Deep G-Buffers for Stable Global Illumination Approximation

Ferit Tohidi Far

March 4, 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 not able to approximate global illumination very well since we only have a single layer of information of the scene, meaning it is necessary to apply further techniques to mimic visual effects that would have otherwise been exerted in a globally illuminated scene. With multiple depth-layers of information, we are able to make a good approximation of the illumination as opposed to using just a single layer. Depth-peeling while collecting G-Buffers for each layer would deliver the best approximation, but would also be more difficult to compute. As a trade-off, we will be using two layers of information and store them in a Deep G-Buffer, since two layers have proven to be sufficient. This way we can achieve real-time global illumination on moderate hardware.

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

Contents

1 Global illumination **2**
- 1.1 Physically based methods . . . 2
  - 1.1.1 Pathtracing . . . 2
  - 1.1.2 Radiosity . . . 3
- 1.2 Computational difficulties of physically based methods . . . 3

2 Traditional rendering (rasterization) **3**
- 2.1 Forward rendering . . . 4
- 2.2 Deferred rendering . . . 4
  - 2.2.1 Geometry-pass . . . 4
  - 2.2.2 Lighting pass . . . 5

3 Visual effects **6**
- 3.1 Ambient occlusion . . . 6
- 3.2 Color bleeding . . . 6
- 3.3 Soft shadows . . . 7
- 3.4 Reflection . . . 7

4 Deep g-buffer **7**
- 4.1 Generating a 2-layer deep g-buffer . . . 8
- 4.2 Global illumination using 2-layer deep g-buffer . . . 8
- 4.3 Double screen space ambient occlusion using 2-layer deep g-buffer . . . 10
- 4.4 Screen space radiosity using 2-layer deep g-buffer . . . 10
- 4.5 Screen space reflections using 2-layer deep g-buffer . . . 11

5 Conclusion **11**

1 Global illumination

In order to understand why computing global illumination is such a big deal, we first need to set the stage by introducing the essentials of rendering.

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). On the contrary, local illumination is computed by only considering direct light, meaning that the resulting image will lack details and visual effects that would otherwise convey realism. These visual effects are ambient occlusion, color bleeding, soft shadows and reflections. Further descriptions are provided in section 4.

1.1 Physically based methods

In order to generate physically based lighting, we need to solve the rendering equation for every single ray of light

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

where

$L_o(\omega)$ is the outgoing light in direction ω,
$L_e(\omega)$ is the emmited light in direction ω,
$f(\omega, \omega')$ is the BRDF,
$L_i(\omega')$ is the incoming light from direction ω'
and $cos(n, \omega')$ is lambert reflectance .

The most popular method for approximating this integral is pathtracing [CJ16]. Radiosity also solves the rendering equation but only for perfectly diffuse surfaces.

We first send a camera ray through each individual pixel of the image plane and then trace the ray back to a light-source. If a light-source is hit, the pixel gets painted with color, else black. This process is repeated many times over for each pixel and then averaged. A maximum hop number caps the amount of times a ray is able to reflect. A hop number larger than 1 (3 in most simple cases is sufficient, but this is also dependent on the complexity of the scene) allows for some global illumination. Direct consequences of this are soft shadows and ambient occlusion. The reflections and refractions are determined by the BRDF, which essentially determines surface interaction of incoming rays. Through this it is possible to attain caustics and transparency. 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. Since each pixel is possibly sampled hundreds of thousands of times to reduce the noise that is induced by diffuse reflections, pathtracing may result in a photorealistic render (figure 1).

Figure 1: A photorealistic computer generated image using pathtracing.

1.1.2 Radiosity is a diffuse global illumination method, meaning it is only meant to be used for diffuse surfaces, as it is has been observed to be one of their physical properties. In simple terms, it solves the rendering equation by splitting the objects in cameraspace into patches and treats each of those as a light emitter and receiver. We iteratively check which patches are exposed to light (an initial light-source is required) and adjust their emmitances accordingly until convergence or until we are satisfied with the look of the render (see figure 2). It takes a while to compute the illumination, but since it is not dependent on the camera view or location, the illumination will stay the same unless an object is moved. Games often preprocess radiosity on scenes with little object movement as it then becomes a quick way of achieving diffuse global illumination [JFH] [CG85].

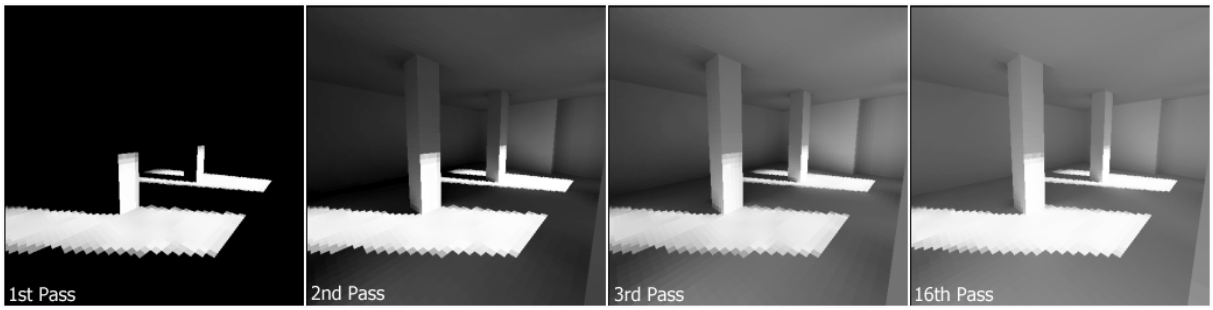


Figure 2: How radiosity globally illuminates a scene. With each iteration there are more light emitters, meaning that there are more possible patches that can be illuminated. In the first iteration, the only light-emitters are the windows (not visible in the frame). After 16 iterations the scene is sufficiently saturated.

1.2 Computational difficulties of physically based methods

With pathtracing we have to take into account thousands of samples of every ray of light with its reflections, so the computational difficulty should become apparent [FP].

Radiosity requires a large amount of patches and a moderate amount of iterations to deliver good results. It can be preprocessed and quickly loaded to illuminate a scene, but would require large amounts of memory for non-static scenes.

Since physics simulations/animations or games practically always have moving objects or lights per frame, the whole illumination would have to be recomputed for both pathtracing as well as radiosity. Because of this, it is impossible to achieve real time rendering using physically based methods on an average system. Game engines are forced to stick to achieving global illumination and the visual effects it induces through the application of plausible-looking and efficiently computable techniques.

Lighting with traditional rendering can also be inefficient if many light sources are involved. This still proves to be challenge when forward rendering as described by modern graphics pipelines is applied. Deferred rendering counters this problem at the cost of having to compute transparency using depth-peeling and having no straight forward anti-aliasing.

2 Traditional rendering (rasterization)

Graphics pipelines describe each step that has to be taken in order to render an image. It is conventional to use GPU's for applications that rely heavily on rendering images. There is no universal graphics pipeline, because these are dependent on the GPU that is used, which means it is convenient that there exist API's like OpenGL that try to generalize the steps that need to be taken in order to render an image and map them to compatible GPU's. This means that most GPU dependent applications abide by the graphics pipeline as described by, for instance, OpenGL (figure 3).

2.1 Forward rendering

With forward rendering, for each object in the scene, we would compute its lighting and then render its fragments to the frame-buffer if they pass the z-test (if it is the current frontmost fragment). Since this is done for each object, it is likely that many fragments will overlap other fragments. However, we already computed all the lighting and shading for that fragment, so if it is not going to be visible anyway, than those computations were effectively wasted. Note that lighting has to be computed in worldspace before the rasterization stage (which projects the scene into cameraspace), so the z-test cannot be done before lighting is computed. This means that the more lights we add to the scene, the more lighting computations are going to be in vain.

2.2 Deferred rendering

The goal of deferred rendering is to defer lighting and shading calculations until after the frontmost fragments are determined, meaning that geometry and lighting computation are **decoupled**. This is done by computing necessary geometry buffers (g-buffers) in a first pass through the pipeline that is called "geometry pass" and rendering them into a texture-buffer of the GPU instead of to the screen. In the second pass (lighting pass) all lighting computations are performed using the previously computed geometry that is stored in the texture-buffer.

With forward rendering we would have to compute the lighting for every fragment of every object for all light-sources in a single pass. This means that the time complexity of computing forward lighting is $O(amount_{fragments} \cdot amount_{lights})$.

If we apply deffered rendering, however, we do not force ourselves to light every fragment as soon as it has been computed. Finding the closest fragment is called "solving the visibility problem". This is done during the geometry pass when filling the z-buffer with its respective z-values, which can be computed by performing perspective correct interpolation between the vertices of the current polygon (in most cases triangles). The z-buffer, along with all other buffers that are collected, are continuously updated if a closer fragment for the same pixel is found. Now all the fragments that we light are only going to be lighted once, which means for the time complexity that the lighting is done in $O(amount_{pixels} \cdot amount_{lights})$ (note that fragments are **potential** pixels).

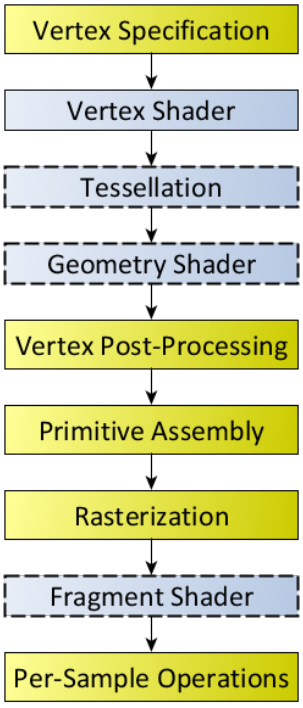


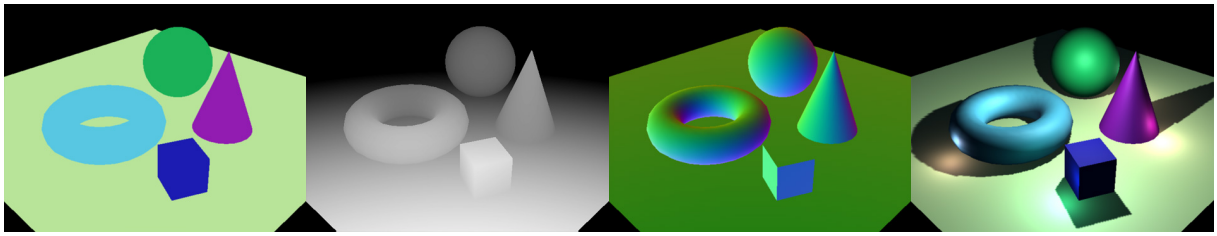
Figure 3: The OpenGL graphics pipeline. The steps within blue boxes are programmable.

Aside from saving some lighting computation for fragments, decoupling geometry and lighting means decoupling the loops in which geometry and light-sources are iterated. Effectively, the computational time drops from $O(amount_{objects} \cdot amount_{lights})$ to $O(amount_{objects} + amount_{lights})$. This is a huge performance gain which allows for more light-sources to be added at the expense of carrying around g-buffers. For all practical purposes, g-buffers have to at least consist of an albedo-buffer, a normal-buffer and a z-buffer. Using at least these g-buffers, it is possible to render an image with basic shading.

2.2.1 Geometry-pass Each geometry-buffer (g-buffer) stores information of some sort for each individual pixel, meaning that they are all two-dimensional arrays using the dimensions of the screen (screenspace). Note that there are more possible buffers to choose from (position-buffer, stencil-buffer, ...), but these three are the most essential:

2.2.1.1 Albedo-buffer

This buffer simply stores the color-values of frontmost fragments. Colors in computer graphics are referred to as "albedo".



(a) (b) (c) (d)

Figure 4: An albedo-buffer (a), a z-buffer (b), a normal-buffer (c) and a scene rendered using all of those buffers (d).

2.2.1.2 Z-buffer

The z-buffer stores depth values of frontmost fragments. These are needed to determine which surfaces are closest and visible to the camera. If two different fragments have the same x and y coordinates in screenspace, then the fragment with the smaller z-value is supposed to be in front of the other. This buffer is also used for depth-peeling and screenspace techniques like SSAO[1].

2.2.1.3 Normal-buffer

The normal-buffer stores frontmost surface-normals that are mostly used for shading and to determine reflection/refraction directions. They can also be used for light attentuation, since the dot product of a normalized surface-normal and the normalized direction of a light-ray is a criterion for light intensity. If the light-ray is orthogonal to the surface-normal, for instance, the dot product will equal zero, meaning that the light has maximum intensity (lambert reflectance).

2.2.2 Lighting pass For any illumination to happen we first need to define some light sources. The following are most commonly used[FP]:

- **Point-lights** are defined by an origin in worldspace and a radius (like a sphere of light).
- **Spot-lights**
- **Directional lights** can have a source that is infinitely far away, thus no fall-off has to be accounted for. The light basically shines in the same direction from everywhere.

At this point we have to choose a shading technique. The most popular ones are gouraud-shading and phong-shading.

Both first compute the normals for each vertex for each polygon in cameraspace. This can be done by averaging the surface normals of the polygons that share the vertex of which the normal is to be computed. Phong-shading uses its vertex-normals to interpolate the normals of each fragment within the polygons in the vertex-shader and computes shaded colors using some illumination-model, which requires the fragment-normals that were just computed, on each fragment in the fragment-shader.

Gouraud-shading is a fast way of computing shading since it only uses the illumination-model on each vertex of its polygons (which would be three times if the polygons are triangles) as opposed to applying the illumination-model on each fragment, as well as computing the normal for each fragment, as is done by phong-shading. On the downside, gouraud-shading may look edgy if the 3d models are made of few polygons, which would make phong-shading preferable in that case. This would conclude local illumination, which means we took care of direct lighting.

In the context of deferred shading, all of this is done in the second pass (the shading or lighting pass). By now it should become apparent why shading in a single pass is inefficient, since all of these illumination computations would be in vein if the visibility test does not pass anyway. The g-buffers that we collect in the first pass (the geometry pass) make this possible.

[1] Screen space ambient occlusion

3 Visual effects

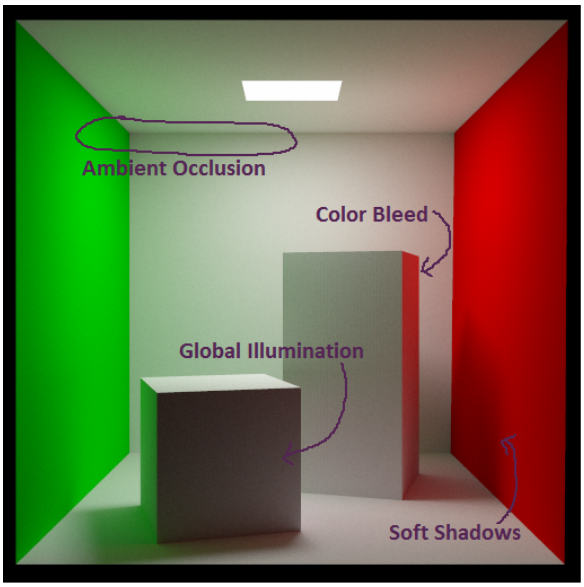


Figure 5: Diverse visual effects caused by global illumination inside a cornell box. The image was rendered using pathtracing.

The following are visual effects that are sought after, since they are fundamental to conveying realism. If the scene was globally illuminated, all of these visual effects would be visible in some way or another, so naturally the problem lies within simulating all of those visual effects efficiently.

Unfortunately, since with traditional rendering (rasterization) we mostly only compute local lighting, we are forced to apply further techniques to attain all of those visual effects in order to approximate a globally illuminatefd scene.

With pathtracing and radiosity, the simulation of all of those visual effects is build in within the rendering technique, though at the expense of reaching non-interactive framerates.

Next to the basic visual effects

3.1 Ambient occlusion

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

This method basically checks the surrounding z-values of each z-value of the z-buffer. Given a radius r and a sampling count N we can compute the ambient occlusion factor AO by checking N randomly picked surrounding z-values within r. In simple terms, if the majority (half or more) of the samples (the surrounding z-values) are smaller than the center z-value (the one we are computing AO for), then AO receives a value of 1, else a value smaller than 1 but larger or equal to 0. These float values can then be put in a texture-buffer to be used in post-processing or they can be used directly. To apply ambient occlusion to the final image, the AO values are multiplied with their respective pixels in the final image. Because $0 \leq AO \leq 1$ this can be seen as tweaking the intensity of the color on a specific pixel.

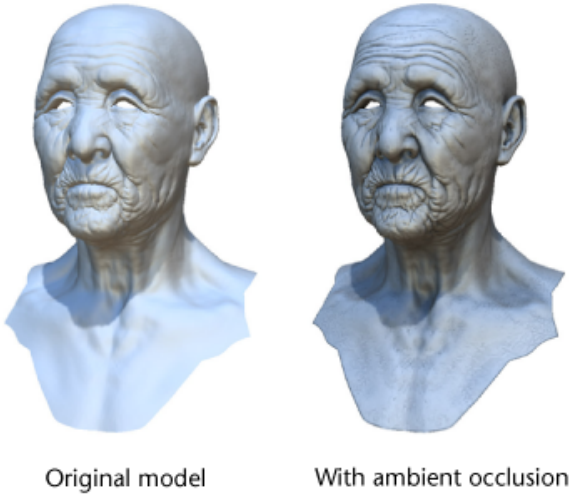


Figure 6: How ambient occlusion affects the realism of a rendered 3d model. The "in-betweens" of the model appear to have depth.

Since it only runs over the z-buffer it is considered screen space. Ambient occlusion (the visual effect) is an effect directly caused by global illumination, leading the method (ambient occlusion) to be considered an efficient way of faking global illumination.

3.2 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 would obviously be convenient to trace rays of some sort. Point-based color bleeding approximates this by using point cloud surfels with direct illumination values [Chr].

3.3 Soft shadows

We can easily compute hard shadows in screen space using shadow mapping. A shadow map is a buffer that contains the distances between the light source and all pixels (visible fragments) of the scene. Having created the shadow map we check if the entry in the z-buffer is larger than the entry in the shadow map. If so, then the fragment is in shadow and is painted black. If not, then the fragment keeps its color and we can proceed to compute its local illumination. This can be extended to soft shadows simply by blending together fragments in shadow with their surrounding fragments.

3.4 Reflection

Reflections can be efficiently computed. If the reflective surface is completely planar, then we can simply project the texture that is supposed to be reflected onto the surface that reflects it. In all other cases, we rely on cube-mapping (a form of environmental-mapping).

Creating a cube-map (figure 7) works like this: we define a cube with our camera in its center. Next, we treat each plane of the cube as an image plane that we render to, meaning we render six times in total, and save those inside a buffer. This buffer now has knowledge about the entire surrounding environment of the reflective object. When computing the color of a vertex of a reflective object, we use the direction vector from the camera to the vertex and use the vertex-normal to get the specular reflection direction, which will ultimately point at the spot on the cube-map that will be reflected. The issue with this is that it lacks the ability to self-reflect [MK].

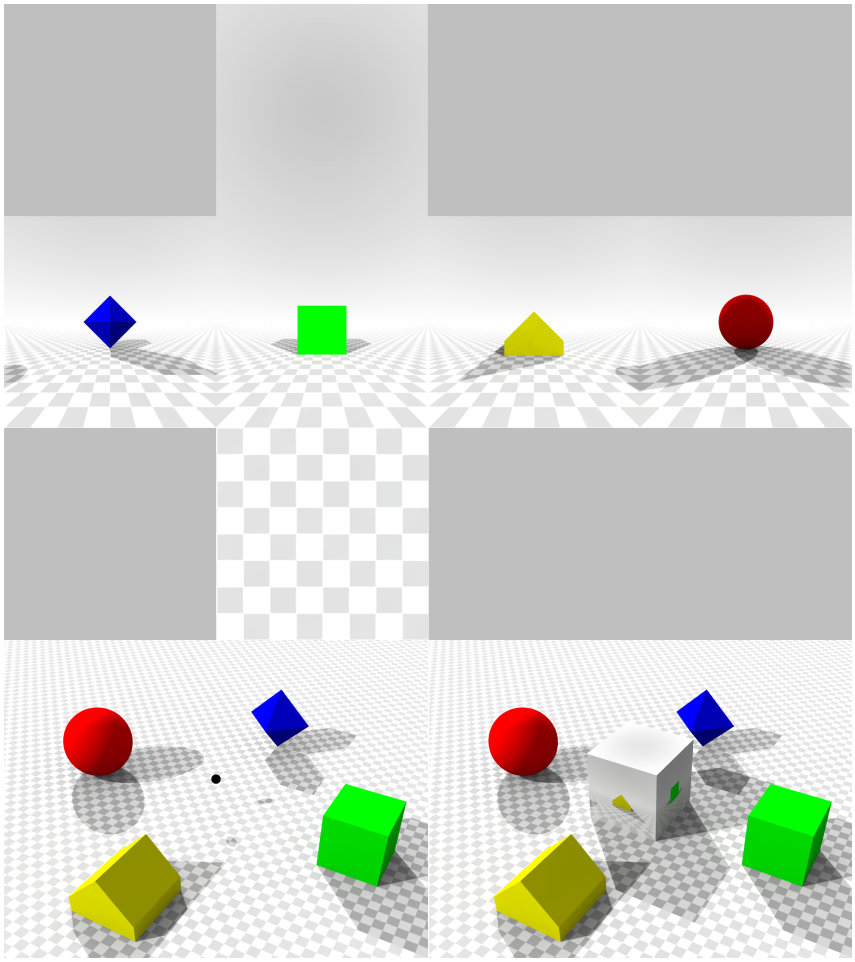


Figure 7: How a cube-map is rendered. The black sphere in the lower image represents the camera. The resulting six renders are put together in the cube-map in the upper image.

4 Deep g-buffer

Deep g-buffers make use of 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 [Mar+16].

Practical observations suggest that the second-closest surface is often not the second-most relevant for shading and visual effects [Mar+16]. To resolve this issue we rely on minimum depth separation, which essentially introduces a distance Δz that has to be exceeded when looking for the next-closest surface immediately after the current one. Setting $\Delta z = 0$ makes the generated 2-layer z-buffer of the deep g-buffer turn into a 2-layer k-buffer[2] [Bav+07].

[2]K-layer z-buffer that stores the z-values of the k-closest polygons in order

How to pick Δz ultimately depends on the depth complexity of the scene and has to be tweaked accordingly. If Δz is too large, it is possible that important visual information might be missed. On the other hand, if Δz is too small, it is possible that unnecessary information might be picked up. According to NVIDIA, $\Delta z = 50cm$ seems to consistently deliver stable results, at least in the renders done in their paper [Mar+16].

4.1 Generating a 2-layer deep g-buffer

NVIDIA proposes two ways of generating a 2-layer deep g-buffer, one of which is straight forward to help demonstrate the idea and one of which is efficient.

The first one is called "Strawman Two-Pass Generation Algorithm" and displayed in algorithm **??**. Z is the 2-layer z-buffer, $S(x, y, z)$ returns other g-buffers needed for shading and $T(tri)$ applies the transformation T to triangle tri (model-view-projection and skinning-transformations). In the first pass we simply collect the first layers g-buffers and in the second pass we peel away the first layers g-buffers to receive the second layer. Since fragments get discarded if their distance to the previous fragment is smaller than Δz the minimum depth seperation constraint is met.

This is essentially two-pass depth peeling, however, there is a more efficient algorithm which only requires a single pass displayed in **??**.

In order to save a pass over the geometry, we rely on an oracle that predicts the first layers z-buffer before it is even rendered. Using that prediction, we can compute fragments for the second layer by peeling the first layer. In total, there are four variants to go about this. t is the index of the frame.

1. **DELAY VARIANT** manages to predict the first layers z-buffer by adding a frame of latency, meaning that what is happening in the scene (animations, object movements and such) and what is being rendered on the screen is out of sync by one frame. This way, the next frames transformation T_{t+1} is known at render time, enabling us to predict first layer z-buffers perfectly (since we do not really predict it anyway). The downside to this is, obviously, the frame latency.

2. **PREVIOUS VARIANT** approximates the first layer z-buffer by taking the previous frames first layer z-buffer. This will deliver good results if camera- and object-movement are minimal, but even then, the errors would appear in the second layer, which is not visible to begin with (excluding transparent objects). It does not guarantee minimum separation.

3. **PREDICT VARIANT** uses velocities from the underlying physics simulation or animation to predict the next transformation T_{t+1}. Alternatively, extrapolation over the change in vertices from the previous and current frame would also work. Accurate velocity prediction will deliver accurate results without latency. In the other case, drawbacks are equivalent to that of the *PREVIOUS VARIANT*

4. **REPROJECT VARIANT** performs the minimum separation test against the first layer z-buffer of the previous frame. The visibility test is done using screen coordinates and z-values computed using the vertices of the previous frame, meaning the visibility test is done in the "past". Like with *PREDICT VARIANT*, moving objects are a source of error, but the velocities computed in this case are perfect, which means the errors are not as bad. Benchmarks suggest that compared to all the other variants, this delivers the most stable performance at the greatest speed.

4.2 Global illumination using 2-layer deep g-buffer

To simulate global illumination we combine direct-light $DIRECT$, ambient-light $AMBIENT$, screen space ambient visibility[3] AV as well as screen space radiosity $RADIOSITY$ and screen space reflections $REFLECTIONS$ by [DS05] [JBC]

$$DIRECT + AV \cdot AMBIENT + RADIOSITY + REFLECTIONS$$

[3] $AV = 1 - AO$

Algorithm 1 Strawman two-pass generation algorithm for generating 2-layer deep g-buffers

```
//1st pass
submit geometry with:
  geometryShader(tri):
    emit T(tri) to layer 0
  pixelShader(x, y, z):
    return S(x, y, z)

//2nd pass
submit geometry with:
geometryShader(tri):
    emit T(tri) to layer 1
  pixelShader(x, y, z):
    if z > Z[0][x][y] + \Delta z:
      return S(x, y, z)
    else:
      discard fragment
```

Algorithm 2 An improved one-pass algorithm for generating 2-layer deep g-buffers

```
submit geometry with:
  geometryShader(tri)
    emit T(t, tri) to layer 0
    emit T(t, tri) to layer 1
  if (VARIANT == Delay) or (VARIANT == Predict):
    emit T(t+1, tri) to layer 2
  pixelShader(x, y, z):
    switch (layer):
      case 0: // 1st layer; usual G-buffer pass
        return S(x, y, z)
      case 1: // 2nd G-buffer layer: choose the comparison texel
        if (VARIANT == Delay) or (VARIANT == Predict):
          L = 2 // Comparison layer
          C = (x, y, z) // Comparison texel
        else if VARIANT == Previous:
          L = 0; C = (x, y, z)
        else if VARIANT == Reproject:
          L = 0; C = (x[t-1] , y[t-1], z[t-1])
        if z C > Z[t-1] [L][x[C], y[C] ] + delta_z: return S(x, y, z)
        else: discard the fragment
      case 2: // Depth only write to predict Z[t+1][0]; no shading
        return // We only reach this case for Delay and Predict
```

4.3 Double screen space ambient occlusion using 2-layer deep g-buffer

Scalable ambient obscurance [MML12], which is an ambient occlusion method, is extended to work with deep g-buffers. To take into account both z-buffers of the deep g-buffer we have

$$AO(X) = \sqrt{\frac{\pi}{N}\Sigma_{i=1}^{N} max(0, A_i^0, A_i^1)}$$

for the ambient occlusion factor AO where N is the sample count, $R(Z, i)$ returns the position of the $i'th$ sample surface using the z-buffer Z and

$$A_i^j = O(X, R(Z[j], i))$$

where X is the position of the z-value that is being checked, Y the position of the sample (a surrounding z-value) and

$$O(X, Y) = (1 - \frac{v \cdot v}{r^2}) \cdot max(\frac{v \cdot n_X - \beta}{\sqrt{v \cdot v + \epsilon}}, 0)$$

evaluates if the z-value at Y is occluded by z-value at X, where r is the sample radius, $v = Y - X$ the direction vector from position X to position Y and n_X is the normal-vector in the normal-buffer at position X.

With the 2-layer deep g-buffers accounted for, SAO now produces a less noisy AO factor with more plausible shading-falloff [Mar+16].

4.4 Screen space radiosity using 2-layer deep g-buffer

We extend screen space radiosity [CS09] to 2-layer deep g-buffers. We have

$$E(X) = \int_{\Omega} \frac{B(Y)}{\pi} max(n_X \cdot \omega, 0) d\omega$$

for the incident irradiance at X caused by the outgoing diffuse radiance $B(Y)$ from the closest point Y in direction $\omega = Y - X$. This can be **numerically estimated** by

$$E(X) = \frac{2\pi}{M}\Sigma_{samples} B(Y) max(\omega \cdot n_X, 0)$$

where ω is normalized.

We sample N points Y from both g-buffer layers while only using the M points for which

$$(\omega \cdot n_X) > 0$$

and

$$(\omega \cdot n_Y) < 0$$

. Skipping this test will cause bias (noise), but at the same time makes it possible to take more samples, which in return reduces variance. This is up to the users preference.

$$B(X) = E(X) \cdot p_X \cdot boost(p_x)$$

is the reflected outgoing radiance caused by the incident irradiance at X, where p_X is the diffuse reflectivity at X, which is amplified by

$$boost(p) = \frac{max_\lambda p[\lambda] - min_\lambda p[\lambda]}{max_\lambda p[\lambda]}$$

where λ is the wavelength or color channel. Choosing $boost(p) = 1$, for instance, will conserve energy. Else it can be used to boost the scattering of radiance of a saturated surface, which will cause a stronger color bleed.

Since not all points in the deep g-buffer produce robust approximations for indirect shading, we compute a confidence value $0 \leq M/N \leq 1$ that indicates how many samples contribute to the final result, with

a larger confidence value implying stronger influence. According to the confidence value, we linearly interpolate between a precomuted lighting solution and indirect deep g-buffer illumination.

Iteratively repeating the above steps across frames will result in multi bounce indirect illumination, causing soft shadows and color bleeding and diffuse global illumination. This is done by computing one bounce per frame, using the previous frames irradiance-buffer E_{t-1}, which would simulate n bounces in n frames. Obviously this method is prone to ghosting, but its effect can be reduced by damping the forwarded irradiance-buffer by a factor $0 < \delta \leq 1$, which in simple words "underestiamtes" illumination. Compensation for this is provided by some environmental lighting from static light probes. The same accumulation procedure is done across deep g-buffer layers.

Finally, to account for any remaining noise due to undersampling, **temporal filtering** is applied by

$$E_t = E(1 - \alpha) + reproject(E_{t-1})\alpha$$

if for the pixel the reprojected point is within $1cm$ of either layer, else

$$E_t = E$$

with the purpose of detecting faulty velocity approximations and discarding accordingly. Nvidia recommends $\alpha = 0.85$ in most occasions, as $\alpha \geq 0.95$ causes lighting latency and artifacts.

4.5 Screen space reflections using 2-layer deep g-buffer

Screen space reflections are adapted to deep g-buffers by reflection raytracing. Each point in cameraspace is projected into the two g-buffer layers via reflection rays, where points with a maximum distance of Δz within both layers are treated as a hit. The outgoing radiance is set equal to the incoming radiance along the reflection direction. If this procedure fails, e.g. if after a maximum distance of $d > \Delta z$ there is still no hit, or if the reflection ray reaches void space, one performs a switch to environmental mapping as described in section 3.4.

5 Conclusion

Deep g-buffers seem to do a good job at faking global illumination in real time. The resulting renders speak for themselves. As stated by NVIDIA, deep g-buffers also have potential to be used for transparency, depth of field and motion blur.

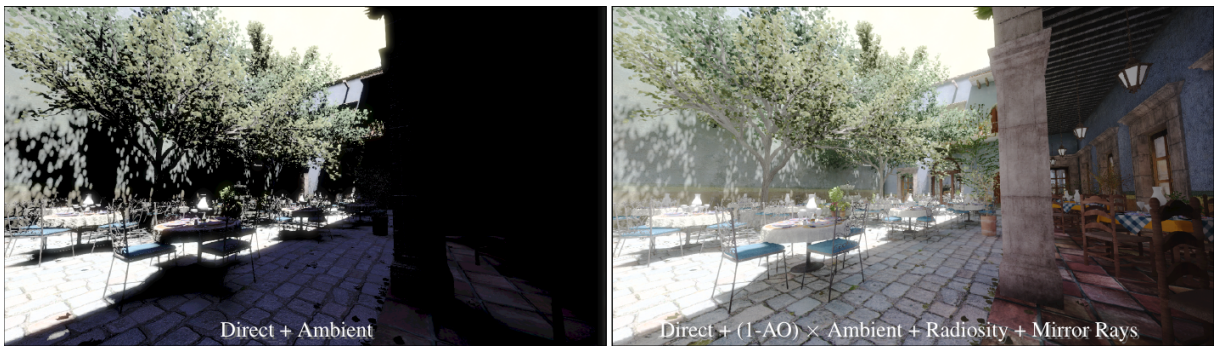


Figure 8: An image rendered without (left) and with the usage of 2-layer deep g-buffers (right). The image was generated in 10.8ms at 1080p using NVIDIA GeForce 980, which implies a theoretical framerate of 92 frames per second.

References

[Aal] Frederik P. Aalund. “A Comparative Study of Screen-Space Ambient Occlusion Methods”. In: ().

[Bav+07] Louis Bavoil et al. “Multi-fragment Effects on the GPU Using the K-buffer”. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D ’07. Seattle, Washington: ACM, 2007, pp. 97–104. ISBN: 978-1-59593-628-8.

[Chr] Per H. Christensen. “Point-Based Approximate Color Bleeding”. In: *Pixar Technical Memo 08-01* ().

[CJ16] Per H. Christensen and Wojciech Jarosz. “The Path to Path-Traced Movies”. In: *Found. Trends. Comput. Graph. Vis.* 10.2 (Oct. 2016), pp. 103–175. ISSN: 1572-2740.

[CG85] Michael F. Cohen and Donald P. Greenberg. “The Hemi-cube: A Radiosity Solution for Complex Environments”. In: *SIGGRAPH Comput. Graph.* 19.3 (July 1985), pp. 31–40. ISSN: 0097-8930.

[CS09] Franck Rochet Nicolas Holzschuch Cyril Soler Olivier Hoel. “A Fast Deferred Shading Pipeline for Real Time Approximate Indirect Illumination”. In: *SIGGRAPH Comput. Graph.* (2009).

[DS05] Carsten Dachsbacher and Marc Stamminger. *Reflective Shadow Maps*. Washington, District of Columbia, 2005.

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

[JBC] E.L. Dagless J. Bulas-Cruz A.T. Ali. “A Temporal Smoothing Technique for Real-Time Motion Detection”. In: ().

[JFH] Andries van Dam John F. Hughes. “Introduction to Radiosity”. In: *EECS 487: Interactive Computer Graphics* ().

[Mar+16] Michael Mara et al. “Deep G-Buffers for Stable Global Illumination Approximation”. In: *Proceedings of the High Performance Graphics 2016*. HPG. 2016, p. 11.

[MK] Geng Chun Tomoyuki Nishita Masanori Kakimoto Tomoaki Tatsukawa. “REAL-TIME REFLECTION AND REFRACTION ON A PER-VERTEX BASIS”. In: ().

[MML12] Morgan McGuire, Michael Mara, and David Luebke. “Scalable Ambient Obscurance”. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG’12. Paris, France: Eurographics Association, 2012, pp. 97–103. ISBN: 978-3-905674-41-5.