5-2013

Towards a filmic look and feel in real time computer graphics

Sherief Farouk

TOWARDS A FILMIC LOOK AND FEEL IN REAL TIME COMPUTER GRAPHICS

By

Sherief Farouk

B.S. Arab Academy for Science and Technology, 2009

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Science)

The Graduate School

The University of Maine

May 2013

TOWARDS A FILMIC LOOK AND FEEL IN REAL TIME COMPUTER GRAPHICS

By Sherief Farouk

Thesis Advisor: Dr. James Fastook

An Abstract of the Thesis Presented

in Partial Fulfillment of the Requirements for the

Degree of Master of Science

(in Computer Science)

May 2013

Film footage has a distinct look and feel that audience can instantly recognize, making its replication desirable for computer generated graphics. This thesis presents methods capable of replicating significant portions of the film look and feel while being able to fit within the constraints imposed by real-time computer generated graphics on consumer hardware.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EQUATIONS

CHAPTER ONE:
INTRODUCTION

Problem Statement

Film footage, due to the photography and production process, has a distinctive look that audiences are familiar with. Audiences can easily tell the difference between film and TV Soap Opera footage after a quick glance, and that affects their perception of the content.

Comedy Troupe Monty Python even used the visual difference between film and TV footage in one of their sketches, titled "The Royal Society For Putting Things On Top Of Other Things", where the characters upon moving to a different location where they're shot using outdoor film cameras exclaim "We've been trapped in film!" [Python 1970]

Computer Generated Imagery (CGI) has a very sharp, defined look that's a result of the rendering process that simulates a perfect camera with no visual errors operating with an infinitely fast shutter along a linear color space - a completely different situation from the one encountered while shooting and processing film footage.

This thesis aims to replicate some of the defining aspects of film footage on commodity consumer hardware in real-time in an attempt to bring CGI as closely as possible to film footage.

Objectives

This thesis aims to replicate a number of the visual artifacts seen in film in a real time computer generated rendering. The low hanging fruit like film grain and camera movement patterns, things where a trivial implementation method exists, have been tackled extensively by both academia and video-game development - the latter being a major driver of advancement in the field of real time computer graphics.

The effects tackled in this thesis are imaging system artifacts like depth of field, lens flare and color grading in addition to projection and display artifacts like color convergence, with the goal being visually acceptable models of simulation suitable for implementation as a part of a real time rendering pipeline running on standard consumer hardware governed by economical, performance and power budgets.

Graphics Processing Units (GPUs) started out with hard-wired algorithms and pipelines and as a result the look and feel of real time hardware accelerated rendering hasn't seen significant variations. With the recent introduction of GPUs with fully programmable pipelines these restrictions have been lifted and a new wave of non-traditional rendering models aimed at generating non-photorealistic imagery emerged such as cel shading where the resulting scene mimics hand-drawn cartoons (the technique was actually named after the celluloids, or cels, used in the production in two-dimensional animation).

The attempt to replicate the film look and feel will explore the new options offered by massively parallel programmable GPUs and try to come up with algorithms that replicate expensive traditionally ray-traced rendering features such as lens artifacts resulting from the diffraction, reflection and refraction of light inside an imaging system while taking advantage of the GPU architecture and fitting within the constraints required for real time rendering.

Review of Previous Work

Attempts at achieving a cinematic look and feel have happened within offline and real-time rendered computer graphics. Computer generated movies like Wall-E attempted to replicate cinema's look and feel to increase the viewer immersion - viewers have become accustomed to the specifics of cinematic format including its limitations and imperfections and while the viewers may not be actively aware of their recognition of cinema's look and feel they can be acutely aware of their lack of recognition of it. Peter Jackson's The Hobbit was screened at 48 frames-per-second, double the traditional cinema frame rate of 24 frames-per-second, and some of the audience came out calling it "camcorder-like" for better or for worse.

Video games have also attempted to create cinematic environments as consumer-grade graphics hardware started getting good enough to handle complex scenes and content creation pipelines became sophisticated enough to include actors for physical and facial performances coupled with motion-capture and facial-capture systems. Franchises like Uncharted have boasted about their cinematic gameplay experience, and the God of War franchise has gone to the length of simulating a virtual camera that moves like an actual cinematic camera on a dolly as opposed to the free camera common in computer graphics made possible by the fact that the scene is synthetic and camera placement or path does not have to conform to physical laws of acceleration and movement or respect barriers in its way. The Dead Space series did away with the in-game user interface or heads up display to maintain a movie-like feel and Mass Effect went to the length of adding artificial film grain - noise artificially added over the rendered frames to simulate the graininess imperfections of film material.

A significant amount of work in that direction, especially in real-time graphics, has not been in the form of traditional literature. The economics of the games industry result in a massive amount of research being done with the target being relatively low turn-around times and an end-product quality that's acceptable by the average consumer, and some times the work doesn't get any form of external peer review at all since implementation details could be considered a competitive advantage. Even for some published works less scrutiny is usually given, and the publication could consist entirely of a talk, an article or a white paper lacking the rigorous in-depth analysis traditionally done by academic literature.

CHAPTER TWO:

RENDERING OVERVIEW

Graphics Processing Units (GPUs) store the display image in a region of memory called the framebuffer. Most commonly an 8-bit per color channel format is used (for 24-bits or 32-bits total per pixel for RGB [Red/Green/Blue] or RGBA [Red/Green/Blue/Alpha - a value used for representing how opaque the pixel is and used for compositing and blending effects] framebuffers respectively) to represent color either in linear or non-linear space, usually referred to as gamma space where the intensity of a color c is equal to c^{γ}. The term gamma comes from the equation used by electronics engineers to describe the brightness response curve of a CRT monitor (equation 1) [Smith 1995]. In the equation ***I*** refers to the intensity or brightness, ***V*** is the voltage applied and ***k*** is a proportionality constant.

$$I = KV^{\gamma} \qquad (1)$$

Since CRT monitors and TVs historically had a response curve corresponding to gamma = 2.2 that is the non-linear framebuffer storage preferred by graphics cards.

The 256 possible values of the color components in unsigned 8-bit per component formats are represented with a floating-point number limited to the range [0, 1.0]. Any values outside this range are clamped to the range's minimum or maximum value by the display hardware.

High Dynamic Range

Color intensities in the real world are not limited to 256 quantized values - the difference between the brightest and darkest spots in a photograph can be on the order of 10,000:1 - such imagery is referred to as High Dynamic Range (HDR) imagery. Several formats are capable of storing images in greater than 8-bit per component precision, and among the most common is OpenEXR, which uses the IEEE 16-bit floating-point format (commonly referred to as *half* or *FP16*) which has one sign bit, five exponent bits and and ten mantissa (fraction) bits as shown in figure 1 to store color components.

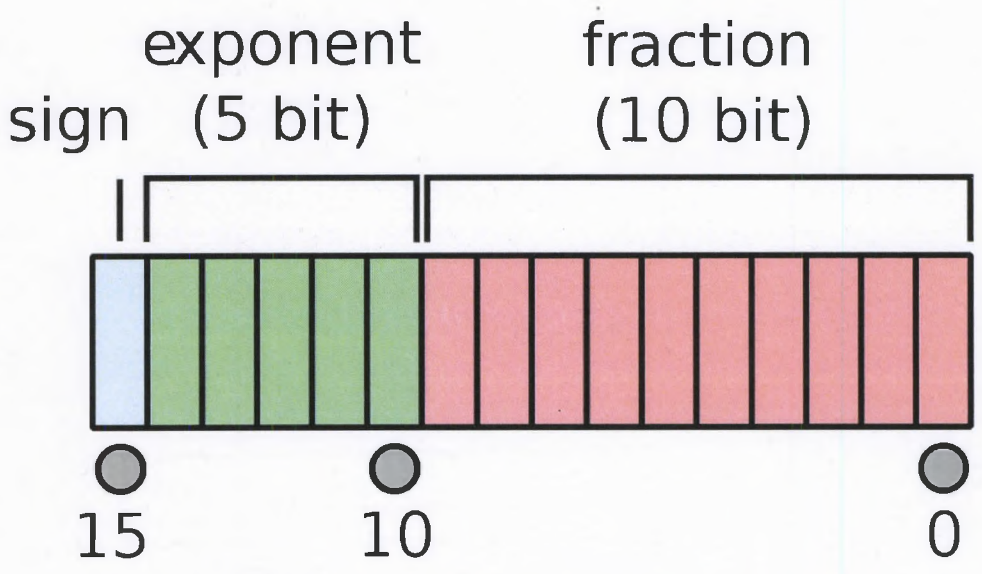


Figure 1: IEEE 16-bit floating point (half) type binary layout.

With a 16-bit floating-point representation the range of intensity values can be as low as 2^{-14} and as high as 65504. The format's bit-count imposes some restrictions on precision, with integer values between 0 and 2048 representable exactly and rounding used for higher values: 2049 to 4096 round to the nearest 2-multiple, 4097 to 8192 round to the nearest 4-multiple, 8193 to 16384 round to the nearest 8-multiple, 16385 to 32768 round to the nearest 16-multiple and 32769 to 65504 round to the nearest 32-multiple.

Conservation of Energy

Saturating framebuffers tend to lose color information outside of the range [0, 1], a scene featuring the sun and a lightbulb (two bright but drastically different objects) when stored in an 8-bit framebuffer might end up representing both the sun and the lightbulb with the same brightness value. Lowering the exposure of that scene in a later post-processing step will yield inaccurate results - color components with values above 1.0 are clipped, and so a region with a brightness of 1000 units would look the same as a region with a brightness of 10 units despite the two orders-of-magnitude difference in intensity - the format's limited precision has caused a loss of energy between the two transformations (rendering and exposure change).

Framebuffer formats with higher precision have been supported in hardware since NVIDIA's NV4X generation chips, known commercially as GeForce 6 Series, which introduced FP16 framebuffer format support. Newer GPUs like the NVIDIA G8X series, known commercially as GeForce 8 Series, introduced other high-precision formats suitable for rendering with more precision (full FP32, or float) as well as a format using the same 32 bits that the 8-bit-per-channel RGBA format uses, but yields a 4-fold increase in the number of color values available

(R10G10B10A2, a format that utilizes 10 bits for each of the Red, Green and Blue components and two bits for Alpha). Hardware support for such high precision formats coupled with higher precision in the fragment processing pipeline make it possible to avoid loss of energy when rendering scenes that rely on heavy / complex post-processing steps.

With saturating 8-bit framebuffers significant loss of data can occur, which can accumulate and magnify errors when post-processing is applied to the image data. One example would be the case of applying a blur on a very bright object - an object with RGB color values of (11.4, 2.4, 12.1) due to the object being a bright source of light (like a lamp) or having strong specular reflections would be stored in a saturating 8-bit framebuffer as an evenly white (1.0, 1.0, 1.0), and blurring would result in a uniformly gray blur along the objects edges - using an FP16 framebuffer preserves in the ratios between the RGB components with the result consisting of a white center with a halo that is mostly purple (since the dominant color components are red and blue) when viewed on an 8-bit display. Figure 2 shows such an object using high-precision pixel storage and the result of applying a Gaussian blur to it demonstrating the energy-conserving property of high-precision framebuffer representations.

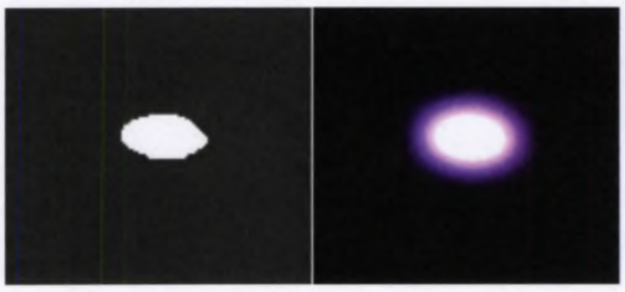


Figure 2: A high dynamic range region with color components exceeding the maximum displayable value of 1.0 as seen on a traditional display after clamping its value to 1.0 (left) and the result of applying a Gaussian blur to that same region - note the preservation of the purple hue around the highlight due to the high-precision framebuffer storage format used. A clamping framebuffer format would have a gray hue surrounding the highlight since all pixels will be stored as white (1.0, 1.0, 1.0) RGB tuples.

The Graphics Pipeline

The task of the GPU is to generate a two-dimensional image of a scene given that scene's description. GPUs started out as simple framebuffer-only devices responsible for managing two-dimensional pixel data and output to the display device, and the CPU was responsible for determining the values of individual pixels to generate the desired image and passing that over to the GPU.

Later GPUs gained significant computation power, and were designed to synthesize an image from a complex high-level polygonal description of a scene (Figure 3 shows an outline of a modern GPU pipeline).

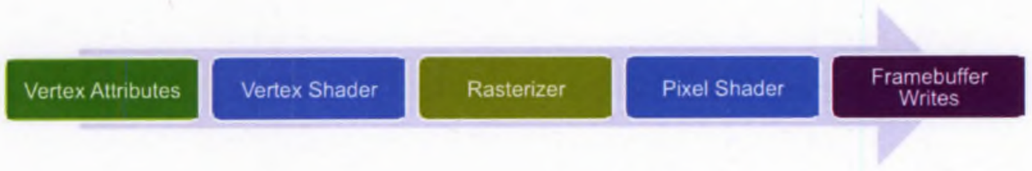


Figure 3: The hardware graphics pipeline. Data flows alongside the arrow and the Vertex Shader and Pixel Shader stages are fully programmable.

Modern GPUs start by taking in vertex attributes like positions and normals in a three-dimensional space and invoke a provided program on each vertex independently (and, logically, in parallel) to transform the vertices to the Normalized Device Coordinates [NDC] space describing the position within the image using a

[-1, 1] range. Other vertex-related attributes are computed at this step such as illumination (if not deferred to the pixel stage, which will be explained shortly).[1]

After each vertex receives a NDC position, and optionally a set of attribute values, the GPU's primitive assembly stage connects the transformed vertices to establish primitives such as points, point-sprites, lines, triangles or quads.

The assembled primitives are then handed on to the rasterization stage[2], where the screen-space pixels covered by each primitive are determined. The screen-space pixels that a primitive affects can be subject to other operations that determine a pixel's visibility.

There are various operations that determine a pixel's visibility. Among these are scissor testing, an operation whereby a pixel is accepted or rejected based on its screen coordinates; stencil testing, an operation whereby a pixel is accepted or rejected depending on values stored in a pixel-for-pixel Stencil Buffer that holds a special defining value for each pixel on display. Hidden surface removal, an operation that is usually handled by maintaining a per-pixel depth buffer that holds

[1] There are lots of different ways to generate an image using programmable hardware - this overview goes over the most common and traditional one, but different use-cases and clever tricks can synthesize images from very little input (such as vertex indices and fragment screen coordinates, tracked by the GPU without needing any external input for the vertex and fragment pipelines).

[2] More recent GPUs have an optional pipeline stage that comes after the vertex stage and primitive assembly known as the Geometry Shader stage - it operates on assembled primitives (and potentially adjacent primitives) and has the ability to kill existing primitives or emit entirely new ones to be passed on to the rasterizer stage. The Geometry Shader stage is not used by any of the techniques discussed here and therefore isn't discussed in detail. For more information [Blythe 2006] provides excellent coverage of the topic.

the distance from the camera to each pixel, such that when an incoming pixel has a depth value that is higher than the one that is already in the depth buffer at that location (signifying that it is farther away from the camera than the existing pixel and therefore is behind an already-rasterized primitive) it is rejected.

The rasterizer is also responsible for interpolating the attributes generated in the vertex stage across the surface of the primitive in a perspective-correct way, so that a color attribute at each vertex of a triangle (Figure **4**) will be smoothly interpolated[3] across the surface of the primitive (Figure 5).

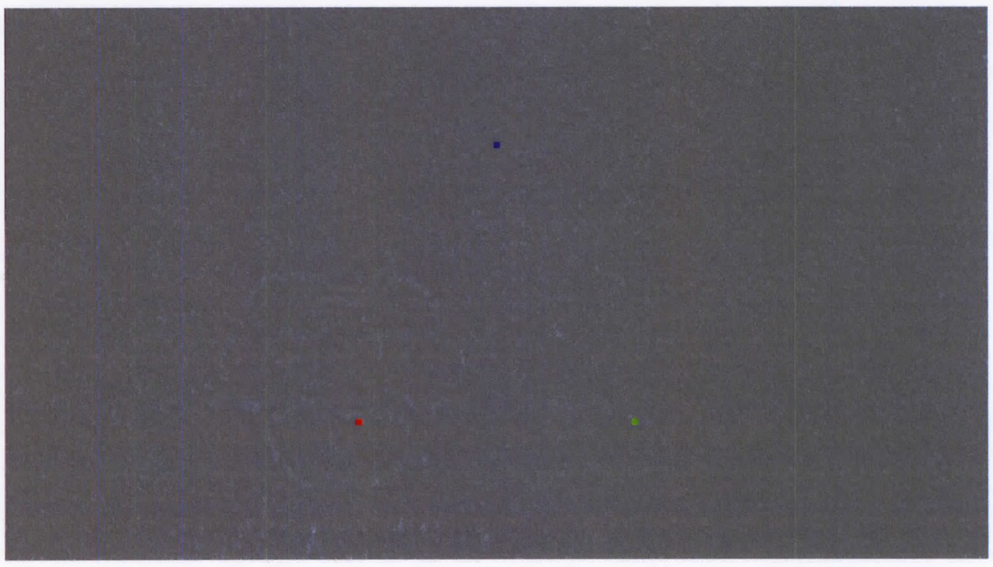


Figure **4**: Three vertices (emphasized), each with a color attribute (red, green and blue).

[3] While smooth interpolation is the default there are other interpolation options available (non perspective-correct, solid and centroid sampling to name a few) but they are beyond the scope of this thesis.

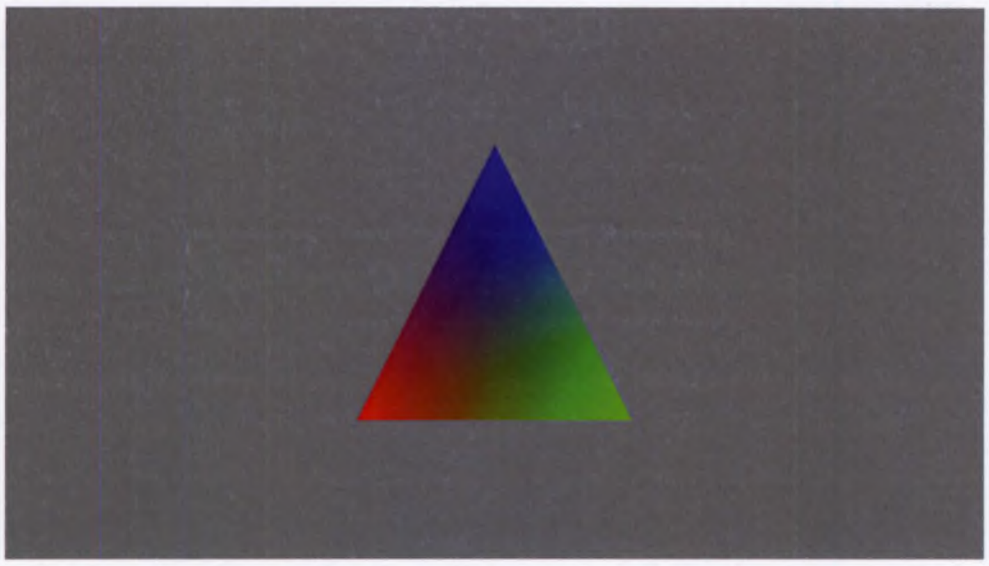


Figure 5: A triangle generated by connecting the vertices in figure 4 and interpolating the color values across the primitive surface.

After the rasterizer calculates the fragments covered by the primitive it passes them along with their interpolated per-vertex attribute values to the pixel shader stage, where a fragment program is executed once for every pixel. given that pixel's attributes, this program is responsible for calculating the final output colors of each pixel written to the framebuffer. Fragment programs on modern GPUs can be arbitrarily complex allowing for the use of per-fragment complex lighting models or simulation of exotic materials.

The programmable vertex and pixel shader stages can be given uniform parameters invariant across vertices / fragments for the object being rendered - as inputs, and they can also access the GPU's on-board memory in the form of either standard C-like arrays of data or in the form of GPU-stored images called textures.

A texture is a one, two or three dimensional collection of one to four component pixels (referred to as a texture element or a texture pixel, commonly abbreviated as texel) and an optional associated set of the identical images in decreasing dimensions known as the mipmap chain that is used for filtering. The mipmap chain is arranged into levels with the source image being level 0, a half-size image being level 1 and a quarter-size image being level 2 etc. all the way down to a 1x1 pixel representation.

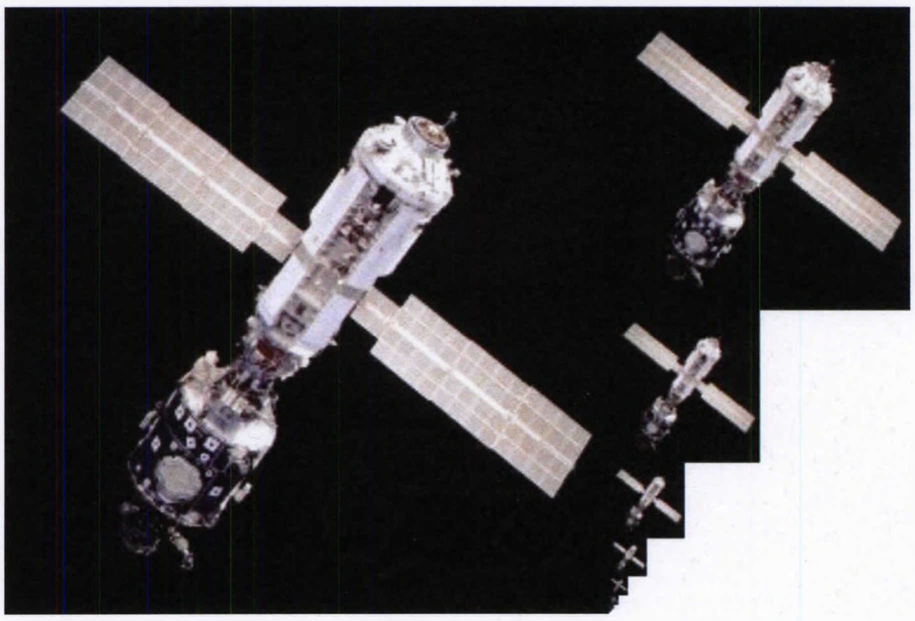


Figure 6: A texture (left) and its associated mipmap chain.

Textures are accessed using a variety of filtering methods and a normalized coordinate space. A texture image with 2 dimensions (m, n) and having from one to four color components per pixel can be uploaded to the GPU and accessed using normalized coordinates in the range [0, 1.0]. These are expanded by the GPU to the ranges [0, m] and [0, n] before sampling the image. In the case where fractional coordinates are generated, the GPU can use one of the supported filtering modes:

- Nearest filtering, where the nearest texel is retrieved.
- Bi-linear filtering where the resulting color consists of the texel closest to the requested coordinate plus a linear interpolation of the texels above, below, to the left and to the right of it, with weights corresponding to their distance from the fractional coordinates.
- Tri-linear filtering where two bi-linear filtered samples are taken from the two mipmap levels closest to the screen-space size of the textured surface and then linearly interpolated.
- Anisotropic filtering where the elongation of the primitive at the sampling point (determined by the screen-space derivatives of the primitive's width and height at the pixel being processed) is calculated and used to determine the number of samples fetched and interpolated to avoid perceived smudginess. Figure 7 shows a comparison between bi-linear filtering and anisotropic filtering.

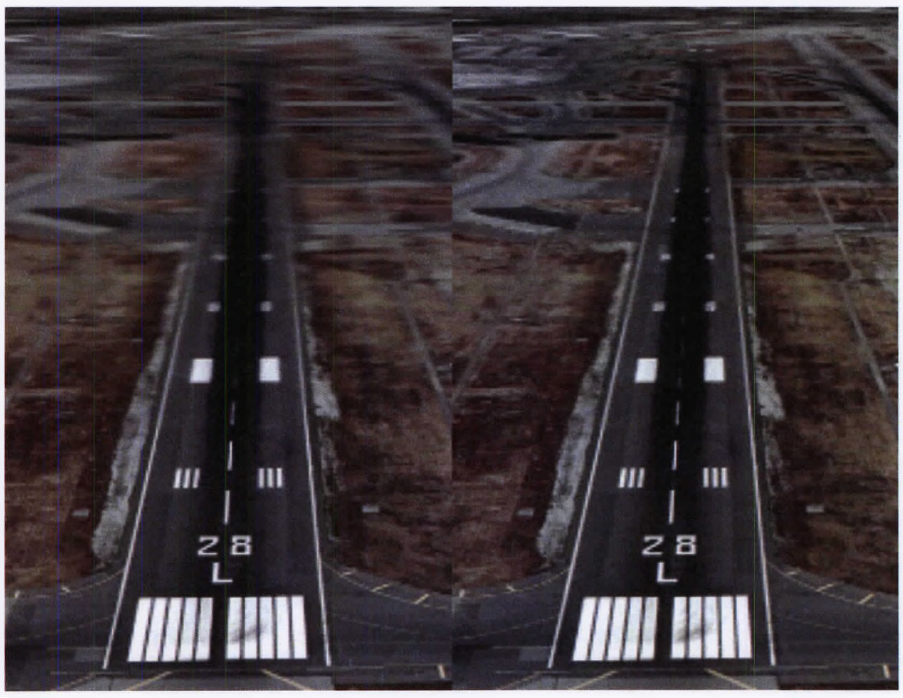


Figure 7: A texture-mapped surface using tri-linear filtering (left) and anisotropic filtering (right). Source: Wikimedia Commons.

Another special kind of texture is the cubemap texture often used to simulate reflective objects. A cubemap is a texture composed of six two-dimensional images logically arranged in a cube shape (hence the name), with each of the six images optionally having a mipmap chain. Figure 8 shows the structure of a cubemap texture representing a scene viewed from six different view points (facing top, bottom, left, right, forward and backwards).

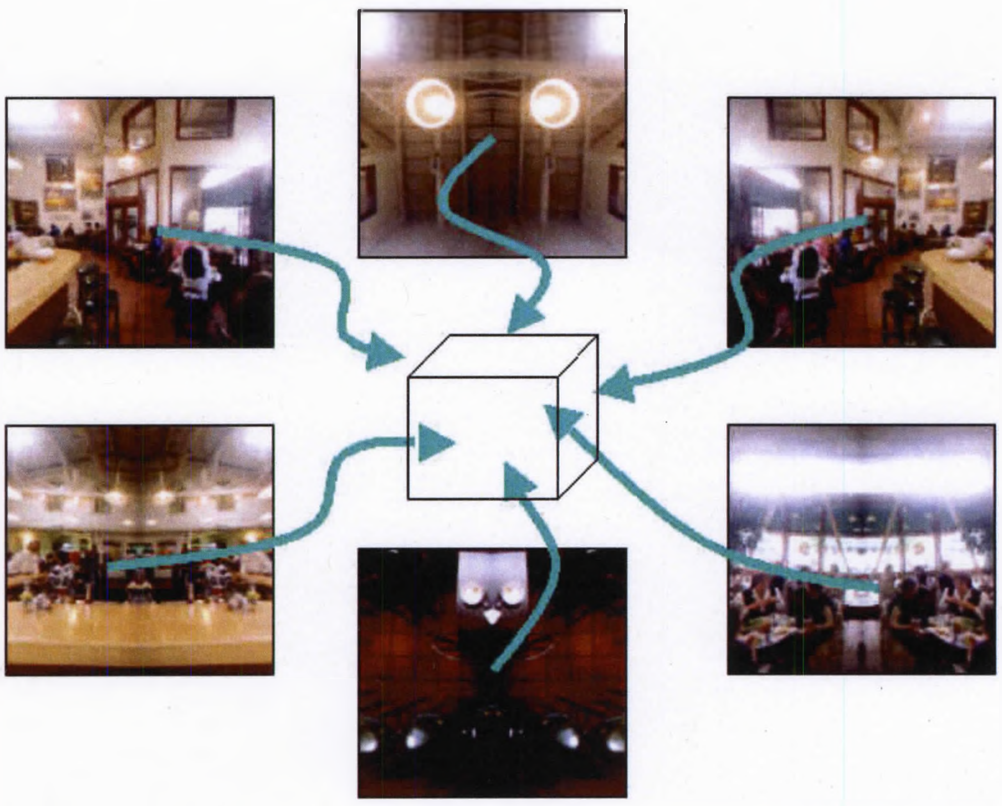


Figure 8: Six different viewpoints of a scene (top, bottom, left, right, forward and backwards) and their mapping onto the surface of a cubemap texture object. Source: http://www.nvidia.com/object/cube_map_ogl_tutorial.html

Cubemap textures are addressed using a three-dimensional vector representing a direction from the center of the cube, and the value returned is the result of sampling the cubemap at the intersection of the addressing vector with the cube's surface. Figure 9 shows an example of a vector used for addressing a cubemap.

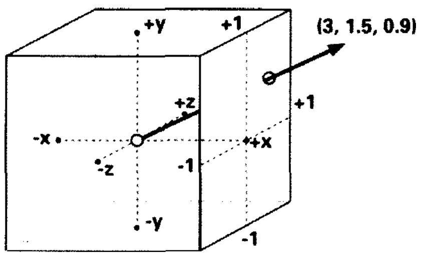


Figure 9: A cubemap texture and an addressing vector along with its intersection on the cubemap's surface. Source: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter08.html

Textures are sampled during fragment shading using per-vertex attributes commonly referred to as texture coordinates. Figure 10 shows a texture (left) and a quadrilateral polygon (right) rendered with the per-vertex attributes for texture coordinates (0, 0) assigned to its lower-left corner, (1, 0) assigned to its lower-right corner, (1, 1) assigned to its upper right corner and (0, 1) assigned to its upper-left corner. The texture coordinate attributes pass through the vertex stage unmodified, where they're interpolated across the surface of the polygon at every fragment. The fragment program uses these interpolated attribute values to sample the texture at every fragment and output that color, resulting in the texture appearing spread over the surface of the polygon.

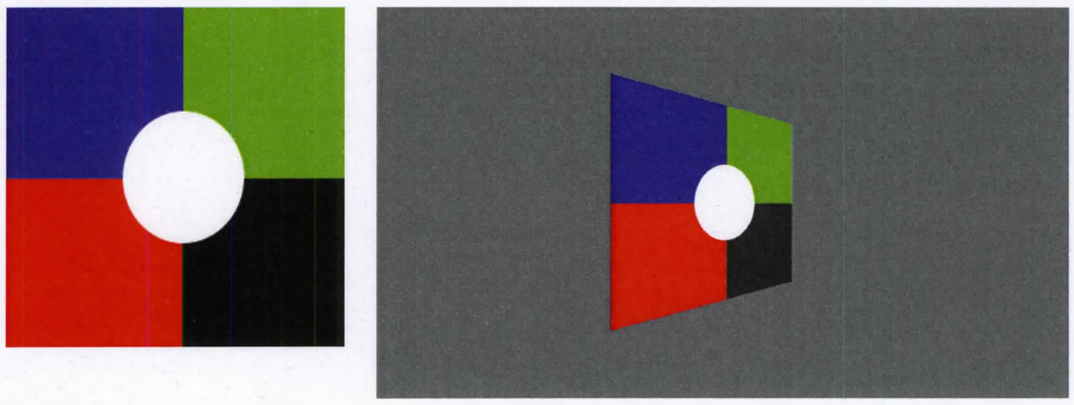


Figure 10: A two-dimensional texture image (left) and a square rendered in a three-dimensional space with perspective projection (right) using the image on the left for texture mapping.

GPUs have the ability to render output directly to one or more textures instead of the display, which is commonly referred to as render-to-texture functionality. Render-to-texture can be used as an intermediate step to dynamically generate textures used in the synthesis of the final displayed image. Figure 11 shows a scene with render-to-texture used to render a wireframe teapot to a texture that is later used to color the surface of a quadrilateral similar to the one in figure 10.

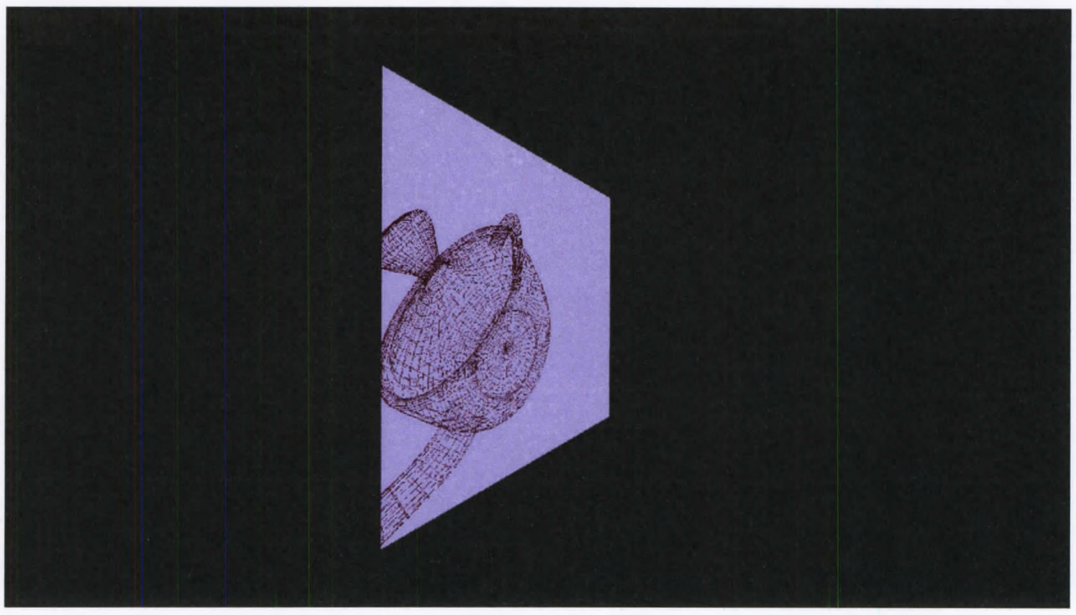


Figure 11: A wireframe teapot is rendered to a texture image and that image is later used to texture a quadrilateral polygon rendered to the display.

Render-to-texture functionality and the closely related concept of off-screen framebuffers enable significantly more complex rendering pipelines. With the introduction of the virtual framebuffer object a new functionality known as Multiple Render Targets (MRTs) was created, enabling the fragment program to output multiple color values to multiple framebuffers.

Figure 12 shows the result of a complex rendering pipeline incorporating Multiple Render Targets functionality. The original texture (left) is used as a texture for a quadrilateral rendered in three-dimensional space with a perspective projection to two texture-based framebuffers at the same time, with the fragment program modifying the output color for each of the two output framebuffers. The two framebuffers are then used as texture inputs for a second pass that renders two side-by-side view plane-aligned quadrilateral polygons textured with the contents generated by the first render pass.

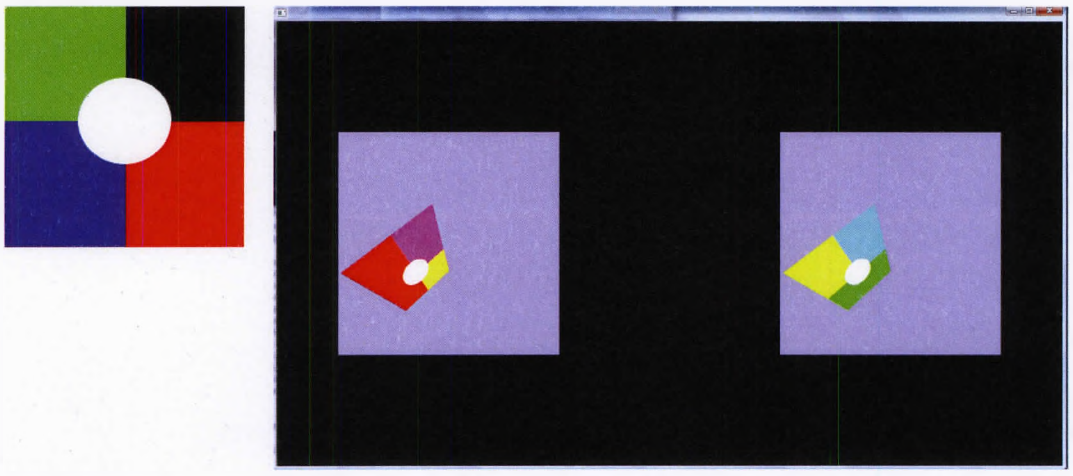


Figure 12: An example of a complex rendering pipeline made possible by using render to texture and multiple render targets functionality. A texture (left) is used to cover the surface of a quadrilateral rendered in a three dimensional environment using a perspective projection with the output written to two texture objects simultaneously, each receiving a different color tint in the fragment program. The resulting textures from the previous step are then drawn on view plane-aligned quadrilaterals in the following render pass.

Render-to-texture can be used to implement image post-processing by rendering the scene in its original form to a texture and then using that texture as input when rendering a view-plane aligned full-screen quadrilateral. Figure 13 shows a texture (left) and an OpenGL window (right) with a scene consisting of a single view-plane aligned full-screen quadrilateral using that texture.

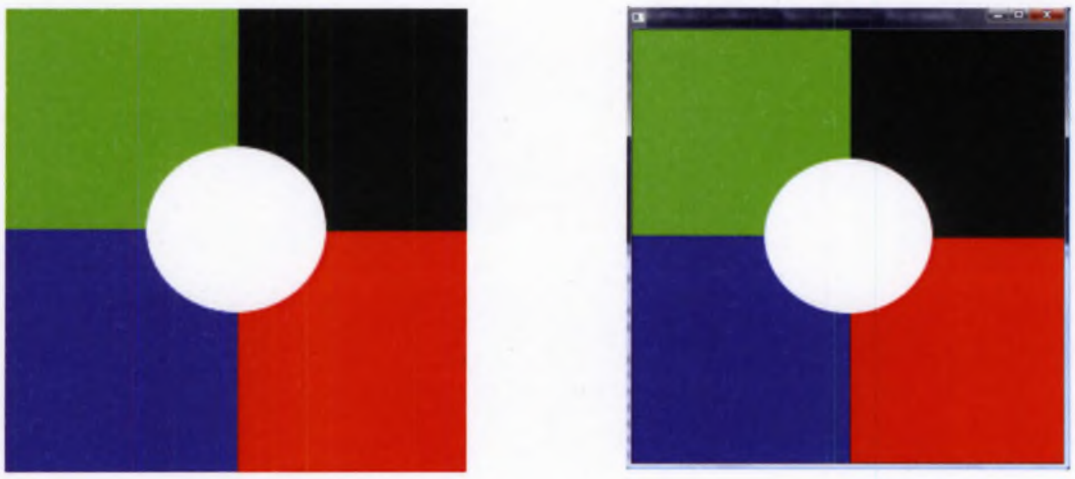


Figure 13: A texture (left) and an OpenGL window with a scene (right) with the texture overlaid on a screen-sized, view plane-aligned quadrilateral with a simple color pass-through fragment program.

Render-to-texture along with a fragment program that is not a trivial color pass-through (the most basic post-processing fragment program) can be used to implement a variety of post-processing effects. Figure 14 shows a black-and-white effect implemented with a fragment program. The fragment program would (at every screen pixel) sample the source texture, convert the sampled color to black-and-white using equation 2 and output that value as the pixel's color.

$$f(\hat{x}) = \hat{x} \cdot (0.299, 0.587, 0.114) \qquad (2)$$

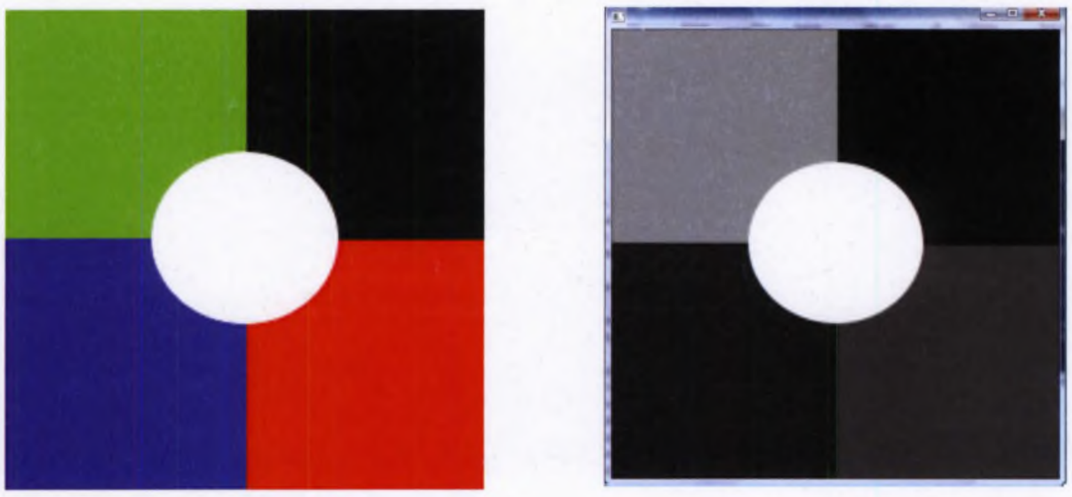


Figure 14: A texture (left) and an OpenGL window with a scene (right) with the texture overlaid on a screen-sized, view plane-aligned quadrilateral with a fragment program that applies equation 2 to the texture's sampled color before outputting it as the final display color.

More complex post-processing pipelines can be created by using two or more textures and alternating between using them as source and destination for rendering. An initial scene, for example, can be rendered to a texture T_1 followed by a post-processing pass rendering the processed scene from T_1 into a texture T_2 followed by another post-processing pass this time using T_2 as its input and T_1 as its output, etc.

ANTI-ALIASING

The images generated by rasterization hardware are the result of sampling polygon coverage and shading at every screen pixel, and as with any sampling function in digital signal processing aliasing can occur - two different signals could look identical (or alias) as a result of insufficient sampling resolution precision.

In rasterized computer graphics aliasing manifests itself spatially as jagged stair step-like edges on polygons. Figure 15 shows a real-time computer generated scene (left) and a magnified close-up (right) showing the jagged edges of polygonal objects.

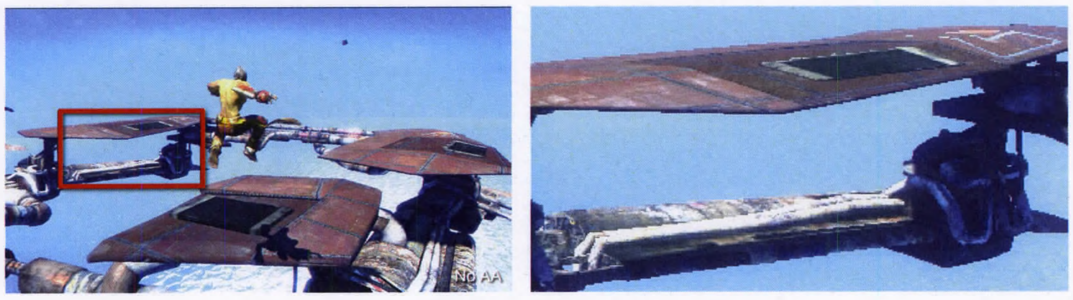


Figure 15: Computer generated scene with a highlighted region (left) and a close up of the highlighted region (right) showing spatial aliasing at polygon edges.

Anti-aliasing in computer graphics is an attempt to fix that problem. Figure 16 shows the same scene from figure 15 with morphological anti-aliasing [Reshetov 2009] applied.

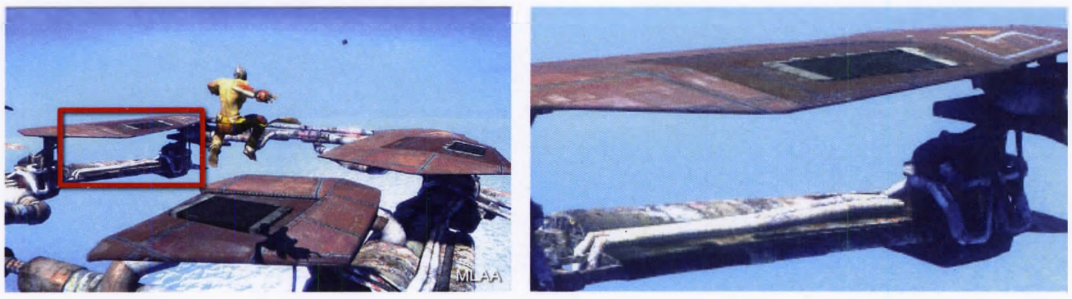


Figure 16: Anti-aliased computer generated scene with a highlighted region (left) and a close up of the highlighted region (right) showing a significant decrease in spatial aliasing compared to the same scene and region from figure 15.

Graphics hardware has traditionally implemented anti-aliasing by rasterizing the scene into a render buffer with more than one sampling point per pixel, effectively rasterizing the scene's polygons at a higher sampling resolution than the final resolution . Figure 17 shows a close up of a single anti-aliased pixel with four sampling points (represented by the black circles) having three solidly colored triangles rasterized across its sub-coordinates.

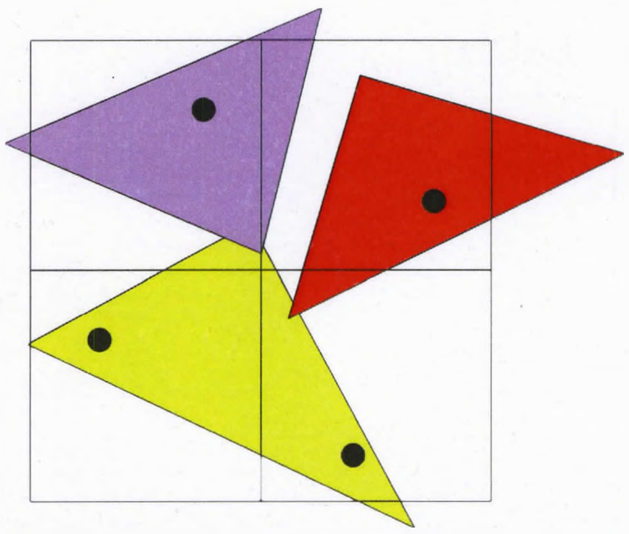


Figure 17: A close up of a single anti-aliased pixel with four sub-pixel sampling points. Image from [Burrows 2004].

The polygons are rasterized at sampling point resolution and a color value for each sampling point is stored. Figure 18 shows the sampling point color values resulting from the rasterization of the polygons in figure 17 at sampling point resolution.

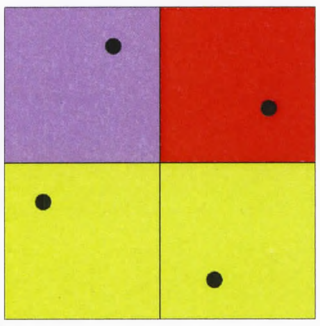


Figure 18: Color values stored at each sampling point from the rasterization of the polygons in figure 17. Image from [Burrows 2004].

After the scene is rendered to the sub-pixel sampled render buffer it is then resolved into a framebuffer with the desired pixel dimensions, usually by linearly averaging the color of the subsamples in each pixel. Figure 19 shows an example of the resolve step acting on the sub-sampled pixel shown in figure 18.

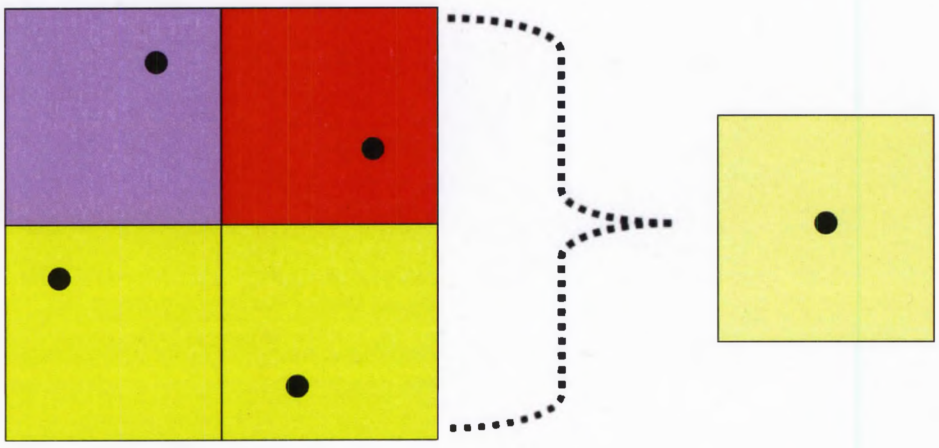


Figure 19: Resolving the four sub-pixel samples from figure 18 into a single pixel by linearly averaging the four colors. Image from [Burrows 2004].

When the same polygon covers multiple sample points in the same pixels two approaches are used by hardware anti-aliasing implementations to determine the shading frequency, which in a programmable pipeline is how often to execute the fragment program.

In super-sampling anti-aliasing a single polygon covering *n* sampling points in a pixel will have the fragment program executed *n* times, once for each fragment, and the *n* different output values stored for each sample. In multi-sampling anti-aliasing a single polygon covering *n* sampling points in a pixel will have the fragment program executed once for that pixel and the fragment program's output value copied into the *n* samples' storage location. Figure 20 shows a polygon being rasterized into sub-sampled rendering buffers with super-sampling anti-aliasing (left) and multi-sampling anti-aliasing (right).

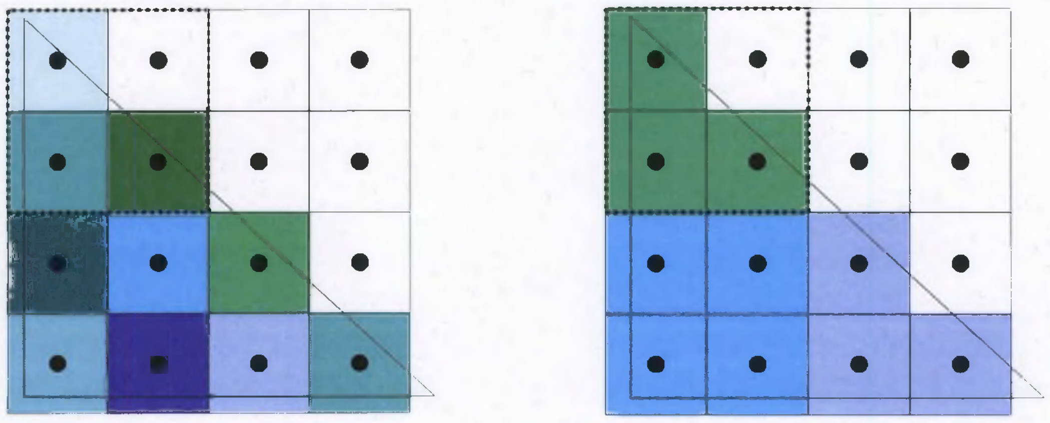


Figure 20: Two identical polygons with high-frequency shading rasterized to sub-sampled buffers with four samples per pixel. Each square represents one sub-pixel sampling point and the large square with a dotted edge on the top left represents the area of one pixel. The polygon on the left is being rasterized into a super-sampled buffer and has each sampling point storing a separate color as a result of each sampling point having executed its own invocation of the fragment program. The polygon on the right is being rasterized into a multi-sampled buffer and has all the sampling points in each pixel storing the same color value as a result of the fragment program being executed only once per pixel. Image from [Burrows 2004].

Anti-aliasing doesn't necessarily have to be implemented using the built-in hardware support and algorithms. Various methods to perform anti-aliasing as a post-process have been proposed: [Chajdas et. al 2011] used sub-pixel reconstruction, [Lottes 2011] used luminosity-based edge detection and blending, [Kaplanyan 2010] used a color deviation based blur coupled with temporal anti-aliasing on distant objects occupying a small screen-space region in an animated scene and [Reshetov 2009] used a CPU-based anti-aliasing which bypasses the GPU entirely. Deciding which anti-aliasing algorithm to use is a complex decision based on many factors tied to the rendering pipeline in use, although post-processing based anti-aliasing algorithms can be added to any pipeline that produces a final (Red, Green, Blue) image as its output.

PIXELS VERSUS FRAGMENTS

Pixels and fragments are two closely-related concepts involved with rasterization-based GPUs, and as a result some confusion about what they represent is common.

A fragment is the unit of input to the fragment processing stage and the fragment program. A fragment consists of multiple data points such as raster position, depth value, stencil value, sub-sample index and any attributes interpolated from the vertex processing stage. A pixel is the output result of fragment processing, and it is merely a one to four element tuple representing the color of the pixel.

CHAPTER THREE:

DEPTH OF FIELD

INTRODUCTION

A lens aperture maps every point in a captured scene to a *Circle of Confusion* (CoC) on the image plane. At a certain distance range the CoC is small enough to be considered a point and the objects to appear sharp and free of any blur - this range is called the *Depth of Field* (DoF). Points outside the DoF are projected on the image plane as circles-of-confusion with a diameter proportional to the size of the lens and their distance from the image plane. DoF is also affected by multiple lens parameters such as the Focal Length and the f-number.

Figure 21 shows a lens image capture system (with aperture ***A*** and focal point at distance ***F***)and an object at distance ***D*** from the lens which has a plane in focus at distance ***P***. Since ***D*** and P are different the object does not lie on the focus plane and therefore its image will appear out of focus on the lens's image plane (at distance ***I*** from the lens) - each point on the object will map to a circle of confusion of diameter ***C*** on the image plane.

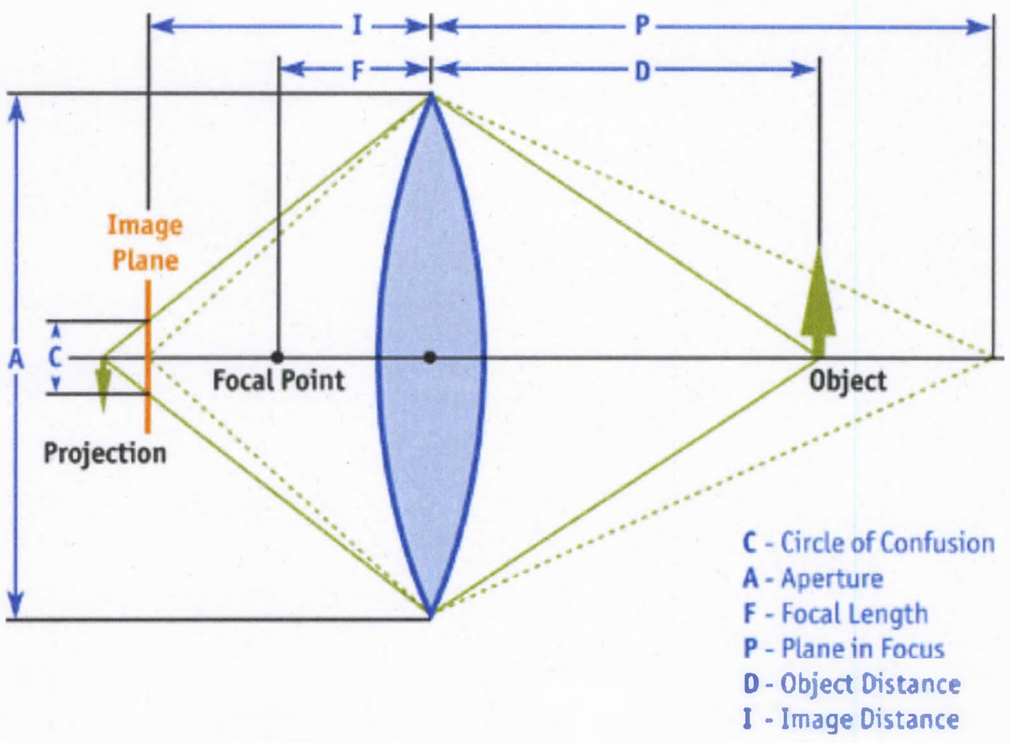


Figure 21: Lens-based Image Capture System and its Circle of Confusion. Image from [Hammon 2007].

Due to the nature of visible light and the range of wavelengths it covers there are some other subtle side-effects related to DoF, like Chromatic Aberration and Bokeh.

Chromatic Aberration is an image effect caused by the inability of optical lenses to focus the entire range of visible light at one point - the Index of Refraction $\boldsymbol{n}$ of a material can be stated as λ / λ_m where λ is the wavelength of light in vacuum and λ_m is the wavelength of light in the material causing the refraction and as a result different wavelengths of light corresponding to different visible colors have slightly different refraction indices different wavelengths of light have different refractive indices ($\boldsymbol{n}$) and are therefore focused at slightly different focal points.

Figure 22 shows an example of chromatic aberration affecting three different wavelengths of light (red, green and blue). The rays entering the lens at the same point get focused at different focal points due to their wavelength affecting their indices of refraction.

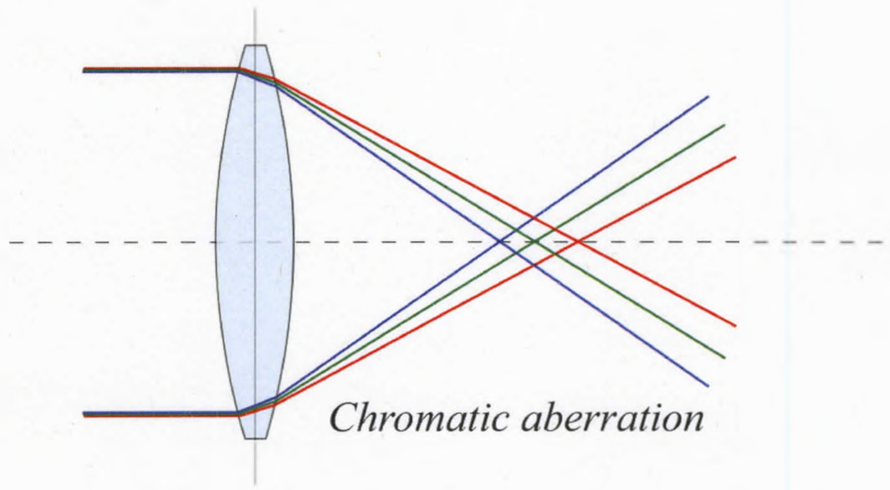


Figure 22: The physics involved in Chromatic Aberration - different wavelengths (colors) have different refractive indices resulting in each color's focal point being slightly different from every other color's. Source: Wikimedia Commons

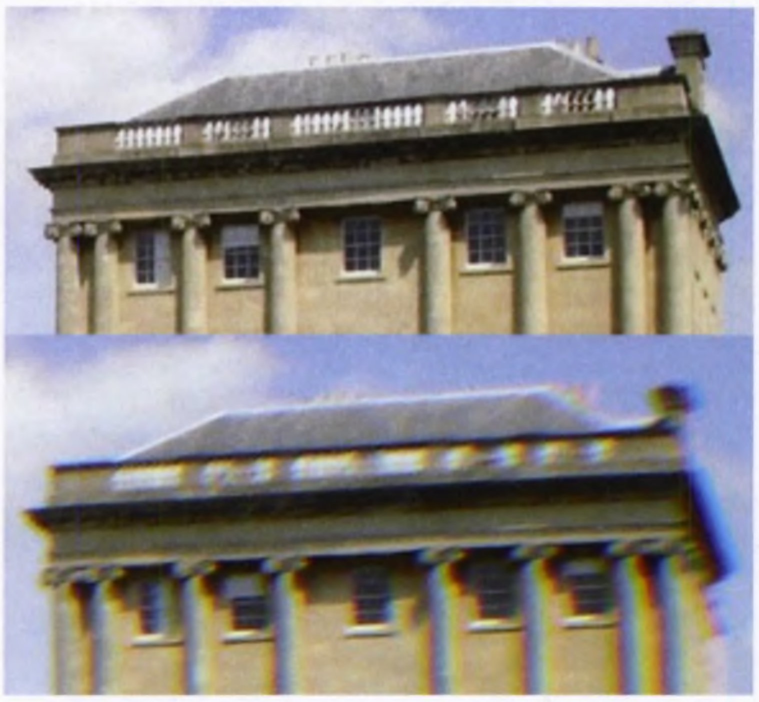


Figure 23: Clear image (top) and the same image with severe Chromatic Aberration. Source: Wikimedia Commons

In Figure 23, the top image is a reference image taken with an imaging system that exhibits little-to-none chromatic aberration while the bottom image exhibits severe chromatic aberration, with the most visible effect being the strong color separation. Chromatic aberration isn't uniform, and the distance between the foci increases the farther you get from the center of the lens. The lower image also exhibits ***barrel distortion*** (the apparent curvature of the columns that increases towards the right) but that is a separate error which occurs alongside chromatic aberration due to the geometry of the lens itself.

Imaging systems with minimal chromatic aberration still affect the final image though in a more subtle way. One of the most common effects of chromatic aberration is the phenomenon known as Purple Fringing - a purple outline that appears around mostly dark regions in a scene (figure 24). Chromatic aberration causes some of the colors of an object to move out of focus purple fringing is a manifestation of that.

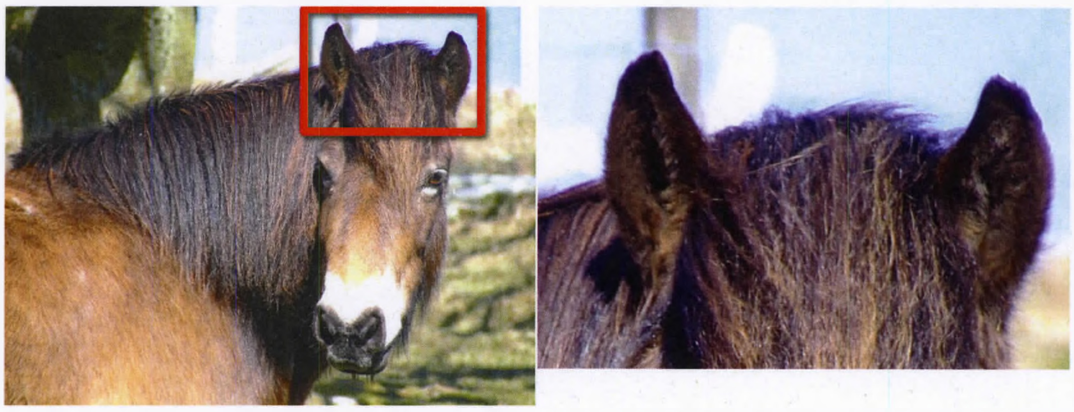


Figure 24: Example of Purple Fringing, a term used to describe the out-of-focus purple "ghost image" around object edges, a subtle yet very common result of Chromatic Aberration. Source: Wikimedia Commons

Accurate depth of field is a desirable property, as the effect contributes heavily to shot composition. Figure 25 shows a scene from The Dark Knight with shallow depth of field, a blurred background, a blurred foreground object and an in focus central character (Batman).

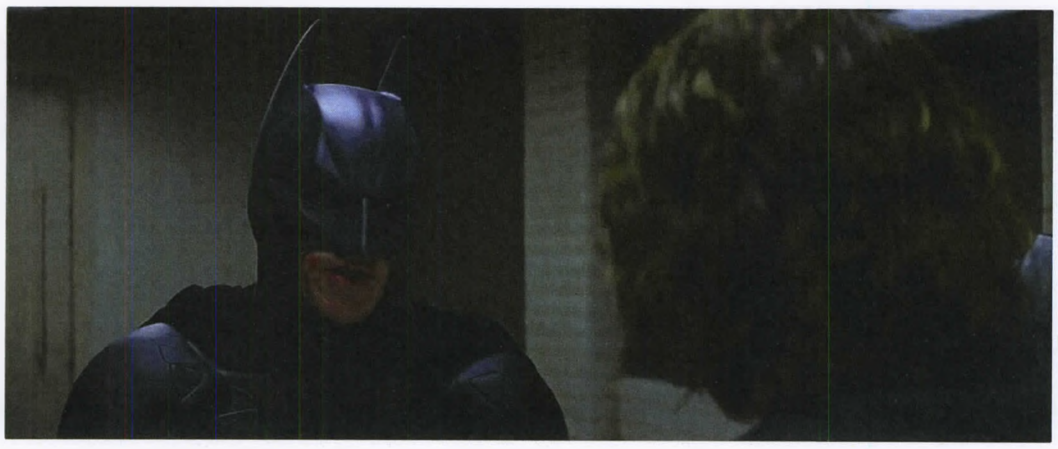


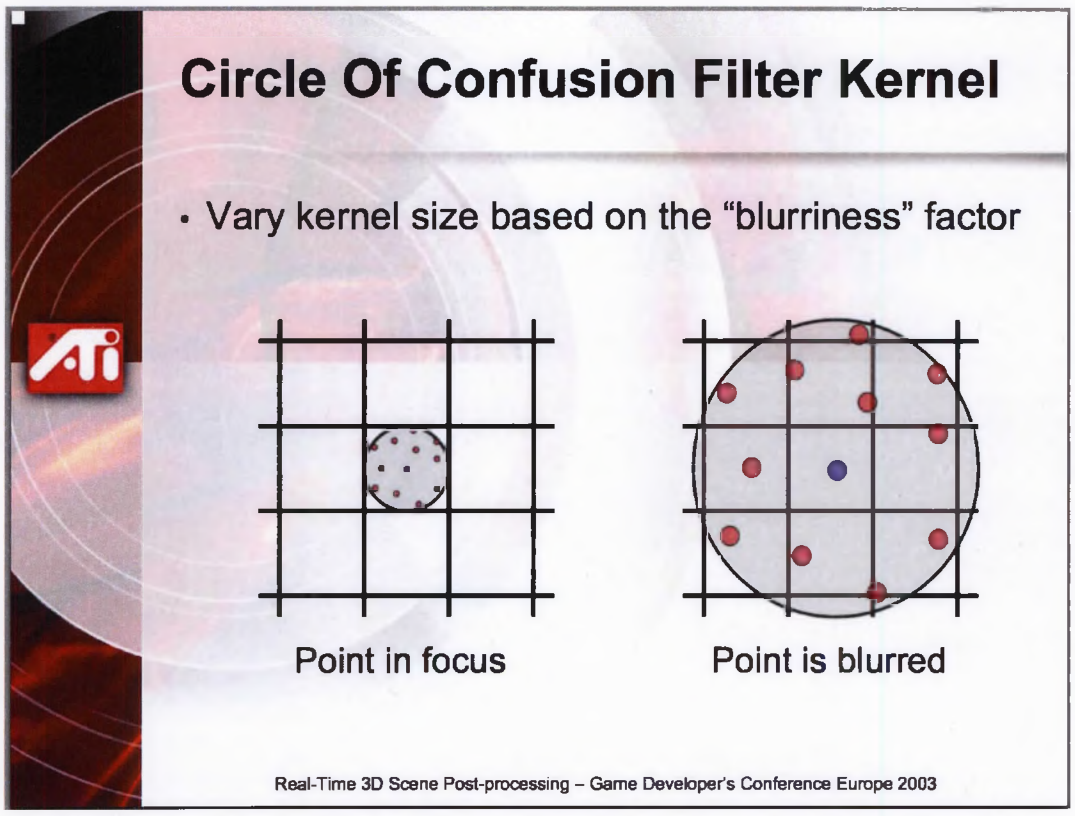
Figure 25: Scene from The Dark Knight. Depth of field is a major contributor to the shot's final look with the shallow depth of field drawing the viewer's attention to the central character (Batman).

Previous Work

The majority of real-time depth of field implementations so far have relied on post-processing a single view image coupled with per-pixel depth info - those can be split into two major categories: Gather Methods and Scatter Methods.

Gather methods run as a post-processing filter that, at each pixel, computes the size of the circle of confusion and uses that size to determine which coordinate from the source image should be fetched and processed to generate the final output color value for that pixel in the image [Rokita 1996; Riguer et al. 2003; Oat 2003; Scheuermann 2004; Bertalmio et al. 2004; Earl Hammon 2007; Zhou et al. 2007; Lee et al. 2009]. Those filters are fast enough to use in an already GPU-taxing rendering pipeline in real-time applications such as video games. Gather methods, however, suffer from *intensity leakage* whereby sharp foreground objects having a leaky "halo" that bleeds over background objects.

Figure 26 shows the basis of the gather method used in [Oat 2003] - a circle of confusion size is calculated by the fragment program at every pixel based on the pixel's distance from the view plane and the camera parameters and that size was used to scale a number of sampling coordinates (red) centered around the pixel coordinates being processed (blue) and the scaled sizes were used to sample the source image with the final color result being the linear average of the sampled values.


Circle Of Confusion Filter Kernel

- Vary kernel size based on the "blurriness" factor

Real-Time 3D Scene Post-processing – Game Developer's Conference Europe 2003

Figure 26: Post-process depth of field filter sampling pattern used in [Oat 2003].

Gather methods usually suffer from intensity leakage, the blurring of in-focus objects onto out of focus parts of the scene where the two intersect on the view plane. Figure 27 shows an example of intensity leakage in the PlayStation 3 video game *Uncharted 2: Among Thieves* with the foreground objects in the highlighted and magnified region (sandbags) having a 'halo' effect around their edges that intersect with the out of focus background.

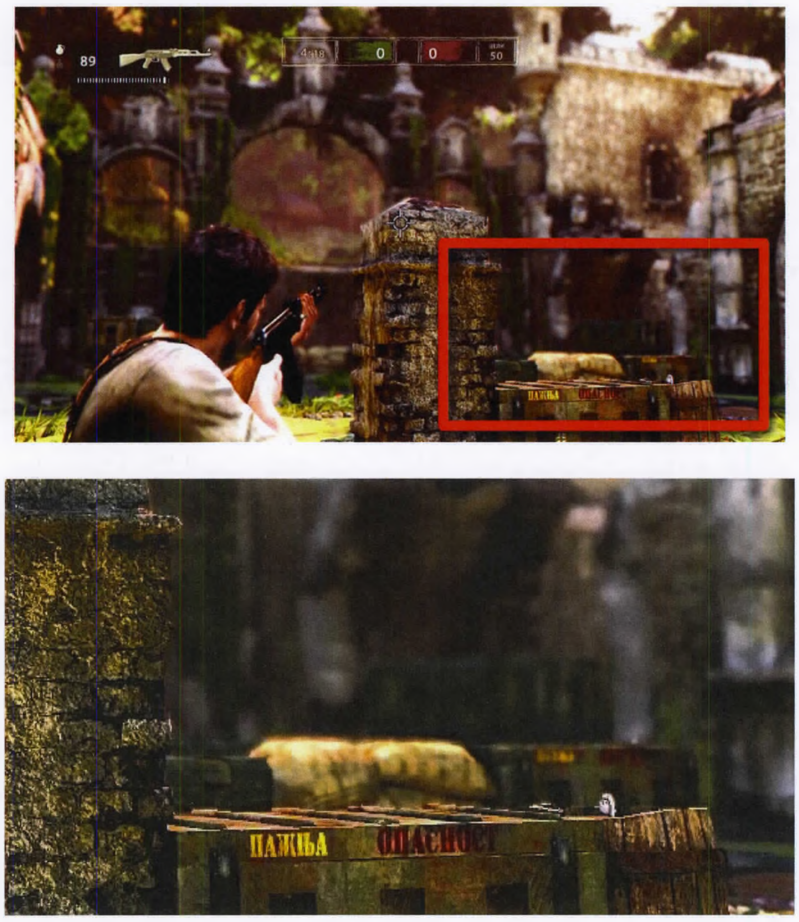


Figure 27: Example of intensity leakage from the PlayStation 3 video game Uncharted 2: Among Thieves. The close-up at the bottom of the highlighted region in the top image shows the foreground in-focus objects leaking into the scene's blurred background.

Intensity leakage in gather methods is usually a side-effect of averaging contributions from a number of pixels to determine the final color of a pixel since the pixels samples wouldn't necessarily belong to objects that all have a similar view-space distance. In the example in figure 27 the contributions from pixels belonging to the foreground sandbags object sampled while rendering the out of focus background pixels that are close to the sandbags in screen space cause the sandbags' color to leak onto the background.

A Scatter Method starts with the input pixels and for each pixel a circle of confusion size is calculated based on its distance from the view plane and camera parameters, just like in gather methods. After the circle of confusion size is calculated the algorithm determines which pixels across the output image the current input pixel affects and proceeds to accumulate the input pixel's contribution to the output pixels.

Figure 28 shows depth of field achieved using a scatter method in the Good Samaritan technology demo. The algorithm achieves scatter writing by generating for each input pixel a non-opaque screen-aligned polygon whose size is proportional to the computed circle of confusion size and additively blends those polygons onto the output image using the alpha-blending hardware.

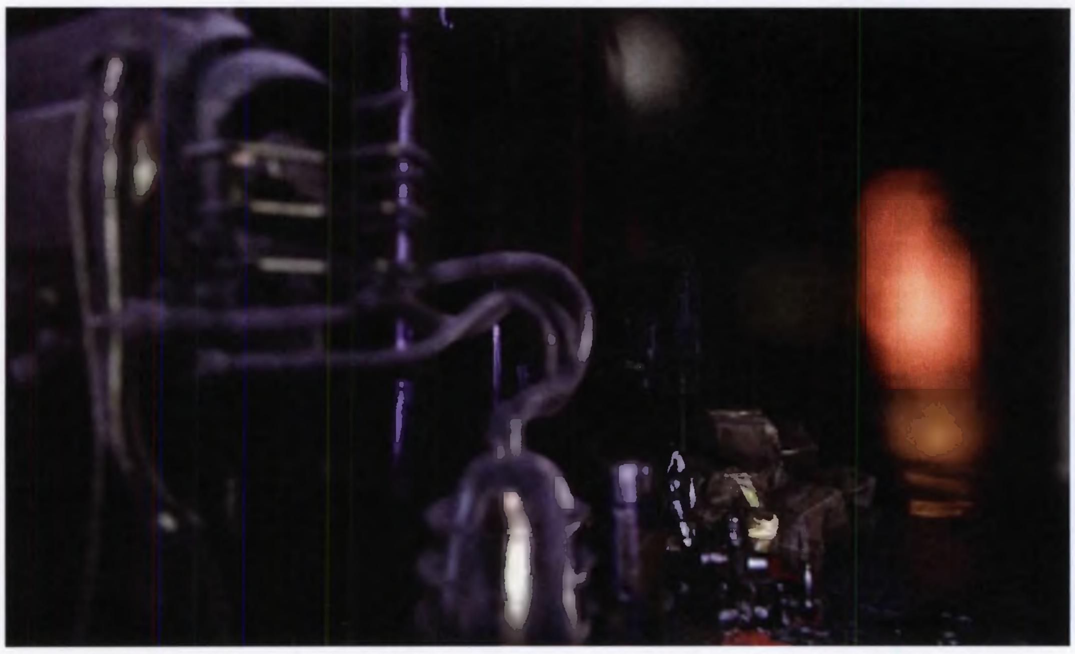


Figure 28: Depth of Field in the Good Samaritan technology presentation.

To avoid intensity leakage the Good Samaritan demo divides the scene into three regions (Figure 29): blurred foreground (closed to the camera than the focus plane), in focus region (at or near the focus plane) and a blurred background (farther from the camera than the focus plane).

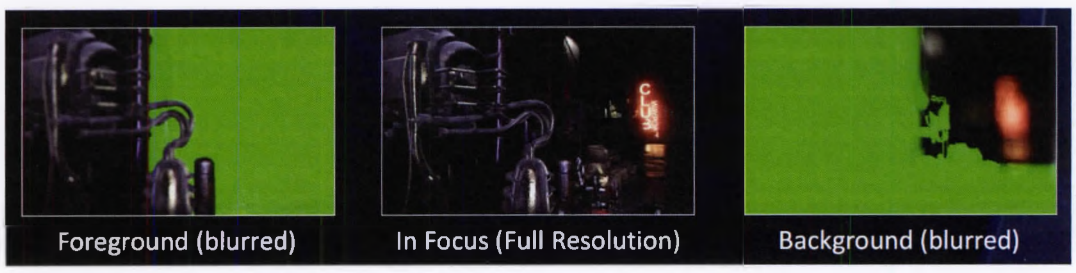


Figure 29: Images from The Good Samaritan presentation highlighting the scene's division into near, far and in-focus regions.

The algorithm used in the Good Samaritan technology demo results in improved image quality but the cost is significantly higher as it taxes the limited number of pixels a GPU can write to a framebuffer every second (referred to as the fillrate), the GPU's vertex processors and rasterization hardware (since an extra polygon is rendered for every pixel) and memory bandwidth used to blend the semi-transparent polygons with the rendered scene background. The technique used in Epic Software's Good Samaritan demo required three top-of-the-line at the time (March 2011) GeForce GTX 580 GPUs running in parallel.

A side-effect of the majority of both gather and scatter methods is that the depth values in the original depth buffer, after applying the Depth of Field effect, no longer represent the depth of the pixels at their raster coordinates - in regions where a near objects bleeds onto regions occupied by a far object. The depth buffer will hold the far value, but the color pixels consist of a mixture of the values from the near and far colors. This isn't a major inconvenience in itself, but the depth buffer is sometimes used as input to a number of other techniques down the line [Mittring 2007, Mittring 2009]. The depth buffer can be used as input for shadowing, fog, particle systems, motion blur and edge based anti-aliasing amongst other things.

Single-image based methods, which operate mainly as a framebuffer post-process, result in inaccurate results in cases where hidden surfaces would affect the final image and a common workaround is to split the scene into a number of layers and sort the objects into each of those layers either during a forward rendering step or as a post-processing step [Barsky et al. 2002, Kraus and Strengert

2007, Lefohn et al. 2006; Kosloff and Barsky 2007, Lee et al. 2009a, Kolsoff et al. 2009, Lee et al. 2009b]. Each layer requires its own depth-buffer for hidden surface removal so multiple render target functionality is usually unsuitable for accelerating layer rendering unless a custom (and therefore non-hardware accelerated) hidden surface removal method is used. On hardware with support for the geometry shader pipeline stage polygon duplication in the geometry stage couples with multiple render targets can remove the need for rendering the scene n-times for n-layers, although the cost is still noticeably higher than for single-view rendering. More recent work has attempted to reproduce lens-related artifacts like chromatic aberration and spherical aberration [Lee et al. 2010].

Proposed Technique

The proposed technique is based on [Oat 2003]'s approach of using a dynamically-sized circle of confusion with a fixed number of texture samples per pixel.

The algorithm works at every pixel by calculating the circle of confusion size based on the pixel's view-space distance from the camera and the depth of field distance and range. After the canonical circle of confusion size $\boldsymbol{S}$ three circle of confusion sizes for the red, green and blue channels $\boldsymbol{S_r}$, $\boldsymbol{S_g}$, $\boldsymbol{S_b}$ are calculated by scaling the original size $\boldsymbol{S}$ by three variables $\boldsymbol{d_r}$, $\boldsymbol{d_g}$, $\boldsymbol{d_b}$ specifying the differences in circle of confusion sizes for each color channel which corresponds directly to how off-center the colors will focus with respect to the lens's canonical focal point.

A number of texture sampling coordinates $\boldsymbol{C} = \{ \boldsymbol{C_0}, \dots, \boldsymbol{C_n} \}$ distributed around a unit circle (figure 30) are supplied as a uniform input to the algorithm and once the circle of confusion sizes are calculated three sets of texture sampling coordinates $\boldsymbol{C_r}$, $\boldsymbol{C_g}$, $\boldsymbol{C_b}$ are generated by centering the texture sampling coordinates from $\boldsymbol{C}$ around the current pixel's coordinates and scaling them by $\boldsymbol{S_r}$, $\boldsymbol{S_g}$ and $\boldsymbol{S_b}$ respectively.

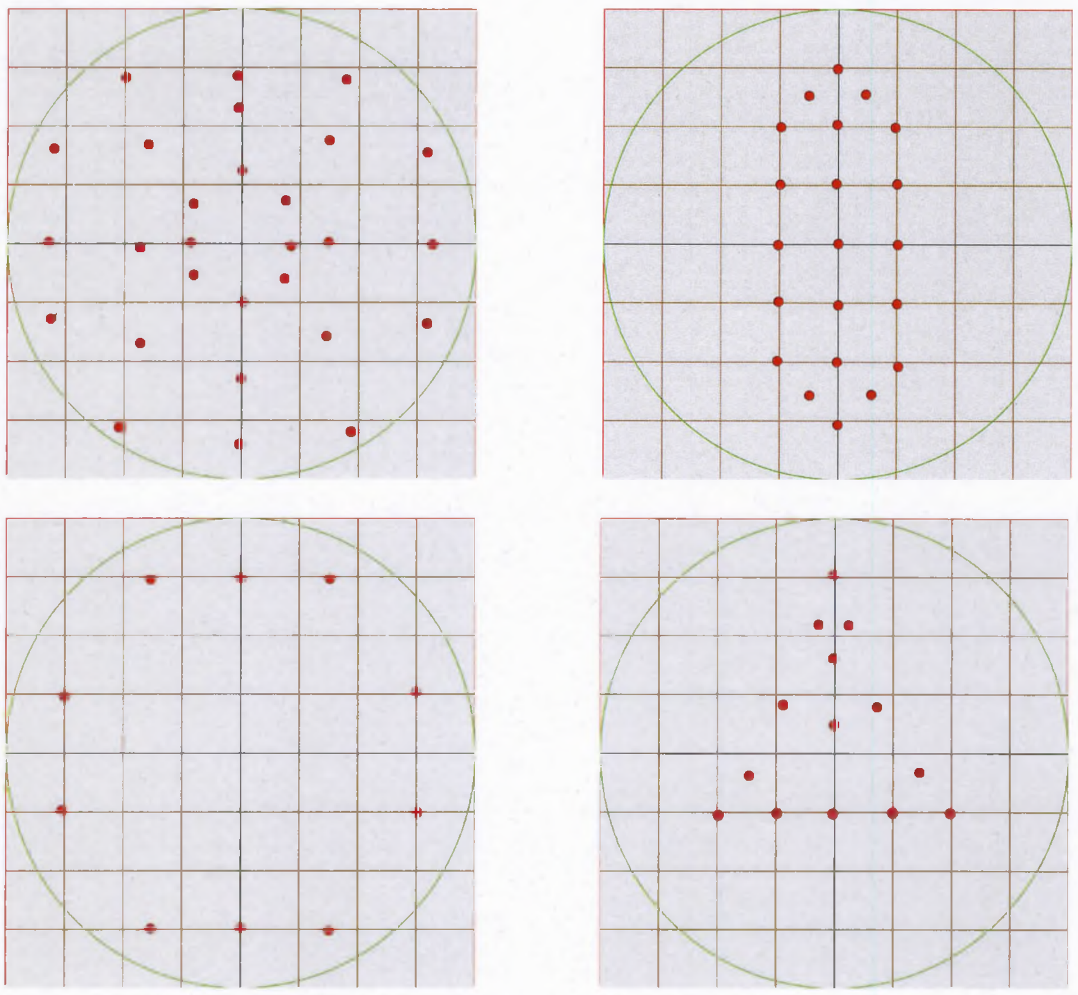


Figure 30: Various sampling patterns corresponding to different aperture shapes.

To extract a color value a passed sample counter ***P*** is initialized to one, an accumulation vector ***A*** is initialized with the value of sampling the source image at the pixel's center and a depth value ***D*** is initialized with the depth of the pixel being processed. Using a list of ***n*** texture sampling coordinates, ***n*** color and depth are samples are taken and, for each sample, if the magnitude of the difference in depth between the sample's depth and ***D*** is less than a threshold value ***Δ*** the sample's color value is added to the accumulator ***A*** and the passed sample counter ***P*** is incremented by one. After ***n*** samples are processed the final color value is calculated by dividing the value in the accumulator ***A*** by the passed sample counter ***P***.

Conditionally accepting only the samples with a depth difference of at most ***Δ*** guarantees that all the samples contributing to the color of a pixel are within an explicitly specified depth range around the pixel's depth, which helps avoid intensity leakage since background samples contributing to foreground samples or the other way around would be rejected due to the large depth difference. The number of sampling coordinates plus one becomes an upper bound to the number of source image pixels contributing to every processed pixel, with the lower bound being one contributing pixel (which can happen in case none of the sampled pixels using coordinates from the sampling coordinates list fails the depth condition, in which case the only contributor to a pixel's color value would be the pixel itself). Figure 31 shows a scene (left) and a visualization of the percentage of rejected samples due to depth (right) - black means zero pixels were rejected and white means all pixels were rejected with grey regions occurring linearly in between the two.

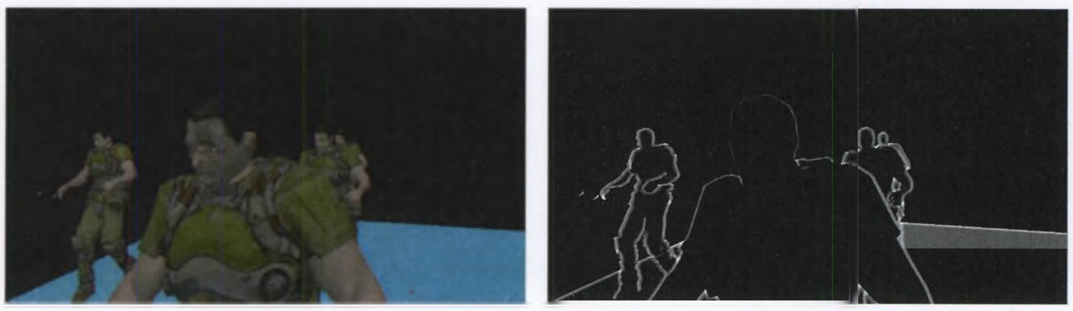


Figure 31: A scene (left) and a visualization of the percentage of rejected samples due to depth (right). Black means zero pixels were rejected and white means all pixels were rejected with levels of grey representing values in between the two extremes linearly.

To compute the final color of each pixel three colors $\boldsymbol{R}$, $\boldsymbol{G}$, $\boldsymbol{B}$ are computed using the color extraction approach using the lists $\boldsymbol{C_r}$, $\boldsymbol{C_g}$, $\boldsymbol{C_b}$ respectively. The final color is the vector consisting of the red component of $\boldsymbol{R}$, the green component of $\boldsymbol{G}$ and the blue component of $\boldsymbol{B}$. Chromatic aberration is a direct result of using three different sizes $\boldsymbol{S_r}$, $\boldsymbol{S_g}$, $\boldsymbol{S_b}$ to calculate three circle of confusion sizes. The case where $\boldsymbol{S_r} = \boldsymbol{S_g} = \boldsymbol{S_b}$ is the equivalent of an ideal lens with no chromatic aberration.

Lenses usually have a non uniform distribution of chromatic aberration intensity, with the edges having more aberration than the center (Figure 32). The scaling parameters $\boldsymbol{d_r}$, $\boldsymbol{d_g}$, $\boldsymbol{d_b}$ can themselves be scaled across the image using values fetched from a texture such as a vignette (figure 33) to vary the chromatic aberration intensity across the image.

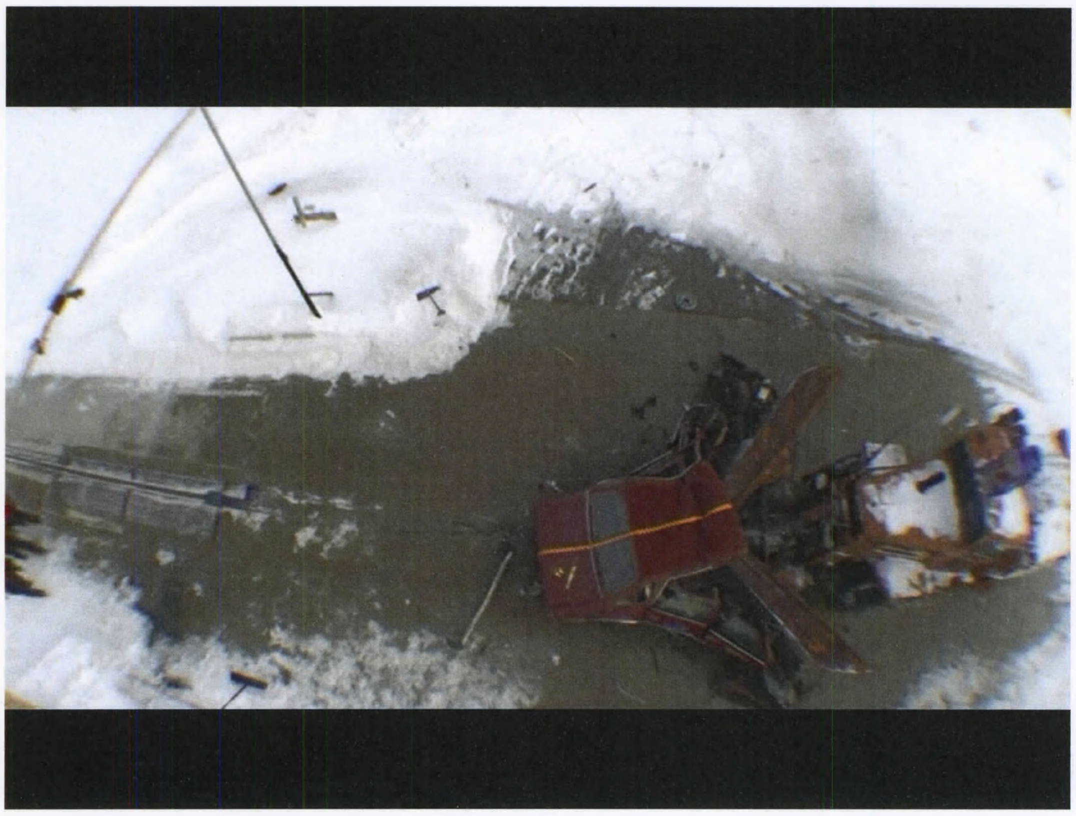


Figure 32: Scene from Mythbusters made using a wide-angle lens exhibiting chromatic aberration. The non uniform distribution of aberration intensity can be seen near the right and left edges which exhibit significant chromatic aberration as opposed to the center which is relatively aberration free.

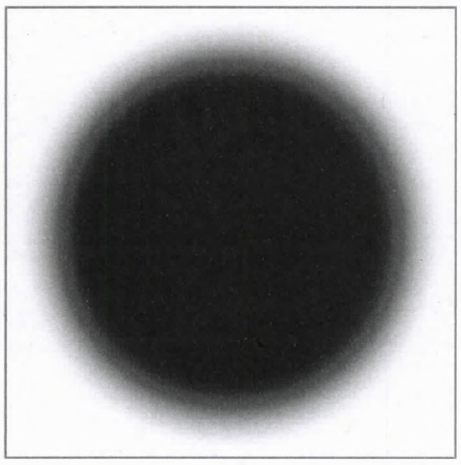


Figure 33: Vignette texture which can be used to scale $\mathbf{d}_r$, $\mathbf{d}_g$, $\mathbf{d}_b$ to get a chromatic aberration distribution similar to figure 32.

ALGORITHM

```
DepthOfField(SourceImage,
             DepthTexture,
             TextureCoordinate,
             C,
             $\Delta$):

    Depth ← sample(DepthTexture, TextureCoordinate)
    A ← sample(SourceImage, TextureCoordinate)
    P ← 1
    S ← ComputeCoCSize()
    $S_r, S_g, S_b$ ← $S * d_r, S * d_g, S * d_b$
    C ← C + TextureCoordinate
    $C_r, C_g, C_b$ ← $C * S_r, C * S_g, C * S_b$

    FetchColors(SampleSet):
        A ← sample(SourceImage, TextureCoordinate)
        P ← 1
        for i in SampleSet:
            D ← sample(DepthTexture, i)
            if abs(Depth - D) <= $\Delta$:
                Color ← sample(SourceImage, i)
                A ← A + Color
                P ← P + 1
        return (A / P)
    R ← FetchColors($C_r$)
    G ← FetchColors($C_g$)
    B ← FetchColors($C_b$)
    return (R.r, G.g, B.b)
```

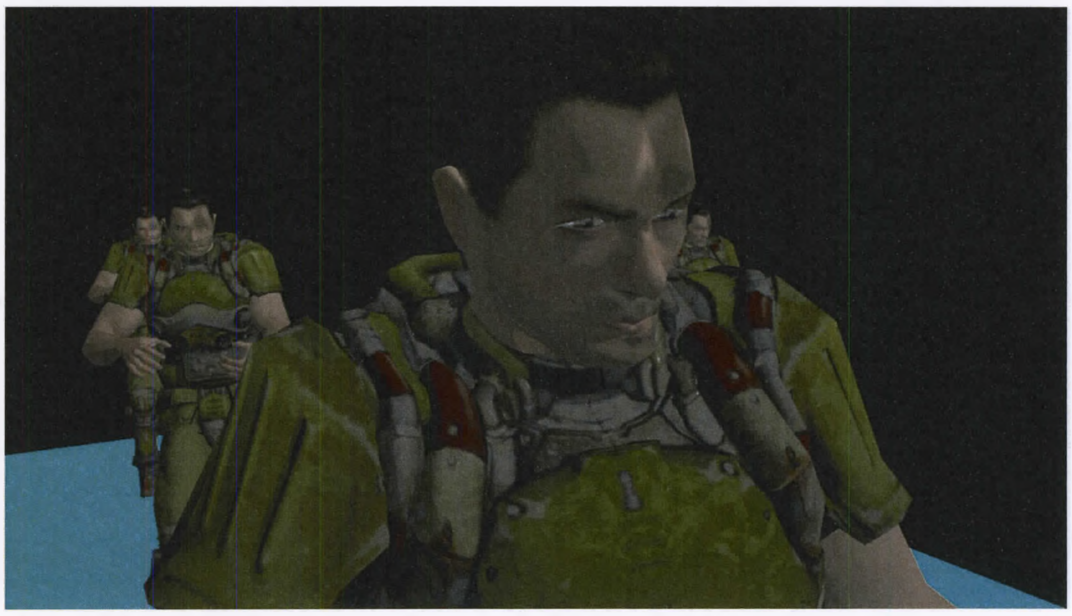


Figure 34: Scene rendered without Depth of Field.

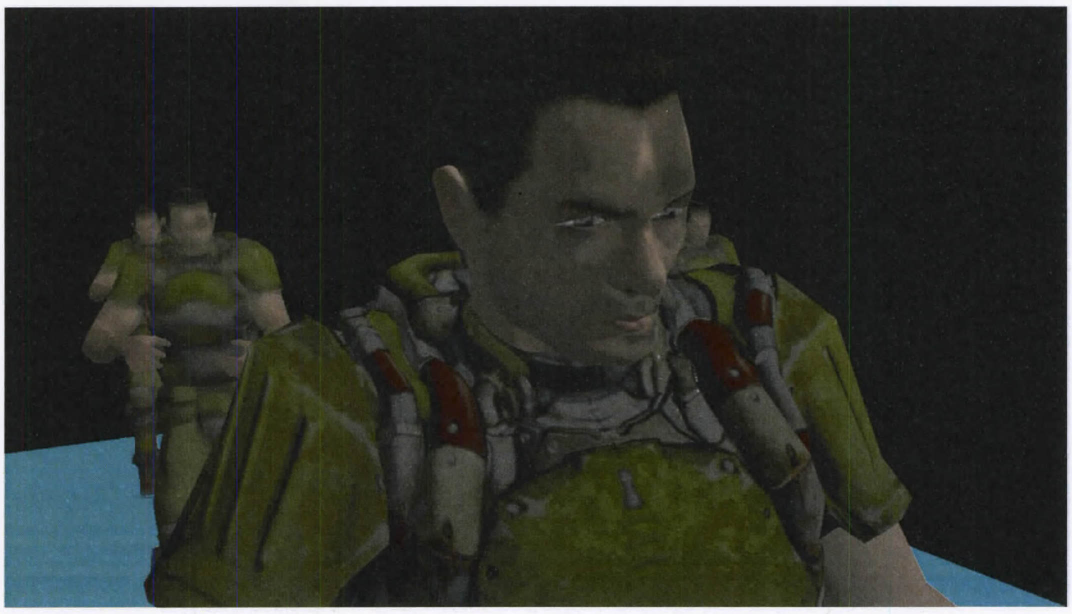


Figure 35: Same scene rendered with DoF filter enabled and the front object in focus - the background out of focus objects don't leak into the foreground in focus object.

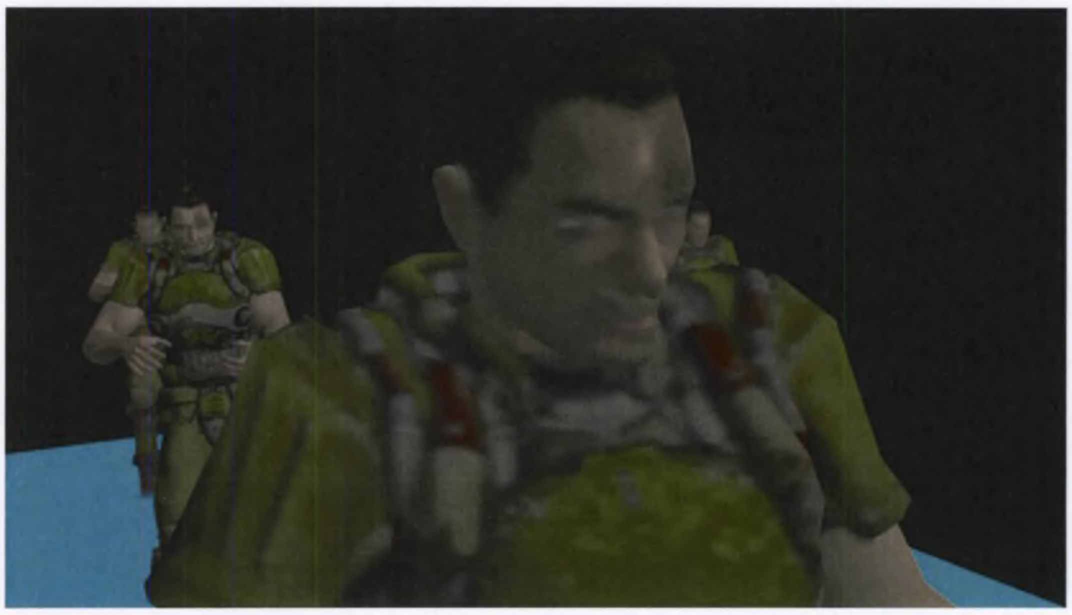


Figure 36: Same scene rendered with DoF filter enabled and the front object out of focus - the front out of focus objects don't leak into the background in focus objects.

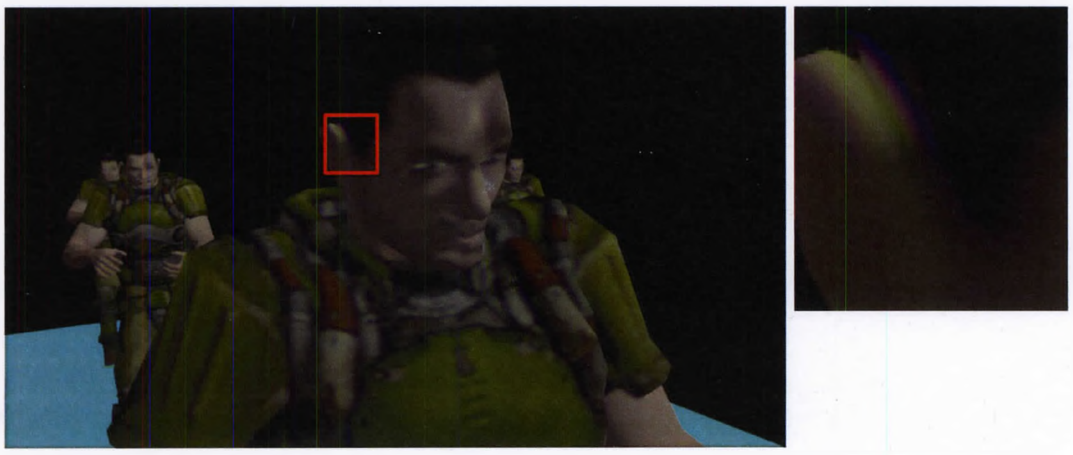


Figure 37: Same scene rendered with depth of field and chromatic aberration enabled ($S_r \neq S_g \neq S_b$). Chromatic aberration artifacts such as purple fringing can be scene in the highlighted region (left) and its closeup (right).

CHAPTER FOUR:
LENS FLARE

INTRODUCTION

Lens Flare is the name given to a set of unintended visual patterns caused by the diffraction and reflection of light as it passes through a camera's aperture and lens system. While the effect is considered a flaw in lenses it has become an identifying element of photographic and video footage, and attempts have been made to reproduce it in computer-generated footage for realism [Pixar 2008].

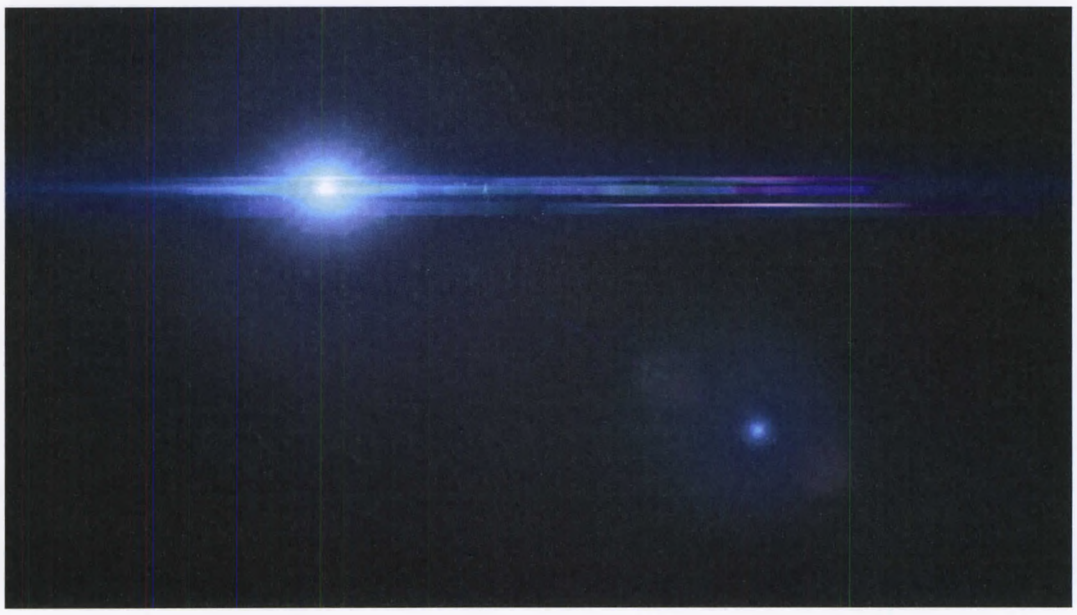


Figure 38: Anamorphic Lens Flare. Source: Wikimedia Commons

Lens flares exhibit characteristic variations based on the specific lens and camera formats used, and some patterns like the lens flare shown In Figure 38 are strongly associated with film productions due to their correlation with the anamorphic lenses (lenses used to shoot wide-screen content while stretching it to fill a 35mm film frame of a different aspect ratio) and film format used. Lens Flare rendering adds a strong camera feel to a scene, and attempts to reproduce it in real-time and offline renderings have been going on for over a decade.

Previous Work

Previous attempts at rendering real-time lens flare have mostly relied on having explicit knowledge of the screen-space position and visibility of a point light source. [Kilgard 2000] relied on pre-made flare texture sprites additively blended to the framebuffer. [King 2000] used a technique similar to [Kilgard 2000] while varying flare texture sprite position and intensity depending on the screen-space position of the light source. [Maughan 2001] introduced a method to vary the lens flare intensity based on the visibility of the light source. [Kawase 2003] used downsized framebuffer post-processing on programmable hardware to generate image-based light streaks and simple lens ghosting that was limited to being a vignette-masked screen-space reflection of the framebuffer contents. [Hullin et. al. 2011] used GPU-accelerated raytracing coupled with geometry generation and pre-computed textures to render a very accurate approximation of lens flare for a given point light, albeit at a very high cost, making the algorithm unsuitable for use on consumer hardware in scenes with area lights or many point lights.

The algorithm presented here is an image-based approach based on [Kawase 2003]. The runtime is independent of the number of light sources in the scene, enabling it to simulate lens flare for any number of point and area lights in a scene, as well as handling secondary light reflections such as the specular high-lights on shiny objects in the scene. The algorithm can also vary the shape of ghosts and light streaks based on a definition of the desired aperture's shape.

Breakdown

Lens flare rendering is divided into three parts: Bloom (halos around bright objects), Light Streaks (aperture-based diffractions) and Ghosting (internal reflections in the lens).

Bright Region Detection

As a pre-processing step a bright-region texture is created by passing the rendered scene through a filter that extracts the bright regions of the scene while blacking out the rest of the image area. The filter is applied after the tone-mapping step since exposure and other tone-mapping parameters affect the brightness of the scene. The output needs to be clamped to the range [0.0, ∞[to avoid having negative color intensities, a concept that has no real meaning.

Detecting bright regions can be done using a variety of methods. The easiest approach is a linear subtraction (equation 3) where the resulting color vector ***y*** is the result of subtracting a color threshold vector ***t*** from the input color ***x***; but the approach used here was based on evaluating the pixel's luminance and passing only pixels with luminance exceeding a threshold value ***t*** (equation 4).

$$f(x) = x - t \qquad (3)$$

$$f(x) = \begin{cases} 0 & \text{if luminance(x)} < \text{t} \\ x & \text{otherwise} \end{cases} \qquad (4)$$

Figure 39 shows a scene rendered into a high dynamic range framebuffer (left) and the result of a bright-region detection pass using the luminance based approach from equation (4) (right). The bright-region texture is used as input to the rendering passes responsible for generating the various components of the lens flare effect.

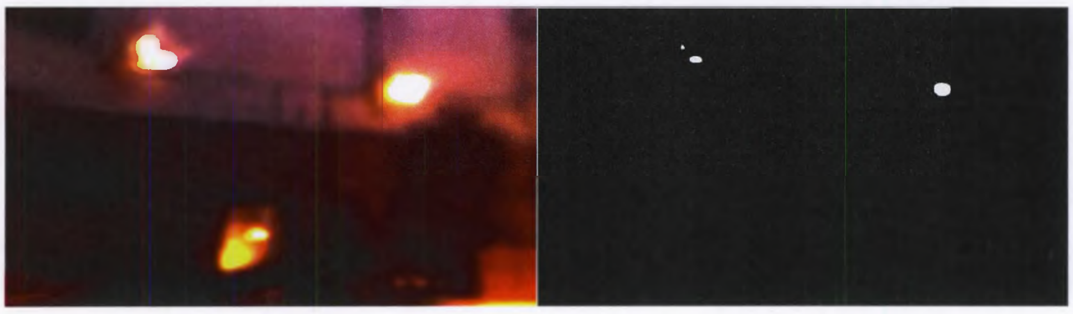


Figure 39: Scene rendered into high dynamic range framebuffer (left) and its corresponding bright-region filter output (Note that the white spots have an RGB ratio that's not necessarily 1:1:1, they merely appear so due to the limitations of the display).

Bloom

Bloom is an effect resulting from bright over-saturated points leaking onto neighboring pixels on an image sensor, with the result looking like halos around bright objects. The effect can be achieved by applying a Gaussian blur [Oat 2003] or Kawase's Bloom Filter [Kawase 2003] to the bright-pass output texture and additively blending the result with the framebuffer.

Figure 40 shows the bloom rendering process from [James 2004] - a source texture (a) is passed through a high-pass filter to isolate the bright regions, with the output stored in a texture (b). The bright regions texture then has a separable Gaussian blur [Oat 2003] applied to it first horizontally then vertically to get the texture (c). A final post-processing pass additively blends the texture from (c) onto the source texture from (a) resulting in the final image with bloom (d).

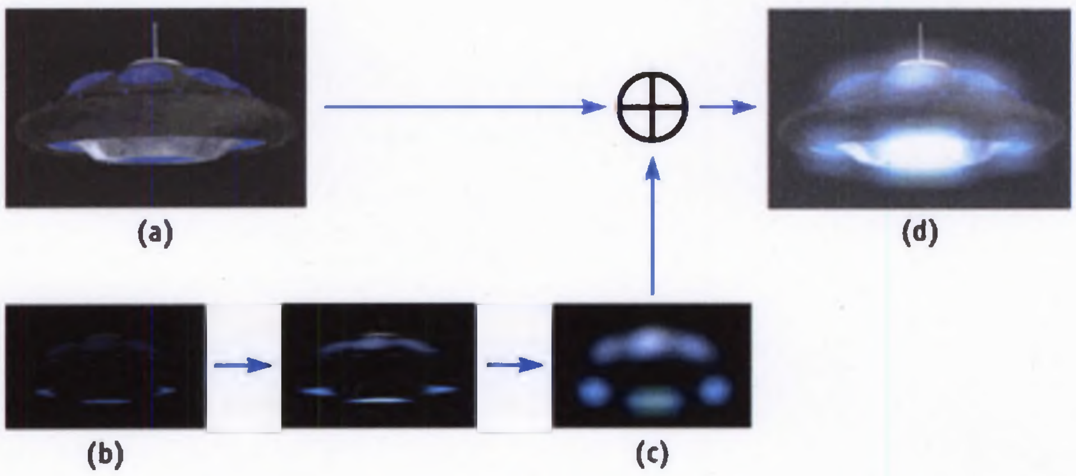


Figure 40: Bloom rendering pipeline. Source texture (a) is passed through a high-pass filter to isolate the bright regions (b) followed by a 2-stage Gaussian blur (c) and then the result is composited additively on top of the source texture resulting in the final image with the bloom effect (d). Image from [James 2004].

Light Streaks

Light streaks are a result of light diffraction around the edges of a polygonally-shaped aperture resulting in star-like shapes originating at bright points in the image. Figure 41 shows an example of light-streaks in video footage from the show *Mythbusters*. The streaks originate at bright points in the scene and spread out in six different directions, suggesting that the camera used had a hexagon-shaper aperture.

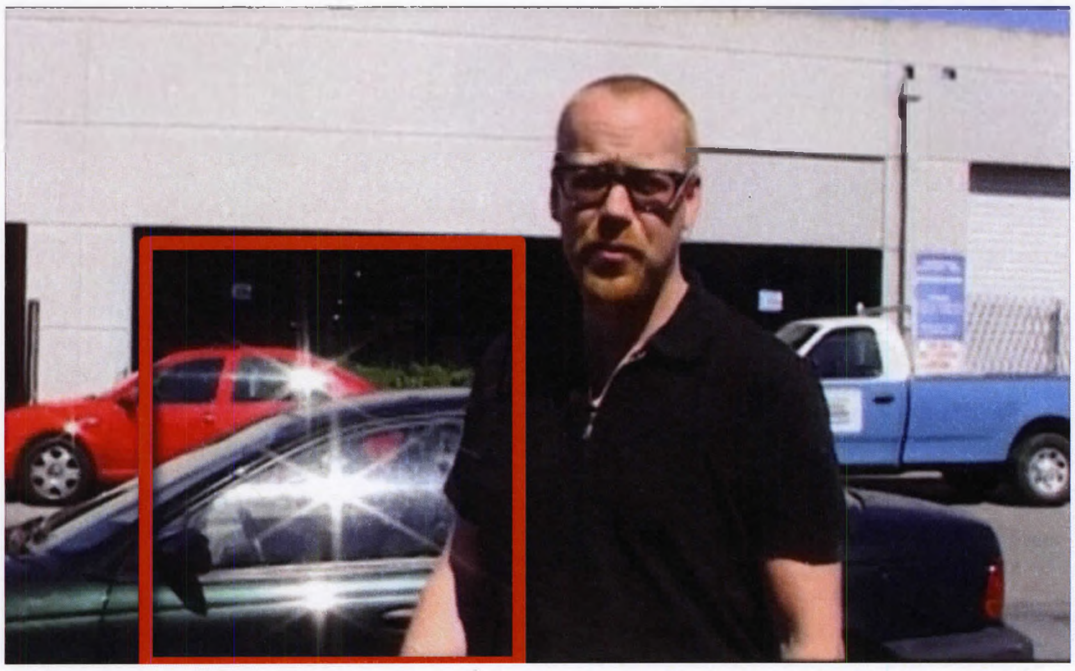


Figure 41: Light Streaks (outlined) captured by a video camera from the TV show Mythbusters.

[Kawase 2003] proposed an algorithm to render light streaks that starts by generating a high-pass texture containing the bright regions in the scene, scaling it to one quarter of its original size followed by applying a directional blur two or more times to end up with a texture containing a single streak (Figure 42) that is then scaled back to the original source size and composited additively on top of the source image (figure 43).

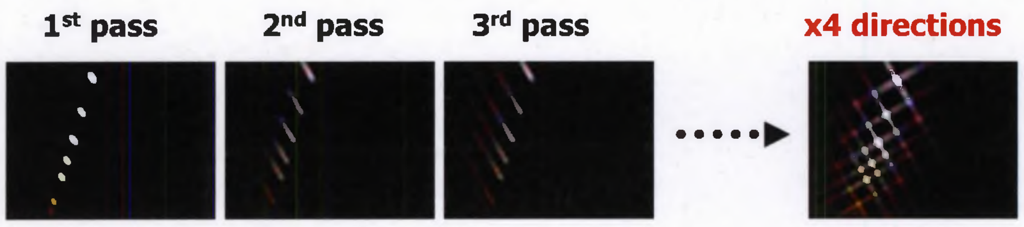


Figure 42: A three-pass directional blur applied to the bright regions in a scene and the resulting streaks image with four-directional streaks from applying four directional blur passes. Image from [Kawase 2003].

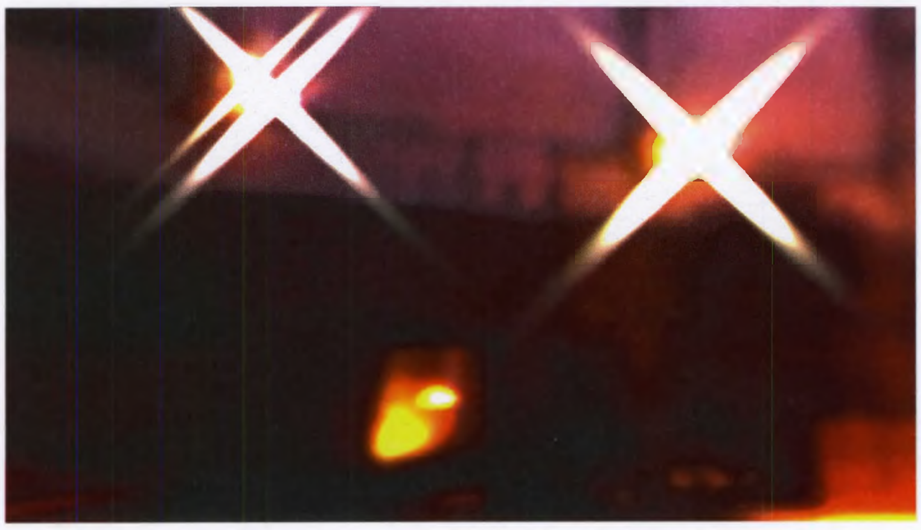


Figure 43: Light Streaks rendered using [Kawase 2003].

Ghosting

Ghosts is the name given to the visual artifact that manifests as aperture-shaped polygonal reflections of bright regions in the scene being photographed. Figure 44 shows an example of lens ghosts (highlighted) in a photograph.

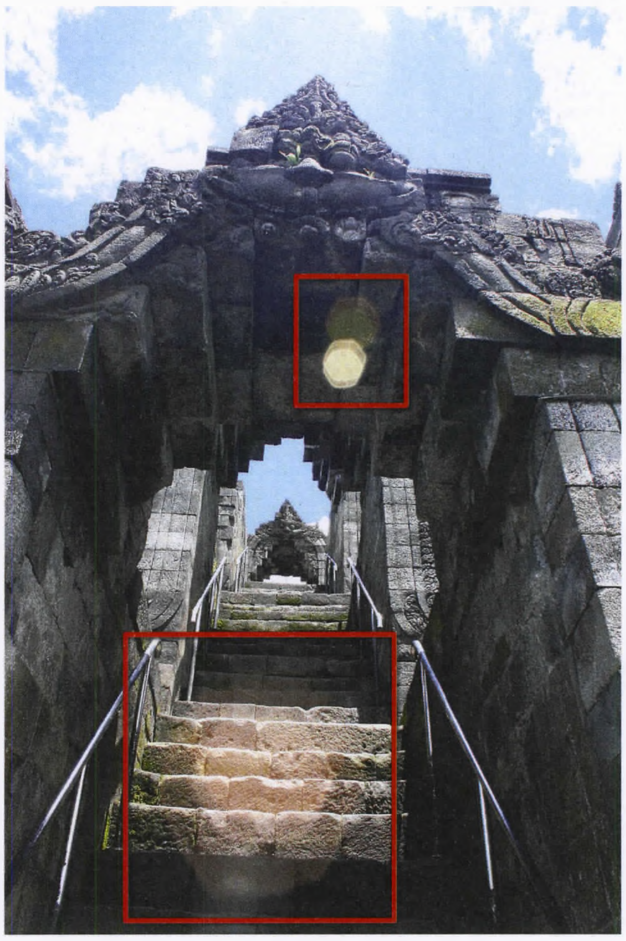


Figure 44: Polygonal, aperture-shaped ghosts in a photograph. Source: Wikimedia Commons

Lens ghosting is heavily present in film. Figure 45 shows a scene from the 2012 movie *The Dark Knight Rises* (left) with a bright light-bar on top of a police car in an otherwise dark scene and a close-up of a highlighted region in the scene (right) showing multiple ghosts.

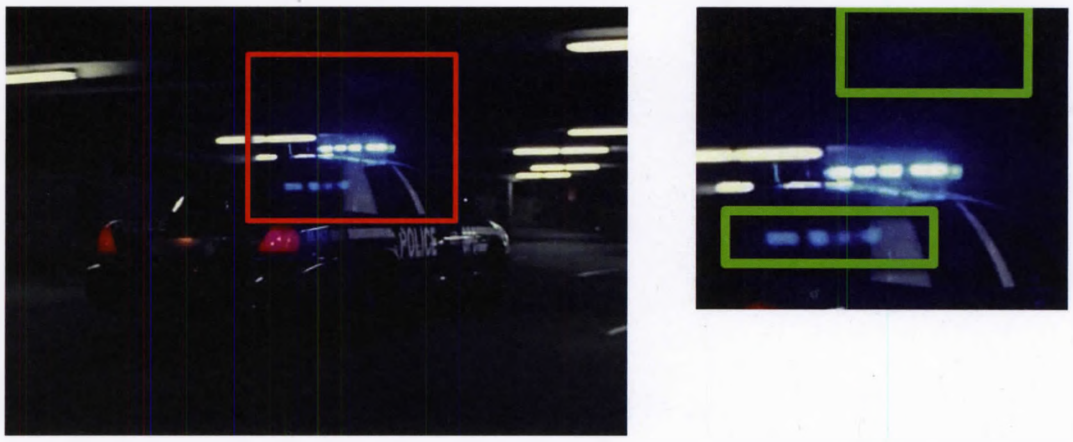


Figure 45: Scene from the 2012 movie The Dark Knight Rises (left) showing lens ghosts in the red highlighted region and a close up of the region (right) with the lens ghosts highlighted in green.

Ghosts can emphasize some color components more than others. Figure 46 shows a scene from The Dark Knight Rises with a highlighted bright light source and its associated ghost inside a red outline (left), and a close-up of the highlighted region (right) with the ghost (outlined in blue) and its source (outlined in green) where the purple tint of the ghost, resulting from some light wavelengths being emphasized more than others.

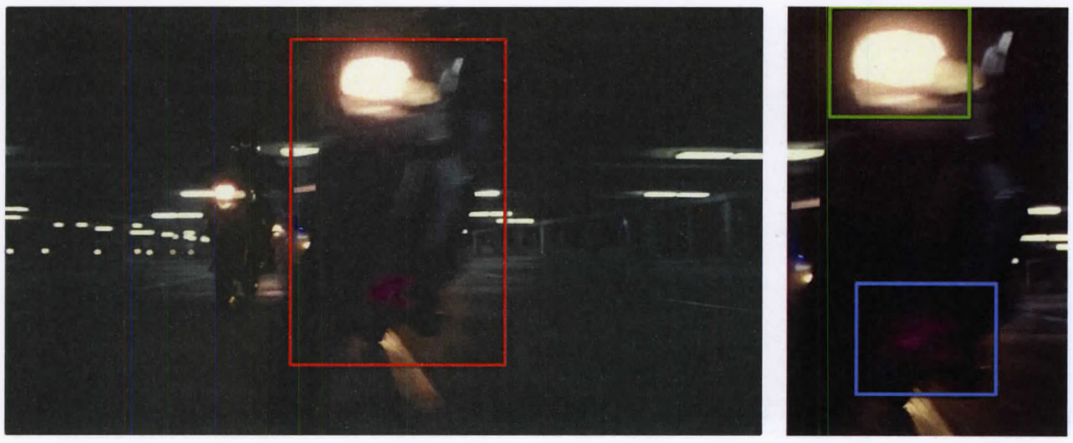


Figure 46: Scene from The Dark Knight with a highlighted region (left) and a close-up of the highlighted region (right) showing a ghost (blue highlight) and its source (green highlight).

PROPOSED TECHNIQUE

[Kawase 2003] proposed using a blurred version of the scene's bright-pass output reflected and scaled around the center of the screen to simulate basic lens ghosting. A problem with this method, however, is that ghosts do not accurately account for the diffraction due to the aperture shape and they maintain a strong resemblance to the original bright-pass output unlike real ghosts which usually have polygonal, aperture-based shapes. The proposed technique also recreates the apparent rotation of aperture shaped ghosts around a vector perpendicular to the resulting two-dimensional image's surface, color emphasis and subtle color shifts due to different wavelengths of light.

The proposed algorithm takes as input a set of $\boldsymbol{n}$ two-dimensional points $\boldsymbol{C} = \{ \boldsymbol{C_1}, \ldots, \boldsymbol{C_n} \}$ distributed within a unit circle representing the aperture diffraction points (the vertices of a hexagonal aperture for example) centered around the unit circle (figure 47).

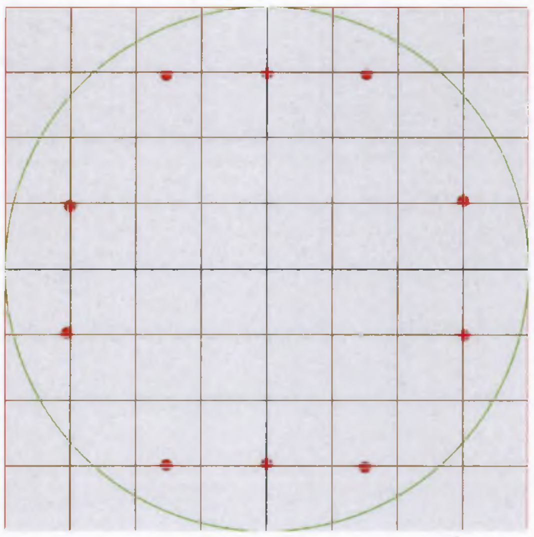


Figure 47: Sample aperture diffraction points (in red) for an octagonal aperture.

To generate a ghost, a post-process pass is executed where at each pixel the set of points in ***C*** is centered around the pixel's texture coordinates and reflected around the center of the image by an amount ***R*** using equation 5.

$$reflected(\hat{x}) = ((\hat{x} - 0.5) \times R) + 0.5 \qquad (5)$$

An expansion scale ***E*** controlling the resulting size and dispersion of the ghosts is determined based on the sampling points and desired ghost shapes. Three offsets O_r, O_g, O_b are used to represent the difference in diffraction strengths with respect to the diffraction of white light for red, green and blue respectively. Three expansion scales E_r, E_g, E_b are generated by multiplying ***E*** with O_r, O_g and O_b respectively. If $O_r = O_g = O_b$ then the three color components will be treated as having the same diffraction properties and no color shifting will exist in the generated ghosts, otherwise there will be a varying color shift for each color component.

To simulate the rotation of ghosts around the vector perpendicular to the surface of the image by an angle $\boldsymbol{\Omega}$ the points in $\boldsymbol{C}$ are transformed using a standard two dimensional rotation matrix with the angle $\boldsymbol{\Omega}$ before being passed as input to the fragment program. The points in $\boldsymbol{C}$ are then scaled by $\boldsymbol{E_r}$, $\boldsymbol{E_g}$ and $\boldsymbol{E_b}$ to yield the sets $\boldsymbol{C_r}$, $\boldsymbol{C_g}$ and $\boldsymbol{C_b}$.

Three color vectors $\boldsymbol{R}$, $\boldsymbol{G}$, $\boldsymbol{B}$ are generated by taking $\boldsymbol{n}$ samples from the bright-region texture - after it has been softened using a blur pass based on the bloom filter from [Kawase 2003] - using the sampling coordinates from $\boldsymbol{C_r}$, $\boldsymbol{C_b}$, $\boldsymbol{C_g}$ respectively and then averaging the color of the $\boldsymbol{n}$ samples. The final ghost image is generated by taking the red component of $\boldsymbol{R}$, the green component of $\boldsymbol{G}$ and the blue component of $\boldsymbol{B}$ to retrieve the final color $\boldsymbol{F}$. A color tint can be applied at this point by multiplying $\boldsymbol{F}$ by a three component tint vector $\boldsymbol{T}$ containing the relative weights for the three color components, so for example a ghost with a strong red tint, moderate green tint and weak blue tint would have $\boldsymbol{T} \approx \{ 1.0, 0.5, 0.2 \}$.

ALGORITHM

```
GenerateGhost(BrightRegionTexture, TextureCoordinate, R, C, E):
    C ← C + TextureCoordinate
    $E_r, E_g, E_b$ ← E * $O_r$, E * $O_g$, E * $O_b$
    $C_r, C_g, C_b$ ← C * $E_r$, C * $E_g$, C * $E_b$
    FetchColors(SampleSet):
        Color ← 0
        Count ← 0
        for i in SampleSet:
            Color ← Color + sample(BrightRegionTexture, i)
            Count ← Count + 1
        return (Color / Count)
    R ← FetchColors($C_r$)
    G ← FetchColors($C_g$)
    B ← FetchColors($C_b$)
    return (R.r, G.g, B.b)
```

Results

A test scene with two bright regions was used for testing the technique, and the resulting ghost(s) are outlined. Figure 48 shows a single ghost generated using a rectangular aperture diffraction pattern. Figure 49 shows a single ghost generated using the same rectangular pattern and a tint vector T that emphasizes the purple color. Figure 50 shows the same ghost rendered using $Er < Eg < Eb$, with the resulting ghost exhibiting non-uniform color component diffraction. Figure 51 shows two rectangular ghosts generated using a different Ω value for each. Figure 52 shows a variety of ghosts rendered using different T, Er, Eg, Eb and Ω values.

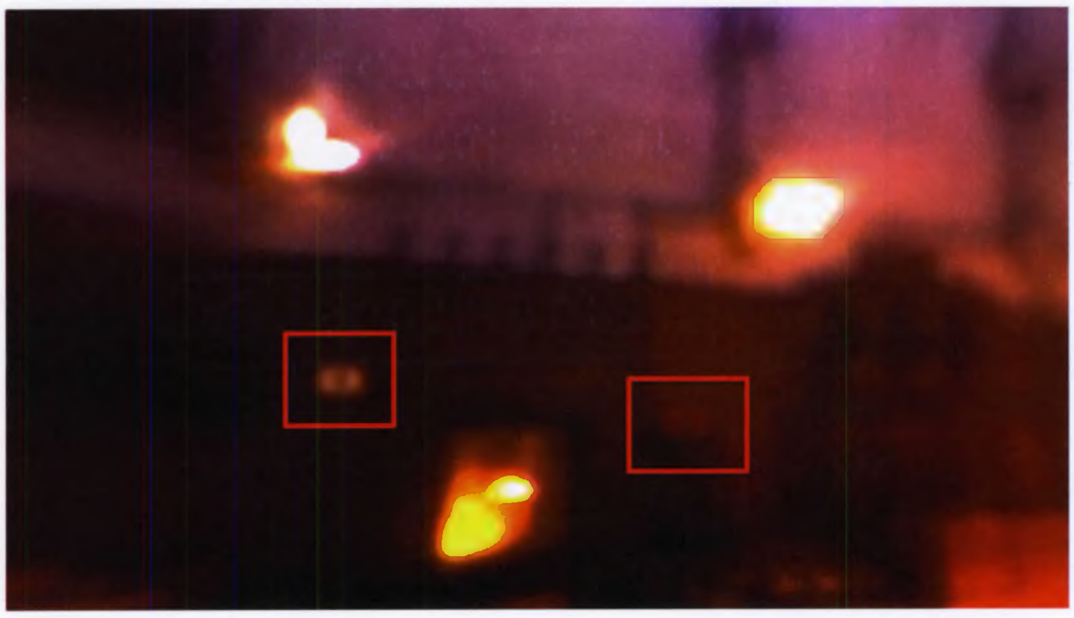


Figure 48: A single rectangular ghost of the two bright regions in the scene.

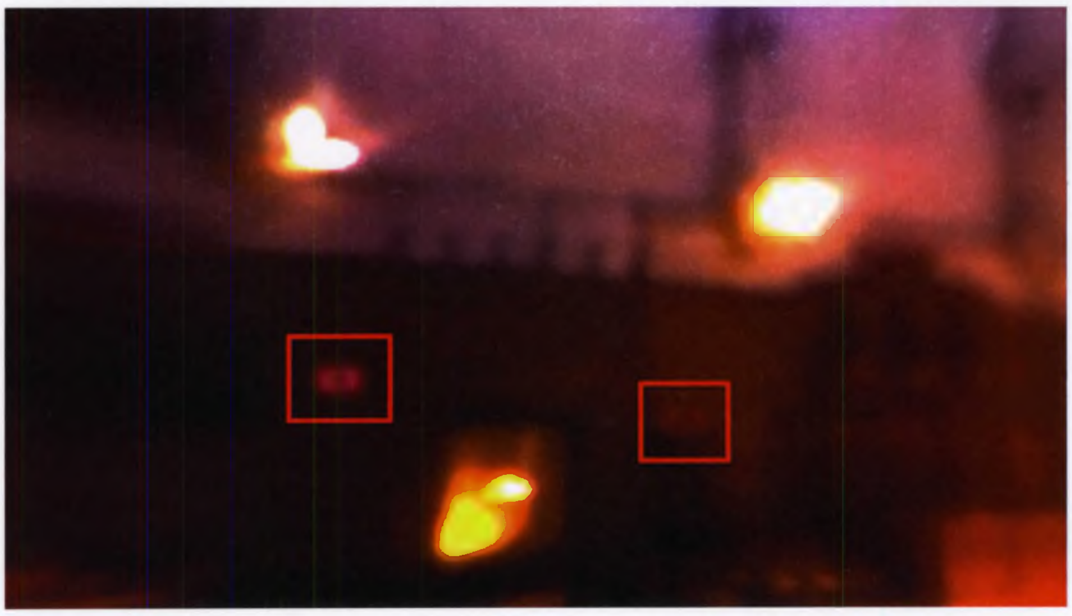


Figure 49: A single rectangular ghost rendered using a tint vector **T** emphasizing purple.

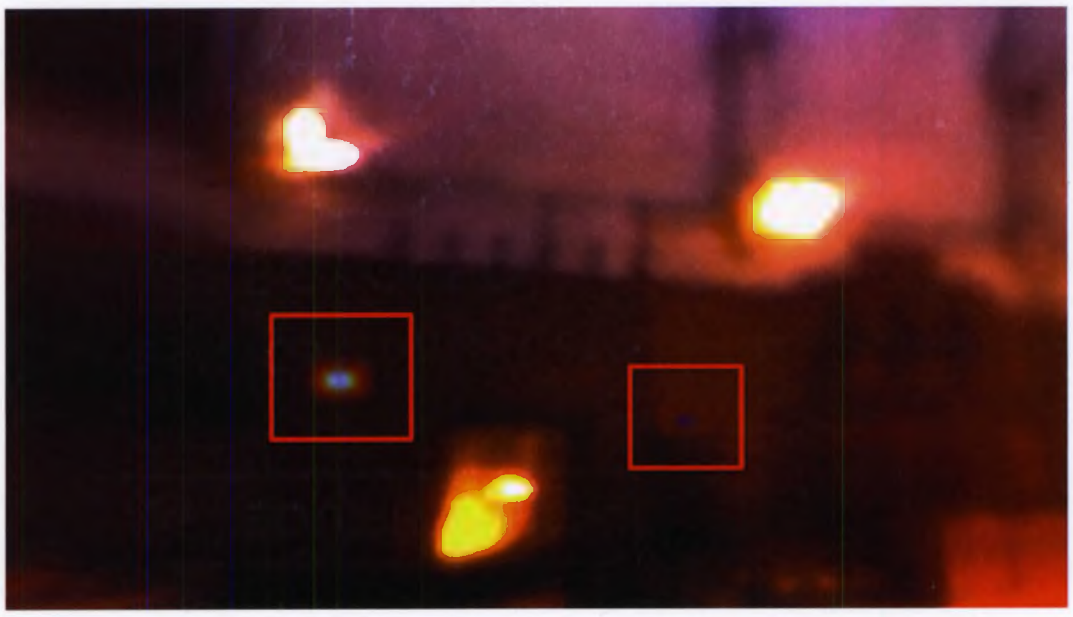


Figure 50: A single rectangular ghost rendered with $\mathbf{E}_r > \mathbf{E}_g > \mathbf{E}_b$. The distribution of the ghost's color components over the ghosts area approximates that of real lenses with significant non-uniform diffraction.

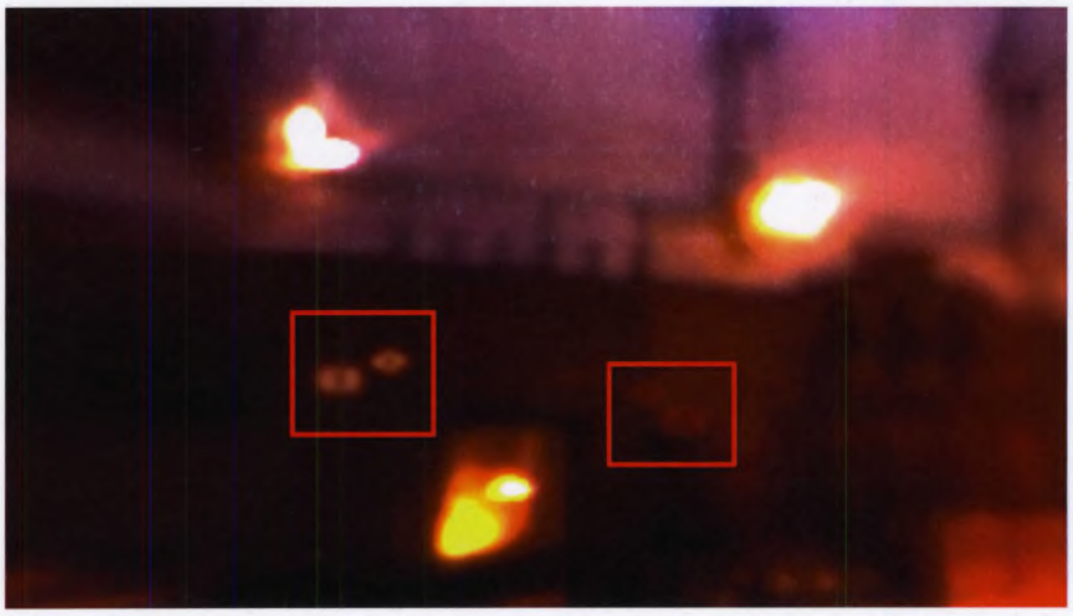


Figure 51: Two rectangular ghosts with different rotation angles ($\mathbf{\Omega}$).

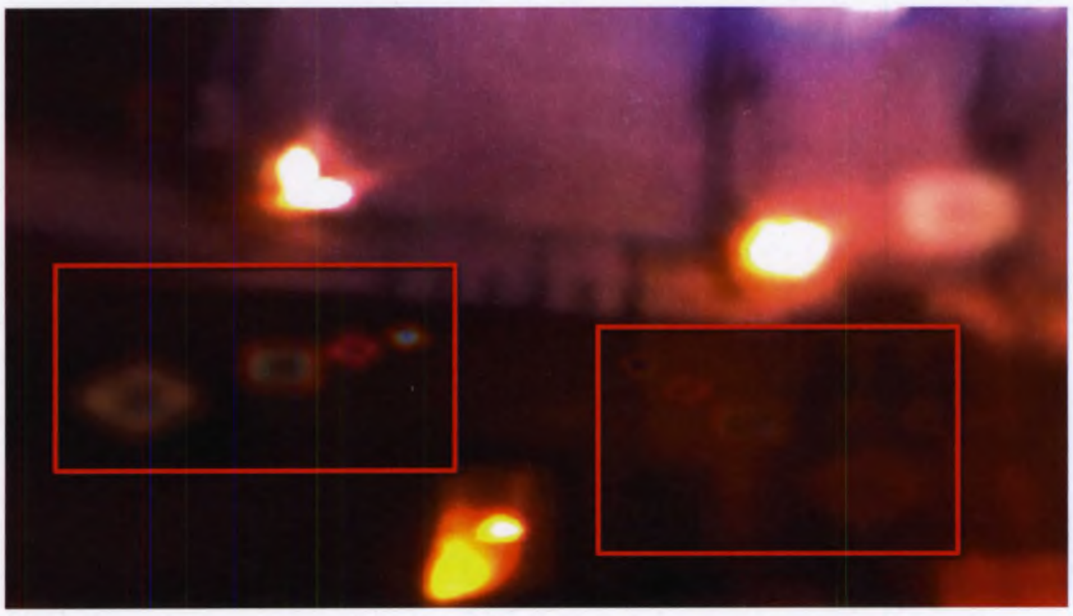


Figure 52: A variety of ghosts rendered using various $\mathbf{E_r}$, $\mathbf{E_g}$, $\mathbf{E_b}$, $\mathbf{T}$ and $\mathbf{\Omega}$ parameters.

CHAPTER FIVE:

TONEMAPPING

INTRODUCTION

While 24-bit monitors can display RGB colors whose components are in the range [0.0, 1.0] colors in reality have a much larger intensity range, and lighting calculations involved in the generation of a computer-generated image can lead to an intensity range that exceeds the displayable [0.0, 1.0] range. When an image is rendered to a traditional 8-bit per color component framebuffer the graphics hardware clamps colors outside the displayable range into the displayable range, but framebuffers using 16-bit or 32-bit floating point color components retain the high intensity values.

Image formats have been created to accommodate high-precision color images. OpenEXR, an image format developed by Industrial Light and Magic, is capable of storing color with 16-bit and 32-bit floating point components. Displaying high-precision formats directly on a traditional display would result in the high intensity color information being lost with the end result no better than traditional low precision image formats.

Tonemapping is the name given to the process used to map colors from one range (usually being the high-precision color range or a large subset of the range [0.0, Infinity[) to another range (usually being the displayable range of [0.0, 1.0]) while preserving as much visual information as possible. Tonemapping can be used to map a large range of color intensities to a smaller displayable range or merely map one displayable range to another for aesthetic reason. Tonemapping operators can be divided into two kinds: local (spatially varying) and local (spatially uniform).

Local tonemapping operators adjust the color of each region based on the intensities of the surrounding regions, an approach that works in practice because the human eye is much more sensitive to local contrast than it is to global contrast. Figure 53 shows a high dynamic range image tonemapped using a local tonemapping operator to fit into the displayable range on traditional computer displays. The overall aesthetic look of local tonemapping operators is very unique, and the aesthetic look of each intensity region (sky, billboards, square) is different due to the tonemapping operator considering each region's intensity locally independently of the overall intensity of the image.

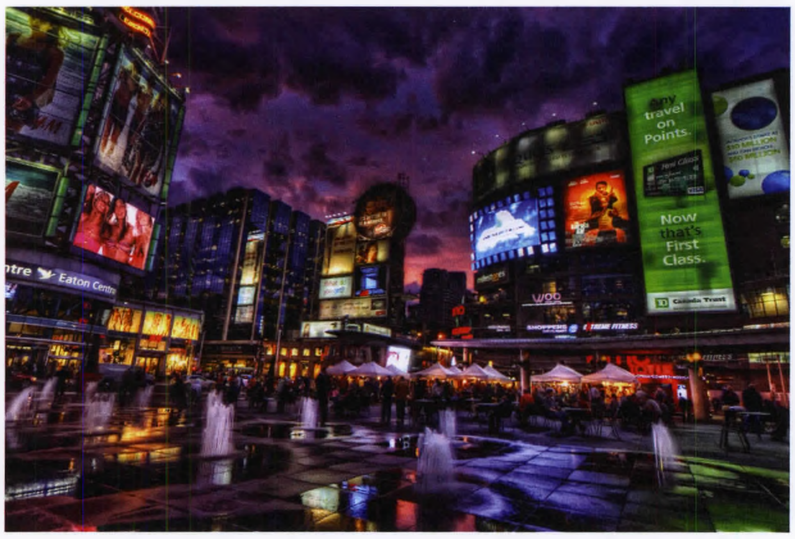


Figure 53: High dynamic range image using local tone mapping operator. The image is a result of combining three shots with different exposure levels and applying a tone-mapping operator to get all the colors in the displayable range, preserving detail in bright and dark regions. Source: Wikimedia Commons

Local tonemapping operators can suffer from artifacts at the edges where bright regions meet dark regions. The edge area will be tonemapped relative to the average intensity of its local surroundings which along that edge will be an average of the dark and bright region intensities, resulting in halo-like artifacts around that area. Figure 54 shows an example of the halo artifacts in a high dynamic range image with a local tonemapping operator applied.

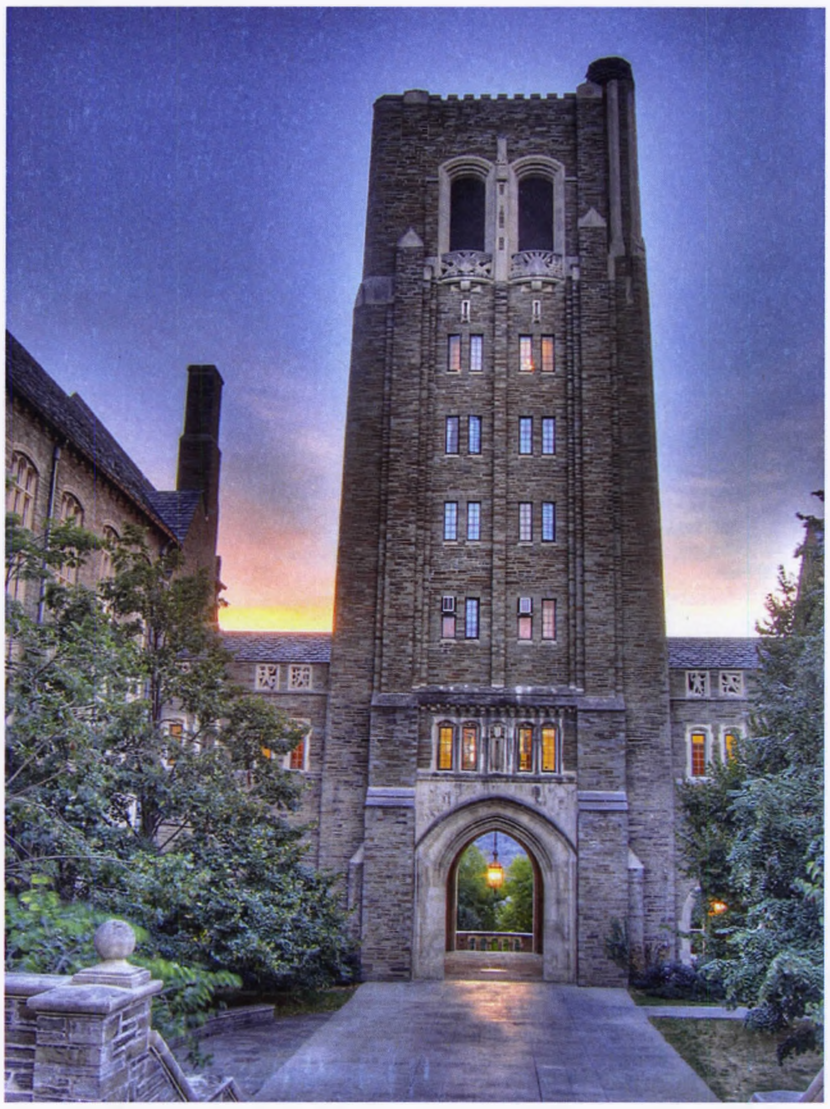


Figure 54: A high dynamic range image tonemapped using a local tonemapping operator exhibiting halo artifacts. The region where the bright background meets the relatively darker foreground building has an average local luminance that is brighter than the foreground luminance and darker than the background luminance, which affects the tonemapping operator. The result it the bright region being locally brighter and the dar region being locally darker, giving the foreground object the halo effect.

Global tonemapping operators adjust the color of the pixels based on image-wide attributes like overall exposure or a color histogram and as a result don't suffer from localized artifacts like local tonemapping operators, although the need to process the entire image to extract the attributes needed for applying the tone-mapping operator could be more costly than sampling local regions.

Figure 55 shows six photographs of the same scene with different exposures and a high dynamic range image generated from the six exposures and tonemapped using a global tonemapping operator.

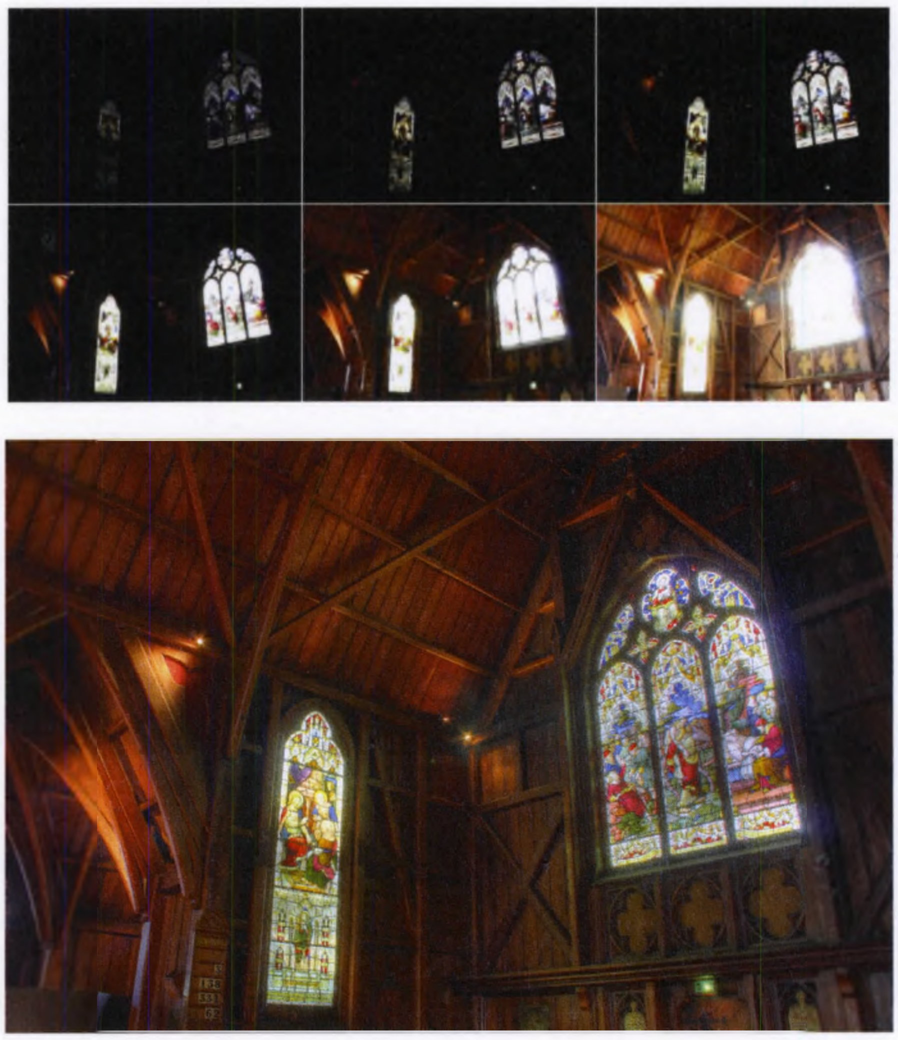


Figure 55: Six different exposures of the same scene (top) and a high dynamic range image (bottom) generated from the six exposures and tonemapped using a global tonemapping operator. Detail from the six source images is preserved in both bright and dark regions. Source: Wikimedia Commons

Another important aspect defining the look of a scene is the film transfer function - film material doesn't respond to colors in a linear way, and film material from different vendors has a different and unique response curve.The film response curve is commonly referred to as the Characteristic Curve, an S-curve with a toe, linear and shoulder regions. Figure 56 shows the characteristic curve for Kodachrome film from [Kodak 1998] with the toe, linear and shoulder regions labeled.

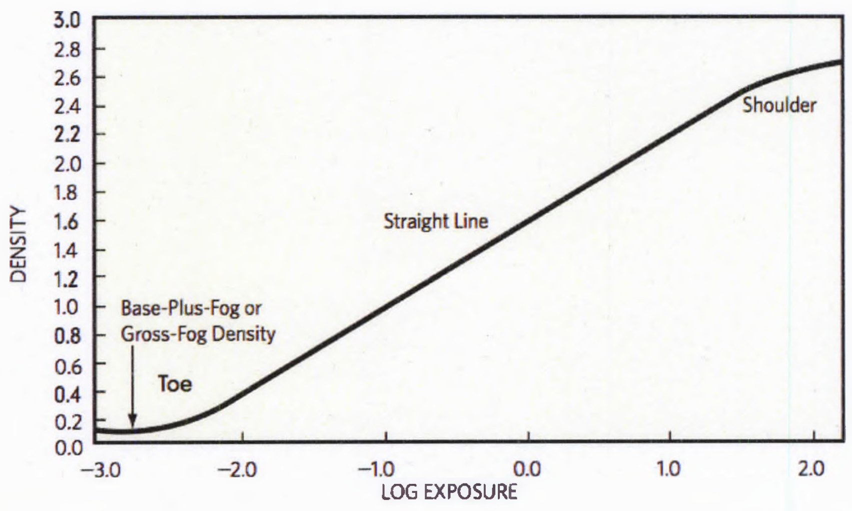


Figure 56: Kodachrome film characteristic curve from [Kodak 1998].

Film characteristic curves are one of the reasons people perceive consumer camcorder footage as "Soap Opera-like" as opposed to the usual, less linear and high contrast distinctive film footage look.

Humans don't perceive color intensities linearly, and a linear scaling of color components to bring them from the range [0.0, Infinity] to the displayable [0.0, 1.0] range results in a significant loss of information. As a result, various tone-mapping operators have been developed in an attempt to display high dynamic range images on traditional low dynamic range displays while preserving as much detail as possible.

[Reinhard et al. 2002], which targeted existing photographic content, provided a number of global tonemapping operators of various complexity aimed at mapping high dynamic range color into the displayable [0.0, 1.0] range. For comparison purposes an average complexity operator (equation 6) from [Reinhard et al. 2002] will be used.

$$L_d(x,y) = \frac{L(x,y)(1+\frac{L(x,y)}{L_{white}^2})}{1+L(x,y)} \qquad (6)$$

Reinhard's operator, like other tonemapping operators aimed at tonemapping photographic content, was designed for use on images that have already been affected by the film characteristic curve and as such its curve (figure 57) is not s-curve like. Computer-generated imagery uses a linear color space with the value of a color component corresponding to the linear intensity or brightness of the component and as a result when Reinhard's operator is applied to computer-generated imagery the result is a flat-looking image lacking the characteristic lights and darks of photographic images that are a result of the s-curve film transfer function.

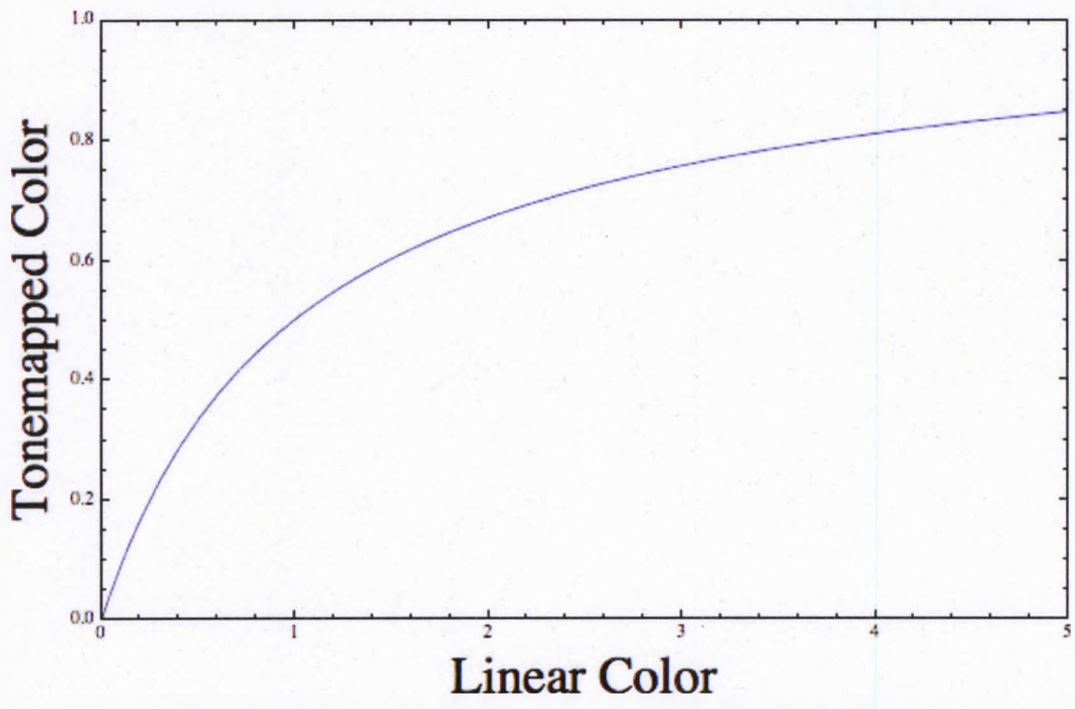


Figure 57: Tonemapping curve from [Reinhard et al. 2002].

For film production vendors provide LookUp Tables (LUTs) for converting linear color from computer generated imagery (most commonly used for visual effects) to a non-linear color space matching their film products' characteristic curves. Look up tables supporting a large range of color values, a necessity for high dynamic range content, are too large to be managed in real-time on current consumer hardware.

When dealing with low dynamic range images [Reinhard et al. 2002]'s shortcomings can be overcome by converting the linear color values into non-linear, film-like space using a texture as a lookup table for color correction [Selan 2005]. The lookup table texture is used as a map from every possible low dynamic range RGB triplet to a corrected non-linear RGB triplet. A cubemap texure with sides that are 256x256 pixels is a perfect match for such use case.

[Selan 2005] talked briefly about extending lookup tables to support high dynamic range content and suggested using non-uniform lattice sampling to concentrate the limited precision of a GPU supported lookup table in the regions where the human visual system is most sensitive. [Hable 2010] attempted a different approach, getting rid of lookup tables in favor of mathematical functions that mimic the visual properties of film.

[Hable 2010]'s tonemapping operator defines a curve mapping function (equation 7) with fitting parameters (***a, b, c, d, e, f***) that is used by the tonemapping operator (equation 8) which applies global tonemapping based on a white-point ***w***.

$$h(x) = \frac{x(ax + bc) + de}{x(ax + b) + df} - \frac{e}{f} \qquad (7)$$

$$H(x) = \frac{h(2x)}{h(w)} \qquad (8)$$

The graph of Hable's tonemapping operator (figure 58) is similar to Reinhard's in its mid section but the low and high color ranges are mapped differently. Hable's operator has a toe-like section at the low color range (Figure 59) which avoids the desaturation of low intensity colors seen with Reinhard's tonemapping operator.

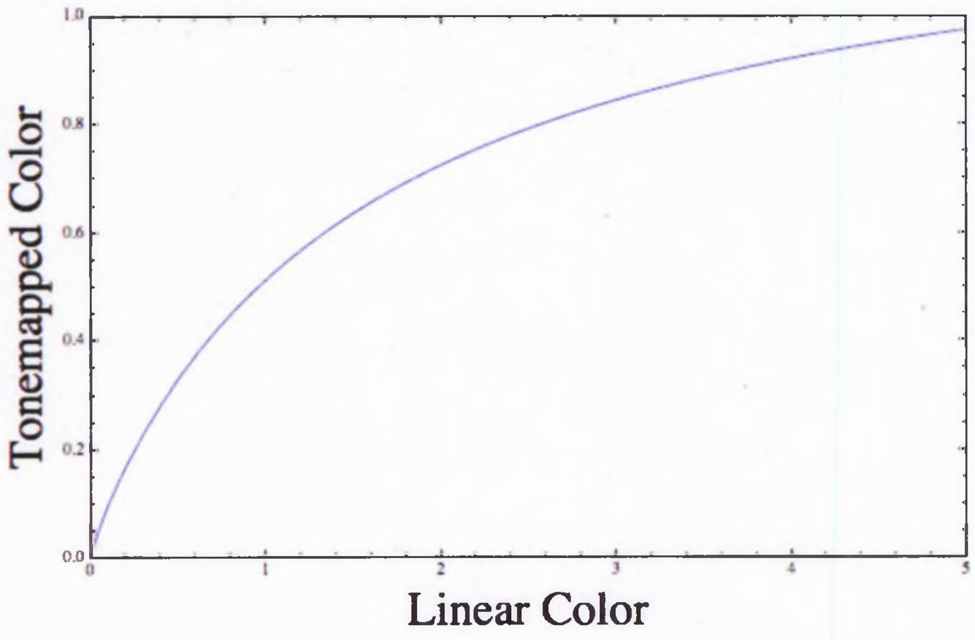


Figure 58: Hable Tone-mapping response curve.

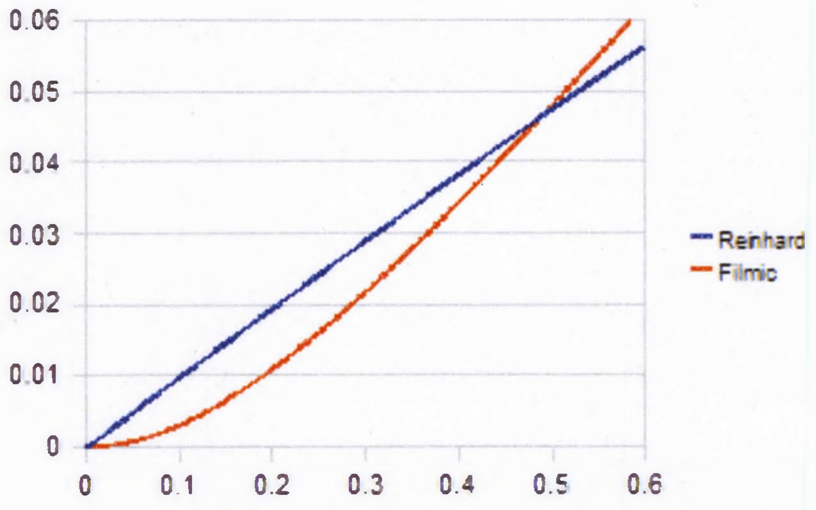


Figure 59: The low end of Reinhard's and Hable's (Filmic) tonemapping operators. Hable's filmic tonemapping operator has a toe section which avoids Reinhard's desaturation of low intensity colors. Image from [Hable 2010b].

Reinhard's and Hable's tonemapping operators map input in the range [0.0, infinity[to the displayable [0.0, 1.0] range of low dynamic range displays; and while this is sufficient as a final step in the pipeline for displaying a high dynamic range image the side-effect of clipping bright values might not be desirable in an intermediate step that is a part of a complex rendering pipeline. Brightness cutoff passes, such as the bright region detection pass used for bloom and lens flare rendering, rely on bright regions (with per-component color values greater than the maximum displayable value of 1.0) retaining their values.

A tone-mapping operator suitable for use as part of a high dynamic range processing pipeline should be able to preserve the contrast between light and dark regions by not clipping highlights while at the same time preserving data that is outside the displayable range for use in subsequent image processing steps.

PROPOSED APPROACH

The proposed approach to tonemapping is a new, purely mathematical operator that does not depend on any lookup tables (and is thus able to tonemap any range of color intensities) while applying an s-curve transfer function.

The tonemapping operator relies on the error function, which is not natively supported by graphics hardware or shading languages. A branch-free implementation was used based on the approximation in equation 9 from [Abramowitz and Stegun 1964] which has a maximum error of $5 \cdot 10^{-4}$. The approximation is especially good since most of the denominator can be computed using very few instructions on GPUs with 4-wide vector data processing support. A branch-free implementation of the error function suitable for use in GPU shading languages can be found in Appendix A.

$$erf(x) \approx 1 - \frac{1}{(1 + a_1x + a_2x^2 + a_3x^3 + a_4x^4)^4} \qquad (9)$$

The proposed tone-mapping function can map colors from the range [0.0, ***m***] to the range [0.0, ***n***] where ***m*** and ***n*** are any two numbers greater than zero. The ability to map inputs between two arbitrary positive ranges gives the operator the ability to be an intermediate part of a rendering pipeline - a high dynamic range image can be tonemapped using this operator for exposure adjustment and s-curve application with the output being another high dynamic range image as opposed to a low dynamic range image limited to the range [0.0, 1.0] as is the case with other tonemapping techniques.

$$T(x) = S_r \times (O - M \times erf(B - (C_b + C_s \times x)^{C_e})) \qquad (10)$$

Equation 10 shows the proposed tonemapping operator ***T***. A range scale parameter (***S_r***) determines the range of the output values, with higher values corresponding to higher output ranges and vice versa. An offset parameter (***O***) linearly affects the tonemapped values' range and exposure. A multiplier (***M***) controls the wideness or narrowness of the output range. The error function's term has an overall bias (***B***) and a linear color bias (***C_b***) acting as linear exposure adjustments, a color scale (***C_s***) affecting the extent of the s-curve's mid region and a color exponent (***C_e***) affecting the toe and shoulder regions.

Figure 60 shows a variety of possible tonemapping curves from the operator in equation 10. The parameters allow for a very flexible s-curve color mapping and exposure adjustment.

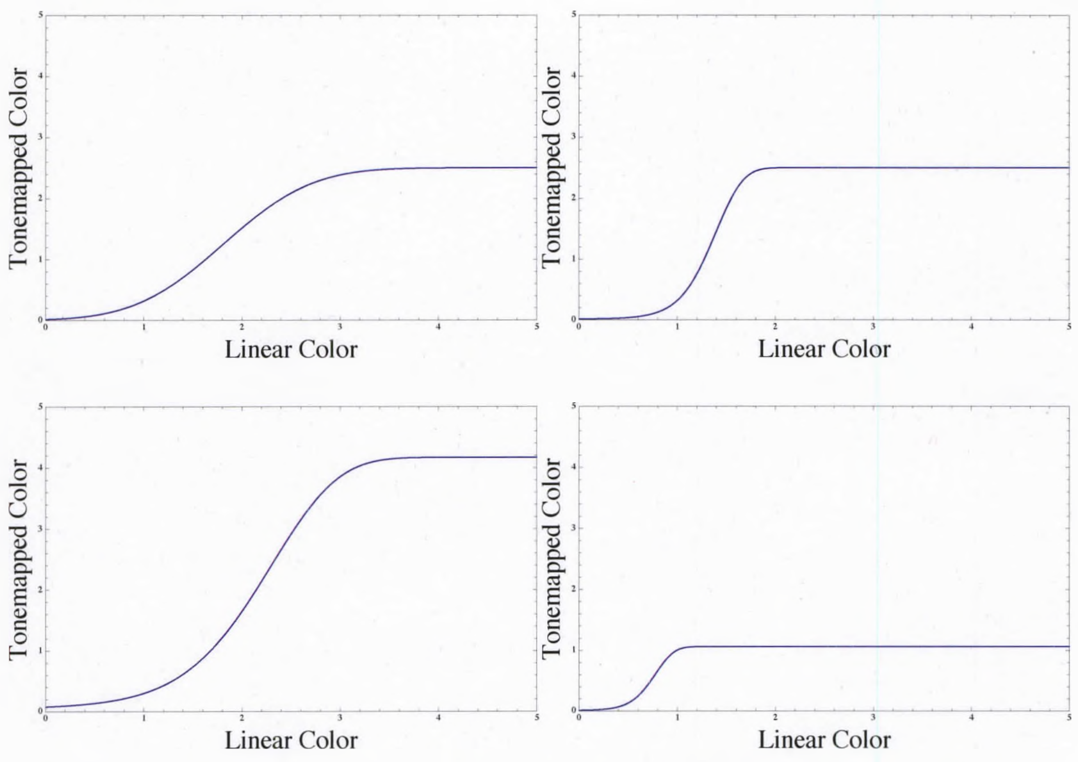


Figure 60: A variety of tone-mapping curves from the operator in equation 10.

Results

The proposed tonemapping operator avoids the unsaturated low intensity regions of Reinhard's operator and the over-bright mid tones of Hable's operator, resulting in a much vibrant look that is close to that of film when applied to linear color content. Figure 62 shows a comparison between Reinhard's, Hable's and the proposed tonemapping operators.

Film material can have different characteristic curves for red, green and blue (figure 61). By applying the proposed tonemapping operator independently to each channel with a separate set of parameter the visual effect of per-component characteristic curves can be reproduced, or the per-component values can be used for color grading as opposed to attempting to recreate a film curve. Figure 63 shows a comparison between uniform curve parameters and per color component curve parameters.

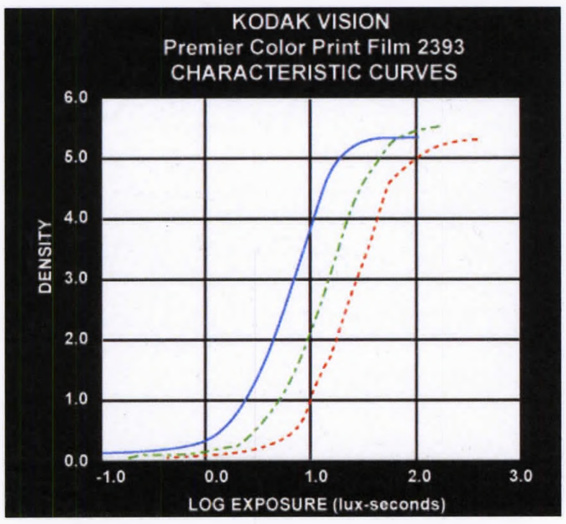


Figure 61: Kodak Premier Color Print Film 2393 characteristic curves for red, green and blue.

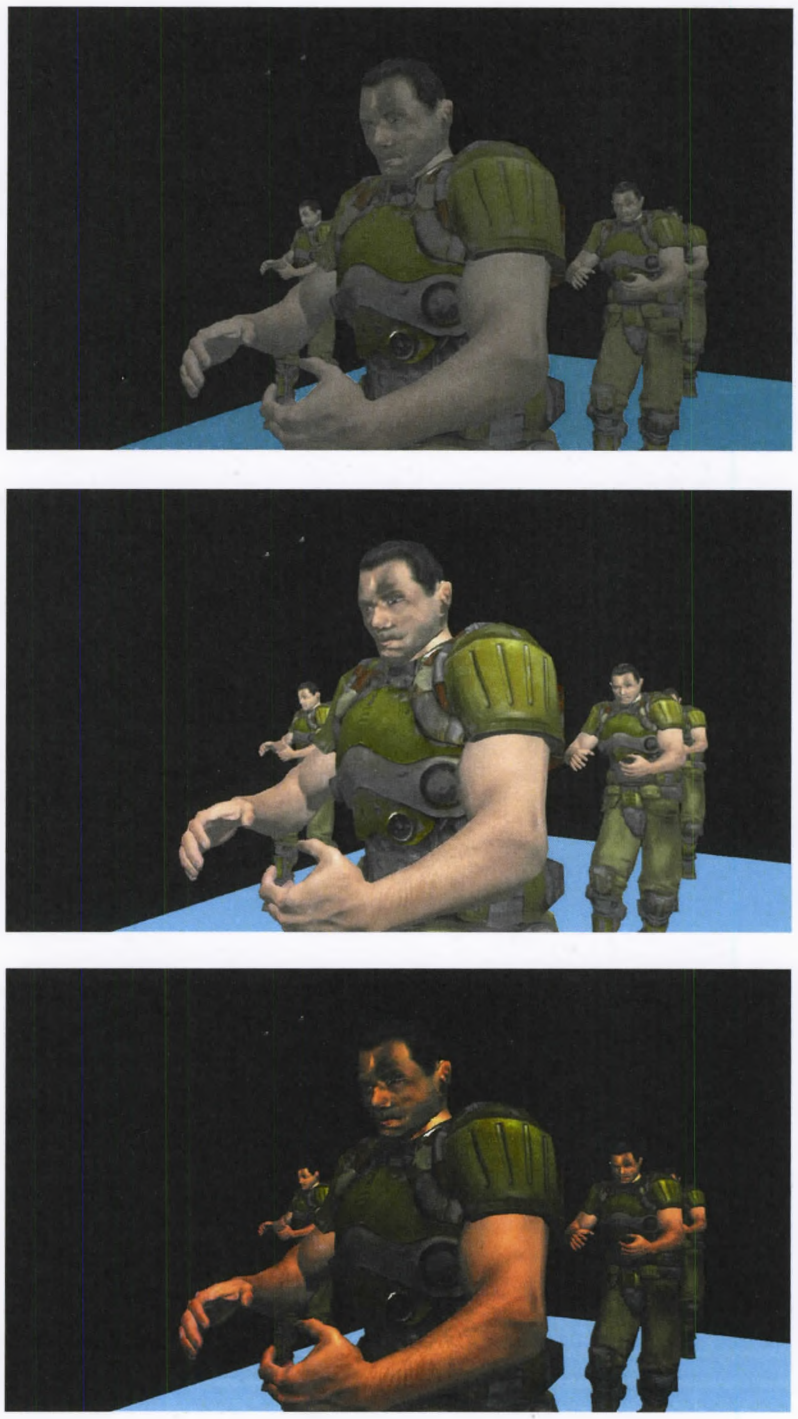


Figure 62: Scene rendered using Reinhard (top), Hable (middle) and proposed tonemapping algorithms.

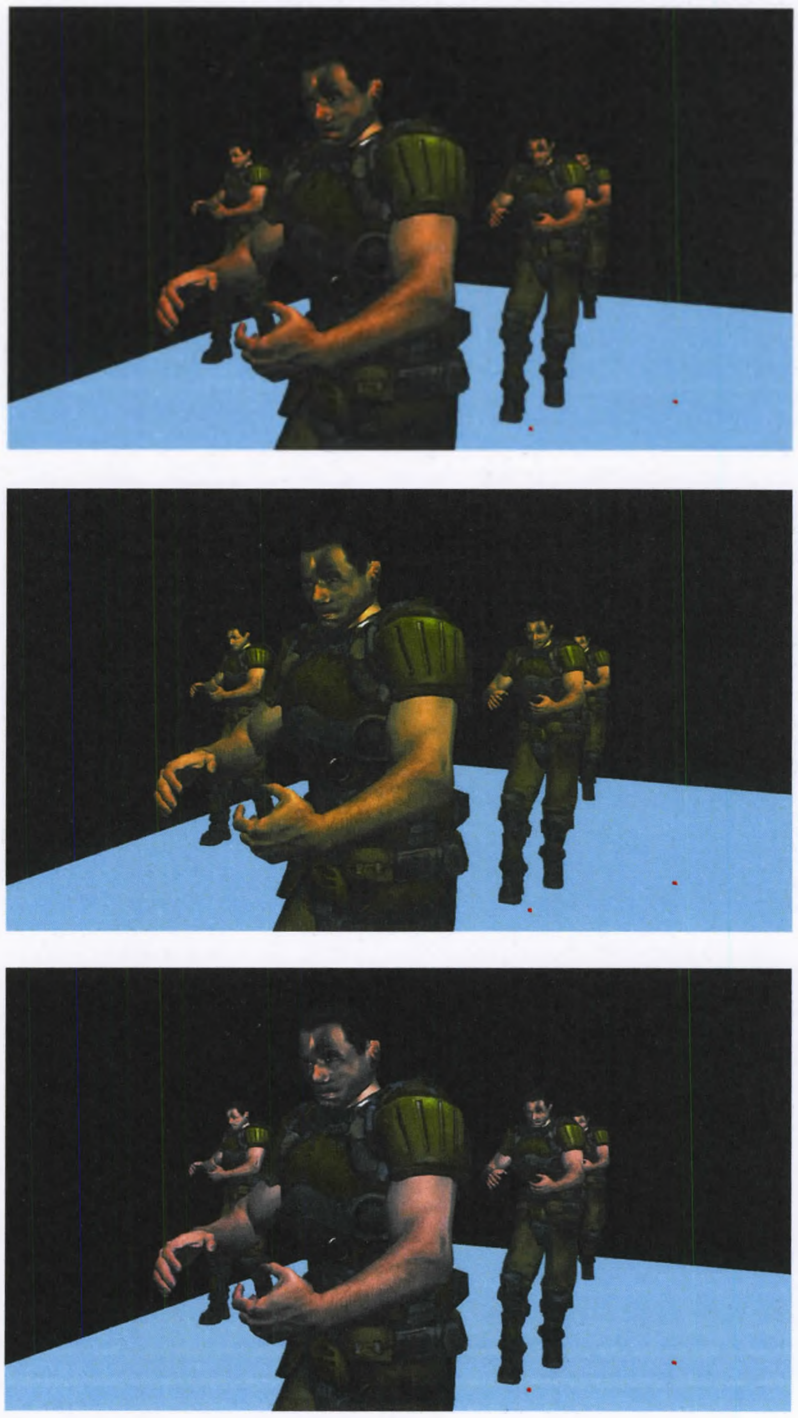


Figure 63: Examples of the proposed tonemapping operator applied to a scene using different curve parameters for each of the red, green and blue color components.

CHAPTER SIX:

FRAME RATE

Film is usually shot and displayed at 24 frames per second (FPS), meaning that the camera captures the incoming light over 1/24th of a second onto a single frame of video. Historically, film cameras used Rotary Disc Shutters with a physical half-circle shaped disk that would expose the film at regular intervals (figure 64) while using the aperture to adjust the amount of light hitting the film based on the scene's brightness and mid-tones.

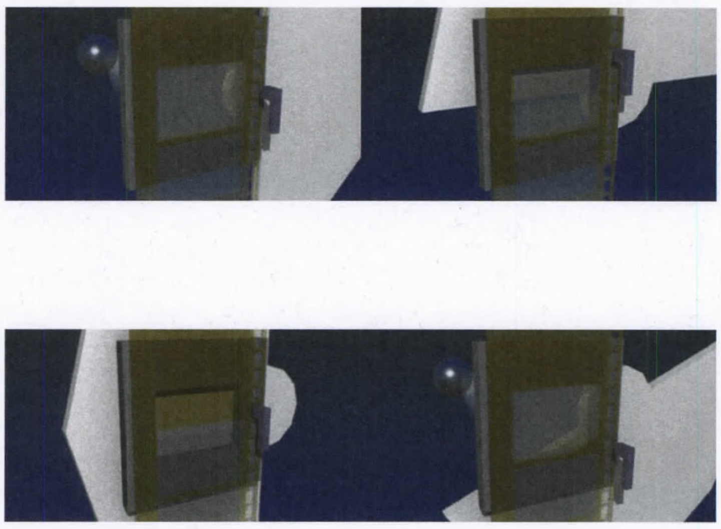


Figure 64: A Rotary Disc Shutter exposing a film (clockwise, starting at top left).

That frame-rate is one major contributor to the traditional Film look, as opposed to the Camcorder look which is a result of camcorders recording at 30 or 60 frames per second.

Audiences are able to differentiate between 24 FPS footage and 50 / 60 FPS footage, usually referring to the latter as the "Soap Opera Look". Recently Director Peter Jackson (Lord of the Rings) screened his new movie, The Hobbit, shot at 48 FPS and the reception of the new high frame rate projection format was not positive [Faraci 2012].

The majority of consumer displays refresh 60 times per second (60 Hz), which makes natively displaying 24 FPS footage impossible since 60 cannot be divided evenly by 24. To overcome this technical limitation a technique called Pulldown is used - since 24 does not divide evenly into 30 or 60, some frames must be repeated more often than others in order to stretch the 24 frame-per-second over a 30 or 60 frame interval suitable for displaying on a 60 Hz screen.

One pattern for doing this is 3:2 pulldown where alternating patterns of 3 and 2 repeated frames are used (figure 65). Another is 2:2:2:4 pulldown (figure 66) where three sets of 2 repeats followed by one set of 4 repeats are used. Professional video editing software like Final Cut Pro usually use 2:2:2:4 pulldown. Various techniques for dealing with frame rate conversion on computer displays have been covered in [Marsh 2001].

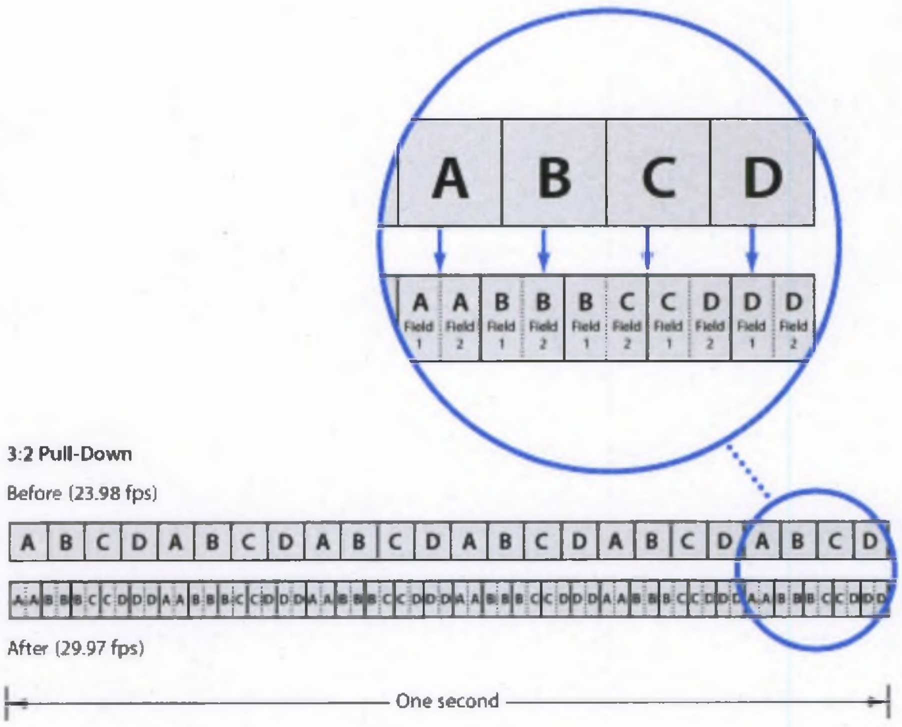


Figure 65: 3:2 Pulldown. Image from Final Cut User Manual.

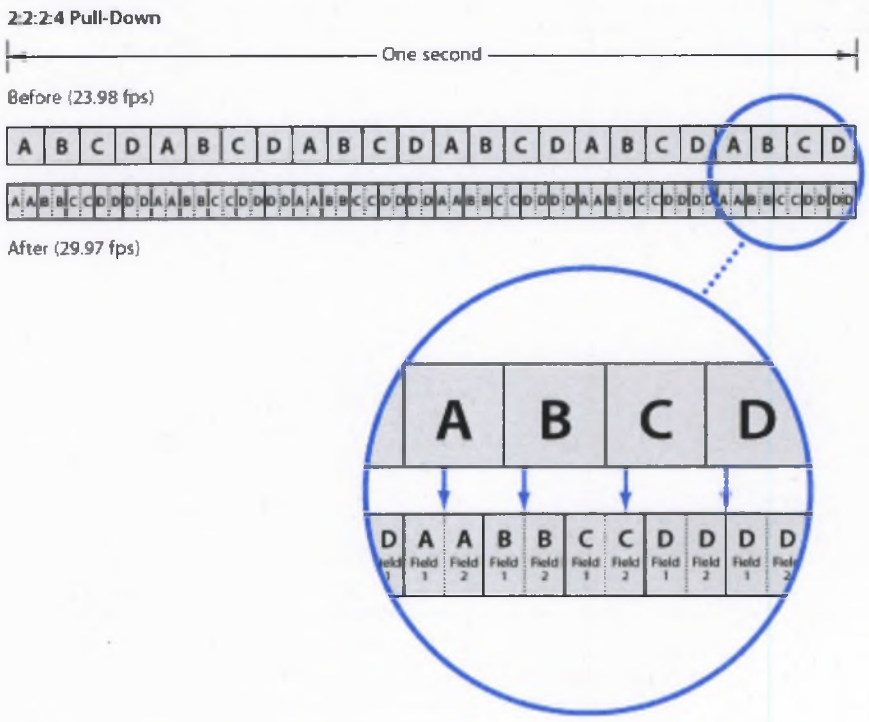


Figure 66: 2:2:2:4 Pulldown. Image from Final Cut User Manual.

On platforms that offer tight integration with the graphics vertical retrace (VBLANK / VSYNC) signals the computer-generated scene can be rendered at a film-like 24 frames-per-second and pulldown algorithms can be applied and used to adapt the output to the screen in use which is typically a 60 Hz device in the case of PCs and gaming consoles. The proof-of-concept application developed for this thesis uses the CoreVideo framework on OS X to implement 2:2:2:4 pull-down.

A side-effect of 24 FPS capture that needs to be emulated in computer-generated renderings is motion blur - since the camera shutter remains open for 1/24 of a second (which is a relatively large time period with respect to the screen-space velocities of the objects being photographed with respect to the camera) in every frame there will be an apparent streaking of objects in motion. The next chapter will cover some of the existing algorithms for emulating motion-blur artifacts in real-time computer graphics.

CHAPTER SEVEN:

MOTION BLUR

INTRODUCTION

Motion Blur is the apparent streaking of moving objects in a still image or a video frame. Figure 67 shows an exaggerated motion blur, the result of a long photograph exposure time, of a bus that's moving with respect to the camera. Stationary objects or objects that don't change their position relative to the camera during the exposure time are not affected by motion blur, such as the telephone booth in the figure.

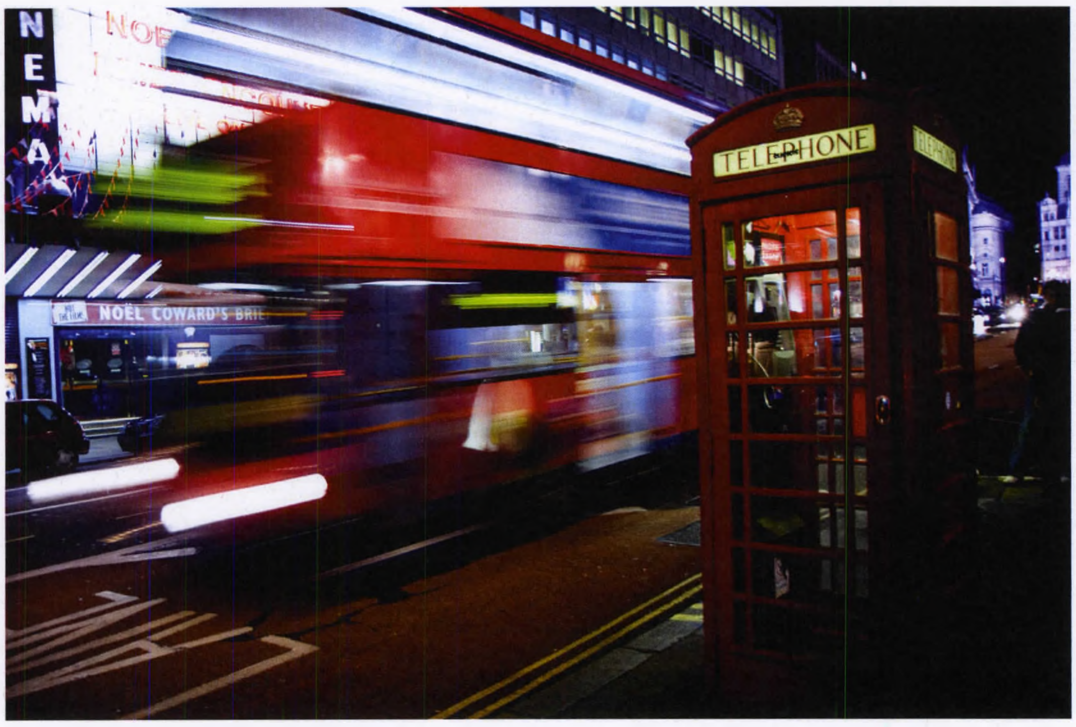


Figure 67: A long-exposure photograph of a moving bus against a stationary background. The bus exhibits exaggerated motion blur due to the long exposure time. Source: Wikimedia Commons

When the camera shutter stays open for a specified duration all the light emitted by an object during that time is captured, and an object that moves significantly during that time period it will show streaks representing the path of motion it took. Motion Blur depends primarily on the shutter duration and the distance the object moves during time the shutter is open. Therefore the smaller the shutter duration and the less distance the object moves the less motion blur it will generate.

Computer graphics simulate a camera with a shutter duration of zero, so in a straight-forward rendering a scene exhibits no motion blur whatsoever. Figure 68 shows a first-person scene from the video game Half-Life 2 - while the interactive scene has the character on the left apparently in motion but the still image offers no cues that can be used to infer the direction or intensity of that motion.

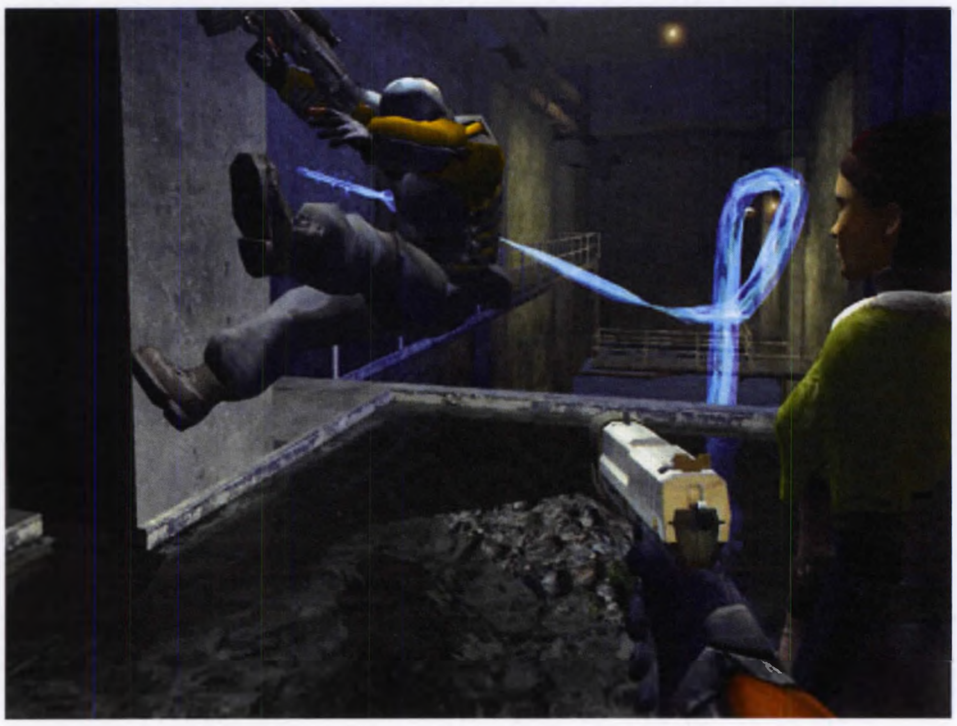


Figure 68: Scene from the video-game Half-Life 2.

Images and videos made using still and film cameras having a finite shutter duration tend to have motion blur of varying degrees which offers visual cues about the motion of the objects in the image. Figure 69 is a still photograph of a drummer caught in the act - the motion blur of his hands and drumsticks cues the viewer to the fact that this images was taken as he was actively playing the drums.

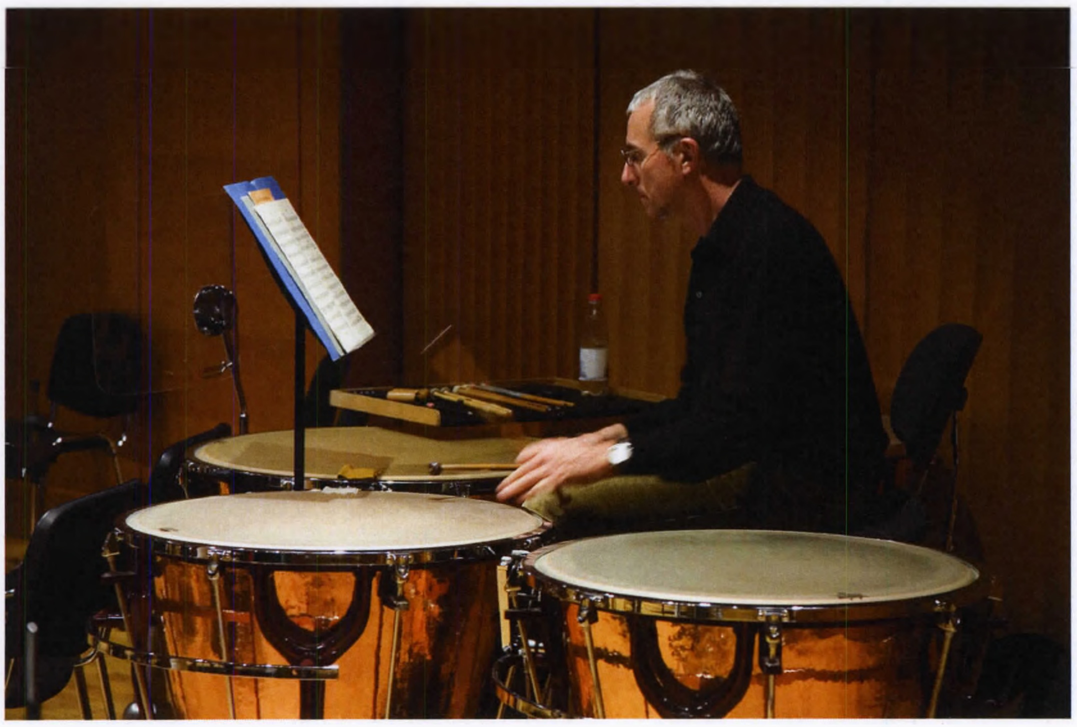


Figure 69: Still photograph of drummer. Source: Wikimedia Commons

Motion blur is very prevalent in film footage. Due to the relatively low frame rate of 24 frames per second, corresponding to a shutter exposure time of around 41.5 milliseconds, significant motion blur can be seen in most still frames of film except those with very little subject and camera motion.

USED TECHNIQUE

The proof-of-concept application developed for this thesis uses the motion blur technique from [Lengyel 2010], which operates as a post-process pass performing a directional per-pixel blur based on the scree-space velocity of the pixel and relies on using the view-space depth gradient in the X and Y dimensions to avoid intensity leakage.

Since the depth of field algorithm proposed earlier guarantees that the depths of all the contributors to a pixel will have a depth difference that is at most $\boldsymbol{\Delta}$, the depth gradient can still be used to estimate object boundaries and avoid intensity leakage when using the algorithm from [Lengyel 2010].

Figure 70 shows a scene with an animated charactered. The direction and intensity of the animation aren't visible in the figure since no motion blur is applied. Figure 71 shows a false-color visualization of the screen-space velocity vector associated with every pixel in the scene and figure 72 shows the result of rendering the scene with applied motion blur based on the screen-space velocities.

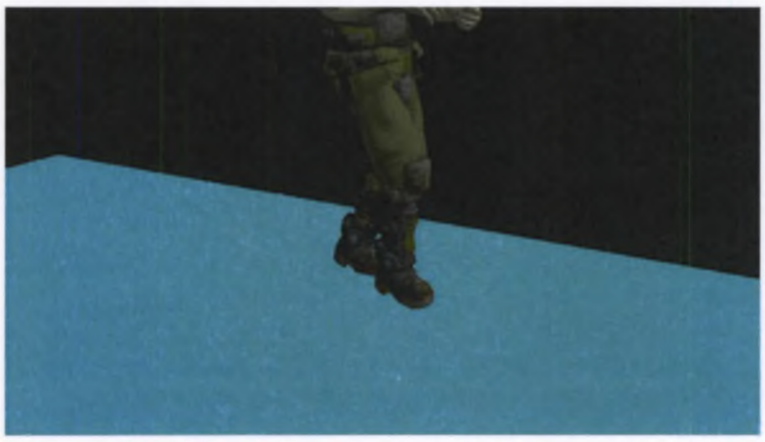


Figure 70: Animated scene rendered without motion blur.

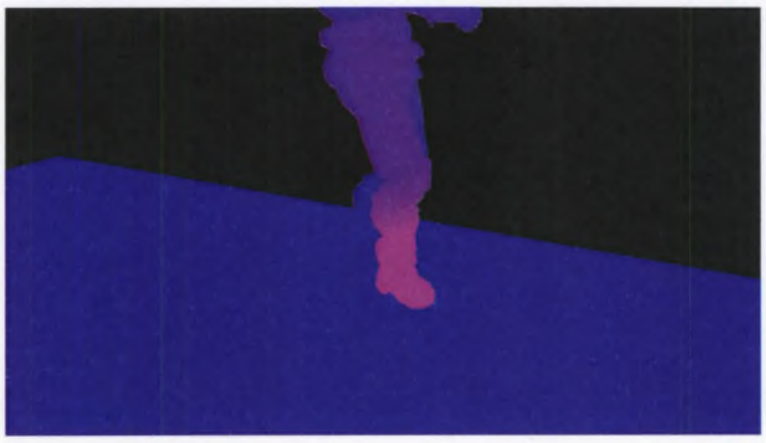


Figure 71: False-color visualization of the per-pixel velocity vector of the scene.

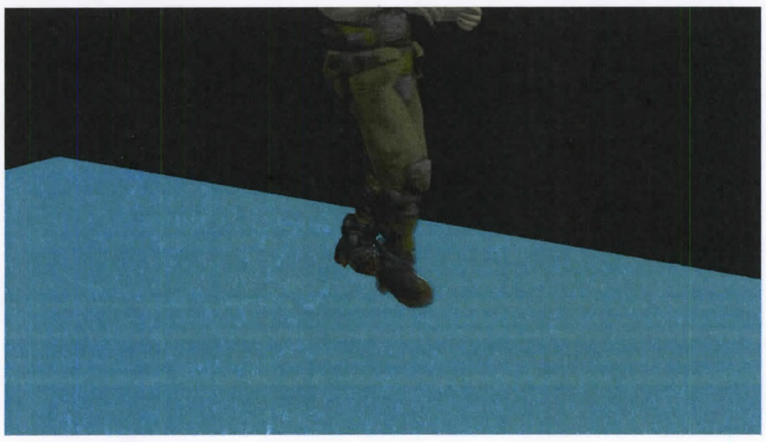


Figure 72: Animated scene rendered with applied motion blur based on [Lengyle 2010].

CHAPTER EIGHT:

PROJECTOR COLOR CONVERGENCE

INTRODUCTION

Film projectors and rear projection televisions suffer from an image artifact known as the lack of color convergence. In perfect color convergence the red, green and blue portions of the image projected onto the display by the rear projection system line up exactly and the image appears artifact free, but in the case of sub-optimal color convergence the three color channels do not line up perfectly and color artifacts around the edges of regions can be seen. Figure 73 shows an example of the color artifacts resulting from a projection system with sub-optimal color convergence.

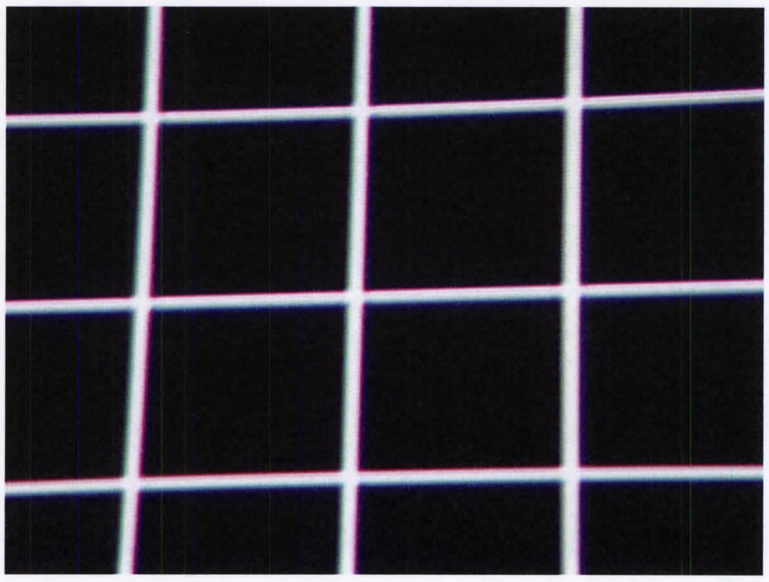


Figure 73: A grid of black squares with white grid lines as seen through a projection system with sub-optimal color convergence. The edges of the black squares bleed some of their color components onto the white grid lines. Image source: (http://www.avsforum.com/t/1100329/lcd-projector-what-should-i-see-up-close).

Extreme lack of color convergence is a distracting artifact which figure 74 shows, but slight pixel or sub-pixel color divergence exists on almost all projection-based systems and is considered to be part of the look and feel of projected images. Reproducing color divergence in computer-generated imagery is a step towards achieving projected film look and feel.

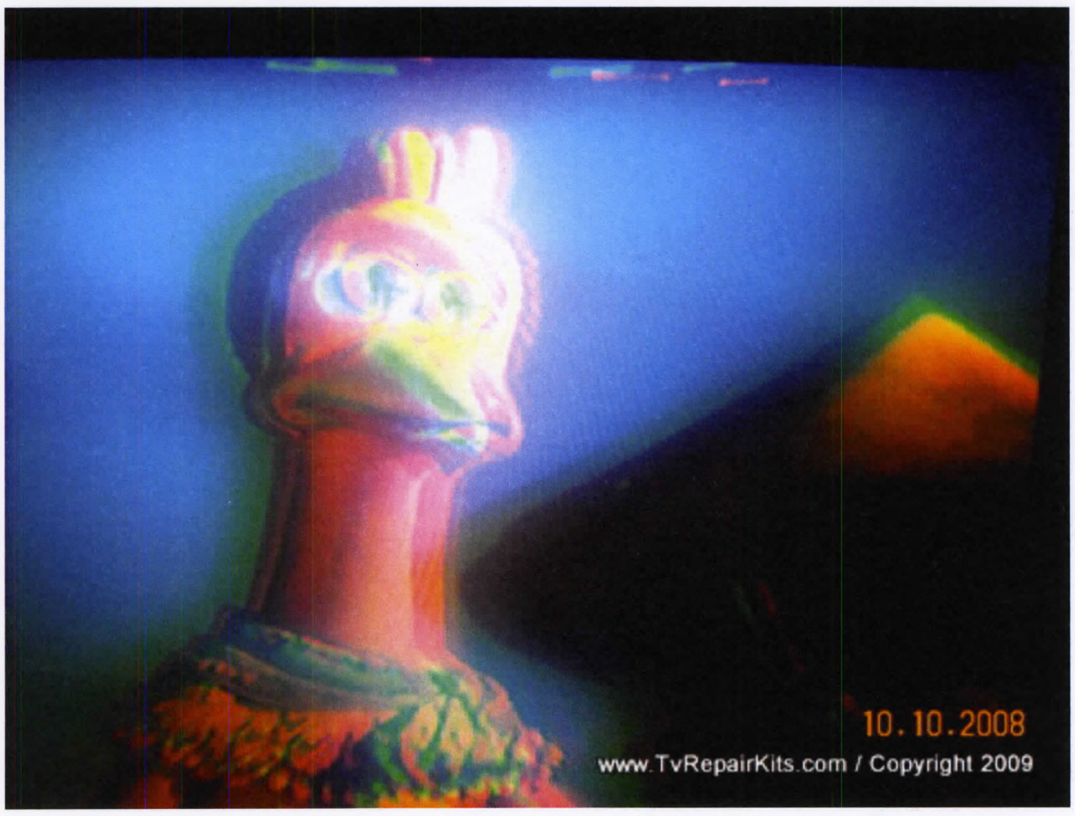


Figure 74: Extreme lack of color convergence in a projection-based display system. Image from TvRepairKits.com.

Color divergence isn't usually uniform - the divergence intensity and direction varies across the display surface. Figure 75 shows an example of a display exhibiting color divergence with non-uniform intensity and direction distribution.

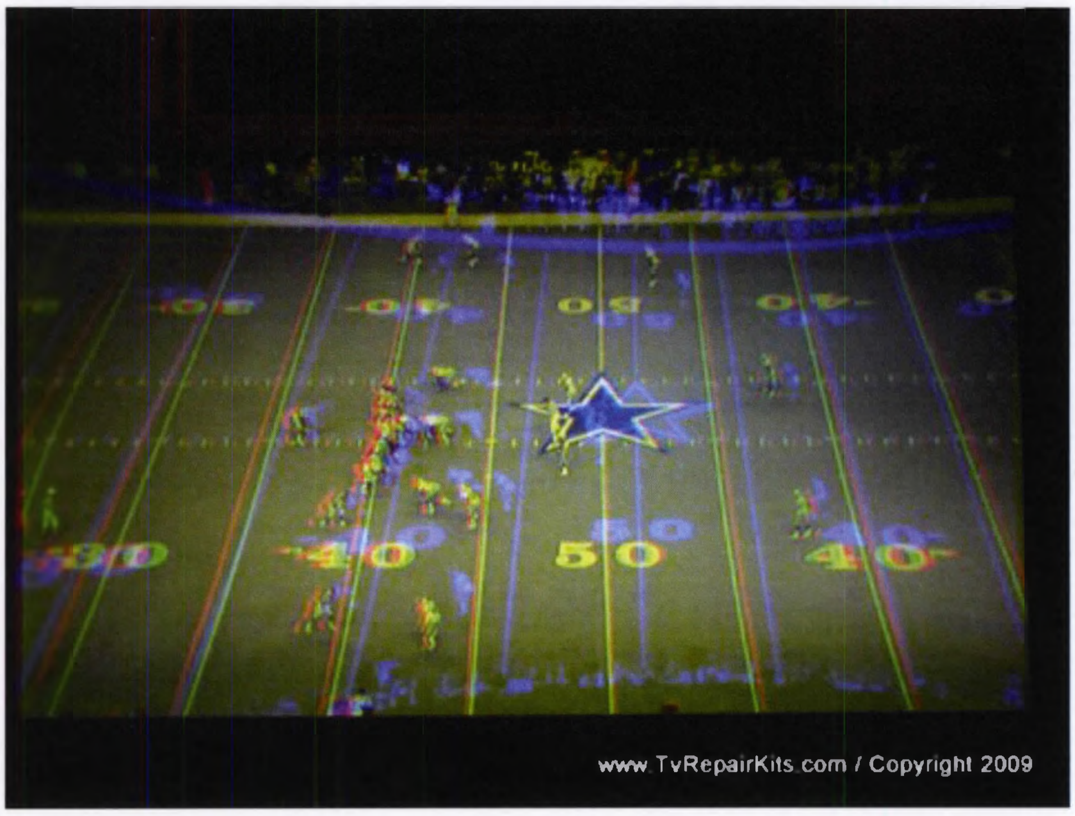


Figure 75: Color divergence across a display surface exhibiting non-uniform direction and intensity. Blue for example diverges upwards in the lower half of the screen and downwards in the upper half, while red exhibits a different divergence intensity across the screen's width, most obvious along the field's yard lines. Image from TvRepairKits.com.

PREVIOUS WORK

[Kawase 2003] proposed using a post-processing pass with three separate sets of texture coordinates passed to the vertex stage which are then linearly interpolated across the surface of the view plane aligned quadrilateral and used to sample the source texture's red, green and blue channels separately. Each of the three sets of texture coordinates would be set to correspond to the shaded pixel's coordinates with an additional offset which would determine the final color divergence - for example, a texture coordinate used for sampling the red color channel with an offset of (ΔX, ΔY) would cause the post-process resulting image to have its red color component diverging from its ideal position by (ΔX, ΔY).

The algorithm presented in [Kawase 2003] suffered from a major drawback: with the diverged texture coordinates generated at a per-vertex level the color divergence is either uniform across the display image or linearly varying across the view-plane aligned quadrilateral used in the post-processing pass due to the per-vertex interpolation of attributes passed to the fragment program. The proposed algorithm offers a way to have independent per-pixel and per color component divergence direction and distance at a performance cost that is only very slightly more than the algorithm in [Kawase 2003] - specifically two additional texture fetches and a few math instructions.

PROPOSED TECHNIQUE

The proposed algorithm operates as a post-processing pass and uses two values at each pixel's color component to specify the color component's divergence direction and distance: a normalized two-dimensional vector $\boldsymbol{D}$ specifying the divergence direction and a scalar $\boldsymbol{\Delta}$ specifying the divergence distance. Equation 11 is then calculated in the fragment program to get the final texture sampling coordinate $\boldsymbol{C}$ from the original texture sampling coordinate $\boldsymbol{T}$. $\boldsymbol{C}$ is calculated for each of the red, green and blue color component channels to get three separate texture coordinates $\boldsymbol{C_1}$, $\boldsymbol{C_2}$, $\boldsymbol{C_3}$. Using each of these coordinates respectively to sample the source texture's red, green and blue channels results in the desired diverged color component values which are then written to the output framebuffer as the result of the post-processing pass.

$$C = T + \hat{D} \times \Delta \qquad (11)$$

Specifying per-pixel and per color component divergence direction and distance is done using two three-component textures with normalized 8-bits per component storage (RGB 8) referred to as the direction texture and the distance texture.

A global value $\boldsymbol{\Delta_{max}}$ is passed to the post-processing fragment program and the $\boldsymbol{\Delta}$ value for each pixel's color component is calculated by sampling the distance texture at the current texture coordinate $\boldsymbol{T}$ and multiplying the desired component's value (which due to the texture's format will be a number in the range [0.0, 1.0]) by $\boldsymbol{\Delta_{max}}$ to get the current pixel component's $\boldsymbol{\Delta}$.

Calculating ***D*** per pixel color component is done by taking advantage of the fact that ***D*** is a normalized two-component vector, so the values of its x and y component will be confined to the range [-1.0, 1.0]. Values in the range [-1.0, 1.0] can be compressed into the range [0.0, 1.0] using equation 12.

$$pack(x) = (x + 1.0)/2.0 \qquad (12)$$

With the components of ***D*** compressed into the range [0.0, 1.0] they can be stored in components of the direction texture and uncompressed in the fragment program using equation 13.

$$unpack(x) = (x \times 2.0) - 1.0 \qquad (13)$$

Since ***D*** is a normalized vector storing the X component only in the direction texture would be sufficient since the Y component can be re-calculated using equation 14.

$$y = \sqrt{1.0 - x^2} \qquad (14)$$

Using equations 12, 13 and 14 three two-dimensional normalized vector fields can be stored in the three-component direction texture, each vector field representing the normalized direction vector ***D*** at every pixel color component. The fragment program samples the direction texture at the current texture coordinate ***T*** and uses the values in the red, green and blue channels to calculate the vector ***D*** for each color component.

Using the per-pixel color component ***Δ*** and ***D*** values calculated in the fragment program and plugging them into equation 11 to get divergence texture coordinates ***C*** allows for the desired per-pixel color component divergence calculation.

ALGORITHM

```
ProjectorSeparation(SourceImage, DistanceTexture, DirectionTexture, T, $\Delta_{max}$):
    Distances ← sample(DistanceTexture, T)
    RedDistance ← Distances.r * $\Delta_{max}$
    GreenDistance ← Distances.g * $\Delta_{max}$
    BlueDistance ← Distances.b * $\Delta_{max}$
    Directions ← sample(DirectionTexture, T)
    RedDirection ← (0, 0)
    RedDirection.x ← Directions.r
    RedDirection.y ← sqrt(1.0 - $Directions.r^2$)
    GreenDirection ← (0, 0)
    GreenDirection.x ← Directions.g
    GreenDirection.y ← sqrt(1.0 - $Directions.g^2$)
    BlueDirection ← (0, 0)
    BlueDirection.x ← Directions.b
    BlueDirection.y ← sqrt(1.0 - $Directions.b^2$)
    Red = sample(SourceImage, T + (RedDirection * RedDistance))
    Green = sample(SourceImage, T + (GreenDirection * GreenDistance))
    Blue = sample(SourceImage, T + (BlueDirection * BlueDistance))
    return (Red, Green, Blue)
```

RESULTS

Figure 76 shows a reference used as an input image to the projector separation algorithm and figure 77 shows the same scene after the projector separation algorithm is applied.

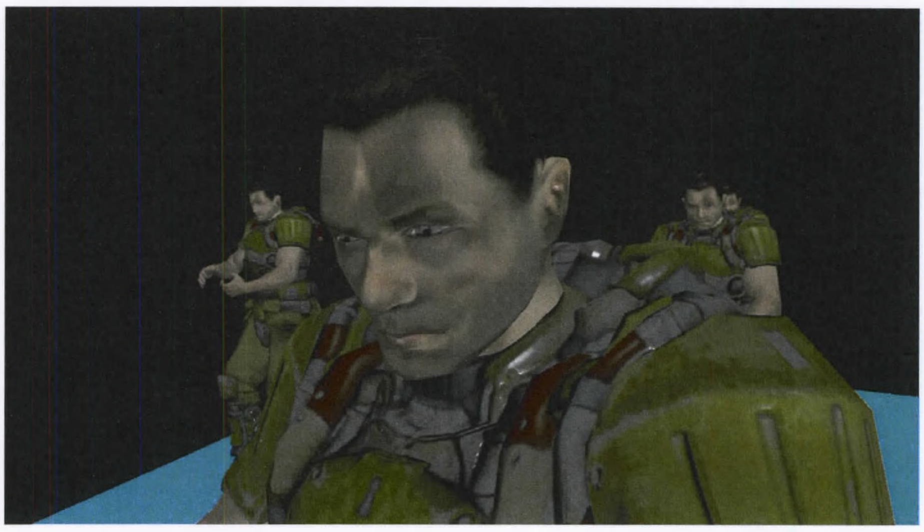


Figure 76: A reference scene serving as an input image to the projector separation algorithm.

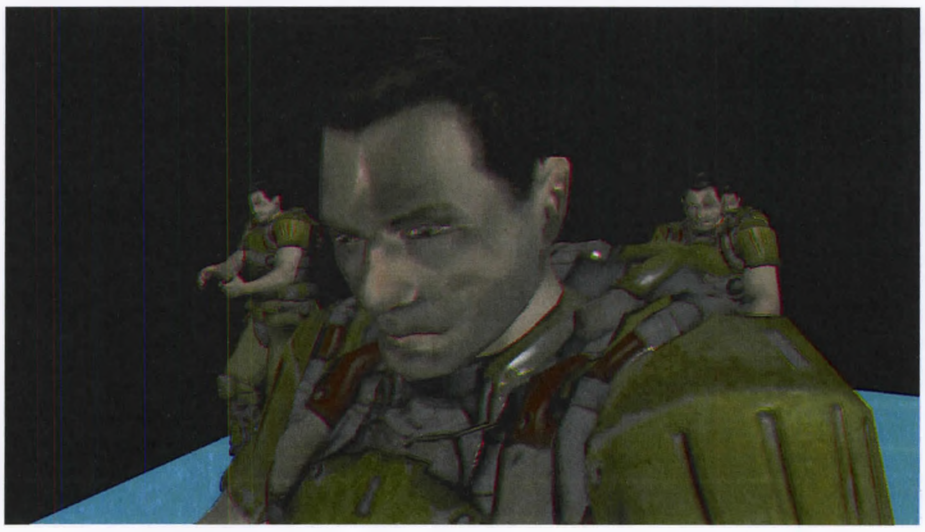


Figure 77: Scene with uniform color separation.

For clarity, figure 78 shows the same scene with the same divergence direction (***D***) values but with a largely exaggerated divergence distance (***Δ***) value.

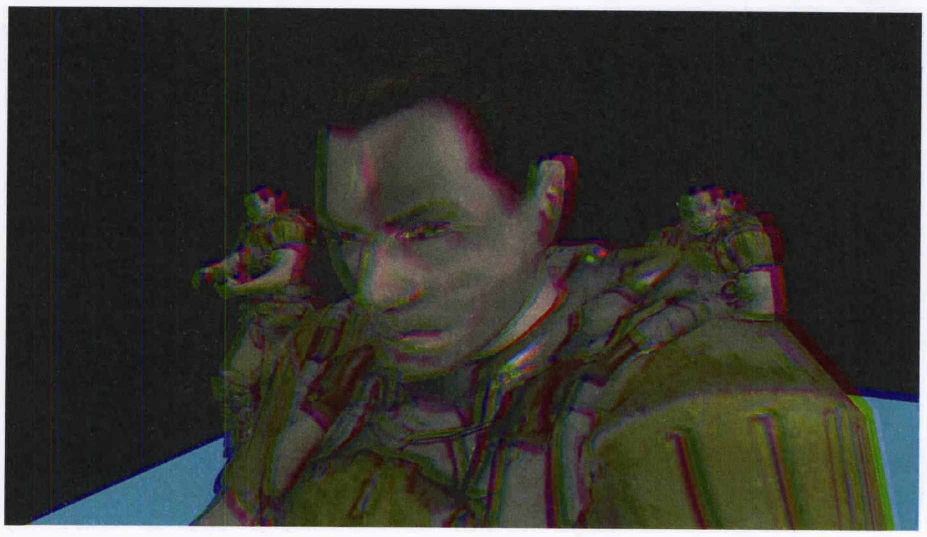


Figure 78: Scene with uniform color separation with an exaggerated distance (**Δ**) values.

Non-uniform divergence distance can be shown using a texture such as the vignette in figure 79. Figure 80 shows the scene with divergence distances determined by the vignette texture and figure 81 shows the scene with the vignette divergence distance texture and an exaggerated divergence distance for clarity.

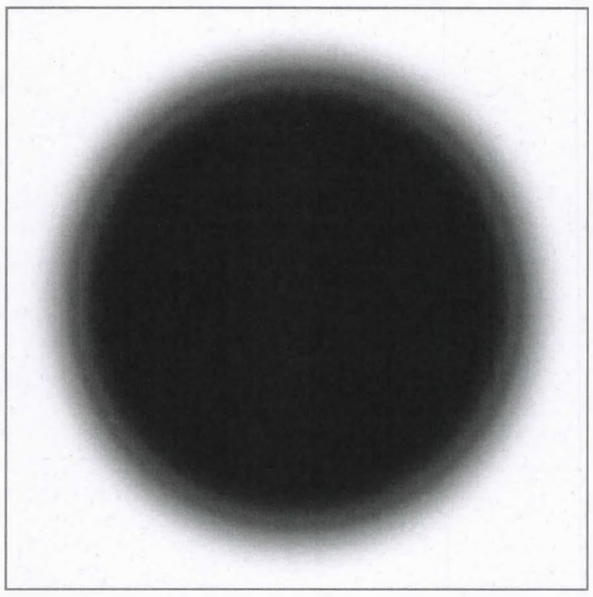


Figure 79: A vignette used as a divergence distance texture to show non-uniform divergence distances.

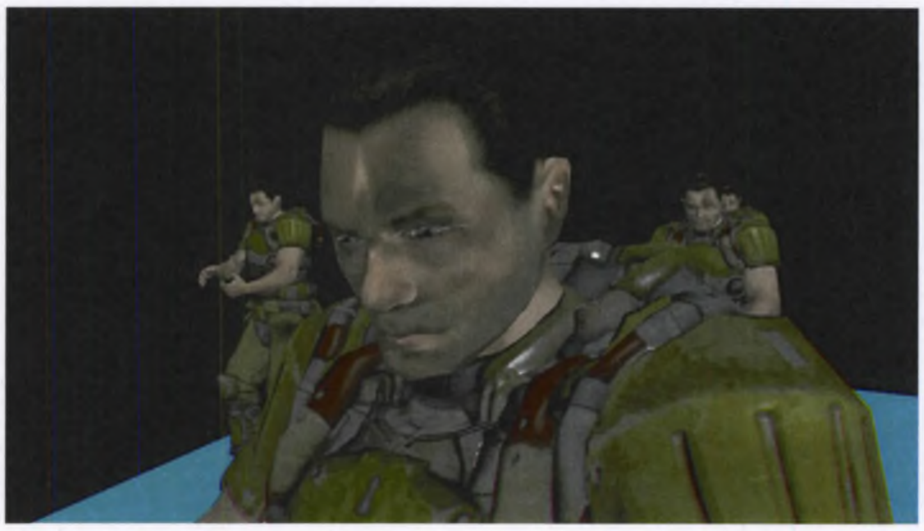


Figure 80: Scene with non-uniform color divergence using the distance texture from figure 79. The center has no color divergence while the edges exhibit divergence.

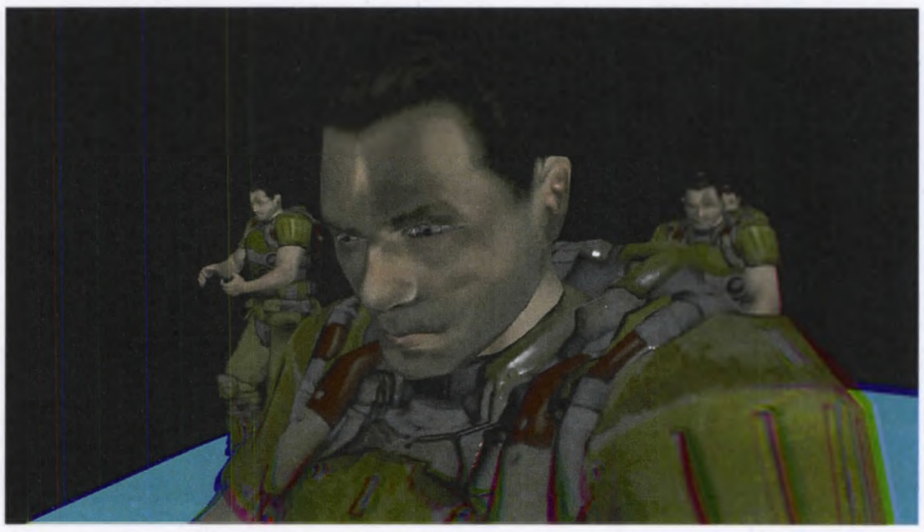


Figure 81: Scene with the same exaggerated divergence distances from figure 78 using the distance texture from figure 79. The center has no color divergence while the edges exhibit divergence.

CONCLUSIONS

The thesis presented algorithms suitable for use in real time rendering on traditional consumer hardware that simulate a variety of the artifacts and visual themes unique to the look achieved by the film imaging process.

The depth of field algorithm presented solves the issue of intensity leakage common in gather methods while adding the ability to simulate non-uniform chromatic aberration at a very low performance cost. The ghost generation algorithm presented allows for rendering ghosts with varying position, orientation and color parameters for both primary and secondary light sources at a low performance cost that is independent of the number of light sources in a given scene. The proposed tonemapping operator provides an extremely flexible method to reproduce a wide variety of color response curves and exposures across a potentially infinite high dynamic range color range for both input and output. Finally, the color convergence algorithm allows for replication of color convergence display artifacts common to projection displays across any vector field describing the direction and intensity of color convergence with pixel accuracy.

The proof-of-concept application designed to implement a real time rendering pipeline incorporating the methods presented in this thesis runs at interactive frame rates on a 2012 MacBook Pro with Retina Display which features a mobile low-power graphics chip, the GeForce 650M by NVIDIA, with the system operating at or under 85 watts of power consumption.

BIBLIOGRAPHY

Abramowitz, M. and I. Stegun (1964). Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.

Barsky, B. A. (2004). Vision-realistic rendering: simulation of the scanned foveal image from wavefront data of human subjects. Proceedings of the 1st Symposium on Applied perception in graphics and visualization. Los Angeles, California, ACM: 73-81.

Barsky, B. A., et al. (2002). "Introducing Vision-Realistic Rendering." Thirteenth Eurographics Workshop on Rendering.

Bertalmio, M., et al. (2004). Real-Time, Accurate Depth of Field using Anisotropic Diffusion and Programmable Graphics Cards. Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium, IEEE Computer Society: 767-773.

Blythe, D. (2006). "The Direct3D 10 system." ACM Trans. Graph. **25**(3): 724-734.
Burrows, M., et al. (2004). Advanced Visual Effects with Direct3D. Game Developers Conference.

Debevec, P., et al. (2004). High dynamic range imaging. ACM SIGGRAPH 2004 Course Notes. Los Angeles, CA, ACM: 14.

Faraci, D. (2012). "CinemaCon 2012: The Hobbit Underwhelms At 48 Frames Per Second." from http://badassdigest.com/2012/04/24/cinemacon-2012-the-hobbit-underwhelms-at-48-frames-per-secon/

Hable, J. (2010). "Filmic Tonemapping Operators." from http://filmicgames.com/archives/75.

Hable, J. (2010). "Why a Filmic Curve Saturates Your Blacks."

Haeberli, P. and K. Akeley (1990). The Accumulation Buffer: Hardware Support for High-Quality Rendering. SIGGRAPH.

Hammon, E. (2007). Practical Post-Process Depth of Field. GPU Gems 3.

Hullin, M., et al. (2011). Physically-based real-time lens flare rendering. ACM SIGGRAPH 2011 papers. Vancouver, British Columbia, Canada, ACM: 1-10.

James, G. (2004). Real-Time Glow. GPU Gems.

Kawase, M. (2003). Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless). Game Developers Conference.

Kilgard, M. (2000). "Fast OpenGL-rendering of Lens Flares." from http://www.opengl.org/archives/resources/features/KilgardTechniques/LensFlare/.

King, Y. (2000). 2D Lens Flare. Game Programming Gems.

Kodak (1998). Basic Sensitometry and Characteristics of Film: 49-60.

Kodak (1998). "Technical Data - KODAK VISION Premier Color Print Film 2393." from http://motion.kodak.com/motion/Products/Distribution_And_Exhibition/2393/tech2393.htm.

Kosloff, T. J. and B. A. Barsky (2007). An algorithm for rendering generalized depth of field effects based on simulated heat diffusion. Proceedings of the 2007 international conference on Computational science and its applications - Volume Part III. Kuala Lumpur, Malaysia, Springer-Verlag: 1124-1140.

Kraus, M. and M. Strengert (2007). "Depth-of-Field Rendering by Pyramidal Image Processing." Eurographics **26**(3).

Krawczyk, G., et al. (2005). Perceptual Effects in Real-time Tone Mapping. SCCG

Lee, S., et al. (2009). "Depth-of-field rendering with multiview synthesis." ACM Trans. Graph. **28**(5): 1-6.

Lee, S., et al. (2009). "Depth-of-field rendering with multiview synthesis." ACM Trans. Graph. **28**(5): 1-6.

Lee, S., et al. (2009). "Real-Time Depth-of-Field Rendering Using Anisotropically Filtered Mipmap Interpolation." IEEE Transactions on Visualization and Computer Graphics **15**(3): 453-464.

Lefohn, A. and J. Owens (2006). Interactive depth of field using simulated diffusion.

Lengyel, E. (2010). Motion Blur and the Velocity-Depth-Gradient Buffer. Game Engine Gems 1.

Luksch, C. (2007) Realtime HDR Rendering.

Marsh, D. (2001). "Temporal Rate Conversion." from http://msdn.microsoft.com/en-us/windows/hardware/gg463407.aspx.

Maughan, C. (2001). Texture Masking for Faster Lens Flare. Game Programming Gems 2.

Mittring, M. (2007). Finding next gen: CryEngine 2. ACM SIGGRAPH 2007 courses. San Diego, California, ACM: 97-121.

Mittring, M. (2009). A Bit More Deferred - CryEngine 3. Triangle Game Conference.

Oat, C. (2003). Real-Time 3D Scene Post-processing. Game Developer's Conference Europe.

Pixar (2008). The Imperfect Lens: Creating the Look of WALL-E. WALL-E Special Edition DVD Extras.

Potmesil, M. and I. Chakravarty (1981). A lens and aperture camera model for synthetic image generation. Proceedings of the 8th annual conference on Computer graphics and interactive techniques. Dallas, Texas, USA, ACM: 297-305.

Python, M. (1970). The Royal Society For Putting Things On Top Of Other Things. Monty Python's Flying Circus.

Reinhard, E., et al. (2002). Photographic tone reproduction for digital images. Proceedings of the 29th annual conference on Computer graphics and interactive techniques. San Antonio, Texas, ACM: 267-276.

Riguer, G., et al. (2003). Real-Time Depth of Field Simulation. Shader X2 - Shader Programming Tips and Tricks with DirectX 9.

Rokita, P. (1996). "Generating Depth-of-Field Effects in Virtual Reality Applications." IEEE Comput. Graph. Appl. **16**(2): 18-21.

Rokita, P. (1996). "Generating Depth-of-Field Effects in Virtual Reality Applications." IEEE Comput. Graph. Appl. **16**(2): 18-21.

Selan, J. (2005). Using Lookup Tables to Accelerate Color Transformations. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.

Smith, A. R. (1995). Gamma Correction.

Wenzel, C. (2005). Far Cry and DirectX. Game Developers Conference.

Zhou, T., et al. (2007). "Accurate Depth of Field Simulation in Real Time." Computer Graphics Forum.

APPENDIX A: BRANCH-FREE ERROR FUNCTION

```
float erf(float x)
{
    float Correction = 1.0;
    if(x < 0)
    {
        x = -x;
        Correction = -1.0;
    }
    float a1 = 0.278393;
    float a2 = 0.230389;
    float a3 = 0.000972;
    float a4 = 0.078108;
    float Denominator = 1.0 + a1*x + a2*pow(x, 2) +
a3*pow(x, 3) + a4*pow(x, 4);
    float Result = 1.0 - (1.0 / pow(Denominator,4));
    return Result * Correction;
}
```

BIOGRAPHY OF THE AUTHOR

Sherief Farouk was born in Cairo, Egypt on June 12, 1986. He attended high school in Heliopolis, Cairo and later attended the Arab Academy for Science and Technology and graduated in 2009 with a Bachelor's degree in computer science. He absolutely loathes speaking about himself in the third person and his favorite animal is the sloth. Sherief is a candidate for the Master of Science degree in Computer Science from the University of Maine in May 2013.