Deferred shading and postprocessing
for each object in the scene
  for each triangle in this object
    for each fragment f in this triangle

      \texttt{gl\_FragColor = shade(f)}

      \texttt{if (depth of f < depthbuffer[x, y])}
      \texttt{  framebuffer[x, y] = gl\_FragColor}
      \texttt{  depthbuffer[x, y] = depth of f}

    \texttt{end if}

  end for
end for
Problem: Overdraw
Missed opportunity: spatial processing

Fragments cannot talk to each other
  • a fundamental constraint for performance

Many interesting effects depend on neighborhood and geometry
  • bloom
  • ambient occlusion
  • motion blur
  • depth of field
  • edge-related rendering effects
Buffers galore (from Eric Haines)

A-buffer - Carpenter, 1984
G-buffer - Saito & Takahashi, 1991
M-buffer - Schneider & Rossignac, 1995
P-buffer - Yuan & Sun, 1997
T-buffer - Hsiung, Thibadeau & Wu, 1990
W-buffer - 3dfx, 1996?
Z-buffer - Catmull, 1973 (?)
ZZ-buffer - Salesin & Stolfi, 1989

Accumulation Buffer - Haeberli & Akeley, 1990
Area Sampling Buffer - Sung, 1992
Back Buffer - Baum, Cohen, Wallace & Greenberg, 1986
Close Objects Buffer - Telea & van Overveld, 1997
Color Buffer
Compositing Buffer - Lau & Wiseman, 1994
Cross Scan Buffer - Tanaka & Takahashi, 1994
Delta Z Buffer - Yamamoto, 1991
Depth Buffer - 1984
Depth-Interval Buffer - Rossignac & Wu, 1989
Double Buffer - 1993

Escape Buffer - Hepting & Hart, 1995
Frame Buffer - Kajiya, Sutherland & Cheadle, 1975
Hierarchical Z-Buffer - Greene, 1993
Item Buffer - Weghorst, Hooper & Greenberg, 1984
Light Buffer - Haines & Greenberg, 1986
Mesh Buffer - Deering, 1995
Normal Buffer - Curington, 1985
Picture Buffer - Ollis & Borgwardt, 1988
Pixel Buffer - Peachey, 1987
Ray Distribution Buffer - Shinya, 1994
Ray-Z-Buffer - Lamparter, Muller & Winckler, 1990
Refreshing Buffer - Basil, 1977
Sample Buffer - Ke & Change, 1993
Shadow Buffer - GIMP, 1999
Sheet Buffer - Mueller & Crawfis, 1998
Stencil Buffer - 1992
Super Buffer - Gharachorloo & Pottle, 1985
Super-Plane Buffer - Zhou & Peng, 1992
Triple Buffer
Video Buffer - Scherson & Punte, 1987
Volume Buffer - Sramek & Kaufman, 1999
Deferred shading approach

**First render pass**
- draw all geometry
- compute material- and geometry-related inputs to lighting model
- don’t compute lighting
- write lighting inputs to intermediate buffer

**Second render pass**
- don’t draw geometry
- use stored inputs to compute lighting
- write to output

**Post-processing pass** (optional, can also be used with fwd. rendering)
- process final image to produce output pixels
Deferred shading step 1

for each object in the scene
  for each triangle in this object
    for each fragment f in this triangle

      \texttt{gl\_FragColor} = \text{material properties of f}
      \text{if (depth of f < depthbuffer[x, y])}
        \texttt{gbuffer[x, y] = gl\_FragColor}
        \texttt{depthbuffer[x, y] = depth of f}
      \text{end if}

    end for
  end for
end for
First pass: output just the materials
for each fragment f in the gbuffer
    framebuffer[x, y] = shade(f)
end for

Key improvement: shade(f) only executed for visible fragments.

Output is the same →
for each object in the scene  
   for each triangle in this object  
      for each fragment f in this triangle  
         gl_FragColor = material properties of f  
         if (depth of f < depthbuffer[x, y])  
            gbuffer[x, y] = gl_FragColor  
            depthbuffer[x, y] = depth of f  
         end if  
      end for  
   end for  
end for  

for each fragment f in the gbuffer  
   framebuffer[x, y] = shade (f)  
end for
Shader which computes lighting based on g-buffer: has code for all material/lighting models in a single huge shader.

```c
shade (f) {
    result = 0;
    if (f is Lambertian) {
        for each light
            result += (n . l) * diffuse;
        end for
    } else if (f is Blinn-Phong) {
        ...
    } else if (f is ...) {
        ...
    } else if (f is ...) {
        ...
    }
    return result;
}
```
Übershader inputs

Need access to all parameters of the material for the current fragment:

- Blinn-Phong: $kd, ks, n$
- Microfacet: $kd, ks, alpha$
- etc.

Also need fragment position and surface normal

Solution: write all that out from the material shaders:

```cpp
{outputs} = {f.material, f.position, f.normal}
if (depth of f < depthbuffer[x, y])
    gbuffer[x, y] = {outputs}
    depthbuffer[x, y] = depth of f
end if
```
G-buffer: multiple textures

- Material properties
- Position
- Normal
Power of Deferred Shading

Can do any image processing between step 1 and step 2!

- Recall: step 1 = fill g-buffer, step 2 = light/shade
- Could add a step 1.5 to filter the g-buffer

Examples:

- screen-space ambient occlusion
- silhouette detection for artistic rendering
Amient occlusion
Modeling flare in the eye

Figure 14: Two highway scenes before and after the scotopic glare algorithm. The orientation of the headlights is made obvious by the degree of glare.

Figure 15: An indoor simulation before and after the mesopic glare algorithm.

Figure 16: The Sun showing through leaves before and after the photopic glare algorithm. The location of the Sun is obvious only after the glare is added. Note that there is no lenticular halo because the pupil of the viewer is contracted.

Our approach has been to model the physical effects, primarily caused by the interaction of light rays and the physiology of the human eye. For many years, researchers in optics, psychophysics, and illumination engineering have attempted to determine the mechanisms behind glare, and to quantify the effects of glare on viewers. A camera lens filter that mimics the underlying mechanisms of glare in human vision has recently been developed, and had better results than conventional glare filters for some effects [1].

Glare effects can be subdivided into two major components: flare and bloom. Flare is composed of a lenticular halo and a ciliary corona (Figure 2), and is primarily caused by the lens [29]. Bloom is caused by scattering from three parts of the visual system: the cornea, lens and retina (Figure 3). The lenticular halo, ciliary corona, and bloom are the dominant contributing factors to glare effects and greatly affect our perception of the brightnesses of light sources.

Rays of the ciliary corona appear as radial streaks emanating from the center of the source. Similar ray patterns associated with other coronas have been studied by physicists and are caused by random fluctuations in refractive index of the ocular media [15].

The lenticular halo is observed as a set of colored, concentric rings, surrounding the light source and distal to the ciliary corona. The somewhat irregular rings are composed of radial segments, where the color of each segment of the ray varies with its distance from the source. The apparent size of the halo is constant and independent of the distance between the observer and the source. This phenomenon is caused by the radial fibers of the crystalline structure of the lens [29, 15].

Bloom, frequently referred to as "veiling luminance" is the "glow" around bright objects. Bloom causes a reduction in contrast that interferes with the visibility of nearby objects, such as the night-time view of the grill between two car headlamps. Bloom is caused by stray light attributed to scatter from three portions of the eye: the cornea, the crystalline lens, and the retina, all with approximately equal contributions [33].

The physiology of the eye and the resultant physical effects are explained in greater detail in the following section. In Section 3 we develop the quantitative aspects of this glare in terms of the point-spread function of the human eye and present an algorithm for generating the digital flare filter that approximates the point spread function. In Section 4 we describe a perceptual experiment which verifies incoming rays.
Modeling flare in a camera lens

Abstract

Lens flare is caused by light passing through a photographic lens system in an unintended way. Often considered a degrading artifact, it has become a crucial component for realistic imagery and an artistic means that can even lead to an increased perceived brightness. So far, only costly offline processes allowed for convincing simulations of the complex light interactions. In this paper, we present a novel method to interactively compute physically-plausible flare renderings for photographic lenses. The underlying model covers many components that are important for realism, such as imperfections, chromatic and geometric lens aberrations, and antireflective lens coatings. Various acceleration strategies allow for a performance/quality tradeoff, making our technique applicable both in real-time applications and in high-quality production rendering. We further outline artistic extensions to our system.

CR Categories:
I.3.3 [Computer Graphics]: Image Generation

Keywords:
Lens flare, Real-time rendering

Contact to authors:
{hullin, hpseidel}@mpi-inf.mpg.de
elmar.eisemann@telecom-paristech.fr
sungkil@skku.edu (Corresponding author)

1 Introduction

Lens flare is an effect caused by light passing through a photographic lens in any other way than the one intended by design—most importantly through interreflection between optical elements (ghosting). Flare becomes most prominent when a small number of very bright lights is present in a scene. In traditional photography and cinematography, lens flare is considered a degrading artifact and therefore undesired. Among the measures to reduce stray light in an optical system are optimized barrel designs, anti-reflective coatings, and lens hoods.

On the other hand, flare-like effects are often used deliberately to suggest the presence of very bright light sources, hence increasing the perceived realism. In fact, nowadays the use of lens flare is every bit as popular in games as it is in image and video editing. For the production of computer-generated movies, great effort has been taken to model cinema lenses with all their physical flaws and limitations [Pixar 2008].

The problem of rendering lens flare has been approached from two ends. A very simple and efficient, but not quite accurate, technique is the use of static textures (starbursts, circles, and rings) that move according to the position of the light source, and are composited additively to the base image. Flares generated from texture billboards can look convincing in many situations, yet they fail to capture the intricate dynamics and variations of real lens flare.

At the other end of the scale, sophisticated techniques have been demonstrated that involve ray or path tracing through a virtual lens with all of its optical elements. The results are near-accurate but very costly to compute, with typical rendering times in the order of several hours per frame on a current desktop computer. Furthermore, many samples end up being blocked in the lens system, which wastes much of the computation time and leads to slow convergence. Also, the solution only holds within the limits of geometric optics. Some phenomena encountered in real lens flares, however, are caused by wave-optical effects. Integrating them into a ray-optical framework is by no means trivial and further increases the computational cost.
Color grading

The Matrix (Warner Bros.); comparison: Digital Media Academy
Summary: Deferred Shading

Pros
- Store everything you need in 1st pass
  - normals, diffuse, specular, positions,...
  - G-buffer
- After z-buffer, can shade only what is visible

Cons:
- transparency (only get one fragment per pixel)
- antialiasing (multisample AA not easy to adapt)

Engines (now)
- Cry Engine / Amazon Lumberyard
- Unreal Engine 3
- Unity 5
How to Fill the Buffers?

- Material properties
- Position
- Normal
Single Render Target

• Requires N passes through scene:
  - `glClear()`: writes every pixel, is fairly expensive
  - Make hardware transform all the vertices again
  - Expensive if you are:
    – animating in your vertex shader
    – have subdivision surfaces generating lots of polygons

• Requires N different trivial shaders to output each variable you need (annoying)
Single Render Target

```c
 glClear();
renderScene(textureShader);

    gl_FragColor = someTexture;

```

```c
 glClear();
renderScene(yellowShader);

    gl_FragColor = yellow;

```

```c
 glClear();
renderScene(normalShader);

    gl_FragColor = normal;
```
Multiple Render Targets

- Requires single pass through scene:
  - Single glClear() call
  - Single transformation of geometry
  - Single shader to output everything
- Fragment shader outputs multiple colors
  - Each to a different color attachment of a framebuffer object
- Different cards/drivers support different # of render targets
  - We use 4, which is widely supported
Multiple Render Targets

glClear();
renderScene(everythingShader);

gl_FragData[0] = someTexture;
gl_FragData[1] = yellow;
gl_FragData[2] = normal;

- Easier and more efficient than multiple passes
- Many “multi-pass” techniques in old papers can be done in a single pass now by using MRTs
Limitations of Deferred Shading

• Each pixel in the g-buffer can only store material and surface info for a single surface.
  – So, blending/transparency is difficult
  – Also, antialiasing is difficult

• For transparency: a “hybrid” renderer
  – Uses deferred shading for opaque objects, forward shading for translucent objects
  – Allows translucent geometry to know about opaque geometry behind it.

• For antialiasing: smart blurring
  – use
Hybrid Rendering

Note how the water effect fades out in the shallows — access to depth of ground.

Image credit: random guy on internet: http://vimeo.com/14337919
Translucency cont.

- Hybrid renderer is the most practical solution.
- Can also do “depth peeling”:
  - Render and shade each layer of translucency using deferred shading
  - Sort of like a “deep” g-buffer
  - Gets very expensive very quickly

- Will probably implement a hybrid renderer to support translucent particle systems later on
Antialiasing

Single shading sample per pixel

Reconstruct by blending nearby samples

- select them by looking for edges
  (Morphological AA [Reshetov 09])

- learn about edges using multisample depth
  (Subpixel Reconstruction AA [Chajdas et al. 11])
Figure 9. Processing textures for the Edgar model: (a)–original aliased image; (d)–antialiased image, processed with MLAA; (b,c)–enlarged regions of left and right images; (e,f,g)–visualization of pixels processed with MLAA at different zoom levels. Pixels belonging to horizontal shapes are marked with green, vertical with red color. Pixels, which are included into horizontal and vertical shapes simultaneously, are shown as blue. Note that aliased pixels (top left) were unintentionally blurred when this paper was created (both in electronic and paper form). [Reshetov 09]
SRAA benefits shading-bound applications. For example, our im-
the new algorithm can better respect geometric boundaries and has
into an existing renderer without modifying the shaders. In this

Abstract

Marvel Ultimate Alliance 2 (see Figure 8), courtesy of Vicarious Visions. Shown: 4 geometric samples/pixel, planes+normals depth metric.

Figure 1:

Subpixel Reconstruction Antialiasing produces an image approaching
a regular 1

Figure 3:

similar time

Figure 4:

← Similar Time →

← Similar Quality →

(a) 1× Shading + Box (poor, fast)
(b) NEW: 1× Shading + SRAA (good, fast)
(c) 16× Shading + Box (good, slow)

[Chajdas et al. 11]
Output: the shaded scene
Drawbacks of Forward Shading

• If $\text{shade}(f)$ is very expensive
  – e.g., many lights, shadow maps, complex shaders
  – overdraw by closer geometry wastes work on each fragment
Drawbacks of Forward Shading

• Many other complex effects: Image processing effects
  – tonemapping, screen-space ambient occlusion, bloom, toon shaders etc., are very expensive
  • Must read rendered image back from the framebuffer, or do entire pass through the scene geometry to render it to a FBO (frame buffer object)
Overdraw: Real Example

(Battlefield 3)
Deferred Shading Step 1

Code structure is nearly the same as forward shading, with one key difference:

```markdown
for each object in the scene
    for each triangle in this object
        for each fragment f in this triangle
            gl_FragColor = material properties of f
            if (depth of f < depthbuffer[x, y])
                gbuffer[x, y] = gl_FragColor
                depthbuffer[x, y] = depth of f
            end if
        end for
    end for
end for
```