Conception, implémentation et analyse d'un framework d'édition interactive d'images gigapixel.

Frédéric van der Essen
Promoteurs: Philip Dutré, Marc Lobelle.

13 août 2010

Abstract

Résumé

Les images gigapixel ont un grand nombre d'applications, de l'art à la médecine en passant par la cartographie. Et plus récemment, les jeux vidéos. Les techniques photographiques permettant de les générer ne manquent pas, l'espace stockage et la bande passante non plus. Cependant les possibilités d'édition et de retouche numérique étaient jusqu'à présent très limitées, aucun framework d'édition d'image ne se révélant satisfaisant. Une ébauche d'un nouveau concept de framework, conçu et réalisé par l'auteur, promettait des performances quasi indépendantes de la taille de l'image éditée, permettant ainsi l'édition interactive d'images gigapixel

Afin de vérifier ces promesses, l'implémentation du framework fut partiellement complétée, afin de se concentrer sur un problème restreint ; la peinture d'images gigapixel. Cela révéla des problèmes de qualité et de performances, qui furent pour la plupart résolus de manière satisfaisante. Un logiciel de peinture utilisant ce framework fut ensuite complété puis testé par 3 professionels de l'art numérique afin d'évaluer la performance du framework et de son implémentation.

Les tests se sont conclus par la satisfaction des testeurs et la réalisation de plusieurs peintures gigapixel.

De nombreuses fonctionalités du framework restent cependant à implémenter et à tester, de même que de nombreuses pistes d'amélioration des performances ont été ouvertes et restent à explorer.

Préface

Ce rapport présente le résultat de mon mémoire réalisé au département de recherche en infographie de la *Katholieke Universiteit Leuven*. Ce mémoire est également la conclusion de mon master d'ingénieur en informatique à L'*Université Catholique de Louvain*.

J'aimerais remercier les personnes suivantes : Philip Dutré de la KUL, et Marc Lobelle de l'UCL pour avoir été mes promoteurs, et Benedict J. Brown pour les avoir assisté dans cette tâche. J'aimerais également remercier les mémorants du département de recherche en infographie de la KUL pour leurs discussions et commentaire à propos de ce mémoire.

Enfin j'aimerais remercier Brice Vandemoortele, Jean-François Brogniet et Jean-Philippe Servais pour avoir participé aux tests utilisateur.

Chapitre 1

Introduction

Les images gigapixel sont des images bitmap constituées de plus d'un milliard de pixel. Ces images apparaissent dans plusieurs domaines : la cartographie, avec les photographies satellites ; la médecine, avec les images issues des microscopes ; l'histoire et la préservation de l'art, avec des scans haute résolutions de tableaux et fresques de grande taille ; L'art numérique et également le jeu vidéo qui utilisent de telles images pour la création de larges environnements.

Les algorithmes de compression et le faible coût de l'espace disque permettent de stocker sans encombre de telles images. Il existe également des logiciels de visualisation efficace. Cependant, lorsqu'il s'agit de les éditer, les logiciels existant ne proposent qu'une sélection très réduite des fonctionalités proposées habituellement pour l'édition d'image mégapixel.

L'édition de telles images requiert une approche et des algorithmes différents. C'est ce que propose le framework nommé *Himalaya*, conçu et partiellement implémenté au préalable par l'auteur de ce mémoire. *Himalaya* est un framework conçu pour pouvoir proposer la plupart des fonctionalités des frameworks existants, avec des performances quasi indépendantes de la taille de l'image à éditer.

Cependant il y a un pas entre la conception et la réalisation, pas que nous avons partiellement franchi avec ce mémoire. En effet, l'implémentation complète du framework est une tâche d'une trop grande empleur pour être envisagée ici. Nous avons donc fait le choix de se concentrer sur un sous-problème difficile, qui est de permettre de peindre de manière interactive une image gigapixel, fonctionalité qui n'est proposée par aucun framework existant.

Avant d'examiner en détail le framework *Himalaya* il est important de se demander ce qu'est un framework d'édition d'image, quelles sont les fonctionalités utiles et nécessaires de ceux-ci. Enfin il faudra regarder de quelle manière les frameworks existant proposent ces fonctionalités, afin de comprendre les enjeux de conception. C'est ce que nous allons faire au prochain chapitre.

Chapitre 2

Présentation des frameworks d'édition d'images

Un framework est une librairie présentant une interface logicielle permettant a un utilisateur d'éditer des images.

Le terme utilisateur sera employé pour désigner en toute généralité des choses assez différentes. Ainsi il désignera parfois l'artiste qui utilise le logiciel d'éditon, le programmeur qui utilise le framework pour concevoir un logiciel, mais aussi un autre programme qui ferait appel à celui ci.

Nous allons tout d'abord examiner quels sont les éléments qui constituent un tel framework, ensuite quels sont les fonctionalités qu'ils doivent présenter à l'utilisateur.

Ensuite un survol des différents frameworks permettra des principes généraux de la conception d'un framework d'édition d'image.

Pour réaliser cette présentation je me suis basé sur une lecture du code source des différents frameworks open-source, afin d'évaluer les structures de données et les algorithmes utilisés. J'ai également étudié les fonctionalités et performances des logiciels fermés, qui bien que gardant secret leur fonctionnement, proposent généralement de meilleures performances et un plus large panel de fonctionalités.

Enfin, une étude des spécifications des principaux formats de fichiers standards permet de se rendre compte des fonctionalités qui sont attendues dans les frameworks.

Après chaque description de chaque framework on trouvera la liste des formats et logiciels ainsi examinés.

2.1 Architecture d'un framework d'édition d'image

Un framework d'édition d'images est composé de trois éléments principaux :

Schéma de représentation de l'image : Ce schéma est consititué de structures de données qui décrivent l'image et les modifications qui lui sont apportées.

Algorithme de rasterisation : Cet algorithme permet d'obtenir une version matricielle de l'image, ou d'une partie de celle-ci. Une telle opération est indispensable car seule la forme matricielle de l'image peut être affichée à l'écran.

Algorithmes d'édition du schéma : Ces algorithmes vont modifier la représentation de l'image afin d'implémenter les différentes fonctionalités du framework.

2.2 Fonctionalités d'un framework d'édition d'image

2.2.1 Opérations de dessin

L'opération de base d'un framework d'édition d'image est bien évidemment de permettre de modifier une image. On distingue plusieurs manières de le faire qui seront traitées différemment selon les implémentations.

Dessin de primitives

Par dessin de primitive on entend l'ajout sur l'image de primitives géométriques, comme des polygones, des lignes, des points, des ellipses, du texte, etc. Ces primitives ont généralement un effet local sur l'image, c'est à dire qu'elles ne la modifient pas dans son intégralité.

On différencie deux approches du dessin de primitive : L'approche dite vectorielle ou l'on utilise un nombre réduit de primitives complexes (texte, courbes de bézier,...), et l'approche bitmap ou l'on utilise un très grand nombre de primitives très simples(ellipses, petits bitmaps, ...) Pour cette seconde approche il est important que cette opération soit peu couteuse en temps et en mémoire, étant donné qu'une peinture peut être constituée de plusieurs millions de telles primitives.

C'est cette seconde approche que nous explorerons avec la peinture d'images gigapixel.

FIG. 2.1: Examples de dessins de primitives

Filtres

Un filtre est une opération qui modifie l'aspect de l'entièreté d'une image. Le temps de calcul d'un filtre varie énormément d'un filtre à l'autre. Les plus simples prennent quelques microsecondes et peuvent aisément être exécutés en temps réel. D'autres prennent plusieurs dixaines de minutes. Certains frameworks ne sauront intégrer que les filtres les plus rapides.

Les filtres peuvent être également partagés en deux catégories, les filtres colorimétriques qui transforment la couleur d'un pixel sans se soucier de la couleur des pixels voisins, et les filtres spatiaux nécessitant eux d'en connaître plusieurs.

Transformations géométriques

Les transformations géométriques modifient la forme d'un objet constituant l'image ; rotation, translation, redimentionnement et perspective sont les applications les plus courantes.

Il y a deux approches pour implémenter les transformations géométriques : l'approche vectorielle consiste à modifier la description de l'image avant la rasterisation. L'approche bitmap consiste à modifier les pixels après leur rasterisation. Cette deuxième approche, nettement plus lente, permet cependant

(a) Image originale (b) Filtre colorimétrique (c) Filtre spatial *Sobel*

FIG. 2.2: Examples de filtres

d'implémenter des transformations non linéaires, telle que les transformées polaires, la correction de distortion de lentille, ou les déformations fluides.

Fusion et modes de fusion

La fusion consiste à fusionner deux images en une. On utilise pour cela une fonction nommée *mode de fusion*. Elle à la forme $f(p_A, p_B, \alpha)$, où p_A, p_B sont deux pixels des images A,B que l'on veut fusionner. α représente un ou plusieurs paramètres additionnels qui peuvent modifier l'effet de la fonction.

Le *mode de fusion* la plus courante est celui de mélange par opacité : $f(p_A, p_B, \alpha) = \alpha * p_A + (1 - \alpha) * p_B$, $\alpha \in [0, 1]$ représentant l'opacité de B, mais il en existe bien d'autres.

On peut utiliser cette fonction pour fusionner deux images en une nouvelle : $C = f(A, B, \alpha)$, ou pour intégrer une image dans une autre $A = f(A, B, \alpha)$

Puisque chaque dessin de primitive consiste en la fusion de celle ci sur l'image de fond, la fusion est une opération centrale dans tout framework. Cependant beaucoup d'entre eux se limitent au seul mode de mélange par opacité.

(a) Image A (b) Image B (c) Blending de B sur A, par *fusion de grain* (d) Blending de B sur A, par *soustraction*

FIG. 2.3: Examples de blending

2.2.2 MODÈLES COLORIMÉTRIQUES

La rasterisation d'une image permet de définir la couleur de chaque pixel. Or il existe de nombreuses manières de définir une même couleur. La manière la plus populaire est de définir la couleur comme un vecteur dans un espace colorimétrique. Chaque espace ayant sa propre utilité :

Le RGB est l'espace utilisé par les écrans et projecteurs, ainsi que les capteur photographiques. Toute couleur doit donc être convertie en RGB avant d'être affichée. Il existe plusieurs espaces RGB définis par des couleurs primaires Rouges, Vertes et Bleues différentes. Le plus utilisé est le sRGB, qui est le standard d'affichage des écrans.

Le Yuv est l'espace utilisé par les vidéos et par certains formats de fichier comme le JPEG. L'espace Yuv permet en effet une meilleure compression des données.

Le CIELab est l'espace utilisé en retouche photo, il permet de manipuler les couleurs de manière conforme à la perception humaine.

Le HSL est un espace qui permet de décrire les couleurs en composantes de teintes, de saturation et de luminosité, notions facilement compréhensibles et manipulables. Cet espace est utilisé pour certains filtres et pour l'édition des couleurs.

Le CMYK est l'espace utilisé par les imprimantes jet d'encre, chaque imprimante ayant ses propres couleurs primaires. Toute image doit donc être convertie dans le CMYK correspondant à l'imprimante avant d'être imprimée.

Il existe encore bien d'autres espaces colorimétriques spécifiques à des applications industrielles particulières.

Pour compliquer le tout, les espaces colorimétriques ne se superposent généralement pas. Ainsi des couleurs qui peuvent s'exprimer dans l'un n'existent pas dans l'autre. Un framework dédié à l'impression doit ainsi prendre garde de n'afficher en RGB que des couleurs disponibles dans le CMYK de l'imprimante.

Une autre manière de décrire les couleurs est d'utiliser un nuancier contenant une liste de couleurs standardisées. Ce système est utilisé dans les images destinées à l'impression afin d'identifier des encres dont l'apparence ne peut être réduite à une simple couleur, comme les encres mattes, brillantes, métallisées, fluorescentes, etc. Dans ce cas chaque pixel possède une certaine quantité de chacune des couleurs utilisées pour décrire l'image.

Quantisation

Les composantes d'un pixel peuvent aussi être quantifiées à différents degrés : 1bit pour les systèmes halftone ; 8bit pour l'affichage à l'écran, les textures de jeux vidéos ; 32 bit flottant permet d'avoir des valeurs de couleurs dépassant les capacité d'affichage d'un écran, ce qui est particulièrement intéressant pour l'édition non destructive d'image utilisée en photographie et en effets spéciaux. La précision apportée par une telle profondeur est également indispensable pour le cinéma numérique.

Extensions de modèles colorimétriques

Enfin, il arrive qu'un pixel ne décrive plus une couleur mais une information quelconque de lá partie de l'objet qu'il représente. On rencontre ainsi la distance du pixel à l'observateur, la normale de la surface, un identifiant unique de l'objet, la vitesse de déplacement de l'objet, etc. Ces informations sont généralement fournies en plus des informations colorimétriques habituelles.

Ce type d'information est typiquement fourni par la caméra ou généré par le moteur de rendu afin de faciliter l'intégration d'effets spéciaux à l'image. Le type d'information utilisés apparaissent et disparaissent avec les technologies, nécessitant une grande flexibilité dans leur gestion. Les *Éditeurs Nodaux* sont particulièrement adaptés à la gestion de ce type d'informations.

2.2.3 Undo / Redo

Un framework a souvent des mécanismes internes permettant une gestion efficace d'annulation et de répétiton des opérations.

2.2.4 Édition non destructive

L'édition non destructive consiste à pouvoir modifier les opérations après leur application sans perte de qualité de l'image.

(a) Couleurs vraies (b) Format HSV en fausses couleurs (c) Vecteur normal de la surface (d) Distance à l'observateur

FIG. 2.4: Examples d'espaces colorimétriques et de métadonnées

2.2.5 COMPOSITION NON LINÉAIRE

La composition non linéaire permet à une opération d'utiliser le résultat d'une opération autre que celle appliquée précedemment. Idéalement le résultat de cette opération ne doit pas être recalculé. Cette fonctionalité permet de combiner des effets simples afin d'obtenir des effets beaucoup plus complexes que ce qui est possible par composition linéaire, et permet souvent à l'utilisateur d'éviter de devoir programmer ses propres effets. De nombreux logiciels présentent ainsi une interface nodale comme alternative à la programmation de pixel-shaders.

2.2.6 ÉDITION PAR PIXEL

Certains frameworks permettent d'éditer individuellement chaque pixel de l'image.

2.2.7 INDÉPENDANCE À LA RÉSOLUTION

La description de l'image est indépendante de la résolution utilisée pour la rasterisation ; Il n'y a pas de dégradation de la qualité de l'image quelque soit la résolution utilisée pour le rendu. L'édition par pixel et l'indpendance à la résolution sont deux fonctionalités mutuellement exclusives.

2.2.8 IMAGES GIGAPIXELS

Les images gigapixels sont les images constituées de plus d'un milliard de pixel, qui peuvent être plus grandes que la mémoire vive disponible. Il ne faut pas confondre édition d'images gigapixel et indépendance à la résolution. Si l'indépendance à la résolution permettent de décrire des documents de tailles aussi grande que désirée et de les rasteriser en des images gigapixel,ils ne permettent pas de décrire ou d'éditer des images de plus d'un milliard de pixels indépendants.

2.2.9 TRAITEMENT D'IMAGES MULTIPLES

Le traitement d'images multiples consiste à savoir appliquer facilement les mêmes opérations sur un large nombre d'images similaires, comme par exemple les photos d'une séance de shooting, les frames d'une vidéo ou d'un moteur 3D.

2.3 Frameworks Bitmaps

2.3.1 SCHÉMA DE REPRÉSENTATION DE L'IMAGE

Dans un framework bitmap l'image est représentée directement sous sa forme rasterisée.

Une extension populaire est de décomposer l'image en une image de fond et une superposition de calques qui sont des bitmaps possèdant chacun leur propre dimension, position, un mode de fusion, et des paramètres d'opacité.

FIG. 2.5: Représentation d'une image par calques

Cette extension permet à l'utilisateur de pouvoir facilement modifier des zones spécifiques de l'image.

Des frameworks extendent encore ce principent en décomposant les calques en sous calques, et ce de manière récursive.

2.3.2 ALGORITHME DE RASTERISATION

Dans un framework bitmap, chaque opération est directement rasterisée à la résolution native lors de son application.

Pour visualiser une région à une échelle différente de la résolution native, On utilise un buffer de visualisation de la taille de cette région dans lequel la région d'intérèt est mis à l'échelle voulue après chaque rasterisation.

Lorsqu'un système de calques est utilisé, on rasterise d'abord directement l'opération sur le calque modifié. On place ensuite l'image de fond dans le buffer de visualisation. Chaque calque y est ensuite fusionné.

Comme la plupart des opérations ne modifie qu'une petite partie de l'image, on aimerait éviter de devoir fusionner l'intégralité des calques à chaque fois. Il y a deux approches pour cela : La première est de demander à chaque opération de spécifier la région qu'elle modifie. On ne modifie ensuite le buffer de visualisation que pour cette région.

La deuxième approche consiste à diviser l'image de fond et les calques en une grille de sous régions aux dimensions régulières appelées tiles. Lors de l'application de l'opération, celle ci ne modifiera que certains tiles. Seuls les tiles correspondants dans les calques et l'image de fond seront fusionnés dans le buffer de visualisation. Les tiles peuvent également être traités en parallèle sur une machine multi-processeur.

Les tiles ont d'autres intérèts que d'accélérer la rasterisation ; En omettant les tiles des zones transparentes des calques ou réduit l'espace mémoire qu'ils consomment. En outre, les tiles peuvent être migrés vers le disque dur lorsqu'il n'est plus possible de tous les stoquer en mémoire, ils sont ensuite récupérés lorsqu'ils sont utilisés.

Il n'est cependant pas toujours possible d'implémenter une opération pour qu'elle fonctionne tile par tile. Dans ces cas on devra placer les tiles dans un buffer temporaire et les récupérer après l'opération.

2.3.3 FONCTIONALITÉS ADAPTÉES AUX FRAMEWORKS BITMAPS

Opérations de dessin

Étant donné qu'il n'est pas nécessaire de maintenir une liste de toutes les opérations appliquées sur l'image, les frameworks bitmaps sont particulièrement efficaces lorsqu'un très grand nombre de celles-ci sont utilisés ce qui est le cas pour les logiciels de peinture.

Undo / Redo

L'undo/redo est implémenté en gardant une copie de la région/tiles modifiée par l'opération avant sa modification. Il suffit ensuite de réutiliser ces copies pour obtenir la version sauvegardée. Garder les copies consomme beaucoup de mémoire, et une limite d'historique est nécessaire pour pallier à ce problème. En revanche, annuler ou refaire une opération ne demande pas de recalculer l'opération et est donc très rapide.

2.3.4 Fonctionalités inadaptées aux frameworks bitmaps

Modèles colorimétriques et métadonnées

L'architecture des frameworks bitmaps limite sévèrement les possibilité de gestion de modèles colorimétriques et de métadonnées. En effet, comme chaque opération est rasterisée dans le calque dès son application, les opérations doivent nécéssairement avoir le même modèle colorimétrique en entrée et en sortie. Elles doivent aussi comprendre le modèle colorimétrique et extensions du calque, ce qui nécessite de devoir soit modifier les opérations à chaque ajout de nouveau type de modèles colorimétriques et d'extensions soit de passer par de couteuses transformations de modèles.

Il exite une solution permettant d'améliorer la situation : Au lieu de placer tous les composants du pixel dans un même calque, on divise le calque en sous calques appelés canaux qui contiennent chacun un seul composant du pixel. Les opérations sont ensuite codées pour prendre des canaux en entrée et en sortie. Il est maintenant possible d'ignorer les canaux inutiles, d'avoir des canaux de différentes précision, et de changer la signification d'un canal lors de l'application d'une opération.

Cette approche à pour inconvénient de ralentir nettement les opérations, puisqu'il faut désormais accéder des zones de mémoire fort éloignées pour lire ou modifier un seul pixel.

En pratique les framework bitmap implémentent un nombre limité de modèles colorimétriques, et imposent le même modèle pour tous les calques d'une image.

Images gigapixel

Il est théoriquement possible de gérer des images de taille plus grande que la mémoire disponible en utilisant les tiles à bon escient. Mais en pratique, chaque opération nécessite d'être appliquée intégralement sur l'image à résolution native, ce qui est infaisable de manière interactive pour des opérations modifiant de grandes parties de l'image.

Édition non destructive

L'édition non destructive requiert de garder une liste de toutes les opérations effectuées sur l'image, afin de pouvoir modifier l'opération désirée, et de refaire celles qui doivent être réappliquées.

Il est cependant possible d'implémenter des opérations non destructives en tant que calques à part entière. Si l'opération est un filtre spatial, il faudra pouvoir étendre le buffer de rasterisation pour que le filtre ait accès aux données nécessaires. Et s'il s'agit d'un filtre de transformation, la zone à modifier dans le buffer de visualisation devra elle aussi être modifiée au fur et à mesure des filtres. Tout cela étant fort compliqué à implémenter, sans garantie de performances, les frameworks confinent généralement l'édition non destructive aux filtres colorimétriques et dessin de primitives.

2.3.5 Framework bitmaps, l'état de l'art

Adobe Photoshop Logiciel de peinture et retouche photo

Corel Painter Logiciel de peinture

Artrage Logiciel de peinture

Gimp ≤ 2.6 Logiciel de peinture et retouche photo open source

Krita Logiciel de peinture et retouche photo open source

Blender3d Utilisé pour la peinture de texture, Logiciel d'animation 3D.

psd Format d'image de photoshop

xcf Format d'image de Gimp.

2.4 Frameworks Nodaux

2.4.1 Schéma de représentation de l'image

(a) En rouge les noeuds d'entrée, en orange les noeuds d'opération, en vert les noeuds de visualisation

(b) Image calculée par le graphe (a)

FIG. 2.6: Example d'édition nodale d'images telle qu'implémenté par *Blender 2.49*

Les frameworks nodaux représentent l'image par un graphe composé de trois types de noeuds.

- Les noeuds d'entrée servent à spécifier les images à modifier et ne proposent que des sorties.
- Les noeuds d'opération prennent en entrée des images et/ou des canaux et/ou des paramètres, et proposent en sortie le résultat des images et/ou des canaux et/ou des valeurs, résultats de l'opération sur les entrées.
- Les noeuds de visualisation n'ont qu'une entrée et servent à visualiser sa valeur, que ce soit un paramètre, un canal ou une image. Dans le cas de canaux ou d'images, le noeud peut également spécifier une sous région de l'image et une échelle.

Le graphe est acyclique et dirigé ; chaque arrète va de la sortie d'un noeud à une entrée d'un autre. Il peut y avoir plusieurs arrètes partant d'une sortie, mais une seule arrivant à chaque entrée.

2.4.2 Algorithme de rasterisation

Pour obtenir la sortie d'un noeud, il faut premièrement obtenir toutes ses entrées, et appliquer l'opération du noeud s'il y en a une. On appliquera donc ce principe de manière récursive en partant d'un noeud de visualisation, la récursion s'arrètant au noeuds d'entrée.

Si le noeud de visualisation spécifie une sous-région de l'image, l'algorithme fonctionne de manière identique, à ceci près que les opérations pouvant transformer géométriquement l'image, la région en entrée ne sera pas la même que la sortie. Il faut donc que le noeud soit capable d'inverser la transformation qu'il effectue afin de demander la bonne région à ses parents.

Si le noeud de visualisation spécifie une échelle, les images fournies par les noeuds d'entrée sont mises à l'échelle avant leur sortie, de même que les paramètres métriques des opérations.

Comme une sortie peut être connectée à plusieurs entrées, une même opération peut être calculée plusieurs fois. Pour éviter cela, chaque noeud peut disposer d'une cache dans laquelle il place le résultat de ses sorties.

Les images utilisées peuvent également utiliser une représentation par tiles afin de bénéficier des avantages de gestion d'images volumineuses.

2.4.3 Fonctionalités adaptées aux frameworks nodaux

Édition non destructive

C'est un des gros points forts des frameworks nodaux. On peut facilement modifier les paramètres et la topologie du graphe et visualiser le résultat.

Composition non linéaire

La composition non linéaire découle de l'architecture en graphe des frameworks nodaux. Les frameworks nodaux sont d'ailleurs les seuls frameworks permettant une réelle édition non linéaire.

Modèle colorimétrique

Un framework nodal offre une liberté totale à l'utilisateur en ce qui concerne les modèles colorimétriques. Si un noeud sort une image en RGB, et qu'une opération attend du YUV, il peut les connecter ensemble. Le canal R sera interprété comme Y, le G comme U et le B comme V. S'il désire garder l'interprétation colorimétrique du RGB, il devra utiliser un noeud qui convertit le RGB en YUV.

Intégrer un nouveau modèle colorimétrique se limite donc à créer des noeuds de conversions et les noeuds d'opération qui peuvent tirer bénéfice de ce nouveau modèle.

L'édition nodale est cependant incapable de gérer les tons directs.

Images gigapixel

La complexité en temps et en mémoire de la rasterisation d'une image ne dépend tout au long du graphe que de la taille de la région du noeud de visualisation. Une borne supérieure de la taille de ces région est la résolution des écrans qui ne dépasse pas les quelques mégapixels. Il reste l'opération d'échelle à effectuer au niveau des région d'entrée. Si on dispose de mipmaps de ces images, alors il est possible d'effectuer des opérations sur images gigapixels de manière efficace.

Traitement d'images multiples

Le même graphe peut très facilement s'appliquer sur un grand nombre d'images de manière automatique, ce qui rend ces frameworks particulièrement adaptés au traitement vidéo.

2.4.4 Fonctionalités inadaptées aux frameworks nodaux

Undo/Redo

On peut s'attendre d'un framework permettant l'édition non destructive d'exceller dans l'undo/redo. Cependant, il n'est pas facile de gérer les changements de paramètres des noeuds et de la topologie du graphe tout en maintenant des caches cohérentes. C'est pourquoi les graphes nécessitent souvent d'être rasterisés à nouveau ce qui peut prendre du temps lorsque celui ci contient des filtres complexes.

Opération de dessin de primitves

Il est tout à fait possible de créer des noeuds dessinant des primitives, et de les connecter entre eux pour avoir une topologie semblant correspondre aux frameworks bitmaps ou vectoriels.

Un premier problème est que les frameworks nodaux appliquent les opérations de transformations sur les pixels constituant la primitive et non sur les paramètres la définissant. Par exemple, il sera impossible de récupérer l'apparence exacte de la primitive après une transformation d'échelle la réduisant à un seul pixel. Cet exemple extrême démontre comment à chaque transformation successive, la primitive est peu à peu dégradée. Un framework vectoriel n'a pas ce genre de problèmes.

Ensuite, un framework nodal est une entité complexe, et il devient impossible à l'utilisateur de gérer un graphe contenant autant de noeuds que nécessiterait la réalisation d'une peinture. Sans parler de la consommation en mémoire du graphe, et de la surcharge de calcul qu'entraine le parcours de celui-ci.

On ne peut pas non plus se permettre de devoir recalculer toutes les opérations de dessin à chaque Undo/Redo, à chaque édition non destructive et à chaque nouvelle visualisation.

Les frameworks nodaux font donc souvent le choix d'ignorer totalement les opérations de dessin. Celles-ci sont alors effectuées sur les images avant leur entrée dans le graphe, et doivent être gérées par un autre framework tel qu'un framework vectoriel ou bitmap.

Certains frameworks nodaux (GEGL) choisissent de proposer des noeuds qui regroupent une séquence d'opération de dessin en une seule. Ces noeuds sont donc obligés de gérer par des mécanismes interne l'undo/redo. De tels noeuds fonctionnant comme des frameworks bitmaps ils ne peuvent pas gérer des opérations de dessin de trop grande taille.

Ces frameworks permettent donc de peindre sur des images gigapixel, mais chaque trait de peinture doit rester de taille mégapixel, ce qui limite l'utilité de cette fonctionalité dans le cadre d'édition d'images gigapixel.

Édition par pixel

L'édition par pixel est problématique pour les mêmes raisons qui rendent problématique le dessin de primitives.

2.4.5 Framework nodaux, l'état de l'art

The Foundry Nuke Logiciel de composition et post processing vidéo

Blender3D Utilisé pour la composition et le post processing vidéo, logiciel d'animation 3D open source.

GEGL Framework nodal open source utilisé par Gimp $\geq$ 2.7 et le logiciel d'acquisition numérique Gnome Scan.

OpenRaster Format d'image public permettant de sauvegarder des graphes d'opérations et compatible avec GEGL.

2.5 Framework vectoriel

2.5.1 Schéma de représentation de l'image

L'image est décrite par un emsemble de primitives géométriques. Ces primitives sont organisées dans un graphe de scène. Dans un graphe de scène, chaque noeud représente une transformation géométrique et une primitive. La transformation géométrique détermine la position, l'échelle, et la rotation de la primitive et de ses enfants. La primitive est elle décrite par son type et les paramètres qui déterminent sa forme et son apparence.

Le graphe de scène est acyclique. chaque noeud peut avoir plusieurs enfants et sauf exception un seul parent.

(a) l'image (b) Le graphe de scène

FIG. 2.7: Représentation d'une image par graphe de scène

L'ordre des enfants d'un noeud est important puisqu'ils déterminent l'ordre dans lequel ils seront dessinés. Ainsi les derniers enfants d'un noeuds seront dessinés au dessus des premiers.

Chaque framework a sa propre liste de primitives géométriques, mais tous disposent au moins de points, lignes, ploygones, ellipses, texte, et courbes de bézier.

2.5.2 ALGORITHME DE RASTERISATION

L'utilisateur commence d'abord par choisir la région qu'il veut rasteriser et à quelle résolution. Le framework crée ensuite un bitmap de cette taille et résolution de la couleur de fond de l'image.

Pour dessiner un noeud du graphe, on applique sa transformation aux paramètres de sa primitive. On dessine ensuite cette primitive sur le bitmap. On Dessine ensuite chaque noeud enfant dans l'ordre en leur appliquant la transformation de ce noeud.

Cette opération est effectuée de manière récursive en partant du noeud racine de l'image.

Afin d'éviter de dessiner les primitives qui se trouvent en dehors de la région à rasteriser, on associe à chaque noeud une boite qui englobe sa primitive ainsi que celles de tous ses enfants. Si la boite est en dehors de la région, on peut ignorer ce noeud et leurs enfants.

Afin d'exploiter l'accélération matérielle, la plupart des frameworks ne dessinent pas directement les primitives, mais les convertissent en triangles qui sont ensuite dessinés à l'aide du matériel. Cette approche peut également s'avérer bénéfique sans accélération matérielle car le dessin de triangle est souvent beaucoup plus rapide qu'un dessin analytique de la primitive, de plus les détails trop petits pour être vus peuvent être détectés lors de la transformation de la primitive en triangles.

2.5.3 FONCTIONALITÉS ADAPTÉES AUX FRAMEWORKS VECTORIELS

Édition non destructive

La description de la scène étant gardée en mémoire, il est facile de la modifier et d'obtenir ensuite une nouvelle visualisation. L'algorithme de rasterisation ne disposant généralement pas de cache, l'intégralité de la région à visualiser doit être recalculée à chaque modification de l'image. Cela n'est pas un problème tant que la taille du graphe de scène reste raisonnable.

Undo / Redo

L'undo/redo est simple à implémenter et aussi efficace que l'édition non destructive.

Opérations de dessin de primitives

Les frameworks vectoriels sont très pratiques pour éditer un dessin constitué de primitives géométriques. Cependant, la région de visualisation doit être rasterisée à chaque changement d'échelle ou à chaque déplacement de celle-ci. Si le nombre de primitives constituant le dessin est trop grand, cela ne peut plus se faire de manière interactive.

Les frameworks vectoriels sont donc généralement utilisés pour les documents textuels, les cartes, les graphes, ou les dessins abstraits qui peuvent être décrits par un petit nombre de primitives complexes. A contrario, les images peintes utilisent un trop grand nombre de primitives pour que de tels frameworks soient efficaces.

2.5.4 Fonctionalités inadaptées aux frameworks vectoriels

Filtres

En associant les filtres aux noeuds on s'attend à ce que le filtre ne s'applique qu'à ce noeud et à ses enfants. Ceci nécessite de faire une rasterisation de ces noeuds dans un buffer séparé afin d'y appliquer le filtre, puis de réintégrer ce buffer dans le buffer en cours. Et ce de manière récursive selon qu'il y ait des filtres dans les noeuds enfants. Tout cela ralentit nettement la rasterisation, d'autant que le rendu dans de multiples buffer ne fait pas bon ménage avec l'accélération matérielle. Les frameworks proposent parfois de tels filtres, mais les performances suivent rarement.

Le format vectoriel SVG propose des sémantiques de filtres plus complexes. Le fait qu'aucun framework ne les implémente correctement atteste de la difficulté de cette entreprise.

Composition non linéaire

La structure arborescente du scene graphe ne permet pas de décrire des compositions non linéaires.

Édition par pixel

Éditer un pixel requiert de créer une primitive couvrant uniquement ce pixel. Si cela est possible, cela crée rapidement une trop grande quantité de noeuds au fur et a mesure que chaque pixel est modifié. De plus, une telle description pose de gros problèmes d'anti-aliasing pour les résolutions non natives.

Images gigapixel

Les frameworks vectoriels décrivant l'image de manière analytique peuvent gérer des images de toutes tailles et résolutions. De plus la description vectorielle utilise beaucoup moins de mémoire. Ces frameworks sont donc pour l'instant la solution de choix pour décrire des documents de grande taille. Cependant il ne s'agit pas là à proprement parler d'images gigapixel.

Modèles colorimétriques

Tant la description de l'image que l'algorithme de rasterisation sont peu adaptés au support de multiples modèles colorimétriques, à une exception près, le modèle par nuancier. Dans ce cas chaque primitive est associée à une couleur du nuancier, et à une opacité. L'image peut ainsi être envoyée sous forme vectorielle à l'imprimante qui va imprimer les primitives une à une en respectant la structure du graphe de scène. L'image peut également être rasterisée pour obtenir pour chaque pixel la proportion des encres à utiliser.

2.5.5 Framework Vectoriels, l'état de l'art

Adobe Illustrator Logiciel d'édition graphique vectorielle.

Corel Draw Logiciel d'édition graphique vectorielle.

Adobe Flash Logiciel d'animation 2D

Inkscape Logiciel open source d'édition graphique vectorielle.

libart Librairie open source de rendu et d'édition graphique vectorielle, utilisée par Gnome Canvas, Inkscape.

Cairo Librairie open source de rendu et d'édition graphique vectorielle, utilisée par Mozilla Firefox, Webkit, Moonlight, FontForge, Poppler, Gtk+

ai Format d'image vectorielles.

svg Format public d'images vectorielles.

ps Format et langage de programation public décrivant des images vectorielles, obsolète à de nombreux égards mais toujours utilisé pour l'impression de documents.

2.6 Mégatexturing ou Sparse Virtual Textures

Les scènes 3D interactives sont de plus en plus grandes et détaillées. Or les détails sont limités par la taille des textures que l'on peut y appliqué, qui sont elle même limitées par l'espace mémoire disponible sur la carte graphique. On était donc condamné jusqu'il y a peu à utiliser une des trois techniques suivantes, ou de les combiner :

- Une grande texture basse résolution
- Une petite texture haute résolution qui se répète sur les surfaces
- Des textures vectorielles.

Aucune de ces techniques ne peuvent texturer un large environnement de manière convaincante.

Le Megatexturing, appelé aussi Sparse Virtual Texture est donc une technique d'affichage de textures plus qu'un framework à part entière. Le but de cette technique est de permettre l'affichage et l'édition d'images gigapixel en tant que textures de scènes 3D interactives. Une image gigapixel étant suffisamment grande pour couvrir l'intégralité de l'environnement à haute résolution.

2.6.1 Schéma de représentation de l'image

(a) La pyramide de tile représentant l'image complète

(b) La texture virtuelle, composée de tiles de la pyramide

Fig. 2.8: Mégatexture

L'image est représentée par deux structures différentes, l'une se trouvant sur le disque et l'autre en mémoire. Celle qui se trouve sur le disque est la plus volumineuse des deux. Elle est composée d'une pyramide creuse de tiles.

Cette structure consiste en l'image gigapixel complète, divisée en tiles, les régions vides ou facultatives de l'image étant représentées par une absence de tiles, afin d'économiser de la mémoire. La structure contient aussi tous les mipmaps de cette image sous forme de tiles de dimensions identiques.

En mémoire se trouve une texture, appellée *texture virtuelle* qui a la taille maximum supportée par le matériel d'accélération, ce qui est beaucoup moins que l'image originale. Cette texture est composée de tiles se trouvant dans la pyramide.

Affichage de l'image

Pour afficher une texture sur un maillage 3D, on assigne à chaque vertice du maillage une coordonnée u, v correspondant à un pixel de cette texture. de la texture. Comme nous désirons afficher la texture gigapixel, ces coordonnées correspondent aux pixels de cette texture. Celle-ci est cependant trop grande pour pouvoir être utilisée telle quelle.

Pour résoudre ce problème on place les tiles qui nous intéressent dans une texture virtuelle qui est suffisemment petite pour pouvoir être utilisée pour l'affichage. Un *pixel shader* est ensuite utilisé pour traduire lors de l'affichage les coordonnées u, v de la texture gigapixel en les coordonnées m, n de la texture virtuelle.

Pour déterminer quels tiles nous intéressent, on s'aide d'un premier rendu dans lequel on examine quelles régions de la texture gigapixel sont affichées, et à quelle résolution. On sélectionne ensuite les tiles qui couvrent cette zone avec la résolution la plus proche de celle affichée. Comme la résolution de l'écran est plus petite que celle de la taille de la texture virtuelle, on disposera toujours d'une place suffisante pour couvrir la quantité de texture affichée.

Récupérer les tiles sur le disque implique un délai non négligeable. Le megatexturing incorpore des techniques permettant de pallier à ce problème qui sortent du cadre de cette présentation.

Édition de l'image

Pour générer et éditer l'image gigapixel, on utilise la technique précédente pour l'affichage, et on utilise les techniques de framework bitmap pour appliquer les opérations ; Celles-ci sont directement rasterisées dans la pyramide de tiles.

État de l'art du MegaTexturing

Le Megatexturing à été développé par John Carmack d'id Software pour le moteur de jeu idTech4 et idTech5. Cette technique est utilisé par les jeux utilisant ces moteurs. D'autres moteurs tels que Torque ont réimplémenté cette technique.

2.7 Comparaison des différents framework

Le tableau 2.7, (page 23) permet d'avoir une vision globale des fonctionalités supportées par les différents types de framework. On remarque que les frameworks proposent des fonctionalités très différentes, ce qui explique pourquoi chaque domaine d'application tend à préférer un framework en particulier.

Ce tableau permet aussi de comparer les frameworks existants aux fonctionalités adaptées au framework Himalaya qui est présenté dans ce mémoire. Cette comparaison reste en partie théorique puisque toutes ces fonctionalités n'ont pas été implémentées et soumises à des tests utilisateurs.

Néanmoins ce framework propose une combinaison intéressante de fonctionalité en combinant Peinture, Dessin de primitives, Édition non destructive, Composition non linéaire et Images gigapixel. Non repris dans ce tableau est la possibilité – encore théorique – de pouvoir utiliser la technique de megatexture

Fonctionalité	Bitmap	Nodal	Vectoriel	Himalaya
Dessin de primitives	●	⊖	●	●
Peinture	●	○	⊖	●
Filtres colorimétriques	●	●	⊖	●
Filtres spatiaux	●	●	⊖	○
Transformations vectorielles	○	○	●	○
Transformations bitmap	●	●	⊖	⊖
Modes de fusion	●	●	●	●
Modèles colorimétriques	⊖	●	⊖	●
Ton direct	⊖	○	●	○
Modèles étendus	○	●	○	⊖
Undo/Redo	●	⊖	●	●
Édition non destructive	○	●	●	●
Composition non linéaire	○	●	○	●
Édition par pixel	●	○	○	⊖
Indépendance à la résolution	○	⊖	●	○
Images Gigapixel	○	●	●	●
Traitement d'images multiples	⊖	●	○	⊖

● : Fonctionalité adaptée au framework.

⊖ : Fonctionalité implémentable dans le framework, mais de manière limitée ou peu performante.

○ : Fonctionalité inadaptée au framework.

TAB. 2.1: Grille de comparaison des fonctionalités.

en combinaison avec *Himalaya* afin de permettre l'edition non destructive des textures d'une scène 3D temps réel.

La suite de ce mémoire se concentrera sur la présentation de l'architecture et des fonctionalités de base du framework Himalaya, pour se concentrer ensuite sur l'implémentation et l'évaluation par test utilisateurs des fonctionalités de peinture et d'édition d'images gigapixel, combinaison de fonctionalité n'étant proposée par aucun des frameworks existants.

Chapitre 3

Présentation du Framework Himalaya

Himalaya est un framework d'édition d'image s'inspirant des frameworks nodaux et de la technique de megatextures. Il fut conçu par l'auteur avant la réalisation de ce mémoire dans le but de créer un logiciel de peinture permettant l'édition non destructive, la composition non linéaire, et d'être compatible avec la technique de mégatexture afin de pouvoir être utilisé dans le jeu vidéo.

3.1 Structures de données

Nous allons présenter ici les structures de données de base du framework. S'il est souvent difficile de justifier la conception de ces structures sans examiner les algorithmes qui les utilisent, il est encore plus difficile de comprendre les algorithmes sans connaitre ces structures. Nous avons donc fait le choix de présenter et d'expliquer tant que possible ces structures, le reste sera présenté à la section suivante qui vous parlera des différents algorithmes.

Les structures et les algorithmes seront également présentés dans le langage *C*, plutôt que dans des formes plus abstraites. Ceci afin de nous rapprocher de l'implémentation — également réalisée en *C* — et donc de pouvoir plus facilement discuter des performances de celle-ci. Il faut toutefois noter que les structures telles que présentées dans ce rapport sont plus simples que celles de l'implémentation. En effet, la gestion des modèles colorimétriques est ommise, de même que des extensions propres à l'implémentation, pour permettre par exemple une déboguage plus facile, et l'introspection sur les données.

3.1.1 Les Tiles

Comme dans bien d'autres framework, les images d'himalaya sont divisés en grilles de tiles. Un tile est donc un bitmap rectangulaire correspondant à une petite région de l'image. Himalaya fait un plus grand usage des tiles que d'autres frameworks puisque toute opération prend des tiles en entrée et en sortie. Un tile représente donc la plus petite unité de traitement, et un intérèt particulier doit être apporté à sa conception.

Structure du tile

Il y a trois caractéristiques importantes à prendre en compte lorsque l'on conçoit un tile :

Sa taille en mémoire Un premier intérèt des tiles est qu'ils sont une unité de mémoire pouvant être suffisemment petite pour tenir intégralement dans les caches du processeur et ainsi éviter des accès cache invalides fort coûteux en temps de calcul. Les processeurs de différent modèles ont différentes taille de cache et différentes manières de les gérer. Plus le tile est petit, plus il sera rapide de les traiter sur une grande gamme de processeurs.

Cependant, si le tile est trop petit, les gains réalisés par sa taille sont perdu par la surcharge de travail que doit faire le framework pour gérer ces tiles.

En outre, si tous les tiles ont la même taille en mémoire, on réduit les risques de fragmentation, et on peut faire des allocateurs optimisés qui allouent les tiles de manière contigue et réutilisent les tiles libérés.

Sa taille en pixel Si tous les pixels ont la même taille en pixel, alors les images de dimensions égales seront toujours divisées en tiles de la même manière, ce qui rend plus simples beaucoup d'algorithmes du framework. la programmation des opérations en sera également facilitée.

Cependant, un framework moderne doit prendre en compte la gestion de modèles colorimétriques, et niveaux de quantification différents. La représentation d'un pixel peut donc avoir des empreintes mémoire différentes, il faudra donc choisir entre avoir des tiles de même taille pixel, ou des tiles de même taille mémoire.

(a) Un tile normal (b) Un tile aux canaux séparés

FIG. 3.1: Deux structures de tiles

La répartition des pixels dans le tile La manière commune de représenter un pixel au sein d'un bitmap est d'avoir toutes ses composantes placées dans des emplacement contigus. Une autre manière est de placer chaque composante dans un bitmap séparé. Un tile étant alors constitué de plusieurs bitmaps. L'intéret de cette technique est que les tiles de même quantisation ont la même taille en pixel, et les bitmaps les constituant la même taille en mémoire. Un autre intérèt est qu'il est maintenant beaucoup plus facile de programmer des opérations qui peuvent gérer un nombre quelconque de composantes par pixel. Ces deux possibilités sont représentées par le schéma 3.1, page 26

Cependant, cela implique que l'accès à un pixel nécessite autant d'accès mémoires que de composantes, ce qui ralentit l'exécution du programme.

Comparaison des tiles

Afin de déterminer quel type de tile, et quelles dimensions sont les plus adéquates, plusieurs expériences ont été réalisées. Le tableau 3.1.1, page 27 représente le résultat d'expériences qui résument les concepts précedemment exposés. Dans ce tableau, deux opérations sont testées : *colorfill* consiste à remplir une image de 8192^2 pixels *RGBA* 8bits/composante d'une couleur unie. *blend* consiste en fusionner deux images de 8192^2 pixels *RGBA* 8 bits/composante par opacité. Ces deux opérations sont utilisées très couramment pour la réalisation de peintures.

Ces opérations sont appliquées selon des schémas différents : *colorfill_full* remplit tous les tiles de l'image, alors que *colorfill_single* remplit un tile choisit au hasard autant de fois qu'il y a de tiles dans l'image. Le même nombre de pixels

Largeur des tiles (pixel)		8	16	32	64	128	256	512	1024	2048
Test	Canaux	Temps de calcul moyen sur 5 expériences (secondes)								
colorfill_single	Normal	0.78	0.64	0.58	0.57	0.49	0.28	0.28	0.28	0.31
colorfill_full	Normal	1.14	0.7	0.65	0.63	0.55	0.44	0.47	0.44	0.51
colorfill_single	Séparé	1.30	1.06	1.04	1.06	0.97	0.79	0.79	0.79	0.80
colorfill_full	Séparé	2.14	1.39	1.16	1.12	1.10	1.03	1.00	1.01	1.01
blend_single	Normal	-	3.12	2.98	3.0	3.08	4.16	13.58	19.97	22.60
blend_inter	Normal	-	5.31	5.20	4.96	5.12	4.85	13.49	20.46	20.64
blend_full	Normal	-	5.39	5.19	5.17	5.0	5.42	13.15	20.44	21.12
blend_single	Séparé	-	6.29	5.64	5.39	6.61	6.50	18.67	27.48	28.92
blend_inter	Séparé	-	6.55	5.69	5.50	6.55	6.74	20.22	27.77	27.48
blend_full	Séparé	-	6.68	5.82	5.45	6.59	6.78	18.39	27.41	27.23

est donc traité dans les deux tests, mais le second devra accéder régulièrement à de nouvelles zones mémoire.

Même chose pour les operations *blend* : *blend_full* fusionne les tiles des deux images, *blend_single* fusionne deux tiles sélectionnés au hasard, *blend_intermediate* fusionne toute l'image sur un seul tile. Ce dernier test est fort représentatif de la manière dont les tiles sont utilisés dans Himalaya.

Ensuite les tiles normales sont comparées aux tiles à canaux séparés.

Analyse des résultats

On peut tirer plusieurs conclusion de ces expériences :

- Les tiles doivent être les plus grands possibles, pour diminuer le nombre d'appels de fonctions, mais ne doivent pas être trop grands, sinon les problèmes de cache dégradent fortement les performances. Cet effet est mis en exergue par le graphe 3.1.1, page 27, qui représente les différents temps de calcul du test *blend* en fonction de la largeur des tiles.
- Les tiles à canaux séparés sont jusque 2 fois plus lents que les tiles normaux.
- Le schémà d'accès aux tiles en mémoire a un impact non négligable mais plus marqué sur les tiles normaux.
- Le temps pour faire la fusion d'un même nombre de pixels peut différer de 970% selon le type de tile, sa taille et la manière dont il est utilisé, puisque cette opération représente la grande majorité du temps de calcul du framework, ce choix est particulièrement important.

Enfin, il ne faut pas oublier que ces expériences ommettent deux facteurs importants : Le fait que la gestion des tiles dans le framework est bien plus lourde, ce qui favorise les tiles plus grands, et le fait que les tiles plus petits permettent plus de précision dans la localisation des opérations, ce qui permet de réduire le nombre de pixels accédés à chaque opération.

Déterminer l'importance de ces deux facteurs requiert d'avoir des données d'utilisation du framework représentatives, par exemple avec des tests utilisateur.

Listing 3.1: Définition des hlFrameNodes

```
typedef struct hl_frame_node{
        /* abscisse du tile */
        int tx;
        /* ordonnee du tile */
        int ty;
        /* tile  */
        hlTile *tile;
        /* Noeuds enfants */
        hlFrameNode *tl;
        hlFrameNode *tr;
        hlFrameNode *bl;
        hlFrameNode *br;
}hlFrameNode;
```

Structure finale

Le choix s'est porté sur des tiles normaux de taille fixe de 32×32 pixels, et ce quelque soit le modèle colorimétrique utilisé.

À noter que pour simplifier la discussion, le reste du rapport par de l'hypothèse que tous les tiles utilisent le modèle colorimétrique RGBA 8bits, qui est également le seul modèle utilisé lors des tests utilisateurs. Dans les discussions suivantes les tiles auront donc la même taille en pixel, mais également la même taille en mémoire.

3.1.2 Les Frames

La Frame est la structure de donnée qui organise les tiles. Elle doit remplir plusieurs fonctions : stocker une image gigapixel, stocker les mipmaps de cette image, et servir de cache aux résultats des opérations, ainsi que les mipmaps de ceux-ci.

L'intérèt des mipmaps est que pour un coût mémoire de seulement 33% de l'image originale, ils permettent d'accéder à une sous région de l'image à une échelle quelconque en un temps ne dépendant que de la taille de cette sous-région, et non de la taille de l'image. Cette propriété est indispensable pour pouvoir travailler sur des images giga-pixel.

La Frame est une pyramide de tile creuses. Cette structure n'est pas nouvelle et est utilisée dans la pluspart des logiciels devant gérer des images gigapixel et leurs mipmaps. Les formats d'échange standard des photos satellites sont par exemple basés sur ces structures.

L'intérèt d'avoir une pyramide creuse est de pouvoir stocker des tiles individuels, ce qui est indispensable pour pouvoir s'en servir comme d'une cache.

Il existe plusieurs structures permettant l'organisation de tiles en pyramide creuses. La plus populaire est le quadtree, et c'est celle qui fut choisie pour l'implémentation des Frames.

Les quadtrees

Les quadtrees sont composés de noeuds ayant chacun une référence optionelle vers un tile, et de quatre références vers les noeuds enfants.

Chaque ensemble de noeuds de la même profondeur représente ainsi un bitmap à l'échelle deux fois plus petite que l'ensemble de noeuds à la profondeur suivante.

Une définition de la structure des noeuds se trouve au listing 3.1, page 28

Placer une image dans le Quad-Tree

Afin d'éviter d'avoir des Quad-Trees inutilement profond, les images sont stoquées au niveau le moins profond pouvant les contenir, soit $ceil(\log_2(max(size_x, size_y)/32))$, $size_x$,$size_y$ représentant la largeur et la hauteur de l'image en pixels. Cette profondeur est ensuite stoquée dans la Frame afin de pouvoir identifier le niveau correspondant à la résolution native.

Référencer les Tiles dans le Quad-Tree

Les tiles sont référencés par trois coordonnées (t_x, t_y, t_z). t_x, t_y représente les coordonnées spatiales du tile, t_x valant zéro à l'extrémité gauche du bitmap, et étant positif à droite. t_y vaut zéro à l'extrémité supérieure du bitmap et est positif vers le bas, et ce quelque soit l'échelle représentée.

t_z représente le niveau d'échelle. Il représente habituellement la profondeur du noeud à laquelle on trouve le tile. Nous avons choisi d'utiliser un système différent, t_z vallant 0 au niveau correspondant à la résolution native, et étant positif pour les échelles intérieures. L'intérèt de ce système est que la référence d'un tile est indépendante de sa profondeur dans le quadtree, qui peut changer lorsque l'on insère ou retire des tiles dans celui-ci.

On peut s'intéresser à la valeur maximale de t_z qui dépend de la taille de l'image stockée dans le quadTree. Comme nous voulons pouvoir indexer les pixels indépendemment, la largeur de l'image ne peut dépasser *MAX_INT*, Ce qui correspond à une valeur maximale de t_z de 26 sur les architectures 32bits, avec des tiles de 32 pixels de coté.

Les coordonnées négatives

Lorsque la Frame sert à stocker un bitmap chargé depuis le disque, les pixels le constituant, et donc les tiles, ont toujours des coordonnées t_x, t_y positives. Cependant, il est pratique de pouvoir également stocker des pixels de coordonnées négatives, afin de disposer du plan complet pour pouvoir stocker le résultat de transformations géométriques. Pour ce faire, la Frame stoque un Quad-Tree par quadrant, et les tiles aux coordonnées négatives sont redirigées vers les Quad-Tree correspondant.

Le Tile de fond

Lorsque la frame est utilisée pour contenir un bitmap, celle-ci dispose d'un Tile de fond. Celui-ci n'est pas repris dans les Quad-Tree et est d'une couleur unie correspondant au 'fond' de l'image, habituellement une couleur transparente. Lorsque l'on tente d'accéder à une coordonnée qui ne correspond à aucun noeud des Quad-Trees, ce que l'on a atteint une zone correspondant au fond de l'image, et le tile de fond est renvoyé. Ceci permet de réduire l'espace mémoire consommé par les zones vides des images.

Insertion, suppression et accès aux tiles

L'insertion, la suppression, et l'accès à un tile se fait en $O(n)$ ou n est la profondeur à laquelle se trouve le tile dans le Quad-Tree. Comme n ne dépasse pas 26, le temps d'exécution de ces fonctions est borné, et on peut donc considérer que'elles ont une complexité de $O(1)$.

De plus l'insertion et la suppression augmentent ou diminuent automatiquement la taille des quad-tree afin de s'assurer que la taille de l'arbre reste minimale. Ceci nécessite cependant de stocker à chaque noeud sa coordonnée t_x, t_y, ce qui augmente leur taille de 40%. TODO / evaluer l'espace pris.

Surchage mémoire

Les FrameNode ont un poids de 28 Bytes, ce qui représente 2,66% de la taille des tiles les plus petits(Alpha 8bit), 0.67% de celle des plus communs(RGBA

Listing 3.2: Définition des hlFrames

```
typedef struct hl_frame{
        /* largeur en pixel du bitmap */
        int sizex;
        /* hauteur en pixel du bitmap */
        int sizey;
        /* tile de fond */
        hlTile *bg;
        /* QuadTrees */
        hlFrameNode *tl;
        hlFrameNode *tr;
        hlFrameNode *bl;
        hlFrameNode *br;
}hlFrame;
```

Listing 3.3: API Publique des hlFrame

```
/* Place un tile dans f aux coordonnees tx,ty,tz */
void    hlFrameTileSet(hlFrame *f, hlTile *t, int tx, int ty, int tz);
/* Enleve et renvoie le tile tx,ty,tz de f */
hlTile *hlFrameTileRem(hlFrame *f, int tx, int ty, int tz);
/* Renvoie le tile tx,ty,tz, si il existe, NULL sinon. */
hlTile *hlFrameTileGet(const hlFrame *f, int tx, int ty, int tz);
/* Renvoie le tile tx,ty,yz si il existe, le tile BG sinon. */
hlTile *hlFrameTileRead(const hlFrame *f, int tx, int ty, int tz);
```

Listing 3.4: Définition des hlImages

```
typedef struct hl_image{
        /* Reference vers la derniere operation de la pile*/
        hlOperation *top;
        /* Reference vers la frame contenant le bitmap
         * a modifier */
        hlFrame *source;
}hlImage;
```

8bit) et 0.14% de celle des plus grands(CMYKA 32bit). Dans le meilleur des cas, le stockage d'un tile ne nécessite qu'un noeud, et 26 dans le pire des cas. On observe lors des tests utilisateurs qu'en moyenne le stockage d'un tile nécessite TODO noeuds, ce qui représente TODO pourcent.

Définition de l'hlFrame

On trouve au Listing 3.2, page 3.2 une définition de la structure des hlFrame. On trouvera également au Listing 3.3, page 3.3, l'API publique des hlFrame.

3.1.3 L'HLIMAGE

Si la frame représente une cache ou un bitmap, l'hlImage représente une image à proprement parler. L'hlImage est composée principalement de deux choses, une Frame représentant la source, c'est à dire une image de départ à modifier, et une pile d'opérations qui modifient l'image. Une défintion de la structure se trouve au Listing 3.4, page 30

FIG. 3.2: une hlImage, avec la source et la pile d'opération

Listing 3.5: API des hlImages

```
/* Rasterise un tile de l'image */
hlTile *hlImageRenderTile(hlImage *img, int tx, int ty, int tz);
/* Insere une operation au sommet de la pile */
void hlImagePushOp(hlImage *img, hlOperation *op);
/* Enleve une operation du sommet de la pile */
hlOperation *hlImagePopOp(hlImage *img);
/* Enleve une operation de la pile et la renvoie */
hlOperation *hlImageRemOp(hlImage *img, int index);
/* Insere l'operation dans la pile */
void hlImageInsertOp(hlImage *img, hlOperation *op, int index);
/* Renvoie l'indice auquel se trouve l'operation d'uid opUid */
int hlImageOpIndex(hlImage *img, int opUid);
```

La pile d'opérations

La pile d'opération représente l'ensemble des opérations qui vont modifier l'image source pour obtenir l'image dessinée, les opérations du bas de la pile s'appliqaunt en premier. Il est possible de manipuler cette pile par l'API publique de l'hlImage. Les opérations les plus communes sont l'ajout et le retrait d'opération au dessus de la pile, l'insertion et la suppression d'opérations se trouvant au milieu. Cette API est résumée dans le listing 3.5, page 31

Les opérations se trouvant à l'intérieur de la pile peuvent être accédées par leur indice de position dans la pile, mais aussi leur *uid*, identifiant propre à chaque opération.

La pile d'opération est implémentée par une liste simplement chainée. Chaque opération référence l'opération *précédente*. L'*hlImage* référence uniquement la *dernière* opération. Les raisons de cet ordre apparaitront clairement à la vue de l'algorithme de rasterisation.

3.1.4 LES HLOPERATIONS

Les hlOperations sont les objets qui représentent toutes les opérations qui permettent de modifier une image. Une définition de la structure se trouve au Listing 3.6, page 32. Certaines opérations particulières pouvant étendre cette structure avec des champs supplémentaires, qui seront détaillés dans les sections appropriées.

Listing 3.6: Définition des hlOperations

```
typedef struct hl_operation{
        /* Reference vers la classe de l'operation */
        hlOpClass  *class;
        /* Reference vers l'operation precedente. */
        struct hl_operation *down;
        /* Frame qui sert de cache aux resultats de
         * l'operation */
        hlFrame *cache;
        /* Reference vers l'image a laquelle appartient
         * l'operation */
        hlImage *image;
        /* HL_MODIFABLE, HL_CACHING */
        int      flags;
        /* Un compteur de references */
        int      refcount;
        /* Un identifiant unique a l'operation*/
        int     uid;
        /* Les parametres de l'operation */
        void    *params;
}hlOperation;
```

Listing 3.7: Définition des classes d'opérations

```
typedef struct hl_op_class{
        /* HL_COLORFILTER, HL_FILTER, HL_DRAW, HL_BLEND */
        int category;
        /* Numero de l'operation*/
        int id;
        /* le nombre de parametres flottants */
        int float_param_count;
        /* le nombre de parametres entiers */
        int int_param_count;
        /* le nombre de parametres de couleur */
        int color_param_count;
        /* le nombre de parametres d'images */
        int image_param_count;
}hlOpClass;
```

Cela représente un minimum de 32Bytes[1] par opération sur une architecture 32bits. Une hlOperation est donc suffisemment légère pour que l'on puisse en utiliser plusieurs millions sans problèmes de mémoire.

La classe d'opération

La classe d'opération contient la description d'un type d'opération, et tout ce qui est commun à ces opérations, Il n'existe qu'une seule instance de chaque classe d'opération. La définition de ces classe est donnée au listint 3.7, page 32.

Les champs des classes d'opérations méritent d'être expliqués plus en détails

`category` . Les opérations sont divisées en plusieurs catégories, selon les propriétés et les paramètres nécessaires à l'application de l'opération. Ceci permettra d'appeler une méthode spécialisée pour chaque catégorie d'opération.

[1] La structure telle qu'implémentée possède de nombreux champs supplémentaires à de fins de déboguage et fait 92Bytes

Ces catégories correspondent également aux différentes catégories d'opération de dessin que nous avons vu au premier chapitre.

HL_COLORFILTER : Les filtres colorimétriques. Ces filtres ne sont pas dépendants de la position du tile ou de son échelle, et s'appliquent sur toute l'image.

HL_FILTER : Les filtres spatiaux. Ces filtres dépendent de l'échelle, de la position du tile, et ne peuvent modifier un tile sans connaitre les données des tiles adjascents.

HL_DRAW : Le dessin de primitives. Ces opérations dépendent de l'échelle et de la position du tile, mais ne s'appliquent que sur une sous-région de l'image. Ces filtres dépendent de l'échelle et de la position du tile, et ne s'appliquent que sur une sous région de l'image.

HL_BLEND] Les opéartions de fusion. Ces opérations dépendent de la position du tile et de son échelle, ainsi que d'une autre *hlImage*

HL_COLORMODEL] Les opérations pour lesquelles les tiles en entrée n'ont pas le même modèle colorimétrique qu'en sortie.

id Le numéro de l'opération, qui l'identifie dans sa catégorie. Par exemple, Le dessin de cercle — `HL_DRAW_CIRCLE` — dans la catégorie `HL_DRAW`

***_param_count** : Le nombre de paramètres de chaque type. Les opérations sont limitées à quatre types de paramètres, les flottants, les entiers, les couleurs, et les images. Cette information est nécessaire pour pouvoir calculer la taille des paramètres de la fonction. L'implémentation utilise un mécanisme similaire mais plus complexe permettant une introspection sur les paramètres.

3.2 Algorithmes

3.2.1 Dessin

Dans Himalaya, le dessin et la rasterisation sont deux opérations séparées et indépendantes. Le dessin consiste simplement à ajouter une opération au dessus de la pile.

3.2.2 Rasterisation

L'algorithme de rasterisation d'himalaya s'inspire de l'algorithme de rasterisation des frameworks nodaux, et du raytracing. Là ou les algorithmes de rasterisation des frameworks bitmaps et vectoriels appellent les opérations dans l'ordre avec lequel elles ont été ajoutées, himalaya les appelles dans l'ordre inverse ; depuis la dernière jusqu'à la première. En outre, comme dans le raytracing, l'algorithme de rasterisation d'himalaya peut se faire indépendemment pour chaque pixel, et donc pour chaque tile.

Ceci a plusieurs intérèts :

- Si une opération a son résultat en cache, on peut réutiliser ce résultat sans examiner les opérations antérieures.
- Si une opération est opaque, c'est à dire qu'elle masque le résultat des opérations précédentes, on peut l'appliquer sans examiner les opérations entérieures.
- Si la rasterisation d'un tile est indépendant de celui de son voisin on peut utiliser un thread par calcul de tile, et donc exploiter facilement le multithread.
- Si la rasterisation d'un tile est indépendant de celui de son voisin, on peut identifier les tiles de la région à visualier, et ne rasteriser que ceux-ci. Le temps de rendu est donc proportionnel à la région à visualiser, et non à la taille de l'image sur laquelle on travaille. Cette propriété est indispensable pour travailler sur des images gigapixel.

Le listing 3.8, page 34 Montre une version simplifiée de l'algorithme de rasterisation. Examinons le en détail :

Listing 3.8: Rasterisation d'opérations

```
1  hlTile *hlOpRenderTile(hlOperation *op, int tx, int ty, int tz){
2          hlTile *tile = hlFrameTileGet(op->cache,tx,ty,yz);
3          if(tile){
4                  if(hlOpCaching(op,tx,ty,yz)){
5                          return hlTileCopy(tile);
6                  }else{
7                          return hlFrameTileRem(op->cache,tx,ty,yz);
8                  }
9          }else{
10                 if(hlOpOpaque(op,tx,ty,yz)){
11                         tile = hlTileNew();
12                         hlOpRasterise(tile,op,tx,ty,yz);
13                 }else{
14                         if(op->down){
15                                 tile = hlOpRenderTile(
16                                   op->down,tx,ty,yz);
17                         }else{
18                                 tile = hlTileCopy(hlFrameTileGet(
19                                   op->image->source,tx,ty,tz));
20                         }
21                         hlOpRasterise(tile,op,tx,ty,yz);
22                 }
23                 if(hlOpCaching(op,tx,ty,tz)){
24                         hlFrameTileSet( op->cache,
25                                 hlTileCopy(tile),tx,ty,yz);
26                 }
27                 return tile;
28         }
29 }
30 hlTile *hlImageRenderTile(hlImage *img, int tx, int ty, int tz){
31         if(img->top){
32                 return hlOpRenderTile(img->top,tx,ty,tz);
33         }
34         return NULL;
35 }
```

Spécification :

pre –

tx, ty, yz sont les coordonnées du tile à rasteriser. Ces coordonnées correspondent aux tiles du résultat, et non à celles de la Frame source.

op est l'opération que l'on désire rasteriser.

returns l'algorithme renvoie un nouveau tile qui est le résultat de toutes les opérations de l'image antérieures et égales à op, appliquées sur les données de la Frame source. Il faut libérer ce tile après utilisation.

post Le résultat de cette opération est mis dans la cache de celle-ci.

Analyse ligne par ligne

2. La première chose à faire est de récupérer le résultat de l'opération dans la cache.

3-8. Si le résultat était dans la cache, on le renvoie

4-5. Si l'opération doit encore garder ce tile en cache, on renvoie une copie.

6-7. Sinon on l'enlève de la cache et on le renvoie.

9-27. Le résultat n'étant pas dans la cache, il faut le calculer.

10-12. On fait appel à l'opération pour savoir si son effet est opaque, et donc indépendant du résultat des opérations précédentes. Si c'est le cas, on crée un tile vide (ligne 7) et on applique l'opération sur ce tile.

13-22. L'opération n'est pas opaque, il faut donc l'appliquer sur le résultat des opérations précédentes.

14-16. S'il y a une opération précédente on récupère son résultat par un appel récursif de cette fonction sur l'opération précédente.

17-19. S'il n'y a pas d'opération précédente c'est que l'on est à la première opération, qui s'applique sur la Frame Source.

21. *tile* vaut désormais le résultat des opérations précédentes. On peut donc lui appliquer l'opération en cours.

23-25. S'il est intéressant de mettre en cache à cette opération, on met une copie du tile en cache.

`hlOpRenderTile(..)` n'est cependant pas accessible à l'utilisateur, puisque celui ci n'a pas d'accès direct aux opérations de la pile. Pour rasteriser l'image il utilise `hlImageRenderTile(..)`

Complexité spatiale et temporelle

La complexité temporelle de l'application d'une opération sur un tile est de $O(1)$ puisque les tiles possèdent un nombre fixe de pixel. La complexité temporelle de l'insertion et du retrait dans la chache est également de $O(1)$. Chaque appel récursif à `hlOpRenderTile(...)` Se fait donc en $O(1)$. La complexité temporelle de la rasterisation d'un tile est donc de $O(n)$ ou n est le nombre d'opérations ajoutées au sommet de la pile depuis la dernière rasterisation de ce tile.

Pour rasteriser une région, de l'image, il faut identifier les tiles correspondant à cette région, et les rasteriser un par un. La complexité temporelle pour rasteriser une région de l'image est donc de $O(m * n)$ ou m est le nombre de tiles dans la région, et n le nombre d'opérations ajoutées au sommet de la pile depuis le dernier rendu de cette région.

Cette complexité est particulièrement adaptée à la peinture, puisque très peu d'opération sont ajoutées sur la pile entre chaque rendu. Elle est également particulièrement adaptée à l'édition d'images gigapixel, puisque la taille de l'image n'intervient pas.

L'algorithme de rasterisation ne met en cache que lorsque l'opération le requiert, et enlève automatiquement les tiles des caches lorsque les opérations ne le requièrent plus. Cela garantit qu'aucun tile inutile ne reste en mémoire. La complexité spatiale dépend donc du nombre d'opérations qui requièrent de mettre en cache. Ceci sera détaille à la section *Gestion de la cache*

Catégories d'opérations

L'algorithme tel que présenté n'est pas adapté à toutes les catégories d'opérations. Les filtres spatiaux nécessitent par exemple plusieurs tiles résultant de l'opération précédente pour en calculer un seul. Quand aux opérations de fusion, elles nécessitent d'obtenir le tile d'une autre image. Ces opérations nécessitent donc un algorithme de rasterisation légèrement différents. Le fonctionnement des opérations de fusion est détaillé dans une section suivante.

3.3 États

La définition de l'hlImage telle que présentée jusqu'à maintenant n'est pas complète. En effet, comment modifier la pile d'opération sans corrompre les caches des opérations ? Comment défaire et refaire facilement des changements dans la pile ?

Pour permettre cela nous avons besoin d'introduire un mécanisme d'état. Un état est un *uid* qui nous permet de sauvegarder et rétablir la pile d'opération

Listing 3.9: Définition des hlImages avec états

```
typedef struct hl_image{
        /* Reference vers la derniere operation de la pile*/
        hlOperation *top;
        /* Reference vers la frame contenant le bitmap
         * a modifier */
        hlFrame *source;
        /* Hashtable associant les etats aux operations */
        HashTable *statelib;
        /* Etat en cours */
        hlState state;
}hlImage;
```

et les paramètres de celle-ci. Ainsi il est possible de sauvegarder l'état de l'hlImage, d'ajouter, d'enlever, de modifier des opérations, et de récupérer l'hlImage telle qu'elle état en chargeant l'état.

Les états sont manipulés avec les fonctions suivantes :

Fonction de manipulation d'États

- `hlState hlStateSave(hlImage *img)` sauvegarde la pile d'opération dans un nouvel état.
- `hlState hlStateGet(hlImage *img, hlState s)` renvoie l'état dans lequel se trouve l'image.
- `void hlStateLoad(hlImage *img, hlState s)` rétablit l'image à un état.
- `void hlStateRem(hlImage *img, hlState s)` supprime un état de l'image.

L'API publique de l'hlImage, combinée à l'API d'état est déjà suffisemment complète pour pouvoir réaliser un logiciel de peinture.

3.3.1 Implémentation des états

Les états sont des références vers des opérations de l'image. Ces références sont implémentées par une table de hachage qui associe à chaque état une opération.

Une fois qu'une opération est référencée par un état, elle ne peut plus être modifiée. Il en va de même pour toute opération antérieure à celle référencée, puisqu'une modification de celle-ci pourrait entrainer une modification du résultat de l'opération référencée.

Pour savoir si une opération est référencée — directement ou indirectement — par un état, chaque opération contient un compteur de référence qui compte le nombre d'états les référençant.

La structure de l'hlImage doit donc être modifiée pour contenir cette table de hachage, ainsi qu'une variable indiquant l'état en cours. Une définition de cette nouvelle structure se trouve au Listing 3.9, page 36.

3.3.2 L'État zéro

Le champ `hlImage.top` ne change pas de fonction avec le système d'état. Cette référence pointe toujours vers l'opération de l'état en cours, qui correspond à la variable `hlImage.state`.

Une fois un état chargé, il reste possible d'ajouter des opérations sur la pile, mais celles-ci ne sont pas sauvegardées automatiquement dans un état. Il est en effet totalement inutile d'avoir un état correspondant à chacune des opérations. Lorsque le dessus de la pile ne correspond pas à un état sauvegardé, l'état en cours vaut la valeur spéciale *zéro*.

La figure 3.3.1, page 37 résume cette situation. On a ici deux états A,B, et une opération non sauvegardée. L'état en cours vallant 0.

FIG. 3.3: une hlImage, avec une pile d'opérations et deux états

3.3.3 SAUVEGARDER L'ÉTAT

Pour sauvegarder un état, on crée un nouvel état, on l'associe à l'opération référencée par `hlImage.top` dans $lstinline*hlImage.statelib*, et on augmente ensuite de 1 le compteur de référence de l'opération référencée, ainsi que de toutes les opérations antérieures. Sauvegarder l'état se fait donc en $O(n)$ où n est le nombre d'opérations antérieures à l'opération à sauvegarder.

3.3.4 CHARGER UN ÉTAT

Pour charger un état, on commence par supprimer toutes les opérations non sauvegardées. On place ensuite dans `hlImage.top` une référence vers l'opération correspondant à l'état, et on met cet état dans `hlImage.state`. Charger un état se fait donc en $O(n)$ où n est le nombre d'opérations non sauvegardées.

3.3.5 SUPPRIMER UN ÉTAT

Pour supprimer un état, on diminue de 1 le compteur de référence de l'opération référencée, ainsi que de toutes les opérations antérieures. Si ce compteur vient à zéro, l'opération est supprimée. L'état est ensuite enlevé de `hlImage.statelib`. Supprimer un état se fait donc en $O(n)$ ou n est le nombre d'opérations sauvegardées par cet état.

3.3.6 MODIFIER LA PILE

Lorsqu'une pile d'opération est enregistrée par un état, on ne peut pas modifier ni les opérations de cette pile, ni l'ordre dans lequel elles se trouvent. Comment alors combiner le système d'état avec l'API de l'hlImage ?

Le principe est simple, si l'on ne peut pas modifier la pile sauvegardée, on peut en revanche modifiée une pile non sauvegardée. On va donc créer une nouvelle pile identique à celle que l'on désire modifier, mais non référencée par un état. devient alors possible de modifier l'opération ou la pile. On appelle ce mécanisme le forking, par analogie au mécanisme similaire des système de contrôle de version de logiciels.

Le forking

Le forking permet de dupliquer la pile jusqu'à une certaine opération, cette opération étant antérieure ou égale aux opérations que l'on veut modifier dans la pile. On dit alors que l'on forke une opération.

Voici comment se déroule le forking d'une opération.

Si l'opération à forker n'est pas référencée Dans ce cas, les opérations suivantes ne sont pas non plus référencées, et rien ne nous empèche de les modifier, il n'est donc pas nécessaire de forker.

Si L'opération à forker est référencée Dans ce cas, on va dupliquer l'opération à forker, ainsi que toutes les opérations suivantes dans une nouvelle pile d'opération. Nous mettons ensuite cette nouvelle pile comme étant la pile courante.

L'algorithme de forking va parcourir la pile d'opération depuis le sommet de celle-ci.

- Si l'opération rencontrée est non référencée, on supprime sa cache, celle-ci sera en effet invalidée par les modifications des opérations antérieures.
- Si l'opération rencontrée est référencée, elle est dupliquée et l'opération se trouvant au dessus dans la pile est modifiée pour pointer vers le duplicata. L'opération dupliquée voit aussi son compteur de référence mis à zéro, et sa cache supprimée, mais garde le même uid.
- Si l'opération rencontrée est référencée, et correspond au sommet de la pile en cours, le sommet de la pile est pointé vers le duplicata de l'opération.
- Si l'opération rencontrée est celle que l'on désire forker, on arrète le parcourt de la pile, après l'avoir dupliquée.

FIG. 3.4: Le forking d'une opération

Le résultat du forking est illustré à la figure 3.3.6, page 38. Le schéma (a) montre la pile et l'opération que l'on désire forker. Le schéma (b) montre la nouvelle pile.

On constate donc que la pile n'a été dupliquée que depuis le sommet en cours, jusqu'à l'opération à forker, le reste de la pile étant gardé commun, ceci permettant entre autre de réutiliser les caches des opérations communes.

En fait de pile, les opérations ressemblent désormais à une forèt. Mais il est impossible pour l'utilisateur du framework de s'en rendre compte. N'est visible depuis l'API publique qu'une simple pile et des états.

L'intérèt de pouvoir référencer les opérations par uid apparait maintenant clair : Cela permet de référencer en tant que même opération toutes les instances de celle-ci dans les différents états.

Listing 3.10: Rasterisation d'opérations

```
hlTile *hlStateRenderTile(hlImage *img, hlState s, int tx, int ty, int tz){
        hlOperation *op = hlStateGetOp(img,s);
        return hlOpRenderTile(op,tx,ty,tz);
}
```

Le forking permet donc de concilier état et modifications pour une complexité temporelle et spatiale de $O(n)$ ou n est le nombre d'opérations sauvegardées qui suivent l'opération à modifier.

3.4 Opération de Fusion

L'opération de fusion permet d'insérer et de fusionner un tile d'une autre image dans la pile. L'image externe doit donc être référencée par un état, et celui ci ne peut pas être l'état zéro, puisque celui-ci ne correspond pas à une image constante. Insérer un tile d'une image, à un état spécifique implique d'être capable de rasteriser celle-ci à l'état désiré sans changer son état en cours. Ceci est fait par la fonction `hlStateRenderTile(...)` que l'on trouve au listing 3.10, page 39.

Cette fonction nécessite l'usage de `hlStateGetOp(...)` qui permet de récupérer l'opération associée à l'état, et la fonction `hlOpRenderTile(...)` que l'on trouve au listing 3.8, page 34. Une fois cette fonction disponible, on peut adapter `hlOpRenderTile(...)` aux opérations de fusion ; `hlOpBlendTile(...)`. On Trouve le résultat de cette adaptation au listing 3.11, page 40. Les modifications se concentrent dans les lignes 21 à 25 :

21-22. On récupère l'image et l'état à fusionner dans l'image courante.

23. On récupère un tile de cette image aux mêmes coordonnées que l'image courante.

24. On fusionne les deux tiles selon les paramètres de l'opération

25. On libère le tile de l'image externe dont on a plus besoin.

3.5 Gestion de la cache

La gestion de la cache peut se résmuer en deux questions :

3.5.1 Quand et où mettre un tile dans la cache

Plus un tile est caché proche du sommet de la pile, plus il est efficace. Le sommet de la pile semble donc être un endroit de choix pour cacher les opérations.

Il est également utile de cacher les tiles sur les opérations référencées directement par un état, puisque celles-ci se retrouveront au sommet de la pile dès que l'état correspondant sera chargé.

Si l'on sait qu'une opération sera l'objet de modifications, il peut être intéressant de cacher à l'opération antérieure, afin d'accélérer la rasterisation après un fork. Pour savoir quelles opérations sont fréquemment modifiées, on s'en remet à l'utilisateur, qui peut marquer les opérations qu'il prévoit de modifier.

La consommation de mémoire est donc de l'ordre du nombre d'états et d'opérations modifiables se trouvant dans la pile.

3.5.2 Quand et quel tile enlever de la cache

Tiles invalides

Il y a certains tiles que l'on *doit* enlever de la cache ; ceux qui ne sont plus valides. Il n'existe qu'une seule façon d'invalider la cache d'une opération : modi-

Listing 3.11: Rasterisation d'opérations

```
hlTile *hlOpBlendTile(hlOperation *op, int tx, int ty, int tz){
        hlTile *tile = hlFrameTileGet(op->cache,tx,ty,yz);
        if(tile){
                if(hlOpCaching(op,tx,ty,yz)){
                        return hlTileCopy(tile);
                }else{
                        return hlFrameTileRem(op->cache,tx,ty,yz);
                }
        }else{
                if(hlOpOpaque(op,tx,ty,yz)){
                        tile = hlTileNew();
                        hlOpRasterise(tile,op,tx,ty,yz);
                }else{
                        if(op->down){
                                tile = hlOpRenderTile(
                                  op->down,tx,ty,yz);
                        }else{
                                tile = hlTileCopy(hlFrameTileGet(
                                  op->image->source,tx,ty,tz));
                        }
                        hlImage *img
= hlBlendOpGetImg(op);
                        hlState s
= hlBlendOpGetState(op);
                        hlTile *tile2 = hlStateRenderTile(img,s,tx,ty,tz);
                        hlBlendOpRasterise(op,tile,tile2);
                        hlTileFree(tile2);
                }
                if(hlOpCaching(op,tx,ty,tz)){
                        hlFrameTileSet( op->cache,
                                hlTileCopy(tile),tx,ty,yz);
                }
                return tile;
        }
}
```

fier une opération antérieure. Ces tiles sont en fait automatiquement supprimés par les fonctions de modification de la pile.

Suppression d'états

Lorsqu'un état est supprimé, l'opération correspondante ne se trouvera plus jamais au dessus de la pile. On peut alors se demander s'il est intéressant de garder la cache de cette opération. Si l'opération (A) n'a pas été supprimée par la suppression de l'état, c'est qu'elle est référencée par un ou plusieurs autres états correspondant à des opérations (C_i), toutes postérieures à A.

A chaque tile caché de toute opération B, se trouvant entre A et toutes les opérations C_i, correspond un tile caché dans A, qui est totalement inutile. En effet, tout rendu depuis les opérations C_i atteindront les tiles des opérations B avant d'atteindre les tiles de A. Ces tiles inutiles gaspillent de la mémoire et gagneraient à être supprimés.

Il existe également des tiles de A qui sont accessibles à partir d'une opération C, ceux-ci sont utiles, et gagneraient à être gardés.

Ceci est résumé par le schéma 3.5.2, page 41 qui représente en gris les tiles utiles et en rouge les tiles inutiles après la suppression d'un état.

FIG. 3.5: Tiles inutiles après la suppression d'un état

Il y a trois manières de gérer ce problème :

a. Garder la cache Dans ce cas, on consomme de la mémoire inutilement mais on évite de supprimer des tiles qui seront utiles par après. Les tiles se verront supprimés de la cache par un mécanisme global de suppression de tiles lorsque le framework manque de mémoire.

b. Supprimer totalement la cache Dans ce cas on libère toute la mémoire, mais on pourrait devoir recalculer certains tiles qui se trouvaient en cache. La complexité temporelle du rasterising revient donc à $O(n)$ ou n est le nombre d'opération dans la pile. Heureusement, on atteint rarement ce cas en pratique.

c. Supprimer uniquement les tiles inutiles Pour cela il faut tout d'abord trouver l'ensemble d'opération accessible depuis tous les C_i, ce qui n'est pas une chose évidente puisque les références entre opérations vont dans l'autre sens. Dans le pire des cas la complexité spatiale et temporelle est de $O(n)$ ou n est le nombre total d'opérations de toutes les piles correspondantes à cette image. La taille de l'ensemble d'opérations accessibles étant également en $O(n)$

Enfin, pour chaque tile caché dans chacune de ces opérations, il faut supprimer les tiles des opérations correspondantes. Cette tâche se fait en $O(n * m)$ ou n est la taille de l'ensemble d'opérations accessibles, et m le nombre maximal de tile caché dans une opération.

Cette complexité semble prohibitive, mais il y a des cas particuliers qui se rencontrent souvent en pratique qui limitent fortement cette complexité. Il reste donc à savoir si les tiles utiles qui sont sauvegardés permettront d'économiser plus de temps de calcul que celui qui est nécessaire pour réaliser cette opération.

Seules les solutions a et b ont été testées et implémentée. En raison de l'absence d'un dispositif global de suppression des tiles, c'est la solution b qui a été retenue.

Manque de mémoire

Enfin, il y a des moments ou l'on doit retirer des tiles de la cache car nous ne disposons plus de mémoire suffisante pour en stocker un nouveau. Aucun tile caché n'étant indispenable à la rasterisation du dessin, on peut supprimer ceux que l'on désire.

Il faut donc trouver le tile le moins utile. il s'agit là d'un sujet vaste qui n'a pas encore été exploré ni implémenté.

Pour l'instant les tiles ne sont jamais retirés de la cache par manque de mémoire. Cela n'a pas posé de problèmes lors des tests utilisateurs, mais ceux-ci ne sont pas représentatifs d'une utilisation prolongée du framework.

3.5.3 Migration de la cache

Il est également possible de migrer les tiles moins intéressant sur le disque dur, ou sur un espace réseau. Cette possibilité n'a pour l'instant pas été explorée ni implémentée.

3.6 Utilisation

Nous allons montrer dans cette section quelques examples qui illustrent l'utilisation de la structure et des algorithmes présentés dans ce chapitre afin de réaliser des fonctionalité utiles, ou d'émuler le fonctionnement d'autres frameworks.

3.6.1 Undo / Redo

Pour implémenter l'undo redo, on va utiliser le système d'état. Après chaque modification de l'image correspondant à une action, on sauvegarde un nouvel état. On va ensuite placer cet état dans la "pile undo". Lorsque la taille de cette "pile undo" dépasse le nombre maximal d'undo possible, on enlève le premier élément de celle-ci et on détruit l'état correspondant.

Pour annuler une action, on charge l'état correspondant à une action antérieure. Pour refaire une action, on charge l'état correspondant à celle-ci. Lorsque l'on désire ajouter une action alors que l'on se trouve au milieu de la "pile undo", il faut supprimer tous les états se trouvant au dessus dans la pile.

Un tel système est efficace ; les opérations de manipulations d'état sont suffisemment rapides pour que leur temps d'exécution soit imperceptible.

La rasterisation après l'annulation ou le rétablissement d'une action est normalement en $O(1)$, grâce aux caches, cependant, telle qu'implémentée, la gestion de la cache supprime des tiles utiles à la suppression d'états, ce qui peut occasionner quelques ralentissements.

3.6.2 Modèle par calques

FIG. 3.6: Modèle par calques à partir d'hlImages.

Il est possible de construire un modèle par calque semblable à ceux utilisés par les frameworks bitmaps en utilisant les opérations de fusion. À chaque

calque correspond une hlImage qui sera modifiée lors des peintures ou des applications de filtres. Une hlImage regroupe ensuite les calques en utilisant des opérations de fusion. Ceci est représenté sur le schéma 3.6.2, page 42.

3.6.3 Composition non linéaire

FIG. 3.7: Possibilités de composition non linéaire

La composition linéaire est possible grâce aux propriétés des opérations de fusion :

- Il est possible à plusieurs opérations de fusion de référencer la méme image, a des états quelconques.
- Il est possible à une opération de fusion de référencer sa propre image, tant que l'état référencé est antérieur à l'opération de fusion.

Il est ainsi très facile d'implémenter les *history brush* qui peignent un état antérieur sur l'image ou les *clone brush* qui peignent une copie de l'image translatée.

En combinant plusieurs images avec des opérations de fusion différentes, il est également possible de créer des réseaux fort complexes offrant des possibilités très proches de celles offertes par les frameworks nodaux.

Ces possibilités sont résumées au schéma 3.6.3, page 43.

Chapitre 4

Localité des operations

Ce chapitre traite de la gestion de la localité des opérations. Une opération est dite locale si elle n'affecte qu'une sous-région de l'image.

Il est rapidement apparu que la gestion de ces opérations par l'algorithme de rasterisation présenté à la section précédente était loin d'être optimale, et ne permettait pas d'atteindre les objectifs d'interactivité du logiciel.

Pour comprendre le problème, il faut d'abord comprendre comment fonctionne le logiciel de peinture.

4.1 Le logiciel de peinture

Le logiciel de peinture possède trois outils principaux : La peinture, l'undo/-redo, et une fenêtre de visualisation.

4.1.1 Background

Le logiciel présente à l'utilisateur une zone de dessin constituée d'un rectangle blanc qui représente la feuille sur laquelle dessiner. Pour ce faire, nous utilisons une Frame unie et transparente comme source de l'hlImage, à laquelle nous ajoutons une opération de dessin d'un rectangle blanc.

4.1.2 Peinture

La peinture se fait par la superposition de primitives appelées *brushes* positionnées à intervales réguliers sur le chemin tracé par l'utilisateur via la souris ou la tablette graphique. Les opérations de dessin de primitives sont regroupées en traits, qui correspondent à une pression continue sur le dessin.

La primitive utilisée pour la peinture est le cercle avec dégradé. Celui-ci est défini par deux rayons : Le rayon interne dans lequel le cercle est opaque, et le rayon externe, auquel se termine le cercle. Entre les deux un dégradé transparent. L'utilisateur peut aussi régler l'opacité et la couleur de la primitive.

Toutes les opérations sont ajoutées dans une seule hlImage, dont l'état est sauvegardé à chaque fin de trait.

Ce mécanisme est résumé au schéma 4.1, page 46.

4.1.3 Undo/Redo

L'undo redo fonctionne de la manière décrite au chapitre précédent. Une pile maintient une liste des états de chaque traits, permettant de défaire et refaire les opérations.

Comme indiqué au chapitre précédent, la consommation de mémoire est de l'ordre du nombre d'états. Comme il n'y a pas de mécanisme permettant de réguler les caches de manière globale, nous limitons le nombre d'états — et donc d'undos — disponibles à 16.

FIG. 4.1: Fonctionnement de la peinture

4.1.4 VISUALISATION

L'utilisateur visualise son dessin en temps réel via une fenêtre de visualisation. Celle-ci est décrite par sa taille en pixels, sa position sur le dessin, ainsi que son niveau d'échelle.

À chaque rafraichissement (entre 30 et 60 fois par secondes), les tiles correspondant à la région sont identifiés, rasterisés, et copiés dans le buffer correspondant à la région afin d'être visualisés.

Les rafraichissement se font indépendemment de l'ajout d'opérations. Un nombre quelconque d'opération peut donc avoir été ajouté entre deux rasterisations de l'image.

L'utilisateur peut modifier la région de visualisation de trois manières différentes :

- Déplacer la région.
- Redimensionner la région.
- Changer l'échelle de visualisation.

Alors que déplacer et redimensionner la région ne demande que de recalculer les quelques tiles de la nouvelle bordure, changer l'échelle demande de recalculer l'entièreté de la région de visualisation. On peut donc s'attendre à ce que le déplacement et le redimensionnement de la région de visualisation soient beaucoup plus rapides que le changement d'échelle.

4.2 Problématique

L'algorithme de rasterisation présenté jusqu'ici n'a aucune notion de localité d'opérations. Quelque soit le tile que nous voulons rasteriser, l'algorithme va appeler chacune des opérations. Si ces opérations ne s'appliquent pas au tile, celles-ci ne vont pas le modifier et vont donc s'exécuter très rapidement. Cependant, dans le cas de la peinture, chaque tile n'est en réalité affecté que par un très petit pourcentage des opérations, et le nombre d'appel à ces opérations inutiles va en pratique ralentir de manière substentielle la rasterisation.

Ce cas est particulièrement problématique lors du déplacement de la région de visualisation. Après avoir fait un large nombre de traits de peinture dans la région courante, l'utilisateur décide de déplacer la région de visualisation. De nouveaux tiles doivent donc être affichés. Pour ce faire l'algorithme de rasterisation va examiner toutes les opérations des traits que l'utilisateur vient de

faire, alors qu'aucun de ces traits n'affectent ces nouveaux tiles.

Il faut donc permettre à l'algorithme de rasterisation de pouvoir éviter d'examiner toute opération qui n'affecte pas les tiles. Au lieu d'avoir un algorithme de rasterisation en $O(n)$ ou n est le nombre d'opérations ajoutées depuis le dernier rendu, nous aimerions avoir pour n le nombre d'opérations ajoutées depuis le dernier rendu *qui affectent le tile*.

Il n'a pas été possible durant ce mémoire de trouver une réponse définitive à ce problème. Nous allons donc passer en revue les différentes options qui ont été considérées, ainsi que leur efficacité. L'analyse de leur efficacité nous permet d'avoir une bonne idée de la direction à prendre pour la poursuite des recherches d'une solution à ce problème.

Deux approches ont été testées : Les opérations vectorisées et les bounding operations.

4.3 Bounding Boxes

Une première étape est d'ajouter à toutes les opérations de dessin une *Bounding Box* — *BB* en abrégé. Celle-ci représente Une région rectangulaire, alignée aux axes de l'image, comprenant les tiles affectés par l'opération. Avant de rasteriser une opération sur un tile, la *BB* est examinée pour voir si le tile est affecté. s'il ne l'est pas, on peut éviter de faire appel à la fonction de l'opération qui aurait de toute manière implémenté un tel mécanisme de manière interne.

Un autre intéret d'un tel système est que la *BB* est générée à la création de l'opération, et non à chaque appel de l'opération, ce qui améliore encore les performances.

Pour réaliser cela, chaque classe d'opération doit donc être capable de générer de telles *BB* .

Si les *BB* améliorent nettement les performances, elles n'améliorent pas la complexité temporelle de la rasterisation. Elles constituent cependant une base solide pour les deux solutions suivantes.

4.4 Opérations vectorisées

FIG. 4.2: (a) Opérations normales, (b) Opérations vectorisées

Vectoriser les opérations consiste à regrouper des opérations du même type en une seule opération qui contient un vecteur de paramètres représentant l'ensemble des paramètres. La *BB* de l'opération vectorisée englobant les *BB* des opérations qui la composent. Ceci est représenté au schéma 4.2, page 47 Les opérations vectorielles ont plusieurs intérêts :

Dessin	Nombre moyen d'opérations par trait
A	105.4
B	199.8
C	117.8
D	149.1
E	117.2

TAB. 4.1: Nombre moyen d'opérations par trait sur les dessins de tests (voir figure 4.9, page 56)

- La structure hlOperation est mise en commun, permettant d'économiser la mémoire.
- Si un paramètre des opérations est constant dans le vecteur, celui-ci peut être mis en commun, ce qui économise de la mémoire. Ce cas de figure se présente régulièrement en peinture ; les traits étant principalement composés de *brushes* de même couleur et de même diamètre.
- Un test sur la Bounding box de l'opération vectorielle permet d'éviter le test de l'intégralité des opérations internes.

Mais ne sont pas sans limites :

- Il n'est pas possible de vectoriser des opérations de type différent. Ce système ne fonctionnerait donc pas pour un pinceau qui peindrait une alternance de cercles et de carrés.
- Il n'est pas possible de placer un état au milieu d'opérations vectorisées.
- L'algorithme de rasterisation doit être lourdement modifié pour pouvoir supporter ce type d'opérations.
- Les opérations vectorisées ne peuvent être modifiées avec l'API classique.

Les performances des opérations vectorisées seront donc contraintes par le nombre d'opérations du même type que l'on trouve entre chaque état. Pour l'application de peinture développée, les opérations sont toujours du même type, et un trait faisant en moyenne une centaine d'opérations (voir tableau 4.1, page 48), le nombre de tests de bounding box à réaliser en dehors des traits est divisé par 100. Le nombre de tests à réaliser sur les traits, augmente en contrepartie d'un pourcent à cause de la *BB* supplémentaire.

Nous n'atteignons donc pas notre objectif de réduction de complexité, mais les opérations vectorisées permettent de réduire de deux ordres de grandeur le coût de la localité des opérations.

4.4.1 IMPACT SUR L'API

Étant donné les limitations des opérations vectorisées, nous sommes obligés de donner à l'utilisateur un certain contrôle sur celles-ci. Nous proposons donc une fonction supplémentaire :

`void hlOpVectorise(hlOperation *op)`

pre : op est une opération qui n'est pas encore insérée dans la pile.

post : op est une opération vectorisée. Toute opération non vectorisée du même type qu'op insérée sur la pile au dessus de op sera vectorisée dans op, jusqu'à ce que op soit sauvegardée dans un état.

Ainsi, l'utilisateur n'a qu'à utiliser cette fonction sur la première opération de chaque trait pour pouvoir vectoriser les traits.

4.4.2 ÉVALUATION DES PERFORMANCES

À part un gain de mémoire au niveau de la description du document, qui reste négligable par rapport au poids pris par les caches, les performances démontrées par les opérations vectorisés sont quasiment identiques à un cas particulier des Bounding Operations, solution plus prometteuse présentée à la section suivante.

Nous nous réfèrerons donc à la section *Évaluation des performancesi / Tailles des* BO pour un apercu des performances des opérations vectorisées.

4.5 Bounding Operations

FIG. 4.3: Une Bounding Operation

Les *Bounding Operations – BO* en abrégé — ne sont pas des opérations au sens original du terme puisque si utilisées correctement, elles n'effectuent aucune modification de l'image. Cependant elles s'utilisent et s'insèrent dans la pile comme des opérations classiques appartenant à une nouvelle catégorie. Leur but est d'offrir une *BB* englobant plusieurs opérations, de la même manière que les opérations vectorisées, mais de manière plus flexible et facile à implémenter :

4.5.1 PRINCIPES D'UNE *BO*

Les *BO* contiennent une *BB* et une référence vers une opération antérieure. La rasterisation des *Bounding Opérations* fonctionne de la manière suivante :

- Rasteriser un tile à l'intérieur de la *BB* consiste à rasteriser l'opération précédente.
- Rasteriser un tile à l'extérieur de la *BB* consiste à rasteriser l'opération antérieure référencée.

En créant une *BO* dont la *BB* englobe les *BB* d'une suite d'opérations qui la précèdent, et en faisant pointer la *BO* sur une opération précédant cette suite, on permet à l'algorithme de rasterisation de contourner la suite d'opérations.

Ce principe est représenté au schéma 4.3, page 49

Les *BO* ne modifiant pas l'apparence de l'image, on peut les placer où l'on veut dans la pile. Ces opérations peuvent donc "enjamber" des états, permettant des groupements qui ne sont pas possibles avec les opérations vectorisées.

Là où les *BO* deviennent très intéressantes c'est qu'on peut les emboiter les unes dans les autres de manière récursive, et ainsi offrir une structure semblable à une *Bounding Volume Hierarchy* — *BVH* en abrégé — ou à des *KD-Trees*. On appelle *Bounding Operation Hierarchy* — *BOH* en abrégé — L'implémentation par des *BO* de ce type de structures. Lorsque les *BO* ne s'imbriquent pas, on parle alors de *Bounding Object Sequence*, — *BOS* en abrégé. Un exemple de *BOH* est représenté au schéma 4.4, pageref 50

Il y a cependant une grande différence entre les *BVH*, et les *BOH* ; là où un *Bounding Volume* peut contenir n'importe quel ensemble d'objets dans la scène, une *BO* ne peut contenir que des opérations qui sont adjascentes dans la pile. Il est très simple de construire un exemple ou les *BVH* excellent et les *BOH* ne font qu'empirer les choses, comme sur le schéma 4.5, page 50. Heureusement de tels

FIG. 4.4: Une Bounding Operation Hierarchy

FIG. 4.5: Cas critique pour les *BOH*

cas arrivent peu en pratique, puisque les outils de peinture tendent à grouper les opérations de manière adéquate.

Il serait cependant possible d'approcher l'efficacité d'un *BVH* en procédant à un réordonancement intelligent des opérations.

Une autre différence de taille est que les *BVH* sont conçus pour organiser des scènes statiques, et non pour des scènes pouvant changer selon les états. Ces différences nous empèchent de reprendre les algorithmes développés pour les *BVH* et les *KD-Trees*, même si nous pouvons en reprendre les idées et les principes.

Une première tâche intéressante consiste à construire la *BOH*. La littérature sur les *BVH* décrit deux types d'algorithmes :

Algorithmes Inline Ces algorithmes construisent la *BVH* au fur et à mesure que l'on ajoute des objets dans la scène — des opérations dans l'image. Ces algorithmes semblent donc particulièrement adaptés à la peinture puisqu'ils fonctionneront au fur et à mesure que l'utilisateur peint. Ces algorithmes

Algorithmes Offline Ces algorithmes construisent la *BVH* à partir d'une scène déjà construite — d'un pile d'opération existante. Ces algorithmes ont l'avantage de pouvoir créer des hiérarchies optimales (selon différents critères). Le problème est qu'ils prennent un certain temps à s'exécuter,

ce qui n'est pas idéal pour un framework dédié à l'édition interactive.

Seul des algorithmes inline ont pour l'instant été explorés et implémentés. Un algorithme crée des *BOS* , et l'autre des *BOH*.

4.5.2 Bounding Operation Sequence

Le premier algorithme testé et implémenté est un algorithme créant un *BOS*

Placement des BO *dans la pile*

Le placement des *BO* dans la pile fonctionne de la manière suivante : Initialement, la *BO* possède une *BB* de taille nulle, et est dans l'état dit *ouvert* Lorsque l'on insère cette *BO* au dessus de la pile, celle-ci prend pour comme opération de référence l'opération précédente.

Lorsque l'on insère une opération sur la pile, au dessus d'une *BO* ouverte, la *BB* de la *BO* est étendue pour englober la *BB* de la nouvelle opération. La *BB* est ensuite repositionnée au dessus de la pile.

L'on peut continuer à insérer des opérations de cette manière jusqu'à ce que la *BO* soit *fermée*, auquel cas elle se comporte comme une opération normale.

Fermer les Bounding Operations

Ce premier algorithme va donc ajouter des *BO* ouvertes, les fermer au moment opportun et puis directement en rajouter une nouvelle ouverte. Il s'agit donc de savoir quand fermer les *BO* .

L'impact des performances des *BO* peut se répartir en deux cas :

Rendu d'un tile à l'extérieur de la *BO* : Dans ce cas, on doit effectuer un test de *BB* de la *BO* uniquement. Comme on aimerait diminuer le nombre de tests de *BB* , on aimerait donc avoir un nombre minimal de *BO* .

Rendu d'un tile à l'intérieur de la *BB* : Dans ce cas, on doit tester les *BB* de toutes les opérations à l'intérieur de la *BO* , que celles ci influence le tile ou pas. On aimerait donc avoir le moins de Rendus à l'intérieur de *BO* . Statistiquement, plus la surface de la *BB* d'une *BO* est petite, moins de tiles seront rendus à l'intérieur. On aimerait donc avec des *BO* avec les *BB* les plus petites possibles.

La fermeture des *BO* doit donc satisfaire deux critères : avoir le moins de *BO* , et avoir les *BO* les plus petites. Ces deux critères étant opposés, nous allons examiner leur importance en examinant différents schémas de fermeture dans la section *analyse de performances*

4.5.3 Bounding Operation Hierarchy

Cet algorithme permet de créer une hiérarchie de *BO* d'une profondeur fixe. Dans cet algorithme chaque niveau fonctionne exactement comme l'algorithme précédent en considérant les *BO* des opérations d'en dessous comme des opérations classiques. Une première analyse d'efficacité se trouve à la section *Analyse des performances*

4.5.4 Impact sur l'API

Les *BO* ont un impact minimal sur l'API. Il suffit d'ajouter une opération de type *BO* avec les paramètres appropriés pour que les mécanismes se mettent en place. Une fois en place les *BO* n'influent quasiment pas sur les actions que l'on peut réaliser sur une hlImage.

Par contre, le fait que les *BO* puissent être déplacées automatiquement par d'autres opérations perturbe l'accès aux opérations par indice. Pour résoudre ce problème, les *BO* sont ignorées lors des accès par indice.

Si l'algorithme de rasterisation ne doit pas être grandement modifié, les algorithmes de modification de la pile doivent être revus pour préserver la cohérence des *BOH*.

Forking

FIG. 4.6: Forker des *BO*

L'opération de forking doit duplique également les *BO* . Le schéma 4.6, page 52 montre une pile contenant des *BO* avant et après forking.

La difficulté réside donc à réassocier les références correctement. Ceci se fait en reprenant l'algorithme de forking en applicant les modifications suivantes :

- Lorsque l'on duplique une *BO* , son duplicata continue à pointer vers l'opération originale. On associe ensuite dans une Hashtable l'uid de l'opération pointée par la *BO* à cette *BO* .
- Lorsque l'on duplique une opération (*BO* comprise), on regarde si son uid est présente dans la HashTable. Dans l'affirmative, la *BO* correspondante pointe désormais sur cette opération, et l'uid est retiré de la HashTable.

L'opération de forking garde donc toujours la même complexité temporelle, mais gagne une complexité spatiale de $O(n)$ où n est la profondeur maximale de la *BOH*, la profondeur des *BOH* étant en $O(log(n))$, où n est le nombre total d'opérations dans la pile.

Retrait d'opérations

FIG. 4.7: Retirer une opération pointée par une *BO*

Le retrait d'une opération dans la pile peut être problématique si un *BO* pointe sur cette opération. Dans de tel cas, il faut détecter les *BO* qui pointent et les faire pointer vers l'opération précédente à celle que l'on veut retirer. Si il n'y a pas de telle opération précédente, il faut remplacer l'opération à retirer par une opération qui ne fait rien. Il faut également supprimer les *BO* qui ne contiennent d'opération autre que des *BO* . Un exemple de retrait d'opération dans une pile contenant des *BO* est représenté au schéma 4.7, page 52

Ceci ne modifie pas la complexité temporelle, mais ajoute une complexité spatiale de $O(n)$ ou n est le nombre de *BO* pointant l'opération à retirer. Ce nombre est également borné par la profondeur de la *BOH*, ce qui ramène cette complexité à $O(log(n))$ ou n est le nombre d'opérations dans la pile jusqu'à l'opération à retirer.

Insertion et Modification d'opérations

FIG. 4.8: Modifier une opération dans une pile contenant des *BO*

Insérer ou modifier des opérations peut invalider les *BB* des *BO* qui englobent ces opérations. Il faut donc détecter ces *BO* , et modifier leur *BB* afin qu'elles contiennent la *BB* de l'opération à insérer, ou la nouvelle *BB* de l'opération modifiée, ce qui se fait en parcourant la pile jusqu'à l'emplacement d'insertion, ou jusqu'à l'opération à modifier de la manière suivante

- Lorsque l'on rencontre une *BO* , on l'associe dans une Hashtable à l'opération pointée.
- Lorsque l'on rencontre une opération, on retire la *BO* associée dans la Hashtable.
- Lorsque l'on arrive à l'emplacement voulu, ou à l'opération à modifier, la Hashtable contient la liste des *BO* à modifier.

Cela ne modifie donc pas la complexité temporelle, mais ajoute une complexité spatiale, en $O(log(n))$ où n est le nombre d'opérations dans la pile.

4.5.5 ÉVALUATION DES PERFORMANCES

Pour évaluer les performances, nous avons enregistré les étapes de création de plusieurs dessins, chacun d'entre eux representant un cas d'utilisation différent. Ces dessins de taille variable ont été réalisés sur une feuille de $4.8 * 10^17$ pixels. Nous avons ensuite rejoué ces étapes (ajout d'opérations, rasterisation, zoom, dézoom, undo, redo, déplacement du viewport) en ommettant les pauses afin de mesurer le temps total pris par le framework pour dessiner le dessin.

Toutes les expériences ont été réalisées sur une machine disposant d'un processeur *Intel Core Duo 2400* à deux cores cadancés à 1.66Ghz, et disposant de 2Go de mémoire vive.

Les cinq dessins sont présentés à la figure 4.9, page 56. Chacun d'entre eux a été réalisé pour simuler un cas d'utilisation différent :

A Ce dessin représente un dessin techinque, composé essentiellement de lignes droites d'orientation similaires, ainsi que d'arcs de cercles.

B Ce dessin représente des hachures, qui peuvent constituer l'essentiel d'un dessin au trait.

C Ce portrait est destiné à représenté un dessin complet qui comprend une variété de technique différentes : traits de contour, remplissages, hachures, ...

D Ce paysage de grande taille contenant de très petits éléments simule l'utilisation du framework pour l'édition d'images gigapixel à des échelles fort différentes.

E Ce dessin est constitué de différents éléments ayant posé problèmes lors des tests de performances, comme par exemple des brushes et des traits de taille très différentes.

Ces dessins sont en noir et blanc pour limiter le nombre d'opération nécessaires à leur réalisation. En effet les dessins ont été faits suffisemment petits pour limiter le temps de calcul lors de l'évaluation des performances. Ces dessins ne sont donc pas totalement représentatifs de ce qui se fait en conditions réelles, mais devraient être suffisemment proche pour nous permettre de savoir quelle solutions gagneraient à être explorées plus en détail sur des données en provenance de tests utilisateurs.

Taille des BO

Le premier test utilise une heuristique qui ferme la boite quand celle-ci contient un certain nombre d'opérations, ou quand un nouvel état est enregistré. À noter que cette heuristique correspond exactement au système d'opération vectorisées, qui produit des temps de calculs quasiment identiques.

Le tableau 4.5.5, page 57, reprend les résultats des expériences pour différentes tailles de boites. Le graphe 4.5.5, page **??** reprend ces données en normalisant les durées d'exécution de chaque test sur la durée la plus rapide.

On peut y voir que les performances augmentent avec la taille des *BO* pour ne jamais redescendre. On en déduit donc que la taille de la boite importe beaucoup moins que leur nombre. Si les performances ne diminuent pas plus au delà d'un certain nombre d'opérations par *BO* , c'est parce qu'à partir d'un certain point, les *BO* sont toutes fermées par l'enregistrement d'un nouvel état.

Aire accumulée

Cette heuristique basée sur l'aire accumulée essaie d'établir un compromis entre la taille des *BO* et le nombre d'opérations contenues dans celles-ci :

A chaque fois que l'on insère une opération dans une *BO* , on évalue l'augmentation de la surface de la *BB* de celle-ci. On normalise ensuite cette augmentation de surface par la taille de la *BB* de l'opération ajoutée. Cette augmentation de surface est accumulée dans un compteur dans la *BO* . Lorsque celle-ci dépasse un seuil, la *BO* est fermée.

Cette heuristique favorise donc des *BO* plus denses, ce qui l'on espère, permettra d'avoir une meilleure répartion des *BO* .

Les résultats des expériences sont repris au tableau 4.5.5, page 57. Celles-ci sont également reprises normalisées sur leur meilleur temps au graphe 4.5.5, page **??**.

On constate tout d'abord que cette heuristique ne permet pas d'atteindre les temps atteints par la précédente. On constate ensuite que l'évolution des améliorations est moins régulière et reflète les différences de structure des différentes images. Cette évolution irrégulière a pour conséquence qu'aucune valeur n'est optimale pour tous les tests, indiquant que la limite doit être modulée en fonction de l'utilisation du framework.

On constate également que cette heuristique fonctionne le moins bien pour les images *A* et *C*, qui contiennent beaucoup de diagonales. Le problème de cette heuristique avec ces diagonales c'est que celles-ci sont particulièrement mal épousées par les *BB* qui sont alignées aux axes. Dans ces cas l'aire est accumulée quadratiquement avec la longueur, et donc les boites sont cloturées trop tôt. Ceci génère un trop grand nombre de *BO* , et comme le résultat de l'expérience précédente nous indique, les performances sont fortement affectées par un nombre trop grand de celles-ci.

BOH

Nous utilisons pour cette troisième heuristique une *BOH* à deux niveaux. Les *BO* du premier niveau se cloturent au delà d'un nombre fixe d'opérations, de la mème manière que la première heuristique. Les *BO* du second niveau ne sont quand à elles cloturées qu'à l'ajout d'un nouvel état. Cette heuristique permet donc d'évaluer les gains que l'on peut obtenir en rajoutant un second niveau. Les résultats sont résumés dans le tableau 4.5.5, page 58, et repris sous forme normalisée par leur meilleur résultat au graphe **??**, page **??**.

On constate que les différences de structure des dessins crée une grande variation dans les performances en fonction de la taille des *BO* internes, ainsi que l'absence d'un optimum commun, prouvant la nécessité d'une heuristique plus complexe pour déterminer le moment optimum de fermeture des *BO* internes.

On constate également que les performances n'atteignent pas les systèmes à un seul niveau pour les Dessins *A*,*B*,*E*. Cela s'explique par le fait que ce dessins artificiels ne présentent pas des traits suffisemment longs et complexes pour qu'une sous-séparation de ceux-ci soit bénéfique.

Par contre, si l'on observe quelque gains pour les dessins *A* et *B*, ceux-ci sont peu importants. Une raison possible est le fait que pour les traits plus petits la *BO* interne est redondante.

4.5.6 Conclusion

Le analyses de performances montrent que les *BO* améliorent de manière substantielle les performances du framework pour la peinture. Cependant des améliorations restent possibles, comme une heuristique de cloture plus intelligente, qui permettrait des niveaux de *BO* supplémentaires qui englobent les états.

Enfin les tests se sont concentrés sur le temps de rendu moyen d'une opération, il pourrait être intéressant de tenter de réduire la variance des temps de rendu afin d'augmenter la fluidité lors d'édition interactive.

(*a*) Dessin technique, 25Mpixels

(*b*) Hachures, 0.5Gpixels

(*c*) Portrait, 25Mpixels

(*e*) Test divers, 0.5Gpixels

(*d*) Paysage avec petits détails, 2Gpixels

FIG. 4.9: Dessins utilisés pour les tests de performances

	Nombre maximal d'opérations par *BO*									
	8	16	32	64	128	256	512	1024	2048	4092
Dessin	Temps de calcul (sec)									
A	249.67	117.81	72.01	52.00	39.13	34.32	32.87	32.84	32.99	32.83
B	847.67	312.65	113.12	70.14	52.35	40.44	39.45	41.32	42.51	42.78
C	1283.64	550.17	354.88	154.36	138.46	163.26	104.11	94.40	108.18	146.60
D	1181.11	466.45	278.11	144.25	170.93	132.44	91.03	89.84	88.35	140.39
E	61.32	38.71	18.21	12.93	11.81	11.19	11.01	11.08	10.72	11.15

	Aire maximale accumulée par *BO*							
	2	4	8	16	32	64	128	512
Dessin	Temps de calcul (sec)							
A	70.85	51.89	41.54	37.51	35.72	34.89	36.23	54.46
B	225.32	139.10	100.74	83.29	45.69	42.70	41.45	43.58
C	585.40	321.23	141.87	163.36	156.43	145.06	143.24	141.66
D	656.04	358.37	158.52	153.24	159.39	149.18	148.01	98.38
E	16.21	12.59	11.73	12.30	11.44	10.62	11.13	10.66

	Nombre maximal d'opérations par *BO*									
	8	16	32	64	128	256	512	1024	2048	4092
Dessin	Temps de calcul (sec)									
A	56.33	35.43	33.72	33.41	33.57	33.32	32.82	34.31	34.76	34.26
B	53.30	70.24	67.47	66.61	64.70	41.40	42.01	43.41	44.54	44.36
C	107.27	144.32	141.42	94.17	91.31	123.96	142.79	141.63	142.71	148.53
D	144.20	88.20	86.48	115.26	140.11	147.81	141.73	101.48	88.51	108.78
E	17.88	17.06	16.89	16.67	16.32	16.59	16.51	16.59	16.88	16.28

Chapitre 5

Anti-aliasing (15 pages)

5.1 Primitive de dessin

5.2 Problèmes d'échelle

5.2.1 Oversampling

5.3 Problèmes de superposition

5.4 Problèmes de bandes

5.5 Problème de blending à faible opacité

5.6 Problème de précision de positionement

Chapitre 6

Test utilisateurs (10 pages)

6.1 Procédure

6.2 Résultats

6.3 Analyse

Chapitre 7

Comparaison d'Himalaya aux autres frameworks

Chapitre 8

Conclusion

Table des matières

1 Introduction **7**

2 Présentation des frameworks d'édition d'images **9**
2.1 Architecture d'un framework d'édition d'image 9
2.2 Fonctionalités d'un framework d'édition d'image 10
2.3 Frameworks Bitmaps 13
2.4 Frameworks Nodaux 16
2.5 Framework vectoriel 18
2.6 Mégatexturing ou Sparse Virtual Textures 21
2.7 Comparaison des différents framework 22

3 Présentation du Framework Himalaya **25**
3.1 Structures de données 25
3.2 Algorithmes 33
3.3 États 35
3.4 Opération de Fusion 39
3.5 Gestion de la cache 39
3.6 Utilisation 42

4 Localité des operations **45**
4.1 Le logiciel de peinture 45
4.2 Problématique 46
4.3 Bounding Boxes 47
4.4 Opérations vectorisées 47
4.5 Bounding Operations 49

5 Anti-aliasing (15 pages) **59**
5.1 Primitive de dessin 59
5.2 Problèmes d'échelle 59
5.3 Problèmes de superposition 59
5.4 Problèmes de bandes 59
5.5 Problème de blending à faible opacité 59
5.6 Problème de précision de positionement 59

6 Test utilisateurs (10 pages) **61**
6.1 Procédure 61
6.2 Résultats 61
6.3 Analyse 61

7 Comparaison d'Himalaya aux autres frameworks **63**

8 Conclusion **65**

Table des matières **66**

Index **69**

Index

F

Filtres 3
colorimetriques 4
spatiaux 4
Framework d'edition d'images . . 3

P

Primitives, dessin de 3