Université de Paris 8

Master Création numérique

Parcours : Arts et Technologies de l'Image Virtuelle

AUTOMATISATION DE LA PEINTURE ANIMEE

Création de nouveaux processus pour les applications temps réel

Alexandre Gomez

Mémoire de Master 2, 2016 - 2017

Remerciements

à Isadora Teles de Castro e Costa pour notre collaboration et son soutien,

à Monique et Daniel Gomez pour leur relecture,

à Simon Gomez pour ses conseils et suggestions,

à Chu-Yin Chen, Jean-François Jego, Cédric Plessiet et Alain Lioret
pour leurs conseils constructifs lors du suivi de mémoire

Résumé

Ce mémoire présente mes propositions de méthodes d'automatisation du rendu non photoréaliste évoquant la peinture animée. Je m'intéresse à leur application possible dans des travaux temps réel. En effet, la recherche dans ce domaine ne s'attache que rarement au temps réel, à ses problématiques et à ses spécificités.

Les techniques de programmation graphique que je propose sont simples à implémenter et ne demandent que très peu d'interaction de la part de l'artiste souhaitant les mettre en œuvre pour produire un résultat original. Ces techniques permettront à des personnes qui travaillent dans de petites équipes de réaliser des projets d'installations artistiques, des jeux vidéo ou des films dans des styles de rendu expressif avec un temps de réalisation acceptable.

Pour s'adapter à différents types de projet, je propose des expérimentations qui peuvent s'appliquer à des images en deux aussi bien qu'en trois dimensions.

Enfin, sont présentés trois projets qui m'ont permis de mettre en pratique et d'approfondir les trois méthodes les plus concluantes de ma recherche.

Abstract

This dissertation presents my proposals for methods of automating non-photorealistic rendering evoking animated painting. I am interested in their possible application in real-time projects. Indeed, research in this field is rarely concerned with real time, its problems and its specificities.

The graphic programming techniques that I propose are simple to implement and require very little interaction from the artist wishing to implement them to produce an original result. These techniques will allow people working in small teams to carry out artistic installation projects, video games or movies in expressive rendering styles with an acceptable completion time.

To adapt to different types of projects, I propose experiments that can be applied to images in two as well as in three dimensions.

Finally, three projects are presented which have allowed me to put into practice and to deepen the three most conclusive methods of my research.

Vocabulaire technique

Ce mémoire comprend de nombreux termes techniques liés au domaine de la programmation graphique. La documentation et les ressources sur le sujet sont pour la plupart rédigées en anglais. Le vocabulaire technique anglais employé dans ce domaine est également usité en français. Pour plus de clarté et de cohérence avec cette documentation et avec les usages, ces termes sont en anglais dans le présent document, écrits en italique et traduits entre parenthèses seulement s'il existe un équivalent pertinent en français.

Table des matières

Introduction

L'ordinateur est un outil de calculs qui, en imagerie numérique, est principalement utilisé pour simuler la réalité le plus fidèlement possible. Les chercheurs ont développé des méthodes qui permettent de créer des images numériques précises et de rendre des objets numériques de manière très réaliste. Ce sont ces logiciels, réalisés dans le but de simuler des images où les formes sont précisément délimitées et où l'exactitude est recherchée, qui sont disponibles pour la création artistique. Et il est difficile aujourd'hui pour les artistes numériques de s'exonérer de ces méthodes et techniques existantes, ce qui conduit à un formatage visuel des images issues de la technologie de création numérique.

Il existe peu de recherches dans le domaine des images numériques non photoréalistes. Celles-ci ont commencé au début des années 90 dans l'objectif de générer des images expressives sur un ordinateur en s'inspirant des méthodes traditionnelles de la peinture (Haeberli, 1990). Bien que la recherche ait progressé dans ce domaine, très peu de chercheurs s'intéressent à leur utilisation dans le domaine de l'infographie temps réel. Les principaux projets utilisant ce type de procédé sont les jeux vidéo dont le style graphique est généralement obtenu par simplification des formes et des couleurs pour évoquer la bande dessinée mais très rarement par des effets de peinture.

L'idée à l'origine de mon projet de recherche est de créer une nouvelle esthétique visuelle sur le support numérique, en écartant l'option consistant à reproduire un style pré-existant. Il est possible de reproduire numériquement certaines techniques traditionnelles, mais le procédé est relativement long à mettre en place, gourmand en ressources et n'est pas forcément adapté au format numérique. Pour gagner en temps de production, est-il possible de créer un outil d'automatisation de la peinture animée qui nécessiterait un minimum d'intervention de l'artiste ?

Actuellement, selon les productions, les techniques varient considérablement. Existe-t-il une méthode qui fonctionnerait dans la plupart des projets et qui serait calculable assez rapidement pour être mise en place dans une application temps réel ?

Pour chercher une réponse à ces problématiques, j'ai dans un premier temps cherché à inscrire ma démarche dans celle de la technique picturale et de son histoire, pour mieux comprendre le rapport de la peinture numérique avec la peinture traditionnelle. Ensuite, ma réflexion s'est orientée sur ce que l'outil numérique est susceptible d'apporter à la peinture en m'appuyant sur les travaux de recherche liés à la génération d'images dans un style non photoréaliste avec des effets évoquant la peinture à l'huile, pour avoir un aperçu des techniques existantes.

La seconde partie du présent document présente les différentes expérimentations réalisées. Je commencerai par décrire le pipeline de création d'une image temps réel et les méthodes utilisées en programmation graphique pour intervenir sur le rendu d'une image. Ensuite je mettrai en pratique et proposerai des solutions fondées sur des expérimentations en deux di-

mensions puis, dans un second temps, sur des méthodes en trois dimensions. L'objectif poursuivi est de rechercher une solution d'automatisation de la peinture animée propice à l'émergence d'un nouveau style de rendu pictural avec la contrainte fixée au départ que cette solution soit utilisable dans la plupart des projets.

Enfin sont présentés trois projets prolongeant les résultats les plus concluants de mes recherches.

1. L'émergence de la peinture animée numérique : la rencontre d'une longue tradition picturale et d'une technologie

1.1 Aperçu de l'histoire de la peinture et de la technique picturale

1.1.1 La peinture des premiers hommes

La première trace de pratique de la peinture par l'homme remonte au paléolithique supérieur (il y a environ 32 000 ans) avec l'art pariétal. Ces peintures étaient réalisées sur les parois des grottes au moyen d'un mélange de pigments naturels tels que de l'ocre ou du charbon de bois. Les outils utilisés étaient variés : peinture avec les doigts, au pinceau ou avec la technique du soufflé qui consiste à expulser de la peinture par la bouche. L'utilisation de techniques rudimentaires permettaient déjà de réaliser des dégradés ou des remplissages de forme par de la couleur en utilisant des pochoirs : la peinture était appliquée avec les mains ou au moyen de peaux. Les principaux sujets représentés étaient des animaux sauvages mais les préhistoriens ont également découvert des peintures abstraites ou encore des silhouettes de main.

Illustration n°1 : Deuxième Cheval chinois, Grotte de Lascaux (paléolithique supérieur), crinière et robe obtenu par projection de peinture et contour tracé au pinceau

Mais la peinture n'était pas la seule technique d'expression artistique utilisée à cette époque préhistorique : étaient également présents le dessin, la gravure et la sculpture, cette dernière a priori plus rarement. Mais quelle que fut la méthode utilisée, l'intention principale était de produire des images sur des surfaces planes ou en relief. Avec la peinture, les hommes ne se contentaient pas de donner de la couleur à un objet. Bien au-delà d'un simple procédé de colorisation, il s'agissait d'un processus de création d'une image consistant à travailler la forme par la couleur.

1.1.2 La peinture : des matières et des techniques spécifiques

Il est parfois difficile d'établir une distinction entre les différents médiums. Ainsi, les deux techniques de représentation en deux dimensions que sont la peinture et le dessin peuvent, de prime abord, sembler similaires. Il me paraît dès lors nécessaire, dans le but de clarifier la suite de mes propos, d'expliciter ces termes : je définirais le dessin comme étant une technique du trait et de la ligne, de la délimitation précise de formes obtenue en utilisant un outil à pointe dure alors que la peinture consiste à travailler une matière colorée, liquide lors de son application et qui se solidifie et se fixe avec le temps. La peinture s'étale quand le dessin se trace.

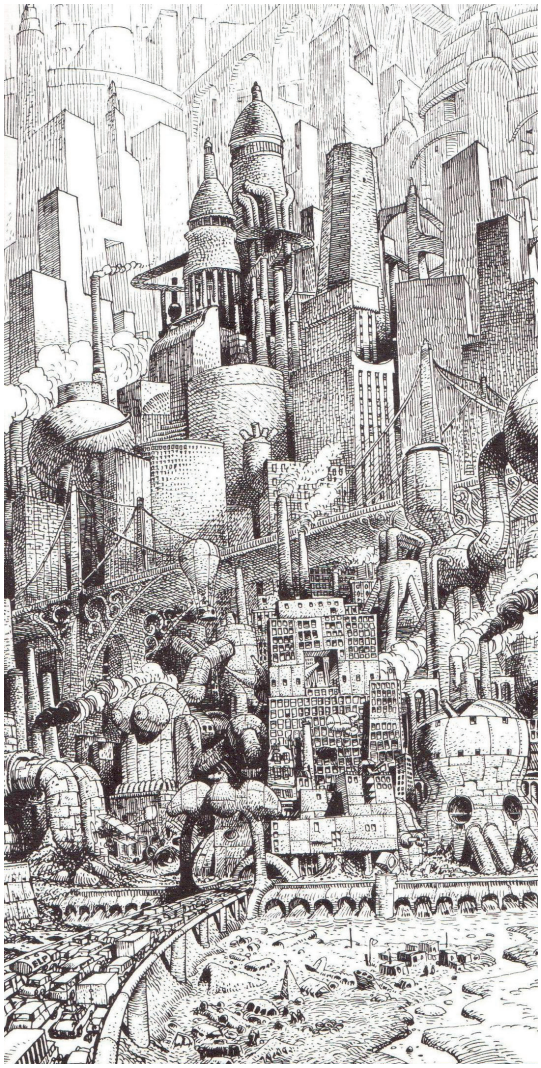


Illustration n°2 : Jean Giraud, *La Déviation* (1973), Arzach, Les Humanoïdes associé, 2000, dessin au trait

Illustration n°3 : Jean Giraud, *Couverture originale de Blueberry: Le cheval de fer* (tome 7) (1970), Gouache sur papier

A chaque technique sont associés des outils. En peinture, les différents outils vont peu évoluer jusqu'au 20ème siècle, époque à laquelle se produit un renouvellement des techniques. En effet, aux côtés des œuvres traditionnellement peintes au pinceau ou au couteau apparaissent des œuvres réalisées avec de nouveaux procédés tels que les boîtes de conserves trouées de Jackson Pollock ou les tirs de carabine de Niki de Saint Phalle.

Illustration n°4 : Niki de Saint Phalle, *Grand Tir - Scéance galerie J* (1961), Peinture, plâtre et objets divers sur panneau d'aggloméré, Collection particulière ; courtoisie galerie G.-P. & N. Vallois, Paris

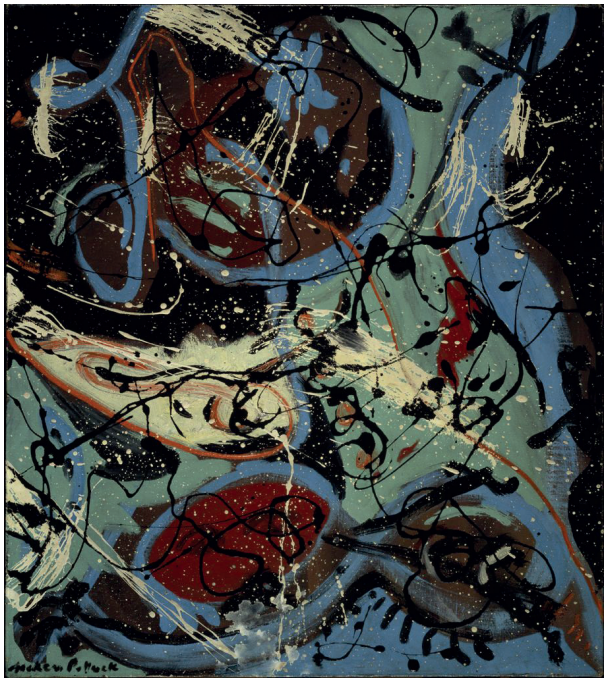


Illustration n°5 : Jackson Pollock, *Compositing With Pouring II* (1943), Hirshhorn Museum and Sculpture Garden, Smithsonian Institution

Au fil du temps et selon les pays, les matières utilisées pour la peinture, terme générique pour décrire cette substance qui sert à construire l'image, ont varié : de la tempera à la peinture à l'huile en passant par l'encre de Chine. Les caractéristiques propres à ces différentes matières ont imposé des techniques d'application très diverses. Ce qui n'a pas empêché, au besoin, de les associer : la gouache peut ainsi être utilisée pour rehausser des aquarelles.

Le choix du support de la peinture par l'artiste est également fondamental. Il varie selon les époques : mur, planche de bois, toile tendue ou papier. Le type de support est souvent associé à une technique spécifique.

1.1.3 La peinture numérique : nouvel outil ? nouveau medium de création ?

A partir de la deuxième moitié du 20ème siècle, la peinture numérique va émerger et bouleverser le processus de conception d'une image peinte. Sur un ordinateur, le support de travail est l'écran. Ce n'est rien d'autre qu'un tableau de données en deux dimensions défini par x lignes et y colonnes, c'est à dire une matrice de pixels. Ceux-ci sont les plus petits éléments composant une image numérique et leur couleur est définie par synthèse additive : il s'agit de mélanger les couleurs primaires rouge, bleu et vert pour former un gamut de couleurs. C'est le processus inverse de la peinture traditionnelle qui fonctionne par synthèse soustractive avec les couleurs primaires que sont le magenta, le cyan et le jaune.

Pour générer des formes, il faut communiquer avec l'ordinateur dans un langage

informatique et élaborer des programmes dans ce langage pour traduire et faire exécuter les instructions permettant d'obtenir le résultat attendu. Les artistes peuvent écrire leurs propres programmes pour générer de manière automatique ou semi-automatique des images dites numériques. Mais il est également possible de recourir à des programmes dit de haut niveau qui permettent à des utilisateurs ne maîtrisant pas la programmation d'utiliser ces outils numériques : il suffit de savoir manier des dispositifs interactifs tels que la souris, des stylets ou des surfaces tactiles à manipuler avec les doigts.

Si l'écran de l'ordinateur est le support du travail du créateur d'images numériques, ces dernières ne sont pas enfermées dans ce format. Ainsi, Harold Cohen a développé dans les années 1970 un programme qu'il fera évoluer au fil des années. Il a commencé en créant des figures abstraites, puis des formes représentatives où les images générées étaient ensuite peintes à l'aide d'un bras robotisé sur une toile réelle.

Illustration n°6 : Harold Cohen avec *Aaron* au Computer Museum à Boston en 1995

L'ordinateur est un outil de simulation virtuelle, c'est à dire qu'il va permettre à la peinture de se libérer des contraintes imposées par les lois de la physique qui régissent le monde réel. « Peindre » avec cet outil est plus simple et ouvre de nouvelles perspectives du fait de la suppression de certaines contraintes matérielles.

Le peintre numérique accède plus facilement à une grande palette de couleurs et à une grande quantité d'outils, il n'est plus soumis à la contrainte du temps de séchage de la peinture.

La peinture virtuelle va être utilisée pour peindre sur des canevas virtuels. Cette nouvelle approche présente encore l'avantage de permettre à l'utilisateur d'avoir un plus grand contrôle sur l'image par la manipulation précise de chaque pixel affiché à l'écran.

Mais la principale différence avec la peinture traditionnelle se situe dans la façon de procéder pour créer une image. Ce processus, appelé *workflow*, décrit l'ordre des étapes de

réalisation d'une création, qu'elle soit artistique ou autre. L'artiste numérique peut travailler de manière non linéaire et éditer, avec des systèmes de calques, différentes couches de peinture. Il peut revenir sur ses choix antérieurs en navigant dans un historique des actions précédemment réalisées, modifier les couleurs de la composition à n'importe quel moment, effacer des éléments de l'image ou mélanger les couleurs par différents modes de fusion.

Ces observations me conduisent à définir la peinture numérique comme un moyen d'étendre le domaine de la peinture traditionnelle sans bien entendu s'y substituer.

Par convention et pour en terminer avec cette définition de la peinture et son rapport au numérique - qui ne sont que des sujets périphériques à mon objet d'étude, je parlerai par la suite de technique picturale plutôt que de peinture, la technique picturale représentant les méthodes et moyens de produire une image qui évoque la peinture.

Enfin, il est intéressant de noter qu'avec la démocratisation récente des outils et techniques liés à la réalité virtuelle émerge une toute nouvelle approche de la peinture avec des outils qui permettent de peindre en trois dimensions. On se retrouve, avec ces techniques, dans un champ de création situé quelque part entre la sculpture et la peinture.

Illustration n°7 : Oculus Story Studio, *Dear Angelica* (2017), film en réalité virtuelle réalisé à l'aide de l'outil Quill

1.1.4 Le style pictural ou l'expression d'un regard sur le monde

L'analyse de la peinture ne se limite pas à s'interroger sur les matières et les outils. La peinture est un art de la représentation. On observe des thématiques récurrentes dans l'Histoire tels que le portrait, le paysage, la nature morte ou les sujets religieux et cela quelle que soit l'époque. Mais les historiens de l'art classent plutôt les œuvres peintes dans ce qu'ils nomment courants ou mouvements, ce qui permet de regrouper des artistes par époque, suivant une certaine vision et une certaine approche de la technique picturale. La notion de style n'est pas directement en rapport avec l'évolution de la technique picturale mais plutôt avec une façon de voir le monde. Celle-ci évolue en fonction de l'état d'esprit d'une époque, des découvertes scientifiques et techniques ou par rapport à un contexte socio-culturel particulier.

L'image picturale naît de l'agencement de couleurs et de formes pour créer une composition. Chaque peintre aura sa propre façon de travailler et donc le résultat sera différent pour chaque individu réalisant une peinture. Mais qu'il est difficile de définir ce qu'est une « bonne » peinture. Dans *Le journal d'un génie*, Salvador Dali classe à sa manière, de façon parodique, différents peintres de l'Histoire en leur donnant des notes dans les catégories technique, inspiration, couleur, sujet, génie, composition, originalité, mystère, authenticité. Comme nous le verrons dans la partie 1.2.1.2, il n'existe pas, pour la peinture, de méthodologie permettant d'évaluer la qualité d'une œuvre, c'est un exercice qui reste très personnel.

	Technique	*Inspiration*	*Couleur*	*Sujet*	*Génie*	*Composition*	*Originalité*	*Mystère*	*Authenticité*
Léonard de Vinci	17	18	15	19	20	18	19	20	20
Meissonier	5	0	1	3	0	1	2	17	18
Ingres	15	12	11	15	0	6	6	10	20
Velasquez	20	19	20	19	20	20	20	15	20
Bouguereau	11	1	1	1	0	0	0	0	15
Dali	12	17	10	17	19	18	17	19	19
Picasso	9	19	9	18	20	16	7	2	7
Raphael	19	19	18	20	20	20	20	20	20
Manet	3	1	6	4	0	4	5	0	14
Vermeer de Delft	20	20	20	20	20	20	19	20	20
Mondrian	0	0	0	0	0	1	1/2	0	3,5

Illustration n°8 : Salvador Dali, Journal d'un génie (1964)

1.1.5 Quelques exemples d'utilisation de la peinture animée dans des œuvres cinématographiques

Pour finir cette rapide introduction à la technique picturale, j'aimerais également évoquer la peinture animée et l'utilisation de la peinture au cinéma. Je vise ici les œuvres dont on peut considérer que la peinture est le principal sujet ou les œuvres qui utilisent la peinture comme technique pour autre chose que la réalisation de *matte paintings* (peinture pour décors de films).

Tout d'abord, plusieurs artistes utilisent la peinture dans leurs travaux d'animation. Ainsi, Aleksandr Petrov a réalisé des films, comme « *The Old Man and the Sea* » en 1999, qui montrent la manière dont la technique de la peinture animée peut être utilisée dans des productions cinématographiques. Il est observé que le nombre d'images par seconde est généralement plus bas que pour la plupart des techniques d'animations. Ceci est dû au temps de réalisation de chaque image, bien plus long que dans la plupart des méthodes de dessins traditionnels.

Illustration n°9 : Aleksandr Petrov, *The Old Man and The Sea* (1999)

Les œuvres cinématographiques comportent également de nombreuses références à la peinture ou à des représentations picturales. Dans le film « *Au-delà de nos rêves* » (*What Dreams May Come*) de Vincent Ward (1998), le personnage principal se retrouve dans une peinture réalisée par sa femme. Dans le film d'Akira Kurosawa « *Dreams* » un personnage se promène dans une peinture de Vincent Van Gogh.

Illustration n°10 : Vincent Ward, *What Dreams May Come* (1998)

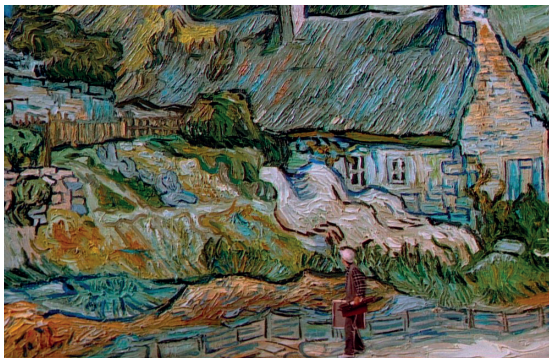


Illustration n°11 : Akira Kurosawa, *Dreams* (1990)

1.2 Les atouts des outils numériques

1.2.1 Des outils technologiques au service de la création artistique

1.2.1.1 Le langage informatique et le modèle de pensée logico-mathématique

Les nouvelles technologies du numérique changent-elles les procédés de production d'une image ? Edmond Couchot explique que l'art numérique « *correspond à un changement d'état profond de la technique : le passage d'une activité empirique à une activité réglée par un raisonnement formalisé.* » (Couchot & Hillaire, 2009). Ce qu'il sous-entend par là est que tout programme informatique est fondé sur un modèle logico-mathématique. L'ordinateur est donc un outil de simulation qui va transformer, à partir d'un raisonnement rationnel, un concept ou une idée en algorithme mathématique.

Techniquement, tout pourrait être transformé en information numérique en communiquant avec l'ordinateur par des langages de programmation. Le monde virtuel ne connaît pas de limite apparente mais la principale difficulté réside dans la formulation des idées à l'aide d'un langage. Il est facile de simuler le comportement d'une vague d'eau en s'inspirant des modèles physiques de dynamique des fluides, il est plus difficile de modéliser et de transcrire des modèles inintelligibles, c'est-à-dire qui ne sont pas issus d'une réflexion rationnelle. Par quel modèle mathématique peut-on traduire l'amour ou la créativité ? Comment communiquer cette information à une machine ?

De plus, les connaissances nécessaires pour parvenir à traduire des concepts en langage informatique ne sont pas à la portée de tous. Avec le développement exponentiel des connaissances scientifiques et des outils technologiques, il est de plus en plus difficile d'acquérir tous les savoirs disponibles dans un domaine spécifique. C'est pourquoi, en matière de production informatique, il est nécessaire de recourir à des interfaces ou à des bibliothèques programmées par d'autres chercheurs pour réaliser certaines opérations. Des spécialistes proposent ainsi des interfaces de haut niveau pour générer des images.

Malgré tout, il paraît absolument nécessaire qu'un artiste maîtrise la programmation, le langage qui va lui permettre de communiquer avec l'ordinateur. La possibilité de communiquer des concepts logico-mathématiques à la machine apporte une grande liberté et permet de s'affranchir des contraintes générées par les outils créés par d'autres qui peuvent conduire à un certain formatage. En outre, comme le dit Edmond Couchot, « *la connaissance de la programmation permet d'échapper au déterminisme de la machine et de ne pas être manipulé par ses outils mais de les manipuler* » (Couchot & Hillaire, 2009).

1.2.1.2 Les deux grandes techniques de rendu

Les images générées par ordinateur peuvent être bi-dimensionnelles ou tri-dimensionnelles et sont virtuelles, au moins au moment de leur création. En imagerie numérique, le « rendu » est l'appellation du processus mis en œuvre pour calculer une image à partir d'informations d'entrée. Deux grandes catégories coexistent : le rendu photoréaliste et le rendu non photoréaliste aussi appelé en anglais NPR pour *non photorealistic rendering*.

Dans la première catégorie, le modèle physique de la lumière est utilisé pour calculer les images créées. Dans la seconde, les techniques ne sont pas fondées sur les propriétés réelles d'illumination ; elles simulent des styles graphiques tels que la peinture, le dessin ou les illustrations techniques. A titre d'exemple, le court métrage *Ryan* contient certains personnages qui sont représentés de manière stylisée.

On peut également citer les films « *Waking Life* » et « *A Scanner Darkly* » de Richard Linklater qui, en utilisant une technique de rotoscopie appelé *Rotoshop*, transforme des séquences filmées en un style rappelant la bande dessinée.

Illustration n°12 : Richard Linklater, *Waking Life* (2003)

Illustration n°13 : Richard Linklater, *A Scanner Darkly* (2006)

Je citerais également certains films du studio Disney comme le court métrage « *Paperman* » qui donne l'illusion que des modèles en trois dimensions (3D) sont réalisés dans un style de dessin fait à la main.

Illustration n°14 : John Kahrs, *Paperman* (2012), Walt Disney Animation Studio

Beaucoup de films Disney intègrent des éléments en 3D qui se mélangent avec les animations réalisées à la main et qui sont donc rendues dans un style non photoréaliste. Ainsi, dans une scène dans « *Tarzan* », le personnage éponyme glisse le long d'un tronc d'arbre en 3D ou encore, dans le film « *Aladdin* », pour la scène de l'éboulement de la caverne, la lave, la plupart des parois de la grotte et le tapis volant sont réalisés en 3D.

Aujourd'hui les techniques de rendu photoréaliste sont globalement maîtrisées et, si les moyens financiers le permettent, il est possible de reproduire virtuellement tout ce qui existe dans le monde réel parce que nous pouvons simuler le fonctionnement de la physique de la lumière. Aujourd'hui, les recherches dans ce domaine s'orientent vers la découverte d'algorithmes plus rapides et plus performants : il s'agit de diminuer les temps de calcul plutôt que d'augmenter le réalisme de la simulation.

A l'opposé, il est difficile d'établir une équation pour le rendu non photoréaliste. Dans son texte « *Non-Photorealistic Rendering and the Science of Art* » (Hertzmann, 2010), Aaron Hertzmann évoque plusieurs problématiques liées à la recherche dans ce domaine. Il n'existe pas de méthodologie de travail précise parce qu'il est difficile d'analyser le processus de rationalisation de la création picturale. Il faut simuler non seulement une technique mais aussi un concept abstrait. Les recherches dans ce domaine conduisent donc à se poser des questions sur la manière dont le cerveau fonctionne pour concevoir des images puis à définir les critères qui permettraient d'évaluer les résultats obtenus. Contrairement au processus d'évaluation mis en œuvre lors de recherches rationnelles menées dans des domaines scientifiques, il est ici difficile d'évaluer le résultat de ces recherches sur des critères objectifs. Il en résulte que la recherche dans le rendu non photoréaliste consiste à tendre vers une meilleure compréhension de l'art et des processus cognitifs mis en œuvre dans la démarche de création.

1.2.1.3 La création d'outils et l'automatisation

L'artiste numérique a aujourd'hui a sa disposition des outils spécifiquement conçus pour générer des images numériques ; il peut également faire appel à des processus semi-automatiques et à des processus automatiques.

Un outil informatique utilisable par un artiste peut être, par exemple, un logiciel de

dessin avec lequel l'utilisateur va réaliser la majeure partie du travail à la main : les images du jeu *SuperBoy*, prototype de jeu vidéo inspiré du jeu « *Super Meat Boy* » et réalisé durant mon année de licence 3, ont toutes été dessinées ainsi.

Illustration n°15 : Alexandre Gomez, *SuperBoy* (2015)

Les techniques dites de rendu automatique vont transformer une information de base (scène 3D, image,...) en une nouvelle image. Ces méthodes peuvent être assistées et guidées par l'utilisateur ou être complètement automatisées.

Dans le cadre de mes recherches, ce dernier procédé retiendra principalement mon attention pour deux raisons : il peut faire gagner du temps en production comparativement au procédé qui requiert un travail manuel et des nouveautés graphiques peuvent émerger de l'algorithmie.

L'idée qui sous-tend mon sujet n'est pas de remplacer ni de simuler de manière réaliste la peinture mais de m'inspirer de la peinture pour trouver de nouveaux processus automatiques de technique picturale dans l'objectif de créer de nouvelles formes d'expression et de représentation.

1.2.2 Le temps réel

1.2.2.1 Le temps de calcul d'une image

Une autre particularité de l'art numérique est la notion de temps réel.

La génération d'une image numérique implique un temps de calcul. Pour qu'un programme puisse être interactif, du moins en imagerie 3D, il faut qu'au moins six images soient calculées par seconde (FPS en anglais pour *frames per second*) (Akenine-Möller et al., 2008).

Pour les jeux vidéo ou les installations interactives, les développeurs s'accordent généralement sur un minimum de 30 ou 60 FPS pour obtenir une image animée fluide et réactive. L'interaction doit en effet être possible immédiatement. Par comparaison, un film est généralement tourné en 24 FPS, ce qui dans ce domaine, est suffisant grâce à l'effet de flou optique généré par le mouvement qui permet à l'œil de mélanger les images entre elles. Il est possible de simuler cet effet, appelé *motion blur* en anglais, en infographie.

Développer pour le temps réel requiert donc une gestion optimale des performances du matériel disponible. Faire tourner un programme à soixante images par seconde nécessite de calculer une image en seize millisecondes, étant précisé qu'une image dans une résolution de 1920 pixels par 1080 pixels représente environ deux millions de pixels.

Il est également important de prendre en compte la fréquence de rafraîchissement du dispositif qui affichera l'image. Il est vain de prévoir d'afficher soixante images par seconde si l'œuvre est destinée à être diffusée sur un support qui a une fréquence de rafraîchissement de 30 Hz (l'hertz est l'unité de fréquence d'un phénomène périodique dont la période est une seconde) : un écran ne peut afficher qu'un nombre d'images correspondant au quotient de sa fréquence divisée par un nombre entier : un écran dont le taux de rafraîchissement est de 60 Hz ne pourra afficher que soixante, trente, vingt, quinze, douze,... images par seconde (Akenine-Möller et al., 2008).

1.2.2.2 Le rendu non photoréaliste dans le jeu vidéo

Dans la recherche de rendu non photoréaliste en temps réel pour le jeu vidéo, on retrouve principalement des effets simulant le dessin en renforçant le tracé des contours ou en utilisant un style de rendu dit *cartoon* ou *toon shading*. Ce dernier consiste à utiliser une palette de couleurs limitée pour définir l'aspect d'un objet. Ces deux procédés sont souvent combinés pour former un effet de *cel shading* utilisé dans des jeux comme « *Jet Set Radio* », « *XIII* » ou « *The Wolf Among Us* ».

Illustration n°16 : Telltale Game, *The Wolf Among US*(2013)

Moins couramment, le concepteur de jeu vidéo recourt à des styles plus originaux comme dans le jeu « *Ōkami* »dont le concept du rendu est fondé sur la technique du lavis utilisée dans les estampes japonaises ou encore dans le jeu « *Sacramento* » avec une imitation de la peinture aquarelle.

Illustration n°18 : Clover Studio, *Okami* (2006)

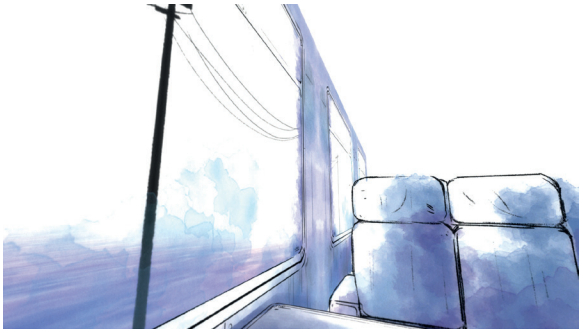


Illustration n°17 : Dziff, *Sacramento* (2016)

L'exposé de Junya Christopher Motomura, qui travaille à *Arc System Work*, lors de la conférence « *GuiltyGearXrd's Art Style : The X Factor between 2D and 3D* » lors de la *Game Developers Conference 2015* (Motomura, 2015) apporte un éclairage très intéressant : il y expliquait son travail de recherche et développement visant à donner une impression d'illustration en deux dimensions (2D) dans un style manga à des modèles en 3D. Au-delà du rendu, il a souligné l'importance de l'animation et abordé les différentes problématiques rencontrées lors de la production. Cette conférence est incontournable pour ceux qui s'intéressent au rendu non photoréaliste. Ce qui vraiment intéressant dans le travail de cette équipe, c'est l'intention des chercheurs. En effet, après avoir constaté qu'ils avaient atteint, sur des productions en 2D, la qualité maximum qu'ils étaient en mesure de produire, ils ont décidé de se lancer un défi technique en réalisant leur jeu en 3D avec comme contrainte de conserver la même esthétique qu'en 2D. L'objectif sous-tendu était de faire progresser la recherche et d'encourager les autres à faire de même.

C'est une démarche originale, car force est de constater qu'il y a très peu de programmes de recherche ou de projets avec de telles prises de risque dans le domaine du rendu non photoréaliste et encore moins dans celui du rendu pictural.

Illustration n°19 : Arc System Works, *Guilty Gear Xrd* (2014)

1.2.2.3 Les images évolutives

Le temps réel a un autre atout majeur : c'est sa capacité à réaliser des images qui vont pouvoir évoluer sans que leur fin soit prédéfinie. Le travail *Genetic Images* (1999) de Karl Sims en est un parfait exemple. Son programme génère une série de onze images qui vont être affichées sur des écrans. Le spectateur se place devant les images qu'il préfère, ce qui permet de définir celles qu'il considère comme les meilleures. Puis une nouvelle génération d'images est produite en fonction de l'endroit où s'est placé le spectateur. Il s'agit d'un système d'algorithme génétique orienté par le choix d'une valeur sélective définie par le spectateur lui-même qui, par croisements et mutations, va produire ensuite une nouvelle génération d'images.

A partir de principes similaires, il est possible d'imaginer des œuvres sans fin qui s'analyseront elles-même pour évoluer.

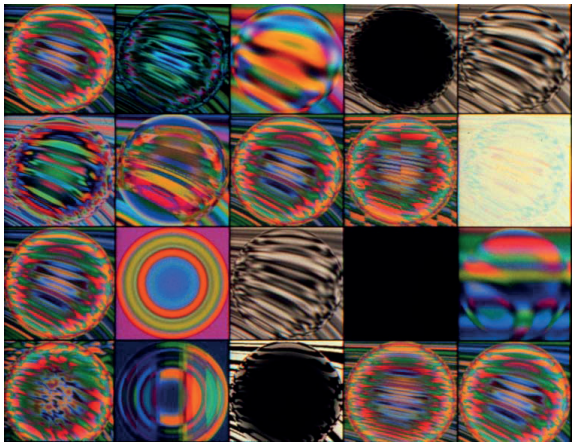


Illustration n°20 : Karl Sims, *Genetic Images* (1999), Echantillon de 20 images représentant une génération

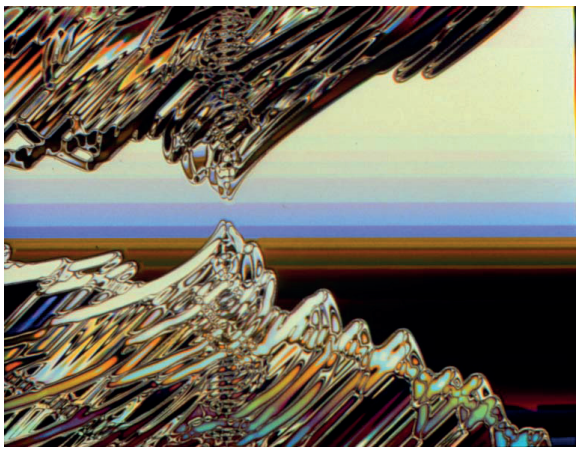


Illustration n°21 : Karl Sims, *Genetic Images* (1999), Un des résultats possibles

1.3 Aperçu de l'histoire des techniques de génération automatique d'effets picturaux

Pour obtenir une image numérique dans un style pictural, les techniques disponibles nous offrent la possibilité de transformer une image en 2D ou une scène en 3D. L'option retenue est communiquée à l'ordinateur en informations d'entrée (*inputs* en anglais).

La scène en 3D permet de disposer d'un plus grand nombre d'informations telles que la distance précise des objets par rapport à la caméra ou l'orientation des faces de ces objets, ce qui augmente d'autant les potentialités de création.

Dans les paragraphes qui suivent, je vais évoquer les recherches existantes dans le domaine du rendu pictural. Je vais tout d'abord parler des techniques fondées sur des images comme informations d'entrée, puis sur celle se basant sur des scènes en 3D. Je finirai par les techniques destinées aux applications temps réel.

1.3.1 Le rendu artistique à partir d'une image : les techniques IB-AR

Il convient en premier lieu de s'intéresser à l'état de l'art en matière de techniques de rendu artistique réalisé à partir d'une image : on part d'une image de base qui est traitée au moyen de processus automatiques ou semi-automatiques pour obtenir en sortie une image avec un style artistique. Ces techniques sont décrites sous l'acronyme générique IB-AR qui signifie *image-based artistic rendering* en anglais.

Les différentes techniques sont généralement expliquées dans des publications de chercheurs diffusées lors de conférences ou dans des magazines par des éditeurs en ligne tels que l'*ACM Digital Library* (*ACM SIGGRAPH*) ou l'*Eurographics Digital Library*, spécialisés dans le domaine de l'infographie (*computer graphics* en anglais).

Pour les publications antérieures à l'an 2000, l'information est disponible en édition papier. Citons les livres de référence que sont les ouvrages « *Non-photorealistic Rendering* » (Gooch & Gooch, 2001) ou « *Non-photorealistic Computer Graphics* » (Strothotte & Schlechtweg, 2002).

Mon historique des techniques IB-AR sera toutefois essentiellement fondé sur deux états de l'art réalisés en 2013, le premier par des chercheurs de l'INRIA (Kyprianidis et al., 2013) et le second par des chercheurs de la Bournemouth University (Hegde et al., 2013).

Les premières recherches en la matière concernaient la simulation de coups de pinceau. Les avancées dans ce domaine ont permis de développer des méthodes dites de *Stroke-based Rendering* (SBR). Cela consiste à placer des traits de peinture avec certaines couleurs, orientations et tailles pour reformer une image.

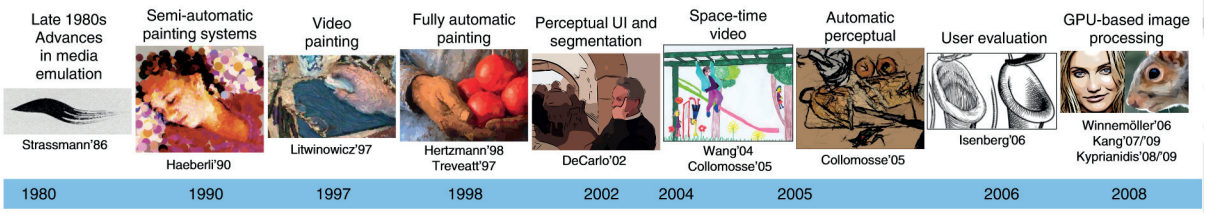


Illustration n°22 : KYPRIANIDIS et al, Chronologie du développement des techniques IB-AR (2013), A Taxonomy of Artistic Stylization Techniques for Images and Video

Finalement le procédé reproduit la manière de faire des peintres traditionnels qui consiste à placer des coups de pinceaux successifs pour construire une image.

Paint by Numbers: Abstract Image Representations est l'une des premières publications à traiter de ce sujet. Éditée en 1990, elle définit les bases de cette technique (Haeberli, 1990). Les coups de pinceaux peuvent être des formes primitives générées par l'ordinateur ou des formes réalisées au préalable, à l'aide de techniques traditionnelles, par les artistes. Leur taille et leur orientation peuvent être définies de manière aléatoire ou à la main ou encore en se basant sur des *low level image processing*. Il s'agit de techniques d'analyse fondées sur l'observation de l'image par l'ordinateur, aussi appelée vision par ordinateur (*computer vision* en anglais) qui permet généralement de détecter des contours, des flux optiques ou des gradients d'intensités.

Illustration n°23 : Paul Haeberli, Utilisation d'une seconde image pour contrôler la direction des coups de pinceau (1990), Paint By Numbers: Abstract Image Representations

Deux approches sont possibles pour traiter ces images : localement ou globalement. Dans la première approche, le programme observe un pixel et les pixels voisins pour évaluer les caractéristiques du coup de pinceau alors que dans la seconde, il agit sur l'image pour la traiter comme un ensemble au moyen de techniques telles que la relaxation ou l'utilisation de champ de forces (*flow field* en anglais).

Il existe des alternatives aux techniques entièrement automatiques : il est ainsi possible de créer une interface qui permet à l'utilisateur de guider le positionnement des coups de pinceaux. Ces techniques semi-automatiques ne seront pas utilement mises en œuvre pour réaliser de l'animation en temps réel.

Le rendu d'image fixe et le rendu vidéo constituent deux champs de recherches distincts. En effet, avec les images animées de nouvelles problématiques se mettent en place. Le *flickering* désigne le clignotement perçu quand il y a trop de variations entre des images successives dans une vidéo. Ce clignotement peut être diminué par des méthodes fondées sur l'analyse du mouvement.

Parallèlement aux techniques de rendu du coup de pinceau se développent des techniques permettant de représenter d'autres styles comme la mosaïque : il s'agit ici de simplifier des formes ou de répéter des motifs.

Dans les années 2000, se développe une méthode appelée *Region based* qui consiste à segmenter une image afin de séparer les différents éléments qui la composent pour y appliquer des effets de différentes manières. Ainsi, le traitement appliqué au feuillage d'arbre sera différent de celui retenu pour traiter un visage.

Les chercheurs continuent à utiliser les techniques de peinture par couches mais avec plus de précision sur les zones dites de *salience* (Itti, 2007) qui correspondent aux zones de focalisation du regard au sein de l'image. Ce principe a été récemment très bien expliqué et développé dans une publication : *From Image Parsing to Painterly* (Zeng et al., 2009).

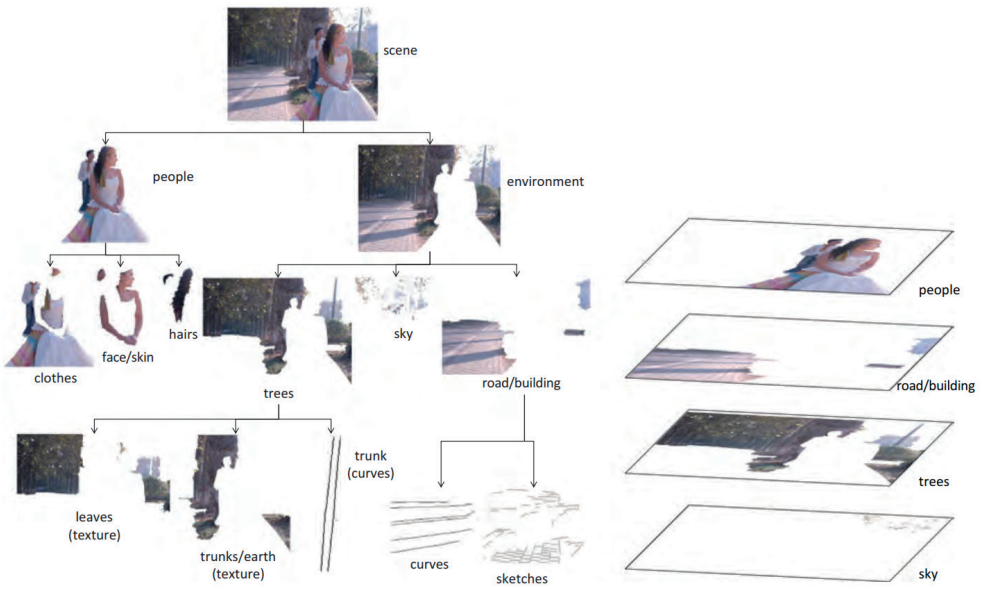


Illustration n°24 : Zeng et al, Analyse et segmentation d'une image (2009), From Image Parsing to Painterly Rendering

Une autre approche du rendu fondée sur la comparaison entre images, appelée *example-based rendering*, a été développée par Aaron Hertzmann dans sa thèse puis avec un groupe de chercheurs qu'il a intégré en 2001 au NYU Media Research Lab (Hertzmann et al., 2001). Cette méthode est exposée ci-après de manière plus détaillée.

Enfin, notons un procédé qui consiste à appliquer des filtres d'effets sur une image. Chaque pixel est ainsi modifié selon un algorithme précis. Avec cette technique, il est notamment possible de définir la nouvelle couleur d'un pixel en lui appliquant la moyenne des couleurs observées chez ses huit voisins les plus proches. Toutefois, les résultats de cette technique sont globalement peu satisfaisants comparés à celles précédemment décrites.

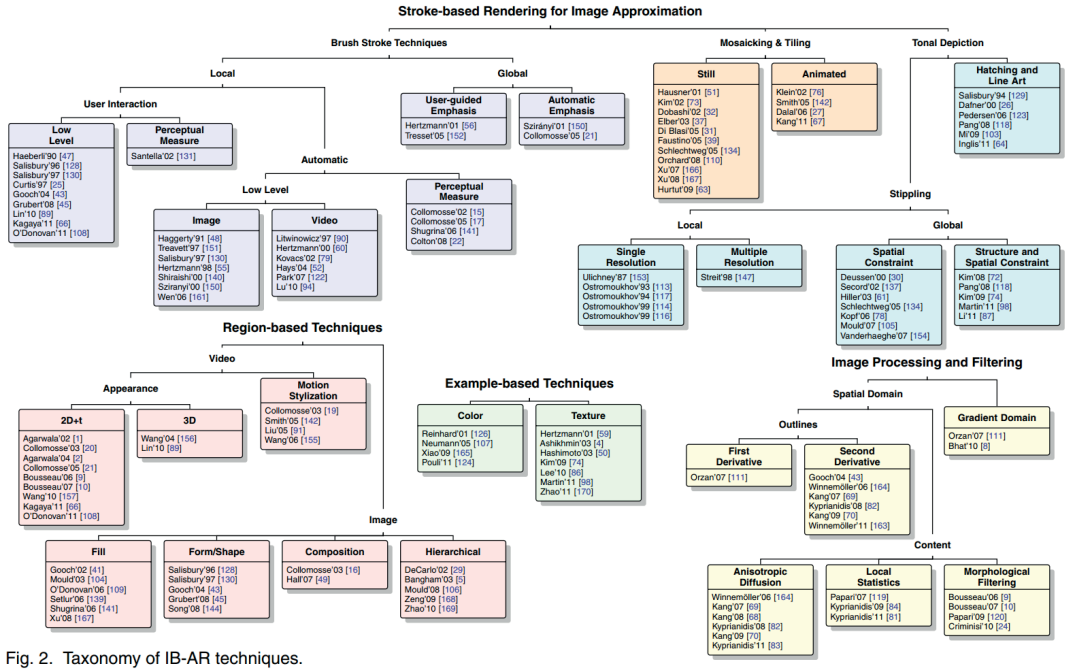


Fig. 2. Taxonomy of IB-AR techniques.

Illustration n°25 : KYPRIANIDIS et al, Taxonomie des techniques IB-AR (2013), A Taxonomy of Artistic Stylization Techniques for Images and Video

Pour recentrer cet état de l'art sur mon sujet d'étude, il convient d'approfondir parmi ces techniques celles qui présentent le plus grand intérêt pour l'automatisation de rendu animé en temps réel.

En premier lieu, évoquons le travail de Peter Litwinowicz qui, en 1997, fut le premier à proposer un filtre automatique d'effet de peinture en gardant une cohérence temporelle, travail décrit dans sa publication *Processing Images and Video for An Impressionist Effect* (Litwinowicz, 1997). Le principe consistait principalement à placer automatiquement des traits rectangulaires en fonction d'une détection de contour des formes.

A l'époque, le résultat était limité à des formes primitives telles que des lignes, des rectangles ou des cercles. En 1998, Hertzmann fut le premier à proposer un système de coups de pinceau courbés en utilisant un système de *splines* qui permet de générer un coup de pinceau dont la couleur sera la moyenne des pixels parcourus par la courbe (Hertzmann, 1998). Il déve-

loppa également avec cette méthode un système simulant la manière dont les peintres utilisent plusieurs couches de peinture superposées. Son programme dessine une première couche de peinture avec des coups de pinceau très larges, et couche par couche, il va s'affiner dans les zones où il y a plus de détails. Sur certains de ses travaux, des détails sont ajoutés après les coups de pinceau à l'aide de texture simulant du relief dans le but de rendre les coups de pinceau plus réalistes.

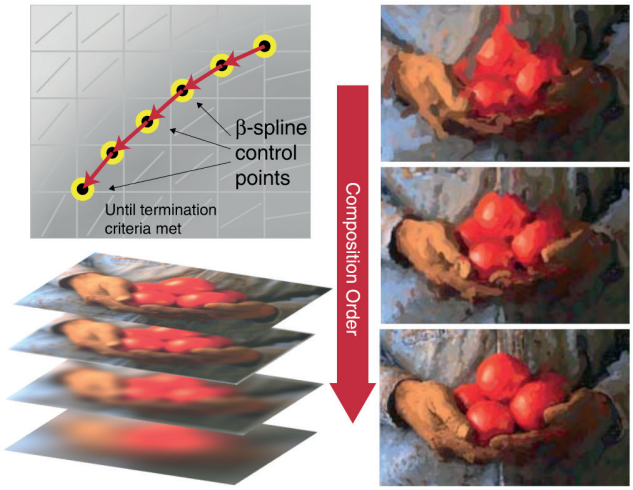


Illustration n°26 : KYPRIANIDIS et al, Algorithme de peinture à l'aide de splines par Hertzmann (1998), A Taxonomy of Artistic Stylization Techniques for Images and Video

Par la suite, des chercheurs ont commencé à s'intéresser au traitement d'images animées et notamment au traitement consistant à appliquer des effets de rendu pictural à des vidéos. Objectif simple de prime abord, mais complexe à mettre en œuvre.

Plusieurs problèmes sont en effet à résoudre : le *flickering* évoqué précédemment ou les problèmes de cohérence temporelle et de glissement qui se présentent également dans l'animation traditionnelle, quand des images successives diffèrent de trop. Dans la plupart des dessins animés, le problème est contourné par l'utilisation d'aplats de couleur. L'animateur Bill Plympton, de manière plus originale, a détourné ce problème de scintillement en se l'appropriant pour créer une identité visuelle personnelle.

Aaron Hertzmann et Ken Perlin ont travaillé ensuite sur un système de peinture en temps réel avec leur *Living Painting* (Hertzmann & Perlin, 2000). Le programme analyse les images et ne font évoluer, sur les images suivantes, que certaines zones ciblées dans la scène. Ils utilisent les techniques précédemment citées pour générer la première image puis créent de nouvelles couches uniquement dans les zones où il y a du mouvement.

Sont ensuite publiées des études sur l'analyse des zones d'attention du regard dans une peinture. Le peintre qui conçoit une œuvre va utiliser différents niveaux de détails en fonction des éléments qu'il peint, du plan dans lequel se trouvent les objets, etc. Cette méthode est transposable en imagerie numérique. Le première principe retenu est la segmentation de

l'image. Il s'agit d'en séparer les différentes parties, arbres, bâtiments, personnes. Chacune de ces parties va être traitée de manière spécifique (Zeng et al., 2009). S'y adjoint un autre principe qui consiste à travailler sur des images de plus basse résolution dans les zones où il y aura moins de détails. Ces méthodes peuvent être complétées par l'utilisation d'outils d'analyse du regard pour identifier plus précisément les zones qui attirent l'attention de l'observateur (Santella & DeCarlo, 2002).

En 2001, la publication « Image analogies » (Hertzmann et al., 2001) introduit le concept de « *Example Based Rendering* » (ou EBR) qui signifie en français « rendu par l'exemple ». C'est le début des techniques d'apprentissage pour le transfert de style en peinture. En prenant une image A et une image A' qui représente A de manière stylisée, on identifie la transformation de A en A' afin de l'appliquer à une image B pour la transformer en B'. Le style de B' est le même que celui de A'. L'algorithme consiste à trouver des zones (appelées *patches* en anglais) similaires entre A et B puis à aller chercher le patch correspondant dans A' pour l'appliquer sur B'. Généralement il observe la luminance, c'est à dire une image en noir et blanc pour ne transférer que les informations sur l'intensité lumineuse et non pas sur les couleurs.

Avec l'évolution des technologies, des chercheurs s'intéressèrent ensuite à l'utilisa-

Illustration n°27 : Hertzmann et al., Un exemple du concept d'analogie d'image (2001), Image Analogies

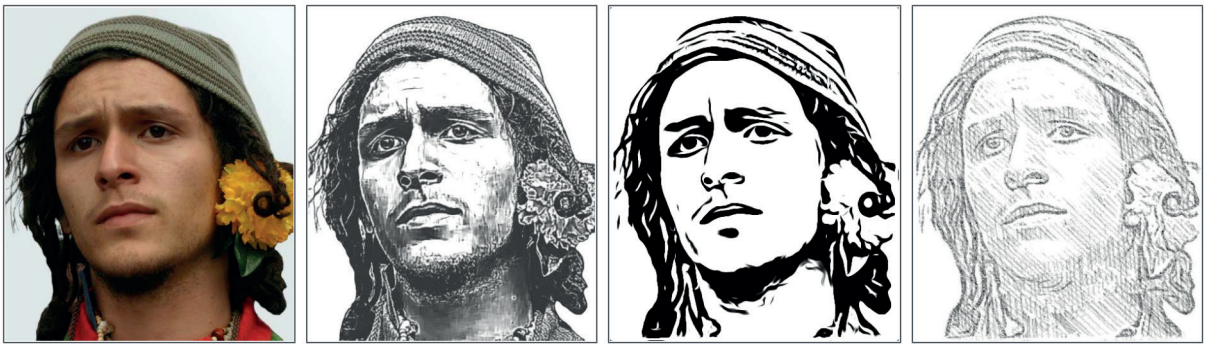
tion des techniques de vision par ordinateur (*computer vision*) à des fins de stylisation de vidéo. Pour régler le problème de la cohérence temporelle, ils ont commencé à élaborer des méthodes d'analyse par pixel en se servant des différences entre l'image courante et les images précédentes. Après les années 1990, l'accent est mis sur les *optical flows* et la segmentation pour améliorer la cohérence temporelle. L'*optical flow*, procédé qui consiste en l'analyse de la vélocité apparente d'un objet dans une séquence vidéo, présente un défaut majeur : il est inefficace dans les zones de couleurs unies.

Certaines recherches s'intéressent à la stylisation du mouvement. Des lignes ajoutées à une vidéo permettent par exemple de symboliser la direction d'un mouvement.

Force est de constater que la plupart de ces techniques ne sont pas réalisables en temps réel et que leur implémentation est impossible sur des architectures parallèles (architectures pouvant exécuter plusieurs tâches simultanément – voir informations plus détaillées dans la partie 2.1.1). Toutefois, avec l'utilisation de filtres d'images, le système peut devenir hautement parallèle et être considéré comme une solution viable pour des applications temps réel. Bien que cette technique ne permette pas d'avoir un contrôle précis du résultat final, elle est

efficace pour du traitement d'images animées (Winnemöller, 2011).

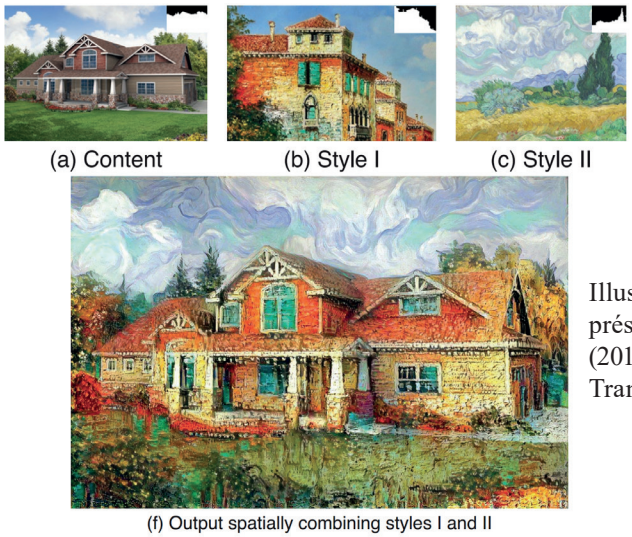
En septembre 2015, la publication « *A Neural Algorithm of Artistic Style* » (Gatys et



(a) Source (b) XDoG (c) XDoG Thresholding (d) XDoG-Hatching

Illustration n°28 : Winnemöller, Différentes variations d'un filtre de différence de gaussiennes (2011), XDoG: Advanced Image Stylization with eXtended Difference-of-Gaussians

al., 2015) propose une approche fondée sur des *Deep Neural Networks* et plus précisément sur les *Convolution Neutral Networks* (Gatys et al., 2016b) pour transférer le style d'une image à l'autre. Pour ce faire, le réseau de neurones recherche des motifs et apprend à les reconnaître ; la méthode est similaire à celles des algorithmes de reconnaissance de visages ou de formes (Taigman et al., 2014). Une image présentant un certain style est analysée afin d'y reconnaître des formes ou des objets particuliers puis l'autre image est analysée à son tour afin d'y identifier des zones similaires. On va donc tenter d'extraire les zones de similarités entre les images, en séparant la reconnaissance de style et de contenu d'une image. Pour affiner le résultat, plusieurs couches sont analysées de manière de plus en plus détaillée et l'opération est réitérée jusqu'à que le réseau de neurones ait analysé l'image, d'un point de vue visuel, pour pouvoir reproduire son style sur une autre. Dans cette première publication, la texture d'une image est transférée à l'autre image, l'image modifiée prenait l'apparence générale et la couleur de l'image définissant le style. Depuis, plusieurs autres travaux ont permis d'améliorer l'algorithme, par expérimentation de son application sur des vidéos (Ruder et al., 2016) et par la recherche de préservation de la couleur (Gatys et al., 2016a). Les résultats produits commencent à être très convaincants.



(a) Content (b) Style I (c) Style II

(f) Output spatially combining styles I and II

Illustration n°29 : Gatys et al, Transfert de style par préservation de la couleur et par mélange de styles (2016), Controlling Perceptual Factors in Neural Style Transfer

1.3.2 Le rendu artistique à partir d'une scène 3D

Une image numérique avec un rendu artistique peut également être produite à partir d'une scène en 3D. Dans *Paint by Numbers: Abstract Image Representations* (Haeberli, 1990), Haeberli utilise ces données pour orienter les coups de pinceau que l'utilisateur place en cliquant avec la souris. Mais ces traces restent confinées dans un espace 2D. C'est en 1996 que Meier, dans *Painterly Rendering for Animation* (Meier, 1996), propose une technique où les coups de pinceaux ne sont plus placés dans un espace 2D mais dans l'espace 3D. Cela permet d'avoir les coups de pinceaux intégrés dans la scène 3D et chaque coup de pinceau a une couleur, une position et une orientation définies dans l'espace permettant d'avoir une très bonne cohérence temporelle.

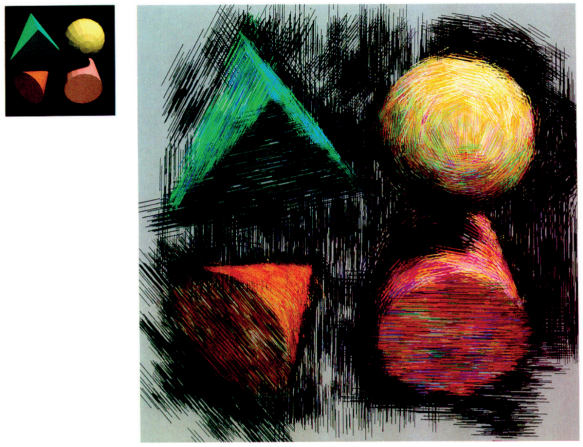


Illustration n°30 : Haeberli, Traits de pinceau par rapport à une géométrie (1990), Paint By Numbers: Abstract Image Representations

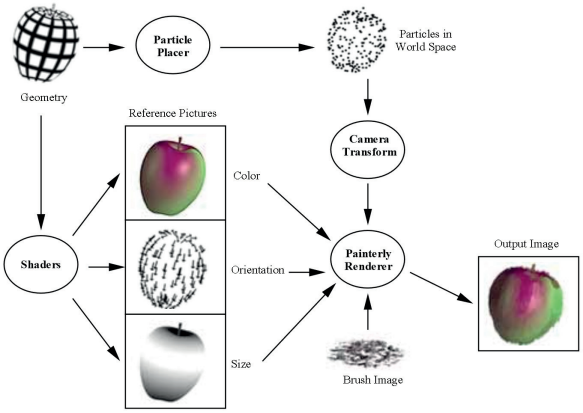


Illustration n°31 : Meier, Un exemple du pipeline du rendu de peinture (1996), Painterly Rendering for Animation

Un bon exemple de rendu aquarelle utilisant les informations 3D est décrit dans *Real-time Watercolor Illustrations of Plants Using a Blurred Depth Test* (Luft & Deussen, 2006) : des informations de volume et de profondeur sont utilisées pour créer l'effet souhaité.

La technique la plus simple mais aussi la plus chronophage consiste à produire directement des textures simulant la peinture. La série *Carnet de Voyage* réalisée pour ARTE illustre bien ce principe tout comme les illustrations 32 et 33 issues du site *Sketchfab*.

Illustration n°32 : Emilie Stabell, *The journey* (2016)

Illustration n°33 : Andrey Lizunov, *Bluff* (2016)

Avec cette technique, plus proche de la méthode traditionnelle, l'artiste a beaucoup de contrôle sur le résultat final. Il est possible d'obtenir une grande variété de styles et les textures sont susceptibles d'être animées. Toutefois, cette technique ne fonctionne bien que pour un éclairage et une ambiance fixes et elle exige beaucoup de travail.

1.3.3 Le rendu temps réel

Pour terminer ce panorama, arrêtons-nous sur les techniques spécifiques de rendu non photoréaliste pour les applications temps réel. Comme précisé en liminaire, l'ordinateur ne dispose que d'un très court temps d'exécution pour générer une image. Par ailleurs certaines techniques citées précédemment doivent être écartées car trop gourmandes en ressources ou parce qu'elles nécessitent des informations non disponibles telles que les images futures qui, par définition, n'existent pas encore à l'instant t dans une application temps réel.

Le livre *Real-time Rendering* (Akenine-Möller et al., 2008) explique en détails le fonctionnement de création d'une image pour le temps réel. Sont principalement utilisées les ressources de la carte graphique et des langages de programmation dédiés à l'image. Ces aspects seront développés en deuxième partie.

La consultation du site *Shadertoy*, qui regroupe des travaux temps réel sur une page web utilisant la technologies webGL, permet de découvrir certains effets qu'il est possible d'obtenir avec de la programmation graphique.

Illustration n°34 : Flockaroo, *watercolor* (2016), https://www.shadertoy.com/view/ltyGRV

Nous avons vu que le numérique étend le domaine de la peinture traditionnelle en nous offrant de nouvelles possibilités esthétiques et de nouveaux outils de travail.

Les recherches en rendu non photoréaliste inspiré de la peinture ne s'intéressent que très peu aux problématiques du temps réel. De plus, il s'agit d'un domaine dont l'évaluation des résultats et les méthodologies ne sont pas précisément définies, ce qui complique la recherche. Ce sont précisément pour ces raisons qui je m'y suis intéressé.

Dans le chapitre qui suit, j'exposerai mes travaux de recherche et développement visant à élaborer une méthode automatique de rendu de peinture pour le temps réel.

2. Recherche et développement : nouveaux processus pour les applications temps réel

Le but de ces recherches est de créer un effet pictural évoquant la peinture à l'huile utilisable dans des applications temps réel. L'idée principale est d'obtenir des résultats se calculant le plus rapidement possible et nécessitant le minimum d'interventions de l'artiste. L'objectif est de mettre en place une intention artistique picturale qui va permettre de sortir des limitations visuelles propres aux moteurs de rendu ou à l'image filmée. Pour cela, je vais principalement utiliser des techniques fondées sur la programmation graphique en C++ et GLSL ainsi que certaines méthodes de composition d'images en temps réel à l'aide du logiciel TouchDesigner 088.

Comment automatiser les processus de stylisation pour gagner du temps dans les productions artistiques ? Est-ce qu'une nouvelle esthétique peut ressortir de ces processus expérimentaux ? Avant de tenter de répondre à ces questions, comprendre la manière dont une image est calculée en temps réel sur l'ordinateur est incontournable. Une meilleure appréhension du processus de création d'une image va en effet permettre de mieux envisager les possibilités et techniques disponibles pour transformer cette image dans un style pictural.

Lors de la réalisation de projets numériques, on dispose non seulement d'éléments en 3D mais également d'éléments en 2D. C'est pourquoi je proposerai des expérimentations dans ces deux domaines, pour pouvoir ensuite choisir les solutions les plus adaptées en fonction du projet à réaliser. Cette approche permettra de commencer à réfléchir à une méthode qui pourra s'adapter à toutes les situations.

Je vais ainsi développer des pistes existantes que je considère comme des techniques qui peuvent aboutir ou proposer de nouvelles méthodes de rendu.

2.1 Précisions techniques liminaires, nécessaires à la compréhension de la démarche

2.1.1 CPU et GPU

Le principe du rendu est de transformer des informations 3D telles que des objets, des lumières ou des modèles d'éclairage en une image en deux dimensions à partir d'une caméra virtuelle. Il existe plusieurs méthodes de rendu. Certaines d'entre elles, comme les moteurs de rendu à lancer de rayons (*raytracer* en anglais), sont peu utilisées en temps réel et ne seront donc pas abordées dans cette étude qui sera circonscrite au fonctionnement du processus de calcul standard d'une image 3D en temps réel au moyen de la carte graphique qui est, aujourd'hui, la méthode la plus utilisée dans les moteurs de rendu.

Le *vertex* (au pluriel *vertices*) est une notion fondamentale : les objets 3D sont représentés par des *vertices* qui sont l'équivalent de points dans l'espace 3D auxquels sont associées des caractéristiques. Les *vertices* peuvent en effet « contenir » plusieurs types d'informations telles qu'une position, une couleur, un vecteur normal ou des coordonnées de texture. Ils peuvent être reliés entre eux pour former des lignes ou des faces triangulaires nommées « primitives ». L'affichage de ces primitives à l'écran est réalisé à partir de librairies graphiques : DirectX, Vulkan ou OpenGL. C'est cette dernière que j'ai retenue.

Pour effectuer les calculs nécessaires au programme visant à rendre une scène 3D, l'ordinateur possède deux unités de traitement, le CPU (pour *central processing unit*) et le GPU (pour *graphic processing unit*).

Le CPU possède généralement plusieurs cœurs performants qui vont effectuer les tâches requises par l'exécution d'un programme informatique. Chaque cœur du CPU travaille en série, c'est-à-dire qu'il effectue les opérations l'une après l'autre : il doit terminer l'exécution d'une première opération avant de pouvoir passer à l'opération suivante. Les lignes de code sont lues de manière linéaire, les unes après les autres. Quand plusieurs cœurs sont disponibles, ils travaillent en parallèle, ce qui permet l'exécution simultanée de plusieurs opérations. Mais la conception du CPU n'est pas optimale pour certaines tâches. En effet, pour afficher une image sur un écran, il faut calculer la couleur de chaque pixel. Et une résolution de 1920 par 1080 pixels demande un calcul de 2 070 600 pixels par image, ce qui est une opération trop coûteuse en ressources pour les quatre à huit cœurs que comporte en général un CPU, surtout quand l'objectif est de calculer soixante images par seconde.

Pour ce type de traitement, il est donc incontournable de recourir à une autre unité de traitement : le GPU. Celui-ci possède un nombre de cœurs beaucoup plus important. Pris

isolément, chaque cœur d'un GPU est certes moins performant que celui du CPU, mais pris dans leur ensemble, les cœurs du GPU permettent d'effectuer en parallèle de très nombreuses opérations simples. Le GPU à un autre avantage : certaines opérations mathématiques sont calculées plus rapidement ; il en est ainsi des opérations de trigonométrie et des calculs matriciels.

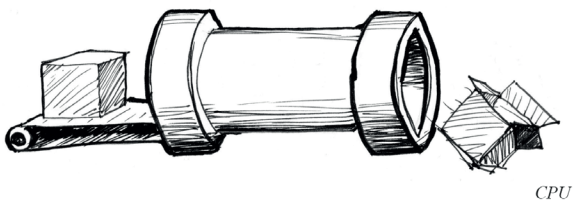


Illustration n°35 : Patricio Gonzalez Vivo, *CPU* , The book of shaders

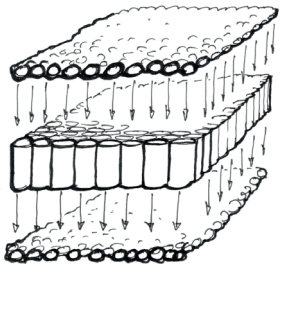


Illustration n°36 : Patricio Gonzalez Vivo, *GPU* , The book of shaders

Le CPU envoie des informations au GPU qui prend en charge les opérations de rendu. Le CPU est ainsi rendu disponible pour exécuter d'autres tâches. Ce système, qui permet au programme principal de continuer à s'exécuter pendant que des calculs se réalisent en parallèle, s'appelle des *asynchronous operations*. Plusieurs librairies (OpenCL, CUDA,...) permettent ce type d'opérations : elles envoient des commandes à un *driver* qui communique ensuite les informations nécessaires au GPU.

Mais attachons-nous d'abord à décrire le déroulement du pipeline graphique pour afficher une scène 3D, ce qui nous permettra d'avoir une approche globale et une meilleure compréhension du rendu d'images temps réel, en 3D aussi bien qu'en 2D.

Une carte graphique possède généralement son propre bloc mémoire appelé VRAM (pour *Video Random Access Memory*). Celui-ci contient au moins le *front* et le *back buffer* ainsi que tous les autres *frame buffers* (un *buffer* est une mémoire tampon permettant de stocker temporairement des données). Ces notions seront explicitées dans les développements qui suivent. C'est également dans la VRAM que seront stockées toutes les textures utilisées par le programme.

2.1.2 Le pipeline de rendu temps réel

Les explications à suivre, relatives au pipeline de rendu temps réel, sont une compilation des informations tirées des ouvrages de référence suivant : « *Real time rendering*» (Akenine-Möller et al., 2008), « *Graphic Shaders* » (Bailey & Cunningham, 2016), « *Mathematic for 3D Game Programming and Computer Graphics* » (Flynt & Lengyel, 2011) et « *3D Math Primer for Graphics and Game Development* » (Dunn & Parberry, 2011). Je ne peux qu'en recommander la lecture à toute personne intéressée par la programmation graphique et les mathématiques appliquées à l'image de synthèse.

Le rendu d'une image temps réel s'effectue en trois grandes étapes : (1) *application stage*, (2) *geometry stage* et (3) *rasterization stage*. Les deux dernières peuvent être divisées en sous-étapes. Il est important de noter que la vitesse d'un programme est conditionnée par l'étape qui a le plus long temps d'exécution : cette étape est appellée *bottleneck* (goulot d'étranglement).

L'*application stage* se déroule sur le CPU. C'est ici que sont effectués les calculs de détections de collision, les simulations physiques ou les animations.

Puis intervient la *geometry stage* au cours de laquelle sont réalisés les transformations des objets 3D, les calculs de modèles de projections, etc. L'exécution de cette étape se déroule sur le GPU et va servir à définir les éléments qui seront dessinés.

Enfin survient la *rasterization stage* : c'est l'étape où l'image est dessinée sur le *viewport* à partir des données fournies au cours de l'étape précédente, complétées des données éventuellement définies par pixel. La notion de *viewport* est fondamentale : c'est la zone de l'écran où l'image sera affichée.

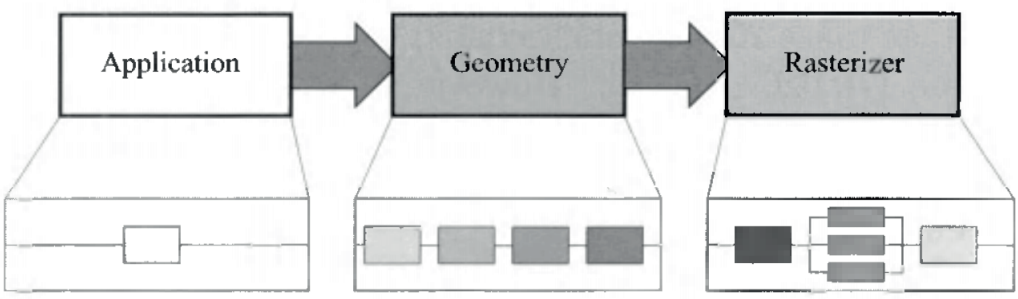


Illustration n°37 : Akenine-Möller et al, *Aperçu du pipeline de rendu graphique* (2008), Real Time Rendering

(1) *Application stage*

A la fin de l'*application stage*, les informations de géométrie, constituées de primitives de rendu qui sont des points, des lignes ou des triangles, doivent être envoyées à l'étape suivante.

C'est la seule étape qui n'est pas décomposée en sous parties. Il est toutefois possible de lancer l'exécution sur plusieurs cœurs en parallèle pour améliorer les performances. C'est également lors de cette étape que seront captées les interactions avec l'utilisateur.

(2) *Geometry stage*

Cette étape se décompose en plusieurs sous-parties : *model & view transform*, *vertex shading*, *projection*, *clipping* et *screen mapping*.

- Model & View transform

Le but de cette étape est de transposer les coordonnées des *vertices* de leur repère local vers un espace défini par rapport à une caméra.

Les coordonnées d'un objet se définissent à partir d'un repère généralement orthographique nommée *model space*, *object space* ou encore *local space* qui correspond à un repère cartésien de trois dimensions. Ce repère va servir à définir une base spécifique à un objet et sert, par exemple, à définir l'emplacement de ses *vertices* de manière locale. Les coordonnées locales d'un objet sont appelées *model coordinates*.

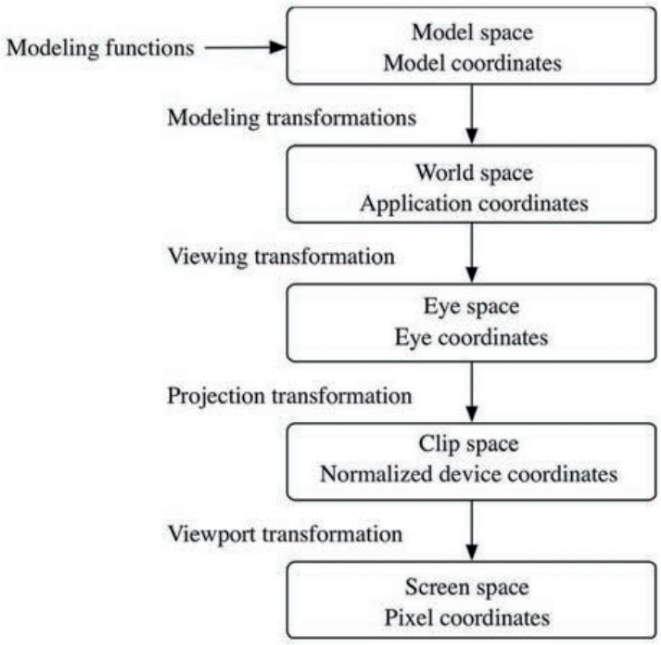


Illustration n°38 : Bailey & Cunningham, Transformation des vertex lors de la geometry stage (2016), Graphic Shaders

Le *world space* est un repère qui va contenir tous les éléments de la scène, tous les objets vont donc disposer d'une origine commune dans cet espace. C'est généralement en *world space* que les calculs d'éclairage sont effectués. La transposition des *model coordinates* en *world coordinates* est appelée une *model transform*.

Pour obtenir un rendu de scène 3D, il faut recourir à une caméra virtuelle qui va déterminer les éléments rendus. Cette caméra est définie par une position dans le *world space* et par une direction vers laquelle elle va pointer. Pour faciliter les calculs des étapes suivantes, la caméra et tous les éléments de la scène vont être transposés dans un *camera space*, aussi appelé *view space* ou *eye space*. Ce repère sert à définir la position des éléments de la scène par rapport à la caméra, celle-ci servant de point d'origine du repère. Pour passer du *world space* au *camera space*, il faut effectuer une *view transform* qui place la caméra sur le point d'origine et l'oriente dans la direction inverse de celle de l'axe (z) : nous sommes dans un repère direct (en anglais *right-handed coordinate system*) où l'axe (y) pointe vers le haut et l'axe (x) vers la droite.

Ces deux transformations peuvent être regroupées dans une *model-view transform* qui permet de passer directement de l'*object space* à l'*eye space*, c'est-à-dire de définir la position des *vertices* par transposition directe des *objet coordinates* en *eye coordinates*.

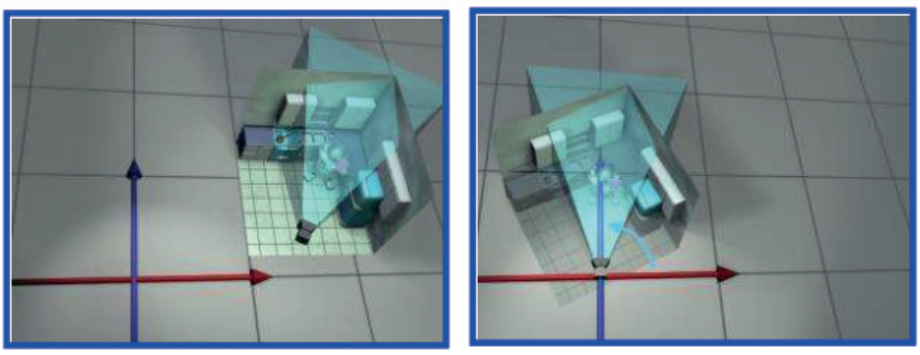


Illustration n°39 : Dunn & Parberry, View Transform, du world space au camera space (2011),3D Math Primer for Graphics and Game Development

- Vertex Shading

Cette étape permet de calculer l'apparence d'un modèle 3D par rapport à son modèle de *shading* et les lumières de la scène. Le modèle de *shading* va définir le matériau d'un objet, la manière dont il sera représenté et dont la lumière va réagir à sa surface. Il existe différentes équations de rendu, appelées *shading equations*, qu'il est possible de calculer lors de cette étape pour réaliser un *per-vertex shading.* Il est également possible de réaliser le calcul du *shading* lors de l'étape de la *rasterisation* pour effectuer du *per-pixel shading.* Ce dernier est légèrement plus long à calculer mais permet d'interpoler les couleurs au niveau du pixel plutôt qu'au niveau du *vertex* (illustration 40). Généralement, les calculs de *shading* sont réalisés par rapport au *world space*, sauf dans certains, cas comme nous le verrons plus tard avec les *litspheres* qui se calculent en *eye space*.

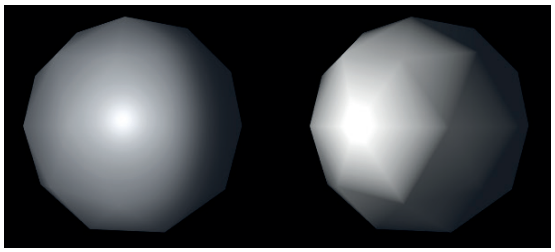


Illustration n°40 : Piotr Sobolewski, A gauche : per-pixel shading / A droite : per-vertex shading (2016), http://blog.theknightsofunity.com/forward-vs-deferred-rendering-paths/

- Projection

La zone de visibilité de la caméra s'appelle un *view frustrum* ou *view volume.* Elle est définie par six limites appelées *clip planes* et par le *field of view* de la caméra. Lors de l'étape de projection, le *view volume* est transformé en un prisme appelé *canonical view volume*. Lors de cette étape, il faut passer d'une *camera space* à un *homogeneous clip space* ou *canonical view volume space* pour faciliter les calculs de l'étape suivante en normalisant les coordonnées x,y,z dans un espace [-1,1]. Après transformation, les modèles ont des *normalized device coordinates* (dénomination retenue dans OpenGL).

C'est lors de cette étape que la vue perspective ou orthographique est transformée en un espace en deux dimensions. Dans une vue orthographique, le *view volume* est représenté par une boite rectangulaire où les lignes qui sont parallèles restent parallèles après la transformation. Dans une vue en perspective, plus l'objet est loin de la caméra plus il va apparaître petit, les lignes qui sont parallèles vont converger à l'horizon et le *view frustrum* est représenté par une pyramide tronquée à base rectangulaire.

La transformation du *view frustrum* en *canonical view volume* se fait à l'aide d'une matrice 4x4 appelée *clip matrix* ou plus communément *projection matrix*. Elle permet donc de passer d'un *camera space* à un *clip space.*

Le passage de l'o*bject space* au *camera space* est possible en une seule opération en utilisant une matrice, contenant le *model-view transform* et la *projection matrix*, appelée *ModelViewProjectionMatrix*.

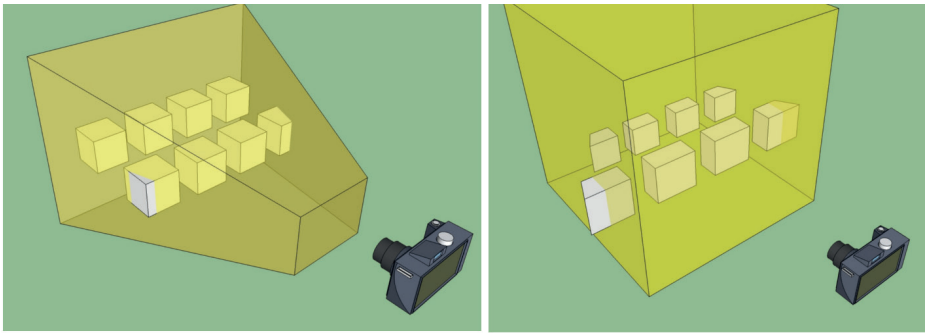


Illustration n°41 : ofBook, De gauche à droite : transformation d'un camera space vers un clip space d'une vue en perspective à l'aide d'une projection matrix, http://openframeworks.cc/ofBook/chapters/openGL.html

- Clipping

Seules les primitives présentent dans le *view frustrum* doivent être envoyées à l'étape de *rasterisation*. Celles qui sont en dehors de la vue ne sont pas envoyées dans les étapes suivantes. Les objets qui sont à la limite du *frustrum* de la caméra requièrent du *clipping*. Ils se retrouvent coupés à la limite du *frustrum* et sont remplacés par de nouveaux *vertices* qui se situent au croisement de la limite du *view volume* et de l'objet. Cette partie est une *fixed-operation hardware*, c'est à dire qu'elle n'est généralement pas programmable.

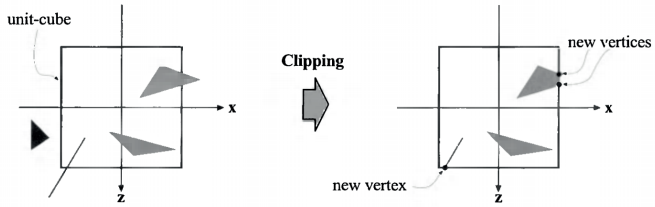


Illustration n°42 : Akenine-Möller et al, Clipping (2008), Real Time Rendering

- Screen Mapping

Également appelée *viewport transformation*, cette étape sert à passer de l'*homogeneous clip space* au *screen space*, aussi appelé *window space*. L'étape du *screen mapping* reçoit donc les primitives qui sont présentes dans le *view frustrum* et qui ont été rognées si nécessaire. Les coordonnées des *vertices* sont encore définies en 3D à ce moment du pipeline. Les composantes x et y de chaque primitive sont être transposées en *screen coordinates*. Celles-ci, complétées des coordonnées en z, sont appelés *window coordinates* et sont représentées dans le *window space*. Les coordonnées x et y sont transformées à l'aide d'une mise à l'échelle pour les faire coïncider avec la résolution de la fenêtre d'affichage, puis translatées pour que l'affichage soit centré dans le *viewport*. La coordonnée z n'est pas affectée par le *mapping* et a donc encore des valeurs comprises entre -1 et 1 (ou parfois entre 0 et 1). Les coordonnées en z sont ensuite écrites dans un *depth buffer* ou *Z-buffer*.

Il faut également être attentif au positionnement de certains systèmes de coordonnées. OpenGL place le repère du *screen space* dans le coin inférieur gauche alors que DirectX

le place dans le coin supérieur gauche, ce qui peut causer des inversions d'image lors du passage de l'un à l'autre.

L'API graphique que j'ai retenue, OpenGL, définit le centre d'un pixel en (0.5, 0.5) et (0.0, 0.0) correspondant donc au coin supérieur gauche d'un pixel. C'est une information qui peut être capitale en cas de recherche de précision dans l'application des textures lors du *rasterisation stage*. DirectX 10 et ses successeurs utilisent ce même système (ce qui n'est pas le cas des versions antérieures à DirectX 10 où le centre d'un pixel était défini en (0.0, 0.0)).

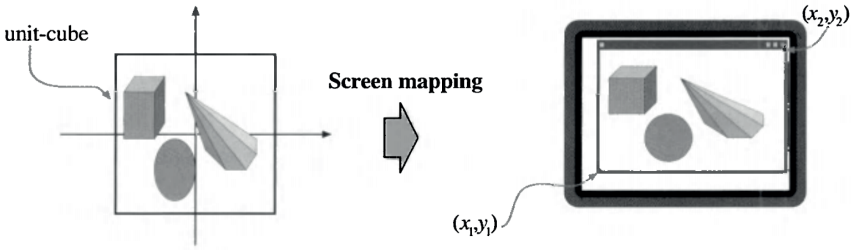


Illustration n°43 : Akenine-Möller et al, Screen Mapping (2008), Real Time Rendering

(3) Rasterization Stage

Après avoir récupéré les informations de la *geometry stage*, la *rasterization stage* va s'occuper du calcul de la couleur de chaque pixel. Ce procédé appellé *rasterization* ou *scan conversion* convertit les triangles présents dans le *window space* en couleur de pixel. Cette étape est divisée en sous-étapes : *triangle setup*, *triangle traversal*, *pixel shading* et *merging*. Avant la *rasterization*, il est possible d'appliquer un *face culling* qui permet de ne pas calculer certaines faces. Il est souvent utilisé pour retirer les polygones qui ne font pas face à la caméra et qui correspondent donc, du point de vue de la caméra, aux faces cachées d'un modèle.

- Triangle Setup

C'est une *fixed-operation hardware* qui calcule les informations présentes à la surface des triangles et réalise, par exemple, l'interpolation des couleurs.

- Triangle Traversal

Cette étape est également une *fixed-operation hardware* qui recherche, pour chaque pixel, le ou les triangles qui le traversent et qui génère un fragment correspondant à la zone d'un pixel recouverte par une géométrie.

- Pixel Shading

Aussi appelé *fragment shading*, le *pixel shading* est l'étape où s'effectuent les calculs de *per-pixel shading* à partir des données interpolées précédemment. Il ressort de cette étape une ou plusieurs couleurs qui seront transmises à l'étape suivante. C'est une opération entièrement programmable et c'est à cette occasion que les opérations d'application des textures sur les modèles 3D sont effectuées. Peut s'ajouter au calcul du *shading* une suite d'opérations de traite-

ment d'image 2D en *screen space* s'exécutant par pixel : opérations de flou ou de saturation des couleurs par exemple. Une fois le fragment calculé, il est envoyé à la dernière étape.

- Merging

C'est lors de cette étape que les problèmes de visibilité sont résolus. C'est plus une étape configurable que programmable.

En premier lieu, le fragment passe par une opération qui ne peut pas être désactivée, le *pixel ownership test*. Ce test détermine si un fragment est visible ou pas dans le *viewport*. Ainsi, si la fenêtre d'un autre programme le recouvre, il ne sera pas rendu.

Puis il est possible d'effectuer un *scissor test*, qui consiste à définir une zone dans le *viewport* : tout ce qui se situe en dehors de cette zone ne sera pas rendu.

Puis vient l'*alpha test* qui correspond à l'évaluation du degré de transparence d'un fragment. Le test compare cette valeur à une valeur constante définie dans l'application, ce qui permet de choisir de ne pas rendre un fragment dont le degré de transparence serait en-deçà de la constante choisie.

Le *stencil test* permet, en stockant des valeurs dans un *stencil buffer*, de spécifier si certains fragments doivent être rendus ou non.

Enfin intervient le *depth test*. Le *depth buffer*, qui fait la même taille que le *color buffer*, a pour chaque pixel une valeur stockée représentant la distance entre la caméra et la primitive la plus proche. Cette valeur est remplacée dès qu'un objet se trouve plus proche de la caméra que la primitive déjà stockée. Il est donc possible de traiter les primitives dans n'importe quel ordre, c'est celle qui a la distance la plus courte par rapport à la caméra qui sera stockée à la fin dans le *Z-buffer*.

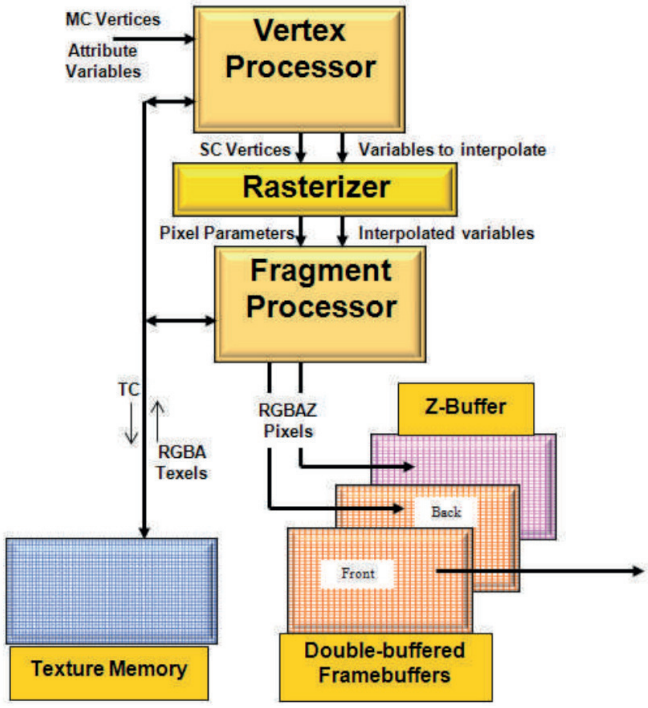


Illustration n°44 : Bailey & Cunningham, Pipeline openGL : système de buffer (2016), Graphic Shaders

Les valeurs de chaque pixel sont stockées dans le *color buffer* qui correspond à un tableau de couleurs. Le *merging stage*, après avoir effectué différents tests facultatifs et le *pixel-lownership* qui lui est obligatoire, combine le *fragment color* de l'étape du *fragment shading* avec les couleurs qui sont actuellement stockées dans le *buffer*.

L'écran va ensuite afficher le contenu du *color buffer* dans le *viewport*. Un système appelé *double buffering* permet d'éviter que l'utilisateur n'observe l'image en train de se former. La scène est rendue en dehors de l'écran dans un *back buffer*. Une fois que le rendu de la scène est terminé, le contenu du *back buffer* est envoyé dans un *front buffer* qui est affiché à l'écran. L'échange, appelé *buffer swap*, est généralement synchronisé avec la fréquence de rafraîchissement de l'écran pour éviter un effet de déchirement de l'image appelé *tearing*. Cela arrive quand le *buffer swap* intervient pendant l'intervalle de rafraîchissement de l'écran, ce qui fait que les parties supérieure et inférieure de l'écran affichent deux images différentes.

Le *frame buffer* est un objet qui contient les *buffers* d'un système de rendu. Il est possible d'avoir plusieurs *frame buffers* dans un programme comme nous le verrons dans la partie 3.1.1.

2.1.3 La programmation GPU

Pour pouvoir contrôler l'affichage des éléments graphiques à l'écran, il faut donc utiliser des langages de programmation qui vont communiquer avec le GPU.

La partie programmable des *shaders* peut être écrite avec des langages de *shading* tels que le HLSL pour les applications fondées sur DirectX ou le GLSL pour celles fondées sur OpenGL. J'utiliserai dans mes recherches le GLSL qui a l'avantage d'être multi-plateforme. Il faut également noter que GLSL utilise des matrices *column-major* et HLSL des matrices *row-major*. Une matrice en GLSL devra être transposée pour être identique en HLSL, et il faut donc être vigilant lors des conversions de l'un à l'autre.

Les langages de *shading* sont divisés en plusieurs parties qui vont agir sur les étapes programmables du pipeline graphique vues précédemment. Pour rendre une scène 3D, il faut au moins utiliser un *vertex* et un *fragment shader*, alors que pour une scène 2D il est possible d'utiliser seulement la partie fragment. Le *vertex shader* va agir sur chaque *vertex* présent dans la scène et regroupe les étapes *model&view transform*, *vertex shading* et *projection*. Au cours du *fragment shader*, parfois appelé *pixel shader*, intervient l'étape de *pixel shading*, c'est le moment où s'effectuent les opérations par pixel. Par la suite, nous utiliserons également des *geometry shaders*, processus qui s'exécutent entre le *vertex* et le *fragment shader* et qui permettent de créer ou de détruire des primitives à la volée. Il existe également des *tesselation shaders* qui se déroulent juste après le *vertex shading* et qui permettent de subdiviser la géométrie d'un objet.

Il est également important de noter que chaque tâche est envoyée au premier cœur disponible sur le GPU et cela implique certaines contraintes. Un cœur n'a pas accès à ce qu'un autre est en train de calculer et il n'a également pas de souvenir de la tâche précédemment exé-

cutée. La logique de programmation est de ce fait différente de celle traditionnellement utilisée pour le CPU, elle peut être un peu déroutante (Gonzalez Vivo, s. d.).

Décomposition simplifiée d'un shader

Il existe différentes versions de GLSL qui accompagnent l'évolution de OpenGL. La syntaxe n'est pas exactement la même d'une version à l'autre. Je me baserai sur la version 120 de GLSL dans l'explication qui va suivre sur le fonctionnement de la programmation de *shader*. C'est une version qui est assez ancienne mais qui a l'avantage de fonctionner avec la quasi-totalité des cartes graphiques présentes sur le marché.

Le but des langages de programmations de *shader* est donc de communiquer avec la carte graphique pour effectuer des opérations de calcul qui, dans notre cas, vont servir à déterminer la couleur des pixels affichés dans le *viewport*.

Les langages de programmation de *shaders* possèdent certaines spécificités. Ils ont leurs propres variables, fonctions et types pré-définis tels que vec4, lerp, mat4 ou sampler2D. Ce sont des langages proches du langage C et qui possèdent des macro-définitions tels que #define, #ifdef ou #endif. Pour contrôler précisément la balance entre précision et performance, il est possible de spécifier un niveau de précision (*lowp*, *mediump* ou *highp*) pour nos valeurs de *float* en début de programme (ex. : *precision mediump float*). Le GPU possède des fonctions mathématiques telles que sin(x), cos(x), exp(x), clamp(x) ou sqrt(x). La donnée essentielle à retenir en la matière est que les valeurs utilisées sont normalisées, c'est-à-dire qu'elles sont comprises entre 0.0 et 1.0.

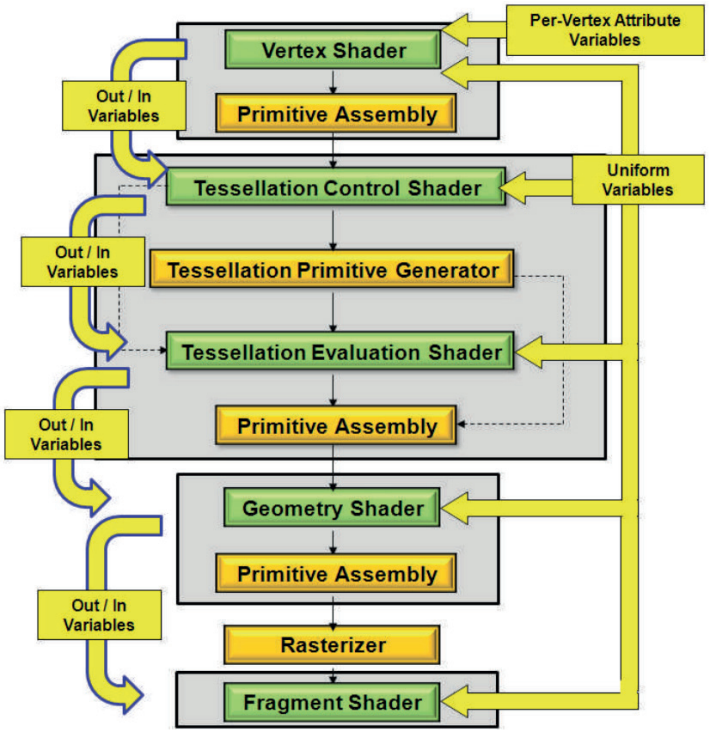


Illustration n°45 : Bailey & Cunningham, GLSL : Aperçu du pipeline (2016), Graphic Shaders

Les *shaders* possèdent également des variables qui leur sont spécifiques. En premier lieu, les *Uniform* sont des données envoyées du CPU vers tous les *threads* du GPU ; elles ne sont accessibles qu'en lecture. Chaque pixel aura accès à cette même donnée, par exemple une texture ou une variable contenant la résolution du *viewport*. Viennent ensuite les *Varying* qui sont des données envoyées du *vertex* au *fragment shader* et qui sont donc différentes pour chaque *thread*. Il existe d'autres types de variables spécifiques telles que *in* et *out* qui remplacent *varying* dans les versions plus récentes de GLSL ou encore *attribute* qui permet de récupérer les données des *vertices* dans le *vertex shader.*

Dans la première partie des expérimentations, je me servirai seulement d'une image en entrée et je n'aurai donc généralement pas d'information 3D en temps réel. Ma seule préoccupation sera la partie *fragment* du *shader.*

Il existe une autre information d'entrée que le *fragment shader* est susceptible de recevoir : il s'agit des coordonnées à l'écran du fragment sur lequel le *thread* est en train de travailler et qui sont contenues dans une variable appelée *gl_FragCoord*. Cette valeur sera donc différente pour chaque *thread* et indiquera la position dans le *viewport* du fragment en cours de calcul. Pour des scènes autres que 3D, cette donnée est utilisée de préférence à celles relatives à la position des *vertices*.

2.2 Exploration de techniques en imagerie 2D

2.2.1 Les filtres d'effets

Le but de ces premières expérimentations est de modifier une image en temps réel en appliquant des effets de post traitement à l'image originelle pour simuler un effet de peinture.

Le traitement d'image consiste en une opération mathématique qui va transformer la valeur des pixels en une nouvelle valeur. Le principal objectif est de changer l'aspect visuel d'une image. Le but recherché peut être de la simplifier pour la rendre plus facilement lisible, comme cela est fait en imagerie médicale. La technique la plus utilisée pour créer des filtres visant à créer des effets de flou ou à détecter des contours est appelée traitement par convolution. Cette méthode est parfaitement adaptée à l'implémentation sur architecture parallèle et conserve généralement une cohérence temporelle quand elle est appliquée à des animations. Elle est généralement légère en termes de ressources de calcul. Il est en effet important que l'opération permettant la mise en œuvre du filtre de peinture consomme le moins possible de ressources car la plupart des ressources de l'ordinateur sont déjà mobilisées pour exécuter le programme de génération de l'image.

Traitement d'image par convolution

Le principe de la convolution est d'appliquer une matrice de convolution, également appelée kernel, à une matrice de pixels représentant un pixel et ses voisins proches. Le résultat est la somme du produit des indices correspondants (Vasiliauskas, 2010).

Soit un kernel de taille 3x3. La convolution sur un pixel « e » pour le transformer en un pixel « e' » donnera le résultat suivant :

kernel = [1 2 3
4 5 6
7 8 9]

pixel = [a b c
d e f
g h i]

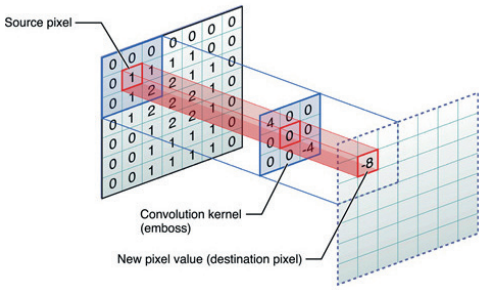


Illustration n°46 : Apple, Kernel convolution,

e' = a*1+b*2+c*3+d*4+e*5+f*6+g*7+h*8+i*9

Le nouveau pixel « e' » n'écrase pas la valeur de l'ancien pixel « e » mais est écrit dans une nouvelle image afin que les valeurs de l'image d'entrée soient toujours utilisées (et non pas celles de l'image modifiée lors de l'opération).

Lors de mes expérimentations, le code sera écrit en GLSL dans Touchdesigner 088.

C'est un logiciel permettant de réaliser du *compositing* en temps réel et d'avoir donc un *workflow* rapide pour réaliser des tests liés au traitement d'image temps réel.

Voici un premier exemple qui permet de montrer la structure du code et les résultats qui peuvent être obtenus. Dans cet exemple, j'applique à l'aide d'un kernel un effet de relief sur une image.

```
out vec4 fragColor;

uniform float effectPower;

#define emboss float[] (2.,0.,0.,0.,-1.,0.,0.,0.,-1.)

vec2 nearPixel(float x,float y){
  // vec4 uTD2DInfos[0].res;     contains (1.0 / width, 1.0 / height, width, height)
  return vec2(x*uTD2DInfos[0].res.x,y*uTD2DInfos[0].res.y);
}
void main()
{
  //init debug color
  vec3 color = vec3(1.0,0.0,1.0);
  //create a kernel
  float[9] kernel = emboss;
  vec2[9] around =  vec2[] (nearPixel(-1.0,-1.0),nearPixel(0.0,-1.0),nearPixel(1.0,-1.0),
                           nearPixel(-1.0, 0.0),nearPixel(0.0, 0.0),nearPixel(1.0, 0.0),
                           nearPixel(-1.0, 1.0),nearPixel(0.0, 1.0),nearPixel(1.0,1.0));
  vec3 sum = vec3(0.0);
  for (int i = 0; i<around.length(); i++)
  {
      sum += texture(sTD2DInputs[0], vUV.st+around[i]).x*kernel[i]*effectPower;
  }

  if (kernel == emboss) sum += 0.5;
  color = sum;
  fragColor = vec4(color,1.0);
}
```

Illustration n°49 : Traitement d'image par convolution, Exemple de code GLSL pour un effet de convolution

Illustration n°47 : Traitement d'image par convolution, Image avant convolution (image extraite du film Starship Troopers (1998) réalisé par Paul Verhoeven)

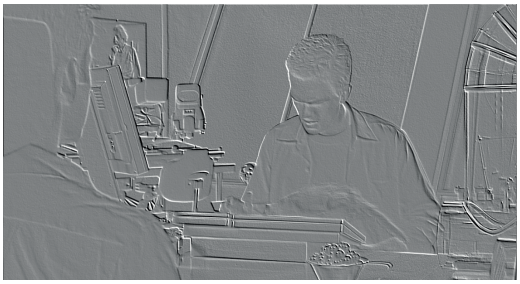


Illustration n°48 : Traitement d'image par convolution, Image après convolution produisant un effet de relief

2.2.1.1 Étirement des pixels

Pour mon premier essai, j'ai d'abord réfléchi à la manière donc je pourrais reproduire un effet de pinceau à partir d'une image en simulant des traces de couleurs. Pour cela, j'ai élaboré un algorithme qui va chercher les pixels les plus clairs pour former des traits de couleurs clairs (Illustration 51). Il est en effet possible, lors du travail sur une image, d'accéder aux pixels qui entourent le pixel à traiter. Les coordonnées du pixel à traiter sont récupérées avec la variable gl_FragCoord et il est décalé par rapport à sa position initiale afin de récupérer les valeurs d'autres pixels. En décalant légèrement certains pixels de l'image, un motif de trait est créé. La figure 51 représente la transformation de l'image 50 par le procédé décrit. Il est observé que rien ne se produit dans les zones les plus sombres, que les traces sont trop grandes et que le niveau global de détails diminue de manière inacceptable.

Illustration n°50 : Etirement des pixels, Image avant transformation (image extraite de la série tv Utopia de Dennis Kelly)

Illustration n°51 : Etirement des pixels, Image modifiée par l'effet d'étirement

2.2.1.2 Transformation d'un groupe de pixels

L'idée explorée ensuite est d'assembler les couleurs les plus claires entre elles pour donner une forme à chaque couleur, de les regrouper pour former un coup de pinceau. Au lieu de les décaler comme dans l'expérimentation précédente, je vais les assembler. Pour chaque pixel, je regarde ses huit voisins et je sélectionne la couleur de pixel la plus claire. Il ressort de l'observation de la peinture dans le monde réel que les coups de pinceaux sont moins visibles dans les zones sombres. La figure 52 nous montre le résultat de cette méthode. On observe que la forme de regroupement des pixels n'est pas assez organique et est bien trop carrée pour évoquer la peinture. Toutes les formes sont également à la même échelle et l'effet ne fonctionne pas dans les zones les plus sombres qui n'ont presque pas été modifiées. Elles paraissent plates du fait d'un manque de variation de luminosité, ce qui donne un manque d'uniformité de l'ensemble.

Illustration n°52 : Transformation d'un groupe de pixels, Image modifiée par l'effet de regroupement

J'ai ensuite testé ce même algorithme sur une autre image (cf. image 53) mais cette fois-ci en ajoutant un bruit à l'image originelle avant d'appliquer le filtre de convolution. Avec cette méthode, l'effet sur l'image (cf. image 54) est bien plus uniforme que lors de l'expérimentation précédente et le résultat est plus agréable à regarder. Certains problèmes persistent et restent à régler comme la dimension des brosses qui sont encore trop grosses et trop carrées. Il est également important de noter que généralement en peinture, la dimension des coups de pinceaux varie en fonction des éléments traités. Il nous manque ici de la variation au niveau de l'échelle et de la forme.

Illustration n°53 : Etirement des pixels, Image avant transformation (image extraite du film PK de Rajkumar Hirani)

Illustration n°54 : Etirement des pixels, Image utilisant l'effet de regroupement avec ajout de bruit et correction colorimétrique

2.2.1.3 Effet aquarelle, inattendu mais intéressant

Lors de ces premières expérimentations, j'ai également cherché à produire un effet à partir du principe suivant : définir la couleur d'un pixel en réalisant la moyenne des pixels qui l'entourent.

Le résultat 55 montre une image résultant de ce processus : on observe que, globalement, le niveau de détails est faible. L'effet évoque l'aquarelle plus que la peinture à l'huile mais le résultat est néanmoins intéressant pour la façon dont les formes et couleurs se mélangent.

Illustration n°55 : Effet aquarelle, Image de personnages avec effet aquarelle

2.2.1.4 La touche picturale, simulation du coup de pinceau

Comment représenter un coup de pinceau de manière moins géométrique ? En introduisant de l'aléatoire pour mieux simuler les variations des coups de pinceaux ? En effet, dans la réalité, il n'y a pas deux coups de pinceau semblables.

L'algorithme expérimenté ici est très simple : le pixel « e' », obtenu à partir du pixel « e », est l'un des huit pixels entourant le pixel « e » choisi de manière aléatoire. Pour orienter les touches de pinceaux, il suffit de retenir une seule ligne de pixels. Par exemple pour réaliser un coup de pinceau dont l'orientation générale serait une diagonale, je choisis un pixel situé entre celui positionné en bas à gauche et celui situé en haut à droite.

Les différents résultats 59, 60 et 61, obtenus à partir des images 56, 57 et 58 en appliquant cette technique, permettent d'observer que l'image est globalement cohérente et que l'effet de simulation du pinceau est plutôt réussi. Le résultat est efficace principalement dans les zones à fortes variations de couleurs ou de contraste, mais la technique ne fonctionne pas dans les zones de couleurs unies comme le ciel.

Illustration n°56 : La touche picturale, Image01 avant modification

Illustration n°57 : La touche picturale, Image02 avant modification

Illustration n°58 : La touche picturale, Image03 avant modification

Illustration n°59 : La touche picturale, Image01 avec filtre de coup de pinceau

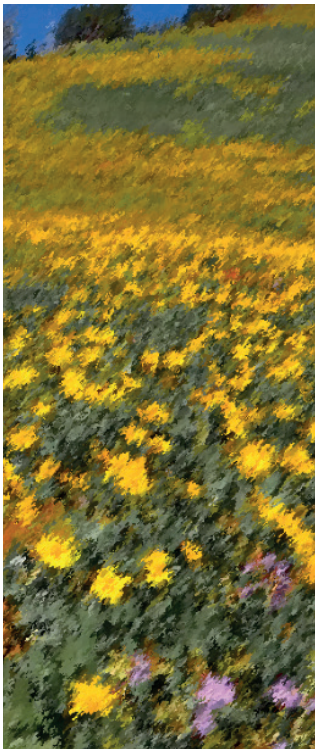


Illustration n°60 : La touche picturale, Image02 avec filtre de coup de pinceau

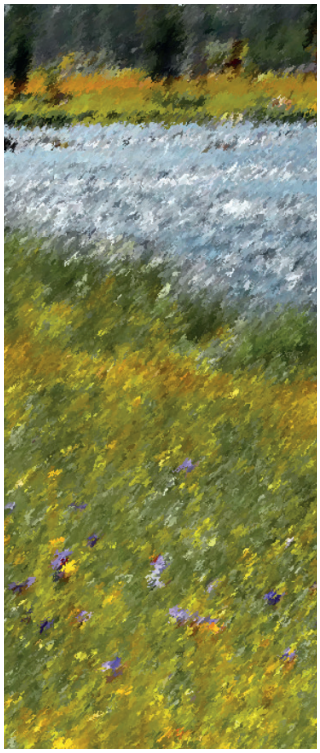


Illustration n°61 : La touche picturale, Image03 avec filtre de coup de pinceau

2.2.1.5 Quelques pistes à explorer en 3D

Un simple regard sur les œuvres réelles réalisées à la peinture permet de comprendre que les peintres appliquent différemment leurs touches de peinture en fonction des objets, de la couleur retenue mais aussi selon la profondeur et l'intensité lumineuse recherchées.

Il ressort de l'observation de différents résultats obtenus en peinture réelle que la composition de la scène, le choix des couleurs et des lumières sont déterminants : il est bien évident que la seule reproduction numérique du coup pinceau ne suffit pas pour transformer une image en peinture, il faut penser composition et couleurs, selon le même processus créatif qu'avec les outils traditionnels.

A partir d'une scène 3D, il est possible de rendre plusieurs images contenant des informations sur la couleur et la profondeur et d'y appliquer un filtre. Cela permet d'appliquer un traitement de l'image différent selon la profondeur pour obtenir plus de variation dans le trait, en fonction des normales ou des *vertex colors* ou encore avec une texture d'orientation. Il serait également intéressant de réaliser des tests en découpant les différents éléments de la scène pour y appliquer des variations de cet effet.

2.2.2 Le transfert de style

Pour réaliser un transfert de style d'une image à une autre, il existe aujourd'hui deux grandes méthodes : *neural networks* et *example based rendering*.

2.2.2.1 Neural networks

Cette première méthode consiste à utiliser des réseaux de neurones. A partir de deux images d'entrées, une image A représentant le contenu voulu et une image B représentant le style voulu est générée une nouvelle image C qui aura la composition de l'image A avec le style de l'image B (exemple : https://arxiv.org/pdf/1508.06576v2.pdf?.).

Mon niveau de connaissances en réseau de neurones est actuellement insuffisant pour me permettre d'en créer un moi-même. J'ai donc utilisé les ressources d'une plate-forme en ligne qui permet d'y parvenir. J'ai soumis deux images, A et B, au style de la peinture « *L'Angélus* » de Millet. Visuellement, le résultat n'est pas convaincant. Beaucoup d'informations sont perdues, certaines formes disparaissent et les couleurs sont mal réparties. Le processus est également trop long pour du temps réel et je manque d'outils pour développer cet axe.

Malgré tout, la recherche dans ce domaine est récente et les résultats commencent à être intéressants. L'aspect visuel est bien mieux conservé dans les travaux les plus récents. Toutefois, les temps de calcul restent trop lents pour envisager de les appliquer à un programme s'exécutant à au moins 30 FPS.

Illustration n°62 : Transfert de style - Neural Networks, Deep Art io : paysage dans le style de l'Angelus de Millet

Illustration n°63 : Transfert de style - Neural Networks, Deep Art io : personnage dans le style de l'Angelus de Millet

2.2.2.2 Example based rendering

La deuxième méthode est fondée sur l'analogie entre deux images et une autre. Le principe de base est de prendre une image A et de la comparer à une image B.

La plus ancienne technique fondée sur ce principe est le système de *litsphere* qui sera détaillée dans la partie 2.3.2 consacrée à l'analyse des normales d'une géométrie en *camera space* pour les comparer à une autre image représentant une sphère (l'objectif est de transférer la couleur de la sphère à l'objet en fonction de l'orientation de la normale).

Les méthodes plus récentes utilisent comme élément de comparaison la couleur et non plus les normales. Par exemple, partons d'une image A et d'une image B. L'image A est réalisée en peinture pour donner une image A'. Puis, en comparant A et B, une image B' de même style que A' est générée. Cette méthode peut être utilisée pour coloriser une image en noir et blanc. Soient une image A et une image B toutes les deux en noir et blanc. L'image A est colorisée pour obtenir une image colorée A'. Puis par analogie d'images on va pouvoir générer par calcul une image B' qui sera colorée dans le même style que A'.

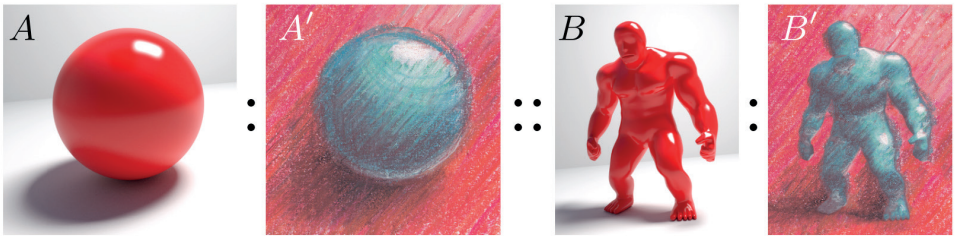


Illustration n°64 : Pavla Sykorova, Le concept d'Image analogies (2016), StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings

Les dernières méthodes publiées sur ce principe d'analogie utilisent, en plus de la couleur des images, des informations d'éclairage appelées LPE pour *lights path expressions* (pour des informations détaillées, voir la dernière publication sur le sujet : « *StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings* » (Fišer et al., 2016). Le principal problème posé par cette méthode est que, pour avoir des résultats corrects, il est impératif de rendre l'image A et l'image B avec le même matériau et avec des conditions d'éclairage

similaires. Il faut idéalement disposer d'un moteur qui gère l'illumination globale pour avoir suffisamment d'informations de lumière ; c'est assez récent dans le domaine du temps réel et relativement gourmand en ressources. Les auteurs de cette publication ont développé un logiciel qui permet d'utiliser leur algorithme. On obtient, sur un ordinateur puissant (gtx970, i7), une image toutes les cinq secondes pour une résolution de 1200*912. J'ai réalisé plusieurs tests et les résultats sont très satisfaisants mais pas adaptés pour du temps réel.

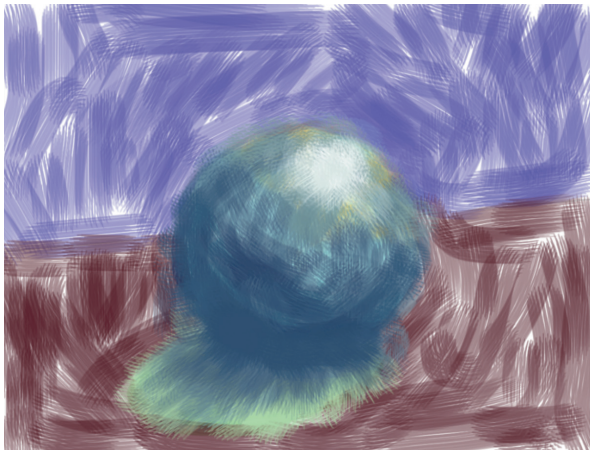


Illustration n°65 : Transfert de style - Example based rendering, Stylit : Image A créée à la main

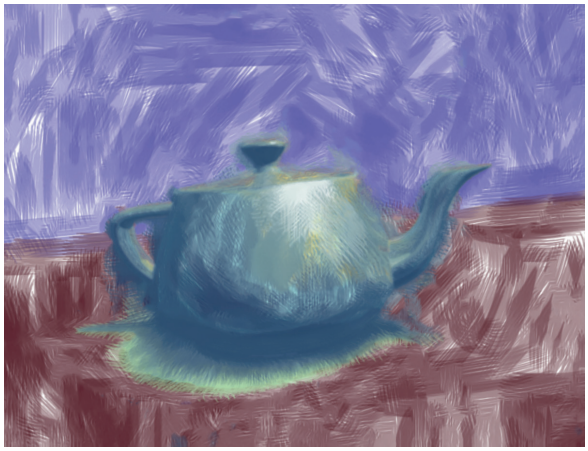


Illustration n°66 : Transfert de style - Example based rendering, Stylit : Image créée à partir du style de l'image A

2.2.2.3 Analogies temps réel

A la suite de ces expérimentations, j'ai tenté d'implémenter un algorithme de comparaison très simple fondé sur la couleur, en GLSL sur Touchdesigner 088, pour observer les performances de ce système en temps réel. Après quelques tests, il est apparu que les calculs étaient trop lents pour du temps réel. Même en programmation de *shader*, il est nécessaire de parcourir l'image, ce qui consomme beaucoup de ressources de calcul. Comme image de référence, j'ai utilisé une simple image de rendu au crayon rouge.

Pour avoir un résultat en temps réel, j'ai procéder par approximation pour le parcours d'image. Le résultat qui suit est supérieur ou égal à soixante images par seconde : il n'est pas précis et des bandes sombres apparaissent là où il ne devrait pas y en avoir.

En montant le niveau de précision, il est possible d'obtenir un résultat entre 20 et 60 FPS, ce qui est encore loin de l'image de référence. De plus, les informations de lumières du moteur de rendu de Touchdesigner 088 sont bien trop basiques, des lignes de couleurs similaires apparaissent le long de la sphère. Ce phénomène est dû aux trop faibles variations d'informations de lumière et à l'absence d'illumination globale. De plus, j'ai eu recours, pour améliorer les performances, à une technique d'analyse pixel par pixel plutôt que par patch pour comparer les différentes images, ce qui n'est vraiment pas précis.

Malgré les progrès réalisés dans le domaine du transfert de style, je n'envisage pas d'utiliser cette technique pour réaliser d'autres expérimentations. Tout d'abord pour la lenteur

des calculs nécessaires et ensuite pour la difficulté de mise en place d'une solution efficace visuellement en l'état des techniques actuelles.

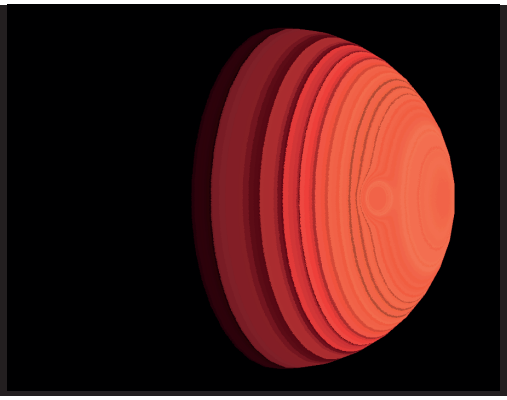


Illustration n°67 : Transfert de style - Analogies temps réel, Modèle approximatif

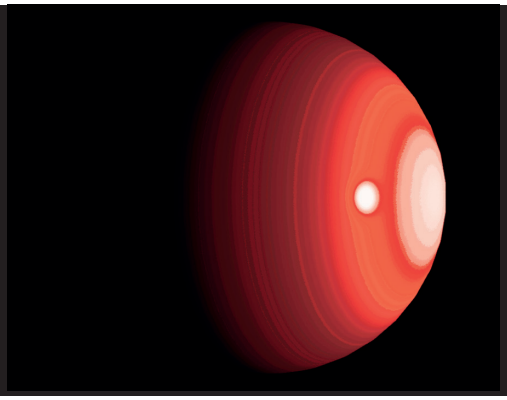


Illustration n°68 : Transfert de style - Analogies temps réel, Modèle plus précis

2.2.3 Le dessin par analyse du mouvement

2.2.3.1 Optical flow

L'idée qui sous-tend cet axe de recherche est la même que pour les filtres : il s'agit de simuler le geste du coup de pinceau en produisant une trace de couleur. Mais au-delà, j'ai voulu analyser les mouvements présents dans la scène pour que ces mouvements laissent une trace.

Pour commencer, j'ai réalisé un *optical flow* en GLSL. Cette technique permet de visualiser un mouvement entre deux images en comparant l'image actuelle avec une image précédente et d'analyser la manière dont ses pixels ont évolué. On peut voir un exemple dans ces deux images :

Illustration n°70 : Le dessin par analyse du mouvement - Optical flow, image A de base tirée d'une vidéo

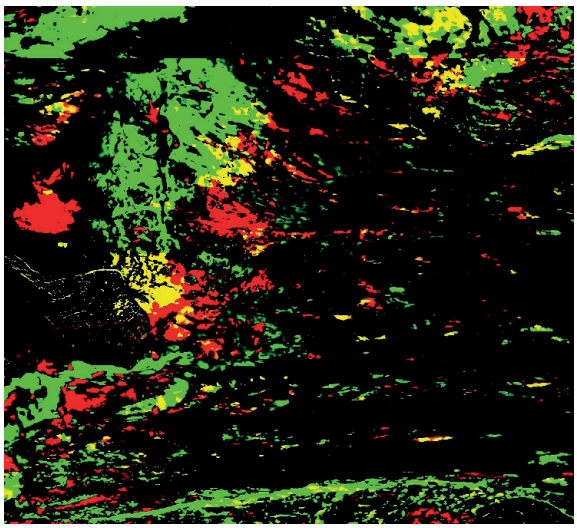


Illustration n°69 : Le dessin par analyse du mouvement - Optical flow, optical flow créé à partir de l'image A

2.2.3.2 Création d'images fixes

Ensuite, en me fondant sur l'*optical flow*, j'ai tenté de « peindre du mouvement » avec les pixels les plus lumineux. Quand un élément bouge dans l'image A, il s'enregistre dans une nouvelle image B si le pixel correspondant de l'image B est plus sombre que celui de A.

Le résultat est assez satisfaisant, mais lorsque j'ai réalisé le test avec une vidéo qui boucle, il arrive un moment où l'image n'a logiquement plus rien à dessiner. Un autre problème a été observé : la couleur est absente dans les zones sans mouvement apparent, comme par exemple dans le bleu du ciel ou dans les zones d'ombre très sombres.

Illustration n°71 : Le dessin par analyse du mouvement - Création d'image fixe, Résultat basé sur une vidéo qui boucle

Il est possible de gérer le niveau d'information à traiter en modifiant le seuil de captation de mouvement de l'*optical flow*.

Cette technique ne fonctionne correctement qu'avec des plans fixes et des éléments dont les mouvements bouclent. Cette image met en évidence le résultat obtenu à partir d'un film où les plans s'enchaînent : on obtient bien un effet de peinture mais la lisibilité de l'image est perdue, on bascule dans l'abstraction.

Illustration n°72 : Le dessin par analyse du mouvement - Création d'image fixe, Résultat basé sur une vidéo avec changement de plan 01

Illustration n°73 : Le dessin par analyse du mouvement - Création d'image fixe, Résultat basé sur une vidéo avec changement de plan 02

2.2.3.3 Création d'images animées

Après ces expérimentations visant à produire une image fixe, je me suis attaché à expérimenter des méthodes permettant de produire une image animée. Pour ce faire, j'ai réutilisé la méthode précédente mais en réduisant la luminosité à chaque nouveau calcul d'image pour, au fil du temps, enlever de la matière, ce qui permet d'avoir constamment un effet d'ajout de peinture. J'ai également ajouté un bruit sur toute l'image pour créer plus de mouvements et de variations. Et avant que les images ne soient traitées, j'ai ajouté le filtre de peinture élaboré lors des expérimentations relatives aux filtres d'images.

Illustration n°74 : Le dessin par analyse du mouvement - Création d'images animées

Illustration n°75 : Le dessin par analyse du mouvement - Création d'images animées

Les résultats sont suffisamment convaincants pour poursuivre les recherches dans cette voie. Il conviendrait d'élaborer d'autres tests comportant des éléments plus évolutifs afin de conserver plus de contrôle sur la forme et les couleurs. Et des tests sur des images générées par ordinateur plutôt que sur des vidéos compléteraient utilement la recherche.

2.3 Exploration de techniques en imagerie 3D

2.3.1 Instanciation sur un modèle 3D

Le projet *Rendering Painting World* (Rahteenko, 2016) a été une source d'inspiration. Des quadrilatères ayant une texture de pinceaux sont placés dans la scène par rapport à une image contenant des informations de profondeur. Chaque quadrilatère représente un pixel d'une image et possède une information de position et de couleur. Ensuite une texture qui va définir l'orientation des coups de pinceaux est réalisée à la main. Le résultat est très satisfaisant mais plusieurs problèmes se présentent. Premièrement l'artiste doit créer à la main une image représentant l'orientation des différentes taches de peintures, ce qui prend beaucoup de temps. De plus, les éléments de l'image ne sont pas évolutifs. En effet, la position des différents quadrilatères est définie au début du programme et n'est pas recalculée ensuite compte tenu du coût de l'opération en ressources. Il est possible d'animer la scène à l'aide d'un *vertex shader* mais de manière très limitée.

La première publication sur le sujet est « *Painterly rendering for animation* » (Meier, 1996). Le principe est d'instancier des géométries sur un modèle 3D pour simuler des coups de pinceaux.

A cette fin, des particules sont générées sur un objet par rapport à ses normales. On leur applique ensuite une texture de la forme d'un coup de pinceau. La dimension des coups de pinceaux varie en fonction d'une texture d'occlusion ambiante : plus elle est foncée plus la dimension de l'image représentant la particule sera petit.

J'ai appliqué un algorithme similaire pour évaluer son efficacité dans des projets en temps réel. L'image 76 montre le modèle de base et l'image 77 le montre dans un rendu fil de fer, afin d'illustrer la structure générale du modèle 3D.

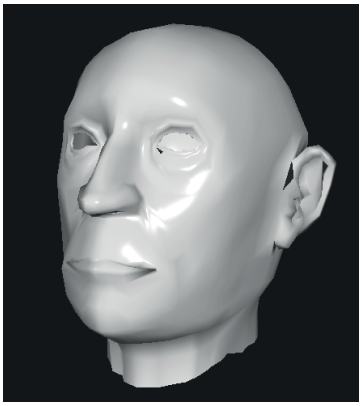


Illustration n°76 : Instanciation sur un modèle 3D, Modèle 3D de base

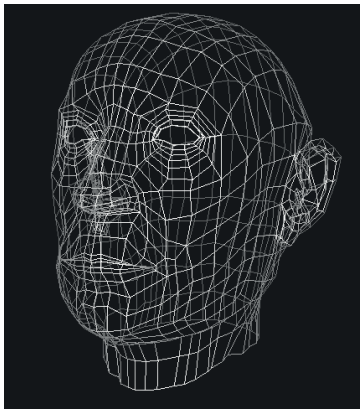


Illustration n°77 : Instanciation sur un modèle 3D - Wireframe, Modèle 3D de base

L'image 78 montre un premier résultat où des géométries, en forme de traces de pinceaux, sont instanciées au niveau de chaque *vertex*. L'avantage d'une technique comme celle-ci est que, dès lors que les instances réagissent à la lumière, on obtient un modèle d'illumination correct. Le résultat pose toutefois problème : les instances ne sont pas réparties de manière uniforme sur toute la surface et des trous se forment là où il y a très peu voire pas du tout de *vertices*.

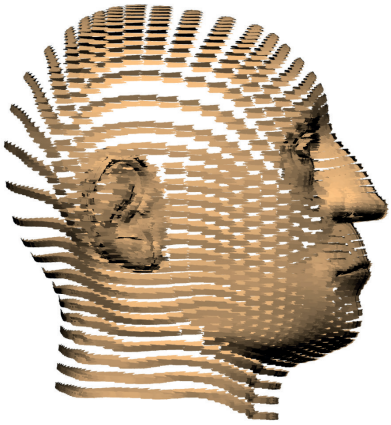


Illustration n°78 : Instanciation sur un modèle 3D, Instances de petite taille

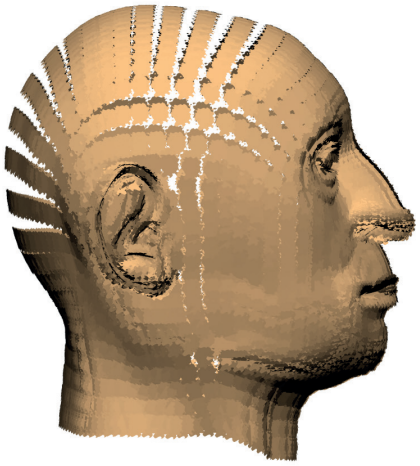


Illustration n°79 : Instanciation sur un modèle 3D, Instances de grande taille

Pour pallier cet inconvénient, j'ai agrandi la taille des instances pour essayer de boucher les trous mais le résultat n'est pas concluant dans les zones où, justement, le maillage est trop dense et les géométries débordent (voir image 79). Une solution pourrait être de faire varier la taille des instances en fonction de leurs distances à d'autres, par rapport à des *vertex colors*.

J'ai ensuite tenté de contourner la difficulté en générant de nouveaux points de manière plus aléatoire. Pour cela deux nouveaux points sont générés par face et les *vertices* de bases ne sont pas pris en considération. Force est de constater que les points sont quand même plus nombreux dans les zones où il y avait plus de faces, le problème n'est donc pas réglé.

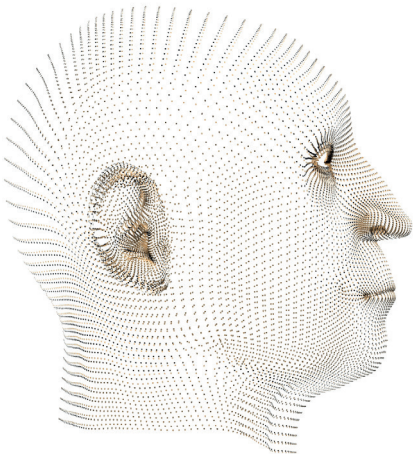


Illustration n°80 : Instanciation sur un modèle 3D, Instances pour un style de dessin

L'idée de tirer parti de la répartition non uniforme des particules a été explorée sans résultat probant pour le rendu de peinture. En revanche, j'ai obtenu des résultats intéressants pour du rendu dans un style de dessin (voir illustration 80). Généralement dans un modèle 3D, les endroits qui possèdent le plus de *vertices* correspondent aux endroits où figurent le plus de détails, ce qui correspond à l'approche du dessin au trait où l'on va ajouter des coups de crayons dans les zones comportant le plus d'informations.

2.3.2 Utilisation de Matcap / Lit sphere

2.3.2.1 Matcap et filtres d'effets

Le procédé appelé *matcap*, ou encore *lit sphere*, est décrit dans « *The Lit Sphere: A Model for Capturing NPR Shading from Art* » (Sloan et al., 2001) L'observation d'une sphère nous révèle toutes les orientations de normales qui font face à la caméra. Donc en analysant un modèle 3D, il est possible d'associer chacune de ses normales à une normale de la sphère. Il suffit alors de dessiner une sphère dans le style de rendu qui nous intéresse, puis de regarder l'orientation des normales d'un modèle 3D en *eye space* et de trouver la normale correspondante dans l'image de la sphère dessinée. Enfin, la couleur correspondante de la sphère est appliquée sur le modèle.

Ce système donne de bons résultats quand il n'y a pas de mouvement de caméra ni de mouvement de lumière, pour une présentation de produit par exemple. C'est la technique généralement employée dans le *viewport* des logiciels de *sculpting* tels que Zbrush.

Illustration n°81 : Utilisation de Matcap / Lit sphere - Matcap et filtres d'effets, Matcap

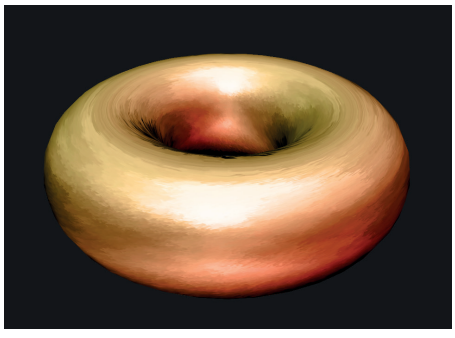


Illustration n°82 : Utilisation de Matcap / Lit sphere - Matcap et filtres d'effets, Application d'un matcap à un torus

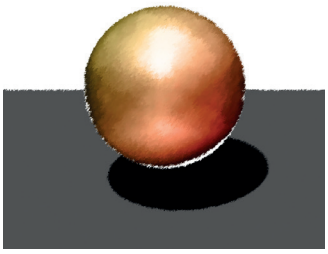


Illustration n°83 : Utilisation de Matcap / Lit sphere - Matcap et filtres d'effets, Problème au niveau des contours lié à la couche alpha du matcap

Le modèle de base n'intègre pas de système de simulation de relief, ce qui donne aux géométries un aspect très lisse, sauf à avoir un maillage très dense et à sculpter le relief, solution peu adaptée pour du temps réel. Il est néanmoins possible d'ajouter un effet de *normal mapping* pour donner plus de consistance au matériau.

Bien qu'il existe des méthodes plus récentes qui se révèlent plus efficaces (Fišer et al., 2016), le *lit sphere shading* présente quelque avantage : il est extrêmement simple à implémenter et demande un temps de calcul très court.

J'ai réalisé des tests en mélangeant cette méthode avec celle de l'expérimentation précédente. L'image 81 représente l'image de base qui sera ensuite utilisée par le matcap. L'image 82 montre un *torus* avec un *shader matcap* fondé sur l'image 81, sans retenir la couche alpha. En appliquant ensuite le *matcap* avec de l'*alpha*, comme dans l'image 83, les images présentent un problème au niveau des contours. Il est certainement plus intéressant de réaliser une texture *matcap* traditionnelle et d'ajouter ensuite un filtre en post traitement pour éviter ces problèmes de transparence.

Illustration n°84 : Utilisation de Matcap / Lit sphere - Matcap et filtres d'effets, Matcap et filtre sur un modèle détaillé

Le résultat n'est pas convaincant non plus à partir de formes très détaillées comme le montre l'image 84 : trop de détails sont perdus pour que la forme finale soit reconnaissable. Mais ici la perte de visibilité provient certainement d'un effet trop accentué. Il conviendrait de rechercher plus de subtilité dans le coup de pinceau et de trouver une solution au problème de limitation des coups de pinceaux au niveau des contours.

2.3.2.2 Ajout du noise et matcap interactif

Afin de mieux identifier les modifications ultérieures, l'image 85 représente la géométrie avec application d'un simple *shader matcap.*

Illustration n°85 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Modèle de base avec un matcap

Illustration n°86 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Modèle avec un noise animé par vertex (image issue d'une animation se déroulant à 12 images par seconde)

Illustration n°87 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Modèle avec un noise animé et un effet de filtre

Illustration n°88 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Noise sur modèle avec maillage plus dense

Illustration n°89 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Modèle ave unc noise seulement sur les bords

Illustration n°90 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Modèle avec un noise et un autre matcap

Après l'application d'un *matcap shading*, j'ai ajouté un bruit sur la géométrie afin d'augmenter les variations au niveau de la forme. Ce bruit va donner un aspect plus naturel à l'image, comme si elle avait été réalisée à la main. Le résultat est satisfaisant tant que le nombre d'images par seconde reste peu élevé comme le montre l'image 86 qui est un extrait d'une animation s'exécutant à douze images par seconde.

J'ai ensuite ajouté l'effet précédemment utilisé pour obtenir l'image 87. On observe un ajout d'information au niveau de la texture mais le niveau de distorsion est ici assez élevé, ce qui peut provoquer de la fatigue visuelle.

L'image 88 est fondée sur le même principe que la précédente mais le maillage est ici plus dense, ce qui accentue le bruit. J'ai également réduit l'effet de convolution.

Pour l'image 89, je me suis arrangé pour que le bruit n'affecte que les bords du modèle afin de réduire le risque de fatigue visuelle. L'effet n'est pas très probant et doit sans doute être adapté en fonction de la nature du projet : il faut évaluer le bon ratio entre qualité d'image, effet de peinture et risque de fatigue visuelle. Les résultats avec une image animée me paraissent très corrects. Comme avec les filtres, le *matcap* est un effet en *eye scape* et possède donc une bonne cohérence temporelle. L'image 90 est l'équivalent de l'image 88 mais avec un *matcap* différent.

Mais comment rendre les *shaders matcap* interactifs ? La solution la plus simple est d'effectuer une rotation de l'image de base. Cette solution qui n'est envisageable que pour certains types de *matcap* présente l'avantage de suggérer l'idée d'une lumière distante qui effectue une rotation.

Une autre problématique se pose pour le traitement des ombres portées qui ne sont pas prises en compte dans le *shader*. Il est toutefois possible d'effectuer des opérations de filtres d'effets sur des *shadow maps* par exemple.

De plus, en gardant la technique de *shading* avec *matcap*, j'ai réalisé des textures animées. L'idée était d'obtenir un effet fluide évoquant l'aspect de la peinture. Je me suis donc servi d'un bruit fondé sur un *fractal brownian motion* pour déformer les coordonnées de texture de la *lit sphere* (voir résultat sur l'image 91). Ensuite j'ai appliqué celle-ci sur le modèle précédent pour obtenir l'image 92. L'image 93 montre le résultat du même procédé avec une autre texture. Le résultat est intéressant mais la perte de contrôle est trop importante. Piste toutefois intéressante à explorer de manière plus approfondie.

Illustration n°91 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Fractal brownian motion appliqué à un matcap

Illustration n°92 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Modèle avec un matcap affecté par un fractal brownian motion 01

Illustration n°93 : Utilisation de Matcap / Lit sphere - Ajout du noise et matcap interactif, Modèle avec un matcap affecté par un fractal brownian motion 02

2.3.3 Instanciation de brushes

L'expérimentation suivante s'inspire des recherches dans le domaine du *Stroke based Rendering*, c'est à dire du placement de *brushes* sur une image dans un espace 3D pour simuler la manière dont un peintre placerait des coups de pinceaux sur une toile.

Ici les instances ne sont pas générées à partir d'un modèle 3D mais à partir d'une image. Pour cela, il faut idéalement quatre informations : les couleurs RGB et la profondeur. A partir de ces informations, on va pouvoir placer des instances dans l'espace pour simuler des coups de pinceaux. La dimension des instances peut évoluer en fonction de la profondeur. L'orientation peut également varier pour donner plus de dynamisme à l'image.

La résolution de l'image détermine le nombre de particules possible, qui définit la précision du résultat final, mais également le temps de calcul. L'image 94 montre un résultat avec de petites instances pour donner une idée générale du fonctionnement.

Les images 95 et 96 illustrent des résultats avec une image de faible résolution et deux formes de coups de pinceaux différentes. On voit dans l'image 96 que le résultat est plus uniforme en utilisant un pinceau de forme ronde plutôt que rectangulaire comme dans l'image 95.

Illustration n°94 : Instanciation de brushes, Petites instances

Illustration n°95 : Instanciation de brushes, Faible résolution et pinceau de forme allongée

Illustration n°96 : Instanciation de brushes, Faible résolution et pinceau de forme ronde

Illustration n°97 : Instanciation de brushes, Haute résolution et pinceau de forme allongée

Illustration n°98 : Instanciation de brushes, Haute résolution et pinceau de forme ronde

Les images 97 et 98 présentent des résultats à partir d'une image en haute résolution avec différents pinceaux. Le temps de calcul est ici bien trop long pour du temps réel.

J'ai ensuite réalisé des tests avec des images dont je n'avais pas d'information de profondeur. L'idée était d'utiliser le *Z-fighting* pour mélanger les différents coups de pinceaux. Quand deux objets 3D se situent à la même profondeur, l'ordinateur doit choisir quel élément rendre à l'image. Le *Z-fighting* produit un artefact visuel en mélangeant des bouts des deux géométrie en une seule.

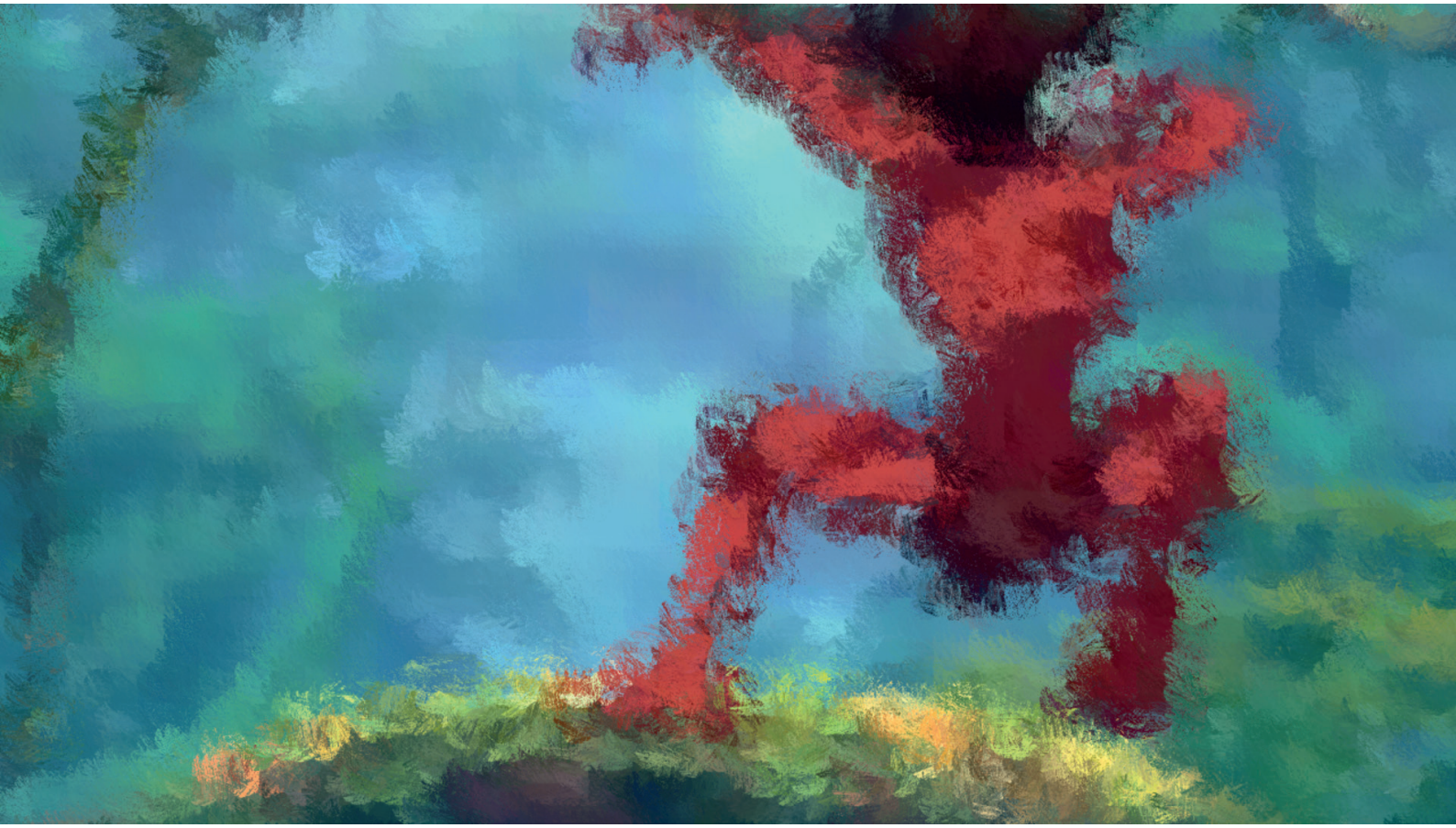


Illustration n°99 : Instanciation de brushes, Z fighting, image de base extraite du film d'animation Tarzan (1999)

Le résultat est concluant malgré la diminution importante du niveau de détails et une délimitation des formes qui devient parfois difficile à lire.

Lors de ces expérimentations, j'ai proposé diverses méthodes qui me semblait être des pistes intéressantes à explorer. Je vais maintenant mettre en pratique, dans des projets artistiques, trois méthodes que je considère comme étant les plus prometteuses. Premièrement un projet qui est une réinterprétation d'une peinture de l'école de Barbizon à l'aide des outils numériques où j'utiliserai la méthode des filtres d'images. Puis une œuvre qui va s'inspirer des expérimentations fondées sur l'*optical flow* en utilisant des agents autonomes qui produiront des traces de peinture pour dessiner des portraits. Finalement, je présenterai une technique automatique et paramétrable, qui est une amélioration de la méthode d'intanciation de *brushes*.

3. Créations artistiques personnelles

3.1 L'installation l'Aliaj Angelus

Ce projet, réalisé en binôme avec Isadora Teles de Castro e Costa sur trois semaines en janvier 2017, est intitulé l'*Aliaj Angelus*. C'est une réinterprétation de la peinture « *L'Angelus* » (1857-1859) de Jean-François Millet qui vise à créer, en utilisant des techniques numériques temps réel, un paysage évoluant avec le temps.

Illustration n°100 : Jean-François Millet, l'Angélus (1857-1859), Musée d'Orsay, Paris

Voici sommairement comment fonctionne l'installation :

✓ les conditions météorologiques de l'environnement virtuel (force du vent, quantité de nuages dans le ciel, la pluie,...) sont définies à partir de données réelles précédemment recueillies ;

✓ au sein de cet environnement virtuel apparaît un couple de paysans : ils vont venir travailler tous les jours dans un champ. Les variables environnementales que sont la météorologie ou les heures de lever et coucher du soleil influent sur leur niveau de fatigue et cette fatigue va elle-même avoir une influence sur leurs actions ;

✓ l'environnement comporte également de la végétation. Le phénotype des différents végétaux est fondé sur un algorithme génétique (chaque espèce de plante possède des gènes spécifiques). A chaque génération, les plantes (et leurs génotypes) évoluent en fonction de la météorologie. Ces gènes vont orienter des systèmes dessinateurs qui donnent leur apparence finale aux végétaux. De plus, les différentes espèces sont réparties dans l'image grâce à un système

d'automate cellulaire, d'une manière qui évolue à chaque génération ;

✓ enfin, la projection de l'image finale donne l'impression de la regarder à travers une fenêtre, phénomène accentué par le recours à un système de *head tracking*.

Pour réaliser ce projet nous avons eu recours à plusieurs logiciels. L'animation des personnages et la traduction de leurs comportements sont calculés dans Unity 5.5.0f3. Le dessin des plantes, leur positionnement et la captation des effets météorologiques sont gérés en C++ à l'aide du *framework* openFrameworks 0.9.8 et l'effet de peinture, la composition de la scène et le système de headtracking ont été réalisés dans TouchDesigner 088 built 62070.

3.1.1 Effets environnementaux

- Soleil

En premier lieu, j'ai collecté les heures de lever et de coucher du soleil, ce qui m'a permis de définir les caractéristiques du ciel que sont sa couleur, la position du soleil et l'éclairage de la scène en fonction de l'heure de la journée. Pour ce faire, j'ai défini la trajectoire du soleil dans intervalle [0.0, 1.0], 0.0 correspondant à son heure de lever et 1.0 à son heure de coucher. Ensuite, en fonction de l'heure de la journée, j'obtiens une valeur qui m'indique la position du soleil sur la course qu'il parcourt en une journée. Par exemple, si son état est à 0.5 il sera à son point culminant.

Pour pouvoir utiliser cette donnée, il m'a fallu convertir la fonction linéaire en une fonction parabolique. Cette fonction est extraite du blog de Inigo Quilez (Quilez, 2017). Les valeurs extrêmes deviennent égales à 0 et la valeur médiane, correspondant à 0.5 en abscisse, devient la valeur maximum.

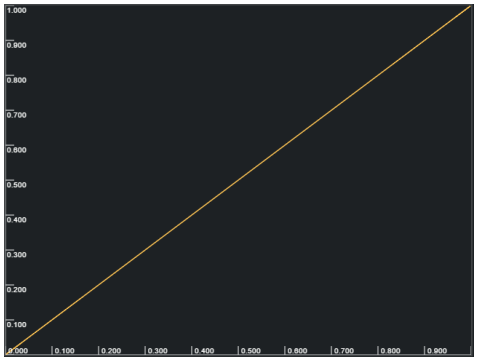


Illustration n°101 : L'installation l'Aliaj Angelus - Effets environnementaux, Trajectoire linéraire du soleil

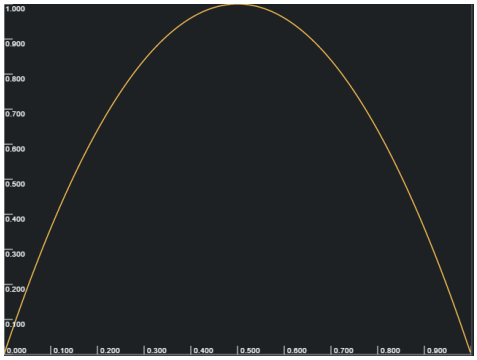


Illustration n°102 : L'installation l'Aliaj Angelus - Effets environnementaux, Trajectoire parabolique du soleil

En réglant la puissance de la parabole, il est possible de définir très précisément la trajectoire du soleil.

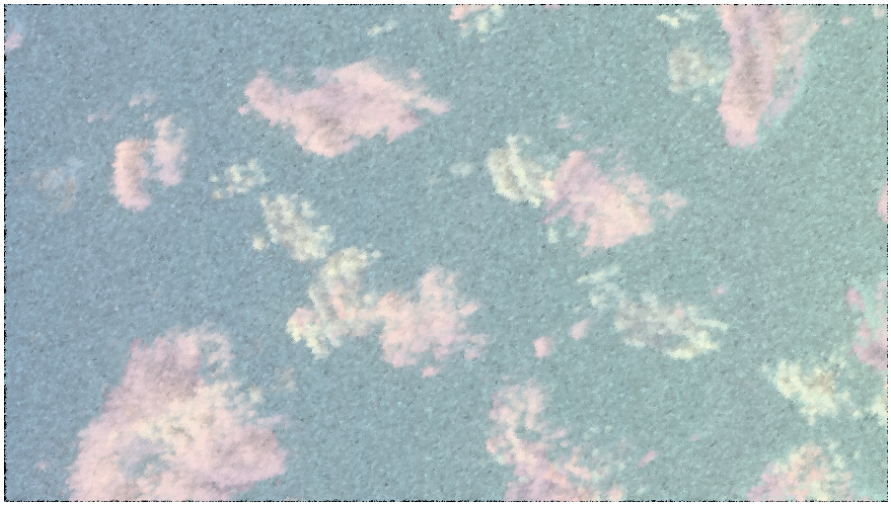


Illustration n°103 : L'installation l'Aliaj Angelus - Effets environnementaux, Peu de nuage en milieu de journée

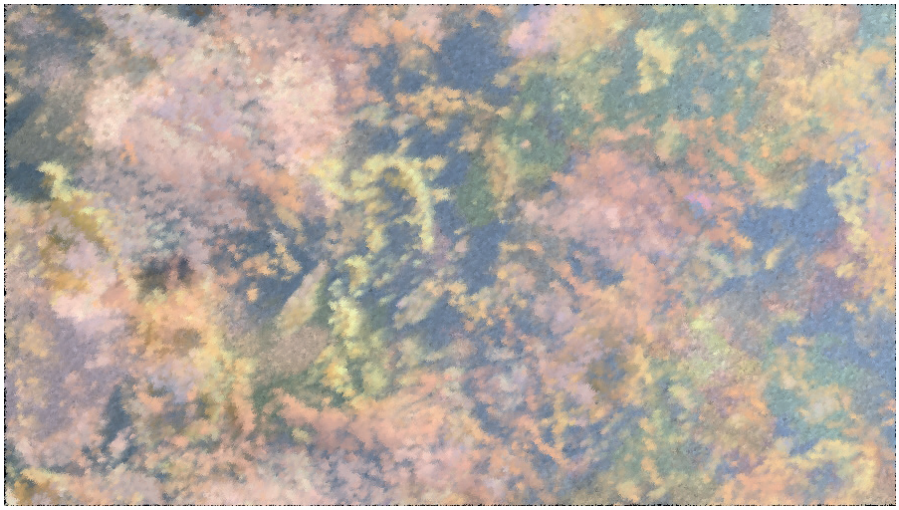


Illustration n°104 : L'installation l'Aliaj Angelus - Effets environnementaux, Beaucoup de nuage le matin

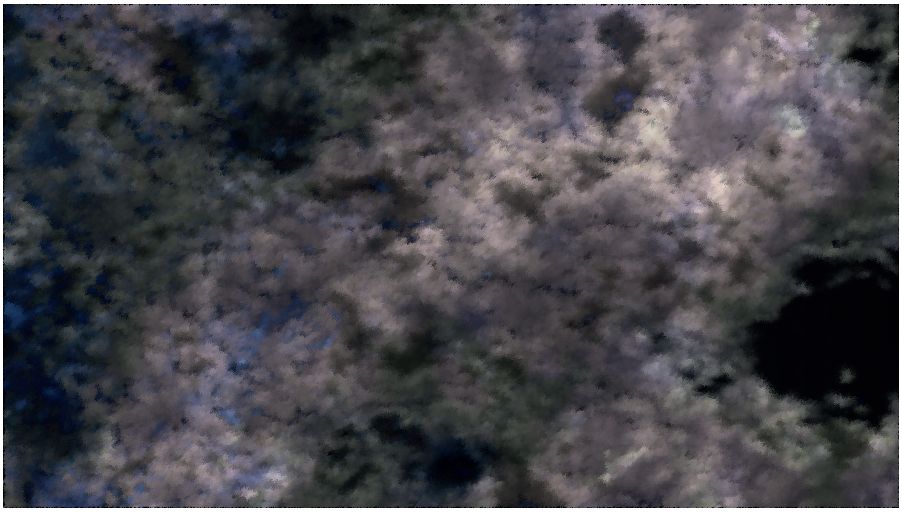


Illustration n°105 : L'installation l'Aliaj Angelus - Effets environnementaux, Beaucoup de nuage la nuit

- Nuages

Les nuages sont réalisés en GLSL par un assemblage de *noises*. Leur quantité ainsi que la couleur du ciel varient en fonction des données météorologiques, sauf la nuit où la couleur du ciel reste constante.

L'avantage des *shaders* génératifs est qu'ils offrent la possibilité de modifier en temps réel la texture affichée. Il aurait été envisageable d'utiliser une technique plus classique en mélangeant plusieurs textures de nuages fixes et en modifiant leurs couches alpha en fonction de la quantité de nuages dans le ciel. Le *shader* GLSL permet également d'avoir des formes qui ne sont pas prédéfinies et qui évoluent constamment. Mais il est plus difficile de les contrôler et de leur donner un style (forme/couleur) spécifique, contrairement à une texture fondée sur une image préalablement réalisée.

- Pluie

Il existe plusieurs méthodes couramment utilisées en infographie pour simuler de la pluie. Sébastien Lagarde explique très clairement sur son blog (Lagarde, 2012) comment, avec l'équipe du studio de production *DontNod Entertainment*, il a réalisé les effets de pluie pour le jeu *RememberMe*.

Pour ma part, j'ai simulé la pluie à l'aide d'un *geometry shader* en m'inspirant de la conférence *GDC Vault - Low Complexity, High Fidelity - INSIDE Rendering* (Gjoel & Svendsen, 2016). La pluie est donc en 3D et elle est principalement calculée par la carte graphique.

Le principe est assez simple. J'ai commencé par placer des *vertices* sur un plan placé au dessus du terrain en nombre proportionnel au niveau de pluie fourni par l'API météorologique. Chaque *vertex* va représenter la position d'une goutte de pluie. A l'aide d'un *vertex shader*, je vais déplacer chaque goutte de pluie verticalement à une vitesse comprise entre 3 et 9 unités de distance par seconde (chiffre fondé sur des mesures réelles et correspondant à une goutte de taille moyenne).

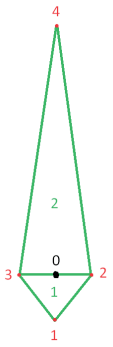


Illustration n°106 : L'installation l'Aliaj Angelus - Effets environnementaux,Géométrie d'une goutte de pluie

Puis dans le *geometry shader*, je récupère la position d'un *vertex* pour créer la géométrie de la goutte. Sur le schéma ci-dessus, le point 0 représente la position du *vertex* de base (il n'est pas utilisé pour l'affichage final). Les points en rouge 1, 2, 3 et 4 représentent l'ordre dans lequel de nouveaux *vertices* sont créés et les surfaces identifiées en vert sont les deux faces formant la goutte. L'ordre de création des *vertices* est fondamental pour le dessin des faces.

Puis pour la partie *fragment* du *shader*, c'est à dire l'affichage de la goutte, j'ai opté pour des gouttes blanches avec une valeur d'alpha assez faible, ce qui présente plusieurs avantages : l'alpha crée un effet de profondeur, ensuite plus les gouttes sont proches de la caméra plus elles sont opaques, avec une adaptation selon l'éclairage et, enfin, le *shader* est très rapide à calculer. La simulation de la pluie est susceptible d'être améliorée en rajoutant des éclaboussures sur le sol de manière aléatoire. Mais l'une des contraintes du projet est qu'il se compose principalement de plans 2D, il est donc difficile de réaliser des éclaboussures sur les objets de la scène. Pour cela, il faudrait détecter les contours par exemple.

Pour un projet de rendu photoréaliste, le niveau de détails ne serait pas suffisant mais dans le cas de l'*Aliaj Angelus*, il donne entièrement satisfaction.

Illustration n°107 : L'installation l'Aliaj Angelus - Effets environnementaux, Légère pluie la nuit

- Végétation

J'ai pris en charge la gestion du dessin de la végétation. Chaque plante devait être dessinée en dehors de l'écran puis envoyée dans un *shader*. Pour cela, j'ai utilisé des *frame buffer objects* (FBO). Ce sont des objets OpenGL qui permettent, entre autres, de dessiner sur des textures dans la carte graphique sans qu'il soit nécessaire de les afficher à l'écran. Une fois le FBO généré, sa texture est ensuite envoyée au *shader* pour générer les différentes instances de plantes qui vont ensuite être placées et dessinées dans la scène. Au total, 65 536 plantes sont présentes dans la scène.

Illustration n°108 : L'installation l'Aliaj Angelus - Effets environnementaux, Champs remplit de végétation

Chaque FBO possède ici une couche alpha qui sert à « découper » les différents végétaux. J'ai rencontré un problème de *Z-fighting*. Les plantes bougeant avec le vent, il arrivait souvent que deux plans se chevauchent. Dans ce cas de figure, comme l'alpha est pris en compte dans le *Z-buffer*, certaines plantes étaient masquées par l'alpha d'une autre plante qui la recouvrait. Pour contourner ce problème j'ai dû ignorer (avec l'instruction *discard* en GLSL, qui signifie « ne pas réaliser le calcul ») les fragments dont l'alpha étaient inférieur à 0.5. Il est en effet souvent plus efficace, pour éviter des problèmes de *Z-fighting*, de ne pas calculer du tout un fragment dont l'alpha est égal à 0.

La gestion de la transition entre les différentes générations de végétaux a également été difficile. La texture d'un FBO ne se rafraîchit pas automatiquement, ce qui permet aux végétaux de se dessiner et aux dessinateurs de laisser une trace. A chaque nouvelle génération, je nettoie complètement la texture du FBO. Lors de l'apparition d'une nouvelle génération, une nouvelle plante remplace l'ancienne. Idéalement nous aurions souhaité réaliser un fondu en réduisant la valeur de la couche alpha à chaque image calculée mais aucune solution viable n'a été trouvée dans le temps du projet. Soit il restait une trace de l'ancienne plante, soit la nouvelle s'effaçait trop vite. Avec un temps d'expérimentation moins contraint, je ne doute pas que nous aurions trouvé une solution qui aurait permis au spectateur de comparer la génération actuelle avec les précédentes et ainsi de mieux comprendre l'algorithme génétique et de voir son impact sur l'aspect de la végétation.

L'avantage de la solution finalement retenue est que l'utilisateur peut distinguer clairement le moment où une nouvelle génération fait son apparition. De plus, les plantes se dessinent suffisamment rapidement pour ne pas laisser la scène vide trop longtemps.

- Head tracking

Pour l'effet de *head tracking*, j'ai opté pour un découpage de la scène en plusieurs plans qui correspondent aux différents plans du tableau original. J'ai pour cela créé plusieurs caméras identiques à l'exception de leurs *clip planes*. Chaque *near* ou *far clip plane* isole les différentes zones. Ces informations sont également envoyées dans les *shaders* GLSL pour éviter le calcul des géomwétries non visibles dans les plans découpés. Pour transférer les images d'un logiciel à l'autre, j'ai utilisé le *framework* Spout[1] qui fonctionne avec OpenGL ou DirectX sur Windows.

Dans le projet finalisé, tout fonctionnait correctement et en temps réel. Toutefois, mes recherches sur le fonctionnement interne de Spout pourraient être utilement approfondies. En effet, j'ai eu de la latence dans la réception des images Spout lorsque je mettais un certain nombre de *shader* GLSL dans TouchDesigner alors qu'en affichant les sorties de Spout dans Visual Studio aucune latence n'était visible. Le problème ne s'explique pas par un manque de capacités de la mémoire de la carte graphique (VRAM) car, à la fin du projet, elle n'était qu'à un quart de ses potentialités. Pour les projets à venir, il conviendrait de trouver une méthode pour vérifier l'utilisation des cœurs du GPU afin d'optimiser mes programmes et mieux gérer les ressources.

1 - http://spout.zeal.co/

3.1.2 Effet peinture

Les effets de peinture sont des assemblages de *fragment shaders* écrits en GLSL. Dans ce projet, j'utilise trois différents *shaders* dont le but principal est d'étirer des couleurs dans certaines directions pour former des coups de pinceaux. Cette technique ne fonctionne pas sur les grands aplats de couleur uniforme. J'ai donc mélangé le ciel avec un bruit animé pour avoir l'effet de peinture sur toute l'image. J'ai également ajouté un effet de trace de peinture sur les nuages pour leur donner plus de consistance. J'ai testé cet effet sur le reste de la scène mais tout devenait brouillon ce qui contrariait l'objectif que nous nous étions fixé de garder une image cohérente tant au niveau de la forme que de la couleur.

Chaque plan de la scène possède un effet de peinture réglé différemment. Dans un premier temps, j'avais envisagé d'affiner les coups de pinceaux avec la distance. Mais les FBO représentant les plans les plus éloignés contenaient beaucoup d'*aliasing* et, si l'effet peinture était trop fin, l'image scintillait. Une des seules méthodes pour avoir un *antialiasing* de bonne qualité est de rendre l'image dans une résolution plus élevée pour ensuite réduire sa taille. Du fait du grand nombre de textures nécessaires pour le projet, cette solution n'était pas envisageable car nous y aurions sacrifié trop de performance.

Illustration n°109 : L'installation l'Aliaj Angelus - Effets environnementaux, Prière

Le principal écueil est le peu de variations dans les formes de coup de pinceau par plan. C'est un problème qui vient des *shaders* où la même opération est effectuée sur tous les pixels concernés. Nous aurions pu segmenter les images en sous-parties en fonction du niveau de détails souhaité. Par exemple, le traitement des visages aurait pu être plus détaillé que celui des vêtements. Je ne suis pas non plus entièrement satisfait par le ciel qui méritait un traitement global plutôt que local, avec des coups de pinceaux bien plus larges. En revanche, le résultat sur les plantes est satisfaisant.

3.1.2.1 Vidéo mapping, la matérialité dans la peinture

En trois semaines de réalisation, nous n'avons pu dégager suffisamment de temps pour élaborer la scénographie du projet. Nous voulions projeter la scène animée sur une toile entourée d'un cadre dans le style des peintures exposées dans les musées.

J'ai tenté de simuler numériquement le relief de la peinture pour lui donner plus de consistance. Pour cela j'ai généré une *normal map* à partir de la peinture et je m'en suis servi sur un *plane* que j'ai ensuite rendu. Mais le résultat n'était pas satisfaisant, on ne retrouve pas la matérialité propre à la peinture.

En revanche, j'ai constaté que lorsque que l'image est vidéo-projettée sur un support en relief tel qu'un mur ou une toile, l'effet numérique gagne une nouvelle dimension, une nouvelle matérialité qui rend le résultat plus crédible.

Ce que je retiens principalement du rendu d'effet peinture est que l'effet appliqué dépend du contenu de l'image, du support de diffusion et de la structure du projet. La principale contrainte est de garder une cohérence à l'image afin que celle-ci reste agréable à l'œil. En effet, le fait de travailler sur des éléments figuratifs rend l'exercice plus difficile et augmente les contraintes.

J'ai réalisé certains tests dont les résultats me semblent plus intéressants visuellement que ceux présentés dans le projet mais ils ont été écartés car ils contrariaient la composition globale du projet et donnaient un résultat moins fidèle à l'aspect recherché.

3.1.2.2 Nouvelle expérience esthétique

Nous avons choisi comme point de départ la peinture « *L'Angélus* » de Millet pour le projet l'*Aliaj Angelus* notamment parce que la scène se déroule dans un champs vide, ce qui nous laissait un espace d'expression que nous pouvions modifier tout en gardant la composition de l'image. Dans la peinture, les deux paysans se recueillent pour la prière de l'Angélus du soir. Ce moment de prière est évoqué par le titre de l'œuvre, par le coucher de soleil, par la position des paysans qui joignent leurs mains pour prier et par l'église au dernier plan, qui sonne la prière. Le tableau se rapporte à un temps précis : ce qui est montré, c'est le moment où le travail s'arrête pour respecter le temps de la prière.

Avec l'*Aliaj Angelus*, ce n'est plus un moment précis qui est montré. La peinture n'est plus statique, une nouvelle temporalité vient ajouter une dimension à l'œuvre. L'art sur

ordinateur est le seul médium permettant de simuler une constante évolution de l'image qui n'a d'autre but que l'adaptation au temps. Le temps n'est plus dans le regard du spectateur mais dans l'évolution du paysage. Les données météorologiques sont captées dans le lieu où est diffusée l'œuvre. La peinture s'adapte à son environnement, à l'espace d'exposition. Le système de *head tracking* permet également de donner un nouveau cadre à la peinture. Elle n'est plus limitée par le cadre de son support. L'artiste laisse entrevoir l'espace induit par le hors champs pour que le spectateur puisse plus facilement imaginer qu'il a en face de lui non plus une image représentant un moment mais un nouvel espace temps où des êtres virtuels évoluent. Selon le moment où le spectateur contemplera l'œuvre, son expérience sera différente. Au-delà de l'admiration d'un instant précis, le spectateur s'intéresse au concept de l'évolution et de l'adaptation. L'artiste numérique propose, plus qu'une représentation d'un moment réel, une représentation d'un monde virtuel.

3.2 Dessinateur de portrait

Dans ce projet, je me suis chargé du rendu. Isadora Teles de Castro e Costa a créé des agents autonomes qui se déplacent dans un cube en trois dimensions et qui peuvent être influencés par l'interaction du spectateur. Ces agents disposent de plusieurs types de comportement, ils peuvent se rassembler, être emportés par le vent ou vagabonder par exemple. Ils sont créés par l'utilisateur et meurent au bout d'un temps donné.

Je me suis donc occupé de la manière dont les agents allaient laisser des traces pour peindre. Pour cela, j'ai instancié au niveau de chaque agent des coups de pinceaux qui s'orientent dans la direction de leurs mouvements. Je laisse également une trace montrant le parcours de chaque entité. L'œuvre représente donc des traces de peintures qui se déplacent dans un espace cubique. Une couleur qui correspond à la couleur d'une image ou une vidéo est associée à la position et au mouvement d'un agent. Les illustrations qui suivent montrent des résultats avec différents paramètres.

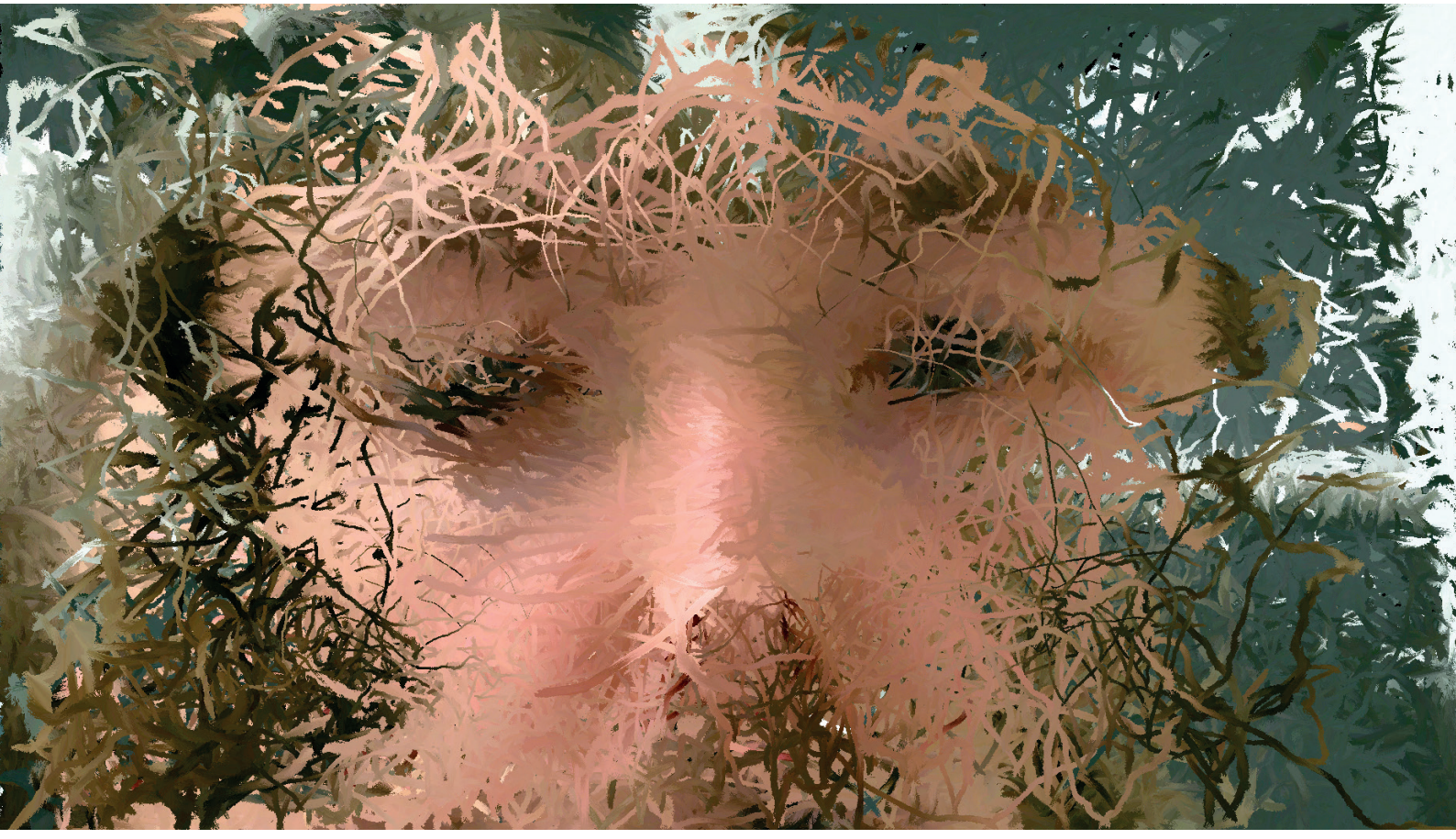


Illustration n°110 : Dessinateur de portrait, Visage Construit

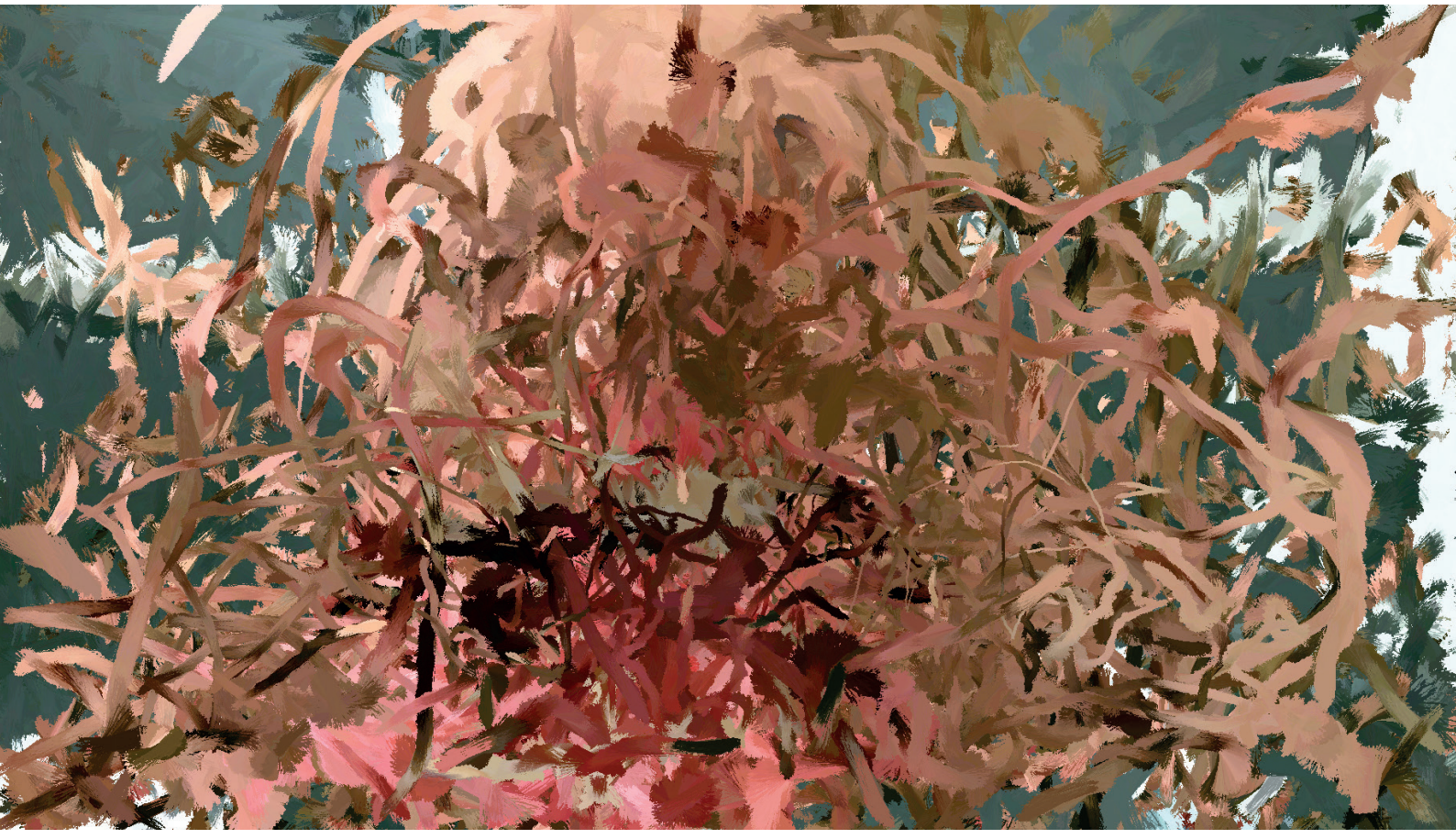


Illustration n°111 : Dessinateur de portrait, Visage Détruit

3.3 Peinture effet 3.0

Dans le cadre de ce projet, j'ai essayé d'approfondir et d'améliorer le principe de l'instanciation de coups de pinceaux, décrit dans la partie 2.3.3, pour transformer des images en peintures. L'intention principale était de chercher à réaliser un effet de peinture avec le moins d'informations de base disponibles pour pouvoir l'appliquer à n'importe quel type d'image. Toutes les solutions explorées ont été écrites en C++ et en GLSL pour obtenir la solution la plus rapide possible afin de pouvoir la réutiliser dans d'autres projets.

Pour cela, j'ai créé un certain nombre de *vertices* correspondant au nombre de coups de pinceaux souhaités dans la scène puis je les ai placés de manière à ce qu'ils remplissent la totalité du *viewport*. Lors des premiers tests, je plaçais un seul type de coup de pinceau avec une position et une orientation aléatoire fixe. Un des premiers résultats peut être observé dans l'illustration ci-dessous. Ce résultat manque de cohérence une fois animé car les coups de pinceaux n'évoluent pas, seule leur couleur change. Il est également difficile de discerner précisément le contour des formes.

Illustration n°112 : Peinture effet 3.0, Orientation fixe et aléatoire des coups de pinceau

J'ai décidé de régler en premier lieu le problème de la rotation des instances. Le résultat le plus concluant fut d'orienter les coups de pinceaux par rapport à leur couleur, ce qui permet d'avoir la même orientation dans une zone de couleurs similaires et d'avoir des transitions douces. Le problème des contours n'est toutefois pas résolu avec cette solution. J'ai donc utilisé un filtre de *sobel* qui permet de faire de la détection de contour. Pour diminuer le problème de débordement des coups de pinceaux, j'ai réduit la taille de ceux qui se trouvent au niveau des contours.

Pour obtenir plus de variation dans l'image, j'ai instancié plusieurs types de coups de pinceaux en même temps. J'ai donc assigné à chaque *vertex* un des quatre coups de pinceaux mis en place. Quand la scène est animée, la répétitivité des formes est apparente.

J'ai alors décidé d'animer les coups de pinceaux. Pour cela, j'ai choisi de manière aléatoire, à chaque image calculée, une des *brushes* disponibles dans ma petite bibliothèque de quatre *brushes*. Toutefois, comme l'image scintille quand le changement se fait trop rapidement, je n'actualise un nouveau coup de pinceau que douze fois par seconde. Avec cette valeur, il n'y a pas d'effet de scintillement ni d'impression de lenteur du programme. Cela permet de réduire le nombre de calculs nécessaires par seconde et j'ai pu ainsi instancier plus d'un million de coups de pinceaux sans perdre de performance.

En réalisant différent essais, je me suis aperçu qu'il faut faire varier les paramètres tels que la taille des *brushes* ou leur nombre en fonction des sujets représentés, du niveau de précision voulu et du style visé. L'activation ou la désactivation du *Z-buffering* rend également deux styles de peinture différents. Un portrait ne sera sûrement pas traité de la même manière qu'un paysage. Les illustrations qui suivent permettent d'observer plusieurs configurations.

Illustration n°113 : Peter Jackson, *The Hobbit* (2012)

Illustration n°114 : Peinture effet 3.0, *The Hobbit* avec 160 000 coups de pinceau de petite taille

Illustration n°115 : Peinture effet 3.0, Nature avec 160 000 coups de pinceau de petite taille

Illustration n°116 : Peinture effet 3.0, Nature avec 40 000 coups de pinceau de grande taille

Illustration n°117 : Peinture effet 3.0, Image tirée du film Avatar (2009) de James Cameron avec 160 000 coups de pinceau de petite taille

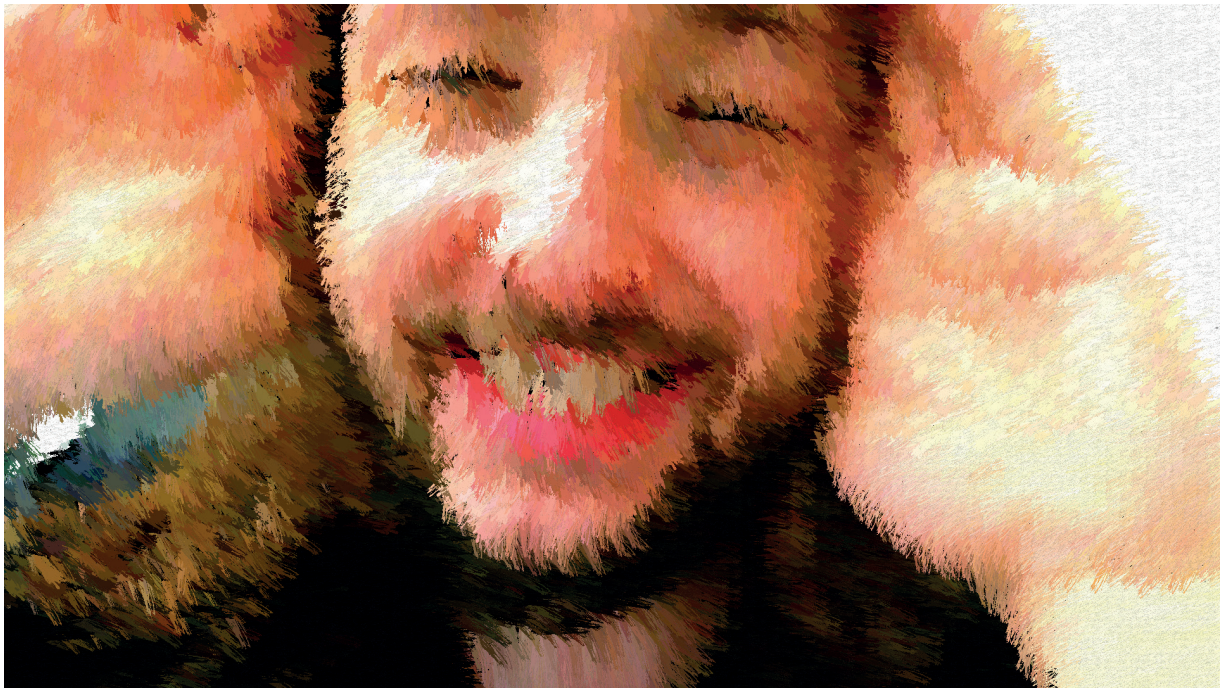


Illustration n°118 : Peinture effet 3.0, Autoportrait avec 40 000 coups de pinceau de grande taille

Conclusion

Au cours de ces recherches, j'ai tenté de trouver de nouvelles méthodes de production d'images calculées en temps réel pour automatiser le rendu pictural.

Il ressort de ces expérimentations trois résultats concluant qui répondent à la problématique de différentes manières. Tout d'abord la technique des filtres d'effets. C'est un procédé qui requiert peu de calculs et qui est applicable à la plupart des projets. Le résultat visuel est cohérent et uniforme. Son principal défaut est la difficulté à gérer précisément la forme des coups de pinceaux et donc à contrôler le style voulu.

La seconde technique, que je considère comme concluante, est celle consistant à laisser des traces pour simuler le mouvement d'un coup de pinceau. J'ai pu expérimenter deux manières de laisser une trace, soit par analyse du mouvement, soit par des agents autonomes. Le résultat de cette méthode de création propose une esthétique originale qui permet d'observer la formation d'une image dans le temps et qui s'intéresse plus au processus de création qu'au résultat lui-même. Cette technique reste toutefois difficile à mettre en œuvre dans le cadre d'un projet à cause de son esthétique très particulière et aux contraintes liées à sa création.

Finalement, c'est la méthode d'instanciation de coups de pinceaux à partir d'une image sur un plan en 3D qui correspond le mieux à ce que je recherchais à obtenir, c'est-à-dire un outil permettant d'automatiser, en temps réel, le processus de rendu évoquant la peinture animée. L'avantage de cette méthode par rapport aux précédentes est qu'elle est facilement modulable avec peu de paramètres tels que le choix du nombre et de la taille des coups de pinceaux. Il est également possible de personnaliser de manière simple une banque de traces de peinture pour personnaliser l'aspect final de l'œuvre.

Les principales nouveautés par rapport aux méthodes déjà existantes sont le système d'orientation des coups de pinceaux basé sur la couleur, la cohérence visuelle dans les séquences animées, le rendu en temps réel et le fait que la méthode est applicable à tout type d'image sans intervention de l'artiste.

C'est une solution qui est intéressante pour gagner du temps lors de productions cherchant à réaliser du rendu dans un style pictural. Je propose une nouvelle esthétique temps réel en gardant un niveau de paramétrisation permettant de contrôler le style de peinture.

Deux points en particulier mériteraient d'être améliorés : l'application de paramètres différents aux différentes zones de l'image en fonction de leur profondeur et la gestion des couleurs et de l'illumination qui sont des points essentiels du rendu pictural.

Bibliographie

Akenine-Möller, T., Haines, E. & Hoffman, N., (2008), *Real-Time Rendering*, Third Edition, A K Peters/CRC Press, Wellesley, Mass.

Bailey, M. & Cunningham, S., (2016), *Graphics Shaders: Theory and Practice*, Second Edition, CRC Press.

Couchot, E. & Hillaire, N., (2009), *L'art numérique*, Flammarion, Paris.

Dunn, F. & Parberry, I., (2011), *3D Math Primer for Graphics and Game Development, 2nd Edition*, A K Peters/CRC Press, Boca Raton, FL.

Fišer, J., Jamriška, O., Lukáč, M., Shechtman, E., Asente, P., Lu, J. & Sýkora, D., (2016), « StyLit: Illumination-guided Example-based Stylization of 3D Renderings », *ACM Trans. Graph.*, vol. 35, n°4, p. 92:1-92:11.

Flynt, J. & Lengyel, E., (2011), *Mathematics for 3D Game Programming and Computer Graphics*, Delmar Cengage Learning, Boston, MA.

Gatys, L. A., Ecker, A. S. & Bethge, M., (2016b), « Image Style Transfer Using Convolutional Neural Networks », *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, p. 24142423.

Gooch, B. & Gooch, A., (2001), *Non-Photorealistic Rendering*, A K Peters/CRC Press, Natick, Massachusetts.

Haeberli, P., (1990), « Paint by Numbers: Abstract Image Representations », *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, p. 207-214.

Hegde, S., Gatzidis, C. & Tian, F., (2013), « Painterly rendering techniques: a state-of-the-art review of current approaches », *Computer Animation and Virtual Worlds*, vol. 24, n°1, p. 4364.

Hertzmann, A., (1998), « Painterly Rendering with Curved Brush Strokes of Multiple Sizes », *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, p. 453-460.

Hertzmann, A., (2010), « Non-Photorealistic Rendering and the Science of Art », *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, ACM, New York, NY, USA, p. 147-157.

Hertzmann, A., Jacobs, C. E., Oliver, N., Curless, B. & Salesin, D. H., (2001), « Image Analogies », *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, p. 327-340.

Hertzmann, A. & Perlin, K., (2000), « Painterly Rendering for Video and Interaction », *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering*, ACM, New York, NY, USA, p. 7-12.

Itti, L., (2007), « Visual salience », *Scholarpedia*, vol. 2, n°9, p. 3327.

Kyprianidis, J. E., Collomosse, J., Wang, T. & Isenberg, T., (2013), « State of the «Art» : A Taxonomy of Artistic Stylization Techniques for Images and Video », *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, n°5, p. 866-885.

Litwinowicz, P., (1997), « Processing Images and Video for an Impressionist Effect », *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, p. 407-414.

Luft, T. & Deussen, O., (2006), « Real-time Watercolor Illustrations of Plants Using a Blurred Depth Test », *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*, ACM, New York, NY, USA, p. 11-20.

Meier, B. J., (1996), « Painterly Rendering for Animation », *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, p. 477–484.

Santella, A. & DeCarlo, D., (2002), « Abstracted Painterly Renderings Using Eye-tracking Data », *Proceedings of the 2Nd International Symposium on Non-photorealistic Animation and Rendering*, ACM, New York, NY, USA, p. 75-ff.

Sloan, P.-P. J., Martin, W., Gooch, A. & Gooch, B., (2001), « The Lit Sphere: A Model for Capturing NPR Shading from Art », *Proceedings of Graphics Interface 2001*, Canadian Information Processing Society, Toronto, Ont., Canada, p. 143-150.

Strothotte, T. & Schlechtweg, S., (2002), *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*, Morgan Kaufmann, San Francisco, Calif.

Taigman, Y., Yang, M., Ranzato, M. & Wolf, L., (2014), « DeepFace: Closing the Gap to Human-Level Performance in Face Verification », *2014 IEEE Conference on Computer Vision and Pattern Recognition*, p. 1701-1708.

Winnemöller, H., (2011), « XDoG: Advanced Image Stylization with eXtended Difference-of-Gaussians », *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, ACM, New York, NY, USA, p. 147-156.

Zanuttini, A., (2012), *Du photoréalisme au rendu expressif en image 3D temps réel dans le jeu vidéo : programmation graphique pour la profondeur de champ, la matière, la réflexion, les fluides et les contours*, Paris 8.

Zeng, K., Zhao, M., Xiong, C. & Zhu, S.-C., (2009), « From Image Parsing to Painterly Rendering », *ACM Trans. Graph.*, vol. 29, n°1, p. 2:1-2:11.

Webographie

Gatys, L. A., Ecker, A. S., Bethge, M., Hertzmann, A. & Shechtman, E., (2016a), « Controlling Perceptual Factors in Neural Style Transfer », *arXiv:1611.07865 [cs].* https://arxiv.org/abs/1611.07865

Gatys, L. A., Ecker, A. S. & Bethge, M., (2015), « A Neural Algorithm of Artistic Style », *arXiv:1508.06576 [cs, q-bio].* http://arxiv.org/abs/1508.06576

Gjoel, M. & Svendsen, M., (2016), « Low Complexity, High Fidelity: « INSIDE » Rendering. » http://www.gdcvault.com/play/1023783/Low-Complexity-High-Fidelity-INSIDE

Gonzalez Vivo, P., (s. d.), « The Book of Shaders, *The Book of Shaders.* » https://thebookofshaders.com/?lan=fr

Lagarde, S., (2012), « Water drop 1 – Observe rainy world | Sébastien Lagarde ». https://seblagarde.wordpress.com/2012/12/10/observe-rainy-world/

Motomura, J. C., (2015), « GuiltyGearXrd's Art Style : The X Factor Between 2D and 3D ». http://www.gdcvault.com/play/1022031/GuiltyGearXrd-s-Art-Style-The

Quilez, I., (2017), « Inigo Quilez :: fractals, computer graphics, mathematics, demoscene and more. » http://www.iquilezles.org/www/articles/functions/functions.htm

Rahteenko, A., (2016), « Gamasutra: Arthur Rahteenko's Blog - Rendering painted world in JG . » http://www.gamasutra.com/blogs/ArthurRahteenko/20160216/265908/Rendering_painted_world_in_JG.php

Ruder, M., Dosovitskiy, A. & Brox, T., (2016), « Artistic style transfer for videos », *arXiv:1604.08610 [cs],* vol. 9796, p. 26-36. https://arxiv.org/abs/1604.08610

Vasiliauskas, A., (2010), « Convolution Pixel Shader. » http://coding-experiments.blogspot.com/2010/07/convolution.html

Table des illustrations