The design and evolution of the UberBake light baking system

DARIO SEYB*, Dartmouth College
PETER-PIKE SLOAN*, Activision Publishing
ARI SILVENNOINEN, Activision Publishing
MICHAŁ IWANICKI, Activision Publishing
WOJCIECH JAROSZ, Dartmouth College

Fig. 1. Our system allows for *player-driven* lighting changes at run-time. Above we show a scene where a door is opened during gameplay. The image on the left shows the final lighting produced by our system as seen in the game. In the middle, we show the scene without the methods described here (top). Our system enables us to efficiently precompute the associated lighting change (bottom). This functionality is built on top of a dynamic light set system which allows for levels with hundreds of lights who's contribution to global illumination can be controlled individually at run-time (right). ©Activision Publishing, Inc.

We describe the design and evolution of UberBake, a global illumination system developed by Activision, which supports limited lighting changes in response to certain player interactions. Instead of relying on a fully dynamic solution, we use a traditional static light baking pipeline and extend it with a small set of features that allow us to dynamically update the precomputed lighting at run-time with minimal performance and memory overhead. This means that our system works on the complete set of target hardware, ranging from high-end PCs to previous generation gaming consoles, allowing the use of lighting changes for gameplay purposes. In particular, we show how to efficiently precompute lighting changes due to individual lights being enabled and disabled and doors opening and closing. Finally, we provide a detailed performance evaluation of our system using a set of production levels and discuss how to extend its dynamic capabilities in the future.

*Both authors contributed equally to this research.

Authors' addresses: Dario Seyb, dario.r.seyb.gr@dartmouth.edu, Dartmouth College; Peter-Pike Sloan, ppsloan@activision.com, Activision Publishing; Ari Silvennoinen, ari.silvennoinen@activision.com, Activision Publishing; Michał Iwanicki, michal.iwanicki@activision.com, Activision Publishing; Wojciech Jarosz, wjarosz@dartmouth.edu, Dartmouth College.

0730-0301/2020/7-ART150 $15.00
https://doi.org/10.1145/3386569.3392394

CCS Concepts: • **Computing methodologies** → **Ray tracing**; *Graphics systems and interfaces.*

Additional Key Words and Phrases: global illumination, baked lighting, real time systems

ACM Reference Format:
Dario Seyb, Peter-Pike Sloan, Ari Silvennoinen, Michał Iwanicki, and Wojciech Jarosz. 2020. The design and evolution of the UberBake light baking system. *ACM Trans. Graph.* 39, 4, Article 150 (July 2020), 13 pages. https://doi.org/10.1145/3386569.3392394

1 INTRODUCTION

AAA games today produce images at real-time frame rates (usually 30 or 60 frames per second) that can rival the realism and complexity of offline rendered movies from just a few years ago. This leaves just 16–30 ms to simulate the virtual environment, react to player input, and produce images showing a wide range of complex light-transport phenomena. This last goal can be especially challenging, as players enjoy games on a variety of hardware platforms and comparable quality needs to be achieved on all of them, including ones less powerful than the state of the art, such as mobile devices or previous generation consoles.

One of the difficulties of the rendering process is computing *global illumination*—the component of the lighting that arrives at each point not directly from a light source, but after some number of bounces off other surfaces in the scene. Given the limited time budget, most modern game engines rely on some form of precomputation or *baking*. Parts of the lighting are computed offline, stored in

some data structure, and efficiently retrieved at run time. This was pioneered by the work of id Software on Quake and Quake 2 [Abrash 2000], with the latter being the first game to feature truly indirect lighting, precomputed and stored in textures.

While recent developments in hardware-accelerated ray tracing [Parker et al. 2010; Wyman et al. 2018] provide hope for limited forms of real-time global illumination, these techniques have so far remained too costly as general lighting solutions in AAA games. With the exception of isolated effects (e.g. mirror reflections) real-time ray tracing is unlikely to supplant current baking-based solutions, or even be universally available, for at least the next console generation (and likely longer for mobile platforms).

The limitations of baked lighting are, however, significant. Any changes to the geometry require a costly, offline update that can often take multiple hours, significantly increasing the iteration time for artists. Because the precomputation is performed assuming static level geometry, any changes at run-time have no effect on lighting. For example, a player might destroy a wall, which should flood the inside of a building with light; however, since the lighting was precomputed with the wall intact, there is no information available about how the lighting distribution inside the room should change when it is no longer there. Even simple interactions like opening doors might leave the level's lighting in an inconsistent state.

We describe the design and evolution of UberBake, a dynamic light baking system we developed to address these issues and which we've since used on multiple AAA games. UberBake was developed over the course of multiple releases and innovation was driven mainly by gameplay and level-design requirements. Particularly, we wanted to implement a system for global illumination in which certain player actions can cause dynamic lighting changes. This allows the lighting to be used not only for dramatic visuals, but also as part of the gameplay, for instance to drive player's attention (e.g., flickering lamp can suggest a point of interest) or as a way to solve the game's puzzles (e.g., shooting lights out before engaging enemies will make them less likely to aim accurately).

The central insight of our work is that we can choose a limited subset of user interactions that affect lighting (enabling/disabling lights and opening/closing doors) and receive many of the benefits of a fully dynamic global illumination solution. This made it possible to 1) efficiently pre-compute lighting changes associated with each interaction and 2) implement a run-time system that, on average, is no slower than our previous fully static implementation and uses a minimal amount of extra memory, that scales linearly with the number of allowed interactions.

1.1 Design criteria

During the development of this system we had to fulfill a set of hard constraints. A system that failed to meet one of them would not have been shippable.

C.1 Near-zero runtime overhead. We want to ship games running at 60 FPS on a wide variety of hardware, from modern gaming PCs, to consoles and smart phones. Since the lighting effects are relevant for gameplay, we cannot disable them on low end platforms. For our system to run on all target platforms, it has to have very little overhead on top of existing static lighting.

C.2 No additional constraints on geometry. Many global illumination algorithms impose specific restrictions on level geometry such as a requiring a minimum wall thickness, preferring axis aligned features, and more. There was already a significant amount of content when we implemented our system. Since it was not feasible to rework much of the content, the system had to work well while only requiring minor level changes.

C.3 No major revisions to engine and tools code. We have a large amount of engineering and art resources invested in existing tools and cannot change them significantly. Additionally, we do not have the resources to rewrite large parts of the engine. We had to extend the existing baking pipeline without a complete overhaul, and implementation time had to be weighed against supporting production or extending the baking pipeline in other ways.

Meeting the above-mentioned constraints was the highest priority and narrowed down our options in the design of the system, but we also strived to optimize for the following design goals.

G.1 Minimize artist iteration time. As opposed to run-time performance, we do not have any hard constraints on baking performance. Still, long bakes increase artist iteration times which we would like to avoid. Bake time should scale with scene complexity and the number of interactive elements per scene. Specifically, we strived to achieve sub 10 minute bake times for preview-quality results to enable fast iteration.

G.2 Minimal content creation overhead. Previous systems had content creators manually label every scene element (lights, geometry, etc.) affected by a particular change in the lighting setup. Particularly, it was necessary to classify elements into being active before, during, or after the change in lighting. This was prone to errors due to mislabeling, caused a large workload on lighting artists, and ultimately, the system not being widely used.

G.3 Maximize implementation orthogonality. We want to be able to add interactive elements and improve the baking code without significant changes to the run-time system. This allows us to expose new functionality to artists without the risk that engine changes pose.

While these constraints and goals might seem overly restrictive in an academic context, to our knowledge they are common in production environments and thus our solutions to them are likely broadly applicable. Still, as in any production lighting system, there are many design decisions we made due to constraints specific to our existing shared technology, artist workflows, and production needs. For example the engine we are using is a forward+ renderer [Harada et al. 2012]. This makes using lightmaps a more feasible solution than they would be when implementing a deferred engine. Even though fully volumetric approaches could provide certain benefits (e.g., a better unification of the lighting on different objects), they come both with higher lookup costs (which is problematic, as we target 60 Hz titles) and their own set of problems (that the artists were not familiar with, as the previous games relied on lightmaps).

This made us keep and extend the existing solutions, rather then rewrite them entirely, but a similar line of reasoning to the one described in this paper could be applied to add dynamism to volumetric representations as well.

Non-goals. Finally, we want to explicitly point out that we do not aim to develop a system for use in a customer facing game engine such as Unity [Unity Technologies 2020] or Unreal Engine [Epic Games 2020], but rather a tool that is used internally. This means we only need to support the hardware that our games ship on, without the need to provide fallback solutions for legacy platform, that may be potentially in use by some customers of these general game engines. We can also take certain liberties in choosing implementation details, as all the features are developed in close collaboration with the people using them. For example, in some cases we can rely on a manual procedure, if we know it will not cause an unnecessary burden for the users and when an automatic one would be difficult or time consuming to implement reliably. Additionally, we did not set out to develop a general solution to dynamic global illumination. Instead we empower artists to decide which dynamic effects are important for the look and feel in each level.

1.2 Existing and Alternative Solutions

There exists a vast body of research on global illumination methods for real-time applications. Before endeavouring to develop a new approach, we carefully considered and evaluated existing techniques against our specific goals and constraints, i.e., runtime performance and support for dynamic geometry.

Real-time Light Transport (dynamic lighting and geometry). Real-time light transport methods support dynamic lighting and geometry with minimal precomputation at the cost of run-time performance. Real-time path tracing offers a conceptually simple framework for computing global illumination, and it has recently gained popularity [4A Games 2019; Hillair 2018; Infinity Ward 2019; Remedy Entertainment 2019; Schied 2019] due to the availability of hardware accelerated ray intersection queries [Parker et al. 2010; Wyman et al. 2018] and recent advances in denoising methods [Koskela et al. 2019; Mara et al. 2017; Schied et al. 2017, 2018]. Many-light methods cast the indirect illumination problem in terms of direct illumination from a potentially large number of related virtual light sources [Dachsbacher et al. 2014; Keller 1997]. Unfortunately the current generation of consoles does not ship with dedicated ray-tracing hardware. Hence, given constraints **C.1** and **C.3**, both real-time path tracing and solving the massive visibility problem in the context of many-light methods remains infeasible.

Using a simplified, volumetric representation of the scene is a common way to decouple geometry from the lighting calculations in order to reduce the lighting and visibility computation time while supporting dynamic geometry and lighting [Crassin et al. 2011; Kaplanyan and Dachsbacher 2010; Laine and Karras 2010; Yudintsev 2019] and can be combined with real-time path tracing [Majercik et al. 2019]. Aside from the non-trivial runtime cost, the main downside of volumetric light transport methods is rooted in the mismatch between the simplified scene representation used for lighting and the scene geometry. Achieving consistent lighting without leaks or interpolation artifacts remains a challenge, often requiring changes to level design [Caurant and Lambert 2018; Hooker 2016; Silvennoinen and Timonen 2015], violating constraint **C.2**.

Precomputed Light Transport (dynamic lighting, static geometry). Precomputed light transport (PRT) methods allow dynamic environment lighting while keeping the runtime cost low under the assumption that geometry is mostly static by performing the expensive visibility calculations offline [Christin 2018; Silvennoinen and Timonen 2015; Sloan et al. 2002]. Direct-to-indirect transport methods generalize the lighting model to allow arbitrary, local light sources [Hašan et al. 2006; Kontkanen et al. 2006; Lehtinen et al. 2008; Martin and Einarsson 2010]. With only a few exceptions, PRT methods remain largely incompatible with arbitrary geometry changes, and those that do [Loos et al. 2011, 2012; Silvennoinen and Lehtinen 2017] impose constraints on the structure of the scene geometry that are too limiting in our context (**C.2**), or have performance costs that are too high (**C.1**).

Precomputed Lighting (static lighting and geometry). At the other end of the spectrum, constraining both lighting and geometry to be static has, naturally, the smallest runtime cost, and is arguably the most common form of global illumination in game production [Barré-Brisebois 2017; Chen 2008; Guinier 2020; Iwanicki and Sloan 2017; McTaggart 2004; Neubelt and Pettineo 2015; O'Donnell 2018]. Despite fulfilling all of our *constraints* we cannot use any of these techniques as is, because they do not allow for *any* dynamic lighting changes. Still, particularly due to their run-time performance characteristics, they serve as a good basis to build upon.

Limited forms of dynamic lighting can be supported by precomputing multiple lighting scenarios and interpolating between them at runtime at the expense of increased streaming memory cost. In contrast to the fixed memory overhead of precomputed transport methods, the memory cost from blending the lighting solutions is temporary and can usually be streamed in and out [Blizard 2017; Caurant and Lambert 2018; Christin 2018; McAuley 2018; Öztürk and Akyüz 2017; Tokarev 2018]. These approaches work well in scenarios where lighting changes are limited and not controlled by the player, e.g., when changing the time-of-day. There, streaming load is easily predicted and at most two different sets of lighting have to be kept in memory. Our whole motivation is to support *player-driven* lighting changes to a large set of interactive elements. Using existing techniques would quickly exhaust our memory budget and streaming in a completely new set of lighting in response to player input is not feasible.

1.3 Summary and overview

In summary, no single existing method is able to readily meet our design goals under the performance constraints. We therefore developed our own system based on precomputed lighting using a mixture of volumetric and lightmapped representations for maximum performance while supporting dynamic geometry changes via efficient local lighting updates. In the following we will first describe our (static) global illumination solution (Section 2) and then go into detail about how we gradually extended it during the development of multiple games to handle dynamic lighting effects.

"Dynamic Light Sets" (DLSs) enable us to turn sets of lights on and off in response to player actions (Section 3) and update their contributions to global illumination accordingly. Finally, in Section 4 we extend DLSs to handle non-linear changes in lighting, such as ones resulting from opening and closing doors.

2 OUR BAKED GLOBAL ILLUMINATION SOLUTION

Before we dive into the *dynamic* part of our system, we describe the basic processes and data structures we use to incorporate *static* global illumination into our games. While doing so we will highlight some of the changes we made to the purely static lighting system to prepare for the introduction of dynamic elements. It turned out that all of those changes also improve our static lighting performance, quality, and memory usage, and are in use even when there are no dynamic elements present in the level.

We build off a static lighting system typical in game production and only provide a high-level view of its workings, sufficient enough to understand the changes to make it dynamic. A more detailed treatment of the static baking system we started with is available in Iwanicki and Sloan [2017]. The techniques described there result in a performant approach to baked global illumination that has been proven to work well in practice in a wide variety of scenarios. Lighting artists, level designers and engineers are familiar with the limitations of this type of system and know how to work around potential pitfalls. These considerations are important in a production system since changing central technology always requires buy-in from all parties. Many of the decisions we will outline in the following are therefore driven by user concerns as much as by technological arguments.

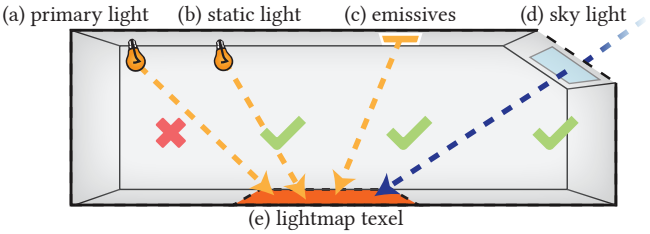
There are four parts to our lighting solution: How we represent (Section 2.1) and store (Section 2.2) lighting, how we precalculate global illumination and what assumptions we make regarding level geometry (Section 2.3), and how we use the precomputed data to incorporate global illumination during shading on both static and moving objects (Section 2.4).

2.1 Representing lighting

When choosing the representation for our lighting data we have a wide variety of options. Not only do we have to decide *how* we store lighting values, but also *which* lighting we store in the first place.

Path notation. For this purpose we introduce some notation to allow us to precisely express paths and their contributions. We extend Heckbert [1990]'s path notation for our use case and denote light sources with L, diffuse reflections with D, and receivers with R. We use multiple different types of light sources, and we will discuss them in detail later in this section. A set of light paths $\mathcal{L}$ can be described by a regular expression with each symbol corresponding to an interaction event. For example, LDDR denotes all paths that start at a light source and end at a receiver via two diffuse bounces. We abuse notation, and will use $\mathcal{L}$ to refer both to a set of paths, as well as the corresponding lighting resulting from those paths.

We largely go with the common industry practice of only precomputing diffuse indirect lighting, that is, paths of the form LD^+R. We convert our run-time material model to purely Lambertian during baking using total hemispherical reflectance (instead of the diffuse albedo component). This gives us a low-frequency approximation to the lighting equation while including much of the energy that contributes to the shaded result. Including specular or direct lighting would require storing data at a much higher resolution to allow satisfactory reconstruction quality. We instead use run-time methods to compute those contributions. That said, in some cases (illustrated below) we do include direct lighting in the bake to trade worse lighting quality for better run-time performance.



Here we show the different types of light paths we compute for the example of a lightmap texel (e). *Primary lights* L_P (a) are the most common light sources in our system. For these we bake indirect lighting only, direct lighting is computed at run-time per pixel. Artists can also place *static lights* L_S (b) for which direct lighting is baked as well, this is done in areas with many lights where we would not be able to compute direct lighting, and particularly shadows, at run-time. We additionally support lighting from emissive geometry L_E (c), and again, this would be too expensive to evaluate at run-time, so both direct and indirect lighting is baked. Note that here *lights* are infinitesimal light sources such as point or spot lights, while *emissives* are triangles with an emissive material. Finally, any light coming from the sky L_{Sky} (d), directly or indirectly, is baked, since computing it at run-time would be prohibitively expensive. This gives us the final baked lighting in the form

$$\mathcal{L}_B = ((L_PD) \mid (L_S \mid L_E \mid L_{Sky}))D^*R. \tag{1}$$

2.2 Storage formats

We store this lighting in different formats depending on memory consumption, run-time access performance, and reconstruction quality considerations. Having multiple storage solutions gives us flexibility in trading off quality and performance depending on the current needs of game and level design. Ensuring that the lighting is consistent despite the different storage formats is important, as it allows us to use them all in a single scene without artifacts (see Fig. 2).

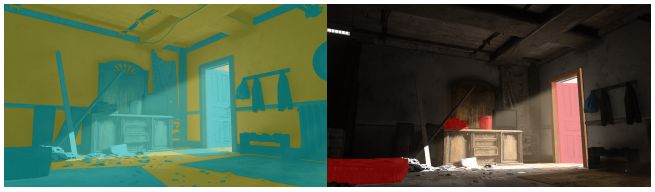


Fig. 2. We use different types of lighting representation in the same scene. On the left is a breakdown of models that use LLGs (teal) and geometry that uses lightmaps (yellow). On the right we highlight dynamic objects in red. Note that dynamic objects still use LLGs, hence the rendering pipeline is unaware of the difference. ©Activision Publishing, Inc.

Directional lighting encoding. Storing simple scalar irradiance would preclude the use of normal maps at run-time, which are critical for appearance fidelity. We therefore store incoming radiance in some basis, e.g., spherical harmonics, which allows us to evaluate irradiance for a given surface normal.

Lightmaps. Lightmaps are the traditional and still widely used way to store lighting data for surfaces in a level. While they can accurately represent surface lighting and are very efficient at run-time, they do come with several downsides. Most importantly, meshes with fine features may require impractically high lightmap resolutions and often exhibit visual artifacts caused by discontinuities in the parametrization. Examples of such difficult meshes are door handles and wires. Still, we use lightmaps for geometry created by level designers in our proprietary level editing tools as well as for large, structural models, such as individual wall segments, or entire buildings. This geometry is mostly comprised of big, flat surfaces, which makes it easy to automatically generate high-quality lightmap UVs. To encode the lighting data we use a variant of Ambient Highlight Direction (AHD) encoding [id Software 1999], with changes by Sloan and Silvennoinen [2018] to improve the reconstruction quality of bilinearly filtered samples. This representation uses 8 bytes/texel which are allocated as follows: one 11_11_10_Float texture (4 bytes/texel) stores the irradiance in local +Z direction (HDR data), another RGBA8 texture (4 bytes/texel) stores the luminance ratio of irradiance in local +Z to ambient radiance (the value is restricted to [0,1] range) and the direction of the highlight. The spatial resolution for lightmaps is determined by the artists, to fit within the prescribed memory budgets. On top of that, we scale the lightmap resolution on each surface relatively in cases when more detail is needed in certain areas. We explored alternatives, but they are more expensive and our lighters preferred this representation.

Local Light Grids (LLGs). Small props such as debris, or intricate ones like cars, doorways or characters are not an ideal application for lightmaps. Debris models might be instanced many times in a level, and, for more intricate models, computing lightmap UVs automatically is prone to failure and edge cases.

Instead we use a data structure we call "Local Light Grids," which was first introduced by Iwanicki and Sloan [2017] to provide a *volumetric* alternative to lightmaps. Instead of trying to store lighting values on the surface of objects, LLGs store them in SH radiance probes *around* the model, akin to an object-centric irradiance volume [Greger et al. 1998]. While Iwanicki and Sloan [2017] used a tetrahedral grid, we decided to go with a simpler Euclidean grid for faster lookups and full decoupling of the lighting, using an oriented bounding box to represent the volume around the model. The main issue with such volumetric storage methods is that the resulting lighting reconstruction misses high frequency detail introduced by visibility changes over the model surface. LLGs solve this problem by storing an additional *self-visibility* term for each model vertex, and accounting for it when interpolating lighting data from the grid probes. An additional benefit of decoupling lighting from visibility is that self-visibility stays the same, no matter where the model is placed in the scene. This allows using instancing to render a model at many scene locations, since the per vertex data is the same for each instance and only the grid probe values change across instances. This representation handles fine features, like door knobs or wires, that would require impractical resolutions with lightmaps. The spatial resolution of LLG defaults to a sample every 1.5 meters, but we use heuristics to change it on a per object basis: if the number of probes used by a single grid is over a certain threshold, we lower the resolution; we add extra probes for objects that can be connected and need a lighting gradients; we also allow overriding the density per-object.

The Global Light Grid (GLG). We now have two representations for lighting on static models, but we are missing a way to shade moving objects. While dynamic objects in our system *do not* affect global illumination, we still want *precomputed* global illumination to influence moving characters, vehicles, and particle effects. One way to handle this is to introduce a *volumetric* lighting representation that allows us to sample indirect lighting at arbitrary points in space. This means that moving objects can evaluate static lighting at whatever position they happen to be in. The resolution of the GLG is variable across the map and determined automatically, based on the distribution of the lighting itself, and the available, gameplay-related information (for instance, playable areas allocate higher resolution GLG). The highest density is a probe every 1.0 meter.

We use a traditional radiance probe grid, distributed over the whole map using a tetrahedral grid [Cupisz 2012; Iwanicki and Sloan 2017]. Each probe (both for the GLG and the LLG) stores radiance in a 3rd order spherical harmonics (SH) basis. This results in 9 coefficients per color channel. The non-DC bands are divided by DC component and scaled by $1/\sqrt{3}$ (linear band) or $1/\sqrt{5}$ (quadratic band) to bring them to the $[-1, 1]$ range (non-negative functions have this bound when projected into SH). For an arbitrary point and normal, we compute the indirect lighting by finding the nearest probes, interpolating their SH values, and evaluating the irradiance for the normal with a convolution. To control light leaking we also store coarse visibility information per tetrahedral face and use it during interpolation to cull non-visible probes [Iwanicki and Sloan 2017]. Volumetric effects sample the GLG directly, while models resample the GLG into a dynamic atlas of LLGs per model. This way the run-time implementation of the lighting lookup can be the same for both static and dynamic objects, with the only difference being the source of the data stored in LLGs. This allows us also to amortize the high cost of GLG lookups—instead of performing such lookups for the millions of visible pixels, we perform it only for the thousands of probe position in the LLG. Just like the LLGs, the GLG stores radiance, which allows us to multiply it by per-vertex self-visibility before performing the cosine convolution.

2.3 Baking via series expansion

Our precomputation uses Monte Carlo ray tracing, but in contrast to alternatives like path tracing [Immel et al. 1986; Kajiya 1986], we structured it as a *series expansion* of the rendering equation, where we compute one bounce at a time, for the whole map. This creates a sequence of final gather [Reichert 1992] passes that can reuse all of the information computed from the previous bounces. With diffuse lighting, each bounce can be stored in the same data structures (lightmaps and LLGs) used for the final rendering. Doing so means that sub-paths are maximally reused, which is biased, but a huge

Fig. 3. Our dynamic lighting system enables us to update global illumination, even in complex scenes. We can go from a completely dark room (left) to a brightly lit one (right) with little performance impact at run-time. In most cases building walls severely limit the influence range of the illumination change. Here the lighting in the adjacent room does not change substantially when we turn on the ceiling light. ©Activision Publishing, Inc.

performance improvement over path tracing where sub-paths are not shared at all. Even though this step runs offline on the CPU and does not impact the run-time of the shipped game, efficiency here was important to us (**G.1**). For the baking pipeline we made the decision to focus on performance on single machines and not distribute over the network. The internal networks in our studios are the limiting factor if we did try and distribute, pushing GB's of data for a couple of minutes of compute is inefficient. Similarly, using a series expansion instead of a path tracer was a conscious decision – we wanted to get a low-quality bake of a level in 5-10 minutes, which would be difficult without using a series expansion. The series expansion also means that data structures only need to be updated *between* bounces, minimizing both the temporary memory that needs to be stored and eliminating the need for locking a read/write data structure, as irradiance caching [Ward et al. 1988] requires. We use Embree [Wald et al. 2014] for tracing the final gather rays and aggressively pre-sort the rays to maximize SIMD coherence.

2.4 Run-time shading

Due to our performance constraints, we have to make sure that we do as little work as possible at run-time. This deems solutions that require searches in depth maps [Majercik et al. 2019], or software interpolation [Silvennoinen and Timonen 2015] too expensive. We also want to simplify the rendering to avoid a combinatorial explosion of shaders. Using LLGs for both dynamic and static models is an example of this, where identical shaders simply run with different resources. We perform expensive operations like evaluating the GLG with compute shaders at sparse locations for dynamic objects, volumetrics and effects. We moved LLG evaluation from vertex shaders, to pixel shaders, and back to vertex shaders in the three games we have shipped using them. This was based on performance constraints, efficiency improvements in the geometry submission pipeline, and the complexity of the content being used on each title.

3 INTERACTIVE LIGHTING UPDATES

Up to this point, all the techniques we describe form a capable and performant, but *static* global illumination system. While we did have some capability to change lighting during gameplay, this was limited to large scale scripted events, such as buildings collapsing. This required many hours of artist and engineering effort to set up in each instance and hence was used sparingly. In this and the following sections we detail how we extended this system to allow for *player-driven* dynamic lighting updates. Our goal was to do so with a minimal set of changes, while preserving the performance and memory characteristics of the static solution. We also made sure that the system was *extensible* and capable of supporting complex lighting changes, as we'll discuss in Section 4.

3.1 Dynamic Light Sets

The first iteration of our dynamic lighting system incorporated "Dynamic Light Sets" (DLSs). This addressed the simple problem of being able to toggle sets of lights at run-time while updating their contributions to global illumination. We chose this as a first step because it is relatively simple to implement and has a large impact on gameplay (e.g., being able to shoot out lights in a first-person shooter). A dynamic light set is a set of primary lights $\{ \mathrm{L}_{\mathrm{P}i} \}$. Following our path notation, its baked lighting contribution is $\mathcal{S} = (\mathrm{L}_{\mathrm{P}1} \mid \mathrm{L}_{\mathrm{P}2} \mid \ldots)\mathrm{D}^{+}\mathrm{R}$. This contribution has to be computed in a separate baking step for each DLS. In our implementation we simply reuse the existing series-expansion baker. Any light in a dynamic light set is ignored in the base bake and a separate pass is run with just the relevant lights enabled.

At run-time, each DLS has an associated blend weight, ω. This weight is computed as the average strength of the lights in the light set. For example, consider a dynamic light set containing two lights illuminating a hallway each at full strength. When one of them is shot out (its strength set to 0), ω now equals 0.5, halving the direct lighting contribution of both lights in the light set.

The final lighting used for shading, $\mathcal{L}$, is then just a linear combination of the base lighting, $\mathcal{L}_{\mathrm{B}}$, and each of the dynamic light set contributions, $\mathcal{S}_j$, multiplied by their respective weight ω_j

$$\mathcal{L} = \mathcal{L}_{\mathrm{B}} + \sum_j \omega_j \cdot \mathcal{S}_j. \tag{2}$$

In levels with no dynamic light sets present, this is equivalent to our static lighting system. As described so far, the approach is conceptually simple and closely related to the technique presented by Öztürk and Akyüz [2017]. Unfortunately, it is prohibitively expensive, so in the following we discuss how we limited the performance impact to scale our system to hundreds of DLSs for single levels.

3.2 Minimal overhead via sparse lighting storage

A major performance concern is that, in theory, each dynamic light set has to compute and store lighting data for every receiver in the level. After all, even though a light's contribution falls off with the square of the distance to the light, it does contribute some light for any given distance. This means, that apart from lights which are fully enclosed, any light might contribute to any receiver in the level. Storing the complete set of data is prohibitively expensive for maps with more than a few dynamic light sets. On large maps it would take multiple gigabytes of memory and make our technique intractable on current hardware. In practice light sets only contribute significantly to a limited region as shown in Fig. 3. To take advantage of this we use a *sparse* data structure to store lighting values for dynamic light sets. The *base bake* runs first and stores data for all receivers. To find relevant receivers, each dynamic

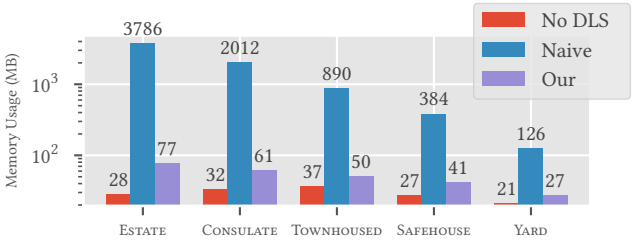


Fig. 4. If we implement dynamic light sets naively, we have to store data for all receivers in the map for each light set, which is prohibitively expensive. Our sparse storage dramatically reduces the memory overhead caused by dynamic light sets. (Note that the y-axis is on a log-scale.). For details on each of these maps, see Table 1 in Section 5.

light set computes direct lighting and two bounces via our series-expansion baker. Then, the mean indirect intensity of the texels lit directly by the sources is used to calculate a threshold (in practice, we use 1% of the mean). Any receivers either lit directly or with intensity higher than that threshold are stored in the update records for a given light set. This limits both the final memory required (see Fig. 4) and also the precomputation time, where the final gather has orders of magnitude more rays to sample. The sparse lighting data structure used during baking is simple. For each active receiver we store an index, pointing to the receiver's location in the base lightmap (for texels) or corresponding probe in the GLG (for GLG probes). The tricky part is keeping performance of the compute shaders high during run-time, and having as little ancillary data as possible, which we address in the following section.

3.3 Fast run-time combination of light sets

The sparse lighting storage we introduced for dynamic light sets reduces memory usage at the cost of run-time complexity. We want to be able to efficiently update our lighting representation (i.e., lightmaps, LLGs and the GLG) when the state of a dynamic light set changes and to do so we have to keep the following set of constraints in mind.

(1) Our solution has to be efficient on the GPU, meaning variable-length data structures and divergent code are problematic.
(2) Since our payload data is so small (8 bytes per lightmap texel), we have to keep any memory overhead of incidental data structures low. Even just storing a single additional index per texel (4 bytes) increases memory storage by 50%.
(3) A common case is that a DLS is completely off and we need to be able to skip these efficiently.
(4) Similarly, we need to be able to skip updates for texels where all the contributing DLSs are unchanged from the previous frame.
(5) We want to directly update the final resource if possible. A lot of PC hardware cannot read-modify-write to texture formats – we can only read or only write the resource in compute shaders. Even where possible, read-modify-write operations have a bandwidth overhead we would like to avoid.

In the following we will discuss why these constraints ruled out naive solutions to the problem, give a high-level overview of our technique, and then describe why and how our approach had to evolve over the last two games using this system.

Naive solutions. One natural way to produce a final lightmap is to *gather* data from DLSs per texel. This is analogous to, for example, matrix palette skinning. Every texel/LG (4 bytes for address) stores the number of DLS that overlap (1 byte), for each overlap it also needs to store the index (1 byte) and the lighting data (8/28 bytes LM/LG). This representation has two problems that make it not viable: It is variable length per texel, which means to index into it we would need to store some base address (even more memory overhead – another 4 bytes), it would have divergence in the shaders (variable loop lengths), and it would be hard to skip work efficiently if all the DLS for a given texel did not change.

The second simple approach would be to store each DLS independently and *scatter* data to the final lightmap. This would require the target texel/LG addresses to be redundantly stored and read-modify-write access to the final resource, but it would make the blend index implicit and have a fixed stride. As an additional upside, we could easily skip work if a DLS did not change or is zero, but only if we did the blends incrementally which can lead to numerical precision problems. If we desire deterministic blend ordering, we would need to update any texel that has any non-zero blend, which can't be reasoned about per DLS.

Our approach. Our technique sits firmly between the two naive solutions discussed above, combining the benefits of both. Instead of either considering individual texels or individual DLSs we cluster texels by the light sets they are affected by in a preprocessing step. This gives us a list of light set combinations with a set of affected texels each. We can then generate one GPU compute dispatch per combination. This fulfills all of the constraints listed above. Within a dispatch, the source information (which light sets to blend) stays constant and thus is the same for each target texel. This means that we require no variable length data structures, avoid divergent code execution, and no redundant data is stored per texel. It is easy to detect if any of the involved light sets for a given dispatch are off and make sure to skip blending these. Furthermore, if none of the light sets for a dispatch has changed, we can simply skip it completely. Finally, we do not need any read-modify-write resources. Each texel belongs to exactly one dispatch and hence is written to once. Even though this technique is conceptually sound and performant, in practice we need to make additional considerations, and these had to evolve as lighting artists started using DLSs more extensively. In particular, creating one dispatch per unique light set combination can lead to dispatches containing very few texels, causing poor GPU utilization and high per-dispatch overhead. This is especially true if there are texels affected by many light sets

Evolution. While developing the first game, artists were instructed to keep the number of DLS in the single digits and try and have no more than 3 overlaps. We created specialized shaders for copying the backing data (all zeros), and 1-3 overlaps. Additionally, there was a fallback shader that was used for texels with more overlaps, or in case a dispatch was too small. This shader copied into a full float_32 scratch buffer (which can always do read-modify-write operations on our target hardware) and looped though the overlaps, sorting them to minimize the number of indices. The fallback shader was not particularly fast, but it was not needed often – a common scenario was a room where the lights might flicker or be shut off (all

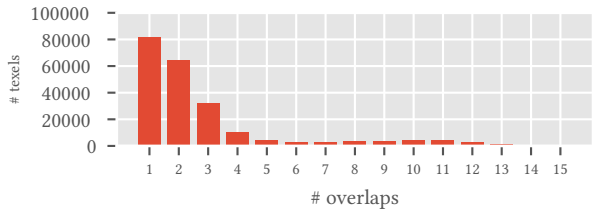


Fig. 5. The number of DLS overlaps (shown here for Estate) per texel grew in the last production. Still, most receivers are only influenced by a small number of light sets.

in the same DLS), which connected to tunnels that had a single DLS and could similarly flicker/turn off. This resulted in large regions with a single set, and 2 DLS overlaps where a tunnel met a room.

The second game had some levels like the first game, but we knew we were going to have to handle much larger numbers of overlaps. Several of the levels were based on stealth gameplay, and individual lights had to be shot out, or turned off at control panels, forcing players to wear night vision goggles and making non-player characters (NPCs) not see the player (radiance was sampled at the player to see if they were visible to NPCs or not). In the hard level, almost every light in a large building is its own set, there are large floodlights from outside that come in through the windows and can be shot out, and there is a fire in the end of the level that adds more sets. This resulted in higher numbers of overlaps that needed to be handled efficiently. Finally, we knew that the multi-state geometry discussed in the next section would also cause additional overlaps

As a first step we increase the number of specialized shaders to handle texels affected by up to 8 dynamic light sets. To eliminate the fallback shader, we introduced two less specialized shaders:

(1) Every texel in a dispatch has the same number of overlaps, but the DLSs could be different. These cannot be "demoted", if any DLS in the dispatch changed, the entire set has to run. This was done for DLS combinations of less than a fixed size (up to 128 texels), allowing us to pack them together, have fewer overall dispatches, but doing some amount of unnecessary work. All overlaps of fewer than 5 DLSs were handled in these shaders.
(2) Two last resort shaders, one for 5-8 overlaps, and one for 9-16. These interleave the data for a single texel, and to compute the "starting address relative to that cluster start (uint16_t)", they store a base index for every batch of 16/8 texels (so amortized to 1/2 bits), and a 2/3 bit per texel with the "count" relative to the base (5,9 so it fits in 2/3 bits). We then just need to compute a prefixsum on the 2/3 bit numbers before the base index used. This has an overhead of 3/5 bits per texel, and causes minimal divergence of data and code. When clustering overlaps for these dispatches, we aimed to minimize the total number of DLSs in a dispatch, since the whole dispatch has to execute if any of the DLSs change.

Both class (1) and class (2) dispatches represent a small percentage of the total number of pixels (see Fig. 5), and the ancillary data is very small relative to anything else (roughly 132KB for the largest map.) Measuring the impact of these optimization shows a 13× speed up, with the lighting in our most challenging map taking just under 20 ms to update using the old shaders, while taking 1.47 ms using the new implementation.

4 MULTI-STATE GEOMETRY

The dynamic lighting system described in the previous section shipped without any further modifications. Motivated by its success we sought to extend it to more complex interactions. Following the same game design driven methodology as before, we decided to tackle the specific issue of opening and closing doors. For context, doors often act as a way to control player progress in first-person shooter campaigns. By letting non-player characters unlock doors, game designers can set the pace of the story and guide the player through complex levels. This means that the player's attention is quite often directed towards an opening door. In levels with dark indoor rooms and bright sun-lit exteriors this poses a challenge for lighting artists as the light flowing through the door is significant (see Fig. 6). Up to now artists had to decide whether to keep the door closed in the baking process (leaving the room dark even when the door is opened during gameplay), or remove it (and ending up with light leaking through the closed door). They could add scripted run-time lights to "cheat" the bounce, but this is both time consuming and expensive. In the following we will describe how we extended the dynamic light set system to allow for dynamic doors without excessively impacting bake-time performance.

Fig. 6. Left: Only base bake. Right: Lighting flowing through door. The indoor lighting can be dominated by the light flowing through the door when opened, making it an important effect to compute. ©Activision Publishing, Inc.

4.1 Doors as dynamic light sets

One of our goals (**G.3**) is to keep the run-time implementation as simple as possible. Since we invested in an efficient implementation of dynamic light sets, we wanted doors to reuse this as much as possible. The challenge was to express the lighting change that happens when opening a door as an *additive component* of the base lighting. We would like to arrive at one set of contributions per door, $\mathcal{S}_\mathrm{P}$, that can be controlled linearly by a weight ω_P which is computed at run-time based on the current "opening angle" of the door.

Preliminary assumptions. To achieve this we make several simplifying assumptions. We reduce the complex non-linear lighting change that happens when the door moves through the scene to two states: "closed" and "open". In the "closed" state, the door model is placed in its completely closed position, while in the "open" state we *completely ignore* the model. Another option would be to place it in some "open" position, but many of the doors in our levels open both outwards as well as inwards and do not have a well defined "open" position. We additionally disregard any light bouncing off the door in its closed state. Computing it would necessitate *removing* light as the door opens, which is not directly compatible with our dynamic

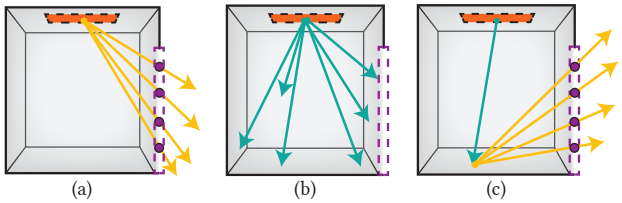


Fig. 7. For each receiver (red), we compute the lighting flowing through the door *directly* (a) by sampling points on its bounding box and casting rays towards it, and one bounce of indirect door lighting by steering final gather rays (b) towards regions of the room that receive strong direct lighting through the door (c).

light set system. While there are ways to achieve this relatively easily, we did not find the missing bounce light to be significant. Finally, while baking dynamic lighting in a level with multiple doors or dynamic light sets we have to make a choice about the state of every other interactive element in the scene. This was not a problem with dynamic light sets themselves. There, lighting is *linearly additive* and the contribution of any individual light set is not affected by whether another light is enabled or not. But in the case of doors, whether a door is open or not *non-linearly* affects light propagation from both dynamic light sets as well as other doors. To get correct results for n DLSs and m doors, we would have to bake *all* $(1+n)2^m$ *combinations* of states, but this quickly becomes intractable. Doors are often relatively far apart. So at the point of shipping the last game using this technology we assumed that for each dynamic light set and door bake, that all other doors are closed. This brings down the number of combinations to just $(n + m + 1)$. Of course, all of these simplifications lead to artifacts in some situations, which we will discuss in Section 6.2 along with possible solutions.

Describing door paths. To reason about our problem we extend the path notation introduced in Section 2 and we will use P to indicate a door in its closed state. As a first step, we identify all the paths that interact with the door at any point. That is, paths of the form

$$\underbrace{\mathrm{LD}^*}_{\text{emitter side}} \overbrace{(\mathrm{PD}^*)^+}^{\text{door interactions}} \underbrace{\mathrm{D}^*\mathrm{R}}_{\text{receiver side}} . \tag{3}$$

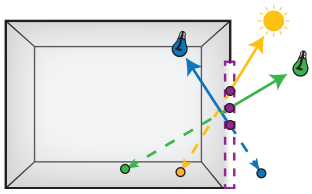
To separate out any light contributions that the door might have, we give it an albedo of 0 in the base bake, removing any paths that interact with it. This allows us to run the base bake as usual, computing any lighting which does not interact with the door efficiently. We are now left with the task of handling the remaining paths.

4.2 Efficient sampling techniques

In the last section we were able to very narrowly define which parts of path space we would like to compute lighting for. The naive approach to do so would be to trace paths starting at each receiver in the level and only count contributions from paths that interact with the door. This would give us the correct result, but doors are usually small compared to the size of the level and the probability of any given path hitting a particular door is low. To efficiently compute the paths that do interact with the door we have to *guide* them towards it. While general path guiding methods [Hey and Purgathofer 2002; Jensen 1995; Müller et al. 2017; Vorba et al. 2014] could be used and we even use the technique by Silvennoinen and Sloan [2019] during the base bake, there is an opportunity to take advantage of the more constrained structure of the door paths. To reduce bake times, we limit ourselves to a certain subset of paths which we observed to have the highest contribution to the overall lighting and we show these in Fig. 7. Namely, we compute any lighting flowing directly through the door and its first bounce. Hence, we want to both guide gather rays towards the door directly and towards areas that are illuminated strongly through the door. Note that here *directly through the door* is not equivalent to *direct lighting*. That is, in addition to connecting to light sources, we also want to take bounce lighting from geometry on the other side of the door into account. Luckily we can use much of the information computed during the base bake to do so. For example, when a ray shot through the door hits a lightmapped model on the other side, we can simply sample the stored indirect lighting.

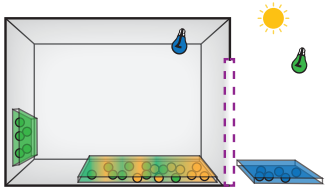
The door as an area light source. For receivers close to the door, many contributing paths are of the form $\mathrm{LD}^+\mathrm{PR}$. Arriving directly from the emitter side, with no bounce on the receiver side as shown in Fig. 7 (a). For receivers on the floor close to the frame, the door subtends a large solid angle. In these cases the door opening forms a (complex) area light source. To compute this contribution we use a stratified sampler to draw points on the door's oriented bounding box and cast rays towards them. The larger the subtended solid angle of the door, the more rays we allocate towards this part of the integral. This is similar to portal sampling strategies used in offline rendering [Bitterli et al. 2015], but in addition to guiding rays towards the portal, these methods also take into account the directional distribution of illumination flowing through the portal. Unfortunately that is very complex in our scenario. As opposed to sampling environment maps, the illumination arriving at the receiver depends not only on the direction but also on its position relative to the portal. We tried using light field importance sampling strategies [Lu et al. 2014] to better distribute samples on the door's bounding box, but found the difference in variance to be minimal.

Clustered shadow photons for path guiding. The other major part of the integral is one-bounce indirect lighting flowing through the door. That is, paths of the form LPDR, shown in Fig. 7 (b). In particular we noticed that we had many scenarios where opening the door would reveal a bright patch of sunlight inside the room and the bounce lighting off this patch would dominate the door lighting. To effectively sample this lighting, we employ a technique inspired by shadow photons [Jensen 1996]. In a preprocess step we uniformly sample points (violet dots) on the bounding box of the door (violet box). For each sample point we send a shadow ray towards a randomly chosen light source (solid arrows). If the shadow test succeeds we know that this light contributes to the light flowing through the door and will hit a surface inside the room. We can then cast into the opposite direction (dashed lines), find the corresponding hit point inside the room, and deposit a shadow photon (green, yellow, and

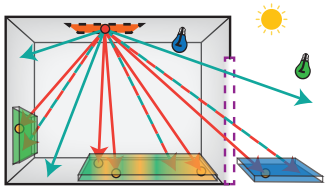


blue dots). This, effectively, gives us a "photon map" of direct light occluded by the door.

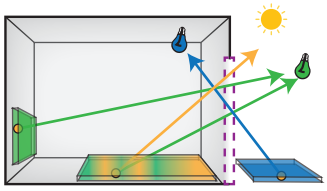
During the gather step we want to send rays towards regions where the density of photons is high, but we additionally need to send out a general gather ray to not miss any part of the integral. To apply multiple importance sampling (MIS) [Veach and Guibas 1995], or even just basic integral splitting, we need to be able to tell whether a uniform gather ray could have been generated by our guiding strategy. This is difficult and expensive if we represent illuminated regions with photon points. To simplify the problem we *cluster* the shadow photons at the end of the preprocess step into oriented bounding boxes (colored areas). This means that we can send out thousands of photons to get a good approximation of the light falling into the room and then reduce that information down to a few (12 in our implementation) bounding boxes that roughly cover areas with strong direct light.



We can now treat these bounding boxes as "area light sources" for the purpose of ray guiding. That is, we do not use them to compute *lighting* directly, as is done in some many-light methods [Luksch et al. 2013], but rather as a proxy to guide gather rays in directions of high contribution. This allows us to accurately combine the guided directions (red) with the uniform hemisphere sampled ones (teal) by computing an intersection with the bounding boxes. If a uniformly sampled ray intersects one of the bounding boxes (red-teal striped) we know that we could have generated it with the guiding technique and we can compute the corresponding MIS weight. Since we have a small set of boxes, this is fast, even without an acceleration structure. Of course, using the bounding boxes means that the approximation of importance is fairly rough and we might miss some features of direct light. We also do not take visibility into account, and in large rooms containing a lot of models, many of the importance-sampled rays might not reach the patches of light. In practice, we have not found this to be a big concern, especially in combination with computing influence regions. Since we combine our guiding strategy via MIS with uniform hemisphere sampling, these issues do not *bias* our estimator and only increases variance in failure cases.



Direct light culling. To further speed up the baking process we use the bounding boxes computed from shadow photons in an additional way. For direct lighting through the door, we need to evaluate all lights in the scene, in theory. Most of them will not contribute and in the general case, culling them is a difficult problem [Dachsbacher et al. 2014]. Keeping track from which light the shadow photons in each bounding box originated, we construct a list of lights per bounding box. When evaluating shading, we find the bounding box the shade point (yellow dots) is in and only use the lights associated with that box. This might mean that we miss some lights if the bounding boxes are too tight. We artificially inflate the bounding boxes to alleviate this. Additionally, artifacts won't be as visible since we only use this lighting contribution while computing bounce light.



5 EVALUATION AND RESULTS

We evaluate our system by thoroughly documenting its performance characteristics in a variety of conditions. Figures 1 and 8 show typical uses of dynamic light sets to enhance gameplay. For a larger set of images and a video showing our system in practice, we refer the reader to our supplemental material. Our intention is to show how the system behaves in practice and hence all the timings and memory statics we show are taken from production content that shipped in a recent game. Table 1 gives an overview of the performance of our system. Note that many levels contain tens of dynamic light sets, while doors are used less often. This is because artists had multiple production cycles to explore DLSs while the door technology came in late during the last production and was only used in the specific situations it was requested for.

Bake-time performance evaluation. Our levels typically cover multiple square miles of terrain and are filled with buildings and props. Multi-hour bake times are not uncommon for environments of this scale, even in our previous static baking system. These times are for the maximum quality setting, as used in the shipping game. During iteration, artists use a lower setting that still produces representative, if somewhat noisy, results with a correspondingly faster bake.

Run-time performance evaluation. The run-time optimizations we presented in Section 3 had a large impact on how many light sets artists could use per level. While on the first game that used the dynamic light baking system, they were instructed to keep light set counts in the single digits for each map, by the time the last

Table 1. Performance and memory statistics for the levels shown in this paper. Run-time performance was measured on a PlayStation 4, while the bakes were performed on a workstation with a recent 18-core CPU. Bake times are for fast / high quality.

	Level Statistics							Performance Statistics		
Level	# LM Texels	# LLG Probes	# GLG Probes	# DLS	# Doors	Memory (no DLS)	Memory (DLS)	Bake (no DLS)	Bake (DLS)	Run Time (sync)
Estate	2,637,824	50,805	229,725	132	0	28 MB	77 MB	10 / 20 min	33 / 133 min	1.47 ms
Consulate	3,276,800	32,783	229,250	61	0	32 MB	61 MB	9 / 18 min	19 / 46 min	0.61 ms
Townhoused	4,194,304	16,524	151,408	24	1	37 MB	50 MB	6 / 14 min	11 / 31 min	0.43 ms
Safehouse	2,363,392	77,110	232,617	14	3	27 MB	41 MB	9 / 18 min	12 / 34 min	0.42 ms
Yard	2,097,152	32,183	135,209	6	0	12 MB	27 MB	5 / 9 min	6 / 11 min	0.21 ms

Fig. 8. Our system enables a variety of gameplay scenarios. The player shoots out a light (left), turning off the corresponding DLS, removing its indirect lighting (middle). While making enemies less accurate, the player has access to night vision goggles (right), allowing them to progress. ©Activision Publishing, Inc.

game shipped we were able to support hundreds of DLSs with little impact on run-time performance. We measured the performance overhead of our system on multiple maps and show the results in Table 1. Under normal gameplay conditions, we observed that dynamic lighting had *no impact* on overall frame time. This is due to the fact that the workload is small and performed on the GPU asynchronously, filling gaps in utilization left by other tasks. Unfortunately, this also makes it impossible to accurately measure their overhead. There is, however, a special debug mode that performs the update synchronously. It also updates *all* the light sets (not only the ones that have had their state changed since the previous frame, like the regular mode). The numbers in the last column of Table 1 are measured using this debug mode. We show, that even when forcing all light sets to update every frame (which does not happen in practice) and forcing synchronous execution (poorly utilizing the GPU), dynamic lighting updates take at most 1.47 ms on our largest production map.

6 DISCUSSION, LIMITATIONS, AND FUTURE WORK

The system we have described performs well, fulfills the criteria we laid out in the introduction and is used in several released as well as upcoming AAA games. In the following we will discuss some historical perspective on the system, its current limitations, how they restrict our use cases, and finally, how we plan to further evolve the system in the future.

6.1 Historical evolution.

Prior to LLGs, we used to store directional lighting information at every vertex of the non-lightmapped meshes. For detailed, finely tessellated meshes, this provided great quality, but the memory footprint and baking time was significant, since the lighting data was unique for each instance. Alternatively, a single directional radiance sample could be used for the entire mesh, but since no self-shadowing information was available, the resulting quality was poor. LLGs, with their dramatically reduced memory footprint, were the first change we made explicitly to support dynamic lighting elements in future games. We initially implemented LLGs under the hood, in our baking code, to accelerate the baking of the per-vertex lighting, which was used in the first game we shipped (2014). For our second game (2016), we moved LLGs to the run-time, and in our third game (2017)—the first to support dynamic updates—we eliminated vertex baking all together. While artists were initially worried about a potential quality loss, we were able to optimize our LLG implementation to the point where we could match visual quality at a fraction of the memory cost.

6.2 Limitations

There are several limitations to our system, some inherent in the design and some due to choices in our particular implementation. Many of the design-caused limitations are common in light baking systems, and we already had to keep them in mind even before introducing any dynamic elements. Most restricting is of course the fact that level designers have to manually label certain interactive elements and dynamic lights. Adding an interactive element requires re-baking the whole map, which can take multiple hours (see Table 1). In practice, this is not a significant additional restriction, since re-baking is common and is caused by many types of changes to the map. While we touched on implementation specific limitations throughout the paper, we recapitulate the most important ones here and provide thoughts on how to lift them.

Fig. 9. Light leaking through the open door due to ignoring the door geometry in the "open" state. ©Activision Publishing, Inc.

Door states and bounce lighting off the door. As discussed in Section 4, we make several simplifying assumptions about the states a door can be in and the states in which it is blocking light. Particularly, we do not compute intersections with the door geometry in its open state at all. This can lead to light leaks like the one shown in Fig. 9. Fixing this is not hard and was simply deemed less important than other tasks at the time. By choosing an "open" state for the door we can include the model in the bake for the open door in the chosen state. This will correctly compute occlusion.

An artifact that is harder to resolve is the missing bounce light off the door when it is closed. This is because currently each door adds exactly one set of lighting contributions. When it is closed, we do not add *any* light, hence, no bounce light. We cannot naively include it in the base bake either since then it would always be

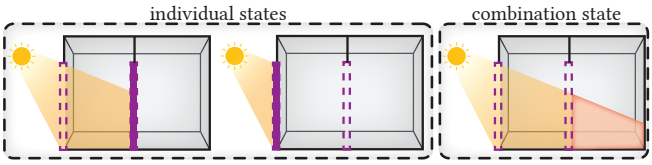


Fig. 10. When we compute individual lighting contributions for doors (left) we can miss important effects that result from combination states (right).

present, even when the door is open. One solution we consider is to let doors contribute an additional set of lighting. Then we could bake the bounce light as one contribution and the light flowing through the door as another and interpolate between them. For exactly two states we can even simplify this back to one set of contributions. Consider $\mathcal{S}_C$ as the light bouncing off a closed door, $\mathcal{S}_O$ as the light flowing through it when it is open and ω_C, ω_O their corresponding blend weights. Since we *linearly interpolate* between the two states we know that $\omega_C = 1 - \omega_O$ and we can express the overall lighting contribution by the door as $\mathcal{S}_P = (1-\omega_C)\cdot\mathcal{S}_C + \omega_O\cdot\mathcal{S}_O = \mathcal{S}_C + \omega_O\cdot(\mathcal{S}_O - \mathcal{S}_C)$. Since $\mathcal{S}_C$ is independent of the door's run-time state, we can add it into the base lighting and we are back to a single set of light contributions per door. Unfortunately this optimization does not generalize to more than two states per door.

Interactions between doors and dynamic light sets. In our system as described in Section 4 each dynamic element stands by itself and light interactions between them as necessitated by geometry changes (e.g., due to doors opening) are not considered. This prohibits correctly computing lighting in situations such as the one shown in Fig. 10 where using the current method, we will never compute the light that reaches the back wall of the second room. Luckily there is a small set of extensions, which we have already implemented since the last game shipped, that allows us to lift this limitations. For each combination state that we want to include we run another baking pass with the given combination of doors open and dynamic light sets enabled. When the combination includes multiple open doors we have to make sure that any contributing path interacts with all of them. We can use the techniques introduced in Section 4.2 as they are to improve baking performance. Note that we do not address the combinatorial explosion directly. Instead we allow artists to select individual light sets and doors that should participate in combinatorial effects and only include combinations where the participating elements' influence radii overlap.

6.3 Future work and outlook

We have only addressed diffuse lighting in this paper. Non-diffuse interactions are handled by reflection probes, where low gloss materials directly integrate against the low frequency incident lighting as an optimization. We use normalized reflection probes [Lazarov 2013] which are divided by irradiance when computing them offline, and multiplied back after sampling environment maps. Just updating the baked lighting data generates plausible specular results, particularly for lighting changes. For geometry changes, we should investigate other ways to update reflection probes in the future. In future productions, we will likely extend the existing system to handle more complex lighting changes. After lifting some of the current limitations as described in the previous section, a simple next step would be to generalize the technique we use for dynamic doors to other similar scenarios. Events such as walls and ceilings getting destroyed will be straightforward and have a large impact on our ability to support a wide variety of level-design needs.

Finally, there is exciting progress being made on real-time ray tracing by academia and industry alike and on the highest end hardware there are scenarios where it could replace baked lighting even now. For wider adoption, the main issue real-time ray tracing techniques will have to face is that the run-time overhead requirements presented here are by no means extravagant or unusually restrictive. AAA games in particular typically require *extremely* performant techniques and overhead is often measured in microseconds. To achieve this with real-time ray tracing in the near future we see many of the ideas presented here coming in useful. For example, doing updates on data structures decoupled from the frame and final resolution, as well as not being completely general (focusing on specific effects) can yield increased fidelity at much lower performance cost. This is why we are interested in exploring the Pareto frontier across the many available axes - speed, memory, precompute time, general vs. specialized techniques, etc. We would expect there are some sub dimensions where ray tracing will win and some contexts where it will take a very long time to replace baked lighting. That being said, lower-end platforms like mobile phones will continue to have difficulty ray tracing complex scenes and as AAA games seek to extend their target audience, we see methods relying on precomputation being useful in the foreseeable future.

ACKNOWLEDGMENTS

We had the good fortune to work with many talented lighters while designing the system: Luka Romel, Vivian Ding, Dave Blizard, Omar Gatica, Velinda Reys, Krzystof Wojcik, Marko Vukovic. Many Ko, Michael Stark and Adrien Dubouchet contributed to UberBake code. We also worked closely with several engineers: Stephanus Fnu, Danny Chan, Michal Drobot, Peter Pon, Akimitsu Hogge and Michael Vance. We are also grateful for encouragement and support from Andy Hendrickson, Dave Stohl and Patrick Kelly.

REFERENCES

4A Games. 2019. Metro Exodus. Microsoft Windows.
Michael Abrash. 2000. Quake's Lighting Model: Surface Caching. (2000). https://www.bluesnews.com/abrash/chap68.shtml
Colin Barré-Brisebois. 2017. A Certain Slant of Light: Past, Present and Future Challenges of Global Illumination in Games. In *Open Problems in Real-Time Rendering (ACM SIGGRAPH Courses)*. https://doi.org/10/ggfk67
Benedikt Bitterli, Jan Novák, and Wojciech Jarosz. 2015. Portal-Masked Environment Map Sampling. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering)* 34, 4 (July 2015). https://doi.org/10/f7mbx7
Dave Blizard. 2017. Lighting in VR, Embracing the Perpetual Newbie. *High Performance Graphics – HotVR Talks*. https://www.highperformancegraphics.org/2017/program/
Guillaume Caurant and Thibault Lambert. 2018. The lighting technology of Detroit: Become Human. In *Game Developers Conference*. https://www.gdcvault.com/play/1025339/The-Lighting-Technology-of-Detroit
Hao Chen. 2008. Lighting and Material of Halo 3. In *Game Developers Conference*. https://www.gdcvault.com/play/253/Lighting-and-Material-of-HALO
Fabien Christin. 2018. Lighting the City of Glass - Rendering 'Mirror's Edge: Catalyst'. In *Game Developers Conference*. https://www.gdcvault.com/play/1023284/Lighting-the-City-of-Glass
Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. 2011. Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum (Proc. Pacific Graphics)* 30, 7 (Sept. 2011). https://doi.org/10/fqnc9v
Robert Cupisz. 2012. Light Probe Interpolation Using Tetrahedral Tessellations. In *Game Developers Conference*. https://www.gdcvault.com/play/1015312/Light-Probe-

Interpolation-Using-Tetrahedral
Carsten Dachsbacher, Jaroslav Křivánek, Miloš Hašan, Adam Arbree, Bruce Walter, and Jan Novák. 2014. Scalable Realistic Rendering with Many-Light Methods. *Computer Graphics Forum* 33, 1 (Feb. 2014). https://doi.org/10/f5twgd
Epic Games. 2020. Unreal Engine. https://www.unrealengine.com/
Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. March/April 1998. The Irradiance Volume. *IEEE Computer Graphics & Applications* 18, 2 (March/April 1998). https://doi.org/10/ckjbb8
Florent Guinier. 2020. GPU Lightmapper: A Technical Deep Dive. https://blogs.unity3d.com/2019/05/20/gpu-lightmapper-a-technical-deep-dive/
Takahiro Harada, Jay McKee, and Jason C Yang. 2012. Forward+: Bringing deferred lighting to the next level.. In *Proc. Eurographics Short Papers*. https://doi.org/10/dtm8
Miloš Hašan, Fabio Pellacini, and Kavita Bala. 2006. Direct-to-Indirect Transfer for Cinematic Relighting. *Proc. SIGGRAPH* 25, 3 (July 2006). https://doi.org/10/cqgn89
Paul S. Heckbert. 1990. Adaptive Radiosity Textures for Bidirectional Ray Tracing. *Proc. SIGGRAPH* 24, 4 (Aug. 1990). https://doi.org/10/bsxgp4
Heinrich Hey and Werner Purgathofer. 2002. Importance Sampling with Hemispherical Particle Footprints. In *Proc. Spring Conference on Computer Graphics (SCCG)*. https://doi.org/10/fmx2jp
Sébastien Hillair. 2018. "Real-time Raytracing for Interactive Global Illumination Workflows in Frostbite & "Shiny Pixels and Beyond: Real-Time Raytracing at SEED (Presented by NVIDIA). In *Game Developers Conference*. https://www.gdcvault.com/play/1024801/
JT Hooker. 2016. Volumetric Global Illumination at Treyarch. In *Advances in Real-Time Rendering Course (ACM SIGGRAPH Courses)*. http://advances.realtimerendering.com/s2016/index.html
id Software. 1999. Quake III Arena. https://github.com/id-Software/Quake-III-Arena
David S. Immel, Michael F. Cohen, and Donald P. Greenberg. 1986. A Radiosity Method for Non-Diffuse Environments. *Proc. SIGGRAPH* 20, 4 (Aug. 1986). https://doi.org/10/dmjm9t
Infinity Ward. 2019. Call of Duty: Modern Warfare. Microsoft Windows.
Michał Iwanicki and Peter-Pike Sloan. 2017. Precomputed Lighting in Call of Duty: Infinite Warfare. In *Advances in Real-Time Rendering in Games, Part I (ACM SIGGRAPH Courses)*. https://doi.org/10/gf3tbc
Henrik Wann Jensen. 1995. Importance Driven Path Tracing Using the Photon Map. In *Rendering Techniques (Proc. Eurographics Workshop on Rendering)*. https://doi.org/10/gf2hcr
Henrik Wann Jensen. 1996. Global Illumination Using Photon Maps. In *Rendering Techniques (Proc. Eurographics Workshop on Rendering)*. https://doi.org/10/fzc6t9
James T. Kajiya. 1986. The Rendering Equation. *Proc. SIGGRAPH* 20, 4 (Aug. 1986). https://doi.org/10/cvf53j
Anton Kaplanyan and Carsten Dachsbacher. 2010. Cascaded Light Propagation Volumes for Real-Time Indirect Illumination. In *Proc. Symposium on Interactive 3D Graphics and Games*. https://doi.org/10/bxjp4v
Alexander Keller. 1997. Instant Radiosity. In *Proc. SIGGRAPH*. https://doi.org/10/fqch2z
Janne Kontkanen, Emmanuel Turquin, Nicolas Holzschuch, and François X. Sillion. 2006. Wavelet Radiance Transport for Interactive Indirect Lighting. In *Rendering Techniques (Proc. Eurographics Symposium on Rendering)*. https://doi.org/10/ggfk62
Matias Koskela, Kalle Immonen, Markku Mäkitalo, Alessandro Foi, Timo Viitanen, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. 2019. Blockwise Multi-Order Feature Regression for Real-Time Path-Tracing Reconstruction. *Proc. SIGGRAPH* 38, 5 (June 2019). https://doi.org/10/ggd8dj
Samuli Laine and Tero Karras. 2010. Efficient Sparse Voxel Octrees. In *Proc. Symposium on Interactive 3D Graphics and Games*. https://doi.org/10/c6n3hz
Dimitar Lazarov. 2013. Getting More Physical in Call of Duty: Black Ops II. In *ACM SIGGRAPH Courses*. https://blog.selfshadow.com/publications/s2013-shading-course/
Jaakko Lehtinen, Matthias Zwicker, Emmanuel Turquin, Janne Kontkanen, Frédo Durand, François X. Sillion, and Timo Aila. 2008. A Meshless Hierarchical Representation for Light Transport. *Proc. SIGGRAPH* 27, 3 (Aug. 2008). https://doi.org/10/cbpkvx
Bradford J. Loos, Lakulish Antani, Kenny Mitchell, Derek Nowrouzezahrai, Wojciech Jarosz, and Peter-Pike Sloan. 2011. Modular Radiance Transfer. *Proc. SIGGRAPH Asia* 30, 6 (Dec. 2011). https://doi.org/10/gfzndt
Bradford James Loos, Derek Nowrouzezahrai, Wojciech Jarosz, and Peter-Pike Sloan. 2012. Delta Radiance Transfer. In *Proc. Symposium on Interactive 3D Graphics and Games*. https://doi.org/10/gfzndh
Heqi Lu, Romain Pacanowski, and Xavier Granier. 2014. Position-dependent importance sampling of light field luminaires. *IEEE Transactions on Visualization and Computer Graphics* 21, 2 (2014). https://doi.org/10/f6v2b5
Christian Luksch, Robert F. Tobler, Ralf Habel, Michael Schwärzler, and Michael Wimmer. 2013. Fast Light-Map Computation with Virtual Polygon Lights. In *Proc. Symposium on Interactive 3D Graphics and Games*. https://doi.org/10/f3s5gf
Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire. 2019. Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields. *Journal of Computer Graphics Techniques (JCGT)* 8, 2 (June 2019). http://jcgt.org/published/0008/02/01/
Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. 2017. An Efficient Denoising Algorithm for Global Illumination. In *Proc. High Performance Graphics*. https://doi.org/10/gfzndq
Sam Martin and Per Einarsson. 2010. A Real-Time Radiosity Architecture for Video Game. In *Advances in Real-Time Rendering in 3D Graphics and Games, Part I (ACM SIGGRAPH Courses)*. https://advances.realtimerendering.com/s2010
Stephen McAuley. 2018. The Challenges of Rendering an Open World in Far Cry 5. In *Advances in Real-Time Rendering in Games (ACM SIGGRAPH Courses)*. https://doi.org/10/gf3tbf
Gary McTaggart. 2004. Half-Life® 2 / Valve Source™ Shading. http://www.decew.net/OSS/References/D3DTutorial10_Half-Life2_Shading.pdf
Thomas Müller, Markus Gross, and Jan Novák. 2017. Practical Path Guiding for Efficient Light-Transport Simulation. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering)* 36, 4 (June 2017). https://doi.org/10/gbnvrs
David Neubelt and Matt Pettineo. 2015. Advanced Lighting R&D at Ready At Dawn Studios. In *Physically Based Shading in Theory and Practice (ACM SIGGRAPH Courses)*. https://doi.org/10/gf3s6p
Yuriy O'Donnell. 2018. Precomputed Global Illumination in Frostbite. In *Game Developers Conference*. https://www.gdcvault.com/play/1025434/Precomputed-Global-Illumination-in
Bekir Öztürk and Ahmet Oğuz Akyüz. 2017. Semi-Dynamic Light Maps. In *ACM SIGGRAPH 2017 Posters*. https://doi.org/10/ggfk9d
Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *Proc. SIGGRAPH* 29, 4 (July 2010). https://doi.org/10/frf4mq
Mark C. Reichert. 1992. *A Two-Pass Radiosity Method Driven by Lights and Viewer Position*. M.Sc. Thesis. Program of Computer Graphics, Cornell University.
Remedy Entertainment. 2019. Control. Microsoft Windows.
Christoph Schied. 2019. Video Series: Path Tracing for Quake II in Two Months. https://devblogs.nvidia.com/path-tracing-quake-ii/
Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proc. High Performance Graphics*. https://doi.org/10/ggd8dg
Christoph Schied, Christoph Peters, and Carsten Dachsbacher. 2018. Gradient Estimation for Real-Time Adaptive Temporal Filtering. *Proc. ACM on Computer Graphics and Interactive Techniques* 1, 2 (Aug. 2018). https://doi.org/10/ggd8dh
Ari Silvennoinen and Jaakko Lehtinen. 2017. Real-Time Global Illumination by Precomputed Local Reconstruction from Sparse Radiance Probes. *Proc. SIGGRAPH Asia* 36, 6 (Nov. 2017). https://doi.org/10/gcqbvn
Ari Silvennoinen and Peter-Pike Sloan. 2019. Ray Guiding for Production Lightmap Baking. In *ACM SIGGRAPH Asia Technical Briefs*. https://doi.org/10/ggfk88
Ari Silvennoinen and Ville Timonen. 2015. Multi-Scale Global Illumination in Quantum Break. In *Advances in Real-Time Rendering in Games, Part I (ACM SIGGRAPH Courses)*. https://doi.org/10/gf3s6n
Peter-Pike Sloan, Jan Kautz, and John Snyder. 2002. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. *Proc. SIGGRAPH* 21, 3 (2002). https://doi.org/10/fgq3kn
Peter-Pike Sloan and Ari Silvennoinen. 2018. Directional lightmap encoding insights. In *ACM SIGGRAPH Asia Technical Briefs*. https://doi.org/10/dtm9
Kirill Tokarev. 2018. Horizon Zero Dawn: Interview With the Team. https://80.lv/articles/horizon-zero-dawn-interview-with-the-team/
Unity Technologies. 2020. Unity. https://unity.com/
Eric Veach and Leonidas J. Guibas. 1995. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proc. SIGGRAPH*, Vol. 29. https://doi.org/10/d7b6n4
Jiří Vorba, Ondřej Karlík, Martin Šik, Tobias Ritschel, and Jaroslav Křivánek. 2014. On-Line Learning of Parametric Mixture Models for Light Transport Simulation. *Proc. SIGGRAPH* 33, 4 (Aug. 2014). https://doi.org/10/f6c2cp
Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *Proc. SIGGRAPH* 33, 4 (2014). https://doi.org/10/gfzwck
Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. 1988. A Ray Tracing Solution for Diffuse Interreflection. *Proc. SIGGRAPH* 22, 4 (Aug. 1988). https://doi.org/10/dk6rt5
Chris Wyman, Shawn Hargreaves, Peter Shirley, and Colin Barré-Brisebois. 2018. Introduction to DirectX Raytracing. In *ACM SIGGRAPH Courses*. https://doi.org/10/djqr
Anton Yudintsev. 2019. Scalable Real-Time Global Illumination for Large Scenes. In *Game Developers Conference*. https://www.gdcvault.com/play/1026469/Scalable-Real-Time-Global-Illumination