TUGAS AKHIR - KI141502

Implementasi Filter Rekonstruksi untuk Menghasilkan Efek Motion Blur pada Game Engine Urho3D

YUSUF UMAR IBNU SYIHAB
NRP 5107100086

Dosen Pembimbing
Anny Yuniarti, S.Kom., M.Comp.Sc.
Ridho Rahman H., S.Kom., M.Sc.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2015

FINAL PROJECT - KI141502

Reconstruction Filter Implementation for Motion Blur Effects on Urho3D Game Engine

YUSUF UMAR IBNU SYIHAB
NRP 5107100086

Advisor
Anny Yuniarti, S.Kom., M.Comp.Sc.
Ridho Rahman H., S.Kom., M.Sc.

INFORMATICS DEPARTMENT
Faculty of Information Technology
Institut Teknologi Sepuluh Nopember
Surabaya 2015

LEMBAR PENGESAHAN

IMPLEMENTASI FILTER REKONSTRUKSI UNTUK MENGHASILKAN EFEK MOTION BLUR PADA GAME ENGINE URHO3D

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi IGS
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:
YUSUF UMAR IBNU SYIHAB
NRP. 5107100086

Disetujui oleh Pembimbing Tugas Akhir:

1. Anny Yuniarti, S.Kom., M.Comp.Sc ..... (pembimbing 1)
   NIP: 19810622 200501 2 002

2. Ridho Rahman H., S.Kom., M.Sc. ..... (pembimbing 2)
   NIP: 19870213 201404 1 001

SURABAYA
JANUARI, 2015

IMPLEMENTASI FILTER REKONSTRUKSI UNTUK MENGHASILKAN EFEK MOTION BLUR PADA GAME ENGINE URHO3D

Nama Mahasiswa : Yusuf Umar Ibnu Syihab
NRP : 5107100086
Jurusan : Teknik Informatika FTIf-ITS
Dosen Pembimbing I : Anny Yuniarti, S.Kom., M.Comp.Sc.
Dosen Pembimbing II : Ridho Rahman H., S.Kom., M.Sc.

ABSTRAK

Motion blur merupakan salah satu efek kamera yang muncul dikarenakan adanya waktu exposure kamera. Efek visual ini biasa disimulasikan pada film dan game untuk menambah kesan realistis.

Beberapa tahun terakhir ini, motion blur mulai umum tersedia pada game engine modern. Sayangnya karena berbagai keterbatasan, beberapa game engine open source masih belum menyediakan motion blur, salah satunya Urho3D. Tugas akhir ini mencoba mengimplementasi filter rekonstruksi untuk membuat efeke motion blur pada Urho3D. Filter rekonstruksi merupakan algoritma gather yang menyimulasikan efek motion blur secara realtime.

Dari hasil implementasi di tugas akhir ini menunjukkan filter rekonstruksi membutuhkan fine-tuning parameter untuk menghasilkan hasil yang cocok untuk situasi scene dan target hardware.

Kata kunci: motion blur, realtime, game engine, open source, Urho3D.

RECONSTRUCTION FILTER IMPLEMENTATION FOR MOTION BLUR EFFECTS ON URHO3D GAME ENGINE

Name : **Student Name**
NRP : **5107100086**
Major : **Informatics Department, FTIf-ITS**
Advisor I : **Anny Yuniarti, S.Kom., M.Comp.Sc.**
Advisor II : **Ridho Rahman H., S.Kom., M.Sc**

ABSTRACT

Motion blur is a visual effect caused by natural camera exposure time. This kind of effect is common used in film and video games to achieve more realistic pictures.

Recently, motion blur is common effect integrated on many modern game engine. Unfortunately, because of limited developer time and power, it still uncommon to see motion blur implemented on open source game engine, one of them is Urho3D. This final project will try to implement reconstruction filter to achieve motion blur effect on Urho3D. Reconstruction filter is an gather algorithm to simulate plausible motion blur in realtime.

This final project implementation resulting the need of fine-tuning parameter to achieve better motion blur effect to suit specific scene and target hardware.

Keywords: motion blur, realtime, game engine, open source, Urho3D.

KATA PENGANTAR

Segala puji dan syukur, kehadirat Allah SWT yang telah memberikan rahmat dan hidayah-Nya sehingga penulis dapat menyelesaikan tugas akhir yang berjudul "***Implementasi Filter Rekonstruksi untuk Menghasilkan Efek Motion Blur pada Game Engine Urho3D***".

Terima kasih kepada Prof. Morgan McGuire yang telah berbagi banyak ilmu dan inspirasi. Terima kasih juga kepada Lasse Örni, Yao Wei Tjong, Chris Friesen, dan kontributor Urho3D yang lain, terima kasih atas kontribusinya untuk kemajuan open source game development. Tidak lupa juga pada Bu Anny Yuniarti dan Pak Ridho Rahman yang telah membantu agar buku ini lebih pantas untuk dibaca.

Semoga buku tugas akhir ini dapat bermanfaat untuk kemajuan teknologi game development open source. Penulis mohon maaf jika masih ada kekurangan dan kesalahan.

Surabaya, Januari 2015

Yusuf Umar Ibnu Syihab

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

DAFTAR KODE SUMBER

DAFTAR PUSTAKA

[1] Linneman J. Do higher frame-rates always mean better gameplay? [Internet]. 2014 Available from: http://www.eurogamer.net/articles/digitalfoundry-2014-frame-rate-vs-frame-pacing.

[2] Wikipedia. Graphics processing unit. [Internet]. [cited 2014 October 5]. Available from: http://en.wikipedia.org/wiki/Graphics_processing_unit.

[3] Epic Games, Inc. MotionBlur Skinning Post Process Feature. [Internet]. [cited 2015 January 5]. Available from: https://udn.epicgames.com/Three/MotionBlurSkinning.html.

[4] Unity. License Comparisons. [Internet]. [cited 2014 October 5]. Available from: http://unity3d.com/unity/licenses.

[5] McGuire M, Hennesy M, Bukowski P, Osman M. A Reconstruction Filter for Plausible Motion Blur. In: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2012 (I3D'12); 2012.

[6] NVIDIA Corporation. GeForce GTX TITAN. [Internet]. [cited 2015 January 5]. Available from: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan.

[7] Galloy M. CPU vs GPU performance. [Internet]. [cited 5 January 2015]. Available from: http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html.

[8] Kinghorn D. Haswell Floating Point Performance. [Internet]. [cited 5 January 2015]. Available from: http://www.pugetsystems.com/blog/2013/08/26/Haswell-Floating-Point-Performance-493/.

[9] Leach G, Archer J. Lecture: Graphics pipeline and animation. [Internet]. [cited 2015 January 5]. Available from:

http://goanna.cs.rmit.edu.au/~gl/teaching/Interactive3D/.

[10] jMonkeyEngine. JME3 and Shaders. [Internet]. [cited 2015 January 5]. Available from: http://wiki.jmonkeyengine.org/doku.php?id=jme3:advanced:jme3_shaders.

[11] Wikipedia. Motion blur. [Internet]. [cited 2014 October 5]. Available from: http://en.wikipedia.org/wiki/Motion_blur.

[12] Chapman J. Motion Blur Tutorial. [Internet]. 2011 Available from: http://john-chapman-graphics.blogspot.com/2013/01/what-is-motion-blur-motion-pictures-are.html.

[13] Qayyum A. 40 Amazing Shots of Motion Blur in Photographs. [Internet]. [cited 2015 January 5]. Available from: http://smashinghub.com/motion-blur-in-photographs.htm.

[14] ozlael. Motionblur. [Internet]. [cited 2015 January 5]. Available from: http://www.slideshare.net/ozlael/motionblur-3252650.

[15] Anderson SD. Reading on the Accumulation Buffer: Motion Blur, Anti-Aliasing, and Depth of Field. [Internet]. [cited 2015 January 02]. Available from: http://cs.wellesley.edu/~cs307/readings/15-accumulation.pdf.

[16] Muller PM. Tutoriel: Effet de Motion blur (flou de mouvement) avec pixel shaders. [Internet]. [cited 2015 January 2015]. Available from: http://www.xna-connection.com/tag/Effets.

[17] Unity Technologies. Motion Blur. [Internet]. [cited 2015 January 5]. Available from: http://docs.unity3d.com/Manual/script-MotionBlur.html.

[18] ucupumar. Motion Blur. [Internet]. [cited 2015 January 5]. Available from: http://urho3d.prophpbb.com/topic433.html.

[19] Chapman J. Per-Object Motion Blur. [Internet]. 2013

Available from: http://john-chapman-graphics.blogspot.com/2013/01/per-object-motion-blur.html.

[20] Urho3D. About. [Internet]. [cited 2014 October 5]. Available from: http://urho3d.github.io/about.html.

[21] Urho3D. Urho3D. [Internet]. [cited 2015 January 5]. Available from: http://sourceforge.net/projects/urho3d/.

[22] Urho3D. urho3d/Urho3D. [Internet]. [cited 2014 October 5]. Available from: https://github.com/urho3d/Urho3D.

[23] Activeworlds, Inc. Z-buffer fighting. [Internet]. [cited 2015 January 5]. Available from: http://wiki.activeworlds.com/index.php?title=Z-buffer_fighting.

[24] NVIDIA Corporation. Motion Blur GL4/GLES3 Advanced Sample. [Internet]. [cited 2015 January 5]. Available from: http://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/opengl_samples/motionblurgl4gles3advancedsample.htm.

[25] Guertin JP, McGuire M, Nowrouzezahrai D. A Fast and Stable Feature-Aware Motion Blur Filter. NVIDIA Corporation; 2013. NVR-2013-003.

[26] Jimenez J. Next Generation Post Processing in Call of Duty: Advanced Warfare. [Internet]. 2014 Available from: http://www.iryoku.com/next-generation-post-processing-in-call-of-duty-advanced-warfare.

BIODATA PENULIS

Penulis, Yusuf Umar Ibnu Syihab lahir di Bandung pada 29 Januari 1989. Penulis adalah anak keempat.dari delapan bersaudara dan dibesarkan di Jakarta dan Probolinggo.

Penulis menempuh pendidikan formal di SDN Pluit 03 Pagi Jakarta Utara (1995-1998), SDN Sukabumi X no 10 Kota Probolinggo (1998-2001), SMPN 1 Kota Probolinggo (2001-2004), dan SMAN 1 Kota Probolinggo (2004-2007). Pada tahun 2007, penulis menempuh pendidikan S1 jurusan Teknik Informatika Fakultas Teknologi Informasi di Institut Teknologi Sepuluh Nopember, Surabaya, Jawa Timur.

Di jurusan Teknik Informatika, penulis mengambil bidang minat IGS dan memiliki kompetensi pada beberapa subjek seperti seni visual, modeling dan texturing 3D, game design, pemrograman GPU, dan sejarah teknologi komputer. Penulis dapat dihubungi melalui alamat email yusufumaris@gmail.com

BAB I
PENDAHULUAN

Pada bab ini akan dipaparkan mengenai garis besar tugas akhir yang meliputi latar belakang, tujuan, rumusan dan batasan permasalahan, metodologi pembuatan tugas akhir, dan sistematika penulisan.

1.1. Latar Belakang

Semakin majunya teknologi, *game* modern semakin mengejar tampilan game yang semakin mendekati dengan dunia nyata. Berbeda dengan kamera didalam *game*, kamera didunia nyata secara alami menghasilkan berbagai efek, contohnya HDR *rendering*, *depth of field* (DOF), *motion blur*, *chromatic aberration*, dan banyak lagi lainnya. Tidak mudah mensimulasikan berbagai efek tersebut kedalam game, dikarenakan game biasa bergerak secara *realtime* dengan kecepatan minimum 30fps (*frame per second*). Dengan kecepatan seperti itu, seluruh algoritma yang digunakan untuk melakukan *render* satu gambar tidak boleh melebihi batas 33ms (*millisecond*) [1].

Untungnya saat ini teknologi komputasi paralel telah berkembang begitu pesat. *Game* modern saat ini bergantung pada hardware khusus yang biasa disebut GPU (*Graphics Processing Unit*). Berbeda dengan arsitektur CPU tradisional, GPU didesain bekerja paralel dengan prinsip SIMD (*Single Instruction Multiple Data*). Sejak era Geforce 3 pada tahun 2001, GPU dapat diprogram melalui bahasa HLSL untuk API DirectX atau bahasa GLSL untuk API OpenGL [2].

Beberapa efek-efek kamera yang disebut diatas sudah umum diimplementasikan sebagai efek postprocessing pada berbagai *game engine proprietary*, contohnya Unreal Engine [3] dan Unity Pro [4]. Berbeda dengan *game engine open source*, dikarenakan terbatasnya waktu dan tim *developer*, banyak efek-efek modern yang belum diimplementasikan.

Contoh efek motion blur pada game engine Unreal bisa dilihat pada Gambar 1.1.

Gambar 1.1 Motion blur pada Unreal Engine [3]

Dengan mulai populernya studio *game indie*, *game engine open source* bisa dijadikan alternatif saat ini dikarenakan relatif mahalnya biaya menggunakan *game engine proprietary*. Sebagai catatan tambahan, Unity menyediakan opsi gratis pada lini produk mereka, tetapi versi ini menghapus fitur RTT (*Render-to-Texture*) yang sangat dibutuhkan untuk membuat efek *postprocessing* [4].

Tugas akhir ini memfokuskan pada implementasi filter rekonstruksi untuk menghasilkan efek kamera *motion blur* sebagai salah satu efek *postprocessing* pada game engine open source Urho3D. *Motion blur* merupakan efek yang terjadi pada gambar bergerak dikarenakan adanya pergerakan objek saat

exposure kamera terjadi. Sedangkan Urho3D merupakan *game engine open source* berlisensi MIT yang pertama kali ditulis oleh Lasse Örni pada Januari 2011.

Algoritma yang dipakai untuk implementasi *motion blur* ini berdasarkan pada paper *A Reconstruction Filter for Plausible Motion Blur*. Algoritma telah terbukti cukup presisi dan dapat berjalan pada *console game* Xbox 360 dengan kecepatan 1,5ms [5].

1.2. Rumusan Permasalahan

Rumusan masalah yang diangkat dalam tugas akhir ini adalah sebagai berikut:

1. Bagaimana menghasilkan *buffer velocity* dari data gerakan objek dan kamera?
2. Bagaimana menghasilkan *buffer* TileMax dan NeighborMax dari data *velocity*?
3. Bagaimana algoritma filter rekonstruksi bekerja?
4. Bagaimana implementasi filter rekonstruksi untuk menghasilkan *motion blur* pada *game engine* Urho3D?
5. Bagaimana pengaruh variabel S dan k pada kualitas gambar *motion blur*?
6. Bagaimana performa filter rekonstruksi efek *motion blur*?

1.3. Batasan Permasalahan

Permasalahan yang dibahas dalam tugas akhir ini memiliki beberapa batasan, di antaranya sebagai berikut:

1. Implementasi *motion blur* hanya pada *game engine* Urho3D saja.
2. Input *motion blur* yang digunakan hanya pergerakan kamera dan objek
3. Hasil akhir merupakan demo dari *motion blur* yang berjalan diatas game engine Urho3D.

4. Dikarenakan relatif intensifnya komputasi algoritma *motion blur*, *platform* demo yang ditarget hanya PC dan tidak untuk *platform mobile* (Android/iOS).

1.4. Tujuan

Tujuan dari pembuatan tugas akhir ini adalah dapat mengimplementasikan efek *motion blur* dengan algoritma filter rekonstruksi pada *game engine* Urho3D.

1.5. Manfaat

Dengan adanya efek *motion blur* pada *game engine* Urho3D, diharapkan menambah daya tarik *game engine open source*, terutama Urho3D, pada kalangan *developer indie*.

BAB II
DASAR TEORI

Pada bab ini akan dibahas mengenai dasar teori yang menjadi dasar pembuatan tugas akhir ini.

2.1. GPU Programming

Sebelum lebih lanjut membahas mengenai ***motion blur***, maka perlu dijelaskan terlebih dahulu bahwa tugas akhir ini banyak menggunakan teknologi GPU dibanding CPU untuk melakukan implementasi. Maka dari itu diperlukannya beberapa penjelasan dasar teknologi dan berbagai istilahnya.

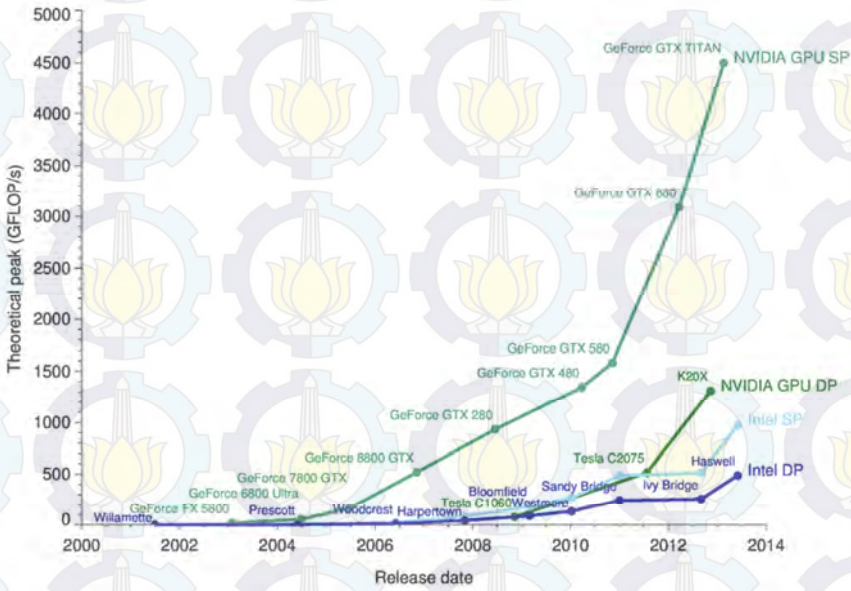
2.1.1. GPU dan sejarah singkatnya

GPU (***Graphical Processing Unit***) merupakan ***hardware*** khusus yang digunakan untuk memproses visual. GPU pada PC dihubungkan melalui slot PCI Express x16 pada motherboard. Desain arsitektur dari GPU ini berbeda dengan CPU biasa (x86/ARM) dikarenakan lebih menekankan pada komputasi yang sangat paralel.

Gambar 2.1 Nvidia Geforce GTX TITAN [6]

Gambar 2.1 adalah contoh gambar GPU yang bernama GeForce GTX TITAN. GPU ini diproduksi oleh NVidia pada tahun 2013 [6]. Dikarenakan performa komputasinyanya yang sangat tinggi, GPU ini populer dikalangan ***consumer*** dan ***professional***. GTX TITAN ini memiliki 2048 ***shader processor*** (***core***) dengan kemampuan komputasi hingga 4500 GFLOPS [7]. Bandingkan dengan CPU Intel Dual Xeon E5 2687 yang memiliki 16 ***core*** 'hanya' memiliki kemampuan komputasi 345 GFLOPS [8].

Untuk perbandingan GPU lain dengan berbagai CPU, bisa dilihat pada grafik di Gambar 2.2.



Gambar 2.2 Perbandingan performa Floating Unit pada GPU dan CPU [7]

Dikarenakan perbedaan arsitektur, GPU di-***program*** menggunakan konvensi yang berbeda dengan CPU biasa.

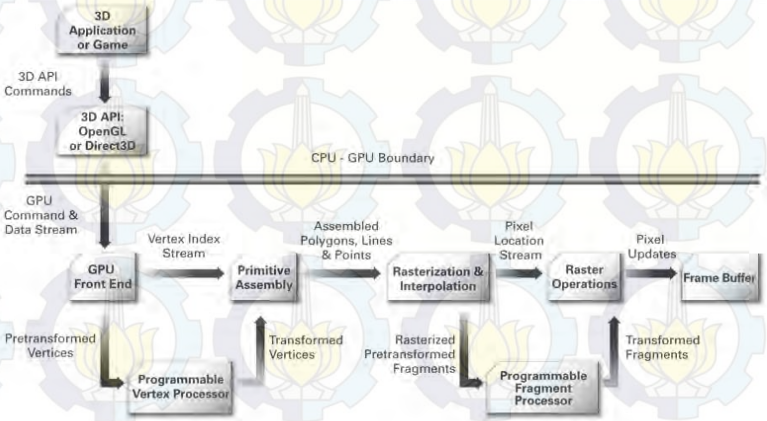
Sebelum era ***programmable shader*** (2001 kebawah), GPU bahkan tidak dapat di-***program*** sama sekali. GPU hanya dapat diperintah melakukan instruksi khusus untuk melakukan ***render*** melalui API OpenGL (versi 1.0) atau DirectX (versi 7.0 kebawah). Setelah Xbox generasi pertama dan Geforce 3 dirilis (2001), GPU mulai dapat diprogram layaknya CPU, biarpun masih banyak keterbatasan [2].

Sejak Geforce 3 ini, GPU secara umum mulai memiliki kemampuan ***pixel*** dan ***vertex shader***. Kemampuan ini memungkinkan ***programmer*** untuk menulis program untuk tiap ***vertex*** dan ***pixel*** yang di-***render***. Hal ini sangat revolusioner saat itu dikarenakan banyak teknik ***render*** baru yang bisa diimplementasikan dengan memanfaatkan kemampuan ini, contohnya ***per-pixel lighting***. ***Per-pixel lighting*** atau ***phong lighting*** sebelumnya hanya bisa dilakukan pada CPU secara ***offline***, namun sejak adanya ***vertex*** dan ***pixel shader***, teknik ini dapat dilakukan secara ***realtime*** pada GPU.

2.1.2. Pipeline GPU

Untuk menghasilkan gambar di monitor/***display***, terdapat berbagai proses yang harus dilalui. Secara umum ***pipeline*** untuk menghasilkan gambar dapat dilihat pada Gambar 2.3

Kotak diatas "***GPU-CPU Boundary***" tidak menjadi bagian dari hardware GPU. Bagian ini diimplementasi pada CPU, bisa berupa aplikasi, ***game***, OS, atau ***driver*** GPU. Perlu diperhatikan biarpun GPU dapat di-***program***, tidak semua bagian dari ***pipeline*** dapat di-***program***. Jika dilihat diperhatikan lagi pada Gambar 01, hanya ***vertex*** dan ***pixel shader*** yang dapat di-***program*** pada GPU [9].



Gambar 2.3 Pipeline GPU [9]

Untuk penjelasan masing-masing kotak bisa dilihat pada Tabel 2.1.

Tabel 2.1 Penjelasan Pipeline GPU [9]

Pipeline	Penjelasan
3D Application or Game	Aplikasi yang ditulis. Aplikasi yang akan memanggil API OpenGL/DirectX. Salah satu tugas utama aplikasi ini adalah menyiapkan berbagai data yang akan dikirim ke GPU untuk melakukan proses ***render***.
3D API: OpenGL or Direct3D	OpenGL atau DirectX menyediakan berbagai perintah untuk GPU. Kumpulan perintah yang tersedia pada GPU berbeda-beda tergantung ***driver*** dan ***feature set*** dari ***hardware*** GPU itu sendiri. ***Driver*** yang mengubah berbagai perintah ini agar GPU dapat menjalankannya. Hampir semua GPU saat ini men-***support*** API

Pipeline	Penjelasan
	DirectX dan OpenGL pada ***driver***-nya. DirectX hanya pada OS buatan Windows, sedangkan OpenGL tersedia pada Windows, Mac OS, dan Linux.
GPU Front End	GPU menerima perintah dari CPU melalui PCI ***bus***. Dari sini perintah ini akan dijalankan oleh GPU.
Programmable Vertex Processor	Atau bisa juga disebut dengan ***vertex shader***, bagian ini adalah bagian yang dapat di-***program***. Program ***vertex shader*** ditulis menggunakan bahasa GLSL pada OpenGL dan HLSL pada DirectX. Program ***vertex shader*** ini berisi kode yang akan dijalankan pada tiap ***vertex*** pada ***vertex buffer***. ***Vertex shader*** paling umum diimplementasi adalah transformasi posisi ***vertex*** dari posisi yang relatif terhadap model ke posisi akhirnya dilayar
Primitive Assembly	Setelah melakukan transformasi, ***vertex*** disusun menjadi bentuk primitif sesuai urutan yang berada pada ***vertex index buffer***.
Rasterization & Interpolation	Posisi ***vertex*** dilayar bisa saja berjauhan biarpun seharusnya terhubung pada satu ***triangle***. Untuk mengisi ***pixel*** diantara ***vertex*** ini dilakukan proses ***rasterization***.
Programmable Fragment Processor	Sama seperti ***vertex shader***, ***pixel shader*** juga berupa program yang ditulis dengan bahasa GLSL atau HLSL.

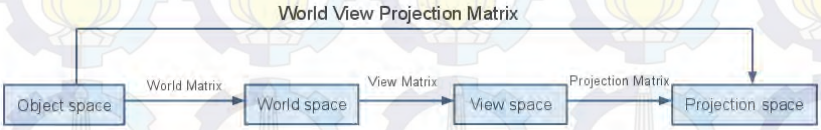
Pipeline	Penjelasan
	Program ***pixel shader*** dilakukan tiap ***pixel*** di layar yang tercakup di hasil proses ***rasterization*** sebelumnya. Berbagai teknik bisa dilakukan memanfaatkan ***pixel shader***. Yang paling umum diimplementasi adalah untuk menghasilkan ***Per-pixel/phong lighting***. ***Pixel shader*** juga bisa digunakan untuk berbagai efek ***post-processing***, diantaranya ***bloom***, ***color correction***, ***pixel shader***, dan banyak lagi lainnya.
Raster Operations	Setelah ***pixel shader*** menghasilkan ***output*** warna dengan format RGBA. Warna tersebut dibandingkan dengan ***pixel*** yang sudah ada di ***framebuffer*** dari hasil ***pixel shader*** sebelumnya. Proses operasi perbandingan ini disebut disebut raster ***operation***. Pada proses ini dua warna tersebut bisa ditambahkan (***add***), dikalikan (***multiply***), atau saling dikalikan dengan nilai ***alpha***-nya kemudian dijumlah (***alpha blending***).
Frame Buffer	***Framebuffer*** merupakan area pada ***memory*** GPU yang menyimpan hasil warna ***pixel*** akhir. ***Framebuffer*** ini yang akan dikirm ke monitor/***display*** untuk ditampilkan. GPU saat ini juga bisa memanfaatkan fitur yang dinamakan ***Render-To-Texture*** (RTT). Dengan RTT, hasil akhir render tidak langsung ditampilkan dilayar, melainkan ditampung dahulu pada ***buffer*** dan digunakan untuk menjadi ***input*** pada proses ***pixel shader***

Pipeline	Penjelasan
	berikutnya.

Sebenarnya masih ada ***shader*** lain selain ***vertex*** dan ***pixel***, yaitu ***geometry***, ***tesselation***, dan ***compute shader***. ***Shader-shader*** tersebut hanya terdapat pada spesifikasi OpenGL 3.0 atau DirectX 10 keatas. ***Shader*** ini mulai umum terdapat pada GPU ***desktop***, tetapi masih jarang terdapat pada GPU ***mobile***.

2.2. Transformasi Matrix Space

Pada pemrograman GPU, pemahaman akan transformasi ***matrix*** sangat diperlukan. Pada bagian sebelumnya dijelaskan bahwa ***vertex shader*** umumnya melakukan transformasi posisi relatif terhadap pusat objek diubah menjadi posisi di layar. Proses yang sebenarnya terjadi bisa dilihat di Gambar 2.4.



Gambar 2.4 Transformasi Matrix Space [10]

Posisi yang diterima pada ***vertex shader*** umumnya berada pada ***object space***. ***Object space*** merupakan koordinat yang relatif terhadap titik pusat dari objek tersebut. Biasanya titik pusat ini didefinisikan oleh artis pada aplikasi pembuat konten 3D, seperti Blender atau 3DS Max.

Posisi ***object space*** ini harus diubah ke ***world space***, posisi koordinat yang relatif dari titik pusat scene. Cara mengubah space adalah dengan mengalikan posisi ***object space*** dengan ***world matrix***.

Begitu seterusnya hingga ***projection space***. ***View space*** merupakan posisi yang relatif terhadap kamera, sedangkan projection ***space*** merupakan posisi akhir di layar. Nilai ***matrix-matrix*** tersebut biasanya didefinisikan di aplikasi (dalam kasus

ini, Urho3D), dan dikirim ke GPU berupa nilai uniform. Karena sifat GPU yang paralel, nilai uniform merupakan nilai yang nilainya sama untuk tiap ***vertex*/*pixel*** dalam satu ***batch shader***.

```
1  WorldViewProjectionMat = WorldMat * ViewMat
   * ProjectionMat
```

Kode Sumber 2.1 World View Projection Matrix

Untuk mempercepat proses perhitungan, biasanya ***World View Projection*** langsung digabung menjadi untuk mengurai kalkulasi ***matrix*** pada GPU. Cara menggabungnya cukup dikalikan saja, seperti pada Kode Sumber 2.1.

2.3. Motion Blur

Motion blur merupakan efek lintasan kabur pada objek yang bergerak cepat yang terjadi pada gambar bergerak atau animasi. Efek ini dikarenakan ada objek yang bergerak saat pengambilan gambar berlangsung [11].

Pada dunia nyata atau fotografi, efek ***motion blur*** bisa didapatkan dengan cara memperpanjang waktu ***exposure***. Semakin lama waktu ***exposure***, semakin banyak cahaya yang masuk ke sensor kamera sehingga gambar terlihat menumpuk searah pada arah gerakan benda yang bergerak [12].

Pada dunia komputer grafis, efek ini bisa dimodelkan dengan algoritma tertentu. Pada ***offline rendering***, algoritma ***brute force*** paling sederhana adalah dengan merender gambar berkali-kali dalam rentang waktu exposure tertentu. Hasilnya kemudian digabung menjadi satu. Sayangnya cara ini hampir tidak mungkin diimplementasikan pada ***realtime rendering*** dikarenakan perlunya melakukan ***render*** gambar berkali-kali hanya untuk menghasilkan satu ***frame*** saja.

Gambar 2.5 Contoh motion blur pada foto di dunia nyata [13]

Beberapa teknik telah dilakukan untuk mensimulasikan efek ini. Dibawah ini akan dijelaskan beberapa teknik dasar untuk mengimplementasikan ***motion blur*** secara ***realtime***.

2.3.1. Accumulation Buffer

Accumulation buffer adalah salah satu teknik ***motion blur*** paling primitif di dunia ***realtime rendering***. Teknik ini umum dilakukan pada era Playstation 1 hingga Playstation 2 (1994-2006) dikarenakan keterbatasan kemampuan ***hardware***.

Teknik ini dilakukan dengan cara melakukan ***alpha blending framebuffer*** saat ini dengan ***framebuffer*** sebelumnya. Teknik seperti ini menghasilkan efek ***trail*** pada objek pada layar.

Gambar 2.6 Accumulation Buffer Motion Blur [14]

Keterbatasan dari teknik ini adalah secara alami akan menghasilkan efek yang kurang presisi. Dikarenakan selalu melakukan ***alpha blending*** tiap ***frame***, teknik ini secara teoritis akan mengambil sampel ***frame*** n, ***frame*** n-1, ***frame*** n-2, dst [15]. Hal ini menimbulkan gambar kabur dan berbayang. ***Motion blur*** yang presisi seharusnya hanya mengambil sampel gambar diantara ***frame*** n dan ***frame*** n-1.

Contoh game yang menggunakan teknik ini adalah Need for Speed Underground (2003). Screenshot dari game ini dapat dilihat pada Gambar 2.7.

Gambar 2.7 Accumulation buffer motion blur pada Need for Speed Underground (2003) [16]

2.3.2. Camera motion blur

Setelah era ***programmable shader*** populer (2006 sampai sekarang), teknik ini muncul. ***Camera motion blur*** merupakan teknik yang cukup populer dikarenakan relatif ringan untuk dijalankan dan ***support*** untuk berbagai macam teknik ***rendering*** (***Forward***/***Deferred***).

Teknik ini mengharuskan ***render engine*** untuk me-***render*** tambahan Linear ***depth buffer***, yaitu ***buffer*** yang digunakan untuk menyimpan kedalaman linear dengan nilai 0 sampai 1, dari ***near clip*** hingga ***far clip***. Sebenarnya data dari standar z-buffer juga bisa digunakan, tetapi dikarenakan presisinya yang sangat rendah, maka z-buffer sangat tidak disarankan untuk teknik ini.

Setelah seluruh gambar selesai di-***render***, dilakukan proses rekonstruksi posisi tiap ***pixel*** pada frame n-1 dari data ***linear depth*** dan ***matrix*** proyeksi kamera ***frame*** n-1. Dari sini tiap ***pixel*** akan ada dua data, posisi saat ***frame*** n dengan ***frame*** n-1. Jika dikurangi maka akan menghasilkan data ***velocity*** dari ***pixel*** tersebut. Data ***velocity*** ini yang akan dilakukan untuk melakukan sampel ***blur*** [12].

Gambar 2.8 Camera Motion Blur [17]

Gambar 2.8 merupakan contoh efek ***camera motion blur*** pada ***game engine*** Unity dengan lisensi ***pro***. Efek ini tidak tersedia pada Unity yang berlisensi ***free*** [4].

Kelemahan dari teknik ini adalah efek ***motion blur*** hanya terasa ketika kamera bergerak, dan bisa menimbulkan masalah jika ada benda yang bergerak di ***scene***.

2.3.3. Per-object motion blur

Per-object motion blur adalah pengembangan lebih lanjut dari ***camera motion blur***. Kelebihan dari teknik ini adalah memungkinkan terjadinya efek ***motion blur*** pada objek yang bergerak meskipun kamera diam.

Gambar 2.9 Per-object motion blur [18]

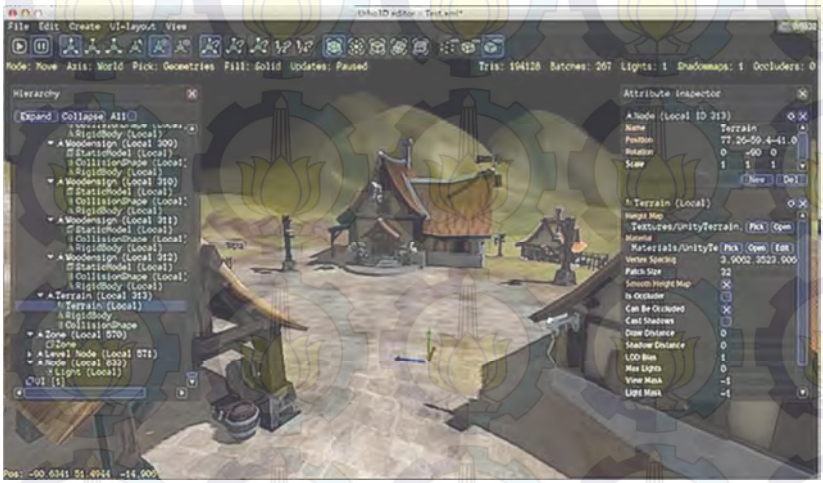
Daripada me-***render*** linear ***depth buffer***. Teknik ini langsung me-***render velocity buffer*** dari awal. ***Velocity buffer*** adalah ***buffer*** yang menampung kecepatan tiap ***pixel*** relatif dari posisi pada ***frame*** sebelum ***frame*** sekarang (frame n-1). Untuk

me-***render velocity buffer*** ini, tiap objek harus memiliki ***matrix camera projection*** dari ***frame*** sekarang (n) dan ***frame*** n-1 [19] .

Biarpun sudah terasa ideal, teknik ini memiliki kelemahan yang cukup fundamental. Dikarenakan ***velocity buffer*** yang hanya memiliki informasi kecepatan tiap ***pixel***, hal ini menimbulkan objek yang bergerak diatas benda diam atau kebalikannya tidak transisi sempurna.

2.4. Urho3D

Urho3D merupakan ***game engine open source*** berlisensi MIT yang pertama kali ditulis oleh Lasse Örni asal Finlandia pada Januari 2011. Urho3D terinspirasi dari ***engine*** OGRE dan HORDE3D. Tidak seperti OGRE yang hanya merupakan ***graphics engine*** saja, Urho3D memiliki banyak fitur yang membantu dalam pengembangan ***video game***, antara lain integrasi ***physics engine***, ***pathfinding***, ***skeletal animation***, ***scene editor***, dan banyak lagi lainnya. Urho3D mendukung target ***platform*** Windows, Mac, Linux, Android, dan iOS. API Urho3D dapat menggunakan bahasa C++ atau bisa juga menggunakan bahasa scripting AngelScript atau Lua [20].



Gambar 2.10 Scene Editor Urho3D [21]

Repository Urho3D di-***host*** pada Github [22]. Hingga proposal ini ditulis, sudah lebih dari 4700 ***commit*** dengan total 27 kontributor telah terlibat.

2.4.1. Scene dan components

Scene pada Urho3D dapat dideskripsikan dengan ***component-based graph***. ***Scene*** berisi hierarki ***node*** yang merepresentasikan seluruh ***scene***. ***Scene*** dikepalai oleh satu root node.

Node merupakan objek-objek yang berada pada ***scene***. Setiap ***node*** memiliki properti nama, transformasi, dan ***component***. ***Component*** merupakan atribut pada ***node*** yang bisa berfungsi bermacam-macam. ***Component*** bisa berupa ***static model***, ***sound playback***, ***logic script***, atau berbagai macam lainnya. Tiap ***node*** bisa memiliki berbagai ***component***.

2.4.2. Proses render Urho3D

Proses ***render*** tiap ***frame*** pada Urho3D diatur oleh kelas Renderer. Kelas ini mengontrol ***setting*** global mulai dari kualitas tekstur, ***material***, ***lighting***, dan ***shadow map***.

Untuk melakukan proses ***render***, Renderer membutuhkan kamera dan ***scene*** dengan komponen Octree. Octree menyimpan dan melakukan ***query*** seluruh objek yang terlihat pada layar. Octree ini akan mempercepat proses ***render*** dikarenakan objek yang tidak terlihat oleh kamera tidak akan diproses lebih lanjut dalam proses ***render***. Kamera dan ***scene*** tersebut harus di-***set*** pada objek ***viewport***, yang bisa dilakukan dengan fungsi SetViewport() pada Renderer.

Viewport ini menggunakan daftar urutan render yang berupa berkas xml. Berkas ini dinamakan RenderPath, akan dijelaskan pada bagian selanjutnya.

2.4.3. Renderpath

RenderPath berisi daftar perintah ***render*** dan ***rendertarget***. ***Rendertarget*** merupakan ***render buffer*** yang tidak langsung ditampilkan dilayar tetapi disimpan pada tekstur untuk ***input shader***. Perintah ***render*** pada RenderPath bisa di-***output***-kan langsung ke layar ataupun ***rendertarget***. Seluruh proses yang disebutkan dalam ***renderpath*** dilakukan tiap ***frame***.

Contoh dari berkas RenderPath adalah seperti pada Kode Sumber 2.2.

```
1  <renderpath>
2  <rendertarget name="depth" sizedivisor="1 1"
   format="lineardepth" />
3  <command type="clear" color="1 1 1 1"
   depth="1.0" output="depth" />
4  <command type="scenepass" pass="depth"
   output="depth" />
5  <command type="clear" color="fog"
   depth="1.0" stencil="0" />
6  <command type="scenepass" pass="base"
   vertexlights="true" metadata="base" />
7  <command type="forwardlights" pass="light"
   />
8  <command type="scenepass" pass="postopaque"
   />
9  <command type="scenepass" pass="refract">
10 <texture unit="environment" name="viewport"
   />
11 </command>
12 <command type="scenepass" pass="alpha"
   vertexlights="true" sort="backtofront"
   metadata="alpha" />
13 <command type="scenepass" pass="postalpha"
   sort="backtofront" />
14 </renderpath>
```

Kode Sumber 2.2 Contoh RenderPath ForwardDepth.xml

Kode Sumber 2.2 merupakan RenderPath bawaan Urho3D bernama ForwardDepth.xml. ***Rendertarget*** tambahan pada RenderPath ini bernama ***depth***, yang berfungsi untuk menampung kedalaman ***pixel*** dengan rentang nilai ***linear*** dari 0.0 ke 1.0.

Untuk perintah-perintah pada RenderPath ForwardDepth, bisa dijelaskan sebagai berikut:

- Clear: me-***reset*** nilai pixel ke nilai tertentu (***color*** '1 1 1 1' berarti putih (RGBA 1), sedangkan ***fog*** berarti nilai fog yang dideskripsikan pada aplikasi)
- Scenepass: dimana ***shader*** tertentu dijalankan. ***Shader*** yang dijalankan sesuai dengan yang didefinisikan pada ***technique*** pada tiap material objek. Technique dan material akan dijelaskan pada bagian berikutnya. ***Scenepass*** juga memiliki atribut ***output***. Jika tidek didefinisikan rendertarget yang spesifik, ***output*** ini akan langsung tampil kelayar.
- Forwardlights: dimana proses ***lighting*** dilakukan.

2.4.4. Technique dan Material

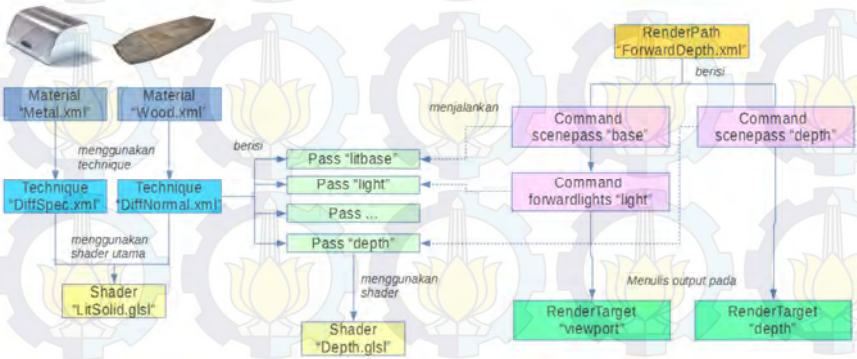
Tiap model yang akan di-***render*** pada Urho3D harus memiliki material. Material ini mendefinisikan ***input uniform*** dan tekstur yang akan dipakai pada ***shader***. ***Shader*** tidak didefinisikan langsung pada material melainkan pada ***technique***. Tiap material harus memiliki paling tidak satu ***technique*** yang dipakai.

Technique merupakan daftar berbagai pass yang akan dijalani oleh suatu material hingga model tampil di layar. ***Technique*** juga yang mendefinisikan ***shader*** yang akan dipakai.

Hubungan antara ***material***, ***technique***, dan ***renderpath*** bisa dilihat pada Gambar 2.11. Pada gambar tersebut terdapat dua objek yang memiliki dua ***material***, ***Metal.xml*** dan ***Wood.xml***. Dua ***material*** tersebut menggunakan ***technique*** yang berbeda meskipun ***shader*** yang digunakan sama. ***Shader*** pada Urho3D bisa berisi

berbagai macam ***preprocessor*** yang didefinisikan oleh ***technique*** sehingga hasil ***compile program shader*** bisa berbeda biarpun menggunakan ***file glsl*** yang sama.

Jika ***technique*** berisi berbagai ***pass*** yang akan dijalankan pada proses ***render*** nantinya, RenderPath yang menentukan ***pass*** apa saja yang akan dijalankan selama proses ***render***.



Gambar 2.11 Hubungan antara material, technique dan renderpath

Pass pada ***technique*** juga bisa memanggil shader lain selain shader utama bergantung kebutuhan.

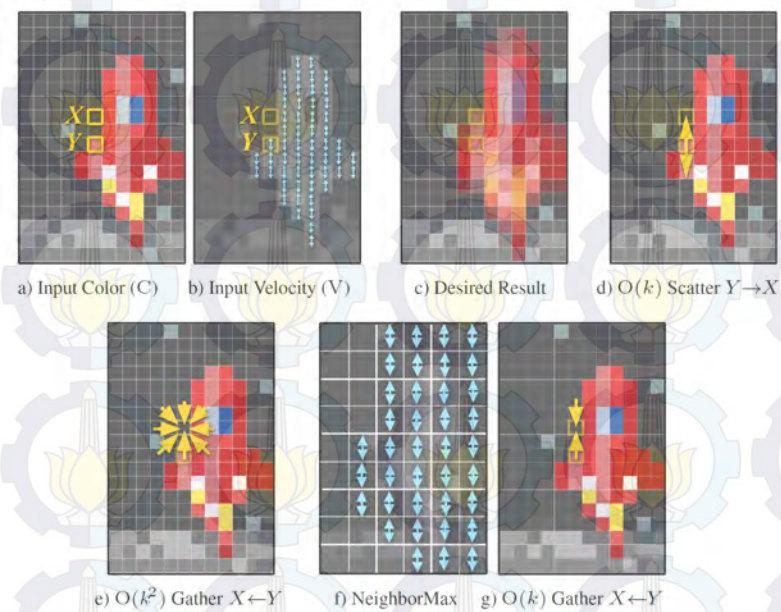
Tiap ***command*** pada RenderPath berkaitan dengan ***pass*** yang berada pada ***technique***. ***Pass*** dari suatu material dari objek akan dijalankan jika ***pass*** tersebut juga didefinisikan pada ***Renderpath***, begitu pula sebaliknya.

BAB III
ANALISIS DAN PERANCANGAN SISTEM

Bab ini membahas tahap analisis permasalahan dan perancangan dari sistem yang akan dibangun.

3.1. Analisis

Filter rekonstruksi adalah teknik yang mencoba mengatasi permasalahan dari ***per-object motion blur*** pada dasar teori. Daripada menggunakan secara langsung nilai ***velocity*** pada ***pixel*** untuk melakukan sampel blur, teknik ini merekonstruksi ***velocity pixel*** tetangga untuk menjadi sampel ***blur***.



Gambar 3.1 Berbagai strategi melakukan blur pada roket [5]

Untuk penjelasan lebih detail dari filter rekonstruksi bisa dilihat pada Gambar 3.1. Gambar (a) menunjukkan roket yang meluncur keatas. Gambar (b) menunjukkan ***velocity buffer***,

kecepatan dari tiap ***pixel***. Jika dianggap velocity pada titik Y mempunyai panjang 5 ***pixel***, maka hasil yang diinginkan dari ***motion blur*** akan terlihat seperti digambar (c).

Gambar (d) menunjukkan salah satu solusi yang bisa dilakukan untuk mendapat hasil seperti gambar c. Caranya adalah dengan melakukan ***scatter*** warna sepanjang 5 ***pixel*** sesuai ***velocity***. Algoritma ini bisa diimplementasikan dengan cara iterasi ***input*** tekstur berulang kali. Sayangnya cara ini sangat tidak efisien pada ***hardware*** GPU yang sifatnya paralel.

Untuk mengatasi hal ini, operasi ***scatter*** tersebut dibalik menjadi ***gather*** seberti gambar (e). Biarpun jauh lebih efisien dibanding ***scatter***, proses ini juga bisa membebani GPU dikarenakan untuk mendapat ***blur*** yang presisi diperlukan iterasi banyak pixel sekeliling titik ***X***.

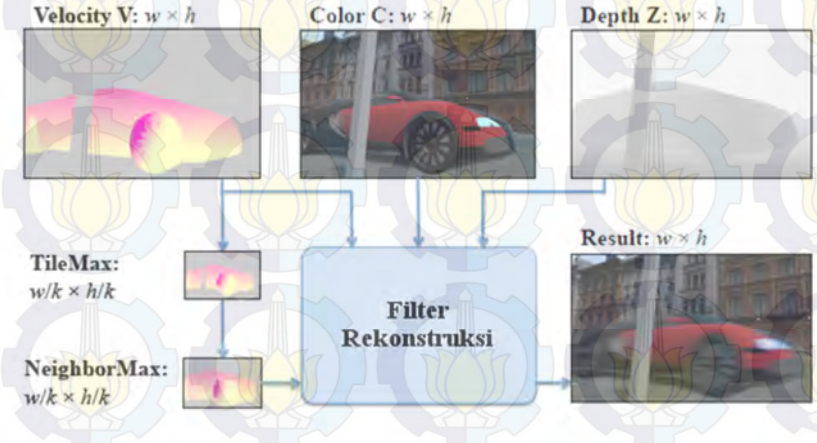
Proses gather ini dapat dioptimasi lagi dengan menggunakan ***buffer*** NeighborMax. NeighborMax berisi nilai velocity terbesar sepanjang rentang nilai ***k***. Contoh ***buffer*** NeighborMax bisa dilihat pada gambar (f). Dengan NeighborMax ini, iterasi ***gather*** menjadi proses 1D saja, sesuai dengan ***velocity*** terbesar, seperti terlihat pada gambar (g).

3.2. Gambaran umum algoritma

Input dan ***output*** dari algoritma filter rekonstruksi ***motion blur*** bisa dilihat pada Gambar 3.2. Implementasi algoritma ini menggunakan lima buffer. Secara ***default***, ***Buffer*** yang dipakai berukuran n = w x h. ***Input*** C (***color***) merupakan warna hasil ***render*** sebelum ***motion blur pass***. ***Input*** V (***velocity***) merupakan ***buffer*** yang berisi vektor selisih antara posisi ***pixel*** sekarang dengan posisi sebelumnya. Dan Z menyimpan kedalaman ***pixel*** relatif terhadap kamera.

Buffer TileMax dan NeighborMax merupakan ***intermediate buffer*** khusus yang dibuat untuk algoritma ini. Dua buffer ini menggunakan ukuran w/***k*** x h/***k***, dimana k merupakan radius

maksimal yang berefek pada panjang maksimal ***blur*** yang akan direkonstruksi.



Gambar 3.2 Diagram input output filter rekonstruksi [5]

Dengan berbagai ***input*** dan filter ***pass*** ini, filter rekonstruksi dilakukan untuk menghasilkan ***output*** gambar final. Filter rekonstruksi dilakukan untuk mendapat nilai ***blur*** pada ***pixel*** yang memiliki ***velocity*** rendah tetapi berdekatan dengan ***pixel*** yang memiliki ***velocity*** lebih tinggi. Caranya adalah dengan melakukan ***gathering*** data dari berbagai data ***input***. Proses gathering ini dilakukan dengan cara mengambil sampel pixel terdekat dengan batas radius k untuk mengetahui nilai blur pada pixel tersebut.

3.3. Input

Algoritma filter rekonstruksi menggunakan tiga input yaitu ***linear depth***, ***color***, dan ***velocity***.

3.3.1. Color

Color buffer merupakan hasil ***render scene*** yang disimpan pada ***framebuffer***. ***Buffer*** ini berisi warna yang tersusun dari channel RGB (***Red***, ***Green***, dan ***Blue***).

3.3.2. Linear depth

Gambar 3.3 Contoh tampilan linear depth buffer [23]

Linear depth memiliki distribusi nilai yang bersifat ***linear***, berbeda dengan ***hardware*** z-buffer yang sifatnya exponensial. Rentang nilai linear depth dimulai dari ***near clip*** (bernilai 0.0) hingga ***far clip*** (bernilai 1.0). Untuk menghitungnya relatif mudah, yaitu dengan membagi posisi z ***view space*** dengan ***far clip***. Gambar 3.3 adalah contoh tampilan ***linear depth*** ketika di-***render***. Semakin dekat dengan kamera, ***pixel*** berwarna lebih gelap yang berarti nilai ***pixel*** mendekati nilai 0.0, semakin jauh ***pixel*** dengan kamera, warna akan terlihat putih dikarenakan nilainya yang mendekati nilai 1.0.

Untuk contoh kodenya bisa dilihat di Kode Sumber 3.1.

```
1  // Posisi pada view space
   vec4 positionVS = WorldViewMatrix *
   vertexModelPos;
2  linearDepth = positionVS.z / farClip;
```

Kode Sumber 3.1 Contoh kode linear depth

3.3.3. Velocity Buffer

Velocity buffer merupakan ***array*** data yang berisi kecepatan ***pixel***. Kecepatan ***pixel*** ini dihitung dari posisi ***pixel*** saat ini dikurangi posisi ***pixel*** di ***frame*** sebelumnya.

Gambar 3.4 Cara menghitung velocity [19]

Pada Gambar 3.4, ***a*** merupakan posisi saat ini, dan ***b*** merupakan posisi di ***frame*** sebelumnya. Contoh sederhana kode untuk menghitung nilai ***velocity*** bisa dilihat pada Kode Sumber 3.2 dan Kode Sumber 3.3. Kode ini dijalankan untuk tiap objek tiap ***frame***. Kode ini didapat dari ***blog*** John Chapman [19]

```
1  uniform mat4 uWorldViewProjectionMat;
2  uniform mat4 uPrevWorldViewProjectionMat;
3
4  smooth out vec4 vPosition;
5  smooth out vec4 vPrevPosition;
6
```

```
7  void main(void) {
8     vPosition = uWorldViewProjectionMat * gl_Vertex;
9     vPrevPosition = uPrevWorldViewProjectionMat * gl_Vertex;
10
11    gl_Position = vPosition;
12 }
```

Kode Sumber 3.2 Contoh Velocity Vertex Shader

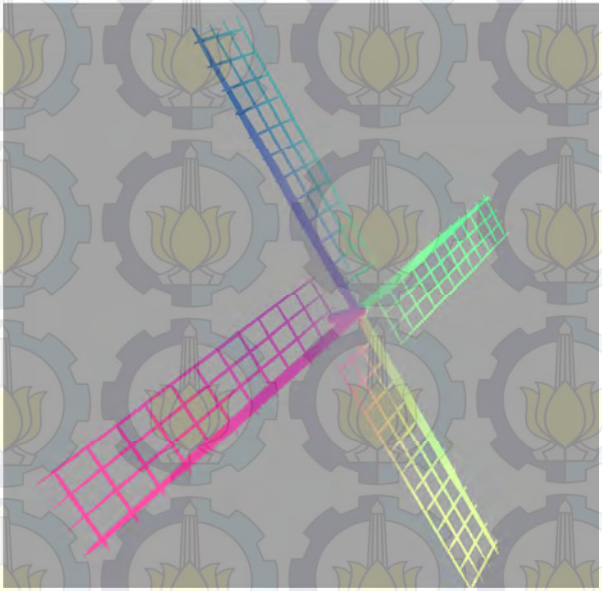
```
1  smooth in vec4 vPosition;
2  smooth in vec4 vPrevPosition;
3
4  out vec2 oVelocity;
5
6  void main(void) {
7     vec2 a = (vPosition.xy / vPosition.w) * 0.5 + 0.5;
8     vec2 b = (vPrevPosition.xy / vPrevPosition.w) * 0.5 + 0.5;
9     oVelocity = a - b;
10 }
```

Kode Sumber 3.3 Contoh Velocity Pixel Shader

Gambar 3.5 merupakan contoh ***velocity buffer*** jika di-***render***. Velocity buffer hanya menggunakan channel R dan G dan mempunyai range nilai dari -1.0 hingga 1.0. Dikarenakan framebuffer hanya menerima range nilai 0.0 hingga 1.0m maka velocity buffer harus diencode dengan dikalikan 0.5 dan ditambah 0.5 seperti pada baris 8 pada Kode Sumber 3.3. Ini yang membuat warna abu-abu pada background menunjukkan nilai RGB 0.5 (channel B menggunakan nilai default 0.5).

Warna-warna lain pada objek digambar menunjukkan nilai velocity pada pixel tersebut. Saat digunakan pada shader, velocity

buffer harus di-decode terlebih dahulu dengan dikalikan dengan 2.0 dan dikurangi 1.0 agar nilai kembali ke range semula (-1.0 hingga 1.0).



Gambar 3.5 Contoh tampilan velocity buffer [24]

3.4. Filter Pass

Algoritma filter rekonstruksi menghasilkan tiga 2D ***pass***, yaitu TileMax, NeighborMax, dan yang terakhir filter rekonstruksi itu sendiri.

3.4.1. TileMax dan NeighborMax

TileMax menyimpan ***velocity*** terbesar per ***tile*** berukuran ***k pixel*** oleh Persamaan (**3.1**). Fungsi vmax membandingkan panjang ***velocity*** terbesar.

$$\mathrm{TileMax}[x, y] = \max_{u \in [0,k)} \max_{v \in [0,k)} (\mathrm{V}[kx + u, ky + v])$$

(3.1)

Selanjutnya NeighborMax menghitung ***velocity*** pada tiap ***tile*** tetangga oleh Persamaan (**3.2)**.

$$\mathrm{NeighborMax}[x, y] = \max_{u \in [-1,1]} \max_{v \in [-1,1]} (\mathrm{TileMax}[x + u, y + v])$$

(3.2)

Dengan adanya NeighborMax ini, tiap ***pixel*** bisa memperkirakan tetangga terjauh yang akan mempengaruhi nilai ***velocity***-nya.. TileMax dan NeighborMax dijalankan dengan resolusi w/k x h/k.

3.4.2. Filter Rekonstruksi

Pass terakhir ini merupakan filter rekonstruksi itu sendiri. Filter ini dijalankan dengan resolusi penuh (w x h). Pada tiap pixel X, hitung kontribusi αY untuk tiap ***neighbor*** Y_i yang berada pada garis vektor ~vN. Jumlah Y_i yang tersebar bergantung dari nilai S (sampel). Semakin besar nilai S, semakin kecil jarak antara Y_i dan semakin banyak ***pixel*** yang dijadikan sampel.

Karena sifat blur yang transparan membuatnya bergantung pada ***pixel foreground*** dan ***backround***. Dengan ini dapat dirumuskan tiga skenario kasus ***blur*** yang akan muncul. Pada kasus 1, Y merupakan ***pixel*** yang ***blur*** yang berada didepan X. Ini adalah sumber dari artifak yang akan muncul nanti.

Pada kasus 2, X yang merupakan ***pixel blur*** sehingga ***pixel*** ini merupakan ***pixel*** transparan. Permasalahannya jika background ditutupi X sehingga nilai warna background tidak diketahui. Untuk itu, esatimasi warna ***background*** diambil dari nilai tetangga yang ***background-***nya terlihat.

Pada kasus 3, X dan Y merupakan ***pixel blur***, sehingga nilai ***blur-***nya berada ditengah-tengah dua ***velocity***.

Pseudo code dari filter rekonstruksi ini bisa dilihat pada Kode Sumber 3.4.

```
1  function reconstruct(X,C,V, Z, NeighborMax):
2  // Velocity terbesar di sekeliling pixel
3  Let ~vN = NeighborMax[[X/kc]]
4  if ||~vN|| ≤ε + 0.5px: return C[X] //
   velocity terlalu kecil(tidak blur)
5  // Sampel pixel saat ini
6  weight = 1.0/||V[X]||; sum = C[X] . weight
7  // Take S - 1 additional neighbor samples
8  let j = random(-0.5, 0.5)
9  for i ∈ [0, S); i ≠ (S -1)/2:
10 // Menentukan jarak antara sampel,
11 // dan ditambah jitter untuk mengurangi
   ghosting
12 t = mix (-1.0, 1.0, (i + j + 1.0)/(S + 1.0))
13 let Y = [X + ~vN . t + 0.5px] // pembulatan
   terdekat
14 // Klasisifikasi fore~ vs. Y relatif
   terhadap X
15 let f = softDepthCompare(Z[X], Z[Y])
16 let b = softDepthCompare(Z[Y], Z[X])
17 // Kasus 1: Y blur didepan X
18 αY = f . cone(Y, X, V[Y]) +
19 // Kasus 2: Y dibelakang X blur; estimasi
   background
20 b . cone(X, Y, V[X]) +
21 // Kasus 3: X blur dan Y blur secara
   bersamaan
22 cylinder(Y, X, V[Y]) . cylinder(X, Y, V[X])
   2
23 // Akumulasi
```

```
24  weight += Y ; sum += Y C[Y]
25  return sum/weight
```

Kode Sumber 3.4 Pseudo-code Filter Rekonstruksi

Untuk fungsi cone, cylinder, dan softDepthCompare, bisa dilihat pada Kode Sumber 3.5.

```
1  function cone(X, Y; ~v):
2      return clamp(1 - ||X - Y|| / ||~v||, 0, 1)
3  function cylinder(X; Y; ~v):
4      return 1.0 - smoothstep(0.95||~v||,
   1.05||~v||, ||X - Y||)
5  // Pada implementasi tugas akhir ini,
   menggunakan nilai SOFT_Z_EXTENT = 0.001
6  function softDepthCompare(za, zb):
7      return clamp(1 - (za - zb)/SOFT_Z_EXTENT,
   0, 1)
```

Kode Sumber 3.5 Filter klasifikasi

Fungsi ***cone*** digunakan untuk mengatahui apakah X berada pada Y ***blur***. Fungsi softDepthCompare digunakan untuk mengetahui apakah zb lebih dekat dari za.

3.5. Perancangan Sistem

Urho3D mengatur berbagai ***input buffer*** ini pada berkas RenderPath. Berbagai ***rendertarget*** yang dibutuhkan dan formatnya dalam Urho3D bisa dilihat pada Tabel 3.1.

Tabel 3.1 Input Rendertarget untuk Filter Rekonstruksi

Rendertarget	Format	Size Divisor
viewport	rgba	1 1
depth	lineardepth	1 1
velocity	rg16f	1 1
tilemax	rg16f	k k

neighbormax	rg16f	k k

Rendertarget viewport merupakan ***rendertarget default*** dari Urho3D. Format yang dipakai oleh ***viewport*** adalah rgba yang berarti buffer terdiri dari ***channel red, green***, ***blue***, dan ***alpha*** untuk transparansi. Tiap ***channel*** memiliki panjang 8 bit (256 variasi), yang membuatnya total menjadi 32 bit (~4 miliar variasi). ***Size divisor*** merupakan pembagi dari ukuran resolusi aplikasi, nilai '1 1' berarti ukuran ***rendertarget*** dibagi 1 berdasar sumbu x dan sumbu y. Ini berarti ***viewport*** memiliki ukuran sama dengan resolusi aplikasi.

Rendertarget depth merupakan ***buffer*** yang berguna untuk menyimpan nilai ***linear depth***. Format ***lineardepth*** merupakan format bawaan urho3D yang khusus untuk menyimpan nilai ini.

Rendertarget velocity menyimpan jarak ***pixel*** relatif posisi ***pixel*** di ***frame*** n-1. Format rg16f dipakai karena ***velocity*** merupakan nilai 2D atau hanya menggunakan sumbu x dan y. Daripada menggunakan format standar rgba (4 x 8 bit), lebih optimal jika menggunakan ***format*** rg16f (2 x 16 bit), dimana ***channel blue*** dan ***alpha*** tidak dipakai.

Rendertarget tilemax dan neighbor max merupakan ***buffer*** yang diproses dari ***velocity***. Buffer ini juga menggunakan format yang sama dengan ***velocity*** yaitu rg16f. ***Size divisor*** yang dipakai oleh ***buffer*** ini adalah nilai k, yaitu nilai ***blur*** maksimal yang direkonstruksi oleh filter rekonstruksi. Semakin besar nilai k berarti semakin kecil ukuran ***tilemax*** dan ***neighbormax***, tetapi semakin panjangnya panjang ***blur*** yang bisa direkonstruksi.

Sayangnya deklarasi nilai ***size divisor*** pada berkas ***renderpath*** merupakan nilai ***fix***, sehingga nilai k merupakan nilai ***hardcode*** yang tidak bisa diubah saat runtime aplikasi.

Data spesifik untuk berbagai rendertarget tersebut diperjelas pada bagian berikut.

3.5.1. Linear depth

Dikarenakan ***linear depth*** sudah tersedia pada instalasi Urho3D secara ***default***, maka yang diperlukan adalah:

- ***Command*** ‘depth’ pada file xml ***RenderPath***
- ***Rendertarget*** ‘depth’ pada file xml ***RenderPath***.

3.5.2. Velocity

Untuk melakukan implementasi ***velocity*** pada Urho3D, beberapa yang dibutuhkan adalah:

- ***Command*** ‘velocity’ pada file xml ***RenderPath***
- ***Rendertarget*** ‘velocity’ pada file xml ***RenderPath***.
- Pass ‘velocity’ pada technique yang dipakai oleh material dari objek
- ***Matrix*** PrevWorldViewProjection untuk menghitung posisi ***pixel*** pada ***frame*** sebelumnya.
- ***Shader*** ‘velocity.glsl’ yang merupakan ***shader*** yang akan dijalankan pada saat ***pass*** berjalan.

3.5.3. TileMax

TileMax merupakan ***intermediate buffer*** yang menggunakan ***input buffer velocity***. Untuk melakukan implementasi ***TileMax*** pada Urho3D, beberapa yang dibutuhkan adalah:

- ***Command*** ‘tilemax’ pada file xml ***RenderPath***
- ***Rendertarget*** ‘tilemax’ pada file xml ***RenderPath***.
- Shader ‘TileMax.glsl’ yang berisi instruksi shader yang akan dijalankan.

3.5.4. NeighborMax

NeighborMax merupakan ***intermediate buffer*** yang menggunakan ***input buffer*** TileMax. Untuk melakukan implementasi NeighborMax pada Urho3D, beberapa yang dibutuhkan adalah:

- ***Command*** ‘neighbormax’ pada file xml ***RenderPath***
- ***Rendertarget*** ‘neighbormax’ pada file xml ***RenderPath***.
- ***Shader*** ‘NeighborMax.glsl’ yang berisi instruksi ***shader*** yang akan dijalankan.

BAB IV
IMPLEMENTASI

Bab ini berisi proses implementasi dari filter rekonstruksi motion blur dan setiap komponen yang dibutuhkannya pada game engine Urho3D.

4.1. Linear Depth Buffer

Linear depth buffer cukup mudah didapatkan pada Urho3D. Pada renderpath tambahkan Kode Sumber 4.1 .

```
1  <!-- DEPTH -->
2  <rendertarget name="depth" sizedivisor="1 1"
   format="lineardepth" />
3  <command type="clear" color="1 1 1 1"
   depth="1.0" output="depth" />
4  <command type="scenepass" pass="depth"
   output="depth" />
```

Kode Sumber 4.1 RenderPath command Linear Depth

Depth buffer akan disimpan pada rendertarget bernama ***depth***.

4.2. Velocity Buffer

Implementasi ***velocity buffer*** tidak tersedia secara ***default*** pada Urho3D. Implementasi ***velocity buffer*** juga membutuhkan ***matrix*** yang tidak tersedia pada Urho3D, yaitu PrevWorldViewProjection (WorldViewProjection yang didapat dari ***frame*** n-1). ***Matrix*** ini bisa didapat dengan mengalikan tiga ***matrix*** lain seperti pada Kode Sumber 4.2.

```
1  Matrix4 PrevWorldViewProjection =
   PrevWorldMat * PrevViewMat *
   PrevProjectionMat;
```

Kode Sumber 4.2 Cara mendapatkan WorldViewProjection Matrix pada frame n-1

Dikarenakan ketiga ***matrix*** tersebut tidak tersedia pada Urho3D, maka diperlukan untuk mengubah ***source code*** dari ***game engine*** ini sendiri. Cara untuk mendapat PrevViewMat dan PrevProjectionMat adalah dengan menambah fungsi pada kelas Camera pada ***source code*** Urho3D agar menyimpan matrix dari ***frame*** sebelumnya setiap kali ***update***.

Untuk PrevWorldMat agak sedikit rumit dikarenakan setiap objek memiliki ***matrix world*** yang berbeda. Sayangnya Urho3D tidak menyediakan cara mudah untuk menambah uniform ***matrix*** spesifik per-objek. Beberapa kelas sepeti Batch, Node, dan StaticModel perlu ditambah fungsi agar menyimpan WorldMatrix dari frame sebelumnya. Penjelasan singkat tugas dari masing-masing kelas tersebut adalah, kelas Node yang menyimpan matrix world, kelas StaticModel yang meng-***update*** nilai matrix, dan kelas Batch yang akan mengirim nilainya ke GPU.

Pada Urho3D, ***velocity*** di-***render*** tiap awal ***frame*** sebelum proses ***render*** warna yang sesungguhnya. Instruksi ***render*** ini harus didefinisikan pada ***resource techniques*** dan ***renderpath***.

Tambahan perintah baru pada ***renderpath*** untuk memenuhi kebutuhan ini. Perintah ini dijalankan sebelum mulai proses ***render*** yang lain dimulai tiap ***frame***. Perintah tersebut bisa dilihat pada Kode Sumber 4.3.

```
1  <!-- VELOCITY -->
2  <rendertarget name="velocity" sizedivisor="1
   1" format="rg16f" />
3  <command type="clear" color="0.5 0.5 0.5 1"
   depth="1.0" output="velocity" />
4  <command type="scenepass" pass="velocity"
   output="velocity" />
```

Kode Sumber 4.3 RenderPath command Velocity

Selain deklarasi pada ***renderpath***, ***velocity*** juga harus didefinisikan pada berkas ***technique*** yang dipakai oleh objek yang

ingin di-***render velocity***-nya. Deklarasi tersebut bisa dilihat pada Kode Sumber 4.4.

```
1  <pass name="velocity" vs="Velocity"
   ps="Velocity" />
```

Kode Sumber 4.4 Deklarasi Velocity pada berkas techniques

Jika dilihat, pass ***velocity*** pada ***technique*** tersebut menggunakan ***vertex*** dan ***pixel shader Velocity***. Ini berarti ***shader*** yang dipakai bernama Velocity.glsl. Isi dari ***shader*** tersebut seperti bisa dilihat pada Kode Sumber 4.5.

```
1  varying vec3 vTexCoord;
2  varying vec4 vPosition;
3  varying vec4 vPrevPosition;
4  varying vec4 vPrevScreenPos;
5
6  void VS()
7  {
8      mat4 modelMatrix = iModelMatrix;
9      vec3 worldPos =
   GetWorldPos(modelMatrix);
10     vec3 prevWorldPos = GetPrevWorldPos();
11     gl_Position = GetClipPos(worldPos);
12     vPosition = gl_Position;
13     vPrevPosition =
   GetPrevClipPos(prevWorldPos);
14     vPrevScreenPos =
   GetPrevClipPos(worldPos);
15     vTexCoord = vec3(GetTexCoord(iTexCoord),
   GetDepth(gl_Position));
16 }
17
18 void PS()
```

```
19  {
20      vec2 curr = (vPosition.xy / vPosition.w);
21      vec2 prev = (vPrevPosition.xy / vPrevPosition.w);
22      vec2 velocity = (curr - prev) * 0.5;
23
24      gl_FragColor = vec4(velocity * 0.5 + 0.5, 0.0, 1.0);
25  }
```

Kode Sumber 4.5 Isi shader Velocity.glsl

Fungsi GetPrevClipPos() berada pada Transform.glsl. Bagian kodenya bisa dilihat pada Kode Sumber 4.6.

```
1  vec4 GetPrevClipPos(vec3 worldPos){
2      vec4 ret = cPrevViewProj * vec4(worldPos, 1.0);
3      return ret;
4  }
```

Kode Sumber 4.6 Fungsi GetPrevClipPos() pada Transform.glsl

cPrevViewPos merupakan ***uniform*** yang dideklarasikan pada Uniform.glsl. Uniform ini didapat dari kode yang diubah pada ***source code*** Urho3D.

4.3. TileMax Buffer

Perintah ***tilemax*** pada ***renderpath*** bisa dilihat pada Kode Sumber 4.7.

```
1  <!-- TILE MAX -->
2  <rendertarget name="tilemax" sizedivisor="8 8" format="rg16f" />
3  <command tag="TileMax" type="quad" vs="TileMax" ps="TileMax" output="tilemax">
4      <texture unit="diffuse" name="velocity" />
```

```
5     <parameter name="TexelSize" value="0 0" />
6     <parameter name="MaxVelocity" value="8" />
7   </command>
```

Kode Sumber 4.7 RenderPath command TileMax

Pada kode tersebut, nilai ***sizedivisor*** menggunakan nilai '8 8' yang berarti ukuran ***buffer*** 1/8 dari resolusi aplikasi. Ini berarti panjang ***blur*** maksimal adalah 8 ***pixel***. Nilai ini yang diganti jika menginginkan ***paramater*** k pada ***paper*** diubah.

Shader dari TileMax.glsl sendiri bisa dilihat pada Kode Sumber 4.8.

```
1   varying vec2 vScreenPos;
2   uniform vec2 cTexelSize;
3   uniform float cMaxVelocity;
4
5   void VS()
6   {
7       mat4 modelMatrix = iModelMatrix;
8       vec3 worldPos = GetWorldPos(modelMatrix);
9       gl_Position = GetClipPos(worldPos);
10      vScreenPos = GetScreenPosPreDiv(gl_Position);
11  }
12
13  vec2 vmax(vec2 a, vec2 b)
14  {
15  float ma = dot(a, a);
16  float mb = dot(b, b);
17  return (ma > mb) ? a : b;
```

```
18  }
19
20  void PS()
21  {
22      vec2 uvCorner = vScreenPos - cTexelSize
    * (cMaxVelocity * 0.5);
23      vec2 maxVel = vec2(0.0);
24      vec2 uvScale = cTexelSize;
25
26      for(float l = 0.0; l < cMaxVelocity; l
    += 1.0)
27  {
28     for(float k = 0.0; k < cMaxVelocity; k +=
    1.0)
29     {
30          maxVel = vmax(maxVel,
    texture2D(sDiffMap, uvCorner + vec2(l, k) *
    uvScale).rg * 2.0 - 1.0);
31     }
32  }
33
34      gl_FragColor = vec4(maxVel * 0.5 + 0.5,
    0.0 , 1.0);
35
36  }
```

Kode Sumber 4.8 Isi shader TileMax.glsl

4.4. NeighborMax Buffer

Perintah ***neighbormax*** pada ***renderpath*** bisa dilihat pada Kode Sumber 4.9.

```
1  <!-- NEIGHBOR MAX -->
2  <rendertarget name="neighbormax"
   sizedivisor="8 8" format="rg16f" />
```

```
3  <command tag="NeighborMax" type="quad"
   vs="NeighborMax" ps="NeighborMax"
   output="neighbormax">
4      <texture unit="diffuse" name="tilemax"
   />
5      <parameter name="TexelSize" value="0 0"
   />
6      <parameter name="MaxVelocity" value="8"
   />
7  </command>
```

Kode Sumber 4.9 RenderPath command NeighborMax

Sama seperti ***tilemax***, ***sizedivisor*** yang dipakai merupakan '8 8'. Selain itu juga, terdapat parameter tambahan yang harus ditambah pada ***shader***, yaitu MaxVelocity, yang juga bernilai 8. Nilai sizedivisor dan MaxVelocity yang harus diganti jika parameter ***k*** pada ***paper*** ingin diganti.

Untuk ***shader*** NeighborMax.glsl bisa dilihat pada Kode Sumber 4.10.

```
1  varying vec2 vScreenPos;
2  uniform vec2 cTexelSize;
3  uniform float cMaxVelocity;
4
5  void VS()
6  {
7      mat4 modelMatrix = iModelMatrix;
8      vec3 worldPos =
   GetWorldPos(modelMatrix);
9      gl_Position = GetClipPos(worldPos);
10     vScreenPos =
   GetScreenPosPreDiv(gl_Position);
11 }
12
```

```
13  vec2 vmax(vec2 a, vec2 b)
14  {
15  float ma = dot(a, a);
16  float mb = dot(b, b);
17  return (ma > mb) ? a : b;
18  }
19
20  void PS()
21  {
22      vec2 tileTexelSize = cTexelSize *
    cMaxVelocity;
23
24  vec2 nx =     texture2D(sDiffMap, vScreenPos
    + vec2(1.0, 1.0) * tileTexelSize).rg * 2.0 -
    1.0;
25  nx = vmax(nx, texture2D(sDiffMap, vScreenPos
    + vec2(1.0, 0.0) * tileTexelSize).rg * 2.0 -
    1.0);
26  nx = vmax(nx, texture2D(sDiffMap, vScreenPos
    + vec2(1.0,-1.0) * tileTexelSize).rg * 2.0 -
    1.0);
27  nx = vmax(nx, texture2D(sDiffMap, vScreenPos
    + vec2(0.0, 1.0) * tileTexelSize).rg * 2.0 -
    1.0);
28  nx = vmax(nx, texture2D(sDiffMap, vScreenPos
    + vec2(0.0, 0.0) * tileTexelSize).rg * 2.0 -
    1.0);
29  nx = vmax(nx, texture2D(sDiffMap, vScreenPos
    + vec2(0.0,-1.0) * tileTexelSize).rg * 2.0 -
    1.0);
30  nx = vmax(nx, texture2D(sDiffMap, vScreenPos
    + vec2(-1.0, 1.0) * tileTexelSize).rg * 2.0
    - 1.0);
31  nx = vmax(nx, texture2D(sDiffMap, vScreenPos
    + vec2(-1.0, 0.0) * tileTexelSize).rg * 2.0
```

```
    - 1.0);
    nx = vmax(nx, texture2D(sDiffMap, vScreenPos
    + vec2(-1.0,-1.0) * tileTexelSize).rg * 2.0
32  - 1.0);
33
        gl_FragColor = vec4(nx * 0.5 + 0.5, 0.0
34  , 1.0);
35  }
```

Kode Sumber 4.10 Isi shader NeighborMax.glsl

4.5. Filter Rekonstruksi

Untuk filter rekonstruksi, ***command*** pada RenderPath adalah seperti pada Kode Sumber 4.11.

```
1   <!-- MOTION BLUR -->
2   <command tag="MotionBlurReconstruction"
    type="quad" vs="MotionBlurReconstruction"
    ps="MotionBlurReconstruction"
    output="viewport">
3   <texture unit="diffuse" name="viewport" />
4   <texture unit="specular" name="neighbormax"
    />
5   <texture unit="environment" name="velocity"
    />
6   <texture unit="emissive"
    name="Textures/Jitter.png" />
7   <texture unit="depth" name="depth" />
8   <parameter name="Samples" value="7" />
9   <parameter name="SoftZDistance"
    value="0.0001" />
10  <parameter name="MaxVelocity" value="8" />
11  </command>
```

Kode Sumber 4.11 RenderPath command Motion Blur

Dari lima ***buffer*** yang disebutkan sebelumnya, empat darinya dijadikan ***input*** pada filter rekonstruksi ini. Selain itu

diperlukan juga tekstur ***jitter*** yang diperuntukkan untuk mendapat nilai ***random*** pada ***shader***.

Parameter Samples (S) adalah banyak sampel yang dipakai untuk melakukan proses ***blur***. Parameter SoftZDistance pada paper dijelaskan untuk membedakan apakah ***pixel*** yang disampel ada didepan atau dibelakang ***pixel*** lain. MaxVelocity sama seperti sebelumnya, yaitu nilai ***k*** pada ***paper***.

Untuk ***shader*** MotionBlurReconstruction.glsl bisa dilihat pada Kode Sumber 4.12.

```
1  uniform float cSamples;
2  uniform float cSoftZDistance;
3  uniform vec2 cTexelSize;
4  uniform float cMaxVelocity;
5
6  varying vec2 vScreenPos;
7
8  void VS()
9  {
10     mat4 modelMatrix = iModelMatrix;
11     vec3 worldPos =
   GetWorldPos(modelMatrix);
12     gl_Position = GetClipPos(worldPos);
13     vScreenPos =
   GetScreenPosPreDiv(gl_Position);
14 }
15
16 // classification filters
17 float cone(vec2 px, vec2 py, vec2 v)
18 {
19     float vlen = max(length(v), 0.001);
```

```
20      return clamp(1.0 - (length(px - py) / vlen), 0.0, 1.0);
21  }
22
23  float cylinder(vec2 x, vec2 y, vec2 v)
24  {
25      float lv = length(v);
26      return 1.0 - smoothstep(0.95 * lv, 1.05 * lv, length(x - y));
27  }
28
29  // is zb closer than za?
30  float softDepthCompare(float za, float zb)
31  {
32      return clamp((za - zb) / cSoftZDistance, 0.0, 1.0);
33  }
34
35  void PS()
36  {
37      /////// RECONSTRUCTION ///////
38      vec2 x = vScreenPos;
39      vec2 tileTexelSize = cTexelSize * cMaxVelocity;
40
41      vec2 vn = texture2D(sSpecMap, x).rg * 2.0 - 1.0; // largest velocity in neighbourhood
42      vn = clamp(vn, -tileTexelSize, tileTexelSize); // clamp vn to max velocity
43      vec4 cx = texture2D(sDiffMap, x); // color at x
44      vec2 vx = texture2D(sEnvMap, x).rg * 2.0
```

```
    - 1.0; // vel at x
    vx = clamp(vx, -tileTexelSize,
45  tileTexelSize); // clamp vx to max velocity
46
    float zx =
47  DecodeDepth(texture2D(sDepthBuffer, x).rgb);
48
49      // Random offset
    float j = texture2D(sEmissiveMap, x *
50  16.0).r - 0.5; // random(-0.5, 0.5)
51
52      // sample current pixel
    float weight = 0.75; // <= good start
53  weight choice??
54      vec4 sum = cx * weight;
55
    float centerSample = (cSamples - 1.0) /
56  2.0;
57
    for(float i = 0.0; i < cSamples; i +=
58  1.0)
59      {
60          if (i == centerSample) continue;
61
            // Choose evenly placed filter taps
62  along vN,
            // but jitter the whole filter to
63  prevent ghosting
            // t = mix(-1.0, 1.0, (i + j + 1.0)
64  / (S + 1.0))
            float t = mix(-1.0, 1.0, (i + j +
65  1.0) / (cSamples + 1));
66
67          // let Y = [X + vN . t + 0.5px] //
```

```
    round to nearest
68          vec2 y = x + vn * t + (0.5 *
cTexelSize);
69
70          // velocity at y
71          vec2 vy = texture2D(sEnvMap, y).rg *
2.0 - 1.0;
72          vy = clamp(vy, -tileTexelSize,
tileTexelSize); // clamp vy to max velocity
73
74          // linear z at y
75          float zy =
DecodeDepth(texture2D(sDepthBuffer, y).rgb);
76
77          float f = softDepthCompare(zx, zy);
// is y in front x?
78          float b = softDepthCompare(zy, zx);
// is x in front y?
79
80          float alphay =  f * cone(y, x, vy) +
81                          b * cone(x, y, vx) +
82                          cylinder(y, x, vy) *
cylinder(x, y, vx) * 2.0;
83
84          vec4 cy = texture2D(sDiffMap, y);
85          sum += cy * alphay;
86          weight += alphay;
87      }
88      sum /= weight;
89
90      gl_FragColor = sum;
91  }
```

Kode Sumber 4.12 Isi shader MotionBlurReconstruction.glsl

BAB V
PENGUJIAN DAN EVALUASI

Bab ini membahas pengujian dan evaluasi pada kualitas gambar dan performa dari implementasi ***filter*** rekonstruksi pada ***game engine*** Urho3D.

5.1. Lingkungan Pengujian

Pengujian implementasi ***motion blur filter*** rekonstruksi ini dijalankan pada AMD E-350 1,6Ghz ***dual-core*** CPU yang menjalankan Windows 7 x64 dengan 2GB RAM. GPU yang adalah AMD Radeon HD 6310 dengan ***memory*** yang di-***share*** dari RAM. ***Hardware*** ini merupakan ***hardware*** yang sudah berumur empat tahun saat buku ini ditulis dan juga termasuk ***hardware low-end*** pada saat itu. Dengan begitu, dengan GPU yang lebih modern saat ini, implementasi filter rekonstruksi seharusnya akan berjalan lebih cepat.

Versi Urho3D yang dipakai untuk implementasi adalah Urho3D master ***branch*** 15 Desember 2014. API yang dipakai adalah OpenGL 2.0.

Untuk pengujian, terdapat tiga ***scene*** yang dipakai, yang pertama adalah ***scene*** berisi ***teapot*** bergerak melewati beberapa objek lain. ***Scene*** kedua sama seperti ***scene*** yang pertama tetapi dengan kincir angin yang bergerak. ***Scene*** yang terakhir merupakan scene kendaraan sederhana mengambil dari contoh 19_VehicleDemo bawaan Urho3D.

Pengujian performa dilakukan menggunakan kelas Profiler bawaan Urho3D. Dikarenakan Urho3D belum bisa melakukan ***profiler*** terhadap ***renderpath command*** tertentu, nilai yang diuji adalah lama waktu satu ***frame*** untuk di-***render*** dalam satuan ***ms***. Data diambil saat demo dan ***profiler*** dijalankan.

5.2. Pengujian Velocity

Gambar 5.1 merupakan gambar yang di-***render*** menggunakan ***teapot*** yang bergerak kekiri dan kekanan.

Color buffer scene *Teapot*

Velocity buffer saat *teapot* bergerak kekiri

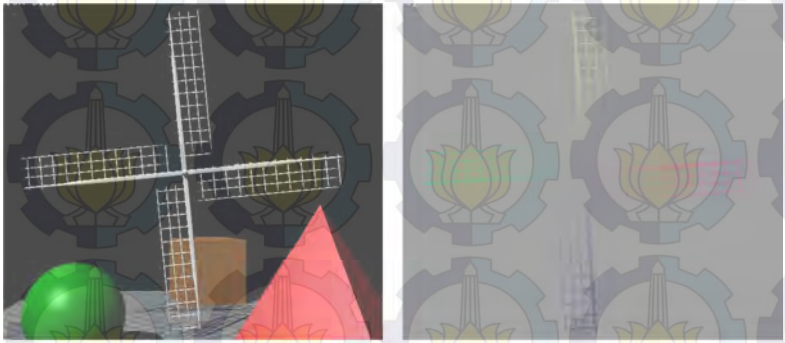
Velocity buffer saat *teapot* bergerak kekanan

Gambar 5.1 Velocity buffer pada scene teapot

Gambar kiri bawah menunjukkan hasil ***velocity buffer*** saat ***teapot*** bergerak kearah kiri. Nilai RGB pada ***teapot*** menunjukkan nilai R=0.26 G= 0.5 B=0.5. Dikarenakan nilai R menunjukkan ***velocity*** pada sumbu X dengan ***range*** nilai 0.0 hingga 1.0, nilai R=0.26 menunjukkan ***teapot*** bergerak kearah negatif X sejumlah 0.25.

Sedangkan gambar kanan bawah merupakan ***velocity buffer*** saat ***teapot*** bergerak kekanan. Hasilnya bernilai RGB R=0.74 G=0.5 B=0.5. Dengan nilai tersebut berarti ***teapot*** ini bergerak kearah sumbu X sejumlah 0.25 unit.

Untuk ***scene*** selanjutnya, sebuah ***fan*** (kipas) bergerak berputar kearah jarum jam. Hasil ***velocity buffer***-nya bias dilihat pada Gambar 5.2.



Color buffer scene *Fan* Velocity buffer saat *fan* berputar

Gambar 5.2 Velocity buffer pada scene fan

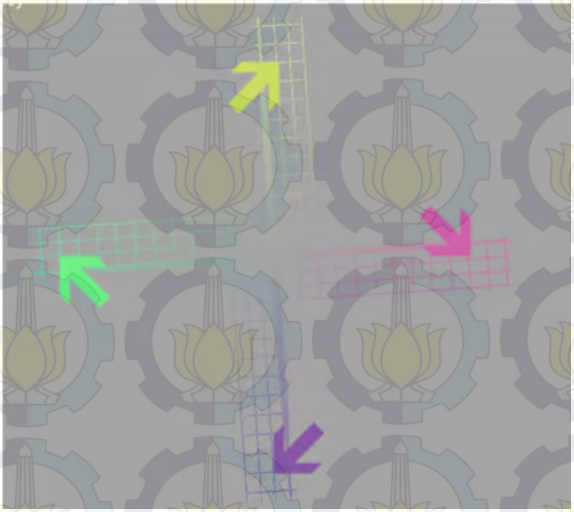
Hasil ***velocity*** tersebut menunjukkan berbagai warna berbeda di setiap pixel fan. Bagian atas kecenderungan ***pixel*** berwarna kuning (R > 0.5 dan G > 0.5). Sebagai catatan, arah positif sumbu Y pada Urho3D merupakan arah keatas. Dengan ini berarti bagian atas fan bergerak kearah kanan bawah.

Untuk data lebih lengkapnya, bias dilihat pada Tabel 5.1.

Tabel 5.1 Warna pixel pada fan

Bagian	Warna	Nilai R	Nilai G	Arah gerak
atas	kuning	> 0.5	> 0.5	kanan atas
kiri	hijau muda	< 0.5	> 0.5	kiri atas
bawah	ungu	< 0.5	< 0.5	kiri bawah
kanan	merah muda	> 0.5	< 0.5	kanan bawah

Jika arah gerak digambarkan pada ***velocity***, maka bisa di ilustrasikan pada Gambar 5.3. Dari gambar ini dapat terlihat bahwa data ***velocity*** yang ditampilkan sesuai dengan arah gerakan ***fan*** yang bergerak sesuai arah jarum jam.

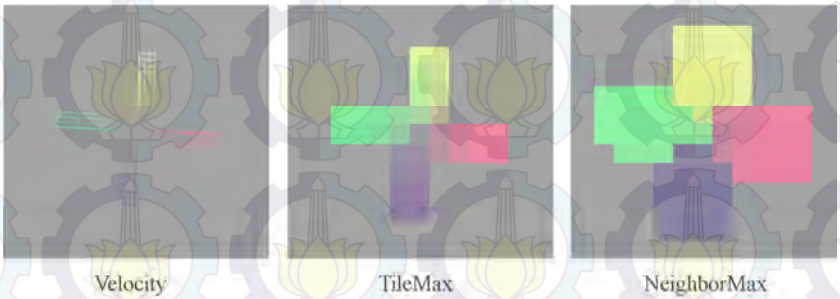


Gambar 5.3 Arah gerak tiap bagian fan

Dua percobaan ini menunjukkan implementasi ***velocity buffer*** telah bekerja dengan semestinya.

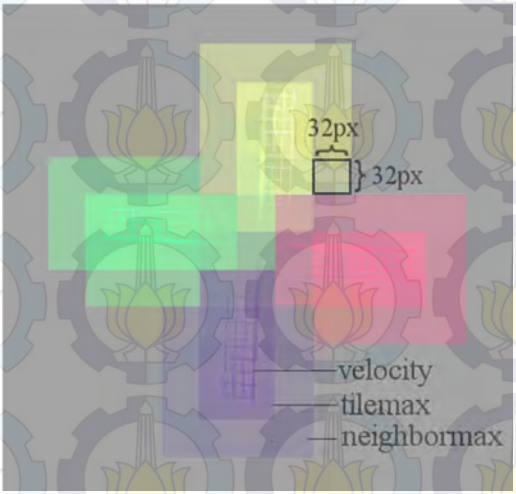
5.3. Pengujian TileMax dan NeighborMax

Gambar 5.4 menunjukkan input ***velocity*** dan hasil ***intermediate buffer*** TileMax dan NeighborMax pada ***scene fan***. Nilai ***k*** yang dipakai pada gambar ini adalah 32 ***pixel***.



Gambar 5.4 Velocity, TileMax, dan NeighborMax pada scene fan

Dari hasil tersebut jika masing-masing ***buffer*** di-***overlay***, akan terlihat seperti Gambar 5.5. Terlihat TileMax memenuhi velocity dan NeighborMax memperluas ukuran TileMax. Ukuran dari ***tile*** juga sama seperti nilai ***k***, yaitu 32 ***pixel***. Dari gambar ini bias disimpulkan bahwa hasil dari implementasi TileMax dan NeighborMax ini berkerja dengan baik.



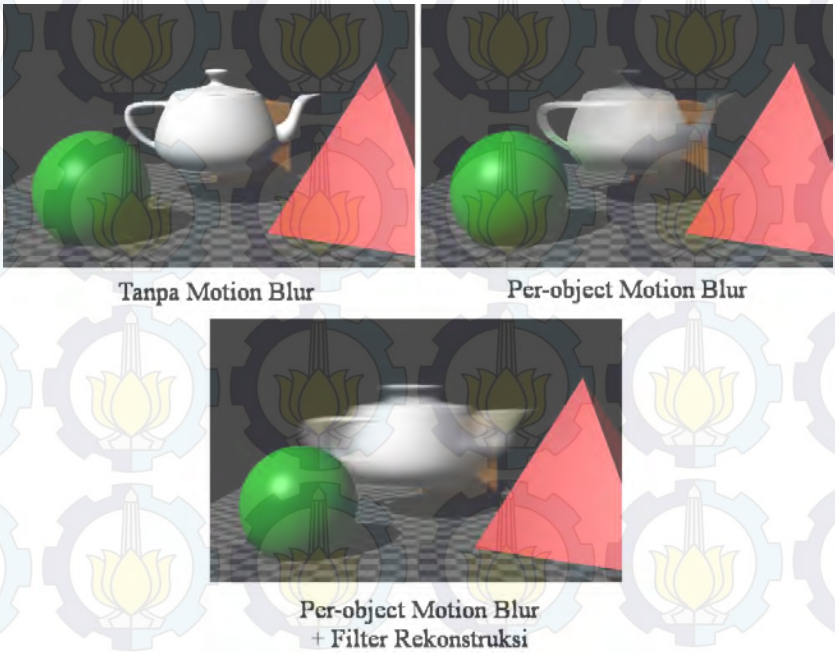
Gambar 5.5 Overlay TileMax dan NeighborMax pada velocity

5.4. Pengujian filter rekonstruksi

Gambar 5.6 merupakan merupakan screenshot yang diambil dari aplikasi demo yang dibuat menggunakan Urho3D. Demo ini menggunakan resolusi 800x600 (Pada screenshot, sebagian dari gambar telah dipotong agar demonstrasi terlihat lebih jelas). Parameter yang dipakai dari paper menggunakan nilai k = 32 dan S = 12.

Model ***teapot*** pada Gambar 5.6 bergerak dari kiri kekanan. Pada gambar kiri atas, ***motion blur*** tidak dipakai. Pada gambar kanan atas menggunakan ***per-object motion blur*** dengan hanya menggunakan ***velocity buffer***. Objek ***teapot*** terlihat kabur tetapi tepi dari objek terlihat tegas. Bagian kiri bawah ***teapot*** juga

terlihat agak hijau dikarenakan proses ***blur*** mengambil ***pixel*** tetangga tanpa mengetahui ***pixel*** tersebut ada didepan objek.

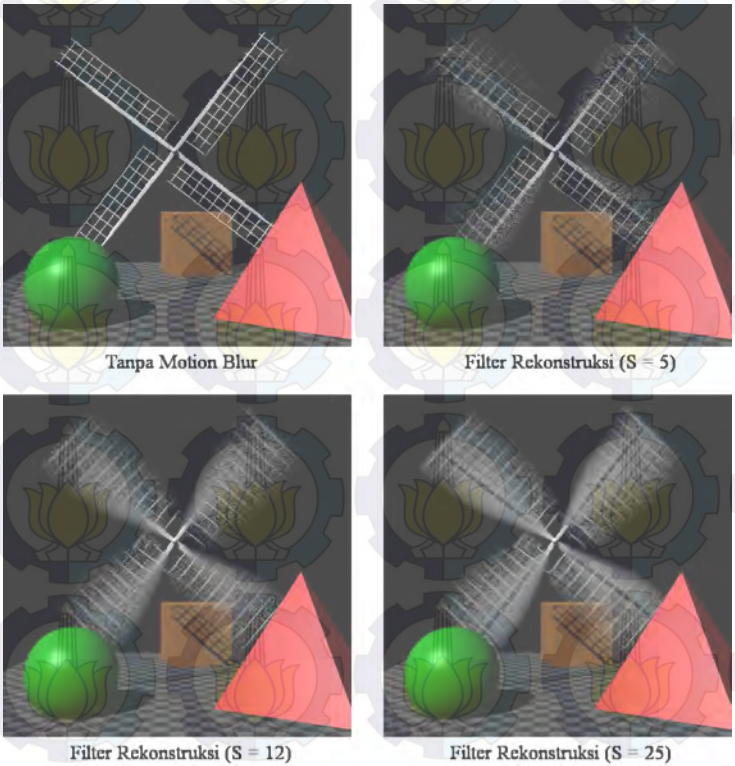


Gambar 5.6 Perbandingan berbagai motion blur

Gambar yang bawah menggunakan ***per-object motion blur*** dengan tambahan filter rekonstruksi. Terlihat teapot kabur sesuai dengan arah gerakan dan terdapat transparansi dengan ***background***. Bagian kiri bawah ***teapot*** juga tidak terdapat warna hijau seperti implementasi pada gambar kanan atas. Dari gambar tersebut bias disimpulkan bahwa filter rekonstruksi telah bekerja dengan semestinya.

5.5. Kualitas Gambar

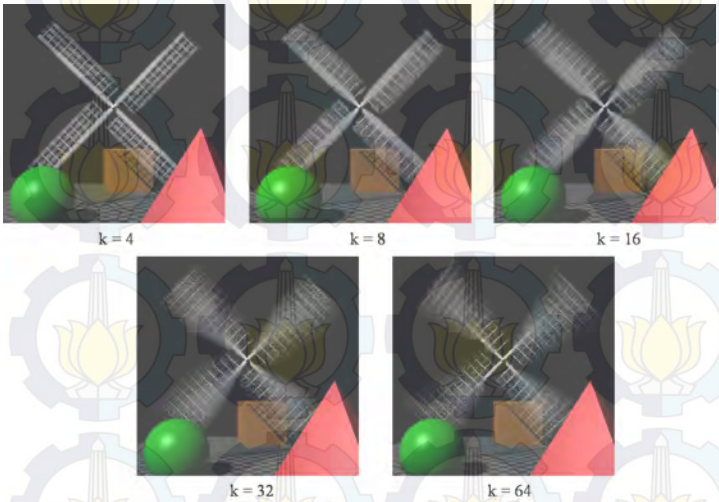
Gambar 5.7 menunjukkan efek pengaruh nilai S (sampel) pada kipas angin yang berotasi. Semakin tinggi nilai S, semakin halus blur dan semakin berkurangnya efek bintik-bintik yang disebabkan oleh ***jitter***.



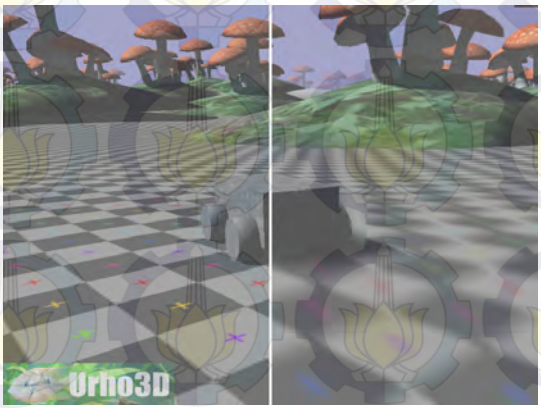
Gambar 5.7 Pengaruh variabel S terhadap kualitas gambar

Gambar 5.8 menunjukkan pengaruh nilai k (kecepatan maksimal). Semakin besar nilai k berpengaruh semakin jauhnya blur yang terlihat. Tetapi semakin besar nilai k juga berefek pada semakin terlihatnya efek blok-blok pada gambar.

Gambar 5.9 diambil dari contoh 19_Vehicle bawaan Urho3D yang telah dimodifikasi.. Sebelah kiri tanpa ***motion blur*** dan sebelah kanan dengan ***motion blur*** filter rekonstruksi dengan k=16 dan S=12. Biarpun secara umum terlihat halus, beberapa artifak blok-blok masih terlihat.



Gambar 5.8 Pengaruh variabel *k* terhadap kualitas gambar



Gambar 5.9 Motion blur dengan filter rekonstruksi pada scene 19_Vehicle

5.6. Performa

Pada bagian sebelumnya telah dijelaskan bahwa semakin besar nilai S (sampel), semakin sedikitnya ***noise*** yang terlihat. Akan tetapi seberapa besar pengaruh nilai ini pada ***render time***. Variasi nilai S menghasilkan data seperti pada Tabel 5.2. Pada tes ini, nilai K seluruhnya dibuat 32.

Tabel 5.2 Pengaruh variabel S pada frametime

k	S	Frametime (ms)
32	5	5,243
32	6	6,041
32	7	5,273
32	8	5,205
32	9	5,273
32	10	5,687
32	11	5,26
32	12	5,715
32	13	6,072
32	14	8,029
32	15	10,175
32	16	18,461
32	17	18,808
32	18	24,632
32	19	25,633
32	20	24,904
32	21	26,797
32	22	30,728
32	23	33,175
32	24	34,387
32	25	34,199

Ada beberapa anomali dari data diatas, contohnya nilai S=6 menghabiskan waktu 6,041ms sedangkan S=7 sampai S=12 menghabiskan waktu dibawah 6,0ms. Hal ini bisa terjadi dikarenakan tidak stabilnya ***frame*** jika ***profiler*** dijalankan. Biarpun begitu, secara umum ***frametime*** bertambah ketika nilai S semakin besar.

Jika diperhatikan lagi dari tabel tersebut, ***frametime*** hingga S=13 cenderung tidak terlalu jauh. Ketika menggunakan S=14, ***frametime*** langsung melonjak menjadi 8ms. Nilai S=15 keatas, ***frametime*** naik begitu melonjak dan tidak cocok lagi untuk ***realtime rendering***. Sebagai catatan, untuk mengasilkan ***game*** dengan target 30fps, ***budget*** dalam satu ***frame*** adalah 33ms untuk seluruh proses ***render*** dan ***gameplay***. ***Motion blur*** hanya sebagian dari dari berbagai proses yang lain.

Biarpun terlihat perbedaan noise pada nilai S=12 dengan S=25 di Gambar 5.7, saat bergerak perbedaan keduanya sebenarnya tidak terlalu jauh dikarenakan cepatnya gambar bergerak. Dengan begitu nilai S=12 atu S=13 bisa dibilang sudah cukup baik untuk bisa diintegrasi pada ***game*** yang sesungguhnya.

Tabel 5.3 Pengaruh variabel k pada frametime

k	S	Frametime (ms)
4	12	6,214
8	12	5,509
16	12	5,583
32	12	6,045
64	12	6,271

Untuk variasi nilai ***k*** bisa dilihat pada Tabel 5.3. Bisa dilihat bahwa tidak ada perbedaan signifikan. Pada bagian sebelumnya dijelaskan bahwa semakin besar nilai ***k***, semakin jauh jangkauan ***blur***, akan tetapi semakin munculnya artifak berbentuk kotak-kotak pada gambar.

Dengan begitu nilai K ini benar-benar sesuai kebutuhan, untuk ***scene*** yang rumit dan banyak ***ovelap*** objek, lebih baik menggunakan nilai k yang lebih kecil. Tetapi hal ini akan berefek pada jangkauan ***blur*** yang kecil. Untuk ***scene*** yang lebih sederhana , nilai ***k*** yang lebih tinggi lebih akan lebih memungkinkan.

BAB VI
KESIMPULAN DAN SARAN

Pada bab ini akan diberikan kesimpulan yang diambil selama pengerjaan tugas akhir serta saran-saran tentang pengembangan yang dapat dilakukan terhadap tugas akhir ini di masa yang akan datang.

6.1. Kesimpulan

Dari hasil selama proses perancangan, implementasi, serta pengujian dapat diambil kesimpulan sebagai berikut:

1. ***Buffer velocity*** digunakan untuk menyimpan jarak ***pixel*** sekarang (***frame*** n) relatif dengan posisi ***pixel*** pada ***frame*** sebelum ***frame*** sekarang (***frame*** n-1). Cara mendapatkannya posisi ***pixel*** pada ***frame*** n-1 adalah dengan mengirim transformasi objek ***frame*** n-1 ke ***frame*** n.
2. ***Buffer*** TileMax dan NeighborMax bisa didapat dengan mencari ***velocity*** terbesar pada suatu ***tile k*** x ***k***..
3. Filter rekonstruksi bekerja dengan ***input*** Color, Linear depth, Velocity, dan NeighborMax. Cara algoritma ini bekerja adalah dengan cara melakukan proses ***gathering*** nilai ***velocity pixel*** tetangga dengan bantuan NeighborMax ***buffer***.
4. Untuk mengimplementtasi ***filter*** rekonstruksi pada Urho3D pertama kali adalah menyiapkan ***buffer***-***buffer*** yang dibutuhkan. ***Velocity buffer*** membutuhkan penambahan ***source code*** dikarenakan tidak adanya tersedianya transformasi ***frame*** n-1. Untuk buffer yang lain bisa langsung ditulis ***shader***-nya dan didefinisikan pada RenderPath. ***Filter*** rekonstruksi juga bisa dibuat sebagai salah satu perintah pada RenderPath yang memanggil ***shader***.
5. Nilai S (sampel) berpengaruh pada banyaknya ***noise*** pada gambar, semakin besarnya nilai S, semakin sedikirnya ***noise***. Nilai ***k*** berpengaruh pada jangkauan ***blur***, semakin besarnya nilai ***k*** berpengaruh pada semakin jauhnya ***pixel***

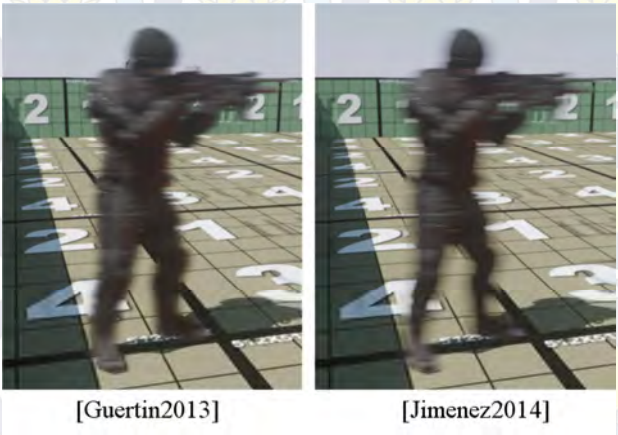
yang bisa dijangkau untuk sampel motion blur. Biarpun begitu, semakin besar nilai ***k*** juga membuat artifak kotak-kotak terlihat lebih jelas.

6. Semakin besar nilai S berefek pada penurunan performa, tetapi nilai ***k*** tidak berpengaruh banyak pada performa.

6.2. Saran

Berikut saran-saran untuk pengembangan dan perbaikan sistem di masa yang akan datang. Diantaranya adalah sebagai berikut:

1. Sampai buku tugas akhir ini ditulis ada beberapa pengembangan algoritma ***motion blur*** yang lebih baru yang muncul, salah satunya "***A Fast and Stable Feature-Aware Motion Blur Filter***" [25] dan implementasi motion blur pada presentasi "***Next Generation Post Processing in Call of Duty: Advanced Warfare*** " [26]. Berbagai teknik ini mencoba untuk mengatasi berbagai artifak yang muncul pada ***paper*** yang dipakai pada tugas akhir ini. Teknik-teknik ini bisa dipelajari lebih lanjut untuk dicoba dintegrasikan pada ***game engine*** Urho3D.



Gambar 6.1 Perbandingan hasil dua metode motion blur yang lebih baru [26]

2. Urho3D masih belum menyediakan cara mudah untuk menambah ***uniform*** spesifik per-objek sehingga masih perlunya untuk menambah fungsi langsung pada ***source code***-nya. Jika ada penelitian khusus untuk mengatasi permasalahan ini, maka akan memudahkan ***developer*** kedepannya.