Université de Montréal

Flou de mouvement réaliste en temps réel

par
Jean-Philippe Guertin-Renaud

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

août, 2014

RÉSUMÉ

Le flou de mouvement de haute qualité est un effet de plus en plus important en rendu interactif. Avec l'augmentation constante en qualité des ressources et en fidélité des scènes vient un désir semblable pour des effets lenticulaires plus détaillés et réalistes.

Cependant, même dans le contexte du rendu hors-ligne, le flou de mouvement est souvent approximé à l'aide d'un post-traitement. Les algorithmes de post-traitement pour le flou de mouvement ont fait des pas de géant au niveau de la qualité visuelle, générant des résultats plausibles tout en conservant un niveau de performance interactif. Néanmoins, des artefacts persistent en présence, par exemple, de mouvements superposés ou de motifs de mouvement à très large ou très fine échelle, ainsi qu'en présence de mouvements à la fois linéaires et rotationnels. De plus, des mouvements d'amplitude importante ont tendance à causer des artefacts évidents aux bordures d'objets ou d'images.

Ce mémoire présente une technique qui résout ces artefacts avec un échantillonnage plus robuste et un système de filtrage qui échantillonne selon deux directions qui sont dynamiquement et automatiquement sélectionnées pour donner l'image la plus précise possible. Ces modifications entraînent un coût en performance somme toute mineur comparativement aux implantations existantes : nous pouvons générer un flou de mouvement plausible et temporellement cohérent pour plusieurs séquences d'animation complexes, le tout en moins de 2ms à une résolution de 1280×720. De plus, notre filtre est conçu pour s'intégrer facilement avec des filtres post-traitement d'anticrénelage.

Mots-clés : flou de mouvement, post-traitement, rendu interactif, filtrage, infographie

ABSTRACT

High-quality motion blur is an increasingly important effect in interactive graphics. With the continuous increase in asset quality and scene fidelity comes a similar desire for more detailed and realistic lenticular effects.

However, even in the context of offline rendering, motion blur is often approximated as a post-process. Recent motion blur post-processes have made great leaps in visual quality, generating plausible results with interactive performance. Still, distracting artifacts remain in the presence of, for instance, overlapping motion or large- and fine-scale motion features, as well as in the presence of simultaneous linear and rotational motion. Furthermore, large scale motion often tends to cause obvious artifacts at boundary points.

This thesis addresses these artifacts with a more robust sampling and filtering scheme that samples across two directions which are dynamically and automatically selected to provide the most accurate image possible. These modifications come at a very slight penalty compared to previous motion blur implementations: we render plausible, temporally coherent motion blur on several complex animation sequences, all in under 2ms at a resolution of 1280×720. Moreover, our filter is designed to integrate seamlessly with post-process antialiasing.

Keywords: motion blur, post process, interactive graphics, filtering, computer graphics

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

À tous ceux et celles

qui m'ont guidé jusqu'ici.

REMERCIEMENTS

Avant tout, je me dois de remercier mes parents, qui m'ont toujours supporté où que j'aille. Dans les hauts comme dans les bas (et il y en a eu !), j'ai pu compter sur eux sans hésitation. Je ne sais pas où je serais sans leurs encouragements à me dépasser et à me donner entièrement à ma tâche, mais je me doute que ce ne serait pas un milieu aussi motivant et excitant que celui-ci.

Par la suite, je veux remercier Derek. J'ai de la difficulté à imaginer un meilleur directeur. Il a su me donner juste ce dont j'avais besoin : un petit coup de pouce çà et là, plusieurs opportunités de travailler avec des sommités du domaine, la latitude que j'apprécie avoir pour travailler. J'espère que nous pourrons continuer sur cette lancée pour les années à venir.

Je me dois aussi de mentionner Morgan, qui a été un collaborateur précieux dans mes travaux, en particulier ceux se retrouvant dans ce mémoire. Je n'aurais pu faire tout ceci sans son aide et ses idées !

J'aimerais remercier les autres membres du LIGUM. Bien que je n'aie pas été l'étudiant le plus présent dans le laboratoire, je savais que chaque fois que j'y allais, j'en ressortirais motivé et avec un intérêt renouvelé.

Je veux remercier Autodesk, et tout particulièrement Éric Bourque, Adam Felt et Bernard Kwok, qui m'ont offert un stage superbe, très intéressant et motivant. L'idée que mes travaux se retrouvent dans un logiciel aussi important et prestigieux que Maya me semble encore folle !

Finalement, il faut que je remercie le FQRNT, d'où j'ai retiré mon financement pour mes deux années de maîtrise, et les membres du jury, Pierre Poulin et Sébastien Roy, qui m'ont offert de précieux commentaires.

CHAPITRE 1

INTRODUCTION

L'évolution spectaculaire de l'imagerie de synthèse au cours des dernières années a progressivement augmenté la demande pour des effets visuels plus avancés. Initialement, le focus était généralement mis sur des textures plus détaillées, des maillages plus denses, puis on ajouta des modèles d'illumination plus performants et réalistes, des systèmes de particules et bien d'autres. Avec l'évolution constante de la fidélité graphique, on commença de plus en plus à se rapprocher du cinéma, tant au niveau de la qualité de l'image que d'un point de vue esthétique.

Cette nouvelle avenue d'améliorations introduisit une série de phénomènes nouveaux, par exemple les effets de halos (*lens flares*) ou de profondeur de champ, typiques des caméras utilisées dans le monde du cinéma. Ce mémoire se penche sur un type d'effet particulier : le flou de mouvement. Le flou de mouvement est un effet que l'on peut observer non seulement dans les films, mais qui a aussi un parallèle dans l'oeil humain, nommé *motion smear* [9], le rendant encore plus naturel d'un point de vue perceptuel. Qui plus est, le flou de mouvement est très utile pour masquer un nombre insuffisant d'images par seconde – là où même 50 images par seconde peuvent occasionner un effet de saccades sans flou de mouvement, on peut facilement descendre à la moitié de cela avec du flou de mouvement. C'est d'ailleurs pour cela que le cinéma, généralement filmé à 24 images par seconde, n'apparaît pas comme une série de clichés, mais bien comme un flux continu.

Le flou de mouvement, d'un point de vue physique, est relié au principe même de l'enregistrement d'une vidéo. Une caméra est généralement composée d'un système de lentilles, permettant de changer le point de focus, de recadrer le champ de vision et plus

généralement de rediriger la lumière incidente vers le capteur. Elle enregistre une image en ouvrant l'obturateur pendant une durée prédéterminée. Cette durée est variable et dépendante de la sensibilité du capteur et de sa taille, de l'ouverture de l'objectif, ainsi que de la luminosité de la scène.

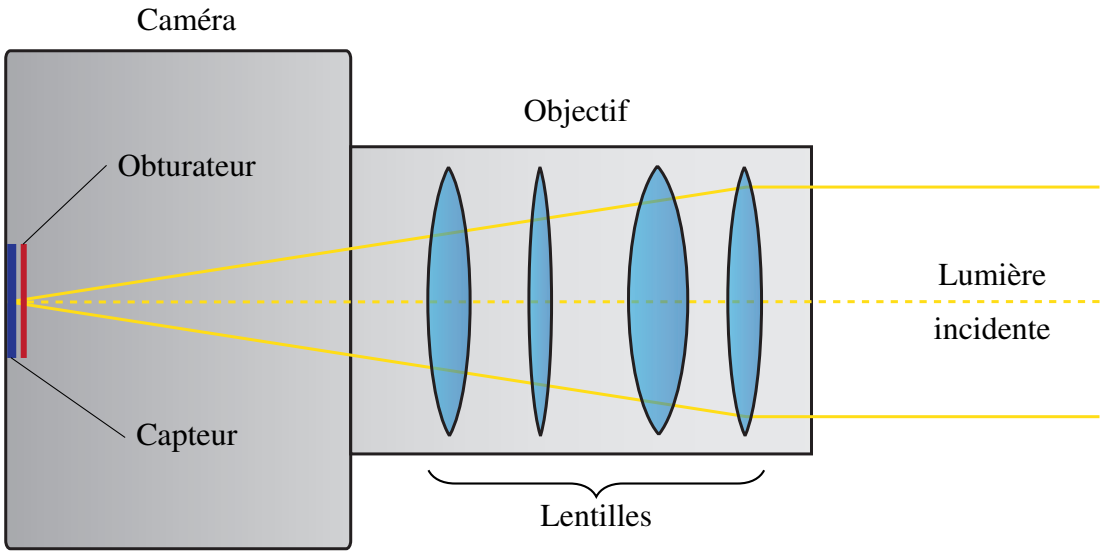


Figure 1.1 : Schéma simplifié d'une caméra typique. La lumière entre par l'objectif, où elle passe à travers un nombre de lentilles, puis arrive à l'obturateur. Si celui-ci est ouvert, la lumière est redirigée vers le capteur, qui convertit celle-ci en impulsions électriques qui sont par la suite traitées et enregistrées par la caméra.

Le flou de mouvement se produit parce que cette durée d'exposition n'est pas infinitésimale. En effet, il est ainsi possible d'avoir un certain mouvement durant l'intervalle d'exposition, que ce soit de la caméra ou du sujet, qui cause un flou là où les objets se déplacent dans l'image. On peut formaliser mathématiquement ce processus en explicitant l'illumination sur un intervalle de temps, ce qui revient à une intégration temporelle. Si on choisit $I(\omega)$ comme étant le contenu du plan de l'image aligné dans une direction ω, on peut alors définir cette valeur en fonction de la luminance (aussi appelée radiance) incidente à un instant t provenant de cette même direction ω, $L(\omega,t)$, et on obtient la

formulation décrite par Navarro et al. [26],

$$I(\omega) = \int_{\Delta T} f(\omega,t)\, L(\omega,t)\, \mathrm{d}t\ . \tag{1.1}$$

On doit aussi prendre en considération les effets du système optique et électronique de la caméra, que nous représentons par $f(\omega,t)$.

Cette formulation reste cependant assez grossière et peu descriptive. On peut donc dériver un modèle plus avancé tel que celui décrit par Sung et al. [42] :

$$I_{xy} = \sum_{l} \int_{\Omega} \int_{\Delta T} r(\omega,t)\, g_l(\omega,t)\, L_l(\omega,t)\, \mathrm{d}t\mathrm{d}\omega\ . \tag{1.2}$$

Premièrement, on divise notre image $I(\omega)$ en une grille de taille fixe mais arbitraire de points, ou pixels. On peut alors exprimer la valeur finale d'un seul pixel, I_{xy}. L'intégration est effectuée pour chaque objet l de la scène, ce qui permet de mieux isoler les déplacements de chacun, et repose sur le terme géométrique $g_l(\omega,t)$, qui est 1 si l'objet est visible au pixel et 0 sinon. Celle-ci se fait aussi sur un angle solide Ω, permettant de prendre en compte les déplacements de la caméra. Finalement, on définit un filtre de reconstruction $r(\omega,t)$, qui considère les divers effets de lentilles et de formes de l'obturateur. $L_l(\omega,t)$ représente la radiance de l'objet l, mais sans explicitement donner son calcul.

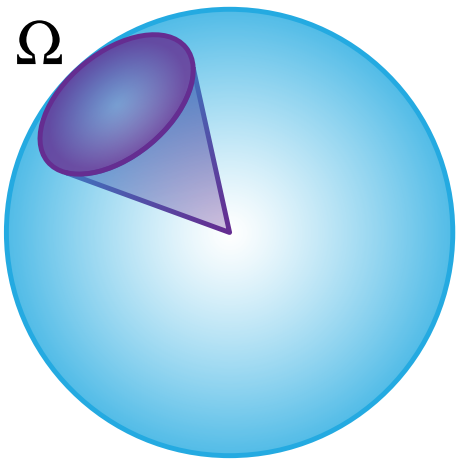


Figure 1.2 : Représentation visuelle d'un angle solide Ω sur la sphère unitaire.

Cette formulation est assez générale pour être utilisable dans une variété de modèles de rendu, y compris les scènes polygonales, analytiques, implicites, ou même comprenant des milieux semi-transparents ou participatifs (gaz, fumée, etc.). Cependant, elle peut être encombrante à calculer en pratique, aussi existe-t-il un grand nombre d'algorithmes qui permettent, à l'aide de diverses techniques, d'approcher une solution semblable.

CHAPITRE 2

FLOU DE MOUVEMENT

Ce chapitre a pour but de faire un tour d'ensemble des techniques de flou de mouvement utilisées de nos jours, ainsi que des idées et de l'évolution qui ont mené jusqu'à celles-ci. Il sera par la suite question des algorithmes qui ont servi de fondements à l'article au cœur de cette thèse, qui sera présenté au chapitre suivant. Le chapitre est ainsi séparé en trois sections distinctes :

- L'**évolution initiale** couvre les balbutiements du flou de mouvement généré par ordinateur, alors que les effets de lumière distribués étaient encore une rareté.
- Le **survol des techniques existantes** passe en revue les techniques plus modernes et avancées pour générer du flou de mouvement, que ce soit en temps réel ou en rendu différé.
- Finalement, l'**état de l'art** décrit en détail l'algorithme de pointe sur lequel l'article de cette thèse s'est fondé.

2.1 Évolution initiale

Les toutes premières utilisations d'un ordinateur pour générer des images de synthèse remontent à la fin des années 50 avec des systèmes tels le TX-2 du MIT. Les premières images étaient crues, sans prétentions d'approcher la réalité, ne serait-ce que légèrement. Dans les années 60, on voit apparaître des articles tels celui de Shack [36] qui sont parmi les premiers à proposer des fonctions de transfert pour l'exposition d'images photographiques, un problème analogue au rendu d'images. Il fallut cependant attendre les

années 70 et surtout les années 80 pour voir une explosion des techniques de rendu, motivée en grande partie par la croissance exponentielle de la performance des ordinateurs de l'époque. On cherchera alors à imiter le dessin, puis le monde réel.

La technique la plus simple pour simuler le flou de mouvement consiste à prendre des clichés instantanés d'une animation à plusieurs moments dans le temps, puis à les superposer pour en faire une moyenne. Cette technique, qu'on appelle *accumulation*, nous renvoie à l'ère des animations image par image faites à la main. Elle fut mentionnée pour la première fois dans le contexte de la synthèse d'images en 1983 par Korein et Badler [16]. L'avantage majeur de cette technique est qu'elle ne requiert aucune capacité particulière du système de rendu : il suffit de faire un grand nombre de rendus indépendants à des moments différents, puis de les combiner ensemble à l'aide d'un filtre quelconque (p.ex. une moyenne). L'inconvénient est qu'il faut un *grand* nombre d'images pour obtenir un résultat convaincant, particulièrement en la présence d'objets minces dans la direction du mouvement. Korein et Badler mentionnent d'ailleurs cette lacune sous le nom d'*effet spaghetti*, où il devient possible de distinguer les images ayant servi au filtrage. C'est pour cette raison que la technique d'accumulation fut rapidement abandonnée en dehors de contextes particuliers.

Une autre technique, provenant du monde du traitement de signal, était plus complexe. Présentée par entre autres Potmesil et Chakravarty [29] et Max et Lerner [22], elle consiste en un filtre de convolution permettant de prendre une image statique et des informations de la scène pour produire une image floue. Potmesil et Chakravarty utilisaient des *point spread functions*, basées sur une analyse de Fourier de la scène, alors que Max et Lerner transformaient la scène selon la magnitude et la direction du mouvement avant de flouter le résultat, puis annulait la première transformation. Dans un cas comme dans l'autre, les techniques étaient plus efficaces qu'une simple accumulation, mais avaient

des lacunes sur certaines scènes les rendant souvent trop peu flexibles.

La première technique viable fut développée en 1984 par Cook et al. [4]. À cette époque, il était déjà fréquent d'utiliser le sur-échantillonnage pour effectuer de l'anti-crénelage sur les images rendues, ce qui rend les silhouettes des objets plus lisses. On lançait ainsi plusieurs rayons dans la scène pour chaque pixel, et ceux-ci retournaient une couleur légèrement différente ; il suffisait alors d'en faire une moyenne. L'innovation était de non seulement altérer la position du rayon dans l'espace, mais aussi dans le temps. Ainsi, on pouvait échantillonner chaque objet à plusieurs moments dans le temps et naturellement créer un flou de mouvement convaincant. Cette technique est, à plusieurs niveaux, semblable à la simple accumulation : elle reste une moyenne de plusieurs clichés dans le temps. La différence majeure est que chaque pixel contient une accumulation d'instants différents, ce qui brise la cohérence des échantillons et permet d'en utiliser un nombre bien plus faible tout en obtenant un résultat acceptable. Cette propriété stochastique de l'algorithme reste prisée encore aujourd'hui.

La technique de Cook et al. fut si révolutionnaire et efficace qu'elle fut utilisée dans le premier film par images de synthèse avec flou de mouvement : c'était André & Wally B., sorti en 1984, le premier court-métrage de Pixar Animation Studios.

2.2 Survol des techniques existantes

Ce survol se veut une vue d'ensemble des principales techniques de flou de mouvement. L'emphase est mise sur les techniques plus récentes ayant un potentiel aujourd'hui, et non sur les techniques historiques qui ont été surpassées depuis. Pour une analyse plus détaillée, le *State of the Art Report* (STAR) sur le flou de mouvement est vivement recommandé [26].

2.2.1 Techniques géométriques

Les techniques géométriques se basent toutes sur le même principe : au lieu de devoir explicitement échantillonner le mouvement en modifiant la coordonnée temporelle lors du rendu, on crée, avant le rendu, un objet modifié qui prend en compte le mouvement. Le premier exemple de ce genre de technique, et une bonne illustration de l'idée en général, vient de Reeves [31], qui a conçu un système de particules où le flou causé par leur mouvement est représenté en remplaçant chaque particule par un segment de ligne anticrénelé représentant la courbe de mouvement de la particule. Bien entendu, cette technique est limitée, ne permettant pas de représenter des particules de formes différentes ou ayant une texture.

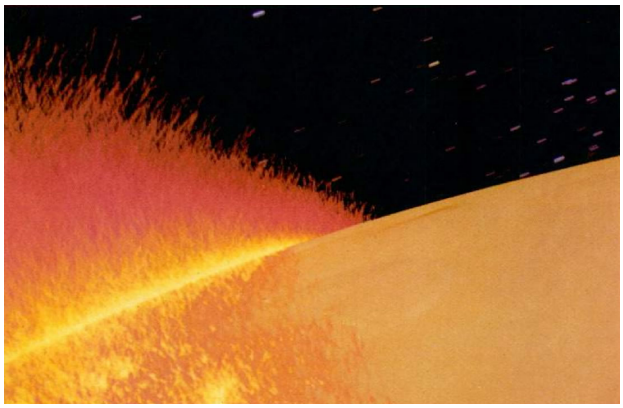


Figure 2.1 : Exemple de rendu du système de particules de Reeves [31].

Figure 2.2 : Exemple de rendu stylistique de *Programmable Motion Effects* par Schmid et al. [34].

Plusieurs applications de cette catégorie de techniques sont destinées à un usage plus artistique que réaliste. On tente plutôt de créer un effet de mouvement convaincant, sans nécessairement se baser sur un modèle physique. La géométrie et le mouvement peuvent alors être simplifiés pour accélérer le calcul.

L'article *Programmable Motion Effects* [34] est un exemple de cette application plus stylistique. Un objet en mouvement est échantillonné

à plusieurs moments dans le temps, puis un méta-modèle est généré depuis ces échantillons, reliant les arêtes correspondantes dans chaque échantillon entre elles pour former des surfaces. Ce méta-modèle peut alors être rendu pour représenter l'objet en mouvement. Cette technique possède l'avantage d'effectuer le rendu sur l'objet arbitrairement, ce qui permet de créer des effets artistiques, par exemple des images stroboscopiques ou des lignes de mouvement, mais il a comme désavantage de ne pouvoir supporter directement les textures : il faut appliquer la texture sous forme de couleur à chaque sommet, forçant plus d'échantillons (et donc plus de calculs) pour correctement représenter des textures à haute fréquence.

Une autre technique présentée par Guan et Mueller [10] s'applique au rendu par points. Chaque point de l'image en mouvement est représenté par une sphère ; ainsi, le mouvement peut être approximé par un ellipsoïde orienté dans la direction du mouvement. Cet ellipsoïde est rendu en 2D sous la forme d'une ellipse avec un filtre Gaussien, permettant d'appliquer le fondu sur le mouvement. Le mouvement est ainsi assumé linéaire, mais comme chaque point est rendu indépendamment, la rotation peut être tout de même représentée par translation de chaque point. De même, c'est l'une des rares techniques géométriques pouvant supporter la déformation d'objets, car ceux-ci sont formés de points indépendants.

2.2.2 Techniques stochastiques

Les techniques stochastiques, particulièrement l'intégration numérique Monte Carlo, sont très populaires pour représenter non seulement des effets distribués comme le flou de mouvement, mais aussi pour effectuer le rendu général d'une scène. N'étant pas intrinsèquement une approximation, il est possible, avec un nombre d'échantillons infini, d'obtenir une image exacte, pourvu que les modèles de la scène soient physiquement

corrects. Cette puissance de calcul en fait un incontournable dans l'industrie du cinéma.

L'algorithme de lancer de rayons distribués de Cook est à la base de la vaste majorité de ces techniques. L'équation 1.2 y est estimée par

$$I_{xy} = \frac{1}{N_j}\sum_{j}^{N_j} i_d(\omega_j,t_j)\ , \qquad i_d(\omega_j,t_j) = \sum_{l} r(\omega_j,t_j)\ g_l(\omega_j,t_j)\ L_l(\omega_j,t_j)\ , \tag{2.1}$$

où chaque échantillon $i_d(\omega_j,t_j)$ représente la contribution de tous les objets l de la scène tels que vus au temps t_j et depuis la direction ω_j. La constante N_j indique le nombre d'échantillons et ainsi contrôle directement le temps de calcul et la précision de l'image résultante.

Un élément orthogonal mais tout de même important pour le flou de mouvement est la technique d'échantillonnage. Les premières implantations du lancer de rayons distribués se contentaient d'un échantillonnage uniforme, mais rapidement des techniques telles l'*importance sampling* ou le *stratified sampling*, bien connues dans d'autres domaines faisant usage de la méthode Monte Carlo, sont venues accélérer les calculs en distribuant les échantillons plus intelligemment.

De la même manière, une technique alternative s'est développée plus récemment, prenant le nom de *stochastic rasterization*. La technique, décrite par exemple par McGuire et al. [23], propose d'utiliser le système de rastérisation des cartes graphiques actuelles pour effectuer un rendu stochastique (donc semblable aux techniques Monte Carlo) d'une scène polygonale. Ceci permet non seulement de simuler des effets de lumière et des modèles de surfaces plus avancés, mais aussi de représenter le flou de mouvement ou la profondeur de champ naturellement.

Outre le calcul naïf des échantillons en quatre dimensions, plusieurs articles se sont concentrés sur des méthodes alternatives de représenter le mouvement dans le temps.

L'une des avenues les plus populaires est d'effectuer une analyse dans le domaine des fréquences ; un article récent d'Egan et al. [5] montre qu'il est possible d'utiliser un filtre de reconstruction pour déterminer une image nette depuis une image très bruitée, échantillonnée en temps et en espace. Cette technique utilise un *sheared filter* (filtre de cisaillement) pour approximer le flou de mouvement et prend en charge les déplacements d'objets, les rotations des BRDFs et de l'illumination et même les ombres dynamiques. Cette reconstruction dépend d'une analyse en espace image des vélocités par pixel des objets de la scène et prend un temps relativement court vis-à-vis de la génération de l'image elle-même. Un article plus récent de Munkberg et al. [25] propose un filtre semblable capable de traiter une image de taille intéressante en moins d'une seconde, ce qui en fait un candidat particulièrement intéressant pour de futures implantations en temps réel. Ce genre de filtres, appliqué à la rastérisation stochastique, pourrait éventuellement remplacer les techniques post-traitement plus approximatives présentement utilisées en rendu temps réel. Un exemple de cette combinaison a été présenté par Shirley et al. [38].

Une autre avenue de recherche se concentre sur l'amélioration de la technique de tracé de chemins elle-même. Bien que généralement orthogonale à la génération du flou de mouvement et d'autres effets distribués, si chaque chemin est plus efficace ou s'il est possible de lancer plus de rayons pour le même coût, cette augmentation de performance est généralement réutilisée en traçant plus de chemins, ce qui permet de mieux représenter les effets tels le flou de mouvement sans augmenter le coût. L'innovation la plus connue dans le monde du tracé de chemins reste sans doute le *Metropolis Light Transport* [44] (MLT), qui se base sur la technique de tracé de chemins bidirectionnel proposé par Lafortune et Willems [17] où on trace un chemin depuis la lumière et un depuis la caméra et on essaie de relier ces deux chemins entre eux. MLT augmente cette technique en faisant usage de *mutations* : lorsqu'on détermine un chemin intéressant,

l'algorithme tente de modifier ce chemin selon des règles prédéterminées pour découvrir d'autres chemins semblables et possiblement difficiles à trouver autrement. Plus récemment, l'algorithme de *Vertex Connection and Merging* [8] tente de combiner le tracé de chemins bidirectionnel avec le tracé de photons bidirectionnel pour améliorer la qualité des chemins résultants, prenant la technique la plus adaptée à la situation selon le cas par *multiple importance sampling*.

Le tracé de photons est d'ailleurs capable de produire du flou de mouvement. Introduit en 1996 par Jensen [14], la technique consiste en un rendu en deux phases : la première lance des rayons depuis les lumières, stockant à chaque intersection des informations relatives au point d'intersection et à la lumière incidente, qu'on appelle *photons*. La seconde phase lance alors des rayons depuis la caméra et échantillonne la scène à chaque intersection pour trouver les photons accumulés en cet endroit et donc pour y estimer la lumière. Le *photon mapping* a d'ailleurs un second nom : le *photon density estimation*. Le tracé de photons peut être lui aussi augmenté pour effectuer du flou de mouvement, tel qu'expliqué par Cammarano et Jensen [2] : on attribue à chaque photon une composante temporelle aléatoire, composante qui sera partagée par tous ses descendants. Lors de la seconde phase, les rayons lancés depuis la caméra utiliseront un filtre en espace mais aussi en temps, moyennant ainsi les contributions passées et futures des photons.

2.2.3 Techniques post-traitement

Les techniques post-traitement sont toutes basées sur une même approche : au lieu de modifier l'algorithme produisant l'image pour y inclure une composante temporelle, on utilise le résultat sans composante temporelle explicite combiné avec des informations supplémentaires sur la scène pour produire une image filtrée qui elle comprendra les

effets de flou de mouvement. La technique de rendu de la scène est ainsi entièrement indépendante de la technique produisant le flou de mouvement, pourvu que les prérequis de cette dernière puissent être satisfaits par la première.

Mathématiquement, on peut exprimer le calcul effectué par ces techniques de cette manière :

$$I_{xy} \approx I(x, y, t_i) \otimes \Phi(t_0, t_\infty, t_i) \, , \tag{2.2}$$

où Φ est une fonction puisant l'information contenue dans les images de tous les temps de t_0 à t_∞. En pratique, cette fonction est généralement restreinte à un plus petit intervalle autour de t_i.

La technique, mentionnée dans la section précédente de Potmesil et Chakravarty [29], est l'un des exemples les plus anciens de ce genre de techniques, l'image source étant rendue sans modification temporelle. Cependant, la vaste majorité des techniques en usage aujourd'hui requièrent l'accès à une donnée critique : un champ de vecteurs, appelé *motion field*, qui représente le mouvement instantané de la scène au moment t_i choisi. Un exemple commun de cette représentation est sous la forme d'une image de la même taille que l'image finale, mais où chaque pixel V_{xy} contient la vitesse vectorielle $\mathbf{v}_{xy}$ de l'objet rendu à ce pixel.

Une technique prenant avantage de ce champ de vecteurs utilise l'information de mouvement vectoriel pour construire une nouvelle géométrie déformée selon ce mouvement [37]. La nouvelle géométrie possède aussi un champ de vecteurs du flux optique, qui explicite le mouvement *apparent* des objets du point de vue de la caméra. Ce flux optique est alors utilisé pour flouter la géométrie déformée dans le sens du mouvement.

Dans l'industrie du jeu vidéo, plusieurs techniques de type post-traitement ont été développées pour simuler un flou de mouvement approximatif dans un budget de temps très petit. Un exemple est donné par Sousa [39] : on calcule les vecteurs de vélocité à

l'aide des matrices de transformation précédentes et suivantes, nous donnant un vecteur de déplacement pour chaque pixel selon le mouvement des objets et de la caméra. Un masque calculé depuis la profondeur à chaque pixel est utilisé pour moduler l'importance du flou de mouvement (les objets proches sont ainsi moins flous, approximant un mouvement courbe avec des données linéaires). La vélocité des objets est alors dilatée (séparément du mouvement de la caméra) pour que le flou se propage au-delà des objets, sans quoi leurs bords resteraient nets. Le flou est calculé itérativement en floutant linéairement selon la direction de la vélocité avec un faible nombre d'échantillons, puis en floutant à nouveau ce résultat plusieurs fois pour obtenir un résultat apparent lisse. Chaque itération supplémentaire a alors un effet multiplicatif : on obtient l'équivalent de n^k échantillons pour k itérations de n échantillons.

2.3 État de l'art

L'algorithme présenté dans ce mémoire s'inscrit dans la catégorie des techniques post-traitement, une évolution s'inspirant des techniques précédentes les plus populaires et efficaces. Pour donner un contexte plus complet que celui présenté dans l'article proprement dit, cette section détaillera l'algorithme de McGuire et al. [24] présenté en 2012 et qui était l'état de l'art avant la présentation de notre nouvel algorithme. Nous nous en sommes grandement inspirés pour élaborer notre algorithme final.

2.3.1 Contexte

Avant tout, il est important de situer l'algorithme. En effet, comme celui-ci est un algorithme en temps réel, il doit se plier à un nombre plus important de contraintes et à des limites plus strictes. En premier lieu, l'algorithme est un post-traitement dans le cadre d'un moteur de rendu de type rastérisation. Ceci limite l'information disponible : il

est impossible (ou, du moins, très complexe et coûteux) d'effectuer des rebonds dans la scène à la manière d'un lanceur de rayons. De plus, l'information de mouvement disponible est entièrement basée sur le mouvement des objets eux-mêmes (mouvement relatif à la caméra, donc incluant les mouvements de celle-ci), ce qui rend le traitement d'effets périphériques liés aux objets impossible, le plus évident étant les ombres. Finalement, le rendu de ces données de mouvement se fait en une seule étape dans un temps très court et sans prendre beaucoup de mémoire, limitant l'information écrite à un seul vecteur par pixel.

Ainsi, l'algorithme ne traite pas ou ne garantit pas le résultat dans les situations ou pour les éléments suivants :

- Les objets cachés derrière d'autres objets, qui ne causeront aucun flou et seront estimés depuis les pixels voisins jusqu'à apparaître dans l'image ;
- Les ombres, qui ne reproduiront pas le flou des objets les causant ;
- Les miroirs ou toute autre surface hautement réflective, qui seront floutés dans la direction de leur mouvement et selon ce qui est réfléchi ;
- Les mouvements non-linéaires, qui seront approximés comme linéaires ;
- Un grand nombre de mouvements différents par pixel, dans lequel cas seul le plus important sera utilisé.

À noter que, malgré le manque de support explicite, plusieurs de ces exceptions resteront visuellement acceptables dans des situations réelles.

2.3.2 Vue d'ensemble

L'algorithme prend en entrée trois tampons :

1. Le tampon de couleur, sans flou ;

2. Le tampon de profondeur, idéalement linéaire ;

3. Un tampon de vélocité.

Ici, le tampon de vélocité est calculé d'une manière précise : on recherche le mouvement en espace image seulement, sans composante de profondeur. De plus, on veut limiter la magnitude du vecteur à une vitesse maximale définie par l'implantation et qui affectera l'importance du flou de mouvement. Toutes ces mesures sont faites en pixels pour simplifier. On aura donc une vitesse

$$\mathbf{v}_{\mathbf{p}_t} = \frac{1}{2}\left(\mathbf{p}_t - \mathbf{p}_{t-1}\right) \cdot E \cdot F \ , \tag{2.3}$$

avec $\mathbf{p}_t$ la position 2D en espace image du pixel au temps t, E le temps d'exposition et F le taux d'images par seconde. Cette vitesse est alors stockée dans un tampon

$$\mathrm{V}[\mathbf{p}_t] = \frac{\mathbf{v}_{\mathbf{p}_t} \max\left(0.5,\ \min\left(\left\|\mathbf{v}_{\mathbf{p}_t}\right\|,\ k\right)\right)}{\left\|\mathbf{v}_{\mathbf{p}_t}\right\| + \varepsilon} \ , \tag{2.4}$$

avec k la vitesse maximale (en pixels) et ε une petite valeur pour éviter les divisions par zéro. On notera que la vitesse obtenue est en fait celle à $t - 1/2$: ceci est une contrainte du temps réel, car on ne peut connaître la vitesse au temps $t + 1$.

Une fois ces données disponibles, l'algorithme procède en trois phases.

2.3.3 Phase *TileMax*

La phase *TileMax* a comme but d'effectuer une dilatation contrôlée du tampon de vélocité. Cette dilatation se fait sur une grille de tuiles (*tiles*) de taille $k \times k$. Pour chaque tuile, on identifie la vélocité dont la magnitude est la plus grande, et on la stocke dans le

tampon *TileMax*. Cette opération peut être formalisée comme

$$\text{TileMax}[x,y] = \underset{u\in[0,k[}{\text{vmax}}\,\underset{v\in[0,k[}{\text{vmax}}\left(\text{V}\left[kx+u,ky+v\right]\right), \tag{2.5}$$

où l'opérateur vmax retourne le vecteur avec la norme la plus grande.

Le but de cette dilatation est de pouvoir déterminer, à un pixel **p** quelconque, quelle est la vélocité la plus importante (donc de plus grande magnitude) qui peut affecter celui-ci. On notera cependant que la phase *TileMax* n'est pas suffisante à elle seule pour couvrir toutes les possibilités : si une vélocité se trouve, par exemple, dans le coin en haut à droite d'une tuile et qu'elle se dirige dans cette même direction, elle affectera la tuile voisine.

2.3.4 Phase *NeighborMax*

Cette observation nous conduit tout droit à la phase *NeighborMax*, qui effectue une dilatation de *TileMax*. Plus précisément, on considère le voisinage immédiat de chaque tuile, donc un total de neuf tuiles, et on sélectionne une fois de plus la vélocité la plus importante. Cette opération peut se formaliser par

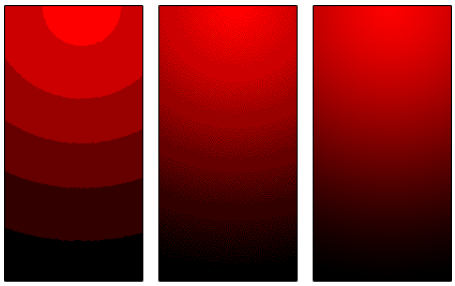
$$\text{NeighborMax}[x,y] = \underset{u\in[-1,1]}{\text{vmax}}\,\underset{v\in[-1,1]}{\text{vmax}}\left(\text{TileMax}\left[x+u,y+v\right]\right). \tag{2.6}$$

On notera ici que cette double opération aurait techniquement pu être effectuée en une seule phase de dilatation de taille $3k$, mais le résultat aurait été beaucoup moins précis. De plus, la performance d'une telle opération aurait été beaucoup moins intéressante que de diviser le travail en deux phases.

2.3.5 Phase finale

La dernière phase de l'algorithme effectue le filtrage proprement dit, utilisant en entrée le tampon de couleur, le tampon de profondeur, le tampon de vélocité et le tampon stockant *NeighborMax*. Le filtre utilise un nombre N constant d'échantillons tel que défini par l'usager. Pour chaque pixel X, on échantillonne dans la direction définie par $\mathbf{v}_N$, la vélocité maximale du voisinage stockée dans *NeighborMax*. Si le mouvement est inférieur à un demi pixel, on retourne immédiatement, car effectuer le filtre serait inutile.

L'échantillonnage se fait à intervalle régulier sur la plage $[-\mathbf{v}_N, \mathbf{v}_N]$. Pour éliminer les bandes de couleur, ou *banding* (voir figure 2.3), on applique un décalage aléatoire $j \in [-0.5, 0.5]$ sur tous les échantillons ; ce décalage est le même pour tous les échantillons du pixel pour éviter l'entassement d'échantillons et pour accélérer l'algorithme.



a) b) c)

Figure 2.3 : Effet visuel des bandes de couleur (a). Le *dithering* (b), ou tramage, est une forme de bruitage qui produit un résultat semblable au décalage aléatoire employé par l'algorithme de McGuire et al. Le résultat devient semblable à un échantillonnage beaucoup plus élevé (c).

Pour chaque échantillon, on effectue une comparaison de la profondeur avec la profondeur du pixel. Cette comparaison "douce" (*soft*) donne une transition graduelle entre les états "devant" et "derrière", représentés par 1 et 0.

À l'aide de ces données, on calcule alors un poids pour l'échantillon comme la somme de trois cas :

1. Le premier cas représente un échantillon flou se déplaçant devant le pixel. Ce cas est significatif si l'échantillon est devant le pixel et si la magnitude de la vélocité à l'échantillon est suffisamment grande pour pouvoir affecter le pixel.

2. Le second cas représente le cas inverse : on tente d'estimer l'arrière-plan lorsque le pixel lui-même est en déplacement en échantillonnant les points aux alentours. Ce cas est donc significatif si l'échantillon est derrière le pixel et si la vélocité au pixel peut rejoindre l'échantillon.

3. Le dernier cas est conçu pour lisser la transition entre les deux cas précédents, par exemple aux silhouettes d'objets en mouvement. Sans celui-ci, les bords des objets sont beaucoup trop nets.

Il suffit alors d'accumuler le poids et la couleur pondérée pour obtenir une couleur finale. On notera que l'algorithme échantillonne aussi explicitement le pixel en lui donnant un poids initial de $1/\|\mathrm{V}[X]\|$.

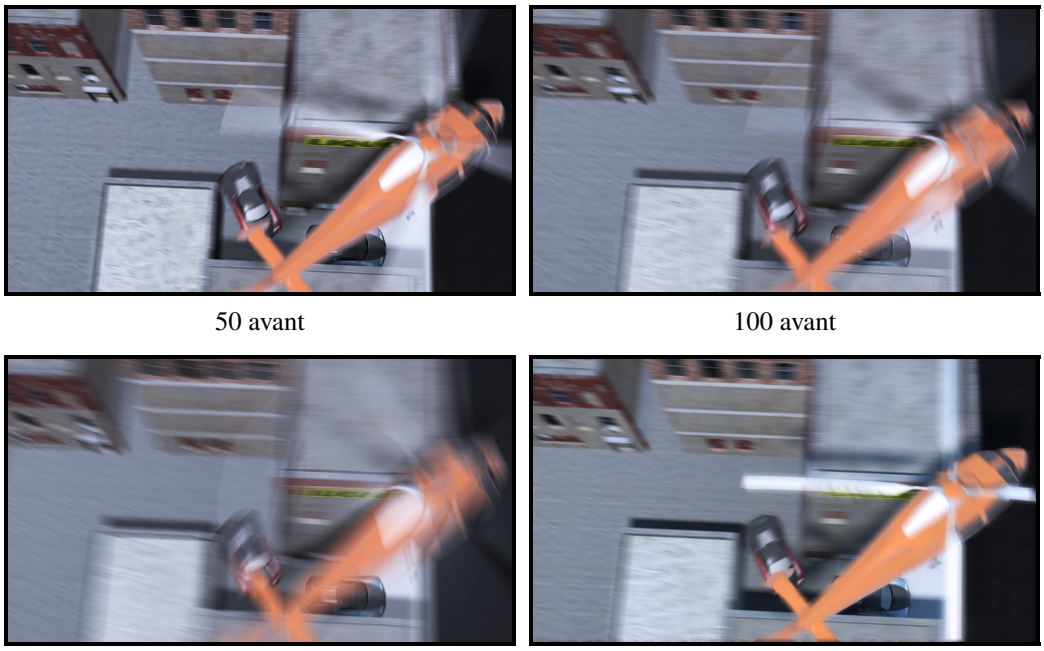
2.4 Sommaire

Il est évident, à la lumière des nombreux articles et travaux cités dans ce chapitre, que le flou de mouvement est un sujet prisé qui a longuement été jugé comme l'un des éléments critiques permettant à l'infographie d'égaler le niveau esthétique obtenu par une caméra cinématographique contemporaine. Ce travail est une continuation de cette tendance, cherchant à son tour à améliorer encore plus le rendu de cet effet complexe, mais essentiel.

Notre algorithme poursuit dans la lignée de l'état de l'art présenté dans la section 2.3. Nous considérons ainsi les mêmes contraintes et les mêmes buts. Un élément notable est l'absence de comparaison avec des résultats dits "ground truth", soit en l'occurrence un flou de mouvement parfait obtenu, par exemple, par accumulation d'un nombre massif d'images. Cette discussion a été omise car nous n'avons pas la prétention d'approcher le "ground truth", mais bien simplement de produire un résultat esthétiquement plaisant. Nos correctifs vis-à-vis McGuire et al. [24] sont tous évidemment supérieurs à l'oeil nu,

ce qui était notre métrique principale dans le cadre de l'article, où les compromis étaient souvent nombreux.

Par souci de complétude, la figure 2.4 offre une comparaison entre des images obtenues par accumulation brute et l'image produite par notre algorithme. On notera immédiatement que, bien que les pales de l'hélicoptère soient visuellement agréables dans notre résultat, elles sont très loin de la réalité en raison surtout de l'hypothèse de mouvements linéaires. D'ailleurs, ce schéma se répète dans toute l'image : les mouvements linéaires sont somme toute assez bien reproduits, mais les mouvements en courbe sont très approximatifs. On remarquera aussi des défauts là où un objet en masque un autre, car aucune information sur ce qui est caché n'est disponible.



50 avant | 100 avant

100 avant et 100 après | Notre algorithme

Figure 2.4 : Comparaison de notre algorithme avec un "ground truth" obtenu par accumulation d'un certain nombre d'images à un ratio de 100 images accumulées pour une image de la séquence originale.

CHAPITRE 3

A FAST AND STABLE FEATURE-AWARE MOTION BLUR FILTER

*Ce chapitre est une version revue et corrigée de l'article "*A Fast and Stable Feature-Aware Motion Blur Filter*" [11, 12], présenté en Technical Report NVIDIA et à la conférence High Performance Graphics 2014. Les auteurs sont Jean-Philippe Guertin, Morgan McGuire et Derek Nowrouzezahrai.*

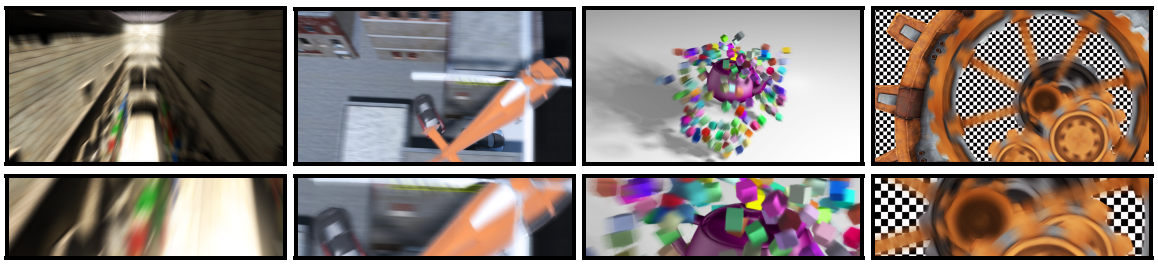


Figure 3.1 : We render temporally coherent motion blur without any motion artifacts, even on animation sequences with complex depth and motion relationships that are challenging for previous post-process techniques. All results are computed in under 2ms at 1280×720 on a GeForce GTX780, and our filter integrates seamlessly with post-process antialiasing and depth of field.

3.1 Introduction

Motion blur is an essential effect in realistic image synthesis, providing important motion cues and directing viewer attention. It is one of the few effects that distinguishes high-quality film production renderings from interactive graphics. We phenomenologically model the perceptual cues of motion blur sequences to approximate this effect with high quality using a simple, high-performance post-process.

We are motivated by work in offline motion blur post-processing [3, 5, 18], where the foundational work on combining reconstruction filters with sampling was set forth. These approaches are too heavyweight for modern game engines, however many of their ideas remain useful. Conversely, adhoc approaches (e.g., blurring moving objects) have existed in various forms for several years, however the question of how to approach motion blur post-processing using a phenomenological methodology has only recently gained traction in the interactive rendering community. This methodology had first found use (and success) for post-process antialiasing [20] and depth of field [6]. We target temporally-coherent, plausible high-quality motion blur that rivals super-sampling, but with a small performance budget and simple implementation. As such, we build upon recent post-processes [24, 41], resolving several of their limitations while producing stable, feature-preserving and plausible motion blur.

Contributions. Previous work relies on conservative assumptions about local velocity distributions at each pixel in order to apply plausible, yet efficient, blurs. Unfortunately, distracting artifacts arise when these assumptions are broken; this occurs when objects nearby in image-space move with different velocities, which is unavoidable even in simple scenes, or when objects under (relative) motion have geometric features of different scales. We eliminate these artifacts and compute stable motion blur for both simple and complex animation configurations. Our contributions are:

- an improved, variance-driven directional sampling scheme that handles anisotropic velocity distributions,
- a sample-weighting scheme that preserves unblurred object details and captures fine- and large-scale blurring, and
- a more robust treatment of tile boundaries, and interaction with post-process antialiasing and depth of field.

We operate on g-buffers, captured at a single time instance, with standard rasterization. Our results are stable under animation, robustly handle complex motion scenarios, and all in about 2ms per frame (Figure 3.1). Since ours is an image post-process filter, it can readily be used in an art-driven context to generate non-physical and exaggerated motion blur effects. We provide our full pseudocode in Appendix I.

3.2 Previous Work and Preliminaries

Given the extensive work on motion blur, we discuss recent work most related to our approach and refer interested readers to a recent survey on the topic by Navarro et al. [26].

Sampling Analysis and Reconstruction. Cook's seminal work on distribution effects [3] was the first to apply different (image) filters to reduce artifacts in the context of stochastic ray tracing and micropolygon rasterization. More recent approaches filter noise while retaining important visual details, operating in the multi-dimensional primal [13], wavelet [27] or data-driven [35] domains. Egan et al. [5] specifically analyze the effects of sample placement and filtering, in the frequency domain of object motion, and propose sheared reconstruction filters that operate on stochastically distributed spatio-temporal samples. Lehtinen et al. [18] use sparse samples in ray-space to reconstruct the anisotropy of the spatio-temporal light-field at a pixel and reconstruct a filtered pixel value for distribution effects. These latter two techniques aim to minimize integration error given fixed sampling budgets in stochastic rendering engines. We also reduce visible artifacts due to low-sampling rates, however we limit ourselves to interactive graphics pipelines where distributed temporal sampling is not an option and where compute budgets are on the order of milliseconds, not minutes.

Stochastic Rasterization. Recent work on extending triangle rasterization to support the temporal- and lens-domains [1], balances the advantages and disadvantages of GPU rasterization, stochastic ray-tracing, and modern GPU micropolygon renderers [7]. Here, camera-visibility and shading are evaluated at many samples in space-lens-time. Even with efficient implementations on conventional GPU pipelines [23], these approaches remain too costly for modern interactive graphics applications. Still, Shirley et al. [38] discuss image-space filters for plausible motion blur given spatio-temporal output from such stochastic multi-sample renderers. We are motivated by their phenomenological analysis of motion blur behavior and extend their analysis to more robustly handle complex motion scenarios (see Section 3.4).

Interactive Heuristics. Some approaches blur albedo textures prior to texture mapping (static) geometry [21] or extrude object geometry using shaders [43], however neither approach properly handles silhouette blurring, resulting in unrealistic motion blur. Max et Lerner [22] and Pepper [28] sort objects by depth, blur along their velocities, and compose the result in the final image, but this strategy fails when a scene invalidates the painter's visibility assumption. Per-pixel variants of this approach can reduce artifacts [32, 33], especially when image velocities are dilated prior to sampling [15, 40], however this can corrupt background details and important motion features when multiple objects are present.

Motivated by recent tile-based, *single blur velocity* approaches [19, 24, 41, 46] (see Section 3.3), we also dilate velocities to reason about large-scale motion behavior while sampling from the original velocity field to reason about the spatially-varying blur we apply. However, we additionally incorporate higher-order motion information and feature-aware sample weighting to produce results that are stable and robust to complex motion, effectively eliminating the artifacts present in the aforementioned "*single-velocity*" ap-

proaches. Specifically, we robustly handle:

- interactions between many moving and static objects,
- complex temporally-varying depth relationships,
- correct blurring regardless of object size or tile-alignment,
- feature-preservation for static objects/backgrounds.

We build atop existing state-of-the-art tile-based plausible motion blur approaches, detailed in Section 3.3, and present our more robust *multi-velocity* extensions in Section 3.4.

3.3 Tile-based Dominant Velocity Filtering Overview

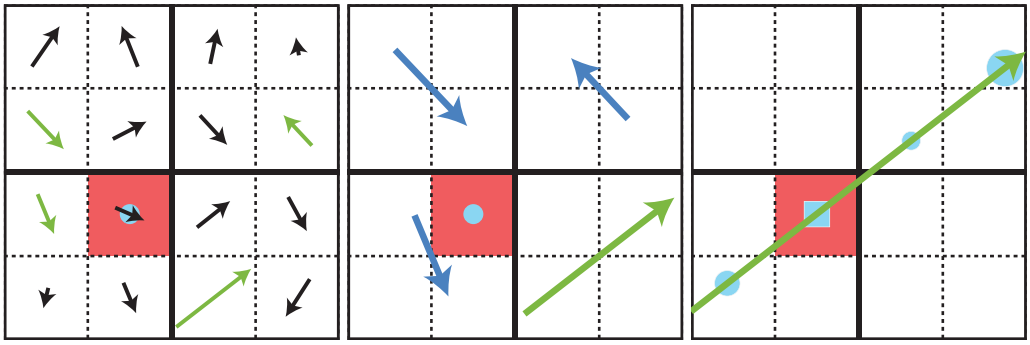


Figure 3.2 : Motion blur with "*single-velocity*" techniques.

We base our approach on recent **single-velocity** techniques that combine phenomenological observations with sampling-aware reconstruction filters [19, 24, 41, 46] (Figure 3.2) where: first, the image is split into $r \times r$ tiles for a maximum image-space blur radius r, ensuring that each pixel is influenced by at-most its (1-ring) tile neighborhood; second, each tile is assigned a *single* dominant neighborhood velocity; lastly, the pixel is blended with weighted taps along this dominant direction.

We review some notable details of this approach below:

- a tile's dominant velocity $\mathbf{v}_{\max}$ (Figure 3.2b; green) is computed in two steps: *maximum* per-pixel velocities are retained per-tile (Figure 3.2a,b; green, blue), and a 1-ring *maximum* of the per-tile maxima yields $\mathbf{v}_{\max}$,
- pixels are sampled *exclusively* along $\mathbf{v}_{\max}$, yielding high cache coherence but ignoring complex motions, and
- sample weighting uses a depth-aware metric that considers only velocity *magnitudes* at sample points.

The "TileMax" pass can be computed in two passes, one per image dimension, to greatly improve performance at the cost of a slight increase in memory usage. Samples are jittered to reduce (but not eliminate; see Section 3.4.5) banding, and samples are weighted (Figure 3.2c) to reproduce the following phenomenological motion effects:

1. a distant pixel/object blurring over the shading pixel,
2. transparency at the shading pixel from its motion, and
3. the proper depth-aware combination of effects 1 and 2.

These techniques generate plausible motion blur with a simple, high-performance post-process and have thus already been adopted in production game engines; however, the single dominant velocity assumption, coupled with the sample weighting scheme, results in noticeable visual artifacts that limit their ability to properly handle: overlapping objects that move in different directions, tile boundaries, and thin objects (see Figures 3.3, 3.7, 3.10, 3.12 and 3.16).

We identify, explain, and evaluate new solutions for these limitations. We follow the well-founded phenomenological methodology established by *single-velocity* approaches and other prior work [19, 24, 38, 41, 46], and our technique maintains high cache coherence and parallelizable divergence-free computation for an equally simple and high-

performance implementation.

3.4 Stable and Robust Feature-Aware Motion Blur

We motivate our improvements by presenting artifacts in existing (single-velocity) approaches. We demonstrate clear improvements in visual quality with negligible additional cost. Furthermore, our supplemental video illustrates the stability of our solution under animation and on scenes with complex geometry, depth, velocity and texture.

3.4.1 Several Influential Motion Vectors

The *dominant velocity* assumption breaks down when a tile contains pixels with many different velocities, resulting in both an incorrect blur *inside* a tile and blur mismatches *between* tiles (see Figures 3.3 and 3.7). This can occur, for example, with rotating objects (see Figure 3.10), objects with features smaller than a tile (see Figure 3.12), or when the view and motion directions are (nearly) parallel (see Figure 3.7).

Specifically, by exploiting this assumption to reduce the sampling domain to 1D, *single-velocity* approaches weight samples along $\mathbf{v}_{\max}$ according only to *the magnitudes* of their velocities, and not the *directions*. This can result in overblurred shading when samples (that lie along the dominant direction) should otherwise not contribute to the pixel but are still factored into the sum, especially if they are moving quickly (i.e., have large velocity magnitudes).

To reduce these artifacts we sample along a second, carefully chosen direction. Moreover, we split samples between these two directions according to the variance in the neighborhood's velocities, while also weighting samples using a locally-adaptive metric based on the deviation of sample velocities from the blur direction. This scheme better resolves complex motion details and still retains cache coherence by sampling along

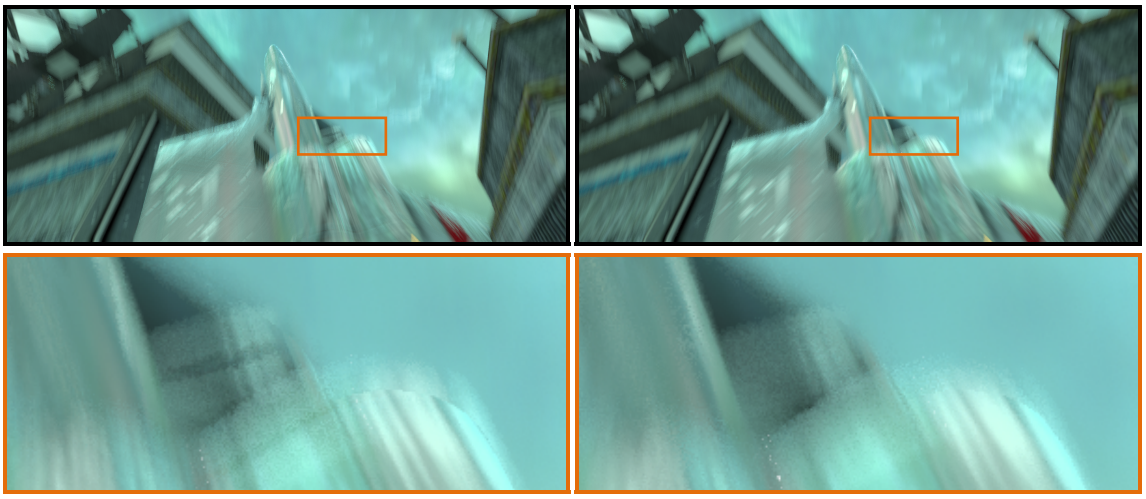
Single Direction Ours

Figure 3.3 : Top: an animation with complex depth and motion relationships. The camera is moving upwards and turning while the car is making a sharp turn and moving up. Bottom: previous approaches (left) cannot handle these cases, resulting in distracting artifacts between tiles; our filter (right) correctly blurs and is temporally stable (see video).

(fixed) 1D domains. Section 3.5 details our algorithm and we provide pseudocode in Appendix I.

Local Velocity Sampling. If a pixel's velocity differs significantly from the dominant direction, then it should also be considered during sampling. As such, we sample along both the pixel's velocity and the dominant velocity directions. This immediately improves the blur for scenes with complex pixel- and neighborhood-velocity relationships (e.g., Figure 3.3), however at the cost of increased noise since we are effectively halving the sampling rate in each 1D sub-domain.

If the pixel's velocity is small, this scheme effectively "wastes" all the samples placed along the second direction. In these cases, we replace the pixel's velocity with the



Single Direction Ours

Figure 3.4 : Complex scene with high velocities in various directions. Even at just 15 samples per pixel, our algorithm produces more pleasing results than previous algorithms.

velocity *perpendicular* to the dominant direction, sampling along this new vector for half of the total samples (and in the dominant direction for the other half). This helps in scenarios where the dominant velocity entirely masks smaller velocities with different directions in the neighborhood. Here, the perpendicular direction serves as a “best-guess" to maximize the probability of sampling along an otherwise ignored, important direction. We also compute this perpendicular direction such that the angle between it and the velocity at the pixel is always less than π, inverting its direction if necessary. If no such important (albeit secondary) direction exists, then samples placed along this perpendicular direction are (at worst) also “wasted”. This choice was motivated by the appearance of more visible artifacts due to “missed” samples (which these perpendicular samples greatly reduce) compared to the added noise from the unused samples. In more complex scenes, such as Figure 3.4, the number of wasted samples and their effect is vastly smaller, even at low sampling rates.

We ultimately combine the ideas of pixel ($\mathbf{v}(\mathbf{p})$) and perpendicular-dominant ($\mathbf{v}_{max}(\mathbf{t})$) velocities at the shading pixel $\mathbf{p}$'s tile $\mathbf{t}$: we place a number (see below) of samples along the *center direction*, interpolating between the normalized directions of $\mathbf{v}(\mathbf{p})$ and $\mathbf{v}^{\perp}_{max}(\mathbf{t})$ as the pixel's velocity diminishes past a minimum user threshold γ (see Figure 3.5),

$$\mathbf{v}_c(\mathbf{p}) = \text{lerp}\left(\mathbf{v}(\mathbf{p}), \mathbf{v}^{\perp}_{max}(\mathbf{t}), (\|\mathbf{v}(\mathbf{p})\| - 0.5)/\gamma\right) . \tag{3.1}$$

Sampling along both $\mathbf{v}_{max}(\mathbf{t})$ and $\mathbf{v}_c(\mathbf{p})$ ensures that each sample contributes usefully to the blur, better capturing complex motions. This approach remains robust to low (or zero, for static objects; see Figures 3.3 and 3.7) pixel velocities.

The number of samples we place along $\mathbf{v}_c$, and their weights (both along $\mathbf{v}_c$ and $\mathbf{v}_{max}$), can significantly impact the final appearance and quality. We address the problems of assigning samples to each direction, and weighting them, separately: first, we detail the sample distribution over the directions; later, we present a more accurate scheme for weighting their contributions. Our final sampling scheme is robust to complex scenes and stable under animation.

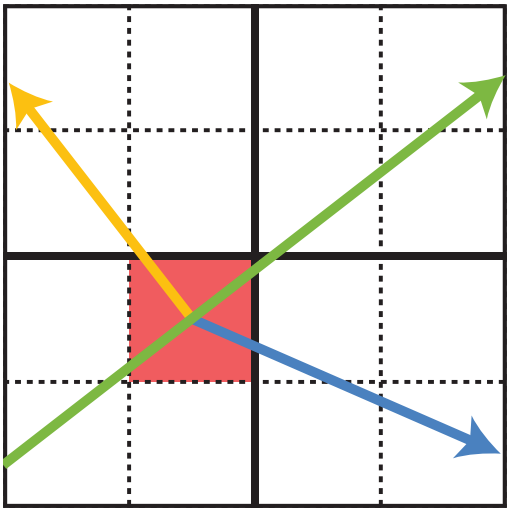


Figure 3.5 : Sampling directions $\mathbf{v}_{max}$ (green), $\mathbf{v}^{\perp}_{max}$ (yellow) and $\mathbf{v}_c(\mathbf{p})$ (blue).

Tile Variance for Sample Assignment.

We first address the segmentation and assignment of samples to $\mathbf{v}_c$ and $\mathbf{v}_{max}$. In cases where the dominant velocity assumption holds, we should sample *exclusively* from $\mathbf{v}_{max}$, as the standard *single-velocity* approaches (i.e., Lengyel [19], McGuire

et al. [24], Sousa [41], Zioma et Green [46]) do; however, it is rare that this assumption holds *completely* and we would like to find the ideal assignment of samples to the two directions in order to simultaneously capture more complex motions while reducing noise. This amounts to minimizing the number of "wasted" samples. To do so, we propose a simple variance estimation metric.

At each tile, we first compute the angular variation between a tile's and its neighbor's maximum velocities as

$$\upsilon(\mathbf{t}) = 1 - \frac{1}{|\mathscr{N}|} \sum_{\bar{\mathbf{t}} \in \mathscr{N}} \text{abs}\left[\mathbf{v}_{\max}(\mathbf{t}) \cdot \mathbf{v}_{\max}(\bar{\mathbf{t}})\right] \ , \tag{3.2}$$

where $\mathscr{N}$ is the (1-ring) neighborhood tile set around (and including) $\mathbf{t}$. The absolute value prevents opposed-sign products from canceling themselves. The variance $0 \leq \upsilon \leq 1$ is larger when neighboring tile velocities differ from $\mathbf{v}_{\max}(\mathbf{t})$. As such, we can use $\upsilon(\mathbf{t})$ to determine the number of samples assigned to $\mathbf{v}_c$ and $\mathbf{v}_{\max}$ with: $\upsilon \times N$ assigned to $\mathbf{v}_c$ and $(1-\upsilon) \times N$ assigned to $\mathbf{v}_{\max}$ (Figure 3.6).

This metric works well in practice, reducing visual artifacts near complex motion and reverting to distributing fewer samples along $\mathbf{v}_c$ when the motion is simple; however, we require an extra (reduced-resolution) render pass (and texture) to compute (and store) υ, which is less than ideal for pipeline integration (albeit with negligible performance impact). While the variance-based sample assignment does improve the quality of the result, conservatively splitting our

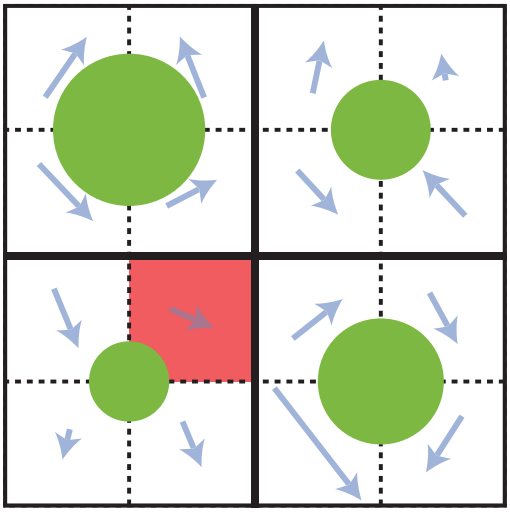


Figure 3.6 : Variance (green) for tiles neighboring pixel $\mathbf{p}$ (red).

samples evenly between $\mathbf{v}_c$ and $\mathbf{v}_{max}$ (i.e., $N/2$ samples for each) generates roughly equal quality results (see Figure 3.7).

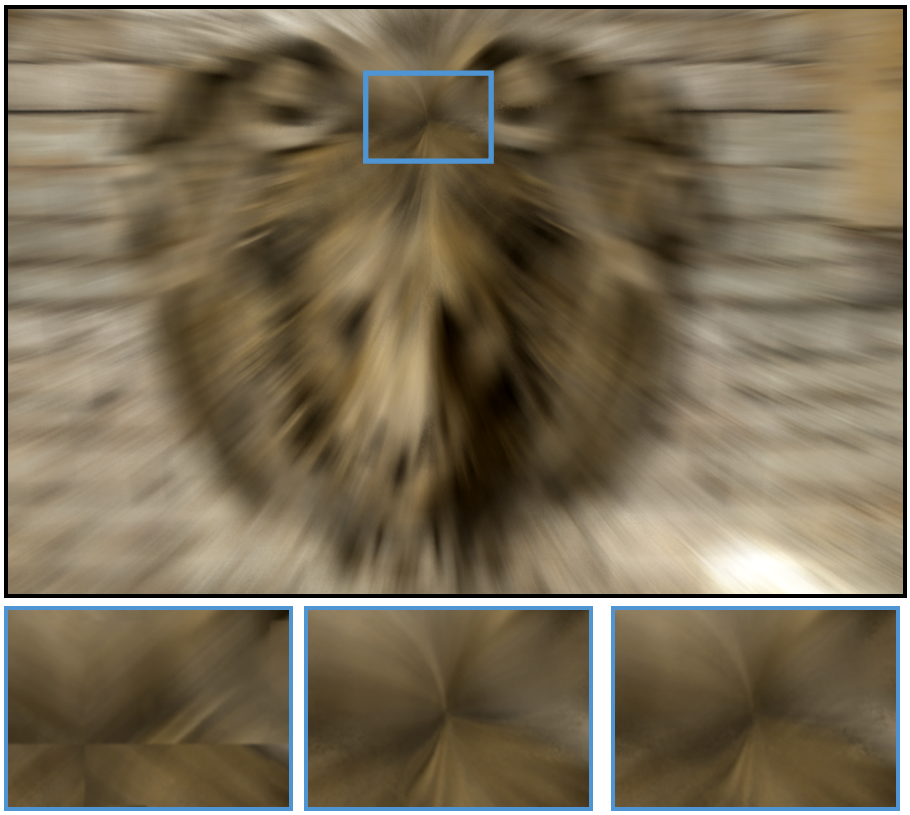


Figure 3.7 : The lion in Sponza moving directly towards the viewer. Bottom: *single-velocity* results (left), our variance-based sample distribution (middle), conservative sample distribution (right) both with $\mathbf{v}_{max}$ and $\mathbf{v}_c$ direction sampling.

We note, however, that the *weighting* of each sample plays a sizable role in both the quality of the final result and the ability to properly and consistently reconstruct complex motion blurs. We detail our new weighting scheme and contrast it to the scheme used in prior *single-velocity* approaches.

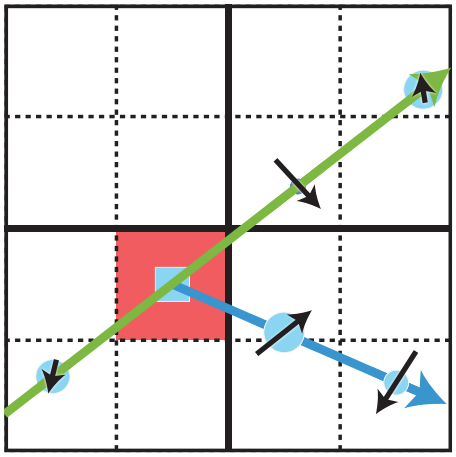


Figure 3.8 : Adapting per-sample weights according to the fine-scale velocity information.

Feature-Aware Sample Weights. *Single-velocity* approaches compute weights by combining two metrics:

- depth difference between the sample point and **p**, and
- the *magnitude* of the velocity at the sample point and at **p**.

This does not consider that samples can still fall on objects that move in directions different than $\mathbf{v}_{max}$ (and even $\mathbf{v}_c$). We instead additionally consider the velocity *direction* at samples when computing their weights. This yields a scheme that more appropriately adapts to fine-scale motions, without complicating the original scheme. Specifically, we will consider the dot product between blurring directions and sampled velocity directions. Recalling the three phenomenological motion blur effects in Section 3.3, we modify the weights corresponding to each component as follows:

1. the contributions of distant objects blurring onto **p**, along the sampling direction, are (additionally) weighted by the dot product between the sample's velocity and the sampling direction (either $\mathbf{v}_{max}$ or $\mathbf{v}_c$, as discussed earlier); here, the weight accounts for the color that blurs from the distant sample onto **p** and, as such, can be adjusted as the sample velocity $\mathbf{v_s}$ differs from the blurring direction,

2. the transparency caused by **p**'s blur onto its surrounding is (additionally) weighted by the dot product of its velocity (actually, $\mathbf{v}_c$) and the dominant blur direction $\mathbf{v}_{max}$; here, the total weight models the visibility of the background behind the pixel/object at **p** and, as such, if $\mathbf{v}_c$ differs from $\mathbf{v}_{max}$, the contribution should reduce appropriately, and
3. the *single-velocity* approaches include a correction term to model the combination of the two previous blurring effects and account for object edge discontinuities; we (additionally) multiply this weight by the *maximum* of the two dot products above; here, the maximum is a conservative estimate that may slightly oversmooth the discontinuities, if present.

These simple changes (see pseudocode in Appendix I) significantly improve quality within and between tiles, particularly when many motion directions are present (Figure 3.7).

3.4.2 Tile Boundary Discontinuities

The tile-based nature of *single-velocity* approaches, combined with their dependence on a single dominant velocity, often leads to distracting tile-boundary discontinuities when blur directions vary significantly between tiles (see Figure 3.10). This occurs when $\mathbf{v}_{max}$ differs significantly between adjacent tiles and, since the original approach samples *exclusively* along $\mathbf{v}_{max}$, neighboring tiles are blurred in completely different directions. Our sampling

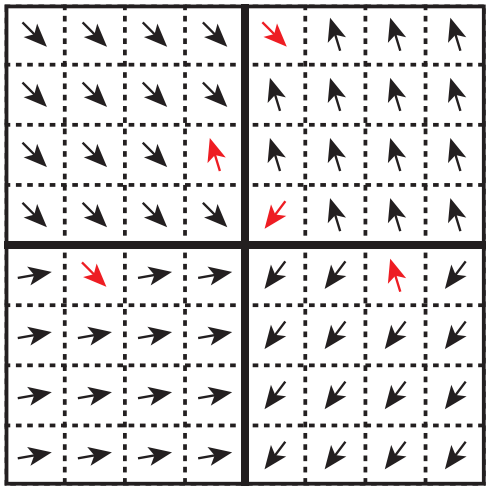


Figure 3.9 : We jitter the $\mathbf{v}_{max}(\mathbf{t})$ sample with higher probability closer to tile borders.

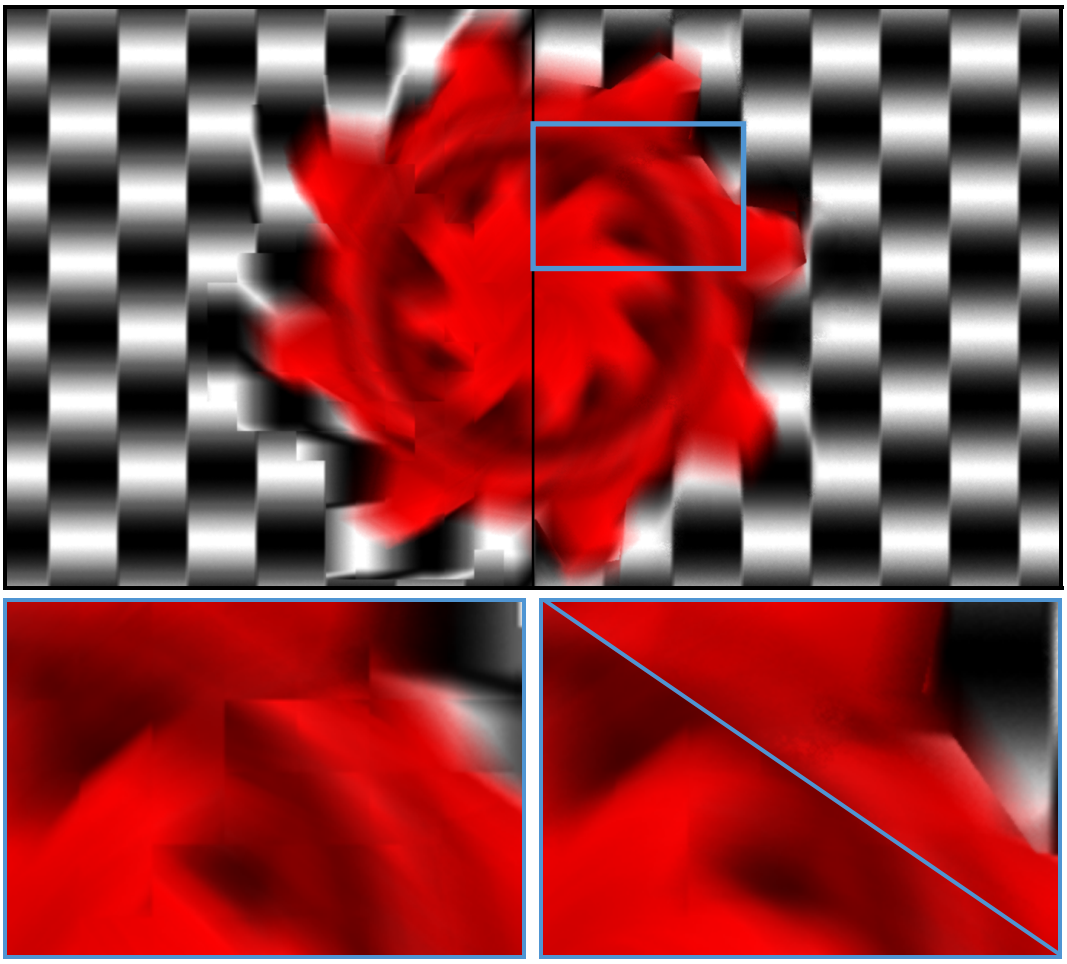
and weighting approaches (Section 3.4.1) already help to reduce this artifact, but we still sample from $\mathbf{v}_{max}$ (albeit not exclusively, and with different weights), and so we remain sensitive (to a lesser extent) to quick changes in $\mathbf{v}_{max}$ (see Figures 3.9 and 3.10).

It is important to note that, while seemingly intuitive, linearly interpolating neighborhood velocities along tile edges produces incorrect results. Indeed, not only is the interpolation itself ill-defined (e.g., interpolating two antiparallel velocities requires special care), but more importantly, two objects blurring on one another with different motion directions is not equivalent to blurring along the average of their directions. The latter causes distracting "wavy" artifacts and often exaggerates, as opposed to masking, tile boundaries.

To further reduce this artifact, we stochastically offset our `NeighborMax` maximum neighborhood velocity texture lookup for pixels near a tile edge. This trades banding for noise in the image near tile boundaries (i.e., edges in the $\mathbf{v}_{max}$ buffer). Thus, the probability of sampling along the $\mathbf{v}_{max}$ of a neighboring tile falls off as a pixel's distance to a tile border increases (Figure 3.9); we use a simple linear fall-off with a controllable (but fixed; see Section 3.5) slope τ.

3.4.3 Preserving Thin Features

Single-velocity filtering ignores the (jittered) mid-point sample closest to the pixel and instead explicitly weights the color at $\mathbf{p}$ by the *magnitude* of the $\mathbf{p}$'s velocity, without considering relative depth or velocity information (as with the remaining samples). This was designed to retain some of a pixel's original color regardless of the final blur, but it is not robust to scenes with thin features or large local depth/velocity variations. Underestimating this *center weight* causes thin objects to disappear or "ghost" (see Figure 3.12). Furthermore, the weight is not normalized and so its effect reduces as N



Single Direction Ours Without/With Blending

Figure 3.10 : Tile edge artifacts. Bottom: *single-velocity* blurring (left) results in disturbing tile-edge artifacts that are reduced, in part, using multi-direction sampling (right; bottom) and, in full, with stochastic $\mathbf{v}_{\max}$ blending (right; top).

increases. These artifacts are distracting and unrealistic, and the weight's dependence on N makes it difficult to control.

We instead set this *center weight* as $w_p = \|\mathbf{v}_c\|^{-1} \times \frac{N}{\kappa}$, where κ is a user-parameter to bias its importance. We use $\kappa = 40$ in all of our results (see Section 3.5 for all parameter settings). The second term in w_p serves as a pseudo-energy conservation normalization, making w_p robust to varying sampling rates N (unlike previous *single-velocity* approaches). Lastly, we do not omit the mid-point sample closest to $\mathbf{p}$, treating it just

as any other sample and applying our modified feature-aware sampling scheme (Section 3.4.1). As such, we also account for relative velocity variations at the midpoint, resulting in plausible motion blur robust to thin features and to different sampling rates (see Figure 3.12).

3.4.4 Neighbor Blurring

Both our approach and previous *single-velocity* approaches rely on an efficient approximation of the dominant neighborhood velocity $\mathbf{v}_{max}$. We present a modification to McGuire et al. [24]'s scheme that increases robustness by reducing superfluous blur artifacts present when the $\mathbf{v}_{max}$ estimate deviates from its true value. Specifically, the `NeighborMax` pass in McGuire et al. [24] conservatively computes the maximum velocity using the eight neighboring tiles (see Section 3.3 and Figure 3.2b), which can potentially result in an overestimation of the actual maximum velocity affecting the central tile.

We instead only consider a diagonal tile in the $\mathbf{v}_{max}$ computation if its maximum blur direction would in fact affect the current tile. For instance, if the tile to the top-left of the central tile has a maximum velocity that does not point towards the middle, it is not considered in the $\mathbf{v}_{max}$ computation (Figure 3.11). The reasoning is that slight velocity deviations at on-axis tiles can result in blurs that overlap the middle tile, however larger deviations are required at the corner (off-axis) tiles. The impact of this

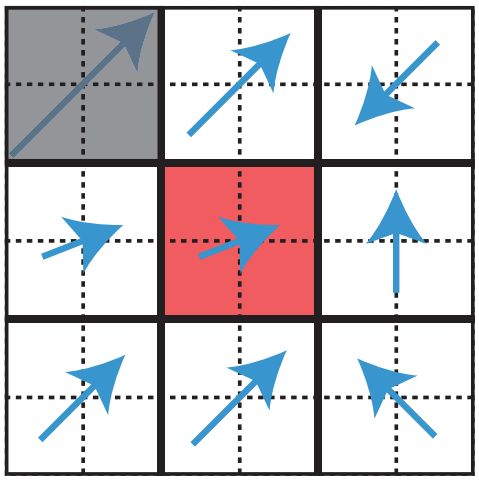


Figure 3.11 : Off-axis neighbors are only used for $\mathbf{v}_{max}$ computation if they may blur over the central tile.

modification is minor in practice due to the rare circumstances, but it still improves the worst case estimate.

3.4.5 Stochastic Noise and Post-Process Antialiasing

We have discussed how to choose the direction along which to place each sample (either $\mathbf{v}_{max}$ or $\mathbf{v}_c$) as well as how to weight them (Section 3.4.1), however we have not discussed *where* to place samples along the 1D domain. Numerical integration approaches are sensitive to sample distributions and, in the case of uniform samples distributed over our 1D domain, the quality of both ours and the *single-velocity* techniques can be significantly influenced by this choice. McGuire et al. [24] use equally-spaced uniform samples and jitter the pattern at each pixel using a hashed noise texture. This *jittered uniform* distribution has been recently analyzed in the context of 1D shadowing problems with linear lights [30] where it was proven to reduce variance better than low-discrepancy and stratified sampling.

We modify this strategy for motion blur integration and use a deterministic Halton sequence [45] to jitter the per-pixel sample sets with a larger maximum jitter value η (in pixel units), which improves the quality of the results (any other low-discrepancy set could be used). Furthermore, as discussed earlier, we do not ignore the central sample closest to $\mathbf{p}$ and, as such, properly take its relative depth and local-velocity into account during weighting. Finally, we reduce the jitter value depending on the sampling rate, which reduces noise at higher N, without causing any banding, by multiplying η by ϕ/N, where ϕ is a user-determined constant that affects the “baseline" jitter level.

Noise Patterns and Post-Process Antialiasing. The noise patterns produced by our stochastic integration are well-suited as input to post-processed screen-space antialiasing methods, such as FXAA [20]. Specifically, per-pixel jitter ensures that wherever there

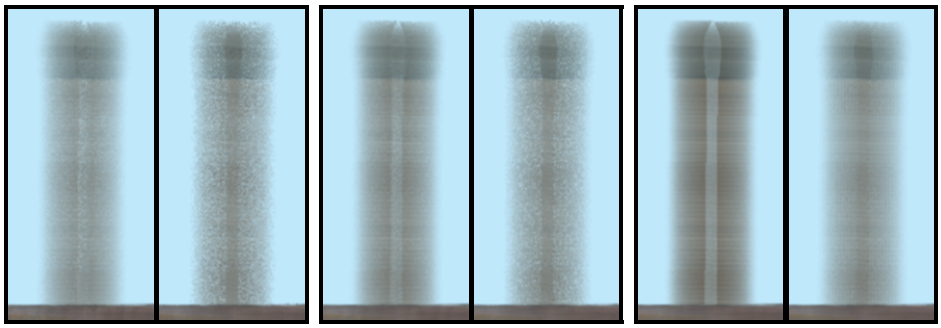


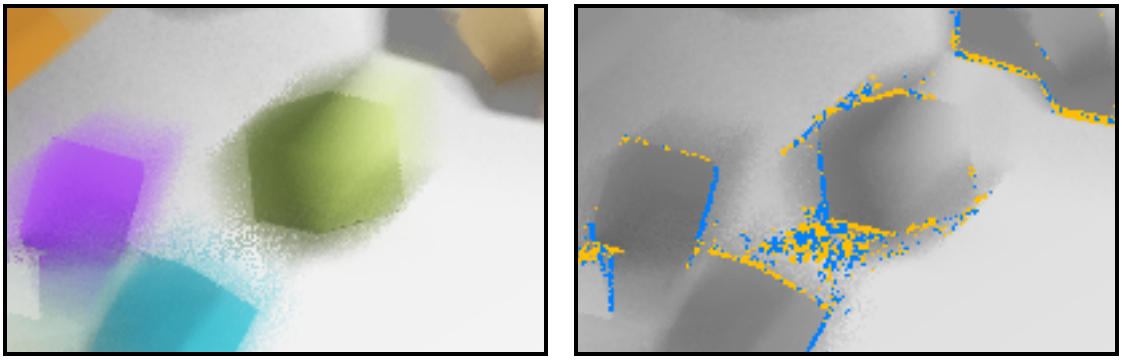
Figure 3.12 : Thin objects like the flagpole in Sponza ghost, to varying degrees depending on the sampling rate, with the *single-velocity* approach. Our approach resolves these details and is robust to changes in the sampling rate.

is residual noise, it appears as a high-frequency pattern that triggers the antialiasing luminance edge detector (Figure 3.13, top right). The post-process antialiasing then blurs each of these pixels (Figure 3.13, bottom left), yielding a much smoother result than the unfiltered image (Figure 3.13, top left). Stochastic $\mathbf{v}_{\max}$ blending (Section 3.4.2) is compatible with this effect.

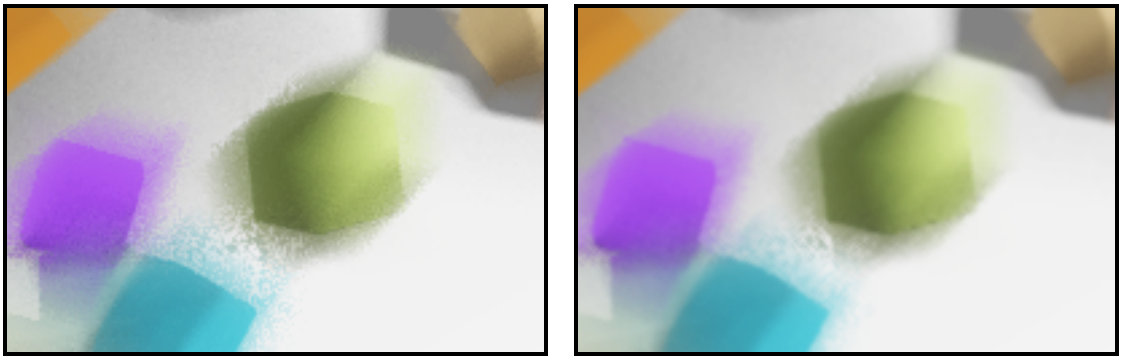
Moreover, it is possible to maximize the noise-smoothing properties of post-processed antialiasing by feeding a maximum-intensity, pixel-sized checkerboard to the edge detector's luminance input. This causes the antialiasing filter to detect edges on all motion blurred pixels, thus suppressing residual motion blur sampling noise (Figure 3.13, bottom right). While this scheme could be improved, for instance by only blurring in the direction of motion, it has the significant advantage of being a trivial modification that leverages modern rendering pipelines (where post-processed antialiasing is common) with negligible performance cost.

3.5 Implementation and Results

All our results are captured live at 1280×720 (with the exception of the teapot scene at 1920×1080) on an Intel Core i7-3770K and an NVIDIA GTX780. We recommend that readers digitally zoom-in on our (high-resolution) results to note fine-scale details. Our performance results are detailed in Table 3.I. We use the same parameter settings for all our scenes: $\{N, r, \tau, \kappa, \eta, \gamma, \phi\} = \{35, 40, 1, 40, 0.95, 1.5, 27\}$. All intermediate textures, `TileMax`, `NeighborMax` [24] and `TileVariance` (Section 3.4.1), are stored in `UINT8` format and sampled using nearest neighbor-interpolation, except for `TileVariance` that required a bilinear interpolant to eliminate residual tile boundary artifacts.



No AA | FXAA Edges



FXAA | FXAA With Checkerboard

Figure 3.13 : Our noise is well suited for standard post-process FXAA edge-detection, and we can further "hint" FXAA using a pixel-frequency luminance checkerboard.

All given parameters are relatively flexible, though we recommend leaving all but N and r as is. It is generally best to keep N at or above 15 samples per pixel to avoid excessive noise, and r should be kept roughly under 100 pixels, since a large r will sharply decrease the blur's accuracy with regards to motion direction and further exacerbate all velocity dilation issues. It is also a good idea to keep r as an integer divider of the screen resolution. A large τ will increase noise and provide decreasing returns in terms of eliminating banding, while a low τ will make tile edges more apparent. The value of κ directly influences the impact of the center sample (ie. the current position) and was balanced against the number of samples used during early development of the algorithm such that the division equalled 1. η should be kept under but close to 1, which avoids over-stretching the blur while decreasing the probability of a repeating pattern becoming visible. A high γ will cause an obvious and distracting spinning effect as the blur changes direction through linear interpolation, but a low γ will cause snapping between the two directions. Finally, ϕ directly controls the amount of jitter and should be altered carefully to avoid unnecessary noise or banding.

We note the importance of properly quantizing and encoding the per-pixel velo-

N	GTX780		GT650M (battery)	
	720p	1080p	720p	1080p
19	1.11 (0.93)	1.96 (2.06)	6.32 (5.79)	14.15 (13.00)
27	1.41 (1.19)	3.17 (2.71)	9.19 (8.02)	20.59 (17.99)
35	1.68 (1.52)	3.77 (3.44)	11.59 (10.25)	25.97 (23.06)
47	2.18 (2.02)	4.91 (4.54)	15.26 (13.61)	34.18 (30.52)

Tableau 3.I : Our performance results (in miliseconds) across resolutions, sample counts, and platforms in the Sponza scene. The camera is undergoing translation and all pixels are covered by the scene in motion. Parenthetical values are from our single-direction reference implementation.

city when using integer buffers for storage (as is typically done in, e.g., game engines). Specifically, a limitation in the encoding used in McGuire et al. [24]'s implementation, $\texttt{V[x,y]} = \mathbf{v}(\mathbf{p}_{x,y})/2r + 0.5$, is that the x and y velocity components are clamped *separately* to $\pm r$, causing large velocities to only take on one of four possible values: $(\pm r, \pm r)$. We instead used a similar computation to that of the article, which normalizes according to the range $[-r, r]$, with $\varepsilon = 10^{-5}$ and the exposure time E in seconds:

$$\texttt{V[x,y]} = \frac{\mathbf{v}(\mathbf{p}_{x,y})}{2r} \times \frac{\max\big(\min\big(|\mathbf{v}(\mathbf{p}_{x,y})| \times E, r\big), 0.5\big)}{|\mathbf{v}(\mathbf{p}_{x,y})| + \varepsilon} + 0.5 \ .$$

We modify the *continuous depth comparison* function (`zCompare`) used by McGuire et al. to better support depth-aware fore- and background blurring as follows: instead of using a hard-coded, *scene-dependent* depth-transition interval, we use the relative depth interval

$$\texttt{zCompare}[z_a, z_b] = \min\left(\max\left(0, 1 - \frac{(z_a - z_b)}{\min(z_a, z_b)}\right), 1\right) \ ,$$

where z_a and z_b are both depth values. This relative test has important advantages: it works on values in a scene-independent manner, e.g., comparing objects at 10 and 20 z-units will give similar results to objects at 1000 and 2000 z-units. This allows smooth blending between distant objects, compensating for their reduced on-screen velocity coverage, all while remaining robust to arbitrary scene scaling.

Motivated by Sousa [41]'s endorsement of McGuire et al. [24]'s *single-velocity* implementation, which has already been used in several game engines, all our comparisons were conducted against an optimized version of the open source implementation provided by McGuire et al. [24], with both our Halton jittering scheme and our velocity encoding. While the variance-based υ metric for distributing samples between $\mathbf{v}_{\max}$ and

$\mathbf{v}_c$ yields slightly improved results over, e.g., a 50/50 sample distribution, we observed no perceptual benefit in using υ during interactive animation; as such, we disabled this feature in all our results (except Figure 3.7).

Post-Process Depth of Field. Earlier, we discussed our approach's integration with FXAA, a commonly used post-process in modern game engines. Another commonly used post-process effect is depth of field (DoF); however, the combination of DoF and

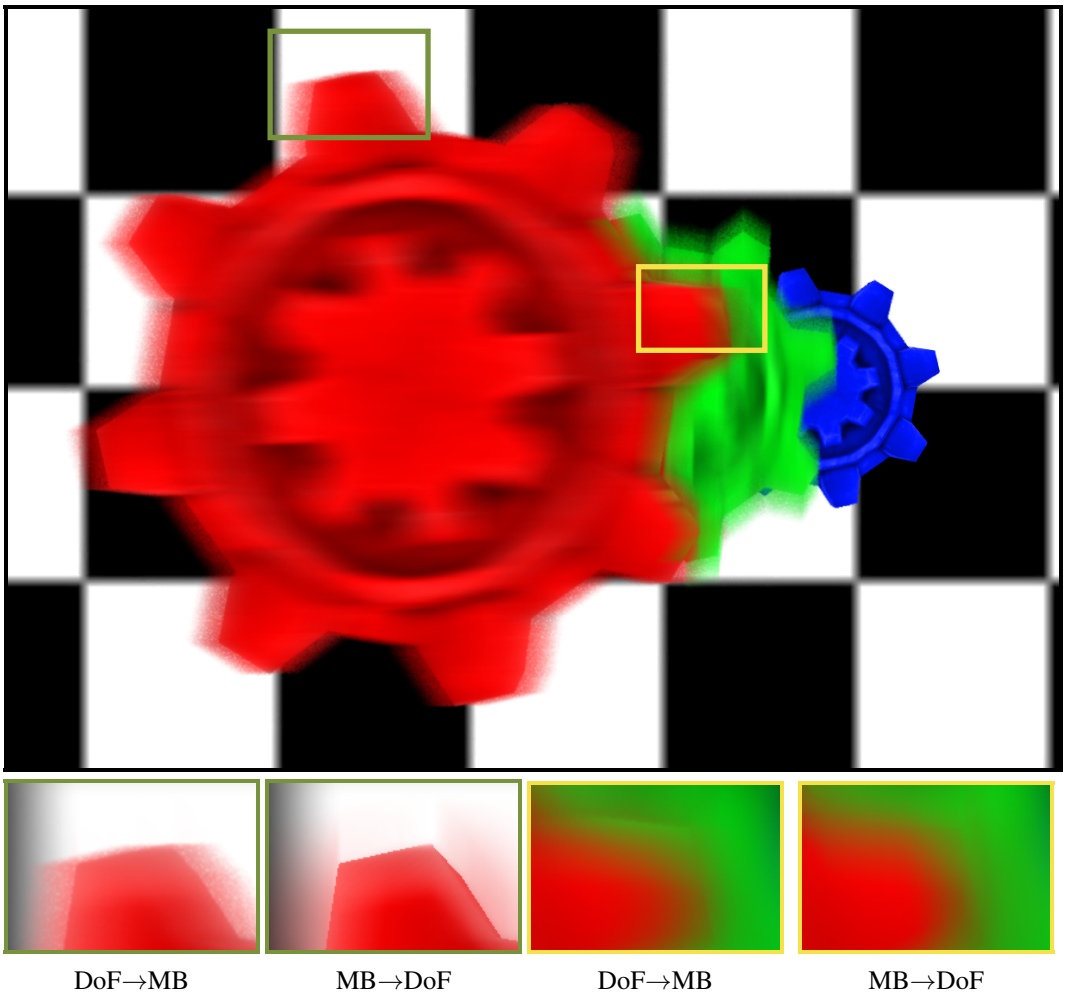


Figure 3.14 : Depth of field (DoF) and motion blur (MB) stress test. Top: MB only. Bottom: Comparison of motion blur and depth of field ordering. We note slightly better results when DoF is applied after MB.

motion blur post-processes is a subject of little investigation. We implemented the post-process DoF approach of Gilham in ShaderX5 [6] and briefly discuss its interaction with our motion blur filter. Specifically, the correct order in which to apply the two filters is not immediately apparent. We experimented with both options and illustrate our results in an especially difficult scene, as far as motion and DoF complexity are concerned: three distinctly colored wheels, each undergoing different translational motion, at three different depths, and with the focus on the green middle wheel (see Figure 3.14).

The differences are subtle and our experiments far from comprehensive, however we note (somewhat counter-intuitively) that applying DoF *after* motion blur yields fewer visible edges in blurred regions and sharper features on in-focus regions. Applying DoF after motion blur has the added benefit of further blurring our noise artifacts.

3.6 Limitations and Future Work

While our approach addresses many limitations from previous work, there remain several interesting avenues for future work.

3.6.1 Noise

The most prevalent remaining artifact in our technique is noise. In most circumstances, we explicitly chose to trade banding for noise. We outline some possible solutions worth investigating that are outside the scope of our work.

Single-velocity Fallback. In images dominated by a single coherent velocity, e.g., Figure 3.13, our approach uses roughly twice as many samples for equal image quality compared to previous work. Correcting this behavior by, for example, dynamically adjusting the sampling directions according to the velocity spread in a scene is one potential

avenue of future work. Our tile-variance experiments (Section 3.4.1) could be repurposed to inform the algorithm on the circumstances where the variance is low enough to only sample in one direction, as opposed to two. However, such adaptive sampling can have unexpected repercussions, particularly on the center sample weight: when half the samples are wasted, the total weight of the sum is effectively halved, but if those samples are dynamically adjusted such that none are wasted, the relative weight of the center sample is halved. Any adaptive scheme must therefore also adjust the central weight to compensate for variations in the total weight.

Adaptive Sampling Rate. Similarly speaking, adjusting the accumulation sample count, perhaps according to the magnitude of the velocity at the tile or pixel, would reduce the cost of the summation when the motion is relatively small, thus allowing larger motion to be oversampled. This could result in a uniform amount of noise throughout the image, whereas our current implementation is noisier in regions of larger velocity. Here, for performance purposes, it would be critical to prevent branching within the same GPU warp.

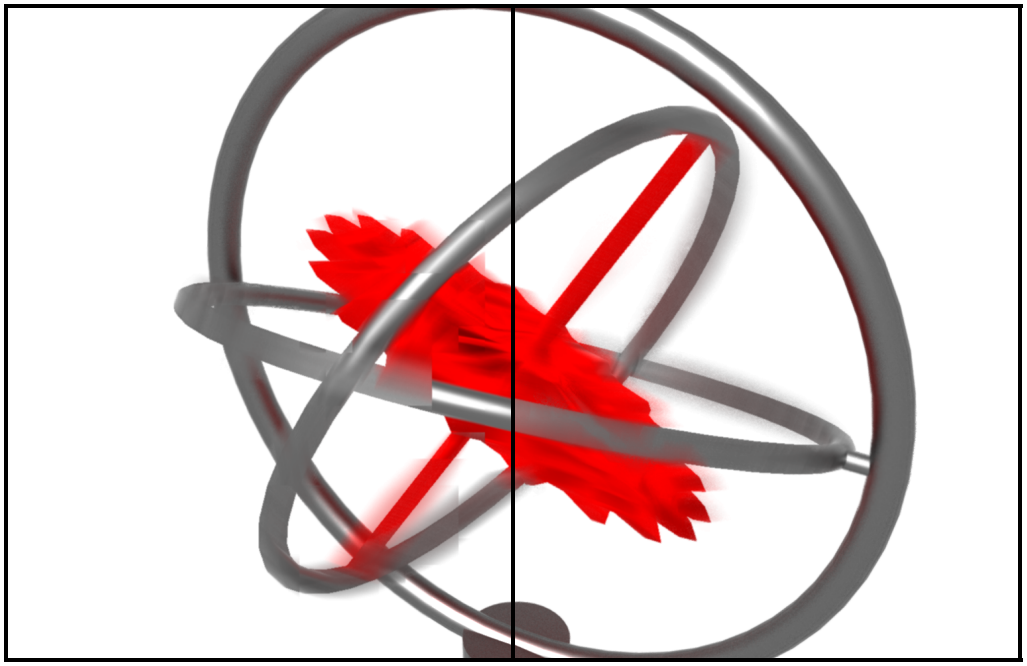
Filtering. We briefly investigate post-process FXAA filtering in Section 3.4.5, however a post-process antialiasing filter designed from scratch for motion blur, e.g., one that re-blurs in the direction of motion (as in Sousa [39]), could potentially help reduce the sampling rate in each pass.

3.6.2 Other Limitations

Acceleration to/from Rest. Albeit less common a scenario, a distracting visual artifact can appear under very long camera exposures: an object under acceleration will blur beyond its initial location because our approach assumes constant object velocity.

Velocity Interpolation. We interpolate velocities in order to smooth transitions during motion. For slow transitions, the effect can be visually distracting as the blur begins in a direction that is completely unrelated to the motion.

Non-linear Motion. As with previous work, we assume linear motion. While this assumption often holds, extreme rotational motion or complex deformation (e.g., Figure 3.15), especially with long exposure times, may result in artifacts.



Single Direction Ours

Figure 3.15 : Rotation stress test: multiple overlapping rotational velocities are a challenging scenario.

3.6.3 Future Work

Time Component. As mentioned earlier, our algorithm, while stable across frames, does not explicitly treat the temporal dimension. Doing so could have multiple advantages, e.g., using previous frame data to reduce noise in the current frame, but would also introduce several complex issues when involving advanced animation that do not follow predictable, linear motion paths. An explicit time component would also allow for acceleration-aware filtering, which could correct overblurring in such situations.

Shadows. As a screen-space post-process, our algorithm is oblivious to any shadowing techniques and their interactions with objects in the scene. An interesting open problem is the proper incorporation of blurred shadows according to the motion of blockers and light sources.

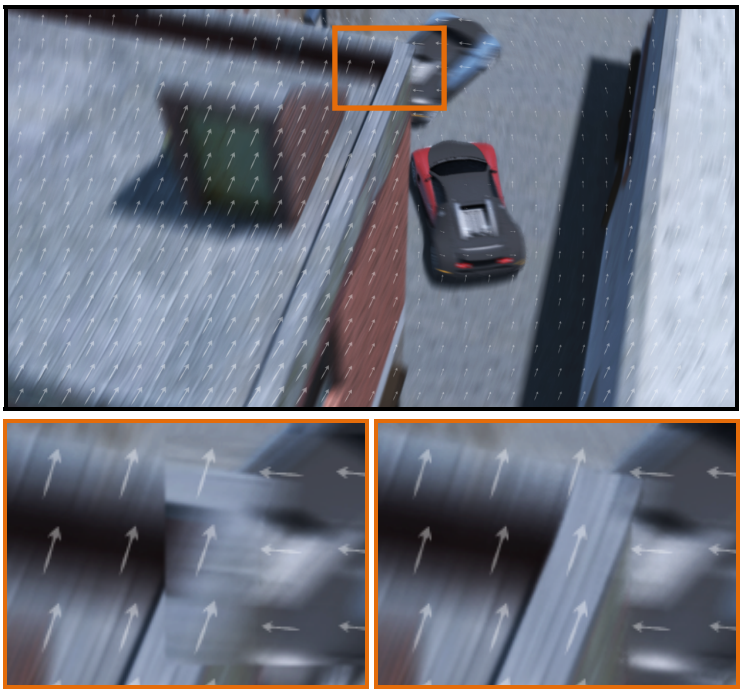
Simultaneous Depth of Field. Combining motion blur and depth of field into a single algorithm is not a new idea and has been explored by, e.g., McGuire et al. [23]. While we briefly investigate the interaction between depth of field and motion blur, we do not attempt to combine both effects into a single algorithm.

3.7 Conclusion

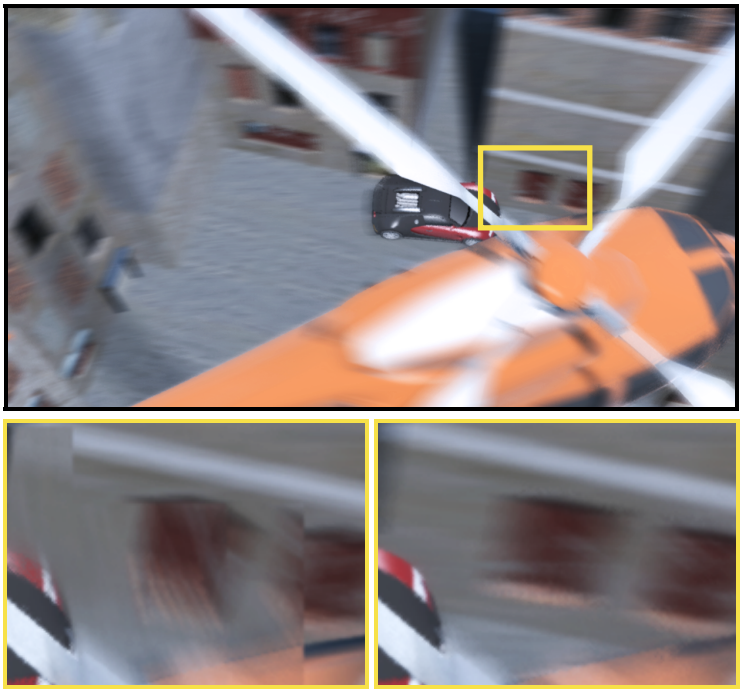
We presented a high-performance post-process motion blur filter that is robust to scenes with complex inter-object motions and fine-scale details. We demonstrated our approach on several such scenes with clear quality benefits, both in images and in animations (see the supplemental video), and at a negligible cost compared to the state-of-the-art. Our approach is temporally coherent, easy to integrate, and readily compatible with other commonly used post-process effects.

Acknowledgments

We thank the reviewers of the paper for their helpful comments. This work was supported by NSERC Discovery and FQRNT New Researcher Grants (Nowrouzezahrai), as well as an FQRNT MSc Scholarship (Guertin). We also thank NVIDIA for equipment donations. The NeighborMax tile culling algorithm was contributed by Dan Evangelakos at Williams College. Observations about FXAA, the use of the luminance checkerboard, and considering the central velocity in the gather kernel are due to our colleagues at Vicarious Visions: Padraic Hennessy, Brian Osman, and Michael Bukowski.



Single Direction Ours



Single Direction Ours

Figure 3.16 : This scene has excellent examples of motion vector dilation causing incorrect blurring when only a single velocity direction is used. Note in particular the car's motion bleeding over the building in the first image, and the obvious tile edges visible in the second image, where the rotor blades dominate the neighboring background motion.

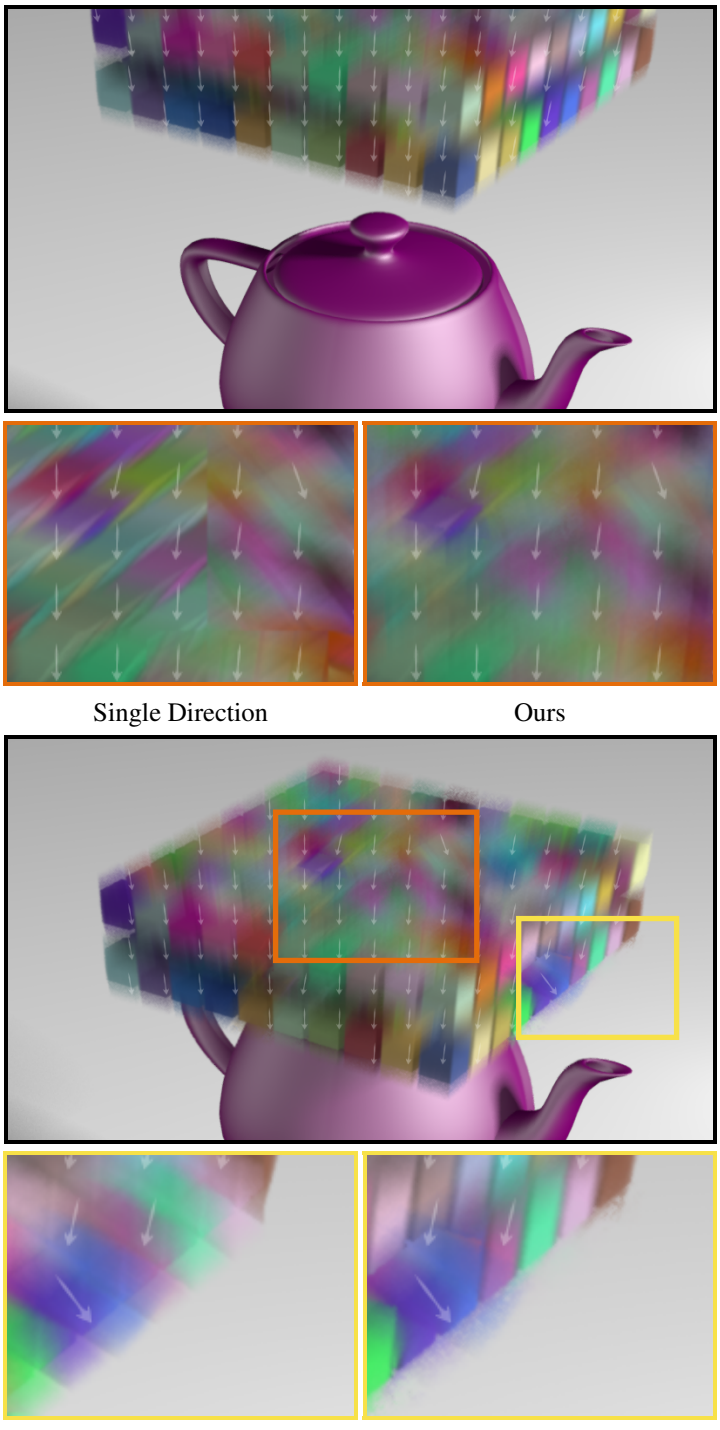
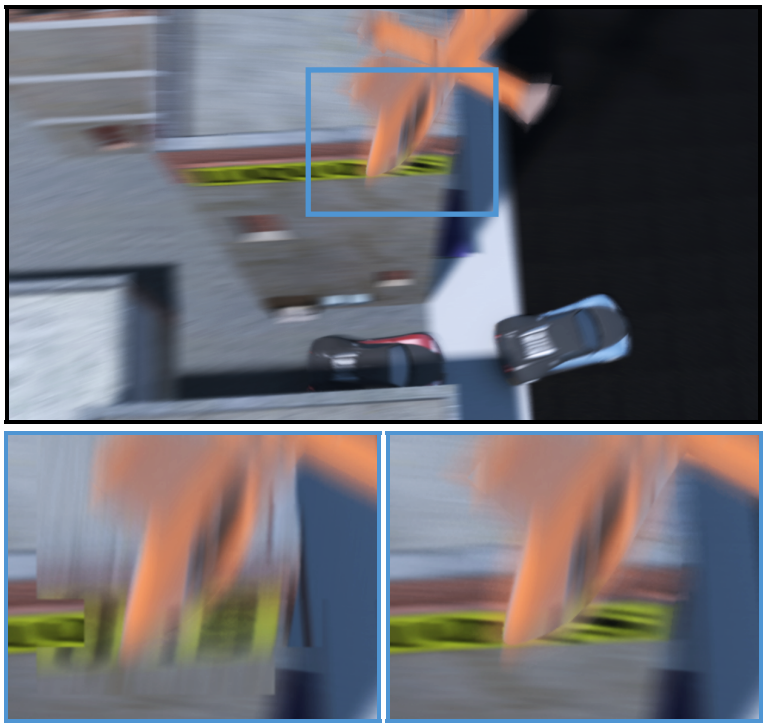


Figure 3.17 : This scene is interesting in that almost all of the motion is in a single direction, but a few cubes have already hit the teapot below, causing their velocity to differ from the neighboring cubes. Their velocity tends to dominate their neighborhood, causing large areas of the image to blur in the wrong direction, when only a few pixels are actually affected by the motion.



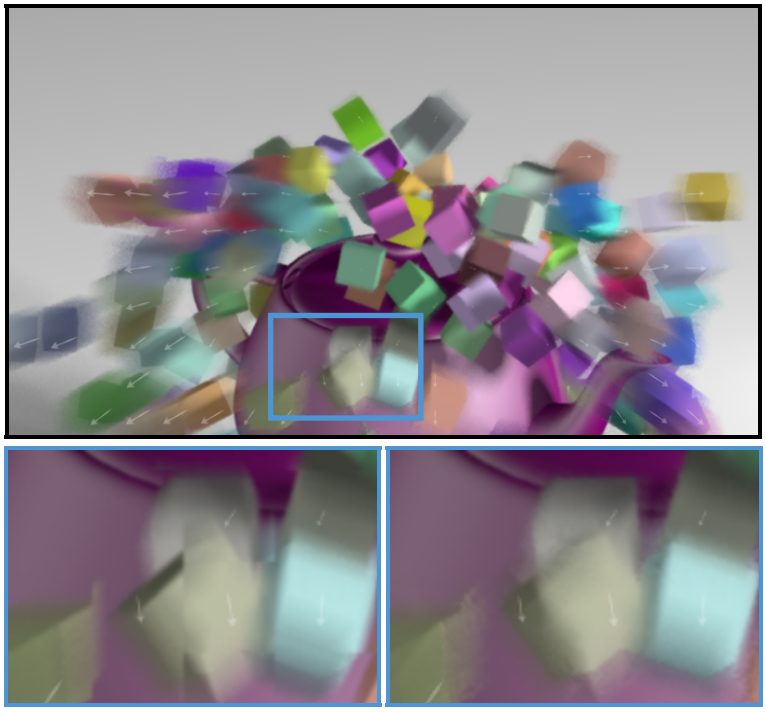


Figure 3.18 : Both images highlight the disadvantage of naive tile-based dilation algorithms, whereby the tile edges can often be visible due to great differences between neighboring tiles.

Figure 3.19 : This complex scene highlights how our algorithm performs very well even in the presence of numerous objects with varying velocity directions and magnitudes.

CHAPITRE 4

CONCLUSION

Ce mémoire s'est intéressé au flou de mouvement, plus spécifiquement à une méthode efficace permettant de générer du flou de mouvement pour des applications temps réel. La première section a effectué une courte revue de littérature donnant une vue d'ensemble des techniques précédentes pour la génération de flou de mouvement dans une variété de cadres d'applications et de contraintes. Par la suite, l'état de l'art a été décrit en plus de détails pour mieux situer les contributions présentées dans le troisième chapitre. Dans celui-ci, nous avons présenté un nouvel algorithme produisant un flou de mouvement plus efficace et plus précis dans un nombre de situations beaucoup plus important sans pour autant affecter la performance globale comparativement aux techniques précédentes.

4.1 Travaux futurs

Outre les nombreuses avenues de travaux futurs déjà mentionnés dans l'article lui-même (voir section 3.6), il serait intéressant d'effectuer des investigations plus poussées sur le domaine de la rastérisation stochastique et, plus généralement, sur les techniques de filtrage stochastiques qui ont grandement gagné en popularité ces dernières années. Le rendu stochastique offre des possibilités excitantes pour le rendu efficace d'effets distribués dans un contexte interactif, et les avancées continuelles en performances du matériel graphique rendent ces techniques de plus en plus attirantes.

Bien que non mentionné dans ce mémoire, nous avons poursuivi nos investigations dans d'autres algorithmes de flou de mouvement et nos résultats préliminaires sont pro-

metteurs. Nous nous sommes penchés sur le problème beaucoup plus complexe du rendu de flou de mouvement non-linéaire : au lieu d'assumer que les vélocités peuvent être représentées à l'aide d'un seul vecteur, nous les stockons sous des formes dites d'ordres supérieures, par exemple des courbes de Bézier de degré deux. Cette avenue de recherche reste peu explorée dans le corpus des techniques de flou de mouvement en temps réel ; soit la technique prend en charge des mouvements de complexité arbitraire (par exemple, le rendu stochastique), soit seuls les mouvements linéaires sont supportés. Une technique servant d'entre-deux pourrait donc être attrayante, particulièrement dans des domaines où la fidélité du mouvement est plus importante, par exemple la prévisualisation cinématographique, mais où une performance interactive est nécessaire.

Finalement, nous avons plusieurs possibilités d'optimisations et d'améliorations pour l'algorithme tel que présenté, par exemple de prendre avantage des phases intermédiaires pour stocker la vitesse maximale et minimale à chaque tuile et voir quelles nouvelles possibilités apparaissent lorsque cette nouvelle donnée est disponible.

BIBLIOGRAPHIE

[1] Tomas Akenine-Möller, Jacob Munkberg et Jon Hasselgren. Stochastic rasterization using time-continuous triangles. Dans *Graphics Hardware*, pages 7–16. Eurographics, 2007. ISBN 978-1-59593-625-7.

[2] Mike Cammarano et Henrik Wann Jensen. Time dependent photon mapping. Dans *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, pages 135–144, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. ISBN 1-58113-534-3. URL `http://dl.acm.org/citation.cfm?id=581896.581915`.

[3] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5 (1):51–72, 1986.

[4] Robert L. Cook, Thomas Porter et Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, janvier 1984. ISSN 0097-8930. URL `http://doi.acm.org/10.1145/964965.808590`.

[5] Kevin Egan, Yu-Ting Tseng, Nicolas Holzschuch, Frédo Durand et Ravi Ramamoorthi. Frequency Analysis and Sheared Reconstruction for Rendering Motion Blur. *SIGGRAPH (ACM Transactions on Graphics)*, 28(3):93 :1–93 :13, 2009.

[6] Wolfgang Engel. *Shader X5 : Advanced Rendering Techniques*. Charles River Media, MA, USA, 2006. ISBN 1584504994.

[7] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark et Pat Hanrahan. Data-parallel rasterization of micropolygons with defo-

cus and motion blur. Dans *High Performance Graphics*, pages 59–68. ACM, 2009. ISBN 978-1-60558-603-8.

[8] Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič et Philipp Slusallek. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, 31 (6):192 :1–192 :10, novembre 2012. ISSN 0730-0301. URL http://doi.acm.org/10.1145/2366145.2366211.

[9] A. Glassner. An open and shut case [computer graphics]. *IEEE Computer Graphics and Applications*, 19(3):82–92, May 1999. ISSN 0272-1716.

[10] Xin Guan et Klaus Mueller. Point-based surface rendering with motion blur. Dans *Proceedings of the First Eurographics Conference on Point-Based Graphics*, SPBG'04, pages 33–40, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. ISBN 3-905673-09-6. URL http://dx.doi.org/10.2312/SPBG/SPBG04/033-040.

[11] Jean-Philippe Guertin, Morgan McGuire et Derek Nowrouzezahrai. A fast and stable feature-aware motion blur filter. Rapport technique NVR-2013-003, NVIDIA Corporation, November 2013. URL http://graphics.cs.williams.edu/papers/MotionBlur13.

[12] Jean-Philippe Guertin, Morgan McGuire et Derek Nowrouzezahrai. A fast and stable feature-aware motion blur filter. Dans *High Performance Graphics*. ACM/Eurographics, June 2014. URL http://graphics.cs.williams.edu/papers/MotionBlurHPG14.

[13] Toshiya Hachisuka, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker et Henrik Wann Jensen. Multidimensional adaptive

sampling and reconstruction for ray tracing. *ACM Trans. Graph.*, 27(3):33 :1–33 :10, août 2008. ISSN 0730-0301. URL http://doi.acm.org/10.1145/1360612.1360632.

[14] Henrik Wann Jensen. Global illumination using photon maps. Dans *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag. ISBN 3-211-82883-4. URL http://dl.acm.org/citation.cfm?id=275458.275461.

[15] Nickolay Kasyan et Nicolas Schulz. Secrets of cryengine 3 graphics technology. Dans *SIGGRAPH Talks*. ACM, 2011.

[16] Jonathan Korein et Norman Badler. Temporal anti-aliasing in computer generated animation. *SIGGRAPH Comput. Graph.*, 17(3):377–388, juillet 1983. ISSN 0097-8930. URL http://doi.acm.org/10.1145/964967.801168.

[17] Eric P. Lafortune et Yves D. Willems. Bi-directional path tracing. Dans *Proceedings of third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93*, pages 145–153, 1993.

[18] Jaakko Lehtinen, Timo Aila, Jiawen Chen, Samuli Laine et Frédo Durand. Temporal light field reconstruction for rendering distribution effects. *ACM Trans. Graph.*, 30(4):55 :1–55 :12, juillet 2011. ISSN 0730-0301. URL http://doi.acm.org/10.1145/2010324.1964950.

[19] Eric Lengyel. Motion blur and the velocity-depth-gradient buffer. Dans Eric Lengyel, éditeur, *Game Engine Gems 1*, pages 235–247. Jones and Bartlett, 2010.

[20] Timothy Lottes. Fast Approximate Anti-Aliasing (FXAA). 2009. URL

http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

[21] Joern Loviscach. Motion blur for textures by means of anisotropic filtering. Eurographics Symposium on Rendering, pages 105–110, Aire-la-Ville, Switzerland, 2005. ISBN 3-905673-23-1.

[22] Nelson L. Max et Douglas M. Lerner. A two-and-a-half-d motion-blur algorithm. *SIGGRAPH Comput. Graph.*, 19(3):85–93, juillet 1985. ISSN 0097-8930. URL http://doi.acm.org/10.1145/325165.325188.

[23] Morgan McGuire, Eric Enderton, Peter Shirley et David P. Luebke. Real-time stochastic rasterization on conventional gpu architectures. Dans *Proceedings of the Conference on High Performance Graphics*, High Performance Graphics '10, pages 173–182, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association. URL http://dl.acm.org/citation.cfm?id=1921479.1921505.

[24] Morgan McGuire, Padraic Hennessy, Michael Bukowski et Brian Osman. A reconstruction filter for plausible motion blur. Dans *Interactive 3D Graphics and Games*, pages 135–142, 2012.

[25] Jacob Munkberg, Karthik Vaidyanathan, Jon Hasselgren, Petrik Clarberg et Tomas Akenine-Möller. Layered reconstruction for defocus and motion blur. *Computer Graphics Forum*, 33(4):81–92, 2014. ISSN 1467-8659. URL http://dx.doi.org/10.1111/cgf.12415.

[26] Fernando Navarro, Francisco J. Serón et Diego Gutierrez. Motion blur rendering : State of the art. *Computer Graphics Forum*, 30(1):3–26, 2011. ISSN 1467-8659.

[27] Ryan S. Overbeck, Craig Donner et Ravi Ramamoorthi. Adaptive wavelet rendering. *ACM Trans. Graph.*, 28(5):140 :1–140 :12, décembre 2009. ISSN 0730-0301. URL http://doi.acm.org/10.1145/1618452.1618486.

[28] Damyan Pepper. Per-pixel motion blur for wheels. Dans Wolfgang Engel, éditeur, *ShaderX6*, pages 175–188. Charles River Media, 2003.

[29] Michael Potmesil et Indranil Chakravarty. Modeling motion blur in computer-generated images. *SIGGRAPH Comput. Graph.*, 17(3):389–399, juillet 1983. ISSN 0097-8930. URL http://doi.acm.org/10.1145/964967.801169.

[30] Ravi Ramamoorthi, John Anderson, Mark Meyer et Derek Nowrouzezahrai. A theory of Monte Carlo visibility sampling. *ACM Trans. Graph.*, 31(5):121 :1–121 :16, septembre 2012. ISSN 0730-0301. URL http://doi.acm.org/10.1145/2231816.2231819.

[31] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, avril 1983. ISSN 0730-0301. URL http://doi.acm.org/10.1145/357318.357320.

[32] Matt Ritchie, Greg Modern et Kenny Mitchell. Split second motion blur. Dans *SIGGRAPH 2010 Talks*, NY, USA, 2010. ACM.

[33] Gilberto Rosado. Motion blur as a post-processing effect. Dans *GPU Gems 3*, pages 575–581. Addison-Wesley, 2008.

[34] Johannes Schmid, Robert W. Sumner, Huw Bowles et Markus Gross. Programmable motion effects. *ACM Trans. Graph.*, 29(4):57 :1–57 :9, juillet 2010. ISSN 0730-0301. URL http://doi.acm.org/10.1145/1778765.1778794.

[35] Pradeep Sen et Soheil Darabi. On filtering the noise from the random parameters in Monte Carlo rendering. *ACM Trans. Graph.*, 31(3):18 :1–18 :15, juin 2012. ISSN 0730-0301. URL http://doi.acm.org/10.1145/2167076.2167083.

[36] Roland V. Shack. The influence of image motion and shutter operation on the photographic transfer function. *Appl. Opt.*, 3(10):1171–1181, octobre 1964. URL http://ao.osa.org/abstract.cfm?URI=ao-3-10-1171.

[37] Clement Shimizu, Amit Shesh et Baoquan Chen. B. : Hardware accelerated motion blur generation. *Eurographics*, 22, 2003.

[38] Peter Shirley, Timo Aila, Jonathan D. Cohen, Eric Enderton, Samuli Laine, David P. Luebke et Morgan McGuire. A local image reconstruction algorithm for stochastic rendering. Dans *Interactive 3D Graphics and Games*, pages 9–14, 2011.

[39] Tiago Sousa. Crysis Next Gen Effects, February 2008. URL http://www.crytek.com/cryengine/presentations/crysis-next-gen-effects. Presentation at GDC 2008.

[40] Tiago Sousa. Cryengine 3 rendering techniques. Dans *Microsoft Game Technology Conference*. August 2011.

[41] Tiago Sousa. Graphics gems from cryengine 3. Dans *ACM SIGGRAPH Course Notes*, 2013.

[42] K. Sung, A. Pearce et C. Wang. Spatial-temporal antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):144–153, avril 2002. ISSN 1077-2626. URL http://dx.doi.org/10.1109/2945.998667.

[43] Natalya Tatarchuk, Chris Brennan et John R. Isidoro. Motion blur using geometry

and shading distortion. Dans Wolfgang Engel, éditeur, *ShaderX2 : Shader Programming Tips and Tricks with DirectX 9.0*, pages 299–308. Wordwave, 2003.

[44] Eric Veach et Leonidas J. Guibas. Metropolis light transport. Dans *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. URL `http://dx.doi.org/10.1145/258734.258775`.

[45] Tien-Tsin Wong, Wai-Shing Luk et Pheng-Ann Heng. Sampling with Hammersley and Halton points. *J. Graph. Tools*, 2(2):9–24, 1997. ISSN 1086-7651.

[46] Renaldas Zioma et Simon Green. Mastering DirectX 11 with Unity, March 2012. URL `https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/gdc12/GDC2012_Mastering_DirectX11_with_Unity.pdf`. Presentation at GDC 2012.

Annexe I

Pseudocode

We include pseudocode that depends on the following helper functions and shorthand operators: `sOffset` jitters a tile lookup (but never into a diagonal tile), `rnmix` is a vector linear interpolation followed by normalization, `norm` returns a normalized vector, $\lfloor \cdot \rfloor$ returns the whole component, and & denotes bitwise **and**. Unless otherwise specified, we use the notation/functions of McGuire et al. [24]. Our function returns **four** values: the filtered color (red (R), green (G), blue (B)) and a luminance (L) value to pass to an optional FXAA post-process.

function `filter`($\mathbf{p}$):

  let $j = \texttt{halton}(-1, 1)$

  let $\texttt{vmax} = \texttt{NeighborMax}[\mathbf{p}/r + \texttt{sOffset}(\mathbf{p}, j)]$

  if ($\|\texttt{vmax}\| \leq 0.5$)

    return $(\texttt{color}[\mathbf{X}], \texttt{luma}(\texttt{color}[\mathbf{X}]))$

  let $\mathbf{w}_n = \texttt{norm}(\texttt{vmax})$, $\mathbf{v}_c = \texttt{V}[\mathbf{p}]$, $\mathbf{w}_p = (-w_{ny}, w_{nx})$

  if ($\mathbf{w}_p \cdot \mathbf{v}_c < 0$) $\mathbf{w}_p = -\mathbf{w}_p$

  let $\mathbf{w}_c = \texttt{rnmix}(\mathbf{w}_p, \texttt{norm}(v_c), {}^{(\|\mathbf{v}_c\| - 0.5)}/_{\gamma})$

  // First integration sample: **p** with center weight

  let $\texttt{totalWeight} = {}^{N}/_{(\kappa \times \|\mathbf{v}_c\|)}$

  let $\texttt{result} = \texttt{color}[\mathbf{p}] \times \texttt{totalWeight}$

let $j' = j\,\eta\,\phi\,/\,N$

for $i \in [0, N)$

let $\mathtt{t} = \mathtt{mix}(-1, 1, (i + j' + 1)/(N + 1))$ // jitter sample

// Compute point S; split samples between $\{\mathbf{v}_{\max}, \mathbf{v}_c\}$

let $\mathtt{d} = \mathbf{v}_c$ **if** i odd **or** $\mathbf{v}_{\max}$ **if** i even

let $\mathtt{T} = \mathtt{t} \times \|\mathtt{vmax}\|$, $\mathtt{S} = \lfloor \mathtt{t} \times \mathtt{d} \rfloor + \mathbf{p}$

// Compute S's velocity and color

let $\mathbf{v}_s = \mathtt{V}[\mathtt{S}]$, $\mathtt{colorSample} = \mathtt{color}[\mathtt{S}]$

// Fore- vs. background classification of S w.r.t. **p**

let $f = \mathtt{zCompare}(Z[\mathbf{p}], Z[\mathtt{S}])$

let $b = \mathtt{zCompare}(Z[\mathtt{S}], Z[\mathbf{p}])$

// Sample weight and velocity-aware factors (Sec. 3.4.1)

// The length of $\mathbf{v}_s$ is clamped to 0.5 minimum during normalization

let $\mathtt{weight} = 0$, $\mathtt{wA} = \mathbf{w}_c \cdot \mathtt{d}$, $\mathtt{wB} = \mathtt{norm}(\mathbf{v}_s) \cdot \mathtt{d}$

// 3 phenomenological cases (Secs. 3.3, 3.4.1):

// Objects moving over **p**

$\mathtt{weight}$ += $f \cdot \mathtt{cone}(\mathtt{T}, 1/\|\mathbf{v}_s\|) \times \mathtt{wB}$

// **p**'s blurred motion

`weight` += $b \cdot \texttt{cone}(\texttt{T}, 1/\|\mathbf{v}_c\|) \times \texttt{wA}$

// Blending term for the two previous cases

`weight` += $\texttt{cylinder}(\texttt{T}, \min(\|\mathbf{v}_s\|, \|\mathbf{v}_c\|)) \times \max(\texttt{wA}, \texttt{wB}) \times 2$

// Summation of colors and weights for normalization

`totalWeight` += `weight`

`result` += `colorSample` × `weight`

return $(\texttt{result}/\texttt{totalWeight}, (\mathbf{p}_x + \mathbf{p}_y)\&1)$