Re: Deep G-Buffers for Stable Global Illumination Approximation

Ferit Tohidi Far

January 20, 2019

Abstract

G-buffers can be used to efficiently render images with large amounts of light sources. This is possible thanks to a process called "deferred rendering". Using g-buffers, we are only able to compute local illumination. By using deep g-buffers instead we can approximate global illumination, which is way more efficient than traditional global illumination methods like pathtracing, while of course not being physically accurate. We can make up for it, though, by also approximating visual effects like ambient occlusion, color bleeding, reflections, depth of field and motion blur to create an acceptable result.

Keywords *nvidia, g-buffer, deep g-buffer, pathtracing, global illumination approximation, deferred shading, deferred rendering*

Contents

1 Global illumination **2**
- 1.1 Physically correct methods . . . 2
  - 1.1.1 Pathtracing . . . 2
- 1.2 Computational difficulties of physically correct methods . . . 2

2 Deferred rendering **2**
- 2.1 How deferred rendering handles lighting more efficiently . . . 2
- 2.2 Deferred shading . . . 3

3 Geometry-buffer (g-buffer) **3**
- 3.1 Frame-buffer . . . 3
- 3.2 Z-buffer . . . 3
- 3.3 Normal-buffer . . . 3
- 3.4 Computing local illumination using g-buffers . . . 3

4 Visual effects **4**
- 4.1 Ambient occlusion . . . 4
- 4.2 Color bleeding . . . 5
- 4.3 Soft shadows . . . 5
- 4.4 Transparency . . . 5
- 4.5 Reflection . . . 5
- 4.6 Depth of field . . . 5
- 4.7 Motion blur (in interactive applications) . . . 5

5 Deep g-buffer **5**
- 5.1 Concept . . . 5
- 5.2 How deep g-buffers approximate visual effects . . . 5

6 Performance and output comparison **5**
- 6.1 Deep g-buffers vs pathtracing . . . 5

1 Global illumination

Global illumination is a lighting effect that is achieved by not only computing direct light, but also indirect light, meaning that it is neccesary to take into account how light reflects and carries information (in the most basic case: color).

1.1 Physically correct methods

In order to generate physically correct images, which is a requirement for creating photorealistic images, we need to solve the rendering equation

$$L_o(\omega) = L_e(\omega) + \int_\Omega f(\omega, \omega')L_i(\omega')cos(n, \omega')\partial\omega'$$

where

$$\begin{aligned}
&L_o(\omega) \text{ is the outgoing light in direction } \omega,\\
&L_e(\omega) \text{ is the emmited light in direction } \omega,\\
&f(\omega, \omega') \text{ is the BRDF}^1,\\
&L_i(\omega') \text{ is the incoming light from direction } \omega'\\
&\text{and } cos(n, \omega') \text{ is lambert reflectance}^2\ .
\end{aligned}$$

The most popular method for achieving this is pathtracing [PHC14].

1.1.1 Pathtracing Pathtracing solves the rendering equation by first sending camera rays through each individual pixel of the image plane and then tracing the ray back to the light source. If the lamp is hit the pixel gets painted painted with color, else black. Direct consequences of this are soft shadows and ambient occlusion. A maximum hop number caps the amount of times a ray is able to reflect. A hop number larger than 3 (3 in most cases, but it is dependent on the complexity of the scene) allows for global illumination. The reflections and refractions are essentially determined by the BRDF, which not only means that objects can be transparent, but we also get caustics[3]. With each surface a ray hits it carries information from that surface, e.g. its color, and reflects it onto the next surface it hits. This causes color bleeding. What all these visual effects describe are is explained in a later section.

1.2 Computational difficulties of physically correct methods

Since we have to take into account thousands of samples of every ray of light with its reflections, the computational difficulty becomes apparent [FP]. Because of this, it is nearly impossible to achieve real time rendering using physically correct methods on an average system. This forces game engines to stick to approximating global illumination and visual effects since it is way more efficient to compute. Add multiple light sources to the scene, and the computational requirements are going to blow up. Deferred rendering, specifically deferred shading, counters this problem at the cost of having to compute transparency using depth peeling.

2 Deferred rendering

2.1 How deferred rendering handles lighting more efficiently

Graphics pipelines describe each step that has to be done in order to render an image. Within a pipeline it is possible in some cases to defer steps to a later stage. Applications that rely on graphics pipelines like that of OpenGL can profit from this. The goal of deferred rendering is to postpone the shading stage. Instead of shading right away, we compute necessary geometry buffers (g-buffers) and cache them for later use. In OpenGL applications this is especially a good idea because in the graphics pipeline each pixel has to go through the vertex- and fragment-shader.... .For all practical purposes, g-buffers have to at least consist of a frame-buffer, a normal-buffer and a z-buffer. Using only these g-buffers, it is possible to compute the shading.

[1] The bidirectional random distribution function basically describes the reflection/refraction of a ray on surface.
[2] Lambert reflectance describes the attentuation of light on diffuse objects based on the lights incident angle.
[3] Caustics are areas with concentrated light. This happens due to light refracting.

2.2 Deferred shading

3 Geometry-buffer (g-buffer)

Each geometry-buffer stores information of some sort for each individual pixel, meaning that they are all two-dimensional arrays.

3.1 Frame-buffer

Color-values of fragments are stored in the frame-buffer. It basically stores the rendered image without fragment-shading[4] applied to it.

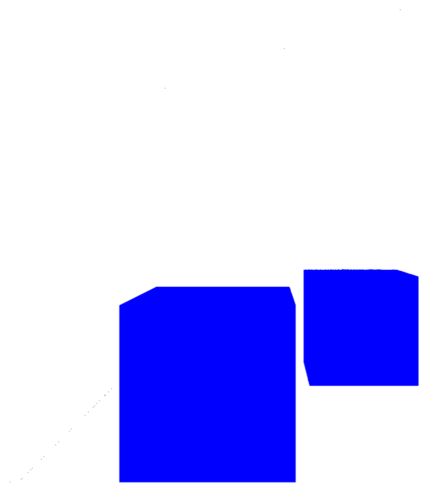

Figure 1: Example of a frame-buffer (also called image-buffer). It shows a cornell box with white walls containing two blue cuboids.

3.2 Z-buffer

The z-buffer stores depth values of fragments. These are needed to determine which surfaces are closest and visible to the camera. If two different fragments[5] have the same x and y coordinates in screenspace[7], then the fragment with the smaller z-value is supposed to be in front of the other. This buffer is also used for screenspace visual effects like screen space ambient occlusion.

3.3 Normal-buffer

The normal-buffer stores surface-normals that are mostly used to determine reflection and refraction directions. They are also used for light attentuation, since they determine the cosine of the surface and the incident light and a larger cosine means lighter light and vice versa (lambert reflectance). Note that there are more possible buffers to choose from, but the three that were mentioned are the most essential in every g-buffer.

3.4 Computing local illumination using g-buffers

After having collected all our g-buffers we can now work on illumination. To do this we need to define some light sources. We distinguish between three types of light: Point-lights, spot-lights and directional-lights [FP]. The simplest one is directional-light. We simply specify an origin in 3d space from which the light rays are sent. A point-light is a directional-light with a radius.

[4]Fragment-shading is ...
[5]A fragment is a point on a surface in 3d space (worldspace[6])
[6]Worldspace is ...
[7]Screenspace is ...

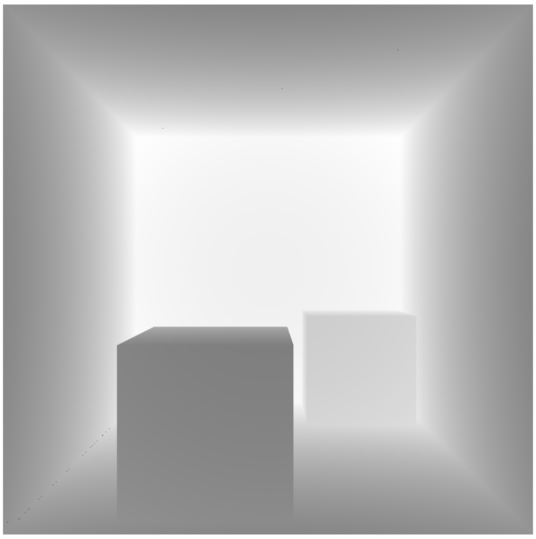

Figure 2: Example of a z-buffer. Since the z-buffer only stores distances as float values instead of actual colors they are interpreted as the grayscale value deduced by dividing each distance by the maximum distance from the cameras point of view.

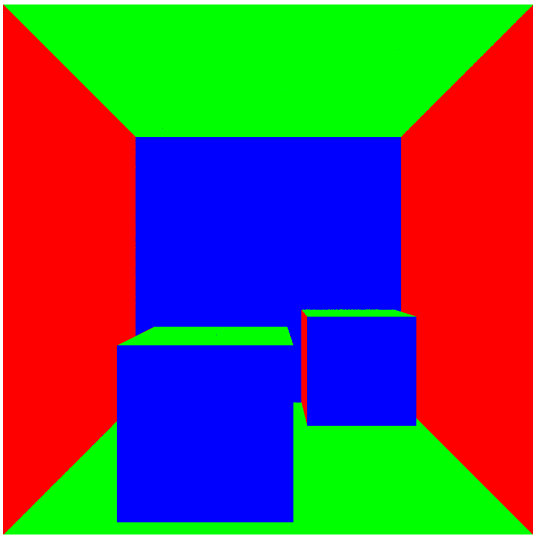

Figure 3: Example of a normal-buffer. Normal vectors are usually normalized, meaning their values range from 0 to 1. Multiplying them with 255 gives us an RGB color. Since negative normals would cause negative RGB values we just take the absolute values.

4 Visual effects

The following are visual effects that are sought after, but some of them are hard or impossible to achieve physically correct without using computationally expensive methods. When using pathtracing, we get most of the following effects for free:

4.1 Ambient occlusion

Ambient occlusion essentially describes how much shading the "in-betweens" of a 3d object gets. This effect can be efficiently approximated by using a method called - ironically - screen space ambient occlusion

(SSAO). This method basically runs an edge detector over the z-buffer and paints those edges black. Since it only runs over the z-buffer it is considered screen space.

4.2 Color bleeding

Color bleeding happens when light directs information from one hit-surface to another. Let A and B be objects. If A reflects light onto B and A's surface is blue, then B will also appear to be slightly blue on the reflected area. To have this happen it is obviously necessary to trace rays of some sort. This can be approximated, though, using ...

4.3 Soft shadows

We can easily compute hard shadows using shadow mapping. This is done by projecting the scene from the light source's point of view and then projecting the scene from the camera's point of view while only actually painting the points with their respective colors if they are hit by light, else they are painted black. To get soft shadows, the points in shade simply get blended together with their surrounding points.

4.4 Transparency

A quick method to achieve transparency is depth peeling. To do this we need two g-buffers at a time ...[Eve].

4.5 Reflection

Reflection ...

4.6 Depth of field

Depth of field ...

4.7 Motion blur (in interactive applications)

Motion blur ...

5 Deep g-buffer

5.1 Concept

Deep g-buffers use a concept similar to depth peeling. Instead of storing information about the closest surface, in an n layer deep g-buffer we also store information about the n-closest surface [MM16].

5.2 How deep g-buffers approximate visual effects

6 Performance and output comparison

6.1 Deep g-buffers vs pathtracing

References

[Eve] Cass Everitt. "Interactive Order-Independent Transparency". In: *NVIDIA OpenGL Applications Engineering* ().

[FP] Francisco Fonseca Fabio Policarpo. "Deferred Shading Tutorial". In: ().

[MM16] D. Nowrouzezahrai D. Luebke M. Mara M. McGuire. "Deep G-Buffers for Stable Global Illumination Approximation". In: *High Performance Graphics* (2016).

[PHC14] Wojciech Jarosz Per H. Christensen. "The Path to Path-Traced Movies". In: *Foundations and Trends R in Computer Graphics and Vision* 10 (2014).