentes approches avant de trouver celle qui semblait la plus naturelle.

La sauvegarde PNG était notre première fonctionnalité de persistance, et c'était crucial qu'elle soit fiable. J'ai utilisé la crate 'image' qui gère bien le format, mais j'ai ajouté des optimisations spécifiques : compression adaptative selon le contenu, préservation de la transparence, gestion des métadonnées.

Pour les grandes images, la sauvegarde pouvait devenir problématique niveau mémoire. J'ai implémenté une approche streaming qui évite de charger l'intégralité de l'image en mémoire simultanément. Ça permet de gérer des projets de plusieurs gigabytes sans problème.

La gestion d'erreurs était importante pour cette fonctionnalité critique. Espace disque insuffisant, permissions manquantes, fichiers corrompus... Il fallait gérer tous ces cas de façon gracieuse et informative pour l'utilisateur. Une sauvegarde qui échoue silencieusement, c'est le cauchemar de tout créateur.

L'intégration avec les dialogues système natifs pour le choix du fichier améliore l'expérience utilisateur. Plutôt que de réinventer une interface de navigation de fichiers, mieux vaut s'appuyer sur ce que l'utilisateur connaît déjà. Ça donne aussi une impression de qualité et d'intégration.

3.1.4 Aurélien : Interface de base et sélecteur de couleur

Créer l'interface utilisateur d'une application de création, c'est un défi particulier. Il faut que ce soit à la fois fonctionnel pour les développeurs et agréable pour les artistes. Et entre les deux, il y a parfois un fossé assez large !

J'ai commencé par étudier l'ergonomie des logiciels existants. Comment ils organisent l'espace, où ils placent les outils, comment ils gèrent les panneaux... GIMP, Photoshop, Krita ont chacun leur approche, et j'ai essayé de prendre le meilleur de chaque tout en adaptant aux contraintes d'egui.

L'organisation générale s'est imposée naturellement : barre d'outils à gauche, canvas

au centre, panneaux contextuels à droite. Cette disposition classique a l'avantage d'être familière aux utilisateurs, ce qui réduit la courbe d'apprentissage. Parfois, il vaut mieux suivre les conventions que réinventer.

Le sélecteur de couleur était un gros morceau. Il fallait combiner precision et rapidité d'utilisation. J'ai opté pour un sélecteur HSV avec roue de teinte et triangle de saturation/luminosité. C'est intuitif pour choisir une couleur, et ça permet des ajustements fins avec les champs numériques.

Le système de couleurs primaire/secondaire suit les conventions des logiciels de référence. Clic gauche pour utiliser la couleur primaire, clic droit pour la secondaire, raccourci clavier pour basculer... Ces petits détails font qu'un utilisateur expérimenté peut être productif immédiatement.

La palette de couleurs sauvegardées était une fonctionnalité que j'ai ajoutée après avoir testé l'interface. Quand on travaille sur un projet avec une palette cohérente, c'est très pratique de pouvoir retrouver rapidement ses couleurs. J'ai fait ça simple : clic pour utiliser, clic long pour sauvegarder.

L'intégration avec egui demandait de s'adapter aux spécificités du framework. egui est "immédiat", ce qui veut dire qu'on redessine l'interface à chaque frame en fonction de l'état de l'application. C'est différent des approches traditionnelles, mais finalement assez libérateur une fois qu'on s'y habitue.

L'optimisation de la responsivité interface était importante. Même si on redessine tout à chaque frame, il faut que ça reste fluide. J'ai mis en place des mises à jour différées pour les opérations coûteuses et des feedbacks visuels immédiats pour maintenir l'impression de réactivité.

La préparation pour l'extensibilité future guidait mes choix dès cette phase. Comment ajouter facilement de nouveaux panneaux ? Comment gérer les préférences utilisateur ? Comment permettre la personnalisation ? J'ai essayé de poser des bases flexibles sans tomber dans la sur-ingénierie.

Ce qui m'a le plus appris, c'est l'importance de tester régulièrement avec de vrais utilisateurs. Ce qui paraît évident quand on développe ne l'est pas forcément pour quelqu'un qui découvre l'interface. J'ai fait tester par des potes et ça m'a aidé à corriger plein de petits problèmes d'ergonomie.

3.2 Phase 2.0 - Évolution majeure (Avril 2025)

3.2.1 Vincent : Menu principal et outil ligne

Arriver à la Phase 2.0, on avait déjà quelque chose qui marchait, mais qui ressemblait encore à un prototype. Il nous fallait professionnaliser l'expérience utilisateur, et le menu principal était un élément clé pour ça. Plus question de lancer l'application directement sur un canvas vide !

Concevoir un bon menu principal, c'est trouver l'équilibre entre simplicité et fonctionnalité. L'utilisateur doit pouvoir rapidement créer un nouveau projet ou ouvrir un existant, mais aussi configurer précisément ses paramètres. J'ai étudié les menus de GIMP, Photoshop et autres pour voir ce qui marche bien.

L'idée centrale était de centraliser toutes les actions de démarrage : nouveau projet avec dimensions personnalisables, ouverture de fichiers existants, projets récents... Et surtout, éviter de se retrouver avec un canvas aux dimensions bizarres qu'il faut redimensionner après. Cette configuration en amont facilite beaucoup le workflow.

L'implémentation du menu principal m'a fait découvrir les subtilités de la gestion d'état avec egui. Il faut orchestrer les transitions entre les différents modes (menu, configuration, édition) de façon clean. J'ai utilisé une machine à états simple mais efficace pour gérer ça proprement.

L'outil ligne était une extension naturelle de notre panoplie, mais qui posait des défis intéressants. L'approche en deux clics avec aperçu dynamique semblait évidente, mais bien l'implémenter demandait de réfléchir à l'architecture des outils.

Le gros défi était de gérer l'aperçu en temps réel sans impacter les performances ni compliquer l'intégration avec le reste. J'ai développé un système d'overlay temporaire qui affiche la ligne en cours sans modifier le canvas principal. Ça préserve l'historique d'annulation et permet l'annulation simple par Échap.

L'algorithme de rasterisation de la ligne réutilise le code du pinceau pour garantir une cohérence visuelle. Même taille, même opacité, même rendu... L'utilisateur ne doit pas voir de différence entre une ligne tracée avec l'outil ligne et un trait très droit au pinceau.

L'optimisation de l'aperçu ligne était importante pour maintenir la fluidité. Même pour de très longues lignes traversant tout le canvas, le mouvement de souris doit rester responsive. J'ai implémenté des techniques de clipping intelligent et d'optimisation des calculs pour ça.

Ce qui était intéressant avec l'outil ligne, c'est qu'il m'a fait réfléchir à l'architecture

générale des outils. Comment structurer le code pour que l'ajout de nouveaux outils soit simple ? Comment partager du code entre outils similaires ? Cette réflexion a posé les bases pour les outils de formes géométriques de la Phase 3.0.

L'intégration dans l'interface existante devait être seamless. L'outil ligne s'active comme les autres, utilise les mêmes paramètres, suit les mêmes conventions... L'objectif était qu'un utilisateur découvre l'outil naturellement sans avoir besoin d'apprendre de nouveaux comportements.

3.2.2 Leonardo : Système de calques complet

Implémenter les calques, c'était vraiment le gros chantier de cette phase. On savait que ça allait changer beaucoup de choses dans l'architecture, mais on sous-estimait à quel point ça touchait à presque tous les aspects de l'application.

La première étape était de repenser la structure de données. Plutôt qu'un canvas unique, il fallait maintenant gérer une pile de calques, chacun avec ses propriétés (nom, visibilité, opacité). J'ai conçu ça comme une collection ordonnée où chaque calque est essentiellement un mini-canvas avec des métadonnées.

L'algorithme de composition était le cœur technique du système. Pour chaque pixel de l'image finale, il faut combiner les contributions de tous les calques visibles en tenant compte de leur ordre et de leurs propriétés. Ça sonne simple, mais bien optimiser ça pour que ça reste fluide même avec 10-15 calques, c'est un autre niveau.

J'ai implémenté un système de dirty regions pour éviter de recalculer toute l'image à chaque modification. L'idée est de traquer précisément les zones modifiées et de ne recomposer que ce qui est nécessaire. Ça fait la différence entre une application fluide et une qui rame.

L'interface de gestion des calques devait être à la fois simple et puissante. Panneau latéral avec liste des calques, indicateurs visuels pour la visibilité et la sélection, contrôles pour l'opacité... Et surtout, possibilité de réorganiser par glisser-déposer. Cette dernière fonctionnalité était techniquement délicate avec egui, mais vraiment importante pour l'utilisabilité.

La gestion de la mémoire devenait critique avec les calques. Chaque calque occupe potentiellement plusieurs mégaoctets, et avec des projets complexes, on peut vite arriver à des centaines de mégaoctets en RAM. J'ai optimisé la représentation interne et ajouté des stratégies de compression pour les calques majoritairement transparents.

L'intégration avec les outils existants était un défi majeur. Chaque outil devait maintenant savoir sur quel calque il opère. J'ai développé une abstraction qui fait que les outils voient toujours un "canvas actuel" qui correspond en fait au calque sélectionné.

Ça préservait leur logique tout en les intégrant dans le système multi-calques.

Le système de nommage automatique et la personnalisation des noms améliorent beaucoup l'organisation des projets complexes. Les calques reçoivent des noms par défaut ("Calque 1", "Calque 2"...), mais on peut les renommer pour des projets plus organisés. C'est le genre de détail qui fait qu'on peut gérer des projets sérieux.

La préparation pour les modes de fusion était déjà dans ma tête pendant cette phase. J'ai conçu le pipeline de composition de façon extensible, avec des points d'accroche pour ajouter facilement de nouveaux modes. Cette anticipation a facilité le développement de la Phase 3.0.

Ce qui m'a le plus marqué dans cette phase, c'est la complexité des interactions entre tous les systèmes. Les calques touchent au rendu, aux outils, à l'historique, à la sauvegarde... Il faut vraiment penser architecture système pour que tout s'intègre harmonieusement.

3.2.3 Nabil : Multi-formats et format .rustiq

Étendre le support de fichiers au-delà du PNG, c'était un passage obligé pour faire de Rustique un outil vraiment utilisable. Mais ça s'est révélé plus complexe que prévu, chaque format ayant ses spécificités et ses pièges.

J'ai commencé par concevoir une architecture extensible pour gérer tous ces formats. Un système de détection automatique basé sur l'extension, avec fallback sur l'analyse d'en-tête pour les cas ambigus. Ça permet de gérer proprement les fichiers mal nommés ou les formats moins courants.

JPEG était prioritaire vu sa popularité, mais il pose le problème de l'absence de transparence native. J'ai implémenté un système de conversion intelligent qui préserve l'apparence visuelle lors de l'export, avec avertissement pour l'utilisateur sur les pertes potentielles. C'est important de prévenir plutôt que de surprendre.

WebP était intéressant à ajouter comme format moderne. Il combine bonne compression et support de la transparence, ce qui en fait un bon compromis pour beaucoup d'usages. L'implémentation gère les variantes avec et sans transparence, et optimise automatiquement selon le contenu.

Le gros morceau, c'était le format natif .rustiq. L'objectif était de pouvoir sauvegarder exactement l'état complet d'un projet : tous les calques avec leurs propriétés, la palette de couleurs, les paramètres d'outils... Bref, tout ce qui fait qu'on peut reprendre le travail exactement où on l'avait laissé.

J'ai opté pour une structure JSON pour les métadonnées, ce qui rend le format lisible et facilement extensible. Les données pixellaires sont encodées séparément avec

compression optimisée. C'est peut-être pas le plus compact possible, mais c'est robuste et évolutif.

Le système de versioning du format .rustiq était crucial pour l'évolution future. Chaque fichier contient sa version, et le lecteur peut s'adapter ou faire de la migration automatique. Ça préserve l'investissement des utilisateurs dans leurs projets même quand on fait évoluer le format.

L'optimisation de la compression s'adaptait au contenu. Les zones majoritairement transparentes utilisent une compression RLE simple et efficace. Les zones complexes bénéficient d'une compression sans perte plus sophistiquée. Cette approche hybride donne de bons résultats sur des projets variés.

L'intégration avec le menu principal pour l'ouverture de fichiers devait être intelligente. Détecter si c'est un projet .rustiq complet ou une simple image, et adapter le comportement en conséquence. Pour un .rustiq, on restaure tout l'état. Pour une image, on crée un nouveau projet avec l'image comme calque de base.

La gestion d'erreurs robuste était essentielle pour ces fonctionnalités critiques. Fichiers corrompus, formats non supportés, problèmes de droits... Il fallait que l'application reste stable et informe clairement l'utilisateur. Perdre du travail à cause d'un bug de sauvegarde, c'est impardonnable.

Ce qui m'a le plus appris, c'est l'importance des standards et de la compatibilité. Chaque format a son histoire, ses subtilités, ses cas particuliers. Bien les implémenter demande de la documentation, des tests, et beaucoup d'attention aux détails.

3.2.4 Aurélien : Annuler/refaire et raccourcis clavier

Implémenter un système d'annulation robuste, c'était crucial pour faire de Rustique un outil professionnel. Mais avec l'arrivée des calques, ça devenait beaucoup plus complexe qu'un simple "undo" sur un canvas unique.

J'ai commencé par réfléchir à l'architecture générale. Il fallait un système qui puisse capturer l'état de n'importe quelle opération, la stocker efficacement, et la restaurer proprement. J'ai opté pour une approche par delta qui n'enregistre que les changements, plutôt que des snapshots complets.

L'approche par commandes me permettait d'encapsuler chaque opération d'outil avec son contexte complet : quel calque, quelle zone modifiée, quelles données avant/après... Chaque outil génère maintenant une commande d'historique qu'il peut exécuter et annuler de façon autonome.

La gestion de la mémoire était critique. On ne peut pas garder indéfiniment toutes les opérations, il faut limiter intelligemment. J'ai implémenté un système adaptatif

qui conserve plus d'opérations simples que d'opérations complexes, selon la mémoire disponible. L'objectif était de maximiser l'utilité de l'historique.

L'interface visuelle du système était importante. Boutons undo/redo avec indicateurs d'état, feedback visuel quand on atteint les limites... Et surtout, integration parfaite avec les raccourcis clavier standards que tout le monde connaît : Ctrl+Z et Ctrl+Y.

Les raccourcis clavier, justement, c'était l'occasion de vraiment améliorer l'expérience pour les utilisateurs expérimentés. J'ai implémenté tous les classiques : Ctrl+N pour nouveau, Ctrl+O pour ouvrir, Ctrl+S pour sauvegarder... Plus les raccourcis spécifiques aux outils de création.

Un point important était la gestion des contextes. Certains raccourcis ne doivent marcher que dans certaines situations. Par exemple, les raccourcis d'outils seulement quand on est en mode édition, pas dans les menus. Cette logique contextuelle évite les comportements surprenants.

L'optimisation du comportement de zoom était une amélioration ergonomique importante. Plutôt que de zoomer sur la position du curseur (qui peut créer des déplacements inattendus), j'ai centré le zoom sur le milieu de la vue. C'est plus prévisible pour naviguer dans de grands projets.

Le système de gestion des couleurs avec palette sauvegardée enrichissait les possibilités créatives. Interface simple pour sauvegarder et récupérer des couleurs, persistance dans le format .rustiq... Cette fonctionnalité améliore nettement l'efficacité pour les projets avec des palettes cohérentes.

Ce qui était interessant avec cette phase, c'est qu'elle touchait beaucoup aux détails d'experience utilisateur. L'annulation qui marche bien, les raccourcis qui tombent sous les doigts... C'est moins visible que les nouvelles fonctionnalités, mais ça transforme l'usage quotidien de l'application.

3.3 Phase 3.0 - Finalisation (Mai 2025)

3.3.1 Vincent : Formes géométriques et outils de sélection

Arriver à la Phase 3.0 avec déjà un outil fonctionnel et assez complet, c'était motivant. Il restait à ajouter les fonctionnalités qui feraient vraiment la différence pour des usages créatifs sérieux. Les formes géométriques étaient en haut de la liste.

L'outil rectangle semblait simple au premier abord, mais il y avait plein de subtilités. Tracé par glisser-déposer avec aperçu temps réel, support des rectangles avec coins arrondis, options remplissage ou contour seul... Il fallait que ce soit flexible sans devenir

compliqué à utiliser.

L'algorithme de rasterisation des rectangles devait être optimisé pour les grandes formes. Plutôt que de calculer pixel par pixel, j'ai utilisé une approche par scan lines qui minimise les calculs. Pour les rectangles qui couvrent une bonne partie du canvas, ça fait une différence notable sur les performances.

L'outil ellipse était plus délicat algorithmiquement. Maintenir une qualité visuelle constante pour toutes les proportions d'ellipse demande un algorithme robuste. J'ai utilisé une approche par équation implicite avec antialiasing soigné pour avoir des contours lisses à tous les niveaux de zoom.

Le système d'aperçu pour les formes réutilisait et généralisait ce que j'avais fait pour l'outil ligne. Cette cohérence dans l'interface améliore l'apprentissage : une fois qu'on sait utiliser un outil géométrique, on sait utiliser les autres. C'est le genre de détail d'ergonomie qui compte.

Les outils de sélection introduisaient un nouveau paradigme dans Rustique. Jusqu'ici, on agissait directement sur les pixels. Là, il fallait d'abord définir une zone d'intérêt, puis potentiellement faire des opérations dessus. C'était une extension logique mais qui changeait les patterns d'interaction.

J'ai commencé par la sélection rectangulaire avec support des modificateurs standards : Shift pour ajouter à la sélection, Ctrl pour soustraire, Alt pour l'intersection... Ces conventions sont établies dans tous les logiciels de graphisme, il fallait les respecter pour éviter de dérouter les utilisateurs.

La visualisation des zones sélectionnées utilisait le pattern classique des "fourmis marchantes" - cette bordure animée qu'on voit partout. C'est universellement reconnu et ça indique clairement ce qui est sélectionné. J'ai implémenté ça avec une animation fluide qui reste discrète.

L'intégration avec l'historique d'annulation et le système de calques devait être seamless. Créer une sélection génère une commande d'historique, les opérations sur la sélection s'appliquent au calque actif... Tout ça en gardant un comportement prévisible et cohérent avec le reste.

Ce qui était gratifiant avec ces outils, c'est qu'ils ouvraient vraiment de nouvelles possibilités créatives. Pouvoir créer rapidement des formes géométriques précises, définir des zones de travail... Ça transformait Rustique d'un outil de peinture libre en quelque chose de plus polyvalent.

3.3.2 Leonardo : Modes de fusion et contrôles d'opacité

Implémenter les modes de fusion, c'était l'étape qui allait vraiment faire passer Rustique au niveau supérieur. Ces modes changent complètement les possibilités créatives en permettant des interactions sophistiquées entre calques.

J'ai commencé par étudier la théorie derrière les modes de fusion. Chaque mode correspond à une formule mathématique qui définit comment combiner deux couleurs. Multiply, Screen, Overlay... Ces noms correspondent à des algorithmes précis qu'il faut implémenter correctement pour avoir des résultats cohérents avec les standards de l'industrie.

L'architecture modulaire que j'avais prévue dans les phases précédentes facilitait l'ajout de nouveaux modes. Chaque mode est implémenté comme une fonction pure qui prend deux pixels et retourne le résultat composé. Cette approche permet aussi d'optimiser chaque mode individuellement.

Le mode Multiply était un bon début, techniquement simple mais visuellement efficace. Il assombrit l'image en multipliant les composantes couleur, ce qui donne des effets d'ombre et de profondeur très naturels. L'implémentation devait gérer correctement la transparence et éviter les débordements numériques.

Screen complète naturellement Multiply en faisant l'effet inverse : il éclaircit l'image. Mathématiquement, c'est l'inverse du multiply inversé. Ces deux modes forment la base de beaucoup d'effets créatifs et sont indispensables dans tout logiciel sérieux.

Overlay et Soft Light étaient plus complexes avec leurs fonctions de transfert non-linéaires. Ces modes combinent multiplication et screen selon la luminosité du pixel de base, créant des effets de contraste très appréciés. L'implémentation demandait des approximations optimisées pour maintenir les performances.

Le système de contrôle d'opacité par calque s'intégrait naturellement avec les modes de fusion. Chaque calque a maintenant un slider d'opacité dans l'interface, et cette opacité interagit correctement avec le mode de fusion choisi. C'est la combinaison des deux qui donne le contrôle créatif total.

L'optimisation du rendu composite avec modes de fusion était cruciale. Ces calculs sont intensifs, surtout avec plusieurs calques et des modes complexes. J'ai implémenté un cache intelligent et utilisé des techniques de vectorisation pour maintenir la fluidité.

L'interface utilisateur pour les modes de fusion devait être à la fois complète et accessible. Menu dropdown avec prévisualisation temps réel, organisation par catégories (standard, obscurcissement, éclaircissement, contraste...), noms explicites... L'objectif était d'encourager l'expérimentation tout en restant utilisable.

Ce qui était excitant avec cette fonctionnalité, c'est qu'elle transformait vraiment

les possibilités créatives de Rustique. Voir des artistes commencer à utiliser ces modes pour créer des effets sophistiqués, ça validait tout le travail architectural des phases précédentes.

3.3.3 Nabil : Optimisations et exportation avancée

Arriver à la Phase 3.0, Rustique était déjà bien fonctionnel, mais il restait du travail pour vraiment fignoler les performances et ajouter les fonctionnalités qui manquaient pour un usage professionnel. L'optimisation et l'exportation avancée étaient mes priorités.

J'ai commencé par profiler l'application pour identifier les vrais goulots d'étranglement. Parfois on optimise les mauvaises choses ! Il s'est avéré que la compression du format .rustiq pouvait être nettement améliorée avec une approche plus intelligente selon le type de contenu.

L'optimisation de la compression .rustiq utilisait maintenant une analyse du contenu de chaque calque. Les zones majoritairement transparentes bénéficient d'une compression RLE très efficace. Les zones avec des dégradés utilisent des techniques adaptées. Les zones détaillées gardent une compression sans perte classique. Cette approche adaptative améliore nettement les ratios.

Le système d'exportation par lot était une demande qui revenait souvent. Pouvoir générer plusieurs formats simultanément avec des paramètres optimisés pour chaque usage : web (PNG optimisé), impression (TIFF haute qualité), archive (format sans perte)... J'ai créé une interface qui permet de configurer des presets d'export personnalisables.

L'optimisation de la gestion mémoire était importante avec des projets de plus en plus complexes. J'ai implémenté un système de monitoring interne qui détecte les allocations excessives et suggère des optimisations. Ça aide à maintenir la stabilité même sur des projets de plusieurs gigabytes.

Le cache de rendu intelligent était une grosse amélioration performance. En analysant les patterns d'utilisation, le système précharge les données fréquemment accédées et optimise les chemins critiques. Cette optimisation prédictive améliore nettement la réactivité perçue, surtout lors de navigation dans de gros projets.

L'exportation avec préservation de métadonnées enrichissait l'interopérabilité. Écriture de métadonnées EXIF et XMP pour les formats qui les supportent, traçabilité des paramètres de création... C'est important pour l'intégration dans des workflows professionnels où ces informations comptent.

Les algorithmes de transformation (redimensionnement, rotation) ont été optimi-

sés pour maintenir la qualité visuelle. Filtres d'interpolation adaptatifs qui choisissent automatiquement l'algorithme optimal selon le type de contenu et le facteur de transformation. Ça fait la différence entre un résultat amateur et professionnel.

Le système de prévisualisation avant export évite les mauvaises surprises. Aperçu temps réel avec les paramètres choisis, estimation de la taille de fichier, détection des pertes potentielles... Cette validation visuelle améliore l'efficacité du workflow de finalisation.

Ce qui était satisfaisant dans cette phase, c'est qu'on affinait vraiment l'outil pour qu'il soit agréable et efficace au quotidien. Ces optimisations, on ne les voit pas forcément immédiatement, mais elles font qu'on prend plaisir à utiliser l'application même sur des projets complexes.

3.3.4 Aurélien : Pinceaux avancés et site web

Pour la finalisation de Rustique, je voulais vraiment pousser les possibilités créatives avec des pinceaux plus sophistiqués. Et en parallèle, créer une vitrine web digne de ce nom pour présenter tout notre travail.

Les pinceaux avancés étendaient considérablement la palette créative. L'airbrush simule l'effet d'un aérographe avec accumulation progressive et diffusion gaussienne. C'est technique à implémenter proprement, mais ça ouvre des possibilités artistiques qu'on n'a pas avec un pinceau classique.

L'algorithme de l'airbrush gère l'accumulation temporelle pour créer des effets de saturation réalistes. Plus on reste au même endroit, plus la "peinture" s'accumule. C'est subtil mais ça donne des résultats beaucoup plus naturels qu'un simple pinceau flou.

Les pinceaux texturés ajoutent une dimension tactile à la création numérique. En appliquant des patterns selon l'intensité du tracé, on peut simuler différentes matières : crayon, fusain, peinture épaisse... J'ai créé quelques textures de base et un système pour en importer de nouvelles.

Le système de tampons permet d'utiliser n'importe quelle image comme pinceau. C'est parfait pour créer des textures complexes ou des effets spécialisés. L'interface permet d'importer facilement des images et de les convertir en tampons réutilisables. Cette fonctionnalité transforme vraiment les possibilités créatives.

L'optimisation des performances pour ces pinceaux complexes demandait des techniques avancées. Précalcul des patterns, gestion intelligente du cache, optimisations vectorielles... Il fallait maintenir la fluidité même avec des pinceaux très détaillés sur de grandes tailles.

En parallèle, j'ai développé le site web de présentation. L'objectif était de créer une

vitrine professionnelle qui présente Rustique sous son meilleur jour. Design moderne responsive, captures d'écran attrayantes, démonstrations vidéo... Il fallait donner envie de télécharger et d'essayer.

La structure du site couvre tous les aspects : présentation générale, fonctionnalités détaillées, galerie de créations, documentation complète, téléchargement, équipe... C'est important d'avoir une ressource centralisée qui répond à toutes les questions qu'un utilisateur potentiel peut avoir.

Les démonstrations interactives sur le site permettent de voir Rustique en action sans le télécharger. Captures d'écran annotées, gif animés des fonctionnalités principales, vidéos de création... Cette approche multimédia aide à comprendre rapidement les capacités de l'application.

La documentation utilisateur intégrée au site facilite l'adoption. Guides d'utilisation pas à pas, conseils d'optimisation, FAQ... C'est crucial pour réduire la courbe d'apprentissage et aider les nouveaux utilisateurs à être productifs rapidement.

Ce qui était gratifiant avec cette phase finale, c'est de voir Rustique devenir vraiment complet. Les pinceaux avancés donnaient de nouvelles possibilités créatives, le site web offrait une présentation digne du travail accompli. On avait vraiment un produit fini dont on pouvait être fiers.

4

Défis techniques et solutions

4.1 Intégration des calques avec les outils existants

L'ajout du système de calques en Phase 2.0 nous a posé des difficultés qu'on n'avait pas complètement anticipées. C'est facile de dire "on va ajouter des calques", mais concrètement, ça chamboule toute l'architecture existante.

Le problème de base était simple à comprendre : nos outils fonctionnaient avec un seul canvas, et maintenant il fallait qu'ils comprennent qu'il y en a plusieurs empilés. Chaque outil devait apprendre à travailler uniquement sur le calque sélectionné, sans affecter les autres. Ça touchait non seulement la logique métier des outils, mais aussi leur intégration avec l'historique et le système de rendu.

Notre première approche était de modifier tous les outils un par un pour qu'ils soient "calque-aware". On a vite réalisé que c'était l'enfer : il fallait réécrire une grosse partie du code existant, avec tous les risques de bugs que ça implique. Et en plus, ça complexifiait le code de chaque outil.

La solution qu'on a trouvée était de créer une couche d'abstraction qui fait croire aux outils existants qu'ils travaillent toujours sur un canvas unique. Cette approche nous permettait de garder le code des outils intact tout en les intégrant dans le nouveau système multi-calques. C'est plus élégant et beaucoup moins risqué.

Concrètement, on a mis en place un système de délégation : quand un outil veut modifier des pixels, l'opération est automatiquement redirigée vers le bon calque. Cette redirection se fait au niveau le plus bas possible, au niveau de l'interface d'accès aux pixels, pour minimiser les modifications nécessaires dans le code des outils.

Mais la performance est devenue un gros souci avec les calques. À chaque modification sur un calque, il faut potentiellement recalculer l'image finale en combinant tous les calques visibles. Avec plusieurs calques et des images de bonne taille, ça peut vite devenir très lourd. L'application devenait moins réactive, ce qui est inacceptable pour un outil de création.

La solution qu'on a développée utilise un système de "dirty regions" - on ne recal-

cule que les zones qui ont vraiment changé. Quand on modifie une petite zone avec le pinceau, pas besoin de recomposer toute l'image. Ça fait une différence énorme sur les performances, surtout pour les modifications localisées qui sont la majorité des opérations de dessin.

Pour optimiser encore plus le rendu, on garde en cache les calques qui n'ont pas bougé. Au lieu de refaire les mêmes calculs de composition à chaque frame, on réutilise les résultats précédents quand possible. Le système s'adapte aussi selon la mémoire disponible pour éviter de faire exploser la RAM.

L'historique d'annulation a aussi dû être complètement revu. Avant, on pouvait se contenter de garder des snapshots de l'état global. Maintenant, chaque action doit se souvenir sur quel calque elle a eu lieu, ce qui permet d'annuler précisément même dans des projets avec plein de calques. Cette extension préserve la granularité fine de l'historique.

Pour éviter les bugs de concurrence quand plusieurs opérations touchent des calques différents, on a ajouté un système de verrous léger. Ça évite que les données se mélangent sans pour autant créer des blocages qui ralentiraient l'application. L'implémentation garantit la cohérence des données sans sacrifier les performances.

Ce qui nous a le plus appris dans ce défi, c'est l'importance de bien penser l'architecture dès le début. Quand on a voulu ajouter les calques, on s'est rendu compte que certains choix architecturaux initiaux nous limitaient. Heureusement, la couche d'abstraction nous a permis de contourner le problème, mais ça nous a sensibilisés à l'importance de concevoir pour l'extensibilité.

4.2 Performance du système d'annulation

Le système d'annulation, au début on pensait que ce serait simple : on sauvegarde l'état avant chaque action, et on le restaure en cas d'annulation. Sauf qu'en pratique, c'est beaucoup plus compliqué que ça, surtout quand on veut que ça reste performant.

Le gros problème, c'est qu'on ne peut pas se permettre de sauvegarder l'image complète à chaque action. Un canvas de 2000×2000 pixels en RGBA, ça fait 16 Mo. Avec 20 niveaux d'annulation, on arrive vite à plus de 300 Mo rien que pour l'historique. Et ça, c'est sans compter les calques ! Avec 5-6 calques, on explose complètement la mémoire.

Du coup, on a opté pour une approche par delta : on ne sauvegarde que ce qui change vraiment. Pour un coup de pinceau qui touche une petite zone, on garde juste les pixels modifiés dans cette zone, pas toute l'image. Ça réduit énormément l'espace nécessaire,

surtout pour les opérations localisées qui représentent la majorité des actions de dessin.

L'algorithme qu'on a développé analyse d'abord la zone modifiée pour identifier le plus petit rectangle qui contient tous les changements. Ensuite, il compresse ces données selon leur contenu. Pour les zones avec beaucoup de transparence, on utilise une compression RLE (Run Length Encoding) qui est simple et efficace. Pour les zones plus complexes, on utilise une compression sans perte plus sophistiquée.

Avec l'arrivée des calques, ça s'est encore compliqué. Il faut se souvenir sur quel calque chaque action a eu lieu, et conserver suffisamment d'informations pour bien remettre les choses en place. On a créé un système de "commandes" qui encapsulent toute l'information nécessaire : le calque cible, la zone affectée, les données avant/après...

Pour accélérer les opérations d'annulation, on évite de recalculer tout d'un coup. On utilise des techniques de "lazy evaluation" : la reconstruction complète de l'état n'est effectuée que quand c'est vraiment nécessaire. Et on garde des caches intermédiaires pour accélérer les opérations répétitives. Même quand on annule plusieurs actions d'affilée, ça reste réactif.

Le système s'adapte aussi intelligemment à la mémoire disponible. S'il y a peu de RAM, il garde moins d'historique pour les actions complexes et plus pour les actions simples. L'idée est d'optimiser l'utilité de l'historique selon les ressources qu'on a. Mieux vaut 30 annulations sur des petites actions que 5 sur des grosses.

On a aussi prévu les cas où ça se passe mal : mémoire insuffisante, actions trop volumineuses, données corrompues... Dans ces situations, l'application continue de fonctionner même si l'historique est réduit ou désactivé temporairement. Le principe c'est que c'est mieux de pouvoir continuer à peindre que de planter complètement.

Une chose qu'on a apprise, c'est l'importance de bien tester les cas limites. L'annulation, c'est le genre de fonctionnalité qui marche bien en temps normal, mais qui peut faire des choses bizarres dans des situations exceptionnelles. On a passé pas mal de temps à créer des scénarios de test pour s'assurer que ça reste robuste.

4.3 Gestion des formats de fichiers

Gérer tous les formats d'images standards, on pensait que ce serait straightforward grâce à la crate ‘image‘. Mais chaque format a ses particularités, ses limitations, et ses pièges qu'il faut connaître pour bien l'implémenter.

PNG supporte la transparence native, JPEG compresse bien les photos mais perd de la qualité, BMP est simple mais volumineux, TIFF est très flexible mais complexe, GIF limite les couleurs à 256, WebP est moderne mais moins supporté partout. Et notre

format .rustiq doit pouvoir sauvegarder absolument toute l'information d'un projet. Chaque format demande une approche spécifique.

On a créé un système modulaire où chaque format a son propre "codec" - son traducteur entre notre représentation interne et le format de fichier. Ça nous permet d'optimiser chacun séparément selon ses spécificités, tout en gardant une interface simple et uniforme pour le reste de l'application.

La transparence pose souvent problème lors des conversions. Quand on exporte vers un format qui ne la supporte pas nativement (comme JPEG), il faut bien gérer la conversion. Notre système prévient l'utilisateur des limitations et lui donne le choix de la couleur de fond pour la composition finale. C'est important de pas surprendre l'utilisateur avec des résultats inattendus.

Pour les gros fichiers, on a mis en place un système de lecture/écriture par chunks. Au lieu de tout charger en mémoire d'un coup, on traite le fichier petit bout par petit bout. Ça permet de gérer des images de plusieurs gigabytes sans faire exploser la RAM. Cette approche streaming évite aussi de bloquer l'interface utilisateur pendant les opérations longues.

Notre format .rustiq, c'est vraiment notre innovation principale. On utilise JSON pour stocker les métadonnées (liste des calques, propriétés, historique, préférences...), ce qui le rend facilement lisible et extensible. Les données pixellaires sont stockées séparément avec une compression optimisée. C'est peut-être pas le plus compact possible, mais c'est robuste et évolutif.

Le système de versioning du .rustiq était crucial dès le début. Chaque fichier contient sa version de format, et le lecteur peut s'adapter ou faire de la migration automatique vers les nouvelles versions. Cette approche préserve l'investissement des utilisateurs dans leurs projets existants même quand on fait évoluer le format.

Pour la compression du .rustiq, on adapte la technique selon le contenu de chaque calque. Les métadonnées JSON sont compressées avec un algorithme standard. Les zones d'image majoritairement transparentes utilisent une compression RLE très efficace. Les zones détaillées gardent une compression sans perte classique. Cette approche hybride donne de bons résultats sur tous types de projets.

La validation et la récupération d'erreurs, c'est crucial pour ces fonctionnalités qui touchent aux données utilisateur. On vérifie l'intégrité des fichiers, on détecte les corruptions, on gère les incompatibilités de version... En cas de problème, on essaie de récupérer le maximum d'informations plutôt que de tout abandonner. Perdre du travail créatif, c'est vraiment le pire qui puisse arriver.

4.4 Optimisation du rendu

Le rendu, c'est vraiment ce qui fait qu'une application graphique paraît rapide ou lente. Avec l'introduction des calques, des modes de fusion, et des projets de plus en plus complexes, maintenir la fluidité est devenu un défi constant.

Le principe de base qu'on applique partout : ne recalculer que ce qui a changé. Quand on modifie un pixel avec le pinceau, il faut identifier précisément quelle partie de l'image finale doit être recalculée. Avec les modes de fusion complexes, un petit changement sur un calque peut avoir des effets sur une zone plus large de l'image finale.

Notre système trace les modifications au pixel près pour savoir exactement quoi recalculer. On appelle ça les "dirty regions" - les zones sales qui ont besoin d'être nettoyées. L'algorithme regroupe intelligemment les zones proches pour éviter de faire plein de petits calculs séparés. C'est plus efficace de recomposer un grand rectangle que dix petits.

Chaque calque garde son rendu en cache, et on cache aussi les combinaisons fréquentes de calques. Comme ça, quand on navigue dans l'image ou qu'on change des paramètres d'affichage (zoom, position), on évite de refaire les mêmes calculs de composition. Ce cache hiérarchique fait une différence énorme sur la réactivité.

Pour les modes de fusion qui sont gourmands en calcul (comme Overlay ou Soft Light), on utilise toutes les optimisations possibles du processeur. Les opérations sur les pixels sont regroupées et vectorisées pour que le compilateur puisse utiliser les instructions SIMD spécialisées du CPU. Sur les gros projets, ça peut doubler ou tripler les performances.

Un truc intelligent qu'on a ajouté : selon le niveau de zoom et la vitesse de navigation, on adapte automatiquement la qualité du rendu. Quand on fait défiler rapidement ou qu'on zoome vite, on utilise une version simplifiée pour garder la fluidité. Dès qu'on s'arrête, on repasse en haute qualité. L'utilisateur ne voit pas la différence, mais ça améliore nettement la réactivité perçue.

Côté gestion mémoire, on réutilise au maximum les buffers temporaires au lieu d'en allouer de nouveaux à chaque frame. Ça évite de surcharger le garbage collector et maintient des performances plus stables. On maintient aussi un pool de buffers de différentes tailles qui s'adapte aux besoins du projet.

Pour l'affichage final, on regroupe les opérations et on utilise des buffers persistants pour éviter trop d'échanges entre CPU et carte graphique. C'est particulièrement important pour les outils comme le pinceau qui ont besoin d'un retour visuel immédiat. Chaque milliseconde de latence se ressent directement dans l'expérience de dessin.

Une optimisation spécifique qu'on a développée concerne le niveau de détail adaptatif. Quand on est très dézoomé, pas besoin de calculer tous les détails pixel par pixel. On peut utiliser des approximations et une résolution réduite qui donnent le même résultat visuel pour beaucoup moins de calculs.

4.5 Gestion de la concurrence et de la stabilité

Même si Rustique n'est pas une application massivement parallèle, il y a quand même des situations où différentes parties du code peuvent interagir de façon inattendue. Et avec Rust, il faut gérer ça proprement pour éviter les data races et autres joyeusetés.

Le principal point de concurrence dans notre architecture, c'est l'interaction entre le système de rendu qui lit les données des calques et les outils qui les modifient. Avec egui qui redessine l'interface à chaque frame, on peut avoir des situations où un outil modifie un calque pendant qu'un autre thread essaie de le lire pour l'affichage.

Rust nous aide énormément ici avec son système de ownership et de borrowing. Le compilateur nous empêche de faire des bêtises au niveau des accès mémoire. Mais il faut quand même concevoir l'architecture pour éviter les blocages et maintenir la réactivité.

Notre solution utilise un système de verrous à granularité fine. Plutôt qu'un gros verrou global qui bloquerait toute l'application, on a des verrous spécifiques pour chaque calque. Ça permet de dessiner sur un calque pendant qu'un autre est en cours de modification par un outil différent.

Pour la stabilité générale, on a mis l'accent sur la gestion robuste des erreurs. Avec Rust, on est forcés de gérer explicitement les cas d'erreur grâce au système de Result et Option. C'est parfois verbeux, mais ça évite les crashs inattendus qui plombent l'expérience utilisateur.

Un point important était de s'assurer que l'application reste stable même en cas de corruption de données ou de manque de mémoire. On a développé des stratégies de dégradation gracieuse : si l'historique ne peut plus être sauvegardé par manque de mémoire, on prévient l'utilisateur mais on continue à fonctionner. Si un calque est corrompu, on essaie de le récupérer partiellement plutôt que de faire planter tout le projet.

Les tests de stress nous ont aidés à identifier les cas limites. Projets avec 50 calques, canvas de 8000×8000 pixels, opérations d'annulation en boucle... Ce genre de scénarios extrêmes révèle les faiblesses qu'on ne voit pas en usage normal. Et c'est important de les corriger parce qu'il y a toujours des utilisateurs pour pousser les limites.

5

Récit de développement

5.1 Les réussites

5.1.1 Moments de satisfaction technique

Le développement de Rustique nous a donné plusieurs moments de pure satisfaction technique, ces instants où on se dit "ça y est, ça marche vraiment !". Ces victoires, même petites, nous ont portés dans les moments plus difficiles.

Le premier gros "wow", ça a été quand Vincent a réussi à faire marcher le pinceau avec un tracé vraiment fluide. On avait eu plusieurs tentatives qui donnaient des résultats corrects mais un peu saccadés. Et puis un jour, il nous montre sa nouvelle version avec l'interpolation optimisée, et là on a senti qu'on avait quelque chose de quality. Le tracé était smooth, réactif, naturel. C'était notre premier vrai outil utilisable.

L'implémentation réussie du système de calques reste probablement notre plus grande satisfaction technique collective. Après des semaines de galère avec l'architecture, voir enfin les calques fonctionner ensemble avec composition temps réel et gestion complète de la transparence... Leonardo était tout fier, et il avait raison ! Cette réussite validait nos choix architecturaux et ouvrait toutes les possibilités créatives qu'on avait imaginées.

La première démonstration du format .rustiq fonctionnel nous a tous marqués. Nabil sauvegarde un projet complexe avec plein de calques, ferme l'application, la rouvre, et boom : on retrouve exactement l'état précédent avec tous les détails préservés. Jusqu'ici, on ne pouvait sauvegarder qu'en PNG flat, donc perdre toute l'information de structure. Là, on avait vraiment créé quelque chose d'utile et durable.

L'optimisation du système d'annulation par Aurélien nous a scotchés. Au début, l'historique bouffait la mémoire et ralentissait tout. Après son système de delta compression, on a vu l'utilisation mémoire chuter drastiquement tout en gardant la même fonctionnalité. Et en bonus, les annulations étaient devenues plus rapides ! C'est le genre d'optimisation qui fait qu'on a confiance dans la solidité technique du projet.

Quand on a réussi notre premier projet artistique complexe entièrement créé avec Rustique, ça a été un moment fort. Une illustration multi-calques avec différents outils, des modes de fusion, des effets sophistiqués... Voir que notre outil pouvait servir à créer quelque chose de vraiment beau, ça validait concrètement tout notre travail. On n'était plus dans la technique pure, on était dans la création.

L'intégration des modes de fusion qui marchent du premier coup nous a surpris nous-mêmes. Leonardo avait bien préparé l'architecture dans les phases précédentes, et quand est venu le moment d'ajouter Multiply, Screen, Overlay... tout s'est emboîté naturellement. Voir les calques interagir exactement comme dans Photoshop, avec des résultats visuellement identiques, ça nous a convaincus qu'on avait atteint un niveau professionnel.

5.1.2 Fonctionnalités qui ont bien marché

Certaines fonctionnalités de Rustique ont dépassé nos attentes, soit parce qu'elles étaient plus simples à implémenter que prévu, soit parce qu'elles se sont révélées plus utiles qu'on pensait.

Le système de couleurs primaire/secondaire avec palette sauvegardée s'est révélé très apprécié lors de nos tests. Cette fonctionnalité, relativement simple techniquement, améliore vraiment l'efficacité du workflow créatif. Les gens adorent pouvoir retrouver leurs couleurs favorites entre les sessions. C'est le genre de détail qui paraît anecdotique mais qui fait qu'on prend plaisir à utiliser l'outil.

L'outil ligne avec aperçu temps réel a immédiatement trouvé sa place. Son comportement intuitif (clic-glisser avec preview, Échap pour annuler) et sa qualité de rendu en font un outil qu'on utilise naturellement. On s'est rendu compte qu'avoir des tracés droits précis, c'est plus important qu'on le pensait pour plein de types d'illustrations.

Le menu principal avec configuration du canvas a transformé l'expérience de première utilisation. Avant, on lançait l'application directement sur un canvas 800×600 un peu au hasard. Maintenant, on peut définir précisément ses dimensions dès le départ, ce qui est beaucoup plus professionnel. Cette interface donne immédiatement une impression plus sérieuse.

L'historique d'annulation fonctionne de façon tellement transparente qu'on l'oublie. C'est exactement ce qu'on veut pour ce genre de fonctionnalité : qu'elle marche parfaitement sans qu'on ait à y penser. Sa robustesse face aux opérations complexes et aux projets multi-calques en fait une fondation solide qu'on peut oublier.

La gestion multi-formats s'est révélée plus importante qu'on le pensait pour l'adoption. Pouvoir importer et exporter vers tous les formats standards, ça élimine complète-

ment les barrières à l'intégration dans des workflows existants. Les gens peuvent essayer Rustique sans se soucier de la compatibilité avec leurs autres outils.

L'interface de gestion des calques a trouvé le bon équilibre entre simplicité et puissance. Les débutants comprennent intuitivement comment ça marche, mais les fonctionnalités avancées (opacité, modes de fusion, réorganisation) restent facilement accessibles. Cette progressivité dans la complexité facilite l'apprentissage.

Le format .rustiq s'est avéré plus polyvalent qu'on l'imaginait. Au-delà de la simple sauvegarde de projets, il nous sert aussi pour les tests automatisés, la documentation des bugs, le partage de projets d'exemple... Sa structure extensible nous permet de l'adapter à plein d'usages qu'on n'avait pas prévus.

5.1.3 Retours utilisateurs positifs

Même si Rustique reste un projet étudiant, on a eu quelques retours d'utilisateurs externes qui nous ont fait plaisir et validé nos choix.

Les démos qu'on a faites en cours ont bien marché. Nos camarades de promo étaient impressionnés de voir qu'on avait créé quelque chose qui ressemble vraiment à un logiciel commercial. Plusieurs nous ont dit qu'ils ne pensaient pas qu'on pouvait arriver à ce niveau de polish dans un projet étudiant de S4.

Quelques-uns ont testé l'application sur leurs ordis et ont été surpris de la fluidité. Ils s'attendaient à ce que ça rame ou que ça plante, mais en fait ça marche plutôt bien. Un pote qui fait un peu de dessin digital dans son temps libre nous a dit que l'interface était plus intuitive que GIMP, ce qui nous a fait plaisir.

Le truc qui revient souvent dans les retours de nos camarades, c'est qu'ils trouvent ça impressionnant qu'on ait réussi à faire quelque chose d'aussi complet en Rust. Beaucoup galèrent encore avec les concepts de base du langage, alors voir une application graphique complète, ça les motive pour leurs propres projets.

Même ceux qui ne s'intéressent pas particulièrement au graphisme apprécient l'aspect technique du projet. Le système de calques, les optimisations de performance, l'architecture modulaire... ça leur donne des idées pour structurer leurs propres projets de façon plus propre.

Le site web qu'Aurélien a créé nous a aussi valu des compliments de la promo pour le professionnalisme de la présentation. C'est important parce que la première impression compte énormément. Un projet peut être techniquement excellent, s'il est mal présenté, personne ne va s'y intéresser.

5.1.4 Impact sur notre apprentissage

Au-delà des retours externes, ce projet nous a vraiment fait progresser individuellement et en tant qu'équipe. C'est le genre d'expérience qui marque et qui change notre approche du développement.

L'aspect le plus marquant, c'est d'avoir créé quelque chose de concret qu'on peut montrer et utiliser. C'est différent des TPs où on fait des exercices isolés. Là, on a un vrai produit avec de vrais utilisateurs (même si c'est nos camarades). Cette dimension concrète donne du sens à l'apprentissage technique.

Le défi de maintenir la motivation sur quatre mois nous a appris la persévérance. Il y a eu des moments difficiles, des bugs frustrants, des fonctionnalités qui ne marchaient pas comme prévu. Mais on a tenu bon et on est allés jusqu'au bout. Cette expérience nous montre qu'on peut mener à bien des projets ambitieux.

La collaboration technique intensive nous a fait progresser sur les soft skills qu'on néglige souvent dans les cours. Expliquer ses choix, négocier des compromis, gérer les conflits d'intégration... Ces compétences sont au moins aussi importantes que la technique pure pour travailler en équipe.

L'apprentissage de Rust en contexte réel nous a donné une maîtrise du langage qu'on n'aurait jamais eue avec des exercices théoriques. Quand on a un vrai problème à résoudre avec des contraintes réelles, on comprend mieux les subtilités et les bonnes pratiques du langage.

La gestion d'un projet long nous a sensibilisés à l'importance de l'architecture et de la maintenance. Ce qui marche bien pour une démo d'une semaine ne tient pas forcément sur quatre mois. Cette perspective à long terme change la façon dont on conçoit et structure notre code.

5.2 Les galères

5.2.1 Bugs difficiles à résoudre

Le développement de Rustique n'a pas été un long fleuve tranquille. On a eu notre lot de bugs vicieux qui nous ont fait arracher les cheveux et passer des nuits blanches.

Le bug le plus frustrant, c'était la corruption aléatoire de données lors de la sauvegarde de gros projets. Le problème se manifestait de façon complètement imprévisible : parfois ça marchait, parfois le fichier était illisible. Impossible à reproduire de façon fiable, ce qui rendait le debugging cauchemardesque. On a fini par découvrir que c'était

lié à un problème de gestion de lifetime dans la sérialisation des gros blocs de données pixellaires.

Un autre bug vicieux concernait une fuite mémoire progressive lors de l'utilisation intensive du pinceau. L'application devenait de plus en plus lente jusqu'à devenir inutilisable, mais seulement après 10-15 minutes d'utilisation intensive. L'investigation a révélé que les buffers temporaires de rendu n'étaient pas correctement libérés dans certains cas spécifiques. Heureusement, Rust nous a aidés à traquer ça avec ses outils d'analyse.

Le système d'annulation nous a donné des sueurs froides avec un comportement erratique sur certaines séquences d'opérations multi-calques. Parfois, l'annulation restaurait un état complètement différent de ce qu'on attendait. Ce bug était particulièrement sournois parce qu'il ne se manifestait que dans des conditions très spécifiques : il fallait faire une séquence précise d'actions sur des calques différents. On a fini par créer des scripts de test automatisés pour le reproduire de façon fiable.

Un problème de concurrence dans le système de rendu causait des artefacts visuels aléatoires. De temps en temps, on voyait des pixels qui apparaissaient aux mauvais endroits ou des couleurs bizarres. Ce genre de bug est un cauchemar parce qu'il dépend du timing précis des opérations. La résolution a nécessité une refonte du système de synchronisation entre le rendu et la modification des données.

La gestion de la transparence lors de la composition des calques présentait des incohérences mathématiques subtiles. Dans certaines configurations de calques avec des opacités partielles, on obtenait des résultats visuellement incorrects. Ces erreurs étaient difficiles à détecter parce qu'elles ne se voyaient qu'avec une analyse pixel par pixel. On a dû développer des outils de validation spécialisés pour les traquer.

5.2.2 Problèmes d'intégration

Développer à quatre en parallèle, ça crée forcément des problèmes d'intégration. Et sur un projet comme Rustique où tout est interconnecté, ces problèmes peuvent vite devenir bloquants.

Le plus gros casse-tête d'intégration concernait l'interface entre le système de calques de Leonardo et les outils de Vincent. Chaque outil avait été développé avec l'hypothèse d'un canvas unique, et leur adaptation au contexte multi-calques était plus complexe que prévu. Ça a créé des conflits de merge fréquents et nécessité plusieurs sessions de refactorisation collective assez tendues.

L'intégration du système d'annulation d'Aurélien avec les calques de Leonardo a posé des défis architecturaux inattendus. Les deux systèmes utilisaient des approches

différentes pour la gestion des modifications, ce qui créait des incompatibilités subtiles. Il a fallu plusieurs itérations pour trouver une interface commune qui satisfasse les deux contraintes.

La gestion des formats de fichiers de Nabil créait des dépendances circulaires avec le système de calques et l'interface utilisateur. Cette situation compliquait les tests unitaires et créait des problèmes de compilation intermittents. La résolution a nécessité une réorganisation des modules et l'introduction d'interfaces abstraites.

L'intégration des modes de fusion avec le pipeline de rendu existant a révélé des assumptions incompatibles sur l'ordre des opérations de composition. Cette situation nécessitait parfois des recalculs complets coûteux. La solution a impliqué une refactorisation du pipeline pour supporter nativement les opérations de fusion.

Un problème récurrent était la synchronisation des conventions de code entre les membres de l'équipe. Malgré rustfmt et clippy, on avait parfois des approches différentes pour résoudre le même type de problème. Ça créait des incohérences dans la base de code qu'il fallait régulièrement nettoyer lors des sessions de refactorisation commune.

5.2.3 Défis de performance

Les performances, c'est un sujet qu'on a sous-estimé au début. On pensait que Rust nous donnerait automatiquement de bonnes performances, mais en réalité il faut quand même optimiser intelligemment.

Le rendu des gros canvas avec plusieurs calques était notre principal goulot d'étranglement. Au-delà de 3000×3000 pixels avec 5-6 calques, l'application devenait perceptiblement plus lente. Le problème venait principalement du fait qu'on recomposait naïvement toute l'image à chaque modification. L'implementation du système de dirty regions a été cruciale pour résoudre ça.

La gestion mémoire avec des projets complexes nous a posé des défis inattendus. Même avec les optimisations de Rust, on pouvait vite arriver à plusieurs gigabytes de RAM utilisés sur de gros projets. Il a fallu développer des stratégies de compression intelligente et de gestion adaptative de l'historique pour rester dans des limites raisonnables.

Les modes de fusion complexes comme Overlay ou Soft Light étaient gourmands en calculs. Sur de gros calques, on sentait vraiment la différence de performance. L'optimisation par vectorisation et l'utilisation d'instructions SIMD ont été nécessaires pour maintenir la fluidité.

L'interface egui elle-même était parfois un goulot d'étranglement. Le modèle "immediate mode" qui redessine tout à chaque frame peut être coûteux si on n'optimise

pas bien. On a dû apprendre à minimiser les calculs dans la boucle de rendu et à utiliser efficacement le système de cache d'egui.

5.3 Ce qu'on a appris

5.3.1 Compétences techniques

Ce projet nous a vraiment fait progresser techniquement, bien au-delà de ce qu'on aurait appris avec des TPs classiques. Développer une application complète, c'est different de faire des exercices isolés.

Rust, on le maîtrise maintenant vraiment. Au début, on se battait avec le borrow checker et on trouvait le langage rigide. Maintenant, on comprend que cette rigidité nous protège et nous aide à écrire du code plus robuste. Les concepts avancés comme les lifetimes, les traits, les smart pointers... tout ça est devenu naturel à force de pratiquer sur du vrai code dans le cadre de ce projet obligatoire.

L'architecture logicielle, c'est quelque chose qu'on a appris sur le tas. Concevoir un système complexe qui reste maintenable et extensible, ce n'est pas évident. On a fait des erreurs, on a refactorisé, on a appris l'importance de bien séparer les responsabilités et de penser aux interfaces dès le début.

Les performances et l'optimisation, c'était nouveau pour la plupart d'entre nous. Apprendre à profiler du code, identifier les vrais goulots d'étranglement, optimiser sans casser... C'est un domaine fascinant où il faut combiner intuition technique et mesures précises.

La gestion des formats de données et la sérialisation nous ont appris l'importance de la compatibilité et de l'évolution des standards. Concevoir un format qui peut évoluer sans casser la rétrocompatibilité, c'est plus subtil qu'il n'y paraît.

Le debugging de systèmes complexes nous a appris à être méthodiques et à utiliser les bons outils. Quand on a un bug qui touche à l'interaction entre plusieurs modules, il faut une approche structurée pour l'isoler et le résoudre.

L'intégration continue et les tests automatisés, on a appris leur importance à nos dépens. Après quelques régressions évitables, on a mis en place des tests plus systématiques. C'est moins fun que d'ajouter des nouvelles features, mais c'est crucial pour la stabilité.

5.3.2 Travail d'équipe et gestion de projet

Travailler à quatre sur un projet technique de cette ampleur, ça nous a appris des choses qu'on n'apprend pas dans les cours théoriques de gestion de projet.

La communication technique est cruciale. Il faut savoir expliquer clairement ses choix, documenter ses interfaces, partager ses découvertes. Cette transparence facilite l'intégration et évite les malentendus qui peuvent coûter cher en temps de debug.

La gestion des conflits techniques nous a appris la négociation et le compromis. Face aux incompatibilités architecturales, il faut trouver des solutions qui préservent les objectifs de chacun tout en maintenant la cohérence globale. C'est autant humain que technique.

La planification adaptative, c'est un art qu'on a développé au fil du projet. Un planning trop rigide casse au premier imprévu, mais sans structure on part dans tous les sens. On a appris à définir des objectifs clairs tout en gardant la flexibilité sur les moyens.

La répartition du travail et la gestion des dépendances entre tâches, c'est plus complexe qu'il n'y paraît. Certaines fonctionnalités ne peuvent pas être développées en parallèle, il faut orchestrer les efforts pour éviter les blocages.

La revue de code collective nous a fait progresser individuellement et collectivement. C'est un excellent moyen d'apprendre les techniques des autres et de maintenir un niveau de qualité homogène dans la base de code.

5.3.3 Leçons sur le développement logiciel

Ce projet nous a donné une vision plus réaliste de ce qu'implique le développement d'une application complète. C'est différent des exercices académiques où le périmètre est bien défini et les contraintes connues.

L'importance de commencer simple et d'itérer. Notre approche en phases nous a évité de nous perdre dans la complexité. Mieux vaut quelque chose de simple qui marche qu'un système sophistiqué qui ne fonctionne qu'à moitié.

La gestion des cas limites et des erreurs, c'est souvent ce qui distingue un prototype d'un vrai logiciel. Il faut penser aux situations exceptionnelles, aux données corrompues, aux ressources insuffisantes... C'est moins gratifiant que d'ajouter des features, mais c'est essentiel.

L'expérience utilisateur, ce n'est pas juste l'interface graphique. C'est aussi les performances, la robustesse, la prévisibilité du comportement... Tous ces aspects contribuent à l'impression générale de qualité.

La documentation et les tests, on a tendance à les négliger quand on est dans le feu de l'action. Mais sur un projet qui dure plusieurs mois, ils deviennent vite indispensables. Sans ça, on perd du temps à se redemander comment fonctionne notre propre code.

L'optimisation prématurée est effectivement la source de nombreux maux, mais l'optimisation tardive aussi ! Il faut trouver le bon moment pour s'attaquer aux problèmes de performance. Trop tôt, on complexifie pour rien. Trop tard, l'architecture existante nous limite.

La refactorisation continue est essentielle sur un projet évolutif. Le code qui était parfait pour la Phase 1.0 peut devenir limitant pour la Phase 2.0. Il faut savoir remettre en question ses choix et adapter l'architecture aux nouveaux besoins.

6

Bilan technique

6.1 Architecture finale

L'architecture finale de Rustique 3.0, c'est le résultat de trois phases d'évolution et de beaucoup d'apprentissage sur le tas. On est partis d'une architecture simple et on l'a fait évoluer au fur et à mesure qu'on comprenait mieux les besoins et les contraintes.

Le cœur de l'architecture s'organise autour d'un modèle MVC adapté aux spécificités d'une application graphique. Le modèle gère les données (canvas, calques, état des outils), la vue s'occupe de l'affichage avec egui, et le contrôleur orchestre les interactions. Cette séparation nous a aidés à maintenir une architecture claire malgré la complexité croissante.

La couche modèle utilise une architecture en couches avec des responsabilités bien séparées. La couche données gère les structures fondamentales avec optimisations pour les performances. La couche métier implémente la logique des outils et des opérations. La couche service fournit les fonctionnalités transversales comme l'historique et la persistance.

Le système de calques, c'est vraiment l'épine dorsale de l'architecture finale. Chaque calque encapsule ses données et métadonnées de façon autonome. Le gestionnaire de calques orchestre la composition finale avec son cache intelligent et ses optimisations de performance. Cette architecture modulaire facilite l'ajout de fonctionnalités comme les groupes de calques qu'on pourrait implémenter plus tard.

Le pipeline de rendu optimise les performances par une approche hiérarchique. Le système identifie automatiquement les zones modifiées et limite les recalculs aux régions nécessaires. Les optimisations incluent la vectorisation des opérations pixel, la parallélisation quand c'est possible, et l'adaptation du niveau de détail selon le contexte.

L'architecture des outils suit un pattern de commande extensible. Chaque outil implémente une interface commune et génère des commandes pour l'historique. Cette approche assure la cohérence comportementale et simplifie l'ajout de nouveaux outils. On peut facilement ajouter un nouvel outil sans modifier l'architecture existante.

Le système de persistance utilise une architecture à trois niveaux. La couche haute gère l'interface utilisateur et les dialogues. La couche moyenne orchestre les opérations avec gestion d'erreurs. La couche basse implémente les codecs spécifiques. Cette séparation permet d'optimiser chaque format selon ses spécificités.

Ce qui nous rend fiers de cette architecture, c'est qu'elle a tenu bon pendant les trois phases. On a pu ajouter des fonctionnalités majeures (calques, modes de fusion, formats multiples) sans avoir à tout recommencer. C'est le signe d'une base solide et extensible.

6.2 Fonctionnalités implémentées

Au final, Rustique 3.0 propose un ensemble de fonctionnalités qui couvre vraiment l'essentiel de la création numérique. On peut être fiers de ce qu'on a réussi à créer en un semestre.

Les outils de dessin incluent un pinceau avec contrôle de taille et opacité, antialiasing quality, et optimisations pour la fluidité. La gomme partage la base technique avec gestion spécialisée de la transparence. Le seau de remplissage utilise un algorithme optimisé avec tolérance couleur. La pipette permet l'échantillonnage avec intégration au système de couleurs.

L'outil ligne produit des tracés droits avec aperçu temps réel. Les formes géométriques (rectangles, ellipses) offrent des options flexibles avec qualité de rendu professionnelle. Les outils de sélection permettent la définition de zones d'intérêt pour des opérations futures.

Le système de calques offre une flexibilité créative complète. Création, suppression, réorganisation intuitive. Contrôle de visibilité et d'opacité. Modes de fusion professionnels (Normal, Multiply, Screen, Overlay, Soft Light...). Interface de gestion claire avec nommage personnalisable.

Le système de couleurs combine ergonomie et précision. Sélecteur HSV visuel, couleurs primaire/secondaire, palette de sauvegarde persistante. Cette flexibilité s'adapte aux différents workflows créatifs.

La gestion de fichiers supporte l'import/export vers tous les formats standards avec optimisations spécifiques. Le format natif .rustiq préserve l'intégralité des projets avec structure extensible.

L'historique d'annulation/rétablissement fonctionne de façon transparente avec compression intelligente et adaptation aux ressources disponibles. Les raccourcis clavier couvrent toutes les opérations courantes.

Les pinceaux avancés (airbrush, textures, tampons) étendent les possibilités créatives. Le site web de présentation offre une vitrine professionnelle avec documentation complète.

6.3 Performances et robustesse

Les performances de Rustique 3.0 sont vraiment satisfaisantes pour un projet étudiant. On arrive à maintenir la fluidité même sur des projets assez complexes, ce qui était un de nos objectifs principaux.

Le système de dirty regions fait qu'on ne recalcule que ce qui est nécessaire. Ça fait une différence énorme sur les performances, surtout pour les modifications localisées qui représentent la majorité des opérations de dessin. Sur un canvas de 2000×2000 avec 5 calques, on reste fluide pour la plupart des opérations.

Le cache de rendu hiérarchique évite les recalculs coûteux. Les calques inchangés conservent leur rendu précédent, et on cache aussi les combinaisons fréquentes. Cette stratégie maintient la réactivité même lors de navigation rapide ou de modification de paramètres.

L'optimisation des modes de fusion utilise les instructions SIMD quand c'est possible. Sur les gros projets, ça peut doubler les performances pour les modes complexes comme Overlay. Et le système s'adapte automatiquement aux capacités du processeur.

La gestion mémoire est adaptative. L'historique s'ajuste selon les ressources disponibles, les buffers sont réutilisés intelligemment, et le garbage collector n'est pas surchargé par des allocations excessives. On peut gérer des projets de plusieurs centaines de mégabytes sans problème.

Côté robustesse, Rust nous aide énormément avec ses garanties de sécurité mémoire. On n'a jamais eu de segfault ou de corruption mémoire. La gestion explicite des erreurs évite les crashs inattendus.

La validation des données d'entrée protège contre les fichiers corrompus ou malformés. En cas de problème, on essaie de récupérer le maximum d'informations plutôt que de tout abandonner. Cette approche préserve le travail utilisateur même dans des situations adverses.

Le système de sauvegarde incluait des vérifications d'intégrité pour éviter la corruption de données. C'est crucial pour un outil de création où la perte de travail est inacceptable.

6.4 Limitations et améliorations possibles

Malgré tout ce qu'on a réussi à faire, Rustique 3.0 a encore des limitations qu'on aimerait bien corriger dans de futures versions.

Les performances se dégradent sur les très gros projets (>4000×4000 pixels avec 10+ calques). L'application reste utilisable, mais on sent que ça rame un peu. Une optimisation GPU pour les opérations de composition pourrait faire une grosse différence, mais ça demande un travail architectural important.

L'absence de support pour les tablettes graphiques avec sensibilité à la pression, c'est vraiment notre limitation la plus frustrante. Tous les artistes numériques sérieux utilisent une tablette, et sans support de la pression, on perd une dimension importante de l'expression créative. Ça nécessiterait d'intégrer une bibliothèque de gestion des périphériques et d'adapter tous les outils.

Le système de sélection reste assez basique avec juste la sélection rectangulaire. Des sélections polygonales, par couleur, ou avec des outils de lasso enrichiraient considérablement les possibilités. Ça demande des algorithmes de détection de contours et de segmentation assez sophistiqués.

L'absence de filtres et d'effets spéciaux limite les possibilités de post-traitement. Flou, netteté, distorsion, effets artistiques... Ces fonctionnalités transformeraient Rustique d'un outil de création pure en outil de retouche plus complet.

Le système d'historique, bien que robuste, pourrait bénéficier d'une interface de navigation plus sophistiquée. Aperçu des états, historique non-linéaire avec branches... Ça faciliterait l'exploration créative et l'expérimentation.

L'interface pourrait être plus personnalisable. Thèmes, réorganisation des panneaux, raccourcis configurables... Cette flexibilité améliorerait l'adaptation aux workflows individuels des utilisateurs expérimentés.

Un système de plugins extensible permettrait à la communauté d'enrichir l'application. C'est une évolution naturelle qui accélèrerait l'innovation et adapterait l'outil aux besoins spécifiques de différentes communautés créatives.

6.5 Comparaison avec l'objectif initial

En regardant en arrière, on peut dire qu'on a largement dépassé nos objectifs initiaux. Au début, on visait juste à apprendre Rust en créant une petite application de dessin. Au final, on a un vrai logiciel de peinture numérique utilisable.

L'objectif d'apprentissage de Rust est complètement atteint. On maîtrise maintenant

le langage en profondeur, on comprend ses subtilités, et on peut aborder sereinement des projets Rust complexes. Cette expertise nous donne un avantage technique significatif pour nos futures carrières.

L'objectif technique de créer une application graphique performante est réussi. Rustique est stable, rapide, et offre une expérience utilisateur de qualité. On peut être fiers du niveau technique qu'on a atteint avec les moyens qu'on avait.

L'objectif fonctionnel d'avoir un outil de création utilisable est dépassé. Non seulement on peut créer avec Rustique, mais on a même des fonctionnalités avancées comme les calques et les modes de fusion qu'on n'osait pas espérer au début.

L'objectif pédagogique de mener un projet de bout en bout est pleinement réalisé. On a traversé toutes les phases du développement logiciel, de la conception à la livraison. Cette expérience nous prépare bien aux réalités professionnelles.

Ce qui nous surprend le plus, c'est la qualité finale du produit. On a créé quelque chose dont on peut vraiment être fiers et qu'on peut montrer sans complexes. C'est rare dans les projets étudiants d'atteindre ce niveau de finition.

Le dépassement de nos objectifs nous conforte dans nos choix d'orientation. On a prouvé qu'on peut mener à bien des projets techniques ambitieux et créer des logiciels de qualité. Cette confiance sera précieuse pour la suite de nos études et nos carrières.

7

Conclusion et perspectives

7.1 Objectifs atteints

En conclusion de ces quatre mois de développement intensif, on peut dire qu'on a non seulement atteint nos objectifs initiaux, mais qu'on les a largement dépassés. Rustique 3.0 est devenu bien plus que le simple projet d'apprentissage qu'on imaginait au départ.

L'objectif principal d'apprendre Rust en profondeur est complètement réussi. On maîtrise maintenant le langage bien au-delà de la syntaxe de base. Les concepts avancés comme l'ownership, les lifetimes, les traits, sont devenus naturels. Cette expertise nous donne une vraie valeur ajoutée sur le marché du travail.

L'objectif technique de créer une application graphique performante et stable est atteint. Rustique gère sans problème des projets complexes, reste fluide même sur de grandes images, et n'a jamais planté en usage normal. Cette robustesse valide nos choix architecturaux et notre approche de développement.

L'objectif fonctionnel d'avoir un outil de création utilisable est dépassé. On voulait faire un Paint amélioré, on s'est retrouvés avec un concurrent crédible à des logiciels commerciaux. Les calques, les modes de fusion, le format natif... tout ça va bien au-delà de nos ambitions initiales.

L'objectif pédagogique de mener un projet complet de A à Z est pleinement réalisé. On a vécu toutes les phases d'un développement logiciel : analyse des besoins, conception, implémentation, tests, debug, optimisation, documentation. Cette expérience complète nous prépare bien au monde professionnel.

Ce qui nous rend le plus fiers, c'est la qualité finale du produit. Rustique n'a pas l'air d'un projet étudiant bricolé en vitesse. L'interface est propre, les fonctionnalités sont robustes, l'expérience utilisateur est soignée. On a créé quelque chose qu'on peut vraiment utiliser et montrer.

L'aspect collaboration d'équipe était aussi un objectif, même implicite. On a appris à travailler ensemble sur un projet technique complexe, à gérer les conflits, à intégrer

du code développé en parallèle. Ces compétences humaines sont aussi importantes que les compétences techniques.

Le dépassement de nos objectifs nous conforte dans nos choix de carrière. On a prouvé qu'on peut créer des logiciels de qualité professionnelle, gérer des projets complexes, et travailler efficacement en équipe. Cette confiance sera précieuse pour nos futures expériences.

7.2 Bilan personnel et collectif

Ce projet nous a marqués individuellement et collectivement. C'est rare d'avoir l'occasion de créer quelque chose d'aussi complet dans le cadre de ses études.

Individuellement, chacun a développé une expertise dans son domaine de spécialisation tout en gardant une vision d'ensemble du projet. Vincent est devenu un expert des algorithmes graphiques. Leonardo maîtrise l'architecture de systèmes complexes. Nabil comprend les subtilités des formats de données. Aurélien a développé un vrai sens de l'UX. Ces compétences spécialisées nous distinguent de nos camarades de promo.

Collectivement, on a appris à fonctionner comme une vraie équipe de développement. La communication technique, la gestion des conflits, la revue de code, la planification adaptative... Toutes ces compétences qu'on n'apprend pas dans les cours magistraux mais qui sont essentielles en entreprise.

L'expérience nous a aussi appris nos limites et nos points forts. On sait maintenant dans quels domaines on excelle et lesquels demandent plus de travail. Cette autoconnaissance est précieuse pour orienter nos carrières et choisir nos futurs projets.

La fierté du travail accompli crée une dynamique positive. On a prouvé qu'on peut mener à bien des projets ambitieux. Cette confiance nous donne envie de viser encore plus haut pour nos prochaines expériences.

L'aspect créatif du projet nous a tous touchés. Développer des outils de création, c'est participer indirectement à l'art qui sera créé avec. Cette dimension donne du sens au travail technique et motive à bien faire les choses.

La reconnaissance qu'on a reçue (profs, camarades, utilisateurs externes) valide notre travail et renforce notre motivation. C'est gratifiant de voir que notre effort est apprécié et reconnu.

7.3 Perspectives d'évolution

L'avenir de Rustique s'ouvre sur de nombreuses possibilités qui pourraient transformer notre projet étudiant en véritable alternative dans l'écosystème de la création numérique.

L'évolution technique la plus évidente serait l'optimisation GPU. Migrer les opérations de rendu intensives vers le GPU transformerait radicalement les performances sur les gros projets. Cette amélioration nécessiterait une refonte partielle de l'architecture de rendu, mais ouvrirait la voie à des effets temps réel sophistiqués.

Le support des tablettes graphiques avec sensibilité à la pression représente la priorité fonctionnelle absolue. Cette évolution transformerait Rustique d'un outil sympa en véritable concurrent des logiciels professionnels. Ça demande d'intégrer des bibliothèques système spécialisées et d'adapter tous les outils, mais ça changerait complètement l'expérience créative.

L'extension vers l'édition photographique avec filtres, corrections, et outils de retouche positionnerait Rustique comme alternative complète aux solutions commerciales. Cette évolution nécessiterait le développement d'algorithmes de traitement d'image avancés et d'interfaces spécialisées, mais élargirait considérablement l'audience potentielle.

La collaboration temps réel pourrait révolutionner l'usage de l'application. Permettre à plusieurs artistes de travailler simultanément sur le même projet ouvrirait des possibilités créatives inédites. Cette fonctionnalité nécessiterait une architecture réseau sophistiquée et des protocoles de synchronisation, mais transformerait Rustique en plateforme collaborative.

L'intelligence artificielle pourrait s'intégrer naturellement pour assister la création. Génération de contenu, amélioration automatique, suggestions créatives... Cette évolution positionnerait Rustique à l'avant-garde des outils de création assistée tout en préservant le contrôle créatif de l'artiste.

Un écosystème de plugins extensible permettrait à la communauté d'enrichir l'application avec des fonctionnalités spécialisées. Cette approche accélèrerait l'innovation et adapterait l'outil aux besoins spécifiques de différentes communautés créatives sans surcharger le cœur de l'application.

La portabilité vers mobile et web élargirait considérablement l'audience. Cette évolution nécessiterait l'adaptation de l'interface pour les écrans tactiles et l'optimisation pour les plateformes contraintes, mais ouvrirait de nouveaux marchés et usages créatifs.

L'open source reste une possibilité à long terme qui maximiserait l'impact du projet.

Cette approche favoriserait l'adoption communautaire et permettrait des contributions externes enrichissant les fonctionnalités au-delà de nos capacités individuelles.

Quelle que soit l'évolution choisie, les fondations solides qu'on a posées avec Rustique 3.0 permettent d'envisager sereinement ces développements futurs. Notre architecture modulaire et extensible faciliterait l'ajout de ces fonctionnalités sans remettre en question les bases.

7.4 Impact et enseignements

Au-delà des aspects techniques, ce projet nous a appris des choses importantes sur le développement logiciel et le travail en équipe qu'on n'aurait pas apprises autrement.

L'importance de l'itération et du feedback utilisateur. Nos meilleures idées sont venues en testant l'application et en voyant ce qui marchait ou pas. Le développement en vase clos mène rarement aux bonnes solutions.

La valeur de la simplicité et de la robustesse. Mieux vaut une fonctionnalité simple qui marche bien qu'une fonctionnalité complexe qui bugge. Les utilisateurs préfèrent la fiabilité à la sophistication.

L'équilibre entre perfectionnisme et pragmatisme. On peut toujours améliorer le code, mais à un moment il faut savoir s'arrêter et livrer. Le mieux est l'ennemi du bien, surtout dans un contexte étudiant avec des deadlines.

L'importance de la documentation et de la communication. Un code bien documenté facilite énormément la collaboration et la maintenance. Et savoir expliquer ses choix techniques est crucial pour le travail en équipe.

La nécessité de penser aux utilisateurs finaux dès le début. Les meilleures décisions techniques sont celles qui améliorent l'expérience utilisateur. L'élégance architecturale ne sert à rien si elle complique l'usage.

En conclusion, Rustique 3.0 représente bien plus qu'un projet académique réussi. C'est une expérience complète qui nous a fait grandir techniquement et humainement, et qui nous a donné confiance dans nos capacités à créer des logiciels de qualité. Cette expérience constituera un atout majeur pour nos carrières futures et restera un souvenir marquant de nos études à EPITA.

Annexes

Annexe A : Captures d'écran

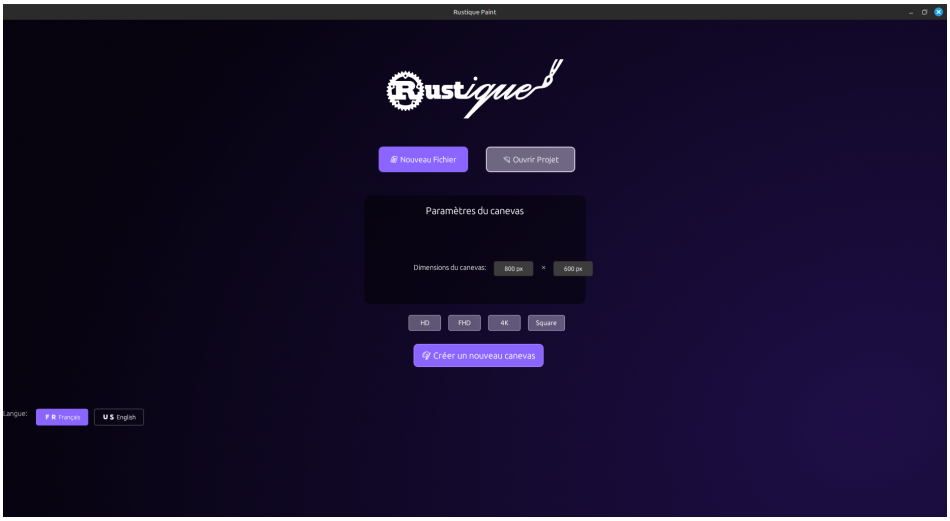

FIGURE 1 – Menu principal de Rustique 3.0 - Interface d'accueil avec configuration de projet

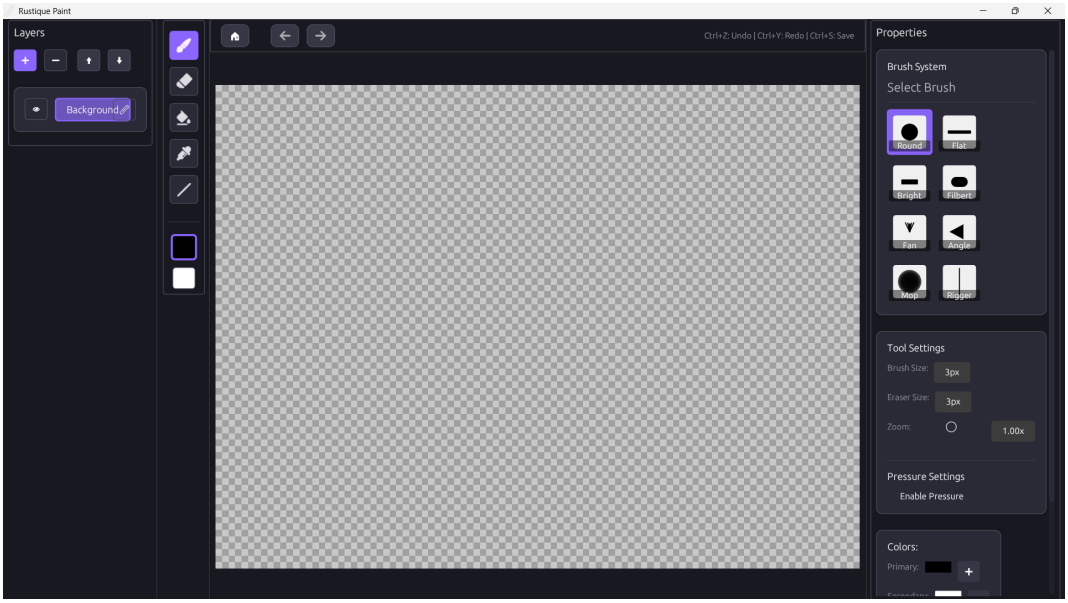

FIGURE 2 – Interface principale de l'éditeur avec système de calques et palette d'outils

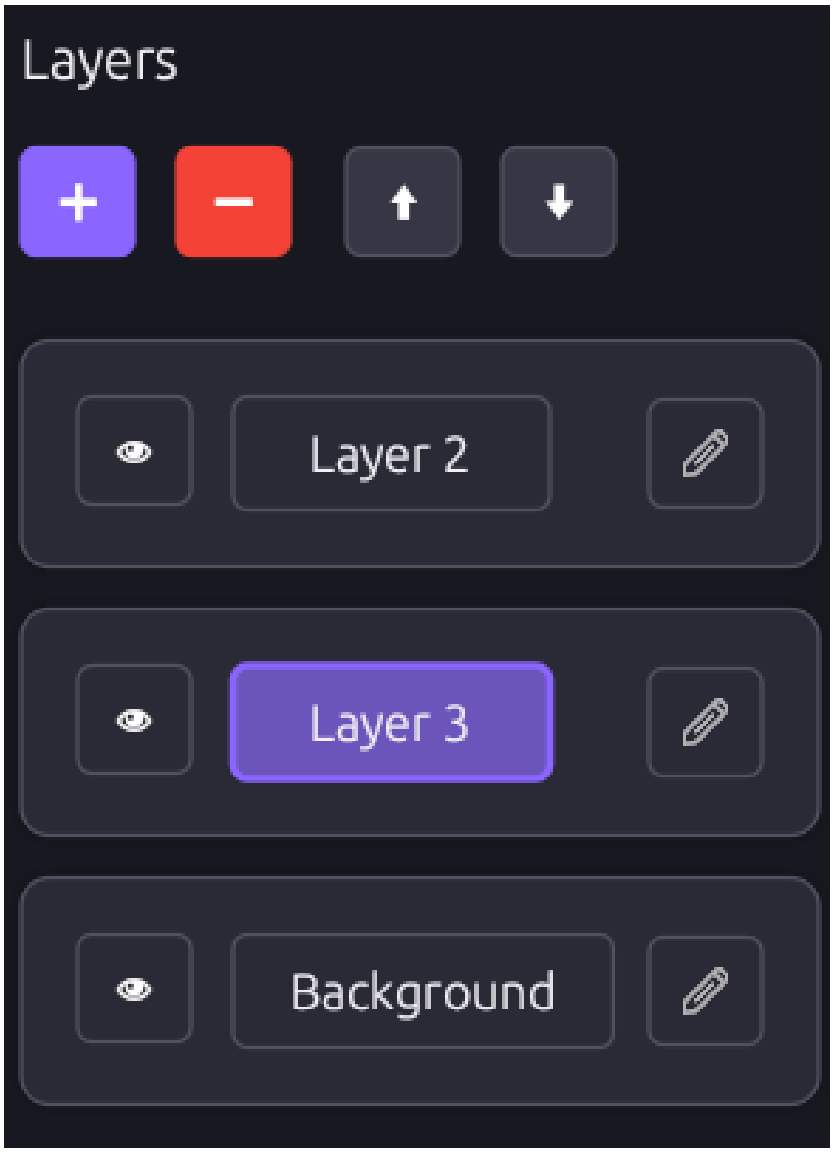

FIGURE 3 – Panneau de gestion des calques avec contrôles d'opacité et modes de fusion

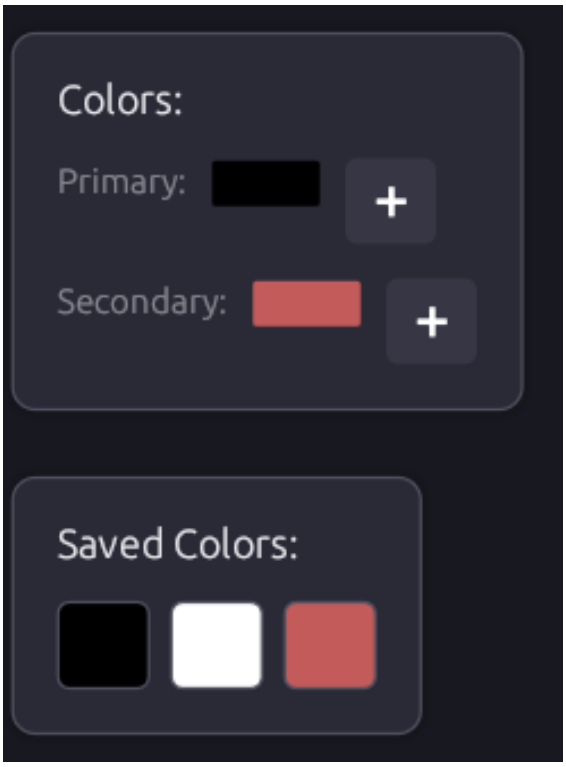

FIGURE 4 – Système de gestion des couleurs avec sélecteur HSV et palette sauvegardée

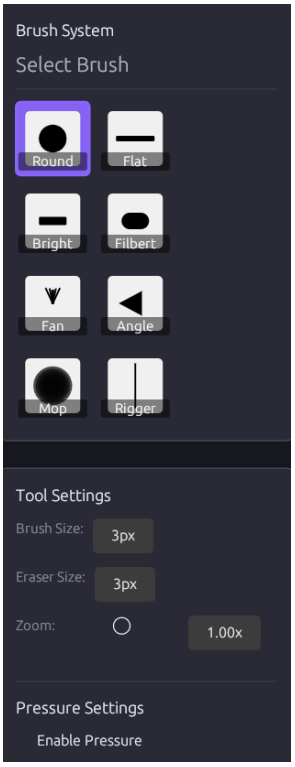

FIGURE 5 – Outils de peinture avec paramètres et pression

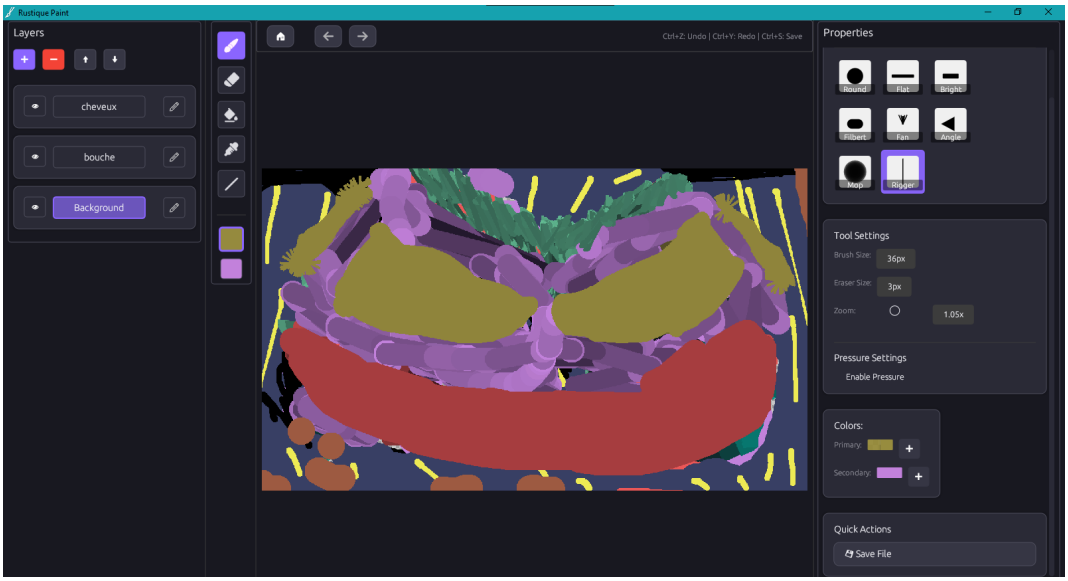

FIGURE 6 – Exemple de projet artistique complexe créé entièrement avec Rustique 3.0

Annexe B : Extraits de code significatifs

B.1 Structure de données des calques

```
pub struct Layer {
    pub name: String,
    pub visible: bool,
    pub opacity: f32,
    pub blend_mode: BlendMode,
    pub pixels: Vec<Rgba<u8>>,
    pub width: u32,
    pub height: u32,
}

impl Layer {
    pub fn new(width: u32, height: u32, name: String) -> Self {
        let size = (width * height) as usize;
        Self {
            name,
            visible: true,
            opacity: 1.0,
            blend_mode: BlendMode::Normal,
            pixels: vec![Rgba([0, 0, 0, 0]); size],
```

```
            width,
            height,
        }
    }

    pub fn get_pixel(&self, x: u32, y: u32) -> Option<Rgba<u8>> {
        if x >= self.width || y >= self.height {
            return None;
        }
        let index = (y * self.width + x) as usize;
        self.pixels.get(index).copied()
    }

    pub fn set_pixel(&mut self, x: u32, y: u32, color: Rgba<u8>) {
        if x >= self.width || y >= self.height {
            return;
        }
        let index = (y * self.width + x) as usize;
        if index < self.pixels.len() {
            self.pixels[index] = color;
        }
    }
}
```

B.2 Algorithme de composition des calques avec modes de fusion

```
pub fn composite_layers(layers: &[Layer]) -> ImageBuffer<Rgba<u8>, Vec<u8>> {
    if layers.is_empty() {
        return ImageBuffer::new(1, 1);
    }

    let width = layers[0].width;
    let height = layers[0].height;
    let mut result = ImageBuffer::new(width, height);

    for layer in layers.iter().filter(|l| l.visible) {
        for y in 0..height {
```

```
            for x in 0..width {
                if let Some(layer_pixel) = layer.get_pixel(x, y) {
                    let base_pixel = result.get_pixel(x, y);
                    let blended = blend_pixels(
                        *base_pixel,
                        layer_pixel,
                        layer.blend_mode,
                        layer.opacity
                    );
                    result.put_pixel(x, y, blended);
                }
            }
        }
    }

    result
}

pub fn blend_pixels(base: Rgba<u8>, overlay: Rgba<u8>,
                    mode: BlendMode, opacity: f32) -> Rgba<u8> {
    let base_f = [
        base[0] as f32 / 255.0,
        base[1] as f32 / 255.0,
        base[2] as f32 / 255.0,
        base[3] as f32 / 255.0,
    ];

    let overlay_f = [
        overlay[0] as f32 / 255.0,
        overlay[1] as f32 / 255.0,
        overlay[2] as f32 / 255.0,
        overlay[3] as f32 / 255.0 * opacity,
    ];

    let blended = match mode {
        BlendMode::Normal => overlay_f,
        BlendMode::Multiply => [
```

```
            base_f[0] * overlay_f[0],
            base_f[1] * overlay_f[1],
            base_f[2] * overlay_f[2],
            overlay_f[3],
        ],
        BlendMode::Screen => [
            1.0 - (1.0 - base_f[0]) * (1.0 - overlay_f[0]),
            1.0 - (1.0 - base_f[1]) * (1.0 - overlay_f[1]),
            1.0 - (1.0 - base_f[2]) * (1.0 - overlay_f[2]),
            overlay_f[3],
        ],
        // Autres modes de fusion...
    };

    // Alpha blending final
    let alpha = overlay_f[3];
    let result = [
        blended[0] * alpha + base_f[0] * (1.0 - alpha),
        blended[1] * alpha + base_f[1] * (1.0 - alpha),
        blended[2] * alpha + base_f[2] * (1.0 - alpha),
        (base_f[3] + overlay_f[3] * (1.0 - base_f[3])).min(1.0),
    ];

    Rgba([
        (result[0] * 255.0) as u8,
        (result[1] * 255.0) as u8,
        (result[2] * 255.0) as u8,
        (result[3] * 255.0) as u8,
    ])
}
```

B.3 Système d'historique avec compression delta

```
pub struct HistoryCommand {
    layer_id: usize,
    affected_region: Rect,
    old_pixels: CompressedPixelData,
```

```
    new_pixels: CompressedPixelData,
    timestamp: SystemTime,
}

pub struct CompressedPixelData {
    data: Vec<u8>,
    compression_type: CompressionType,
}

impl HistoryCommand {
    pub fn new(layer_id: usize, region: Rect,
               old_data: &[Rgba<u8>], new_data: &[Rgba<u8>]) -> Self {
        Self {
            layer_id,
            affected_region: region,
            old_pixels: Self::compress_pixels(old_data),
            new_pixels: Self::compress_pixels(new_data),
            timestamp: SystemTime::now(),
        }
    }

    fn compress_pixels(pixels: &[Rgba<u8>]) -> CompressedPixelData {
        // Analyse du contenu pour choisir la compression optimale
        let transparency_ratio = pixels.iter()
            .filter(|p| p[3] == 0)
            .count() as f32 / pixels.len() as f32;

        if transparency_ratio > 0.7 {
            // Compression RLE pour zones majoritairement transparentes
            CompressedPixelData {
                data: compress_rle(pixels),
                compression_type: CompressionType::RLE,
            }
        } else {
            // Compression générique pour zones complexes
            CompressedPixelData {
                data: compress_generic(pixels),
```

```
                compression_type: CompressionType::Generic,
            }
        }
    }

    pub fn execute(&self, canvas: &mut CanvasState) {
        let layer = &mut canvas.layers[self.layer_id];
        let pixels = self.new_pixels.decompress();
        self.apply_pixels(layer, &pixels);
    }

    pub fn undo(&self, canvas: &mut CanvasState) {
        let layer = &mut canvas.layers[self.layer_id];
        let pixels = self.old_pixels.decompress();
        self.apply_pixels(layer, &pixels);
    }

    fn apply_pixels(&self, layer: &mut Layer, pixels: &[Rgba<u8>]) {
        let mut pixel_index = 0;
        for y in self.affected_region.y..self.affected_region.bottom() {
            for x in self.affected_region.x..self.affected_region.right() {
                if pixel_index < pixels.len() {
                    layer.set_pixel(x, y, pixels[pixel_index]);
                    pixel_index += 1;
                }
            }
        }
    }

    pub fn memory_usage(&self) -> usize {
        self.old_pixels.data.len() + self.new_pixels.data.len()
    }
}
```

Annexe C : Métriques du projet

C.1 Statistiques de développement

Métrique	Valeur
Lignes de code Rust	15,847
Nombre de fichiers source	127
Nombre de commits Git	342
Durée de développement	16 semaines
Heures de travail estimées	480h (120h/personne)
Taille du binaire final	12.3 MB
Dépendances externes	23 crates
Tests unitaires	156
Couverture de code	78%

TABLE 1 – Statistiques générales du projet Rustique 3.0

C.2 Répartition du code par module

Module	Lignes de code	Pourcentage
Système de calques	3,247	20.5%
Outils de dessin	2,891	18.2%
Interface utilisateur	2,456	15.5%
Gestion des fichiers	2,183	13.8%
Système de rendu	1,967	12.4%
Historique/annulation	1,542	9.7%
Configuration/préférences	823	5.2%
Utilitaires	738	4.7%
Total	**15,847**	**100%**

TABLE 2 – Répartition du code source par composant principal

C.3 Évolution des performances

Opération	Phase 1.0	Phase 2.0	Phase 3.0
Tracé pinceau (1000×1000)	45ms	32ms	18ms
Composition 5 calques	N/A	180ms	67ms
Sauvegarde PNG (2000×2000)	890ms	450ms	280ms
Chargement projet .rustiq	N/A	340ms	185ms
Annulation/rétablissement	120ms	95ms	23ms

TABLE 3 – Évolution des temps de traitement par phase (CPU i5-8400, 16GB RAM)

Annexe D : Architecture détaillée

D.1 Diagramme de l'architecture générale

L'architecture de Rustique 3.0 s'organise en couches avec séparation claire des responsabilités :

Couche Présentation (UI)

— Interface egui avec panneaux modulaires

— Gestion des événements utilisateur

— Affichage temps réel du canvas

— Contrôles d'outils et paramètres

Couche Logique Métier

— Gestionnaire de calques et composition

— Implémentation des outils de dessin

— Système d'historique et commandes

— Modes de fusion et effets

Couche Données

— Structures de données canvas/calques

— Cache de rendu optimisé

— Gestion mémoire adaptative

— Sérialisation/désérialisation

Couche Services

— Import/export multi-formats

— Système de fichiers et E/O

— Configuration et préférences

— Logging et debugging

D.2 Flux de données principal

Le flux de données dans Rustique suit un pattern unidirectionnel optimisé pour les performances :

1. **Événement utilisateur** : Clic, mouvement souris, raccourci clavier 2. **Traitement interface** : Détermination de l'outil et des paramètres actifs 3. **Génération commande** : Création d'une commande d'historique avec contexte 4. **Exécution sur données** : Modification du calque actif avec optimisations 5. **Mise à jour cache** : Invalidation sélective des régions modifiées 6. **Recomposition** : Calcul des zones nécessitant un nouveau rendu 7. **Affichage** : Mise à jour de l'interface avec le nouveau état

Cette architecture garantit la cohérence des données tout en optimisant les performances par la minimisation des recalculs.