**The Computer Games Journal manuscript No.**
(will be inserted by the editor)

# Realistic Lighting for Interactive Applications Using Semi-Dynamic Light Maps

**Bekir Öztürk · Ahmet Oğuz Akyüz**

Received: date / Accepted: date

**Abstract** Light mapping is a technique that works by precomputing the lighting of a scene to speed up expensive lighting calculations at run-time. It is a commonly employed technique for computer games, in particular mobile games, due to the difficulty of creating realistic lighting effects on low power devices. The primary drawback of this technique is that the scene state that is dependent on the precomputed data cannot be changed at run-time. This limitation of light mapping significantly decreases the interactability of applications that use light maps. In this paper, we present a method to remove some of these restrictions at the cost of additional texture memory and small CPU/GPU workload. This allows changing the color and intensity properties of selected light sources at run-time, while keeping the benefits of the light mapping technique. It also makes it possible to show or hide the selected objects without giving rise to shadow artifacts and illumination inconsistencies. Our algorithm computes the light maps separately for each light source. Regions shadowed by each semi-dynamic object are also captured and stored. These maps are later combined at run-time to correctly illuminate the scene. Despite the increase in the generation time of the precomputed data, the overhead of the method is low enough to make it useful in many real-time applications. This makes our method especially suitable for interactive computer games designed to run on mobile and low-end graphics processors.

**Keywords** Light mapping · rendering techniques · interactive real-time rendering

Bekir Öztürk is with Microsoft, Czech Republic · Ahmet Oğuz Akyüz is with Middle East Technical University, Department of Computer Engineering, Ankara, Turkey. E-mail: bozturk@microsoft.com,aoakyuz@metu.edu.tr

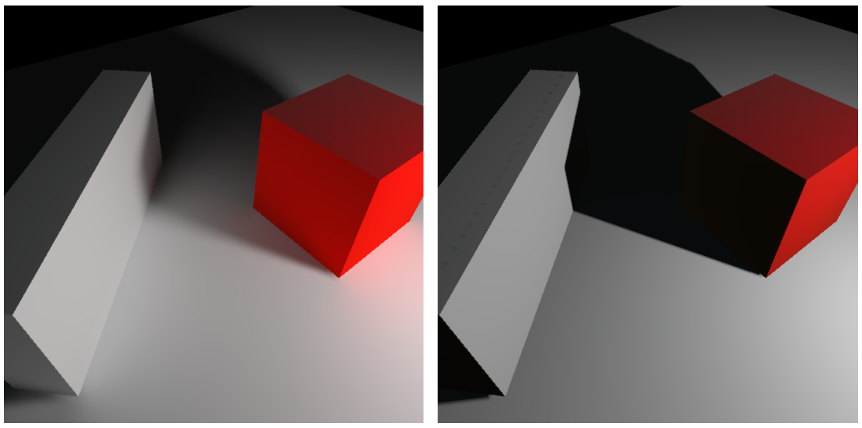
**Fig. 1** Using high quality rendering techniques, light maps (left) can generate more realistic results than real-time lighting (right).

## 1 Introduction

Lighting is a very important aspect of rendering that significantly effects the realism of the generated imagery. Many high quality rendering techniques use a ray tracing based approach to simulate the propagation of light (Jensen 1996, Heckbert 1990). These methods yield high quality results that are sometimes indistinguishable from photographs. Reflections, refractions, and scattering are some of the effects that can be simulated with these methods. However, it is often not possible to use these methods in real-time applications since rendering times are too high to achieve high frame rates in today's hardware.

A common remedy for this problem is to use precomputed data (Jensen 1996, Heidrich & Seidel 1998, Ramamoorthi & Hanrahan 2001, Good & Taylor 2005, Sloan et al. 2002). This data contains information about the lighting of the scene. At run-time, an efficient method is used to compute the correct light values and generate realistically lit images using this data.

One of the mainly used lighting techniques in real-time applications, especially computer games, is light mapping (Bastos et al. 1997, Kopylov et al. 1998). In this technique, all of the surfaces in the scene are uniquely mapped to a texture called a light map. During the precomputations, the lighting of the scene is simulated using a high quality lighting simulation method. Light received per unit area is calculated and stored in texels for all surfaces in the scene. This operation is known as *light baking*. At run-time, the lighting of a point can easily be calculated by looking up the value from the light map. Light maps do not store information about the source of the light. Multiple light sources can contribute to the value of a single texel. An important benefit of this is that the light map size and run-time performance are independent of the number of lights in the scene. Area lights, indirect illumination, and ambient occlusion (Iones et al. 2003) are some of the effects that can be efficiently used with light mapping (Figure 1).

Light mapping also has some drawbacks and restrictions. In common implementations of light mapping, texels contain only intensity values. Direction of the light is not stored. For this reason, light maps are said to be view-independent (Luksch et al. 2013). This means that view-dependent techniques, such as specular shading, cannot be simulated using light maps.

Another and possibly the chief drawback of using light maps is the restrictions imposed on the scene's state. After the baking of the light map is completed, the light received at any point in the scene is already decided. Therefore, no actions can be taken to change the lighting of the scene without re-baking the light map. For instance, the intensities, colors, and positions of the light sources cannot be changed. Objects cannot be repositioned or removed from the scene. Newly added objects will not be illuminated. Objects that are removed from the scene will continue to cast shadows. These problems can be avoided by excluding these objects and light sources from light maps. In this case, the realism of the rendered image will be hampered since lighting calculations will have to be done using real-time lower quality methods.

In this paper, we propose a method to mitigate one of the main challenges of light maps, namely the requirement of fully static objects and lights. Our method allows adding and removing objects and lights to a scene as well as varying the colors and intensities of light sources. Different from the earlier work, we accomplish these without compromising the lighting quality by resorting to real-time shading approximations.

## 2 Related Work

Accurate lighting simulation in real-time for dynamic environments is the envy of the most computer graphics research. Many techniques have been developed to achieve this goal. While earlier techniques relied on precomputation to store the results of advanced lighting simulations (Cohen & Greenberg 1985, Weghorst et al. 1984, Heckbert 1990, Myszkowski & Kunii 1995, Heidrich et al. 2000, Sloan et al. 2002), recent worked started to push the boundaries of GPU computing to allow path tracing in real-time (Wald et al. 2019). In the following, we briefly review some of the precomputation based techniques that aim to enable realistic lighting in interactive and real-time applications. However, given the large body of works on this topic, we refer the interested reader to excellent surveys for more comprehensive reviews (Akenine-Moller et al. 2018, Kautz 2004, Ramamoorthi et al. 2009).

### 2.1 Photon Mapping

Photon mapping is one of earlier the techniques that takes advantage of pre-computed lighting data (Jensen 1996). It is a two pass algorithm, in which the first pass involves depositing photons on surfaces and the second pass uses these photons to compute the contribution of light at each surface point.

Photon mapping was not intended for real-time applications, but rather for improving the convergence rates over traditional path tracing methods (Pharr et al. 2016). Later studies aimed at reducing the memory costs of photon maps (Good & Taylor 2005, Hachisuka et al. 2008, Hachisuka & Jensen 2009) as well as making it more accurate (Kaplanyan & Dachsbacher 2013) and real-time (Mara et al. 2013, Knaus & Zwicker 2011).

## 2.2 Environment Mapping

Environment maps (EM) store the reflections around a point to compute the shading on mirror-like objects with better efficiency (Voorhies & Foran 1994, Haeberli & Segal 1993, Heckbert 1986, Heidrich & Seidel 1998, Miller & Hoffman 1984, Greene 1986). The generation of an EM starts with selecting a point within or near reflective objects. From this point, the scene view is rendered and saved into either 2D or cubemap textures. When rendering a point, the reflected color is sampled from the environment maps instead of sending reflection rays into the scene. In cases where the reflective object is greatly distant from the surrounding environment, changes to the object position can be tolerated without recomputing the EM. In other cases, the EM must be updated. One of the solutions to this problem is to generate a limited number of EMs along the path of the moving object (Cabral et al. 1999, Hakura et al. 2001, Meyer & Loscos 2003). Two closest EMs can then be blended using the distance of the object as the blending weight.

## 2.3 Irradiance Mapping

While EMs are used for reflective objects, irradiance maps (IM) are used for representing the incident lighting received by diffuse objects. Ramamoorthi and Hanrahan showed that by using only 9 parameters, any diffuse reflection map can be closely estimated (Ramamoorthi & Hanrahan 2001). The main idea is to use spherical harmonic (SH) coefficients. Due to the blurry nature of IMs, the values of high frequency coefficients are small enough to be neglected. On static objects where coefficients are stored on the surface, all possible reflection directions on a surface point form a hemisphere instead of a sphere. Taking advantage of this fact, Habel and Wimmer showed that it is sufficient to use only 6 spherical harmonics coefficients for irradiance normal mapping of static objects (Habel & Wimmer 2010).

## 2.4 Precomputed Radiance Transfer

Sloan et al. (2002) have shown that by projecting the lighting distribution on a SH basis and by precomputing the radiance transfer on every vertex (or texel) of an object, advanced lighting computations can be done in real-time. The primary assumption of this technique is that the scene or the objects are lit by only distant light sources. Local lighting effects are not supported. Furthermore high frequency components of the lighting will be blurred during the projection onto the SH basis. Although the original technique requires a large amount of memory to store the radiance transfer coefficients, later work proposed compression techniques for these coefficients (Sloan et al. 2003). The low-frequency (Tsai & Shih 2006) as well as distant lighting (Kristensen et al. 2005) requirements have been addressed by future studies.

## 2.5 Light Mapping

Light mapping is a technique that uses textures to store the irradiance on surfaces. The technique was first utilized for the 3D game Quake in 1996 (Abrash 2000, Sa'dyah et al. 2018). Each surface represented in the texture is uniquely mapped to it via texture coordinates. The resulting texture is known as the *light map* and the process of generating a light map is called *baking*. During rendering, all lighting calculations of baked objects are omitted and light values are directly sampled from the light map.

### *2.5.1 Baking*

There are two common methods used for storing baked data in a scene. The first one involves storing light data in vertex attributes (vertex-baking). The second method uses textures (texture-baking) and samples light value of each pixel in the pixel shader (Larsson 2010). The former approach is more efficient since it neither requires high resolution textures nor sampling of the light map in the pixel shader. It also avoids the costs of generating non-overlapping UV coordinates for triangles prior to baking. For large polygons under detailed lighting, however, storing light values in vertices often results in low quality results (Cohen et al. 1986). Texture-based methods offer better details independent of the triangle size, if memory requirements can be met.

To cope with the shortcomings of vertex-based methods, one of the common methods is to divide the existing polygons into smaller ones (Cohen et al. 1986, Baum et al. 1991, Heckbert 1992, Kopylov et al. 1998). Li et al. (2012) subdivided meshes into smaller triangles where required. An input light map texture is analyzed and a necessary subdivision level is calculated for each triangle. This results in a higher number of triangles. The light map texture can be completely removed after the vertex baking is complete. According to the presented results, the rendering times of vertex-baked subdivided meshes are comparable to texture-baked unmodified meshes, while memory requirements are significantly reduced (Li et al. 2012).

For most of the triangles in a typical scene, vertex-baking is sufficient. Rest of the triangles, however, cannot be lit with vertex stored light data since it is not possible to store high frequency details on only 3 vertices. Schäfer et al. (2012) merged vertex-baking and texture-baking into a hybrid solution. The proposed method uses vertices to store lighting by default for efficiency reasons. For triangles where vertex storage is insufficient and high frequency changes in lighting causes artifacts, textures are used. The algorithm first detects for which triangles the texture-based approach is required. The texture coordinates for light map sampling are only generated for these triangles. To prevent artifacts at the shared edge of two triangles that use different methods, a custom shader is used for blending.

### *2.5.2 Reacting to Changes*

In most light baking methods, a change in the scene state requires a complete re-baking of light maps, even if the change only affects a small part of the scene. Long baking times prevent designers to quickly iterate and achieve the desired look. Using a Many-Light Global Illumination approach, Luksch et al. (2019) proposed a method to update only the changed parts of the light map. The procedure involves detecting virtual point lights (VPL) that were affected by the changes in the scene. These VPLs are then removed from the corresponding point cluster and new VPLs are generated according to the new state of the scene.

In an earlier study, Luksch et al. (2013) combined multiple virtual point lights into virtual polygon lights to reduce the baking time of light maps. Despite the overhead of grouping VPLs into polygon lights, this method provides a significant speedup to the overall baking process. To allow interactive editing, the light map is temporarily generated with a direct-indirect hybrid illumination method when a light source or object is moved. This process starts with subtracting the affected light from the light map and a real-time direct illumination is temporarily used for these light sources. As high quality light map with indirect illumination is generated, it is blended together with the existing one. While this allows faster iterations during editing, light map generation times are not low enough to be used in real-time applications, even on high-end devices. Depending on the application, however, baking light maps at run-time is not always impossible. Hoffman and Mitchell managed to update the light map of an outdoor environment for the changing light conditions (Hoffman & Mitchell 2001). The baking time was reduced to 6 seconds with the help of approximations and precomputations. Although this cannot be considered real-time, it provides some sense of interactivity.

### *2.5.3 Directional Light Maps*

Conventional light maps store diffuse reflection of each texel, meaning that changes due to the view direction are not accounted for. For this reason, light maps are said to be view-independent (Iones et al. 2003, Kopylov et al. 1998). While this allows for faster rendering in interactive applications, it prevents using some of the view-dependent methods such as normal mapping. To add directionality to light maps, three light maps were used instead of one in Valve's Source Engine (Mitchell et al. 2006). Each of the light maps store light coming from one direction where each direction is a basis vector of a pre-defined basis in the tangent space.

To summarize, despite the presence of many techniques that strive to provide real-time global illumination for interactive applications, they each make compromises to trade lighting accuracy with real-time interactivity. In contrast, the technique that we propose in this paper is independent of the global illumination solution that is desired to be used. We show that with minimal increase to offline and online storage and processing costs, certain dynamic properties can be assigned to selected objects and light sources.

### *2.5.4 Light Mapping in Computer Games*

Adding global illumination (GI) support to computer games using real-time rendering is a challenging task. Only recently, with the introduction of Nvidia RTX (Burgess 2020), rendering GI effects using real-time ray tracing solutions have started to become possible. However, this is still a time-consuming process used exclusively in AAA games for high-end hardware. Even as of 2020, only 14 games are identified to support RTX-based ray tracing (Martindale 2020 (accessed September 30, 2020).

The majority of the computer games that simulate GI use light mapping. Light mapping offers important advantages for computer games such as high quality lighting and predictable run-time performance (Larsson 2010). As the light map is generated at an offline stage, the improved lighting quality does not result in reduced performance.

Many well-known game titles dating back to Quake 1 and including well-known games such as Half-Life 2 (Mitchell et al. 2006), Halo 3 (Chen & Liu 2008), Killzone 2 (Vos 2014), Mirror's Edge (Larsson & Halen 2009), and Prototype 2 (O'Conor & Blommestein 2012) use various variants of light mapping. As light maps are generated offline, most of these games use light mapping only for static objects. Dynamic objects are either rendered using lower-quality lighting schemes or other approximations such as light volumes (Kaplanyan & Dachsbacher 2010). Furthermore, mobile games almost exclusively use light maps and therefore can greatly benefit from the algorithm described in the following section.

To summarize, given the importance of lighting in computer games many methods have been devised with different use-cases, advantages, and disadvantages. Before we present our algorithm in the next section, we briefly present their properties in Table 1.

## 3 Algorithm

Our algorithm is based on an offline light map generation stage (3.2) followed by a run-time stage (3.3) during which light maps are combined and sampled. A sample scene shown in Figure 2 is used to better explain each of the steps.

**Table 1** A review of lighting techniques that are commonly used in games.

| **Method** | **Pros** | **Cons** |
|---|---|---|
| Photon mapping (Jensen 1996) | ○ Generates very realistic results | ○ Not fast enough to be used in real-time apps |
| Environment maps (Voorhies & Foran 1994) | ○ Allows baked mirror-like reflections | ○ Only works around a single point<br>○ Distance of object to surrounding environment should be high<br>○ Reflecting objects as well as reflective object should be static |
| Irradiance environment maps (Ramamoorthi & Hanrahan 2001) | ○ Allows baked diffuse reflections | ○ Only works around a single point<br>○ Distance of object to surrounding environment should be high<br>○ Reflecting objects as well as reflective object should be static |
| Forward rendering | ○ Fast enough to be executed at run-time<br>○ Supports dynamic objects and lights | ○ Low-quality lighting and shadows |
| Deferred rendering (Hargreaves & Harris 2004) | ○ Fast enough to be executed at run-time<br>○ Supports dynamic objects and lights | ○ Low-quality lighting and shadows |
| Light mapping (Abrash 2000) | ○ Offers realistic lighting<br>○ Light count does not hamper performance<br>○ Very low run-time cost | ○ Illuminated objects should be static<br>○ Additional texture memory needed |
| Directional light maps (Mitchell et al. 2006) | ○ Supports view-dependent light mapping | ○ Dynamic objects do not receive the same global illumination used for static objects<br>○ Three-times the original texture memory is used<br>○ Incurs some run-time processing costs |
| Improved light mapping (Hoffman & Mitchell 2001) | ○ Allows light map baking at run-time | ○ Not fast enough to be real-time<br>○ Only works on a specific outdoor scenario |
| Improved light mapping (Luksch et al. 2013) | ○ Reduces bake time on many-light GI algorithms. | ○ Not fast enough to be used in real-time apps |
| Improved light mapping (Luksch et al. 2019) | ○ Allows incremental baking to support quick offline iterations | ○ Not fast enough to be used in real-time apps |
| Semi-dynamic light maps (Ours) | ○ Objects and lights can still be somewhat dynamic while keeping light map benefits. | ○ Only on-off state can be toggled<br>○ Additional texture memory and processing at run-time is needed |

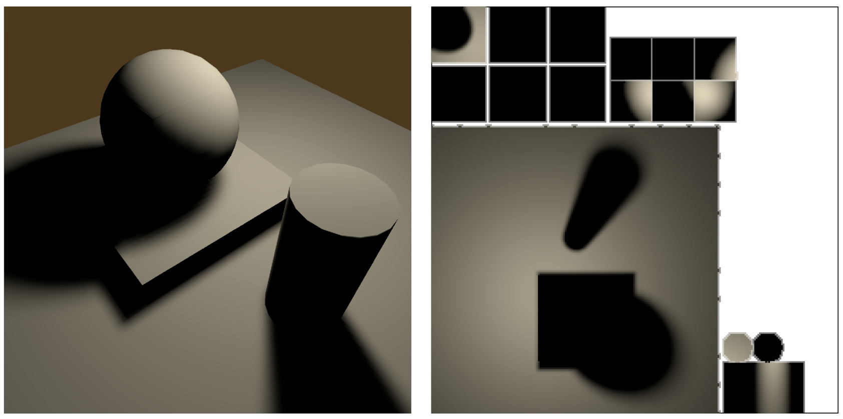

**Fig. 2** Image on the left shows the rendered sample environment composed of a plane, a box, a cylinder, and a sphere. There is one point light source above the objects with a slightly yellow color. The light map is shown on the right.

First, an overview of the method is provided followed by a detailed description of each step.

## 3.1 Overview

The input to our algorithm is a game scene in which certain objects and lights can be turned on and off (i.e. semi-dynamic) depending on the player's interactions with them. For example, the player may enter a room and turn on a desk lamp. She can then pick up certain objects from the table and turn the light off again before leaving the room. We call the first stage of our algorithm where the designer marks these objects as the marking stage (3.2.1). Next, we obtain an initial light map in which all semi-dynamic lights are off and semi-dynamic objects do not cast shadows (3.2.2). This is called the base light map. Then we compute the independent contribution of each semi-dynamic light into a different texture (3.2.3). Up to this point the shadows of semi-dynamic objects are not accounted for. To take them into account, we render a shadow map for each semi-dynamic light and object combination. Also the shadow maps with respect to the base light map are also computed (3.2.4). In order to save space, we process the shadow maps to compute what we call as light patches (3.2.5). These patches are combined into atlasses to reduce the texture count (3.2.6). This concludes the offline processing stages of our algorithm.

The run-time stage is responsible to construct a single light map in which the current state of the scene is correctly represented. First, the base light map is extracted from the atlasses (3.3.1). Then for each enabled semi-dynamic light its contribution map is created (3.3.2). This is followed by the removal of the light corresponding to the on-off states of each semi-dynamic object (3.3.3). Then these light maps are combined with the base light map using the current intensity and color of each semi-dynamic light (3.3.4). Finally, if the states of the semi-dynamic lights or objects change, we update the light map to reflect the current state.

## 3.2 Offline Stage

### *3.2.1 Marking*

As each semi-dynamic light source and object will incur processing costs, the first step of the offline stage is marking those that we want to be dynamically changed in run-time. This is a simple process in which the designer sets a Boolean flag for each light source and object to indicate its semi-dynamic property. In the sample scene shown in Figure 2, the sphere, cylinder, box, and the light source were marked as semi-dynamic. The ground plane is not marked as it will be fully static at run-time.

### *3.2.2 Computation of the Base Light Map*

As static light sources and objects will not change in run-time, generating separate maps to store their contributions is unnecessary. A traditional light mapping algorithm is used to compute their contribution as shown in Algorithm 1. The resulting light map is called the *base light map.*

**Algorithm 1** Calculating the contribution of static light sources and objects

```
1: for each light source $l$ that is marked as semi-dynamic do
2:     Turn off $l$
3: end for
4: for each object $o$ that is marked as semi-dynamic do
5:     Turn off $o$
6: end for
7: $B \leftarrow$ Generate light maps by "baking", which will contain the contribution of static
   light sources and objects.
8: return $B$
```

Note that the base light map is computed by turning on all static lights and turning off all semi-dynamic objects. In order to allow semi-dynamic objects cast shadow under this static illumination, we compute separate light maps by toggling on and off each object one by one. However, as these lights maps are almost identical to the base map except for the shadowed regions, their differences from the base map are stored using a compact representation as explained in the following sections.

### *3.2.3 Computation of the Light Contribution Maps*

In order to change the properties of a light source, its contribution to the scene's lighting needs to be known. In this step, we calculate the contribu-

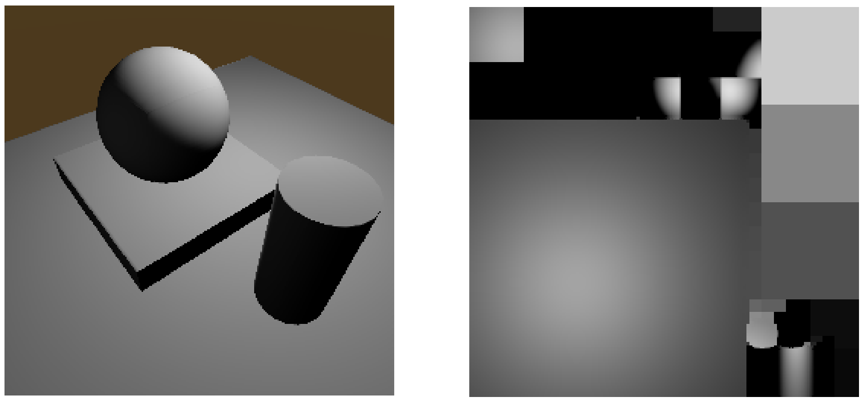

**Fig. 3** The image on the left shows the state of the scene during the calculation of the contribution map of the light source. The image on the right is the contribution map of the light source.

tion of each semi-dynamic light source $l$ to the light map of the scene. The resulting map is called the "Contribution Map of $l$" and is represented by $C_l$ (Algorithm 2). The acquired contribution map is a single channel texture that stores all the texels that are illuminated by the light source as well as the intensity of the light at each texel.

**Algorithm 2** Calculating the contribution of each semi-dynamic light $l$

```
1: for each light source $k$ in the scene do
2:     Turn off $k$
3: end for
4: for each semi-dynamic object $o$ in the scene do
5:     Set $o$ to not cast any shadows
6: end for
7: Turn on $l$
8: $l.color \leftarrow white$
9: $l.intensity \leftarrow 1$
10: $C_l \leftarrow$ Generate the light map of the scene
11: Remove 2 of the channels of $C_l$ {Since all the channels store the same data, which 2 of
    the channels to discard is unimportant.}
12: return $C_l$
```

As shown in the algorithm, the color and intensity of the light source were set to white and 1, respectively. This allows us to store the contribution maps in a more compact way. If the color of the light source was preserved in this process, the generated map would contain 3 channels. However, since each texel in this map is illuminated by the same light source, the color of each pixel will be the same. Storing the color as auxiliary data instead of including it in every pixel significantly reduces the storage costs as only a single channel texture needs to be used. Furthermore, setting the intensity of the light source to 1 allows us to directly multiply the light map texture by the desired intensity value at run-time.

### *3.2.4 Computation of the Object Shadow Maps*

In this stage, we calculate the effect of each object to the scene's illumination. With semi-dynamic lights, however, the scene does not have a single illumination state. Whenever a light source is turned on or off, the state changes. The effect of the object to the illumination of the scene should be calculated separately for each state. This quickly becomes unmanageable as the semi-dynamic light source count increases, since a scene with $n$ light sources would have $2^n$ states.

Fortunately, an object's effect does not need to be calculated separately for each state of the scene. Instead, we can store the effect of the object to the light's contribution map. In this case, the state of other lights becomes irrelevant for two reasons:

– A light source's contribution to a scene's light map is independent of other light sources.
– The amount of light of a single light source that is blocked by an object is independent of other light sources in the scene.

Therefore, for each semi-dynamic object, $n$ calculations are necessary for a scene with $n$ semi-dynamic light sources. The steps of this process are shown in Algorithm 3. The resulting map, $S_{o_l}$, represents how much light is blocked by a semi-dynamic object $o$ from a semi-dynamic light $l$.

**Algorithm 3** Calculating the contribution of semi dynamic object $o$ to semi-dynamic light $l$

```
 1: for each light source $k$ in the scene do
 2:     Turn off $k$
 3: end for
 4: for each semi-dynamic object $m$ in the scene do
 5:     Set $m$ to not cast any shadows
 6: end for
 7: Turn on $l$
 8: $l.color \leftarrow white$
 9: $l.intensity \leftarrow 1$
10: Set $o$ to cast shadows
11: $t \leftarrow$ Generate the light map of the scene.
12: Remove 2 of the channels of $t$
13: $S_{o_l} \leftarrow C_l - t$ {Illustrated in Figure 4}
14: return $S_{o_l}$
```

### *3.2.5 Generation of Light Patches*

The light contribution and object shadow maps may contain large amount of black regions as can be seen in Figure 4. This is due to the potentially

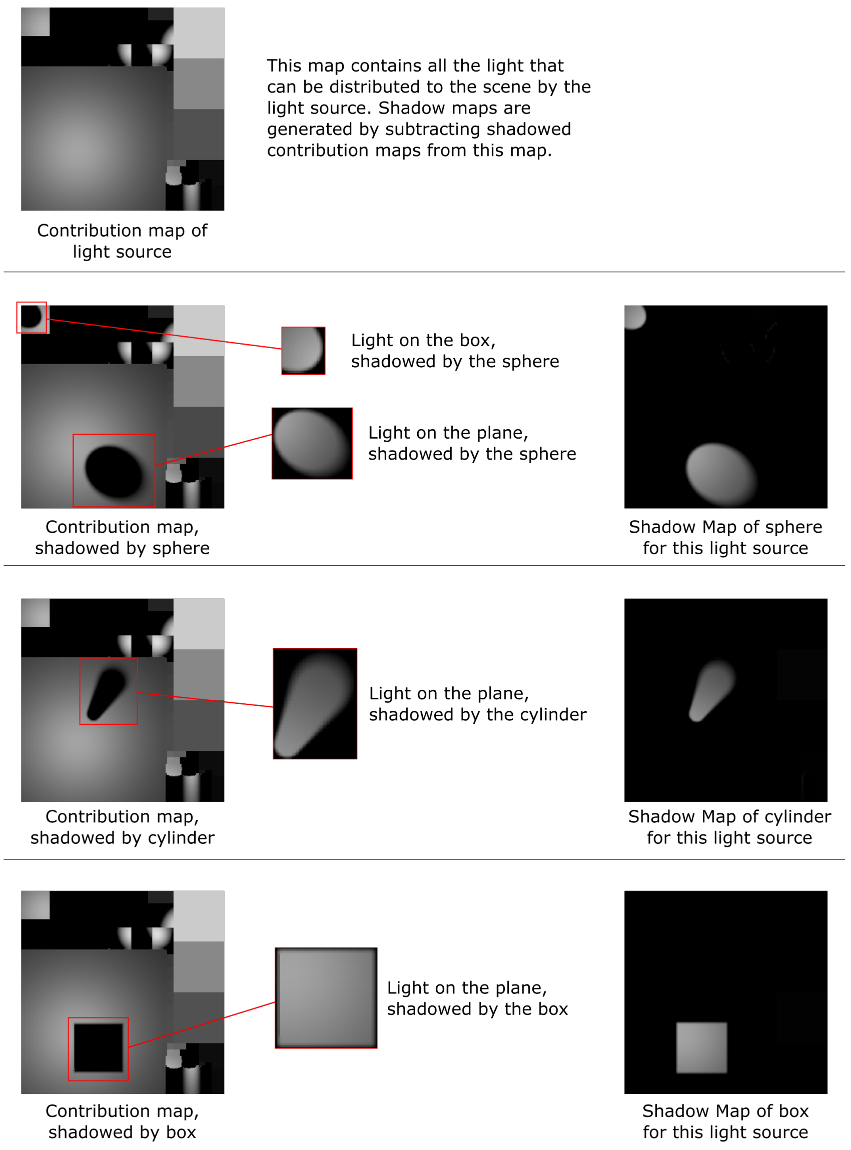
This map contains all the light that can be distributed to the scene by the light source. Shadow maps are generated by subtracting shadowed contribution maps from this map.
Contribution map of light source
Light on the box, shadowed by the sphere
Light on the plane, shadowed by the sphere
Contribution map, shadowed by sphere
Shadow Map of sphere for this light source
Light on the plane, shadowed by the cylinder
Contribution map, shadowed by cylinder
Shadow Map of cylinder for this light source
Light on the plane, shadowed by the box
Contribution map, shadowed by box
Shadow Map of box for this light source

**Fig. 4** Shadow maps store how much light is shadowed by an object from a light source. On the left, shadowed contribution maps that are used to generate the shadow maps are shown for each object. The resulting shadow maps are shown on the right.

local effects of individual lights and object shadows. For reducing the memory footprint, for each object we extract the corresponding regions in each texture and store them as small light patch textures (Algorithm 4).

**Algorithm 4** Generation of light patches

```
 1: for each map $m$ from all base, contribution, and shadow maps do
 2:     for each baked object $o$ in the scene do
 3:         let $b$ a bounding box on the light map with zero area
 4:         let $a \leftarrow o.mappedAreaOnLightMap$
 5:         for each pixel $p$ in $m$ within the borders of $a$ do
 6:             if $p \neq black$ then
 7:                 enlarge $b$ to contain $p$
 8:             end if
 9:         end for
10:         if $b.area \neq 0$ then
11:             $t \leftarrow$ create a black texture with dimensions $b.width$ and $b.height$
12:             Copy all the pixels within $b$ from $m$ into $t$
13:             Store $t$
14:         end if
15:     end for
16: end for
```

The reconstruction of base maps, contribution maps, and shadow maps from these light patches will be necessary in later stages. However, this cannot be done using only the light patches. This is because it is not known from which position each light patch was taken. Therefore, the position data of each light patch needs to be stored. This data is stored in an array as pairs where each pair consists of:

– Texture, representing the light patch,
– Position in the light map that the given patch was taken from.

In our implementation, the position represents the index of the leftmost-bottom pixel of the light patch in the light map.

### *3.2.6 Generation of Light Atlasses*

The previous step produces many small light textures. These textures are significantly smaller compared to the size of the light map. Except the base map, they also store only a single channel data. Using many small textures creates a significant overhead during rendering compared to using a few large textures, since it is not possible to take advantage of batching (Verbeiren & Lecluse 2010, NVIDIA 2004).

In this step, all the light patches are combined into large textures called *atlasses*. Because of hardware limitations on texture sizes, it is not always possible to put all the light patches into a single texture atlas. In this case, multiple smaller atlasses are generated. The process of generating atlasses is illustrated in Figure 5.

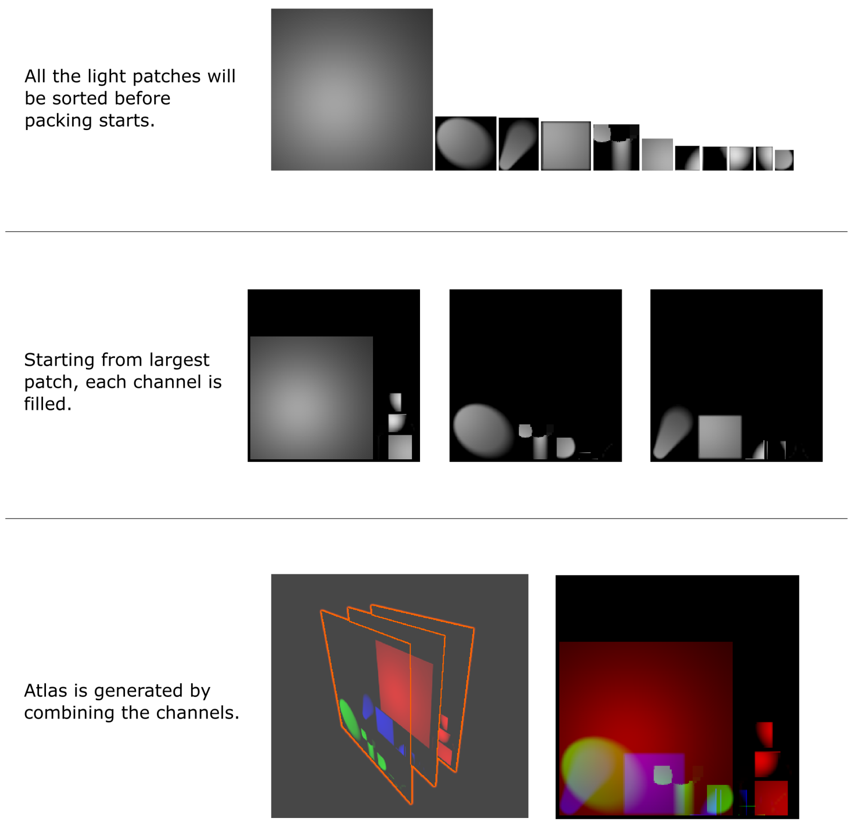
All the light patches will be sorted before packing starts.
Starting from largest patch, each channel is filled.
Atlas is generated by combining the channels.

**Fig. 5** The light patches are placed into atlasses. Multiple color channels are used to reduce the atlas count.

In this process, first an atlas whose size is equal to the largest patch size is created. The patches are then attempted to be inserted in the order of decreasing size. If a patch cannot be inserted to the current channel, a new color channel is created (we use up to 4 color channels, i.e. RGBA textures). If the maximum color channels are reached, we either create a new atlas or double the size of the current atlas. This choice is made by comparing size of the current atlas $S_a$ to the total size of all the remaining light patches $S_l$. If $2S_a < S_l$ and doubling the current atlas size will still be below the texture size limit of the hardware, the dimensions of the atlas are doubled. Otherwise, a new atlas is created. In our observation, this heuristic leads to a good balance between the number of atlasses and the size of each atlas. In later stages, atlasses will be used to access each light patch. To be able to retrieve a light patch from the generated atlasses, the following auxiliary data is stored in a configuration file:

– Index of the atlas,
– Channel containing the data of the light patch,
– Position of the leftmost-bottom pixel of the patch within the atlas,
– Size of the patch.

## 3.3 Run-Time Stage

The offline stage was responsible for creating an efficiently packed light map structure represented by one or more texture atlasses and auxiliary data to describe it. The run-time stage, on the other hand, is responsible for unpacking this representation into a single light map that reflects the current state of the scene. The resulting light map can be directly used within a traditional rendering pipeline. The only requirement is that it must be updated if the lighting or the object visibility state of the scene is changed.

### *3.3.1 Reconstructing Base Map*

Before the processing of semi-dynamic objects and light sources begins, static light sources are handled first. At this step, the base map that was created during the offline stage will be reconstructed from the atlas of light patches. This process is described in Algorithm 5 and illustrated in Figure 6.

**Algorithm 5** Reconstructing the base map

```
1: A three-channel black texture $t$ is created. The size of this texture is the same as the
   size of the scene's light map.
2: for each light patch $p$ in the atlas that corresponds to the current state of semi-dynamic
   objects do
3:     if $p$ belongs to base map then
4:         Using the auxiliary data stored at 3.2.6, determine the original position of $p$.
5:         To the determined position on $t$, copy pixels of $p$ from containing atlas as shown
           in Figure 6.
6:     end if
7: end for
8: return $t$
```

### *3.3.2 Reconstructing Contribution Maps*

Next, we reconstruct the contribution maps of each semi-dynamic light source. The process is similar to that of the base map, except a single channel texture is used as the light color is not stored inside the contribution maps. The details of this step are listed in Algorithm 6.

For simplicity, the algorithm given here stores a light map sized texture for each semi-dynamic light source. This may not be desirable since the required texture memory will linearly increase with each light source. Fortunately, since the effect of a single light source can be calculated independently of other light

**Algorithm 6** Reconstructing the contribution maps

```
 1: for each semi dynamic light source $l$ do
 2:     if $l.turnedOn \neq true$ then
 3:         continue
 4:     end if
 5:     $C_l \leftarrow$ create a single channel black texture where size of the texture is the same as
        the size of the scene's light map.
 6:     for each light patch $p$ that belongs to $l$ do
 7:         $area \leftarrow p.mappedAreaOnLightMap$
 8:         copy $p$ into area $area$ of $C_l$ as shown in Figure 6.
 9:     end for
10:     $C_l$ is the contribution map of $l$
11: end for
```

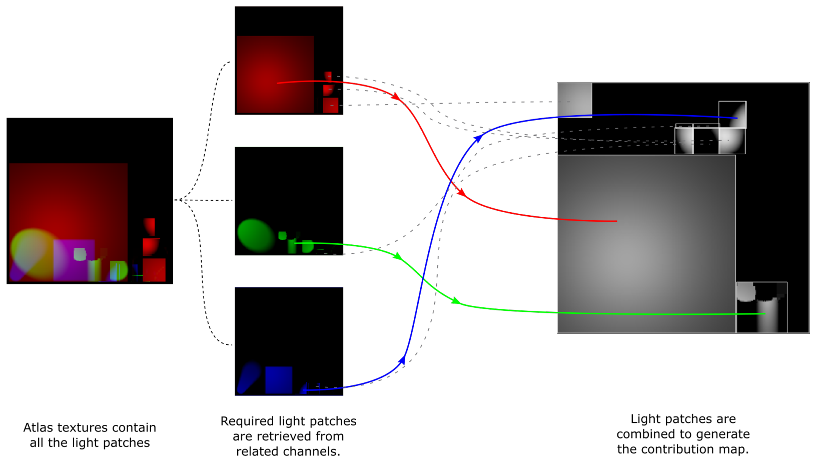
Atlas textures contain all the light patches
Required light patches are retrieved from related channels.
Light patches are combined to generate the contribution map.

**Fig. 6** Light patches from the corresponding atlas channels are placed onto a texture to create the contribution maps. This process is repeated for each semi-dynamic light source.

sources, it is possible to sequentially process them so that a total of one light map sized texture will be used for any number of semi-dynamic light sources.

This algorithm can also be implemented in other ways to take advantage of the GPU. Instead of copying values pixel by pixel on the CPU, an orthographic camera can be used to generate the same result by rendering correctly positioned quads with an additive shader. An example of this technique is explained in the next section.

### *3.3.3 Shadowed Contribution Maps*

Contribution maps store the full potential of a semi-dynamic light source. Displayed semi-dynamic objects, however, may block some of the light from these light sources. In this step, we remove the light shadowed by semi-dynamic objects to acquire the shadowed contribution maps. A shadowed contribution map for light $l$ can be represented with $sC_l$.

For efficiency purposes, we compute the shadowed contribution maps directly on the GPU. To accomplish this, we set the $C_l$ as the render target and the texture atlas from which the shadow patch will be retrieved as the texture source. The texture coordinates are computed from the auxiliary data that was stored in Section 3.2.6. The render target will be updated by drawing a quad, which is aligned with the original location of the patch in the light's contribution map. A subtractive blending operation will be used to subtract the values stored in the shadow patch from the values in the render target. This process is illustrated in Figure 7.

### *3.3.4 Generating The Final Light Map*

As the final step, we combine the shadowed contribution of each light source and the base map to generate the final light map. While doing so we also incorporate the run-time intensity and color of each light source as described in Algorithm 7.

**Algorithm 7** After the application of this algorithm, $t$ will contain the single and final light map of the scene given the current state of the semi-dynamic objects and light sources.

```
1: let $t$ be the light map of the scene
2: $t \leftarrow B$ {copy basemap onto the texture}
3: for each reconstructed shadowed contribution map $sC_l$ do
4:     for each pixel $p$ sampled from $sC_l$ do
5:         $c = p * l.intensity * l.color$ {This will convert the pixel to 3-channels}
6:         $t[p] = t[p] + c$ {Copy the calculated value onto the final light map.}
7:     end for
8: end for
9: return $t$
```

Similar to the approach explained in the previous section, operations in this algorithm can be executed with the help of the GPU. In our implementation, a shader was used to multiply and add pixels onto the target texture.

### *3.3.5 Updating The Light Map*

If the state of a semi-dynamic light or object gets toggled, it becomes necessary to update the final light map. In this case, a naïve approach would involve recreating the final light map by using the current state of the scene. However, in case only a few lights or objects are changed, repeating the entire process is not necessary. The existing light map can be updated by first subtracting the previous contribution of the affected light source and then adding its current contribution (Algorithm 8).

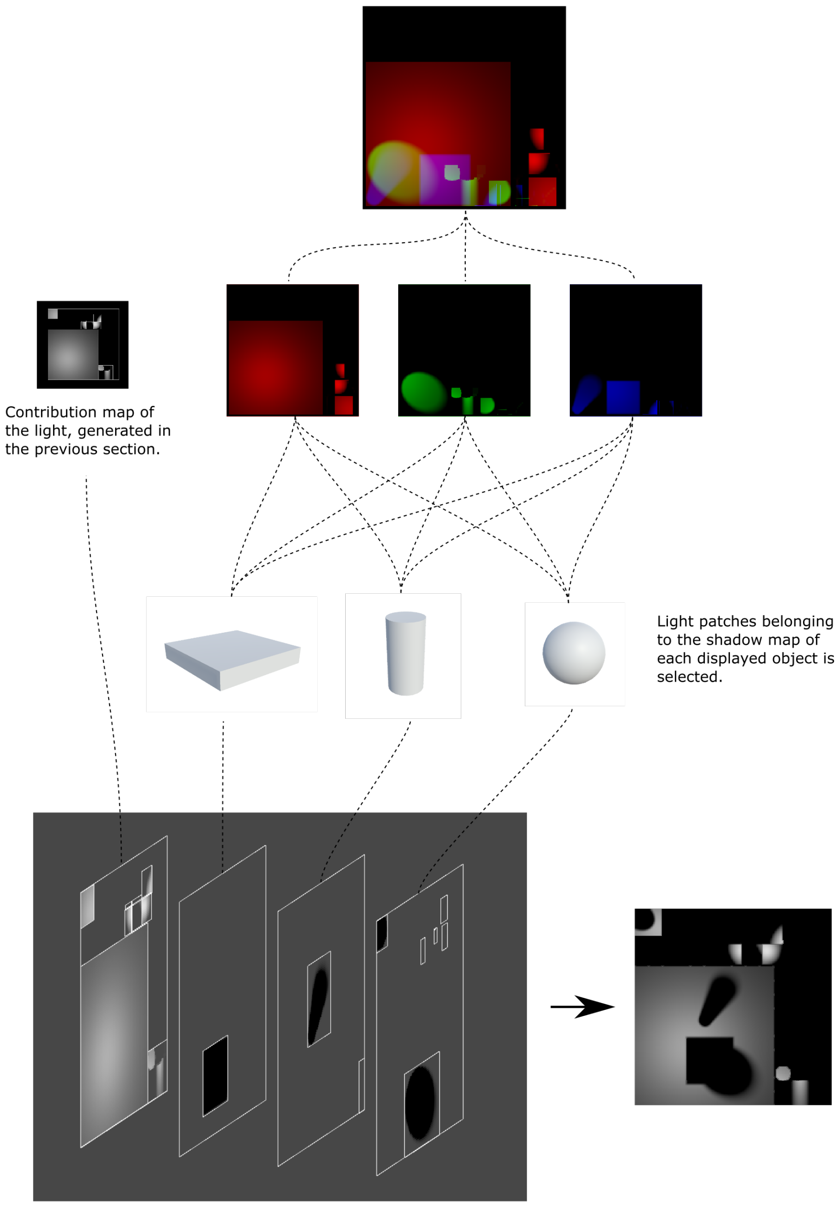
Contribution map of the light, generated in the previous section.
Light patches belonging to the shadow map of each displayed object is selected.

**Fig. 7** Depending on which semi-dynamic objects are enabled, the previously generated contribution map is shadowed.

**Algorithm 8** The algorithm for efficiently updating the light map.

```
1: Let $l$ be the affected light source
2: Let $t$ be the final light map
3: $sC_l \leftarrow$ calculate the shadowed contribution map of $l$ as explained in 3.3.3 for the scene
   state *before it was modified*
4: $sC_l \leftarrow sC_l * l.previousColor * l.previousIntensity$
5: $t \leftarrow t - sC_l$
6: $sC_l \leftarrow$ calculate shadowed contribution map of $l$ *after it was modified*
7: $sC_l \leftarrow sC_l * l.newColor * l.newIntensity$
8: $t \leftarrow t + sC_l$
9: return $t$
```

A further simplification is possible if only the intensity or color of a light source is changed. In this case, it is sufficient to compute $sC_l$ only once. The final light map can be updated by modulating it with the color/intensity difference before and after the state change as shown in Algorithm 9. This simplification can also be used for the case when a light source is turned on or off as this is merely a form of intensity change.

**Algorithm 9** The simplified algorithm that can be used if only the color or intensity of a light source is changed. Replaces lines 4 to 8 in Algorithm 8.

```
1: $c \leftarrow l.newColor * l.newIntensity - l.previousColor * l.previousIntensity$
2: $t \leftarrow t + sC_l * c$
```

## 4 Results

In this section, the results of our algorithm for different scenes will be presented. The baking times, storage size of the generated light maps, required memory, and processing costs at run-time will be compared. The input scenes used in the comparisons differ in semi-dynamic light and object count as well as in total surface area of the baked objects. All tests are executed on a desktop computer with the following specifications:

- Intel i7-4790 3.6GHz CPU
- NVIDIA GeForce GTX 770 2GB
- 8 GB RAM
- Samsung 840 EVO 250GB SSD

Snapshots of sample scenes used in the tests are shown in Figures 8 and 9. A brief description of each scene is provided below:

**Library:** A small rectangular indoor environment with all four walls covered with bookshelves. Lights are evenly distributed within the room and all the lights are at the same height from the ground. All of the shelves create some shadow on at least one other shelf. Most illuminated points in the

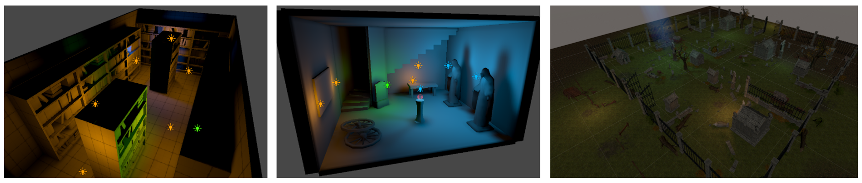

**Fig. 8** Three simpler sample scenes used for evaluations. From left to right: *library*, *dungeon*, and *cemetery*. See text for characteristics of each scene.

map receive light from 3 different light sources. The scene consists of 26000 triangles defined by 40000 vertices.

**Dungeon:** A room with large walls and small objects. There are statues, wheels, a table, a candle, a gravestone, and a painting in the room. There are 8 point light sources. The shadows of the objects in the room mostly do not overlap. The scene consists of 2200 triangles defined by 3400 vertices.

**Cemetery:** A square outdoor environment with a single large plane as the ground with gravestones, trees, statues, and rocks on top of it. There is one dim directional light source in the scene. Other light sources are point light sources and evenly distributed throughout the environment. The shadows of objects rarely overlap. Aside from the self shadowing of objects, all the shadow generated by the objects fall onto the ground plane. The scene consists of 55000 triangles defined by 75000 vertices.

**Room:** A large room with wooden furniture inside. There are bookshelves with many books, tables, boxes, chairs, foods, carpets, two bathing tubs, and barrels. This scene consists of 74000 triangles defined by 103000 vertices and is detailed enough to be a level in a mobile game or a top-down PC game.

**Inn:** Our most complex scene, which matches or even exceeds the complexity of a game level typical of mobile games. It has the same artistic style as the room scene, but it is much larger with many rooms. There is a large dining room containing tables with dishes and utensils as well as cupboards, curtains, and carpets. There are 2 bedrooms that contain beds, tables, candles, book shelves, barrels, and small boxes. Besides these, there is a cellar, two living rooms with fire places, and one bathing room with many props inside. Overall, the scene consists of 225000 triangles defined by 310000 vertices. All the light sources are either point lights or spot lights, distributed across the rooms.

## 4.1 Baking Times

### *4.1.1 Number of Semi-Dynamic Light Sources*

For a scene with $n$ semi-dynamic lights and no semi-dynamic objects, the baking process needs to be repeated an additional $n$ times. Initially, it may appear that the time spent on generating the light maps will be $n$ times more.

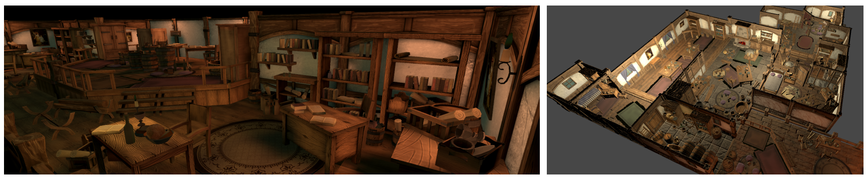

**Fig. 9** Two more complex sample scenes used for evaluations. From left to right: *room* and *inn*. See text for characteristics of each scene.

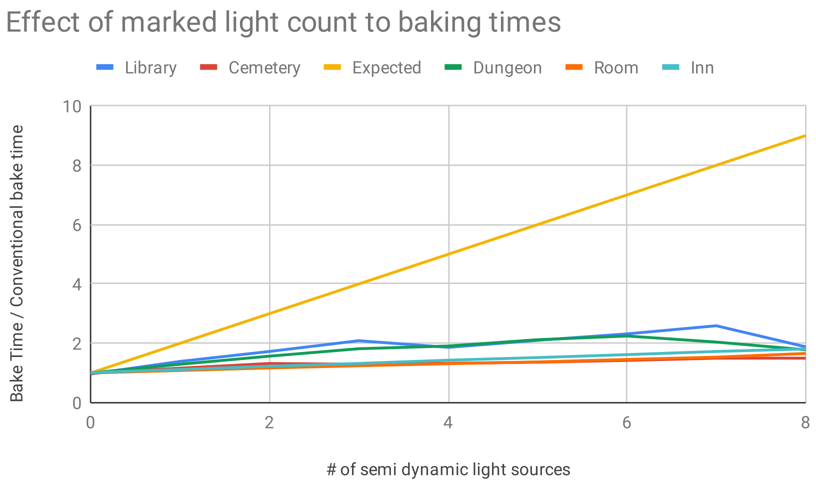

**Fig. 10** Baking times for increasing semi-dynamic light count. Note that baking times do not increase linearly as would be expected by the yellow line. The ratios of the actual baking times versus the conventional baking times remain only slightly above one.

However, this is not the case thanks to photon mapping based light map generation approaches. For the conventional case, a single baking operation requires casting $p$ photons per light. In total, $np$ photons are cast for $n$ light sources. For the semi-dynamic case, each baking process only casts photons from a single light source. Each of the $n$ operations casts only $p$ photons, which also adds up to $np$ photons in total. Therefore, the total baking time for the semi-dynamic case is not significantly more than the baking time for the conventional case as can be seen in Figure 10.

Aside from the advantage gained over photon based baking methods, there are other factors that decrease the time spent on consecutive baking operations. For instance, light map $uv$ generation for static surfaces and scene geometry processing are only needed to be done once during the first baking operation. This data can be used in the remaining bakes, further lowering the cost of additional bakings. The average times spent on each of the remaining bake operations compared to the first bake operation are shown in Figure 11. In this figure, the percentages on the y-axis represent the time ratio (scaled

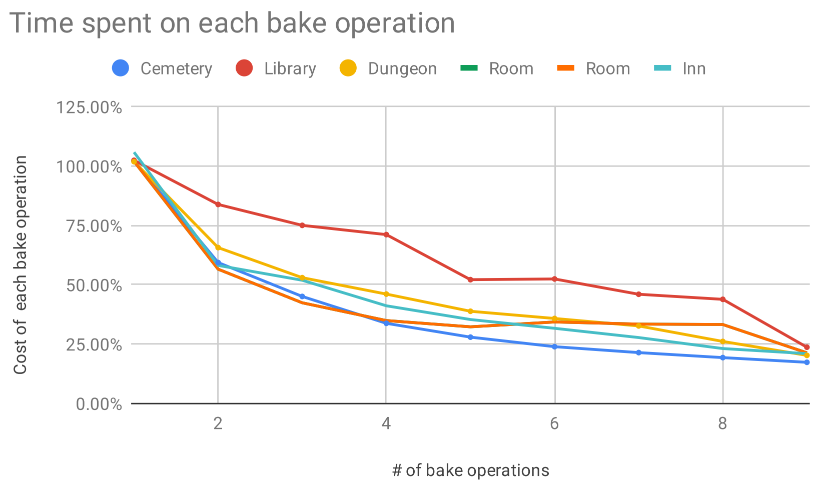

**Fig. 11** The average time spent on each bake operation decreases with each additional bake. The y-axis represents the time ratios (scaled by 100) between each pass of the semi-dynamic case and the conventional case.

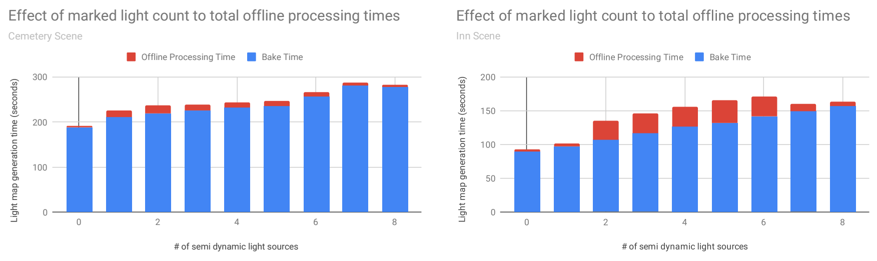

**Fig. 12** The processing costs of generated light maps (red) generally take a much smaller time compared to the time of the baking operation (blue). Results shown for the *cemetery* scene (left) and the *inn* scene (right).

by 100) between the indicated semi-dynamic baking pass (x-axis) and conventional light baking when all the lights are turned on. The initial data point is slightly above 100% because of the extra work that is done in the first pass of the semi-dynamic case compared to the conventional case.

The generation of semi-dynamic light maps does not only consist of multiple executions of baking operations – there is also the cost of processing the generated maps. This cost depends on many factors including the light count, area of overlapping shadows, surface area of illuminated semi-dynamic objects, vertex count of all the baked objects, etc. Although this cost may vary from scene to scene, it is comparatively lower than the bake times as shown in Figure 12.

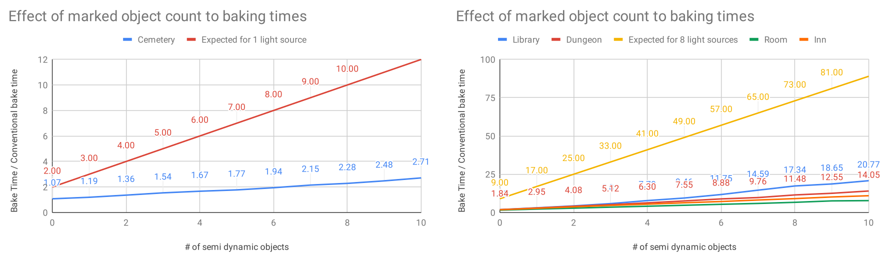

**Fig. 13** Bake times increase slowly compared to the increase in the number of semi-dynamic objects. On the left, a single semi-dynamic light source is used with up to 10 semi-dynamic objects for the *cemetery* scene. On the right 8 semi-dynamic light sources are used for the *dungeon*, *library*, *room*, and *inn* scenes.

## 4.2 Number of Semi-Dynamic Objects

The number of semi-dynamic objects is another factor that affects light map generation times. For a scene with $n$ semi-dynamic light sources and $m$ semi-dynamic objects the total baking operation count, including those for the static lights and objects is $1+nm$. However, this does not mean that the baking will take $nm$ times more, as explained in the previous section. We use two examples to illustrate this result in Figure 13. On the left, a single semi-dynamic light is used with up to 10 semi-dynamic objects. On the right, 8 semi-dynamic lights are used again with up to 10 semi-dynamic objects. As can be seen in both figures, baking times are significantly lower than what would be expected by a linear relationship.

The number of required baking operations can be reduced if object/light interactions can be determined earlier. Normally, light maps are baked for each object/light pair. If it can be determined that the object does not receive any light from a light source, this baking operation can be omitted. Measuring the distance between the light source and the object can be used to determine this interaction. Although we did not implement this extension, it can yield significant improvement for light sources that are attenuated by using the inverse square law.

## 4.3 Storage Space

In this section, we evaluate how much texture space is used in semi-dynamic light mapping vs. conventional light mapping. We first make our analysis based on increasing the semi-dynamic light count and then on increasing the semi-dynamic object count. For comparison, we take the texture memory that will be used in conventional light mapping as a reference. Our results are summarized in Figure 14.

On the left of this figure, one can see the ratio of the atlas memory in our approach to the memory that will be used if none of the lights were semi-

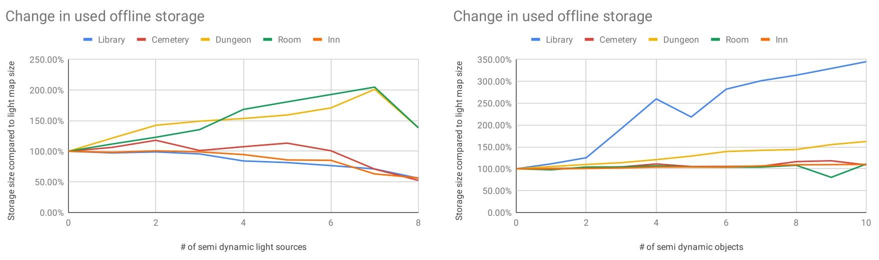

**Fig. 14** The ratio of texture memory used in semi-dynamic light mapping and conventional light mapping for both increasing semi-dynamic light count (left) and semi-dynamic object count (right).

dynamic. It is important to note that the light counts are the same in both approaches – we simply increase how many of them are semi-dynamic along the $x$-axis. It can be seen that the texture memory does not noticeably increase and it may actually decrease for some scenes as we make more lights semi-dynamic. This is because semi-dynamic light maps are stored in single channel textures whereas regular light maps require three channel textures.

On the right of Figure 14, we show the effect of increasing the semi-dynamic object count for a fixed number of semi-dynamic lights. In this case, the memory usage generally increases as we need an extra texture to store the difference between the on-off states of each object. The degree of this increase depends on how much texture area this difference occupies. For the *library* scene this increase is more noticeable due to the presence of larger semi-dynamic objects. For the other scenes, however, there is only a small increase.

Auxiliary data is another factor that increases the storage requirements. This data is used to map light patches from atlasses to light maps. In our implementation, each mapping costs 35 bytes and therefore its total memory consumption is negligible compared to the texture sizes.

## 4.4 Run-Time Processing Costs

Before the first frame and depending on the scene's state changes, the light map may need to be regenerated at run-time. Based on the number of light patches as well as light map and atlas sizes, the cost of this process may vary. In our tests, the generation of light maps took as low as 0.25 milliseconds. In complex scenes with 8 semi-dynamic lights and 8 semi-dynamic objects, the cost went as high as 2 milliseconds (Figure 15). Even though we believe that these numbers are low and can be afforded in many applications on a large variety of hardware, it may affect the smoothness of some applications running on low-end devices. However, it should be noted that updating the light map is more costly when the on/off state of a semi-dynamic object is altered. If only the color, intensity, or on/off state of a light source changes the light map can be updated efficiently as discussed in Section 3.3.5.

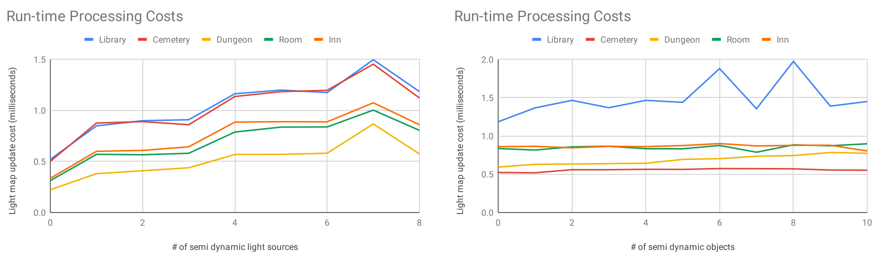

**Fig. 15** Depending on the number of semi-dynamic lights and objects as well as other parameters of the scene such as the object dimensions, the projected area of objects on light map textures, etc., the regeneration of the final light map may take different amounts of time. In our setup, this time varied between 0.25 and 2 milliseconds, which is low enough to not incur a significant cost on the frame rate.

## 5 Discussion

In the previous section, we presented several results to demonstrate the impact of our algorithm on several parameters such as storage and run-time performance. For a visual demonstration of our algorithm we refer the readers to the supplementary online materials.

Comparing our algorithm side-by-side with respect to other approaches is difficult for two main reasons. Firstly, the majority of existing light mapping solutions are implemented in commercial game engines, making it difficult if not impossible to create meaningful side-by-side comparison setups.

Secondly, our method is not an approximation – it simply allows using the same global illumination algorithm for objects and lights that can be toggled on and off. Existing approaches, however, use approximations to blend dynamic objects with the light mapped static environment. Assume that a game-object that can be turned on and off is supposed to cause color bleeding on a nearby surface. If existing solutions render this object using real-time shadings methods, their results would be devoid of this effect. For example, a static green box would cause color bleeding on a nearby surface, but a dynamic red box would not. However, in our case, the red box would also participate in color bleeding as it would be shaded using the same global illumination algorithm as all other objects. This applies for other indirect illumination effects, such as soft-shadows, as well. We demonstrate this effect in Figure 16.

The visual quality of our results could only be matched by methods that can rebake light maps and use them simultaneously at run-time. Though there are many studies on improving the light map baking performance (Luksch et al. 2013, 2019), no method has hitherto been developed to allow real-time light map baking on today's hardware without significantly compromising the quality. Hoffman and Mitchell (2001) has managed to update the lighting of their scene at run-time, but their method relied on approximations that cannot be used as a generic lighting solution. Semi-dynamic light maps, on the other hand, bring the ability to make changes to a baked scene, without having to

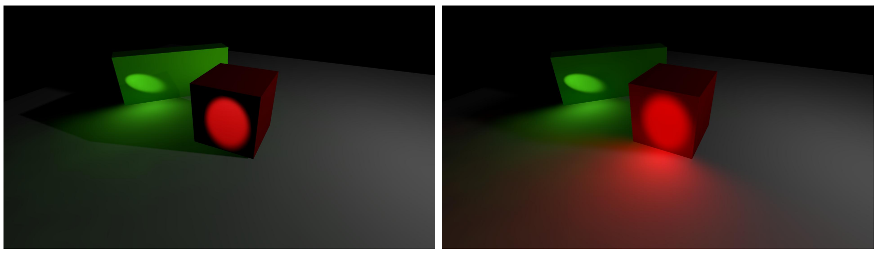
**Fig. 16** As the red cube is semi-dynamic, most real-time lighting methods would fail to reproduce indirect illumination effects such as color bleeding (left). In our case (right), semi-dynamic objects will be part of the global illumination algorithm in the same way as static objects. This allows soft shadows and color bleeding effects to work seamlessly with these objects as well.

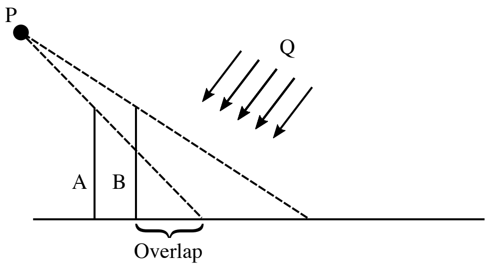

**Fig. 17** The region where the shadows of two semi-dynamic objects ($A$ and $B$) with respect to a light source $P$ overlap may have incorrect lighting if both objects are enabled at the same time and the region also receives light from another light source, $Q$.

re-bake the light maps. By simple addition, subtraction, and multiplication operations on precomputed textures, the scene lighting can be updated to the new state in a fraction of the frame time. This is achieved by using additional texture memory, but we believe that these costs are low enough to make semi-dynamic light mapping a useful technique in many applications, an example of which is given in the next section.

Finally, we would like to emphasize one limitation of our approach. If shadows of semi-dynamic objects overlap, it can cause incorrect lighting in the region where the overlap occurs. A configuration that can cause this problem is illustrated in Figure 17. Here, two semi-dynamic objects $A$ and $B$ are illuminated by a point light, $P$. The overlap region is also lighted by a directional light, $Q$, from another direction such that the overlap region should not be in full shadow. In our algorithm, the light values will be deducted twice from the overlap region if both $A$ and $B$ are enabled. However, it should only be deducted once as $A$ already occludes $B$. This would cause the overlap region to attain a lower light value than it should. To remedy this problem, one can compute different light maps for each combination of the on-off states of semi-dynamic objects for each light source. However, this could be intractable if a large number of semi-dynamic objects are present in the scene.

## 6 Application: Game Design Example for Semi-Dynamic Lights

In this section, we describe a sample game scenario that can suitably benefit from the techniques presented in this study. For this example, we refer to the *inn* scene whose properties were defined in Section 4. In general, a valuable use-case for semi-dynamic lighting is an environment where players can interact with the scene by collecting objects, putting out candles/fireplaces, or by placing objects at pre-defined locations. For the inn scene, the following scenario would allow such interactions.

The players enter the scene and are expected to find a spell book from one of the bookshelves. But almost all the lights are initially out, so they first find the candles and torches to light the rooms that they are visiting. Unlike conventional light maps, semi-dynamic light maps here allow turning on/off lights. Flickering of the light can also be simulated by animating the intensity and color of the lights. Once the players find and pick the book, it will be removed from its place, which also cannot be done with conventional light maps. According to the instructions on the book, players will then need to collect the ingredients and the tools required for the spell that are distributed across the rooms. All the collected objects will then need to be placed in a pre-defined location in front of the fireplace. Placing the objects is possible by initially baking the scene with copies of the objects already in the pre-defined locations. At run-time, the objects will start in a hidden state. When the player attempts to place the collected ingredient into their locations, the previously baked copy will be made visible, which will already have the correct lighting.

The readers may notice that different scenes and scenarios can easily be constructed to benefit from the affordances of the presented technique. We note that without semi-dynamic light maps such scenarios would need to be rendered with real-time lighting, which would require more processing power and would still yield lower quality results.

## 7 Conclusion and Future Work

We presented a method called semi-dynamic light maps to mitigate some of the limitations of conventional light mapping. Our method is based on efficiently packing a large number of different scene configurations in a texture atlas, and using this information to update the light map texture in real-time as required by lighting and object visibility changes. Through several experiments, we have shown that our method does not incur significant memory or processing costs. It is particularly suitable for applications where a real-time full lighting simulation is not affordable despite the desire for realistic and dynamic lighting.

As a natural extension, we aim to extend our work for more dynamic light mapping scenarios. It could be possible to capture the lighting of a scene for two different key positions of the same object and interpolate the results for intermediate ones assuming, for example, that the shadow will always be casted on the ground plane.

As for improving the regeneration of the light map, it could be possible to divide this process into smaller tasks to be executed over multiple rendering passes. In this case, the contribution of only some of the lights can be calculated in each pass. During the last pass where the update will be reflected to the screen, the previously generated maps can be quickly combined to generate the final light map. When implemented on a CPU, this process can also be executed in multiple threads to take advantage of thread-level parallelism.

**Compliance with Ethical Standards:** Authors declare that they have no conflict of interest.

## References

Abrash, M. (2000), 'Quake's lighting model: Surface caching', *Graphic Programming Black Book* .

Akenine-Moller, T., Haines, E. & Hoffman, N. (2018), *Real-time rendering*, AK Peters/CRC Press.

Bastos, R., Goslin, M. & Zhang, H. (1997), Efficient radiosity rendering using textures and bicubic reconstruction, *in* 'Proceedings of the 1997 Symposium on Interactive 3D Graphics', I3D '97, ACM, New York, NY, USA, pp. 71–ff.

Baum, D. R., Mann, S., Smith, K. P. & Winget, J. M. (1991), Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions, *in* 'Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques', SIGGRAPH '91, ACM, New York, NY, USA, pp. 51–60.

Burgess, J. (2020), 'Rtx on—the nvidia turing gpu', *IEEE Micro* **40**(2), 36–44.

Cabral, B., Olano, M. & Nemec, P. (1999), Reflection space image based rendering, *in* 'Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques', SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp. 165–170.

Chen, H. & Liu, X. (2008), Lighting and material of halo 3, *in* 'ACM SIGGRAPH 2008 Games', pp. 1–22.

Cohen, M. F. & Greenberg, D. P. (1985), The hemi-cube: A radiosity solution for complex environments, *in* 'Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques', SIGGRAPH '85, ACM, New York, NY, USA, pp. 31–40.

Cohen, M. F., Greenberg, D. P., Immel, D. S. & Brock, P. J. (1986), 'An efficient radiosity approach for realistic image synthesis', *IEEE Computer Graphics and Applications* **6**(3), 26–35.

Good, O. & Taylor, Z. (2005), Optimized photon tracing using spherical harmonic light maps, *in* 'ACM SIGGRAPH 2005 Sketches', SIGGRAPH '05, ACM, New York, NY, USA.

Greene, N. (1986), Applications of world projections, *in* 'Proceedings of Graphics Interface and Vision Interface '86', GI + VI 1986, Canadian Information Processing Society, Toronto, Ontario, Canada, pp. 108–114.

Habel, R. & Wimmer, M. (2010), Efficient irradiance normal mapping, *in* 'Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games', I3D '10, ACM, New York, NY, USA, pp. 189–195.

Hachisuka, T. & Jensen, H. W. (2009), Stochastic progressive photon mapping, *in* 'ACM Transactions on Graphics (TOG)', Vol. 28, ACM, p. 141.

Hachisuka, T., Ogaki, S. & Jensen, H. W. (2008), Progressive photon mapping, *in* 'ACM Transactions on Graphics (TOG)', Vol. 27, ACM, p. 130.

Haeberli, P. & Segal, M. (1993), Texture mapping as a fundamental drawing primitive, *in* 'Fourth Eurographics Workshop on Rendering', Vol. 259, Citeseer, p. 266.

Hakura, Z. S., Snyder, J. M. & Lengyel, J. E. (2001), Parameterized environment maps, *in* 'Proceedings of the 2001 Symposium on Interactive 3D Graphics', I3D '01, ACM, New York, NY, USA, pp. 203–208.

Hargreaves, S. & Harris, M. (2004), Deferred shading, *in* 'Game Developers Conference', Vol. 2, p. 31.

Heckbert, P. (1992), Discontinuity meshing for radiosity, pp. 203–226.

Heckbert, P. S. (1986), 'Survey of texture mapping', *IEEE Comput. Graph. Appl.* **6**(11), 56–67.

Heckbert, P. S. (1990), Adaptive radiosity textures for bidirectional ray tracing, *in* 'Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques', SIGGRAPH '90, ACM, New York, NY, USA, pp. 145–154.

Heidrich, W., Daubert, K., Kautz, J. & Seidel, H.-P. (2000), Illuminating micro geometry based on precomputed visibility, *in* 'Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques', SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp. 455–464.

Heidrich, W. & Seidel, H.-P. (1998), View-independent environment maps, *in* 'Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware', HWWS '98, ACM, New York, NY, USA, pp. 39–ff.

Hoffman, N. & Mitchell, K. (2001), Real-time photorealistic terrain lighting, *in* 'In Proceedings of the 2001 Game Developers Conference', Game Developers Conference 2001.

Iones, A., Krupkin, A., Sbert, M. & Zhukov, S. (2003), 'Fast, realistic lighting for video games', *IEEE Comput. Graph. Appl.* **23**(3), 54–64.

Jensen, H. W. (1996), Global illumination using photon maps, *in* 'Proceedings of the Eurographics Workshop on Rendering Techniques '96', Springer-Verlag, London, UK, UK, pp. 21–30.

Kaplanyan, A. & Dachsbacher, C. (2010), Cascaded light propagation volumes for real-time indirect illumination, *in* 'Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games', pp. 99–107.

Kaplanyan, A. S. & Dachsbacher, C. (2013), 'Adaptive progressive photon mapping', *ACM Transactions on Graphics (TOG)* **32**(2), 16.

Kautz, J. (2004), Hardware lighting and shading: a survey, *in* 'Computer Graphics Forum', Vol. 23, Wiley Online Library, pp. 85–112.

Knaus, C. & Zwicker, M. (2011), 'Progressive photon mapping: A probabilistic approach', *ACM Trans. Graph.* **30**(3), 25:1–25:13.

Kopylov, E., Khodulev, A. & Volevich, V. (1998), The comparison of illumination maps technique in computer graphics software, *in* 'Proc. of 8th Int. Conf. on Computer Graphics and Visualization', pp. 146–153.

Kristensen, A. W., Akenine-Möller, T. & Jensen, H. W. (2005), Precomputed local radiance transfer for real-time lighting design, *in* 'ACM Transactions on Graphics (TOG)', Vol. 24, ACM, pp. 1208–1215.

Larsson, D. (2010), 'Pre-computing lighting in games', *SIGGRAPH 2010 courses* .

Larsson, D. & Halen, H. (2009), The unique lighting of mirror's edge, *in* 'Game Developers Conference', Vol. 1.

Li, Y., Zhou, G., Li, C., Qiu, X. & Wang, Z. (2012), 'Adaptive mesh subdivision for efficient light baking', *The Visual Computer* **28**(6), 635–645.

Luksch, C., Tobler, R. F., Habel, R., Schwärzler, M. & Wimmer, M. (2013), Fast light-map computation with virtual polygon lights, *in* 'Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games', I3D '13, ACM, New York, NY, USA, pp. 87–94.

Luksch, C., Wimmer, M. & Schwärzler, M. (2019), Incrementally baked global illumination, *in* 'Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games', I3D '19, ACM, New York, NY, USA, pp. 4:1–4:10.

Mara, M., Luebke, D. & McGuire, M. (2013), Toward practical real-time photon mapping: Efficient gpu density estimation, *in* 'Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games', ACM, pp. 71–78.

Martindale, J. (2020 (accessed September 30, 2020)), *Here are all the games that support Nvidia's RTX ray tracing.*

Meyer, A. & Loscos, C. (2003), Real-time reflection on moving vehicles in urban environments, *in* 'Proceedings of the ACM Symposium on Virtual Reality Software and Technology', VRST '03, ACM, New York, NY, USA, pp. 32–40.

Miller, G. S. & Hoffman, C. R. (1984), Illumination and reflection maps, *in* 'ACM SIGGRAPH 1984 course notes', SIGGRAPH '84.

Mitchell, J., McTaggart, G. & Green, C. (2006), Shading in valve's source engine, *in* 'ACM SIGGRAPH 2006 Courses', SIGGRAPH '06, ACM, New York, NY, USA, pp. 129–142.

Myszkowski, K. & Kunii, T. L. (1995), Texture mapping as an alternative for meshing during walkthrough animation, *in* G. Sakas, S. Müller & P. Shirley, eds, 'Photorealistic Rendering Techniques', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 389–400.

NVIDIA (2004), 'Improve batching using texture atlases, sdk white paper'.

O'Conor, K. & Blommestein, J. (2012), Lighting the open world of new york zero for prototype 2, *in* 'ACM SIGGRAPH 2012 Talks', SIGGRAPH '12, ACM, New York, NY, USA, pp. 34:1–34:1.

Pharr, M., Jakob, W. & Humphreys, G. (2016), *Physically based rendering: From theory to implementation*, Morgan Kaufmann.

Ramamoorthi, R. & Hanrahan, P. (2001), An efficient representation for irradiance environment maps, *in* ‘Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques’, SIGGRAPH ’01, ACM, New York, NY, USA, pp. 497–500.

Ramamoorthi, R. et al. (2009), ‘Precomputation-based rendering’, *Foundations and Trends® in Computer Graphics and Vision* **3**(4), 281–369.

Sa’dyah, H., Fathoni, K., Basuki, D. K. & Basofi, A. (2018), The fundamental topics of static global illumination algorithms for 3d games, *in* ‘2018 IEEE 3rd International Conference on Communication and Information Systems (ICCIS)’, pp. 237–241.

Schäfer, H., Süßmuth, J., Denk, C. & Stamminger, M. (2012), ‘Memory efficient light baking’, *Computers & Graphics* **36**(3), 193 – 200. Novel Applications of VR.

Sloan, P.-P., Hall, J., Hart, J. & Snyder, J. (2003), ‘Clustered principal components for precomputed radiance transfer’, *ACM Transactions on Graphics (TOG)* **22**(3), 382–391.

Sloan, P.-P., Kautz, J. & Snyder, J. (2002), Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments, *in* ‘ACM Transactions on Graphics (TOG)’, Vol. 21, ACM, pp. 527–536.

Tsai, Y.-T. & Shih, Z.-C. (2006), All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation, *in* ‘ACM Transactions on Graphics (TOG)’, Vol. 25, ACM, pp. 967–976.

Verbeiren, D. & Lecluse, P. (2010), ‘Optimizing 3d applications for platforms based on intel® atom™ processor, white paper’.

Voorhies, D. & Foran, J. (1994), Reflection vector shading hardware, *in* ‘Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques’, SIGGRAPH ’94, ACM, New York, NY, USA, pp. 163–166.

Vos, N. (2014), ‘Volumetric light effects in killzone: Shadow fall’, *GPU Pro* **5**, 127–147.

Wald, I., Usher, W., Morrical, N., Lediaev, L. & Pascucci, V. (2019), ‘Rtx beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location’, *Proceedings of High Performance Graphics* .

Weghorst, H., Hooper, G. & Greenberg, D. P. (1984), ‘Improved computational methods for ray tracing’, *ACM Trans. Graph.* **3**(1), 52–69.