



CENTRO UNIVERSITARIO
DE TECNOLOGÍA Y ARTE DIGITAL



Máster Universitario en Computación Gráfica y Simulación

2018

Trabajo Final de Máster

Estudio sobre sistemas de anti-aliasing e
implementación de anti-aliasing temporal

Hugo Ferrando Seage

Tutor: Dr. Alberto Sánchez Campos

Resumen

Debido a la resolución limitada de las pantallas al rasterizar gráficos 3D se deforman ciertas líneas y curvas. Este fenómeno se llama aliasing. La demanda de gráficos cada vez más realistas ha propiciado la creación de diferentes técnicas y algoritmos para disimular estos defectos, sin tener que recurrir necesariamente a pantallas de mayor resolución.

El anti-aliasing se ha vuelto una técnica crucial para mejorar la calidad de imagen en el software con gráficos tridimensionales. Este campo de la computación gráfica lleva años desarrollándose, con nuevas técnicas publicadas continuamente.

Este proyecto tiene como objetivo el análisis de las distintas técnicas existentes en el estado del arte y la implementación de un algoritmo de anti-aliasing temporal, compatible con motores con deferred shading y comparable en calidad al multisampling, junto a otras propiedades.

Abstract

When rasterizing 3D graphics using computer screens, due to their limited resolution, some lines and curves will become deformed. This phenomena is called aliasing. The ever increasing demand for more realistic graphics has driven the creation of new techniques and algorithms to hide these artifacts, without necessarily using higher resolution screens.

Anti-aliasing has become a crucial technique to advance image quality in graphics software. This field has been explored for years, with new approaches being developed continually.

This project aims to analyze different anti-aliasing techniques developed and the implementation of a temporal anti-aliasing technique, compatible with deferred shading engines and with a similar quality to multisampling, amongst other benefits.

Índice general

1. Introducción	11
2. Planteamiento del problema	13
3. Objetivos	15
4. Estado del Arte	17
4.1. Tipos de Aliasing	17
4.1.1. Aliasing en bordes de geometría	18
4.1.2. Aliasing de texturas	18
4.1.3. Specular Aliasing	20
4.1.4. Motion Aliasing	20
4.2. Tipos de Anti-aliasing	26
4.2.1. Filtrado	26
4.2.2. Supersampling Anti-aliasing	27
4.2.2.1. Ordered Grid Supersampling	27
4.2.2.2. Rotated Grid Supersampling	27
4.2.2.3. Quincunx Supersampling	27
4.2.3. Multisampling	30
4.2.4. Post Processing Anti-aliasing	31
4.2.4.1. Fast Approximate Anti-aliasing	31
4.2.4.2. Morphological Anti-aliasing	35
4.2.4.3. Enhanced Subpixel Morphological Anti-aliasing	35
4.2.5. Temporal Anti-aliasing	36
4.2.6. Machine Learning Anti-aliasing	36
5. Desarrollo	39
5.1. History Buffer	39
5.2. Jitter	40
5.3. Motion Vectors	41
5.4. Combinación de texturas	43

5.5. Neighborhood Clamping	43
5.6. Sharpening kernel	45
6. Resultados	47
7. Conclusión	53

Glosario

DSLR Digital single-lens reflex camera. Cámara fotográfica digital con una única lente y sensor electrónico.

FPS Frames per second. Número de fotogramas por segundo. En software gráfico refleja el rendimiento del mismo.

OpenGL API estándar para el desarrollo de software con gráficos 2D o 3D con aceleración por hardware[24].

rasterizar Conversión de una imagen descrita con alguna primitiva (curvas, vectores, triangulos) a un conjunto de pixels.

SSAO Screen Space Ambient Occlusión. Técnica gráfica para calcular la exposición a la luz ambiente en cada punto de una escena.

TXAA Anti aliasing temporal desarrollado por Nvidia.

Capítulo 1

Introducción

La calidad de imagen exigida en las aplicaciones gráfica es cada vez mayor, tanto en ámbitos lúdicos como para la generación de efectos visuales en películas o en videojuegos, como en aplicaciones civiles, educativas, médicas[32], o psicológicas. El realismo exigido por los usuarios es grande, pues el entrenamiento permanente de consumo de gráficos por ordenador ha provocado que las expectativas de cualquier gráfico o imagen sean elevadas. Si bien la tecnología no ha parado de avanzar, la calidad exigida tanto en aplicaciones 2D como 3D provoca la eminente necesidad de optimizar procesos de simulación o renderizado.

Una de las técnicas a optimizar es el anti-aliasing, ya que aporta más credibilidad a cualquier escena 3D e incluso en imágenes 2D. Uno de los ámbitos donde es imprescindible el anti-aliasing es en el renderizado de textos de los sistemas operativos gráficos, como los de escritorio o móviles. Los programas de edición de imágenes también cuentan con numerosos algoritmos de anti-aliasing con el objetivo de mejorar las imágenes o fotos.

Debido a la resolución limitada de las pantallas, será necesario usar este tipo de técnicas para emular la calidad que ofrecen las pantallas con una mayor densidad de pixels. Por la potencia del hardware actual, las técnicas tradicionales como el supersampling para realizar dicho efecto es impráctico. Por esa razón ha sido necesario el desarrollo de algoritmos más sofisticados que alcanzan una calidad similar, pero con un impacto en el rendimiento más asequible.

En definitiva, el aliasing es un problema fundamental en la computación gráfica y a día de hoy no existen ningún algoritmo perfecto que lo solucione a tiempo real. Por ese motivo existen una multitud de alternativas, cada una con sus ventajas e inconvenientes, que se exploran en este trabajo final de máster.

Capítulo 2

Planteamiento del problema

El aliasing siempre ha sido un problema en el campo de la computación gráfica. Desde la década de los 70 se han usado técnicas como la interpolación bicubica para el renderizado de imágenes por ordenador con ray-tracing.

Durante la década de los 80 y buena parte de los 90 hubo un avance constante en términos de potencia, resolución y la capacidad de mostrar cada vez más colores (por ejemplo del paso de gráficos CGA a VGA). Aún así el rendimiento necesario para aplicar muchas de las técnicas de anti-aliasing solo se podían usar con workstations profesionales, como las IRIS de Silicon Graphics.

A mediados de los 90, con la llegada de la tarjetas aceleradores de gráficos 3D a ordenadores más modestos, las compañías como ATI, NVIDIA y 3DFX empezaron a integrar soluciones de anti-aliasing a sus productos y APIs.

A la vez que evolucionaba el hardware los gráficos cada vez requerían más potencia. Las técnicas como el supersampling reducen drásticamente el rendimiento, por lo que en muchos casos se empezaron a usar optimizaciones como el multisampling.

Con la llegada de los shaders programables y el deferred shading el multisampling empezó a no poder ser usado en muchos casos. Por ello se desarrollaron algoritmos como el MLAA, FXAA y SMAA que son compatibles tanto con el forward shading como el deferred shading y con un impacto muy bajo en el rendimiento.

Al contrario que el supersampling, el multisampling y el post-processing anti-aliasing no solucionan todos los tipos de aliasing que tiene una imagen. Para intentar aunar las ventajas del supersampling con el rendimiento y compatibilidad del post-processing anti-aliasing se han empezado a desarrollar técnicas de anti-aliasing temporal, como el TXAA.

Capítulo 3

Objetivos

Este proyecto tiene dos objetivos principales:

1. Analizar el estado del arte en el campo de la computación gráfica en relación al anti-aliasing. Explorar los problemas que presenta el aliasing en gráficos 2D y 3D, la evolución de las soluciones y los desarrollos en curso.
2. El segundo objetivo es el de integrar una solución de anti-aliasing temporal para mejorar la calidad de imagen de un motor gráfico. El resultado final deberá tener una calidad equivalente a la de otras soluciones más convencionales como el supersampling, pero con un impacto en el rendimiento mucho menor, como con un post processing aliasing tradicional (FXAA, SMAA).

Capítulo 4

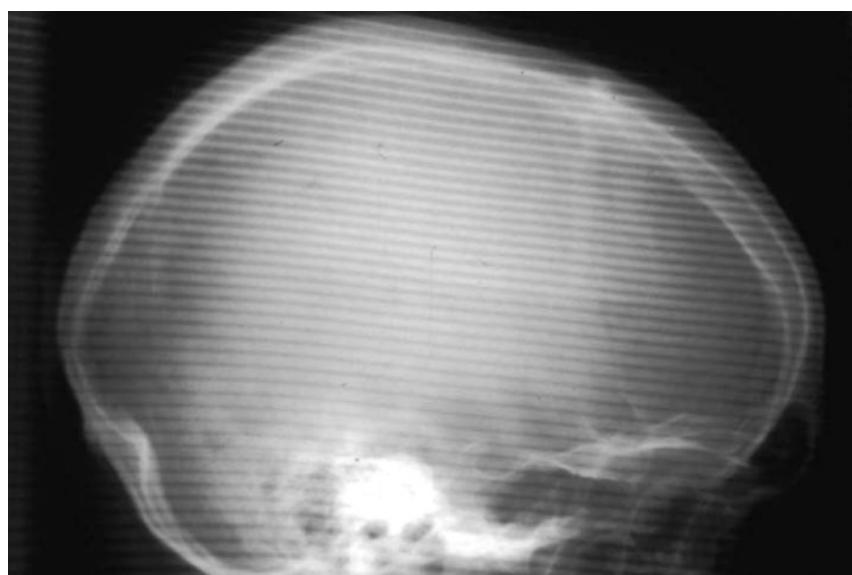
Estado del Arte

4.1. Tipos de Aliasing

El aliasing es la deformación de ciertos elementos gráficos al ser rasterizados y plasmados en una pantalla con una resolución finita[1]. Normalmente los bordes de la geometría que tengan ángulos que no se alineen perfectamente con los pixels presentarán bordes de sierra abruptos, que no son fieles a la escena como se vería de forma natural.

Este proyecto se centra en aliasing dentro del campo de la computación gráfica, pero afecta a otras disciplinas, como el procesamiento de señales (por ejemplo al digitalizar señales de audio analógicas). En general, al representar valores continuos en algún medio discreto siempre habrá algún defecto de este tipo[1]. Además el aliasing no solo es un problema en aplicaciones lúdicas. En la figura 4.1 se puede apreciar aliasing en una radiografía.

Figura 4.1: Aliasing en una radiografía[32]



Existen otros tipos de aliasing, como en el interior de texturas o al mover la cámara rápidamente.

4.1.1. Aliasing en bordes de geometría

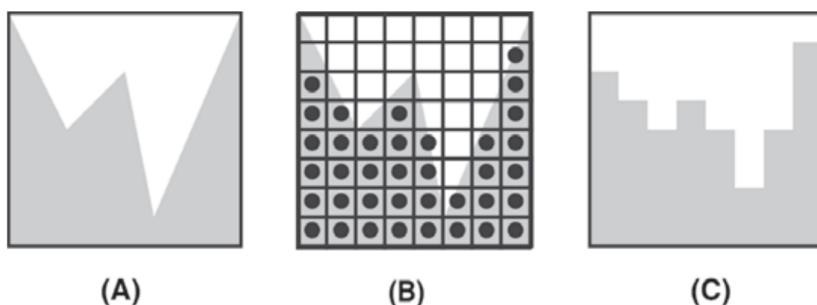
Seguramente el aliasing más común y conocido en cuanto a la computación gráfica. Es un efecto que se produce en los bordes de la geometría de la escena. Este efecto se manifiesta como un borde con dientes de sierra, como en la figura 4.3. Es apreciable sobre todo en líneas diagonales y en curvas, ya que en líneas planas (tanto horizontales como verticales) se alinean con el array de pixels de la pantalla.

Esto es debido al point sampling que se emplea durante el rasterizado de la escena (figura 4.2).

Este tipo de aliasing también está muy ligado a la densidad de la pantalla donde se esté viendo la escena. En un display con una densidad de pixels muy alta, siempre que el renderizado sea hecho con la misma resolución que la pantalla, será más difícil apreciar este tipo de aliasing. Aún así, con la tecnología actual, la densidad no siempre es lo suficientemente alta. Por ejemplo, una pantalla UHD (resolución 3840×2160) de 65" tiene una densidad de 67.78 DPI, donde el aliasing sigue siendo notable.

En pantallas de alta densidad (>200 DPI[31]), como la de varios smartphones, este tipo de aliasing no se manifestará.

Figura 4.2: Point Sampling[19]



4.1.2. Aliasing de texturas

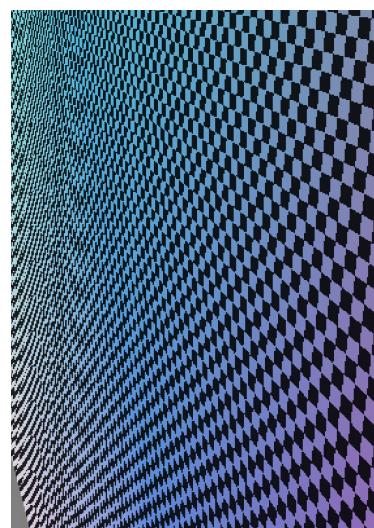
Dentro de las texturas también puede haber aliasing. Al igual que pasa en los bordes de la geometría, al cambiar de un color a otro rápidamente, debido al sampling limitado, se pueden producir estos mismos defectos.

Este tipo de aliasing no está limitado al rendering de gráficos 3D en tiempo real, si no que se puede apreciar incluso en imágenes con una alta resolución. En estos casos, dependiendo de la textura, se puede ver un patrón de Moiré, como en la figura 4.4.

Figura 4.3: Aliasing en Hitman (2016)



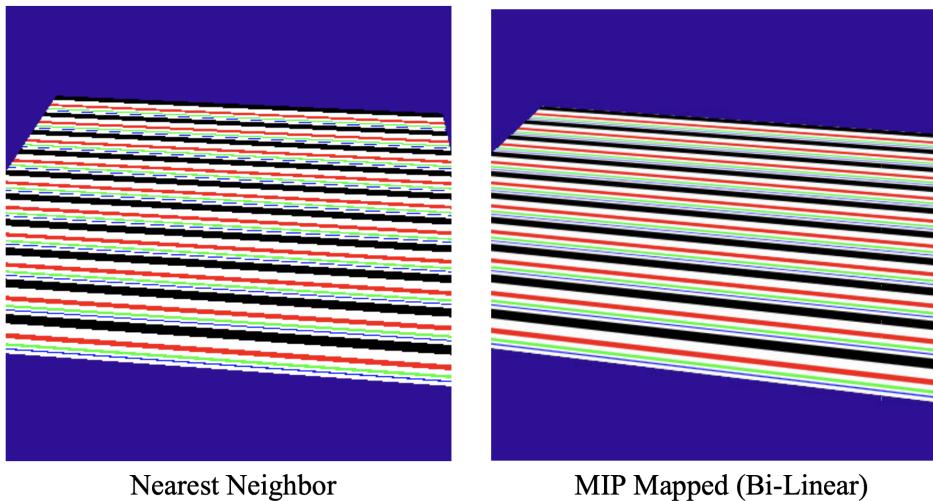
Figura 4.4: Patrón de Moiré[4]



La generación de MIP Maps puede solucionar este tipo de aliasing en ciertas condiciones. Los MIP Maps son versiones alternativas de las texturas usadas por un software gráfico. Estas versiones contienen texturas de menor resolución filtradas, lo que puede aliviar el aliasing y patrones de Moiré producidos por usar downsampling de texturas con una alta resolución con nearest neighbor, como se puede apreciar en la figura 4.5.

Estas texturas se suelen usar para objetos que estén a alguna distancia determinada de la cámara. Si se pre-filtrasesen las texturas originales serían demasiado borrosas y se empeoraría su calidad.

Figura 4.5: Anti-aliasing con Mip Maps



4.1.3. Specular Aliasing

El modelo clásico de iluminación usado en OpenGL consiste en componentes de luz ambiente, difusa y especular[19]. La iluminación especular se produce cuando la luz incide directamente sobre un material. Las propiedades del objeto dictarán como de brillante aparece.

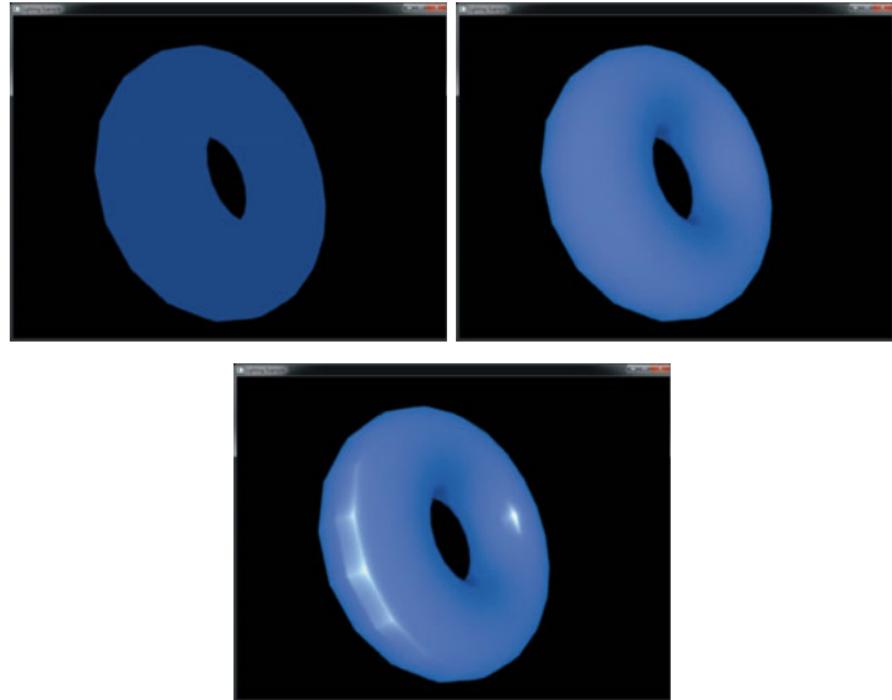
Por ejemplo, un espejo es completamente especular, ya que refleja toda la luz que recibe. Otros materiales muy especulares son algunos metales y plásticos. En cambio, la pizarra es un objeto con un valor especular muy bajo y que no producirá apenas reflejos.

El cálculo de esos reflejos pueden contener aliasing, al igual que las texturas. En la figura 4.7 se puede apreciar como los reflejos de un suelo metálico debido al mismo problema que en el aliasing dentro de texturas.

4.1.4. Motion Aliasing

A veces al mover la cámara también se pueden producir efectos de aliasing. El efecto no se puede apreciar en una imagen fija, pero al verlo el movimiento es un efecto que

Figura 4.6: Componentes de iluminación en OpenGL. Ambiente (superior izquierda), difuso (superior derecha) y especular (inferior)[19]



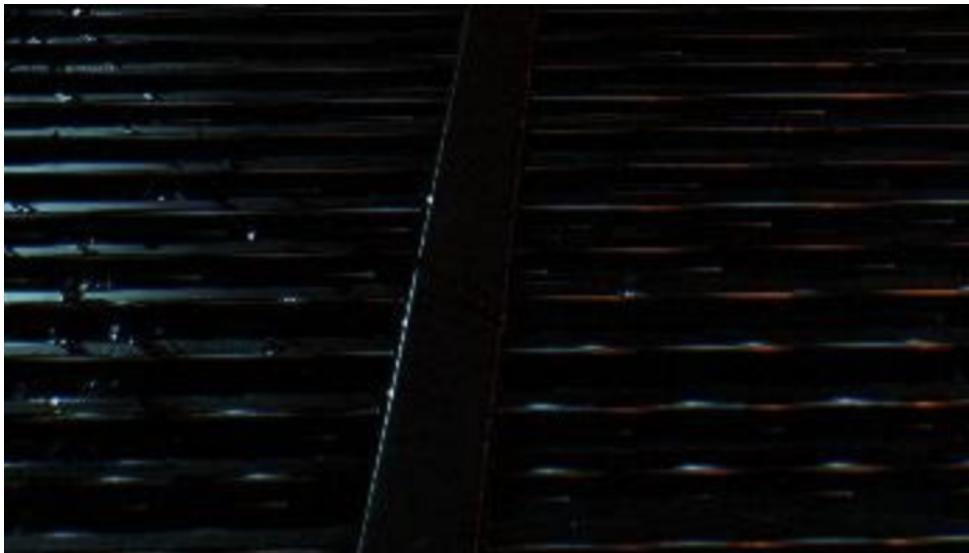
distrae. Recuerda al parpadeo de una luz, que una vez que la cámara permanece quieta desaparece (también conocido como crawling).

Este tipo de aliasing se suele solucionar con técnicas de blurring, que recuerdan más al efecto del cine a 24FPS que a técnicas de anti-aliasing como las que se explican en los siguientes secciones.

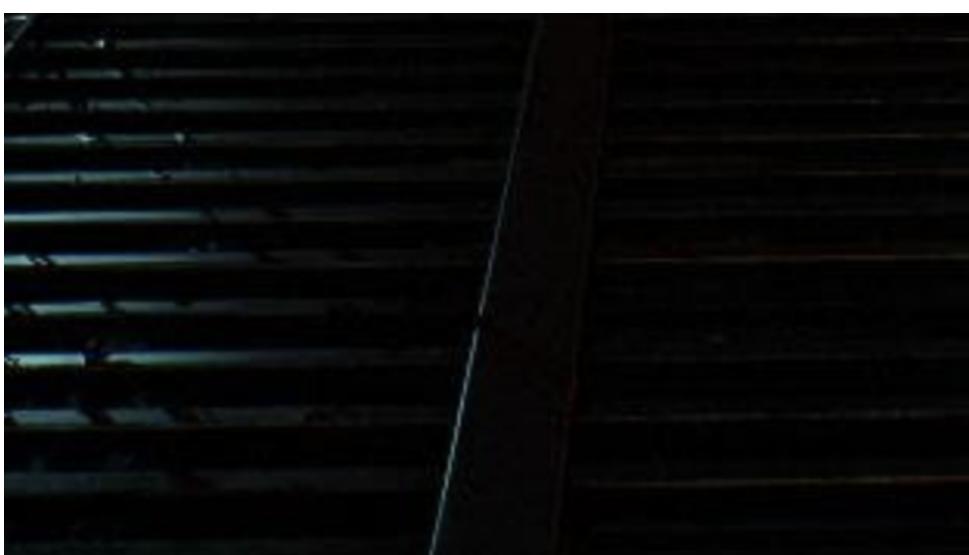
Estos efectos tienen como objetivo simular el efecto que tiene el tiempo de exposición en una cámara. El tiempo de exposición es el tiempo que el objetivo tiene para capturar luz para cada frame de un vídeo, o para una foto. Normalmente, en cine se usa 1/48 de segundo[18]. Cuanto más tiempo de exposición más blur habrá (aunque los fotogramas serán más claros). En la figura 4.8 se puede ver el efecto que tienen los diferentes tiempos de exposición en una DSLR.

Una de las primeras técnicas era la de usar un buffer de acumulación. Consiste en escoger una ventana de n frames anteriores al actual y combinarlo con una opacidad variable. Con cada frame esta ventana se actualiza. Esto intenta emular como funciona el tiempo de exposición de una cámara. Su principal ventaja es que funciona con objetos translúcidos y partículas sin tener que hacer nada especial, ya que afecta a toda la escena. Si el framerate es muy alto (alrededor de 120FPS) y fluido el resultado puede ser muy bueno, pero en software donde el rendimiento es menor (menos de 60FPS) la imagen se verá demasiado borrosa y el efecto no será el deseado. De hecho como se puede ver en

Figura 4.7: Specular aliasing en Doom (2016)

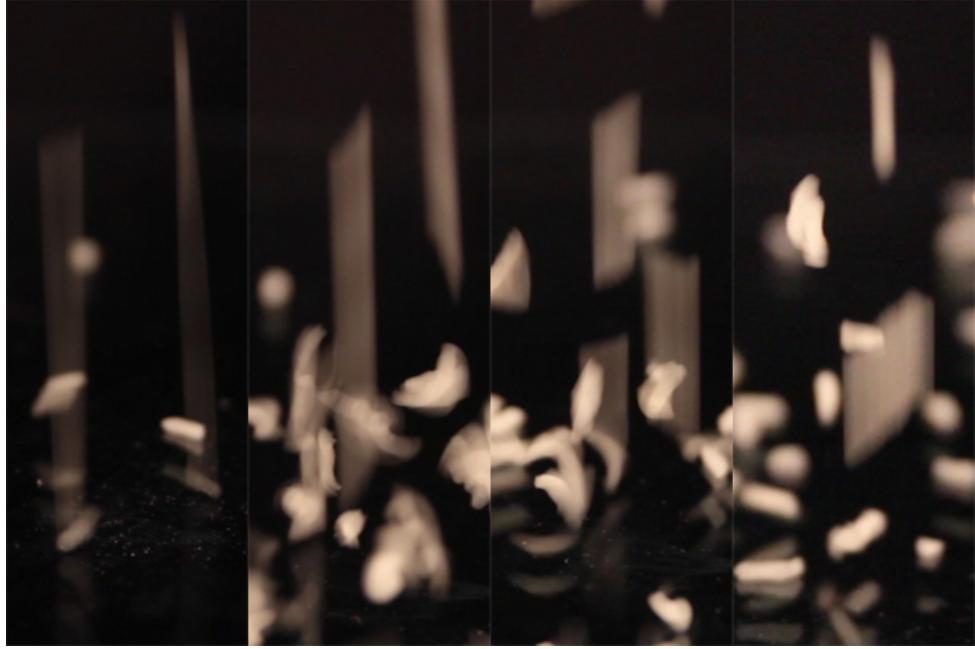


(a) Anti-aliasing desactivado



(b) TSSAA

Figura 4.8: Comparación de tiempo de exposición (360° , 180° , 90° , 45°)[18]



la figura 4.9 puede llegar a ser más desagradable que el propio aliasing.

Figura 4.9: Grand Theft Auto 3: Vice City en PS2



Con la llegada de los shaders se empezó a usar per pixel motion blur. Esta técnica consiste en guardar un buffer de velocidad y un history buffer donde se almacena el frame anterior y combinar texturas. Esto permite más flexibilidad que la técnica del accumulation buffer, ya que dependiendo del valor del velocity buffer se puede aplicar un efecto mayor o menor de blurring. Normalmente, cuanto más se ha movido un pixel de sitio, mayor será el efecto de blurring. Esto se ejecuta como un efecto de post procesado, es muy rápido y los resultados pueden ser muy convincentes incluso con framerates bajos. Una de las desventajas es que es complicado calcular el blurring en escenas con transparencias o en

partículas.

El per object motion blur es un refinamiento de la anterior, donde en vez de calcular donde estaba cada pixel en el frame anterior, el cálculo se hace por cada polígono de la escena por separado. Eso significa que es posible añadir más blurring a algún objeto y dejar el resto de la escena con mayor detalle, como en la figura 4.10

Como se puede ver en la figura 4.11, estos tipo de blurring puede crear un efecto de silueta, pero depende de la implementación. La cantidad de samples usados para la generación del blurring también afecta a la calidad del mismo (y rendimiento), como se puede apreciar en la figura 4.12.

Figura 4.10: Per object motion blur en Doom (2016)



Figura 4.11: Siluetas sobre objetos con blur en Crysis (2007)[10]



Figura 4.12: Diferentes niveles de calidad de motion blur en Crysis (2007)[10]



4.2. Tipos de Anti-aliasing

El anti-aliasing es el conjunto de técnicas cuyo objetivo es el de disimular o eliminar las imperfecciones vistas anteriormente.

4.2.1. Filtrado

En gráficos raster (bidimensionales) el anti-aliasing se realiza mediante filtros. Para entender el filtrado en imágenes es necesario pensar en pixels no como pequeños cuadrados en una pantalla, si no en muestras de una función (normalmente 3 muestras, una por cada canal RGB)[2]. Los filtros (o kernels) transforman las funciones para intentar suavizar los cambios bruscos de color.

En la figura 4.13 se muestra una comparación de las diferentes formas en las que cada filtro interpola valores de una imagen con un solo canal. Cada punto representa la muestra de la función que se visualizará en la pantalla.

Figura 4.13: Comparación de interpolación usando diferentes filtros[16]

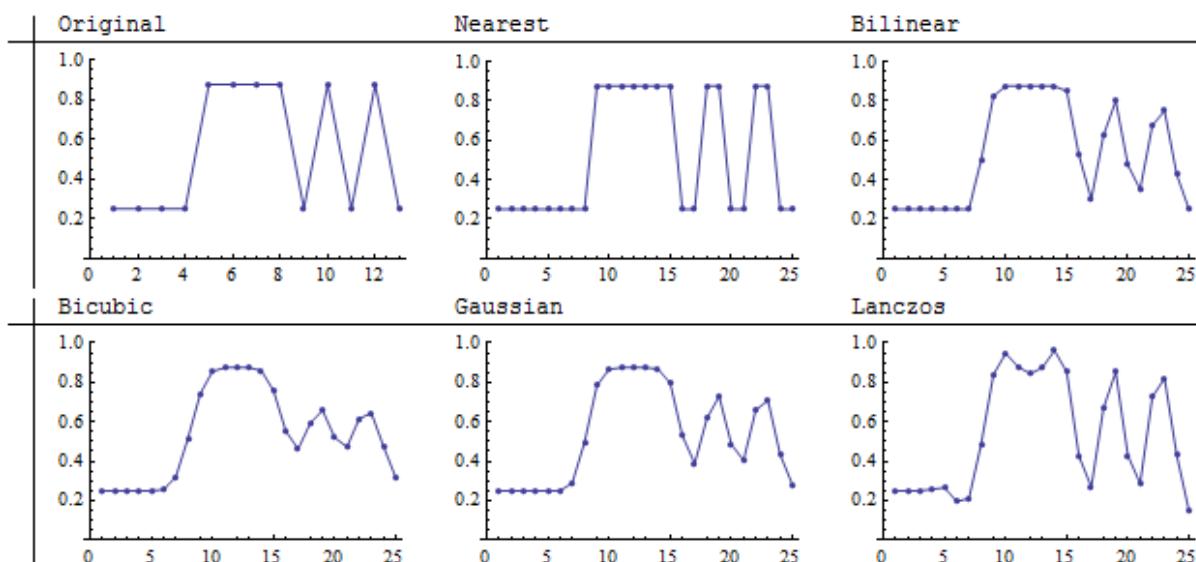
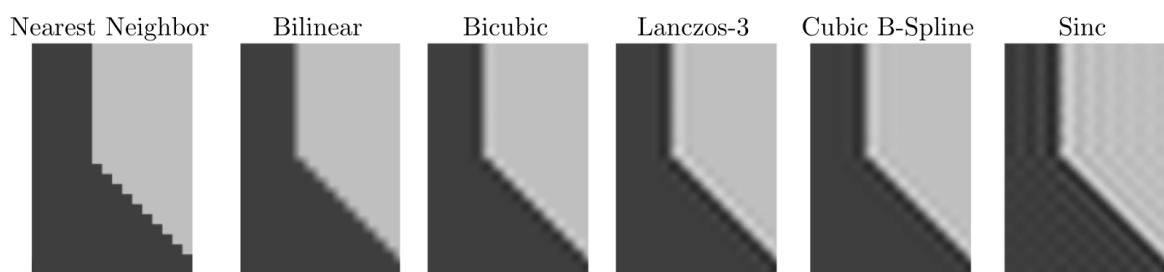


Figura 4.14: Comparación de resultados de diferentes filtros[14]



4.2.2. Supersampling Anti-aliasing

El supersampling consiste en minimizar el aliasing usando imágenes con una resolución mayor a la usada para su visualización (anti-aliasing espacial). Cada pixel en la pantalla será representado por más de un pixel en la imagen. Para el color final normalmente se hace la media del valor de los colores de los múltiples pixels[3].

Estas técnicas consumen más ancho de banda y memoria, ya que los buffers tienen que guardar más información por pixel que la imagen original.

La calidad del supersampling viene dado por la cantidad de samples por pixel y la técnica para saber que pixels usar en la imagen con mayor resolución para cada pixel de la pantalla.

En comparación a otras técnicas de anti-aliasing esta corrige aliasing de texturas (ver figuras 4.15 y 4.16) y especular (ver figura 4.7), además de aliasing de geometría. Esto se debe a que los subsamples los coge de en toda la escena.

4.2.2.1. Ordered Grid Supersampling

En esta técnica de supersampling la distribución de los samples es uniforme. Debido a la naturaleza regular de este patrón, los subpixels se encuentran situados en forma de dos columnas y dos filas. Para líneas muy planas, tanto en horizontal como vertical el anti-aliasing no será muy efectivo, ya que muy rápidamente tocará dos subpixels en fila o columna y hasta el final no tocará los otros subpixels, como se puede ver en la figura 4.17.

Este efecto se puede reducir usando más samples, pero el impacto en el rendimiento será todavía mayor.

Desde 2014 Nvidia ha publicado una técnica llamada Dynamic Super Resolution[20], que consiste en renderizar software a mayor resolución y usar OGSS con un filtro de blur gaussiano. En 2015 AMD implementó una técnica similar llamada Virtual Super Resolution.

4.2.2.2. Rotated Grid Supersampling

En el caso del RGSS se cambia el lugar de los subsamples para minimizar la cantidad de columnas y filas. La distribución sigue siendo uniforme, pero el problema del aliasing en líneas muy planas se soluciona, como se puede apreciar en la figura 4.18, sin tener que recurrir a aumentar de manera significativa el número de subsamples.

4.2.2.3. Quincunx Supersampling

Este patrón es interesante como técnica de supersampling ya que usa subsamples localizados en las esquinas de los pixels, lo que permite reutilizar varios subsamples entre pixels (figura 4.19). Esto permite una calidad similar a la de un anti-aliasing 4x pero con

Figura 4.15: Resultados de SSAA en Borderlands 2



(a) Anti-aliasing desactivado



(b) SSAA 4x

Figura 4.16: Resultados de SSAA en Borderlands 2

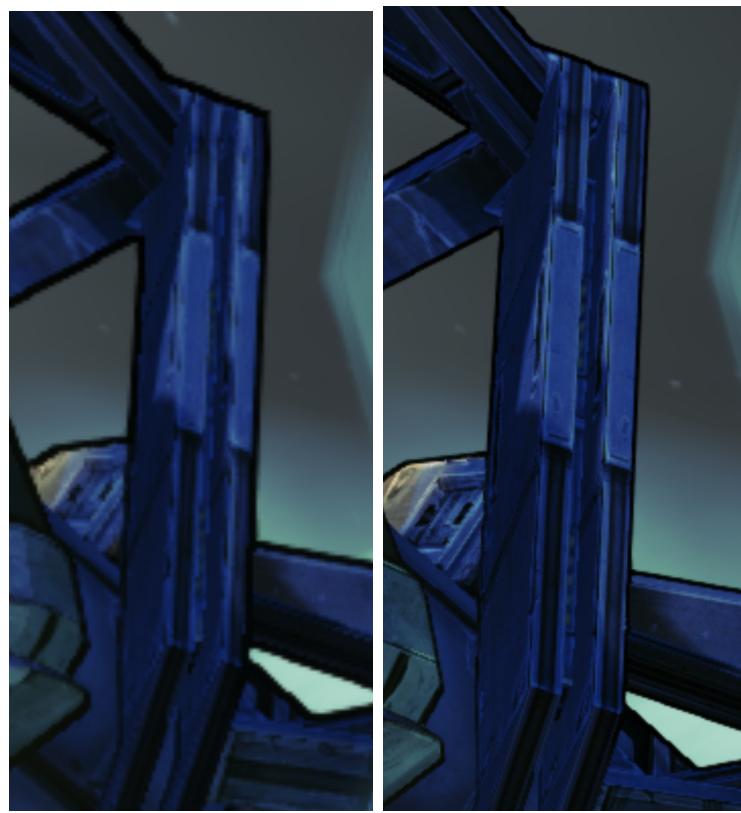


Figura 4.17: Ordered Grid Super-Sampling[3]

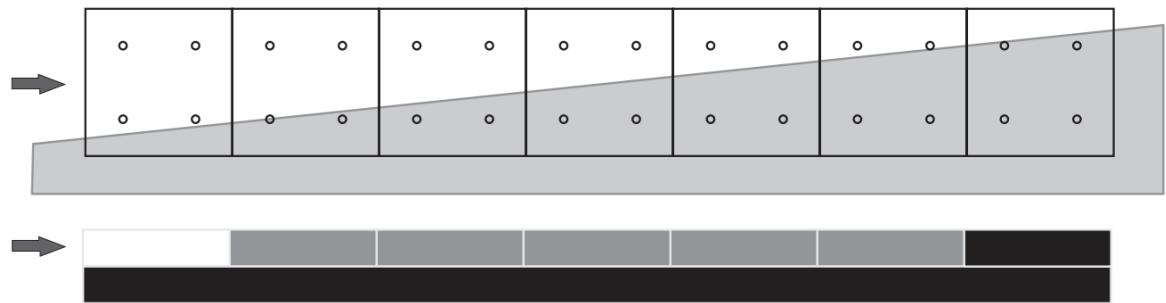
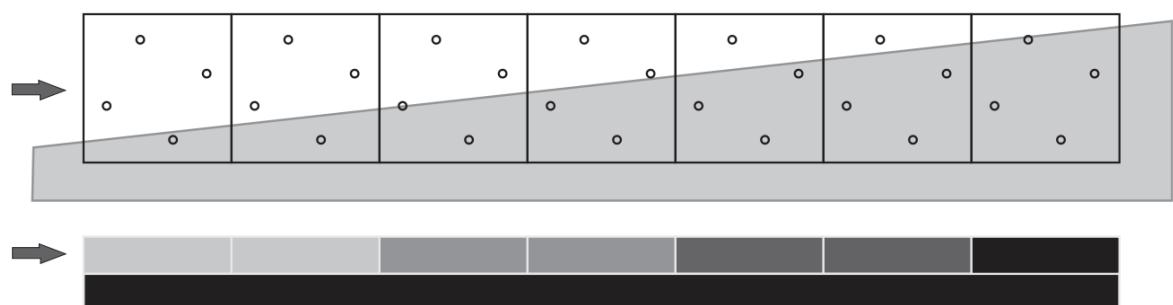


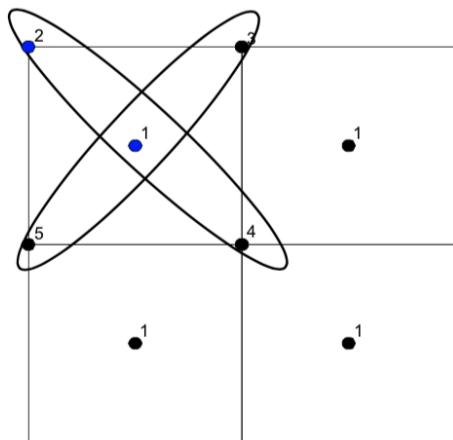
Figura 4.18: Rotated Grid Super-Sampling[3]



un coste computacional de un supersampling 2x. Aún así al reutilizar subsamples produce resultados algo más borrosos que otros métodos.

Aunque este patrón se pueda usar para supersampling Nvidia lo implementó en hardware desde la GeForce 3[5] para acelerar el calculo de su multisampling. Sony al utilizar una GPU de Nvidia en la PS3 también disponía de esta tecnología en la consola.

Figura 4.19: Quincunx Pattern[5]



Quincunx Sample Pattern

4.2.3. Multisampling

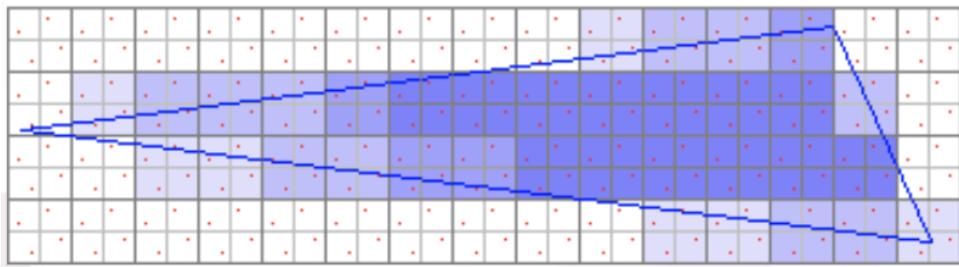
El multisampling es una optimización del supersampling. Se siguen usando subpixels, pero en vez de resolver el color para cada subpixel el color se resuelve una única vez por pixel. Después de resolver el color se calcula el stencil buffer y depth buffer con subpixels, y dependiendo de la cantidad de subpixels que estén contenidos dentro del triangulo se usará un porcentaje del valor del color calculado. Esto reduce drásticamente el impacto en el rendimiento de la aplicación, ya que solo hay que acceder una vez por pixel a la textura, pero solo soluciona aliasing en los bordes de la geometría[7][8].

En el centro de un polígono todos los subpixels van a estar contenidos dentro, pero el color solo se calcula una vez (por ejemplo en el centro del pixel), por lo que se usará el 100 % del color, pero puede contener aliasing. En cambio, en los bordes, algún subpixel estará fuera de la geometría, por lo que se usará un porcentaje del color calculado, suavizando el borde.

Otra de las desventajas del multisampling anti-aliasing es que, tradicionalmente, no se puede usar en motores con deferred shading. Debido a que el calculo del color final de cada

pixel se realiza en un pixel shader posterior al cálculo del g-buffer (figura 4.22), al hacer el resolve del multisampling no está lista toda la información necesaria. Desde OpenGL 3.2 y Direct3D 10.1 esto ha cambiado gracias al explicit multisampling[27]. La creación del g-buffer ahora se puede realizar usando texturas multisampled y al calcular la luz se puede acceder a los subsamples mediante la función texelfetch. Si se realiza de esta forma hay que sacrificar la creación automática de mipmaps y filtrado de texturas, aunque siempre se pueden implementar manualmente en shaders. Aún así muchos desarrolladores han optado por usar post-processing anti-aliasing al usar deferred shading.

Figura 4.20: Multisampling



4.2.4. Post Processing Anti-aliasing

Este tipo de anti-aliasing ha ganado popularidad durante los últimos años, debido en parte al uso del deferred shading en los últimos motores gráficos. Los algoritmos presentados a continuación se pueden considerar filtros de rasters, como los de la sección 4.2.1, pero están pensados para ser usados en motores gráficos. Además, en muchos casos, usan información que no está presente en la textura final, como pueden ser texturas de profundidad o vectores de movimiento,

4.2.4.1. Fast Approximate Anti-aliasing

El FXAA[12] es un sistema de anti-aliasing de post procesado desarrollado por NVIDIA. Se implementa como un único pixel shader y tiene diferentes niveles de calidad. El impacto en su rendimiento es muy bajo en comparación a métodos con resultados similares (inferior a 1ms).

Recibe la imagen final en formato RGB y la transforma en valores de luminancia por pixel. Usando el contraste entre pixels adyacentes detecta los bordes (figura 4.23). Cada borde es clasificado como horizontal o vertical. Dependiendo de los valores de luminancia y orientación de cada borde se aplica un pequeño offset para cada pixel y se hace un resampling para calcular un nuevo color que reduzca el aliasing[12][15].

Figura 4.21: Resultados del MSAA



(a) Anti-aliasing desactivado



(b) MSAA 4x

Figura 4.22: Generación de G-Buffer en Metal Gear Solid V: The Phantom Pain[29]

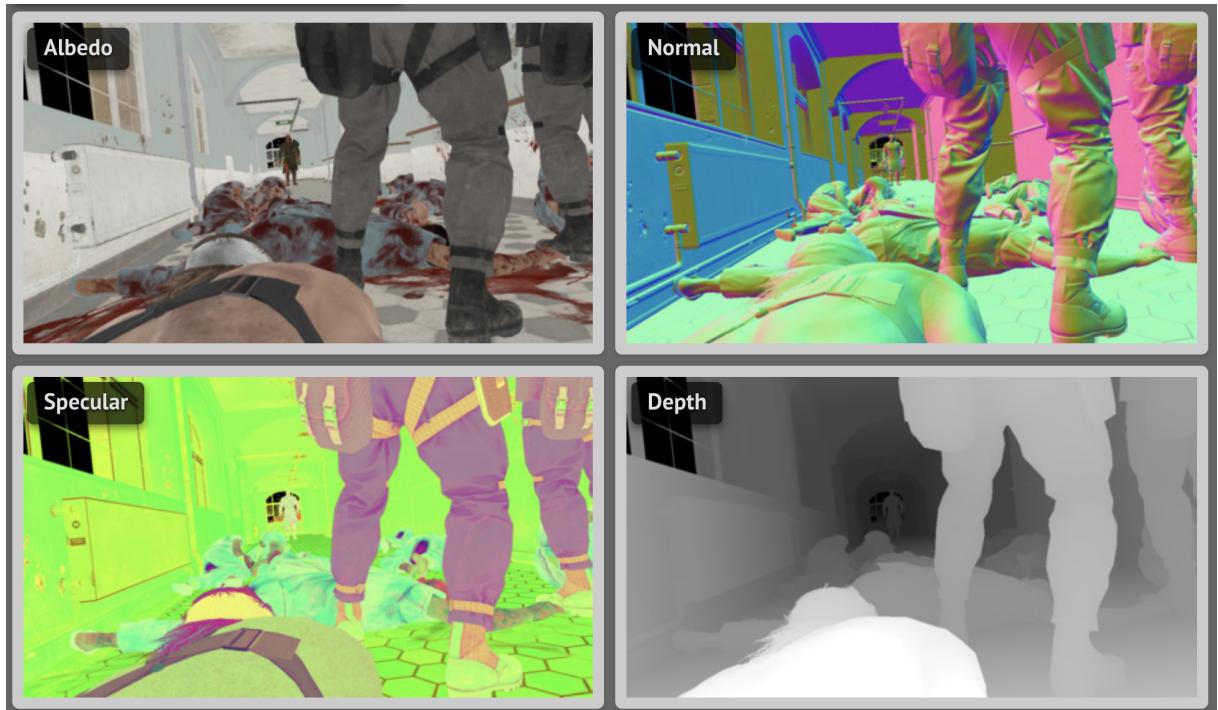


Figura 4.23: Detección de bordes usando valores de luminancia[15]

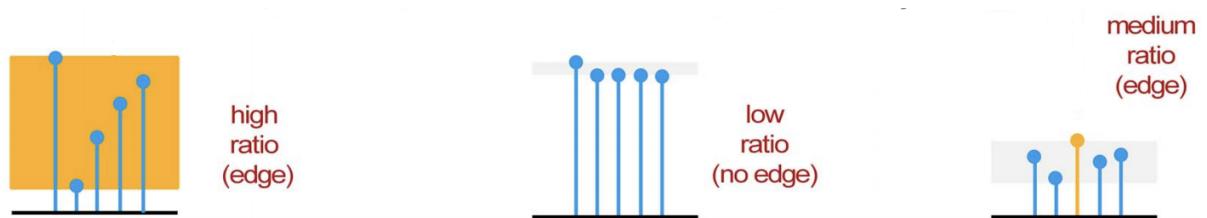
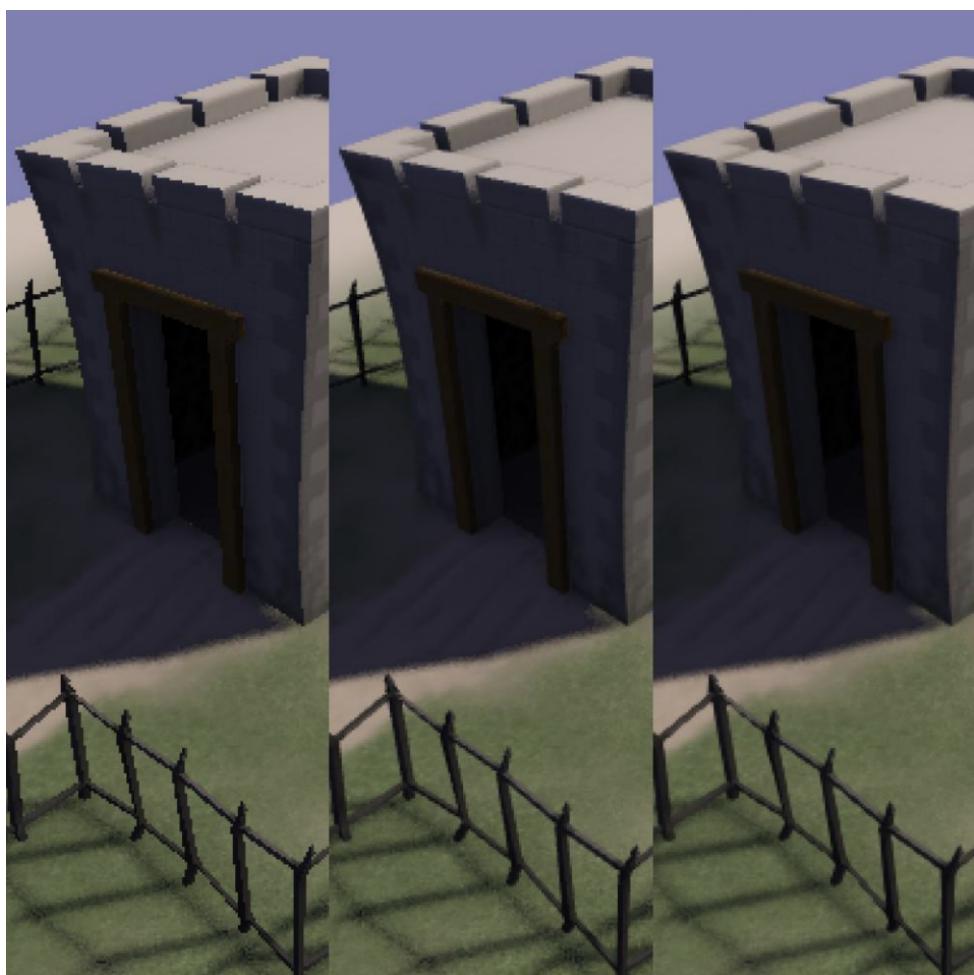


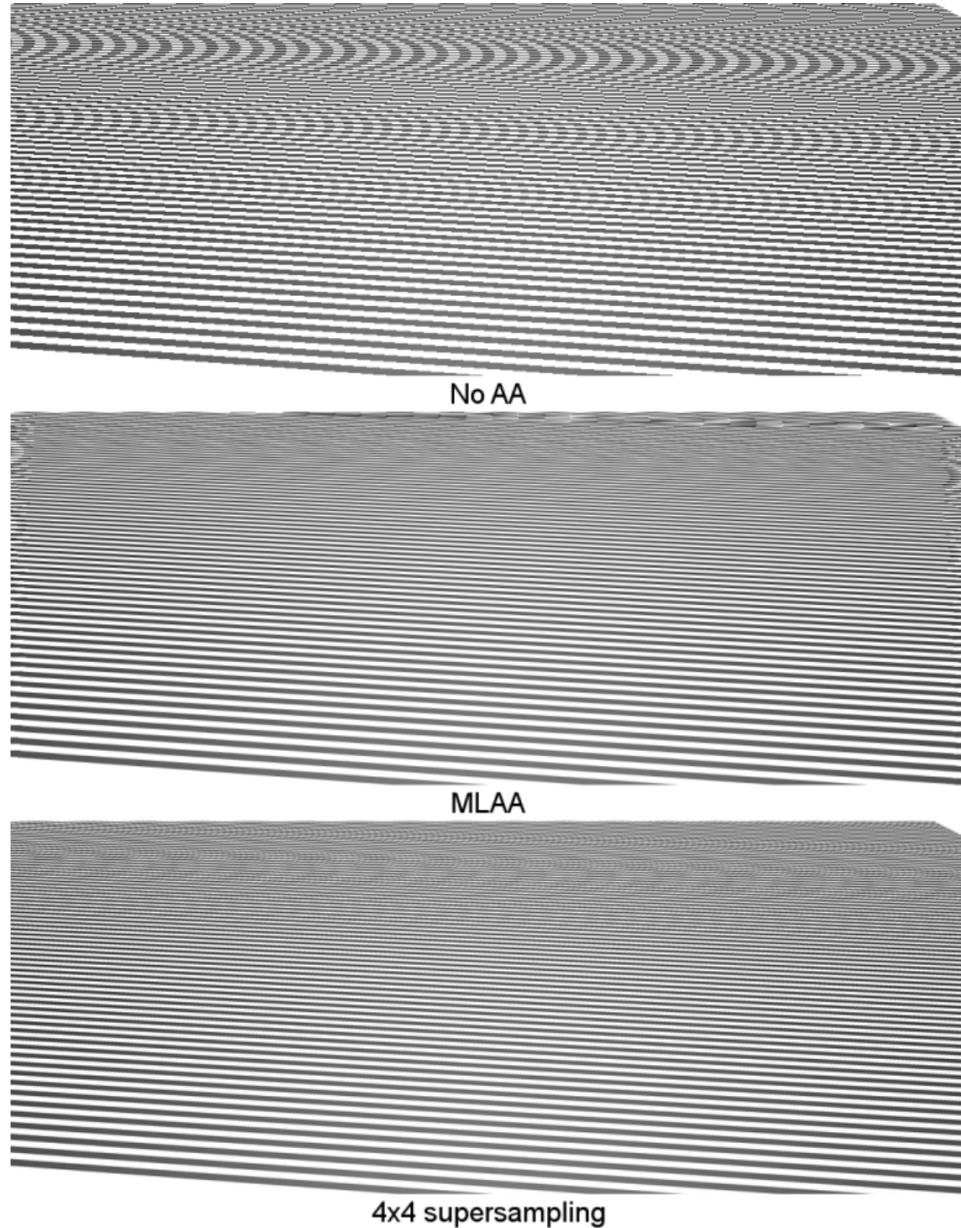
Figura 4.24: Escena con AA off, MSAA 4x y FXAA



4.2.4.2. Morphological Anti-aliasing

El MLAA[13] se basa en buscar los bordes discontinuos basándose en los colores de los pixels. Consigue discernir entre bordes con forma de U , Z y L . Finalmente mezclará los bordes con los colores de los pixels adyacentes.

Figura 4.25: MLAA en blanco y negro



4.2.4.3. Enhanced Subpixel Morphological Anti-aliasing

El SMAA[17] se basa en MLAA, pero aplicando unas variaciones con el fin de mejorar el resultado final. La calidad de imagen es más elevado, pero tiene un coste computacional algo mayor.

Figura 4.26: MLAA con color



Las variaciones con respecto a MLAA son:

- Consigue clasificar más formas de bordes, incluidas las líneas diagonales
- Usa contraste local en vez de color como métrica de comparación de valores entre pixels
- Puede funcionar con más de un sample por pixel, junto a un supersampling tradicional

En la figura 4.27 se puede ver la calidad del resultado final. Como con el resto de técnicas de anti-aliasing de post-procesado, aunque la aplicación no tenga soporte nativo para SMAA (como es el caso de Halo 1) se puede inyectar mediante software, gracias a que se ejecuta como una pasada final a la imagen.

4.2.5. Temporal Anti-aliasing

El temporal anti-aliasing es un tipo de anti-aliasing de post procesado, pero que amortiza el coste computacional usando múltiples samples en múltiples frames. Esto presenta unas complicaciones a la hora de escoger samples en frames anteriores en imágenes no estáticas.

Su implementación y resultados se explican en más detalles en los siguientes capítulos.

4.2.6. Machine Learning Anti-aliasing

Están empezando a aparecer técnicas que usan algoritmos de machine learning (como redes neuronales profundas) para generar una imagen con mayor calidad a la original. Este tipo de problemas son ideales para algoritmos de machine learning, ya que es fácil conseguir datos con los que entrenar las redes y no existe ninguna solución algorítmica ideal que se pueda usar en tiempo real[22][11] (ver figura 4.28).

Figura 4.27: Comparación de antialiasing en Halo 1 (2001)



(a) Anti-aliasing desactivado



(b) SMAA

Nvidia, desde la arquitectura Turing, ha comenzado a explotar estas técnicas para mejorar la calidad del anti-aliasing[26]. Esta arquitectura cuenta con hardware específico que acelera tareas de deep learning. Nvidia entrenará redes neuronales para cada software en cuestión con imágenes reales usando supersampling x64. La red ajustará sus valores en cada iteración del entrenamiento hasta llegar a un nivel satisfactorio de calidad. Estas redes entrenadas se distribuirán a los usuarios de tarjetas Turing.

Figura 4.28: Super Res[26]

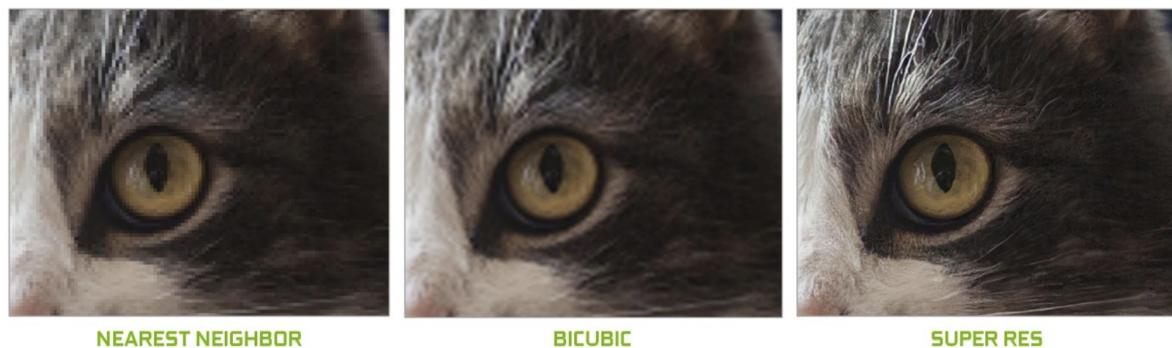
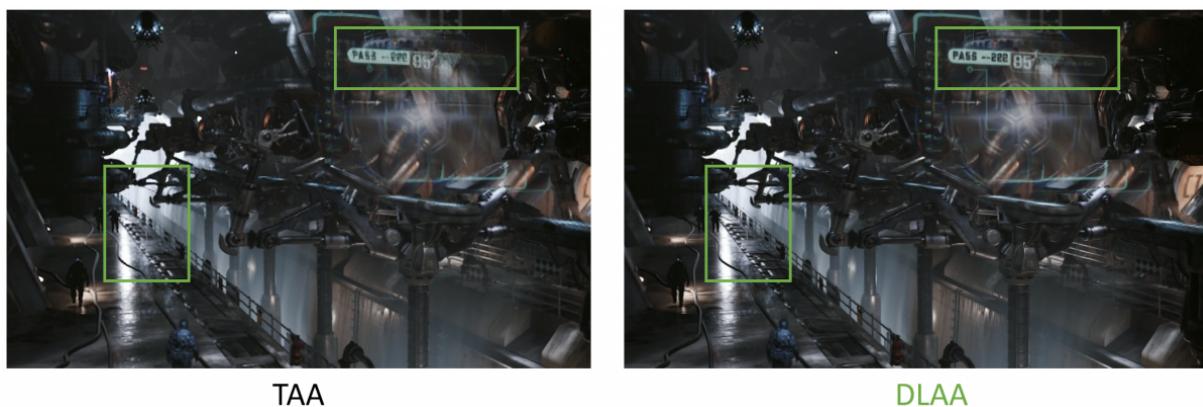


Figura 4.29: DLSS[26]



Capítulo 5

Desarrollo

Para este proyecto se ha integrado un sistema de anti-aliasing temporal en un motor de gráficos 3D que usa OpenGL como API gráfica. Su implementación se ha realizado mediante shaders GLSL y código C++ 14. El motor gráfico sobre el que se ha implementado es propio y se ha desarrollado durante el Máster Universitario en Computación Gráfica y Simulación de U-Tad en el curso 2017–2018.

De manera general, el motor ha tenido que ser modificado para incluir funcionalidad de post-procesado de imágenes mediante Frame Buffer Objects, métodos para la realización del jitter, y shaders que calculan el anti-aliasing en sí. No se ha empleado ninguna función de anti-aliasing que proporciona OpenGL de manera automática, como el uso de texturas `GL_TEXTURE_2D_MULTISAMPLE`.

En primer lugar es necesario poder guardar el resultado del frame anterior, desplazar la cámara ligeramente cada frame (creando subsamples únicos en cada frame), calcular el desplazamiento de la cámara, mezclar colores del frame anterior al actual, y finalmente aplicar un filtro de sharpening.

5.1. History Buffer

Debido a la necesidad de combinar varias texturas de frames anteriores, para calcular el anti-aliasing de un frame n es necesario tener acceso al frame $n-1$. Aún usando más de 2 subsamples solo es necesario guardar el último, siempre y cuando tenga el anti-aliasing ya incluido.

También se puede optar por guardar n frames para n subsamples y combinarlos en un pixel shader, pero esto ocupa mucha más memoria y es impráctico.

Para esto es necesario crear un FBO y guardar el resultado de la textura como variable, que durante el siguiente frame será pasada al shader como un uniform sampler2D, y así poder acceder a sus pixels.

5.2. Jitter

En primer lugar es necesario elegir los subsamples a usar en cada frame. Recordemos que en vez de calcular varios subsamples en cada frame, el temporal anti-aliasing calcula un solo sample por frame. Cada frame cambiará el lugar del subsample a calcular. Gracias a que solo se calcula un subsample por frame el impacto en el rendimiento será mucho menor al del supersampling tradicional.

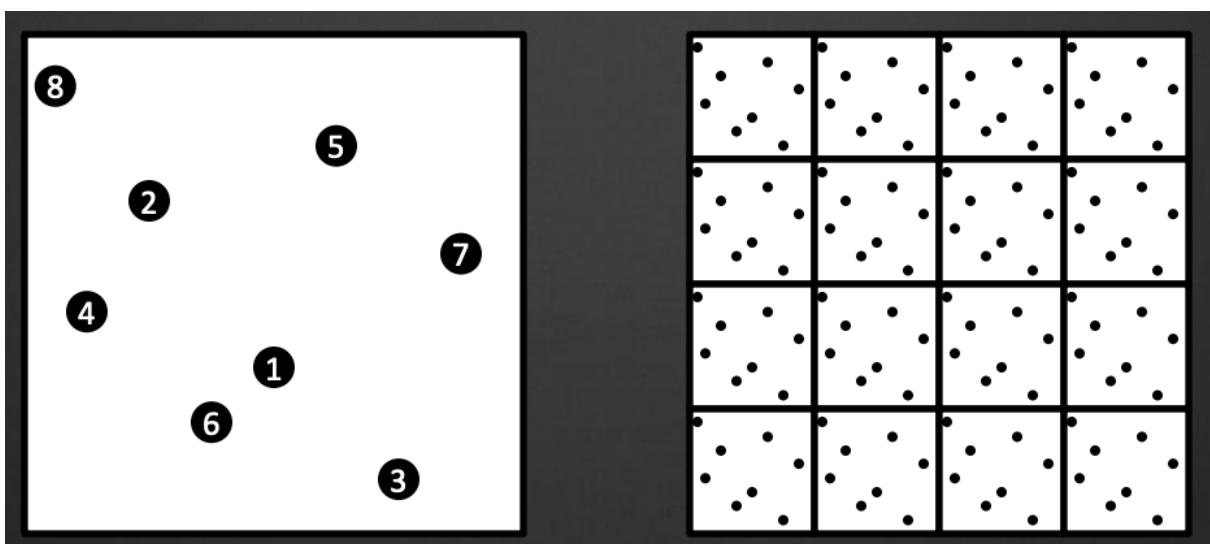
Para elegir los subsamples se pueden usar los grids vistos anteriormente en la sección sobre supersampling[28]. Aún así en la práctica se suelen usar secuencias de Halton (figura 5.1).

Las secuencias de Halton proporcionan una baja discrepancia, lo que significa que no se van crear clusters ni en espacio ni en tiempo. Esto proporciona una calidad de subsamples muy alta por lo que la calidad del anti-aliasing puede ser muy elevado. Siempre que esté bien implementado los resultados serán mejores que los del anti-aliasing por hardware implementado hasta ahora[28].

Una vez se ha calculado la secuencia de Halton en cada frame se alterará la matriz *MVP* para mover el sample al lugar del subsample que toque en ese frame. Este lugar cambiará cada vez que se genera un nuevo frame.

En este punto si viésemos la imagen seguiríamos notando el aliasing y la imagen parecería que estuviera vibrando ligeramente.

Figura 5.1: Jitter con secuencia Halton



```
float Halton(int Index, int Base) {  
    float Result = 0.0f;
```

```

float InvBase = 1.0f / Base;
float Fraction = InvBase;
while (Index > 0)
{
    Result += (Index % Base) * Fraction;
    Index /= Base;
    Fraction *= InvBase;
}
return Result;
}

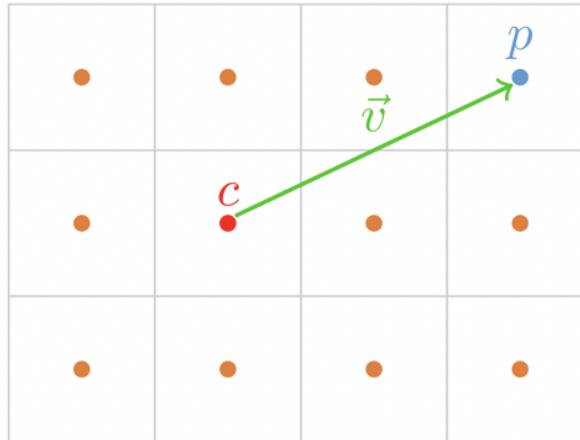
```

5.3. Motion Vectors

El segundo paso para la implementación del temporal anti-aliasing consiste en la creación de un velocity buffer. Este buffer contendrá una textura fp16 con dos canales de color (rojo y verde) donde cada pixel representa un vector.

Ese vector representa cuanto se ha movido un pixel del frame actual (n) al anterior ($n-1$). Cada pixel viene definido por sus posiciones en x e y . Para calcular el velocity buffer se usa su posición actual $C = x_n y_n$ y su posición previa $P = x_{n-1} y_{n-1}$. En el velocity buffer se almacena \vec{CP} , como se ilustra en la figura 5.2.

Figura 5.2: \vec{CP} [23]



Para la creación del velocity buffer en una escena dinámica es imprescindible saber no solo el movimiento de cámara del frame anterior, si no como se han movido los objetos. Esto se complica si hay objetos translúcidos como ventanas, o que reflejan luz como los espejos[25].

En este proyecto se ha optado por usar el algoritmo descrito en el capítulo 27 de GPU Gems 3[9] para su implementación. Solo es necesario usar un pixel shader junto al history

buffer, el depth buffer y las matrices VP_n y VP_{n-1} . La ventaja de este algoritmo es que no depende mucho de la implementación del motor. Nvidia incluye una librería que hace estos calculos en el SDK PostWorks[33] (usado para la implementación de TXAA).

Otra posibilidad es computar la velocidad durante una segunda pasada del vertex shader de cada polígono, pero dependiendo de su implementación puede resultar algo más lenta. Aún así debería resultar en vectores más precisos y en algunas ocasiones será necesario[21].

```
#version 330 core
uniform sampler2D depthTexture;

in vec2 TexCoords;
uniform mat4 VP;
uniform mat4 PrevVP;

out vec4 velocity;

void main() {
    float zOverW = texture2D(depthTexture, TexCoords).r
        * 2.0f - 1.0f;

    vec4 H = vec4(TexCoords.x * 2.0f - 1.0f,
                  (1.0f - TexCoords.y) * 2.0f - 1.0f,
                  zOverW, 1.0f);

    vec4 D = H * inverse(VP);

    // Original position
    vec4 worldPos = D / D.w;

    // Prev. position
    vec4 currentPos = H;
    vec4 previousPos = worldPos * PrevVP;
    previousPos /= previousPos.w;

    velocity = vec4((currentPos.xy - previousPos.xy) / 2.0f,
                   0.0f, 1.0f);
}
```

5.4. Combinación de texturas

Una vez se han calculado los motion vectors la combinación de texturas del history buffer y el frame actual es sencilla. Bastará con combinar ambas texturas pixel a pixel en el lugar correspondiente, usando interpolación lineal. Un valor de 5 % es lo que se usa en algunos videojuegos[25] comerciales.

Si se usa blurring es recomendable usar un factor de interpolación lineal dinámico, dependiendo del contraste local del pixel con los pixels que lo rodean.

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D currentTexture;
uniform sampler2D historyAATexture;
uniform sampler2D motionTexture;

void main() {
    vec2 historyVel = texture(motionTexture, TexCoords).xy;

    vec2 uvLast = TexCoords + historyVel;
    FragColor = mix(texture(currentTexture, TexCoords),
                    texture(historyAATexture, uvLast),
                    0.3f);
}
```

5.5. Neighborhood Clamping

Los motion vectors a veces no son suficientemente precisos como para poder usarlos por si solos. Si un objeto oculta a otro durante un cambio de frame el pixel resultante tendrá un color incorrecto, ya que se combinarán texturas de dos objetos diferentes. Esto puede resultar en un efecto llamado ghosting (figura 5.3).

Para solucionar esto se puede usar neighborhood clamping. Cada pixel se compara con el valor de los pixels adyacentes. Si el cambio de color es mayor o menor al de los pixels adyacentes en el history buffer se hace un clamp del valor del pixel resultante.

Usando técnicas de blurring también puede aliviar el efecto del ghosting ya que suaviza transiciones de un frame al siguiente. De hecho si se hace per pixel o per object motion blur

se pueden reutilizar los mismos motion vectors necesarios para el Temporal anti-aliasing. Aún así este efecto por si solo no suele ser suficiente.

Para conseguir esto se puede modificar el shader que combina texturas para primero realizar el clamping.

Figura 5.3: Ghosting[28]



```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D currentTexture;
uniform sampler2D historyAATexture;
uniform sampler2D motionTexture;

#define KERNEL_SIZE 9
const float step_w = 1.0 / 1920;
const float step_h = 1.0 / 1080;
const vec2 offset[KERNEL_SIZE] = vec2[KERNEL_SIZE](
    vec2(-step_w, -step_h), vec2(0.0, -step_h), vec2(step_w, -step_h),
    vec2(-step_w, 0.0), vec2(0.0, 0.0), vec2(step_w, 0.0),
    vec2(-step_w, step_h), vec2(0.0, step_h), vec2(step_w, step_h)
);

void main() {
```

```

vec2 historyVel = texture(motionTexture, TexCoords).xy;
vec2 uvLast = TexCoords + historyVel;

vec4 maxVal = vec4(0.0f, 0.0f, 0.0f, 0.0f);
vec4 minVal = vec4(1.0f, 1.0f, 1.0f, 0.0f);

for (int i = 0; i < KERNEL_SIZE; ++i) {
    vec4 tmp = texture2D(historyAATexture, uvLast.xy + offset[i]);
    maxVal = max(tmp, maxVal);
    minVal = min(tmp, minVal);
}

vec4 color = mix(texture(currentTexture, TexCoords),
                    clamp(texture(historyAATexture, uvLast), minVal, maxVal),
                    0.60f);

FragColor = color;
}

```

5.6. Sharpening kernel

Debido a la naturaleza del temporal anti-aliasing la imagen final puede resultar demasiado suave. Los bordes quizás resulten algo difusos. Este efecto puede no ser del agrado del usuario. Una posible solución es usar un kernel de sharpening que resalta los bordes de la geometría.

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

```

#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D finalTexture;

#define KERNEL_SIZE 9
const float kernel[KERNEL_SIZE] = float[KERNEL_SIZE](
    0.0, -1.0 , 0.0 ,
    -1.0 , 5.0 , -1.0 ,

```

```

    0.0 , -1.0 , 0.0
);

const float step_w = 1.0 / 1920;
const float step_h = 1.0 / 1080;
const vec2 offset[KERNEL_SIZE] = vec2[KERNEL_SIZE](
    vec2(-step_w, -step_h), vec2(0.0, -step_h), vec2(step_w, -step_h),
    vec2(-step_w, 0.0), vec2(0.0, 0.0), vec2(step_w, 0.0),
    vec2(-step_w, step_h), vec2(0.0, step_h), vec2(step_w, step_h)
);

void main(void)
{
    int i = 0;
    vec4 sum = vec4(0.0);

    vec4 tmp = texture2D(finalTexture, TexCoords.xy + offset[0]);
    sum += tmp * kernel[0];
    tmp = texture2D(finalTexture, TexCoords.st + offset[1]);
    sum += tmp * kernel[1];
    tmp = texture2D(finalTexture, TexCoords.st + offset[2]);
    sum += tmp * kernel[2];
    tmp = texture2D(finalTexture, TexCoords.st + offset[3]);
    sum += tmp * kernel[3];
    tmp = texture2D(finalTexture, TexCoords.st + offset[4]);
    sum += tmp * kernel[4];
    tmp = texture2D(finalTexture, TexCoords.st + offset[5]);
    sum += tmp * kernel[5];
    tmp = texture2D(finalTexture, TexCoords.st + offset[6]);
    sum += tmp * kernel[6];
    tmp = texture2D(finalTexture, TexCoords.st + offset[7]);
    sum += tmp * kernel[7];
    tmp = texture2D(finalTexture, TexCoords.st + offset[8]);
    sum += tmp * kernel[8];

    sum = clamp(sum, 0.0, 1.0);

    FragColor = sum;
}

```

Capítulo 6

Resultados

A continuación se encuentran algunos resultados de la implementación del anti-aliasing temporal.

La figura 6.1 contiene la imagen que creaba el motor sin modificaciones. Se puede ver una cantidad considerable de aliasing en el borde del cubo, al estar ligeramente inclinado. Activando el anti-aliasing temporal, como en las figuras 6.2 y 6.3, se aprecia una mejora notable en la calidad de los bordes. Activando más samples la calidad aumenta.

El anti-aliasing temporal también mejora el aliasing en reflejos. Comparando las figura 6.4 con la figura 6.5 se puede apreciar un borde más suave, también en el interior de la geometría. Esto no es posible solucionarlo con un multisampling anti-aliasing tradicional.

Por último, en la figura 6.7 se puede ver el efecto visual producido por el filtro de sharpening. Comparado con la figura 6.6 las bordes resaltan mucho más, y eliminan la suavidad de los bordes. Comparado con la imagen original (figura 6.8) puede resultar excesivo. Afortunadamente el filtro es opcional y el usuario final tiene la opción de activarlo o no, dependiendo de su preferencia.

Figura 6.1: Anti-aliasing desactivado



Figura 6.2: Temporal anti-aliasing con 8 samples



Figura 6.3: Temporal anti-aliasing con 16 samples



Figura 6.4: Reflejos especulares con anti-aliasing apagado

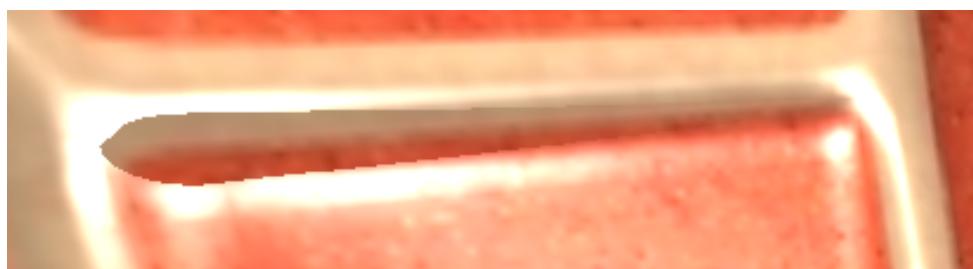


Figura 6.5: Reflejos especulares con temporal anti-aliasing



Figura 6.6: Filtro de sharpening desactivado

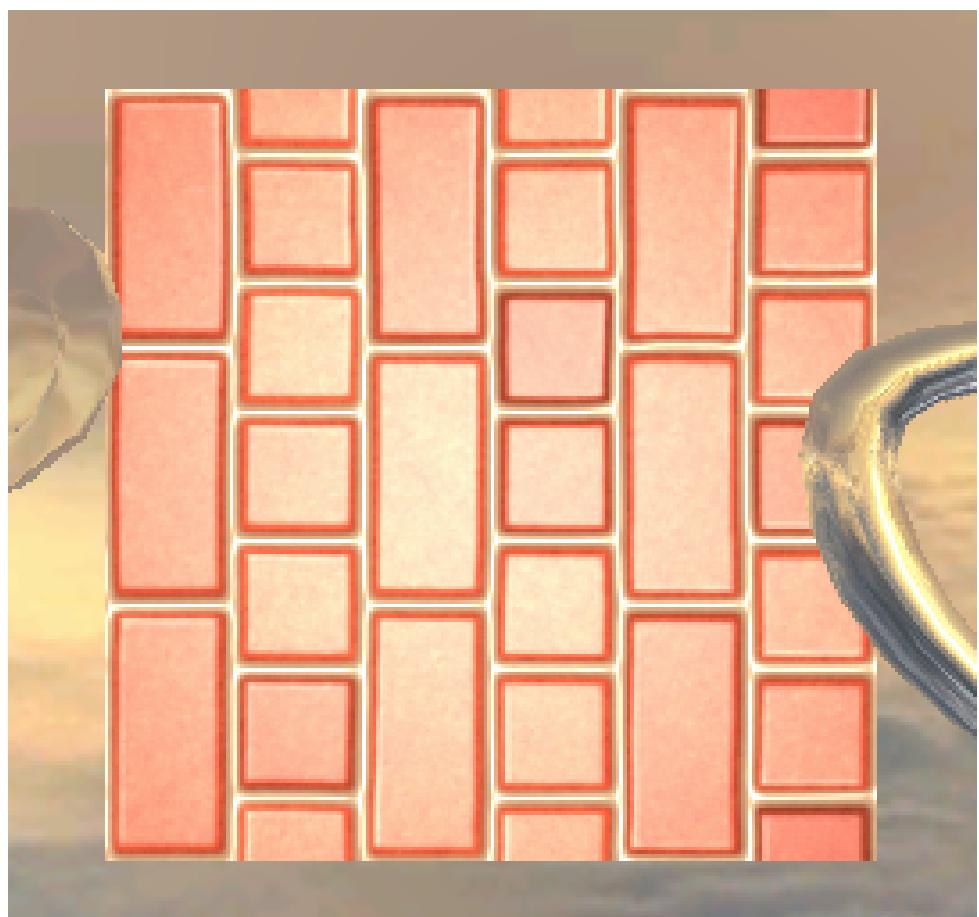


Figura 6.7: Filtro de sharpening activado

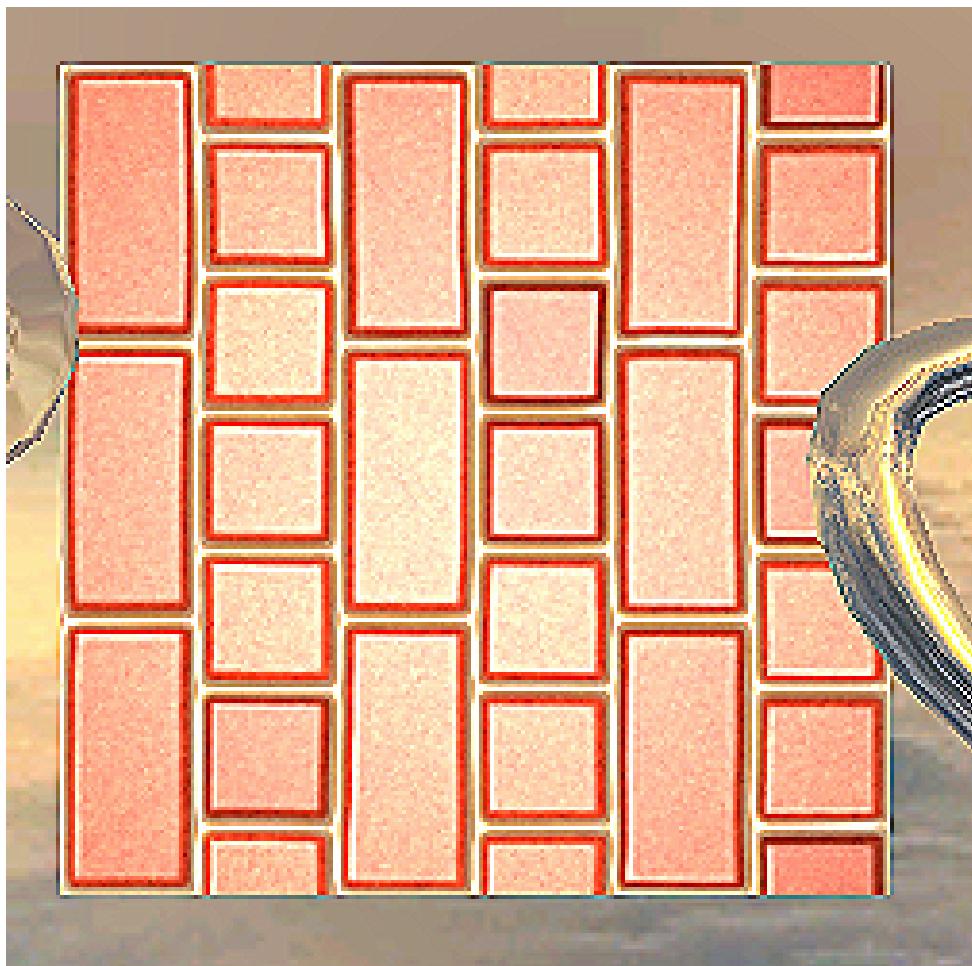
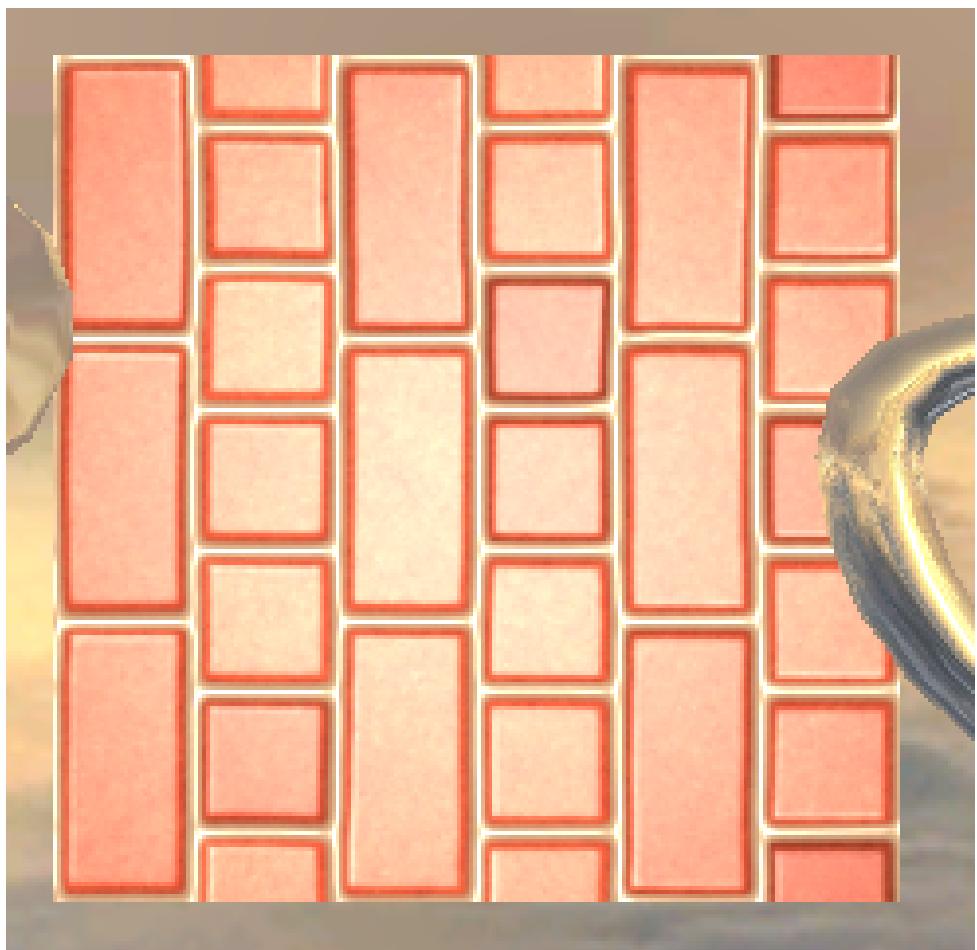


Figura 6.8: Temporal anti-aliasing desactivado



Capítulo 7

Conclusión

Para lograr unos resultados excelentes con respecto a la calidad de imagen es indispensable implementar un sistema de anti-aliasing robusto, capaz de solucionar los varios aspectos del aliasing. El approach que ha habido durante los últimos años refleja esta realidad fielmente. De pasar a usar multisampling y acelerarlo por hardware, a usar sistemas de post-procesado implementados completamente en software, pasando por métodos híbridos, hasta culminar en el anti-aliasing temporal.

El anti-aliasing temporal soluciona una gran parte de problemas del aliasing, como en bordes de geometría, en texturas, en reflejos especulares y además se puede reutilizar parte del pipeline (los motion vectors) para la implementación de un motion blur que solventa el aliasing de movimiento. Además de reaprovechar el pipeline para solventar el motion aliasing se puede usar para otros efectos visuales como el SSAO, que aportan todavía más realismo a las escenas. Su compatibilidad y rendimiento, al igual que los sistemas de post-procesado, hacen que sea un candidato perfecto para sistemas que se ejecutan en tiempo real, como los videojuegos.

Sin embargo, su implementación no es nada trivial. Si bien es cierto que su implementación para una imagen estática no requiere demasiada precisión, para escenas dinámicas, con reflejos y transparencias requieren que la implementación sea extremadamente precisa. Si no es así, aparecerán defectos visuales mucho peores que el aliasing, como por ejemplo el ghosting o un resultado excesivamente suave. Además no llegará a ser una solución drop-in, como pueden ser el FXAA o el SMAA, por ejemplo. Esos algoritmos, puramente de post-procesado, los puede llegar a inyectar el usuario, aunque el desarrollador no los haya incluido en su software. En cambio, el temporal anti-aliasing requiere información y modificaciones que están muy ligadas al motor gráfico usado.

Con la popularización del hardware específico para tareas de aprendizaje automático, tanto en tarjetas gráficas como en dispositivos móviles, el siguiente paso en el campo del anti-aliasing puede llegar a ser un algoritmo de este tipo. Estos algoritmos aplicados al software en tiempo real todavía están en su infancia, pero en otros entornos se usan con

muy buenos resultados.

En definitiva, el anti-aliasing temporal saca todo el potencial que ofrecen los motores gráficos y se convertirá en el sistema usado por defecto, reemplazando técnicas como el multisampling.

A un nivel más personal, este proyecto ha satisfecho una inquietud mía de sobre el funcionamiento de estas técnicas que tengo desde hace años. En un futuro se pretende extender el sistema para incluir mejoras como soporte para escenas dinámicas, transparencias, reflejos, motion blur, SSAO, etc.

Bibliografía

- [1] D. P. Mitchell y A. N. Netravali, «Reconstruction Filters in Computer-graphics», *SIGGRAPH Comput. Graph.*, vol. 22, n.º 4, págs. 221-228, jun. de 1988, ISSN: 0097-8930. DOI: 10.1145/378456.378514. dirección: <http://doi.acm.org/10.1145/378456.378514>.
- [2] A. R. Smith, «A Pixel Is Not A Little Square, A Pixel Is Not A Little Square, A Pixel Is Not A Little Square! (And a Voxel is Not a Little Cube)», Technical Memo 6, Microsoft Research, inf. téc., 1995.
- [3] K. Beets y D. Barron, «Super-sampling Anti-aliasing Analyzed», 2000.
- [4] M. E Goss y K. Wu, «Study of Supersampling Methods for Computer Graphics Hardware Antialiasing», vol. 12, ene. de 2001.
- [5] Nvidia, «HRAA: High-Resolution Antialiasing through Multisampling», inf. téc., 2001. dirección: https://www.nvidia.com/object/feature_hraa.html.
- [6] J. Sachs, «Image Resampling», inf. téc., 2001.
- [7] N. Sébastien Dominé, «OpenGL Multisample OpenGL Multisample», inf. téc., 2002.
- [8] 3. Center, *Multisampling Anti-Aliasing: A Closeup View*, 2003. dirección: http://alt.3dcenter.org/artikel/multisampling_anti-aliasing/index3_e.php.
- [9] H. Nguyen, *Gpu Gems 3*, First. Addison-Wesley Professional, 2007, ISBN: 9780321545428.
- [10] Crytek, «Crysis Next Gen Effect», inf. téc., 2008.
- [11] D. Glasner, S. Bagon y M. Irani, «Super-resolution from a single image», en *2009 IEEE 12th International Conference on Computer Vision*, sep. de 2009, págs. 349-356. DOI: 10.1109/ICCV.2009.5459271.
- [12] T. Lottes, «FXAA. NVIDIA White Paper», inf. téc., 2009.
- [13] A. Reshetov, «Morphological Antialiasing», en *Proceedings of the Conference on High Performance Graphics 2009*, ép. HPG '09, New Orleans, Louisiana: ACM, 2009, págs. 109-116, ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572787. dirección: <http://doi.acm.org/10.1145/1572769.1572787>.
- [14] P. Getreuer, «Linear Methods for Image Interpolation», vol. 1, sep. de 2011.

- [15] N. Timothy Lottes, «FXAA 3.11 in 15 Slides», inf. téc., 2011.
- [16] *What is Lanczos resampling useful for in a spatial context?*, 2011. dirección: <https://gis.stackexchange.com/questions/10931/what-is-lanczos-resampling-useful-for-in-a-spatial-context>.
- [17] J. Jimenez, J. I. Echevarria, T. Sousa y D. Gutierrez, «SMAA: Enhanced Subpixel Morphological Antialiasing», *Comput. Graph. Forum*, vol. 31, n.º 2pt1, págs. 355-364, mayo de 2012, ISSN: 0167-7055. DOI: 10.1111/j.1467-8659.2012.03014.x. dirección: <http://dx.doi.org/10.1111/j.1467-8659.2012.03014.x>.
- [18] *Quick Tip: How Does Shutter Speed Affect Video?*, 2013. dirección: <https://photography.tutsplus.com/articles/quick-tip-how-does-shutter-speed-affect-video--photo-12092>.
- [19] D. Shreiner, G. Sellers, J. M. Kessenich y B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, 8th. Addison-Wesley Professional, 2013, ISBN: 0321773039, 9780321773036.
- [20] Nvidia, «GeForce GTX 980 Whitepaper», inf. téc., 2014. dirección: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.
- [21] B. Wronski, *Temporal supersampling and antialiasing*, 2014. dirección: <https://bartwronski.com/2014/03/15/temporal-supersampling-and-antialiasing/>.
- [22] J.-B. Huang, A. Singh y N. Ahuja, «Single Image Super-Resolution from Transformed Self-Exemplars», en *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.
- [23] J. Jimenez, «Filmic SMAA Sharp Morphological and Temporal Antialiasing», inf. téc., 2016.
- [24] J. Kessenich, G. Sellers y D. Shreiner, *OpenGL®Programming Guide: The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V*, 9.^a ed. Addison-Wesley Professional, 2016, ISBN: 9780134495491.
- [25] N. D. Ku XU, «Temporal Antialiasing In Uncharted 4», inf. téc., 2016.
- [26] Nvidia, «NVIDIA TURING GPU ARCHITECTURE», inf. téc., 2018.
- [27] K. Nvidia, «OpenGL ARB texture multisample», inf. téc., 2009-2014. dirección: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_texture_multisample.txt.
- [28] U. Brian Karis, «High Quality Temporal Supersampling», inf. téc.
- [29] A. Courreges, *Metal Gear Solid V - Graphics Study*. dirección: <http://www.adriancourreges.com/blog/2017/12/15/mgs-v-graphics-study/>.

- [30] Durand y Cutler, *Sampling, Aliasing, & Mipmaps*. dirección: http://groups.csail.mit.edu/graphics/classes/6.837/F03/lectures/23_aliasing.pdf.
- [31] Google, *App resources overview, Android Developer Documentation*. dirección: <https://developer.android.com/guide/topics/resources/providing-resources#AlternativeResources>.
- [32] S. U. of New York, *Digital Radiography Image Artifacts*. dirección: <http://www.upstate.edu/radiology/education/rsna/radiography/artifact.php>.
- [33] Nvidia, *NVIDIA PostWorks*. dirección: <https://developer.nvidia.com/postworks>.

Índice de figuras

4.1.	Aliasing en una radiografía[32]	17
4.2.	Point Sampling[19]	18
4.3.	Aliasing en Hitman (2016)	19
4.4.	Patrón de Moiré[4]	19
4.5.	Anti-aliasing con Mip Maps	20
4.6.	Componentes de iluminación en OpenGL. Ambiente (superior izquierda), difuso (superior derecha) y especular (inferior)[19]	21
4.7.	Specular aliasing en Doom (2016)	22
4.8.	Comparación de tiempo de exposición (360° , 180° , 90° , 45°)[18]	23
4.9.	Grand Theft Auto 3: Vice City en PS2	23
4.10.	Per object motion blur en Doom (2016)	24
4.11.	Siluetas sobre objetos con blur en Crysis (2007)[10]	25
4.12.	Diferentes niveles de calidad de motion blur en Crysis (2007)[10]	25
4.13.	Comparación de interpolación usando diferentes filtros[16]	26
4.14.	Comparación de resultados de diferentes filtros[14]	26
4.15.	Resultados de SSAA en Borderlands 2	28
4.16.	Resultados de SSAA en Borderlands 2	29
4.17.	Ordered Grid Super-Sampling[3]	29
4.18.	Rotated Grid Super-Sampling[3]	29
4.19.	Quincunx Pattern[5]	30
4.20.	Multisampling	31
4.21.	Resultados del MSAA	32
4.22.	Generación de G-Buffer en Metal Gear Solid V: The Phantom Pain[29]	33
4.23.	Detección de bordes usando valores de luminancia[15]	33
4.24.	Escena con AA off, MSAA 4x y FXAA	34
4.25.	MLAA en blanco y negro	35
4.26.	MLAA con color	36
4.27.	Comparación de antialiasing en Halo 1 (2001)	37
4.28.	Super Res[26]	38
4.29.	DLSS[26]	38

5.1.	Jitter con secuencia Halton	40
5.2.	\vec{CP} [23]	41
5.3.	Ghosting[28]	44
6.1.	Anti-aliasing desactivado	47
6.2.	Temporal anti-aliasing con 8 samples	48
6.3.	Temporal anti-aliasing con 16 samples	48
6.4.	Reflejos especulares con anti-aliasing apagado	48
6.5.	Reflejos especulares con temporal anti-aliasing	48
6.6.	Filtro de sharpening desactivado	49
6.7.	Filtro de sharpening activado	50
6.8.	Temporal anti-aliasing desactivado	51