Realtime global illumination using compressed pre-computed indirect illumination textures

Chris Bahnsen Antoine Dewilde Casper Pedersen Gabrielle Tranchet Claus B. Madsen*

Department of Architecture, Design and Media Technology
Aalborg University
Niels Jernes Vej 14
9220 Aalborg Øst, Denmark

ABSTRACT

In this paper, we present a way to render images in real time, with both direct and indirect illumination. Our approach uses precomputed indirect illumination images, produced at certain intervals, which need not be constant. When rendering a scene, the two closest images are then interpolated and added to the direct illumination to produce the total illumination. Depending on the type of image produced, the algorithm allows a camera to move, and even objects to be added or modified at runtime to some extent. Finally, we will see that the amount of data to store and process can also be reduced using a dimensionality reduction algorithm such as PCA.

Keywords

Global illumination, Realtime Rendering, Principal Component Analysis, Phong Shading, Compression

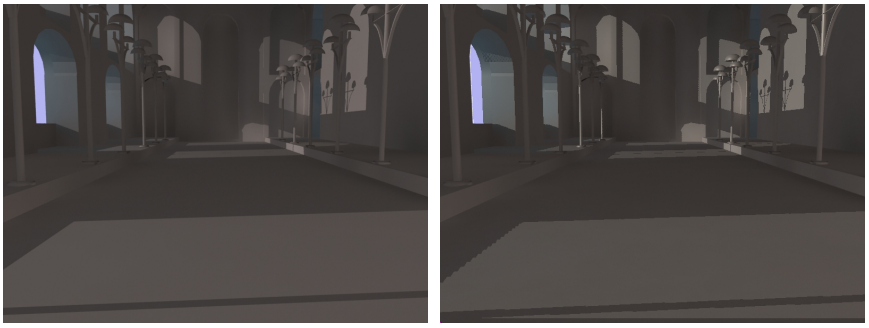
1 INTRODUCTION

Rendering global illumination in real time is a challenging problem of today's computer graphics. Although several techniques exist to compute a full global illumination model, and hence produce a realistic scene, these techniques are usually extremely expensive in terms of computation time, and so are not usable in interactive applications, even though effort has been made to make these renders as fast as possible [WKB+02]. On the other hand, algorithms than can achieve interactive sampling intervals usually do not take into account the whole illumination model for a dynamic scene and, in the best cases, only compute a few bounces for light rays [NPG03].

Current state-of-the-art techniques to render global illumination include, but are not limited to, ray tracing, photon mapping, and usual algorithms used for interactive applications such as the use of Phong shading with an ambient term to simulate indirect light. Most algorithms that aim at rendering indirect lighting efficiently, then use a combination of these techniques, together with other machine learning elements such as interpolation, clustering or neural networks. A method using clusters to render efficiently global illumination has been studied by Christensen et al.[Chr99]. For instance, interpolating images characterizing a luminance distribution can be done in a non-linear way, using the data to train a neural network [DRP09]. In a similar way, Christensen et al.[Chr99] has written about how to make a faster photon map for rendering global illumination, using a technique similar to clustering as it will group together during one stage of the algorithm photons having similar irradiance.

In this paper, our goal is to be able to render a scene with daylight using precomputed indirect illumination images. The technique we will explain allows for the light source to change dynamically on a set path, and the camera to move freely within boundaries, as well as dynamic objects to be added and moved around the scene, if we accept minor, probably unnoticeable inaccuracies in the indirect illumination. Although this paper will focus on daylight, considering the sun as the only light source, this technique is actually applicable to any scene where all the possible lighting conditions are known beforehand.

The idea our algorithm is based on is quite simple: we consider the fact that the illumination of a scene is actually given by both the direct illumination, and the indirect illumination. Since the direct illumination changes frequently when it comes to daylight, and since computationally efficient algorithms to produce it exist, we will just compute them in real time. In our test program, we used Phong shading, coupled with basic shadow mapping; any other model that gives realistic



(a) 3DSMax render (b) Our render

Figure 1: Comparison between a render of total lighting done in Autodesk 3DSMax 2011 and with our technique.

results in real time would work of course. The indirect illumination, however, will be precomputed at certain time steps and stored into images used as textures. To render a certain frame, we will then just compute the direct illumination (without any ambient term), and add the indirect illumination that we get by interpolating the two frames the closest to the one we are rendering. The interpolation to approximate results is a method that has proved itself, for instance in [RGS09] where it is discussed that computing an approximating physically plausible illumination over physically correct illumination takes less computation time and gives, as the name states, very plausible results. Other methods, including [RGKS08], also include compressed pre-computed information in order to speed up rendering (in this case, precomputed depths and coherent surface shadow maps).

So, the problem can be narrowed down to finding the right sampling interval for indirect illumination, and reducing the amount of data to process as much as possible by compressing the precomputed frames. Defining a good sampling interval is important so that the preprocessing is not too long, as we try to avoid rendering similar images, but the quality of the interpolation is still acceptable. Reducing the amount of data is important both for disk space issues, and to reduce the loading time.

In the next following sections, we will explain the algorithm and the above-mentioned issues in more details. In section 2, we will give a general overview of our algorithm. In section 3 we will explain our method in further details, including the possible refinements (compressing the data using PCA, and defining a dynamic sampling interval). Finally, section 4 will be dedicated to some results analysis, and considerations for future inspection. All our example images feature an indoor scene inspired by the castle of Koldinghus in Denmark, as the first application considered for the method is a virtual tour of the castle.

*cbahns08 | adewil11 | cped08 | gtranc10@student.aau.dk, cbm@create.aau.dk

2 OVERVIEW OF THE APPROACH TAKEN

As explained before, the main idea of our algorithm is to separate the direct illumination from the indirect lighting. Since the direct illumination can be computed easily and holds high-frequency data which makes it hard to compress, we will use off-the-shelf algorithms to calculate it in real time; in our demo program, Phong shading coupled with basic shadow mapping. The indirect illumination, however, will be precomputed at certain timesteps and stored in a compressed form. When rendering a frame, we will then interpolate between the two closest frames computed, and add that indirect illumination to the direct light. Figure 1 shows the result we get, compared to a scene rendered in a standard 3D modeling program that uses ray tracing and photon mapping.

However, as we would like to compress the indirect illumination images, we would like to get rid of as much high-frequency data as possible. This is the reason why we also compute the contribution from the skylight separately. Actually, in our approach, we will consider the skylight as being a constant contribution that just has to be scaled depending on the time of the day. In our demo program, we use a precomputed image for the skylight, which we scale depending on the position of the sun. This gives acceptable results for our application but, of course, any other technique that gives good results and can run in real time can be used, such as [NJTK95] where a model of skylight is built and its illuminance is calculated.

Figure 3 shows the different elements needed. The final render of an arbitrary frame will then be the sum of:

- The direct illumination, computed in real time.
- The indirect illumination without skylight. This is obtained by interpolating between the two precomputed frames closest to the one we are rendering.
- The skylight, precomputed according to the lighting model of your choice.

Aside from compression issues, removing the direct illumination from the precomputed frames also has other advantages. For instance, it allows for dynamic objects to be included into the scene with only minor errors in indirect lighting. Depending on the type of object, these errors might not even be noticeable to the untrained, naked eye. However, compared to the scene with the teapot rendered in a 3D modelling program, the color bleeding and the blocking of indirect illumination of the teapot is absent in our render. Figure 2 shows an example of such a scene, where a teapot has been added dynamically.

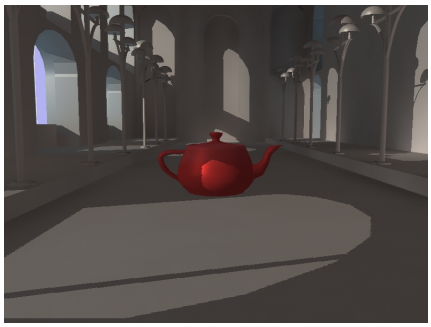


Figure 2: A scene with global illumination rendered by our technique, where a teapot has been added dynamically.

3 DETAILS ABOUT THE METHOD USED

3.1 Basic idea

Implementing our approach in an application is done in two steps: first of all, pictures for the indirect lighting must be generated, at some point before the application starts. Only after that step can we actually render anything. In the next subsection, we will focus on the issues related to the generation of the images. Let us just assume that we were able to produce images containing the indirect illumination (without skylight). We will also assume for now that these images are uncompressed and taken at regular intervals; we will deal with compression and dynamic sampling interval issues in the next section. In general, a value that produces high-quality results while still keeping the number of precomputations reasonable, is a framerate of one picture every five minutes – which, for a sequence of a full day, gives 288 images to render.

So, let us assume that we have a set of images containing information about the indirect illumination. In our tests, we used bitmap images, which gives the advantage of being encoded over 24 bits, hence containing more information than compressed file formats. To render the scene, we will use a custom set of shaders that will compute the direct illumination, and "paste" the indirect illumination and the skylight on it.

Loading images from disk takes a significant amount of time. Because of that, loading the precomputed images on-the-fly results in a significant drop in frame rate, while still achieving interactive frame rate though. So, if the amount of data to load is acceptable (which it should be once we introduce compression), and if the application requires fluid animations, it might be beneficial to consider pre-loading all the images in memory at startup, keeping pointers to the different data, and accessing them when needed.

The next step is to take advantage of the GPU's computing capabilities to interpolate and render the scene. Let us say that we want to render a frame for an arbitrary time. We will first find the closest precomputed images (one before the current frame, one after), then we will interpolate them to get the correct luminance value. With pictures taken at a fixed frequency, the index of the image just before the current frame would be

$$N = \text{floor}\left(\frac{H \times 60 + M}{\Delta T}\right) \tag{1}$$

where H and M represent the time that is represented in the frame, ΔT is the time between two precomputed images, and $N+1$ is the image right after the current frame.

Once the two images closest to the current frame are retrieved, they are then passed as textures to a specific fragment shader. This fragment shader calculates the fragment color by interpolating the two frames and adding the result to the direct illumination and skylight contributions, as shown by Equation (2). In our test renders, we used basic linear interpolation, which gives plausible results since the indirect illumination has only low-frequency information. Other kinds of interpolation will be discussed in section 4.2. The equation used to compute the total luminance of a pixel is:

$$\begin{aligned} L(t) = {} & L_{\text{direct}}(t) + L_{\text{sky}}(t) + (1-\alpha)\,L_N(t) \\ & + \alpha L_{N+1}(t) \end{aligned} \tag{2}$$

where L_N and L_{N+1} are texels retrieved by sampling the textures calculated in Equation (1). Since we currently have a fixed interval between each precomputed frame, calculating the value for α is quite straightforward. The computation is given in Equation (3).

$$\alpha = \frac{(M \bmod \Delta T) - \text{offset}}{\Delta T} \tag{3}$$

where offset is the time (in minutes past midnight) of the first frame.

The next issue to consider is how to do the texture lookup, and consequently how to generate the illumination textures themselves. Basically, three different methods can be considered, depending on the type of application:

Figure 3: The different elements that make up the final result. Left: direct illumination (computed at runtime). Center: indirect illumination (precomputed at several time steps). Right: skylight (precomputed once and scaled)

- If the camera cannot move at all inside the application, simply pasting the texture onto the final render might be enough. In that case, the illumination texture is a simple 2-dimensional render as shown in Figure 4.
- If the camera can rotate but not move, using cube maps might be an option. In that case, the illumination texture is a cubemap-looking texture or set of textures (one for each direction).
- In the general case where the camera is allowed to move freely, using texture atlases might be the option. In that case, the texture is "pasted" onto the geometry, and the files are a list of textures.

All options, however, have their shortcomings. The first option makes having reflective objects in the scene difficult, as rendering the reflection cube map for such objects would be tricky. Indeed, when rendering the cube map for a particular object, the illumination image for that specific render has to be used. In other words, this technique needs 6 sets of images for each reflective object (unless indirect illumination is disabled in the cube map of course). Furthermore, the illumination cube map for that object has to be aligned with the actual render of the scene, which is not trivial.

Figure 4: Illumination texture used if the camera is not allowed to move

The second technique has the same problem as the first one, plus the fact that a cube map is also used for the actual rendering of the scene. Furthermore, since a cube map takes six times as much space as a regular 2D texture, the amount of data to precompute rises by almost as much.

Thus, in the end, it turns out that the third method should be the best suited in most of the applications. It does not have problems of alignment, and the amount of data to process is acceptable if the scale of the scene is limited, as in this example. However, the amount of memory for storing the frames will increase with the complexity of the scenes and thus be an expensive solution for dense scenes. In the specific case where the camera is not allowed to move, then the first or second method might be best.

In the case of our demo program (and hence all the pictures of this article), we did not allow the camera to move, and so chose to implement the first technique. The results we got could easily apply to another technique though, as the main change is the way lookup coordinates are computed, and illumination textures are generated.

Up till now, we assumed that we had indirect illumination textures available for several time steps. It is now time to define how to get those images. This is what the next subsection will be about.

Tips and tricks for rendering images with indirect illumination only

Obtaining pictures with indirect illumination only has to be done before the program starts running. Since our method's only requirement is that indirect illumination pictures are available at runtime, the way these images are obtained is actually flexible: any method that provides the right output would work – and finding the optimal way to produce them might be an optimization in itself. As finding such an algorithm is a different subject, we will only present here the simple method we implemented, and that gives good results, while it might be overly simplified.

For producing the indirect illumination pictures the commercial modeling tool Autodesk 3DS max® is used. The model of Koldinghus used in the demo program is modeled in this tool that can also produce rendered images with lighting. The settings used are set up to be as similar as possible to the demo program so as to enable a comparison of the resulting illumination later

on. These settings are with regards to light and material color and intensity etc. A point light source is inserted in the scene to act as the sun. A point light is chosen to get the same shadow casting as in the demo program where a point light is used in the shadow map pass and in the direct light computations. The sun is animated both in terms of movement where it's following a path resembling the sun's passing on a chosen day, and in terms of intensity to give a realistic sunrise and sunset. The skylight is added to render its contribution to the global illumination. The images for the two light sources are rendered separately as we want to add them to the final illumination separately. Firstly a render with the point light including only direct light is done, which is a sequence of images corresponding to a day. Next global illumination for this light source is rendered and another sequence is produced. These two sequences are subtracted to obtain the indirect illumination only, which is the image sequence that will be used as textures.

One image is only rendered for the skylight as it is, as mentioned earlier, a static light source which we just multiply by an intensity value. All the rendering is done with the state-of-the-art Mental ray renderer which gives physically realistic looking results of global illumination using photon maps and final gather to compute diffuse indirect illumination.

3.2 Refinements

Compressing the textures using PCA

We apply the principal component analysis (PCA) on the indirect illumination frames in the time-lapse sequence in order to reduce dimensionality and the amount of storage required to save the individual frames. With a resolution of 800×600 pixels and a frame every five minutes, this results in a uncompressed file size of about 400 MB for the total sequence. We use the PCA to represent as much of the uncompressed information as possible by maximizing the variance in the time-lapse sequence onto a lower-dimensional subspace.

The PCA is implemented on only 180 of the total 288 frames to save computational time and requirements. The discarded frames are all placed in the night and evening and contain only immerse darkness. The image vectors for the different frames are put together to produce 480 000 vectors with 180 dimensions, a vector for each pixel. When the PCA is applied on those vectors, we get 180 eigenvectors with variance as shown in Figure 5.

Taken into account that the variance is plotted onto a semi-logarithmic plot, we see that the amount of variance contained by the single eigenvector is dramatically decreasing as we go through the eigenvectors. With only five eigenvectors, we may thus capture 95 % of the total variance, and if we double the numbers of eigenvectors to ten, they contain 98.6 % of the variance in the time-lapse sequence. In this case, we get a compression of $94,5\%$.

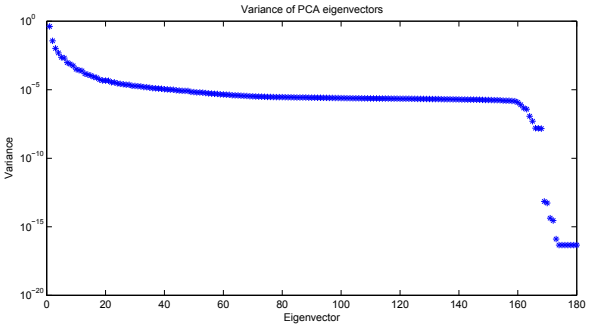


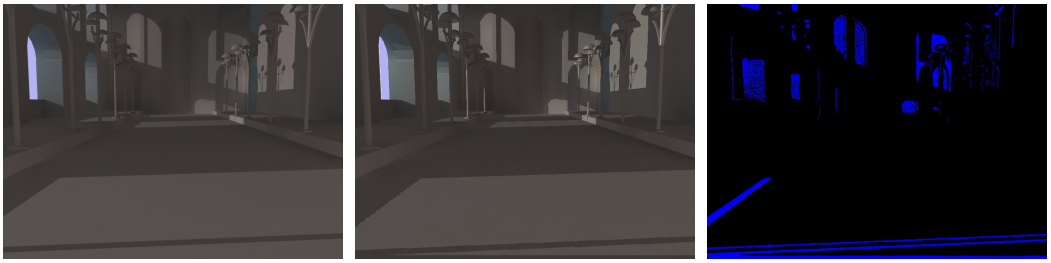
Figure 5: Variance of eigenvectors used for PCA

In order to get the image vectors onto the ten eigenvectors, the transposed of the assembled frame vectors are projected on each eigenvector. To restore the images, the reverse process is simply executed. With ten eigenvectors with a dimension of 180 and ten projected vectors with a dimension of 480 000, we are able to reconstruct the image sequence. The 121st reconstructed frame of this sequence is shown in Figure 7 on page 7 together with the original, with little noticeable difference. Further comparison of the frames is found in section 4.3 on page 8.

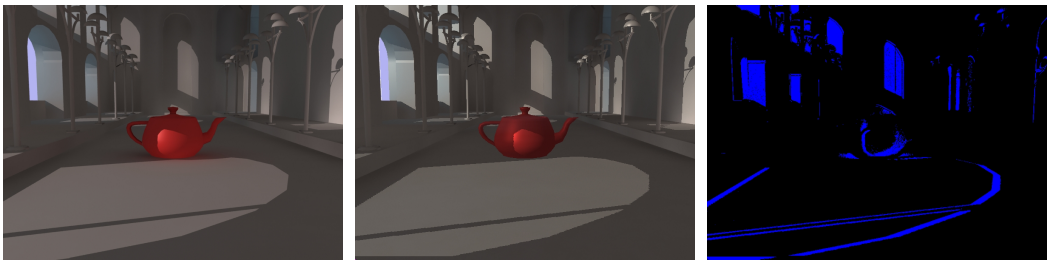
Dynamically select the sampling interval depending on light changes

As mentioned we want to have a realistic looking passing of a day with regards to the indirect illumination with the least amount of data. If we can discard some of the images which hold the indirect illumination, and instead interpolate images to produce the illumination, so that it is not noticeable for the user, then we can achieve exactly this lower data amount that is desirable. For this a algorithm has been produced which can be explained in some simple steps;

1. First we have two equally big intervals with images representing the day.
2. The first and the last image in each interval is linearly interpolated to a middle frame and the produced images is compared to the first and last one of the interval.
3. If the interpolated image is noticeably different from the first or last in the interval, the interval is split in two, and a new frame is rendered for the time of the interpolated image, the middle frame. This new frame becomes the first and last image in the two new intervals.
4. This continues for every interval produced until the images needed are rendered. With this relative simple algorithm you save the computation time of doing big and complicated renders of global illumination by only rendering the images needed to make



(a) Reference scene (b) Our render. All the objects are taken into account (c) Perceptual difference between the two results



(d) Reference scene (e) Our render. The teapot is added at runtime (f) Perceptual difference between the two results

Figure 5: Differences between our render and a reference. Top: all objects are taken into account for the indirect lighting. Bottom: dynamic objects are added

the indirect illumination look realistic for the sequence of the day.

The way the interpolated frame in an interval is compared to the first, is by utilizing a tool called PerceptualDiff [YCS+04]. This tool uses knowledge about the just noticeable difference [SJ10], in the field of color, to calculate the number of perceivable different pixel in the two images and then evaluates if a person is able to see a difference between the two images.

For the demo program the algorithm is used on the sequence of indirect illumination images which contains images for every five minutes. This reduces the sequence density and thereby the data loaded by the demo program, which also reduces the loading time on startup. Since the images now don't have the same spacing between them, an application like the demo program, using them also requires this info to interpolate the sequence correctly.

On Figure 6 the result of applying the algorithm is seen. The pictures at night are left out because they are totally black. We see that there is a lot of activity at sunrise and sunset, and also at two instances in between where the light source representing the sun goes behind a wall with no windows (mimicking sunrise and sunset). The peaks around 900 minutes are due to increased activity at the right wall with the lamp posts. The amount of images needed per hour varies from twelve, meaning that every frame in the sequence needs to be rendered, to intervals of 30 minutes which gives only two per hour.

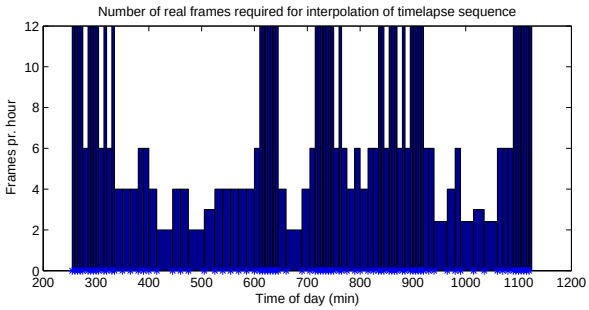
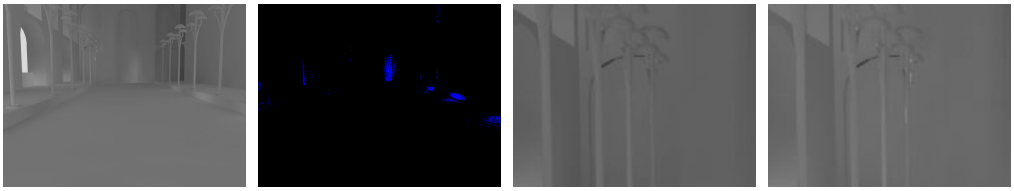


Figure 6: The dynamic sampling interval through the day

4 RESULTS AND DISCUSSION

The very first and most trivial way to analyze the results obtained is just to look at the rendered pictures and judge, with the naked eye, how they look. Figure 1, in section 2, shows a render of a random frame in our program, and compares it to the same scene, rendered in a state-of-the-art 3D program. At first glance, we can easily agree on the fact that both images look similar.

This, however, is not a sufficient way of analyzing results, and we will need a way to quantify the difference between the two techniques. In this section, we will consider the state-of-the-art render as the standard, and compare our results to this reference. For that, we will, like in 3.2, use the notion of perceptual difference [YN04], which describes the difference between two images in terms of how a human would perceive it. First of all, we will just analyze our basic renders



(a) Original frame (b) Perceptual difference of restored frame with 10 eigenvectors (c) Blow-up of region with 10 eigenvectors (d) Blow-up of region with 20 eigenvectors

Figure 7: Comparison of information loss produced by PCA. A gamma correction of 2 has been applied to the frames.

(with and without objects added dynamically), then in the next sections we will analyze the impact of the various compression methods on the final result.

4.1 Perceptual difference between our renders and a reference

As explained above, we will focus on the perceptual difference, in normal conditions, between an image rendered in our program and an image rendered using the commercial program 3DSMax by Autodesk. For that, we will use [YCS+04], a utility program that computes the perceptual difference between two images. Its implementation is highly based on [YN04].

Figure 5 summarizes the results for two types of scene:

- The top row shows a scene where all the objects have been taken into account during the preprocessing stage, so that the indirect illumination is correct.
- The bottom row shows a scene where the red teapot has been added during runtime, i.e. it hasn't been taken into account while calculating the indirect illumination. In the reference picture, however, the whole scene is subject to indirect illumination, and so the teapot is included in it.

As we can see, in the first case, the perceptual difference is mainly due to minor differences in the way both scenes are rendered, because the light's intensities can't exactly match. The perceptual difference is then mainly due to these discrepancies in direct illumination, as well as a small geometrical mismatch in the areas covered by shadows. All the regions in shadow, lit by indirect illumination only, present no noticeable difference.

Unsurprisingly, results are similar when an object is added dynamically. In Figure 5f, we see that there are only few errors on the teapot itself as well as the nearby floor where the indirect illumination is approximated; the other discrepancies are due to the setup in both environments, as explained above.

4.2 Choosing the right interpolation function

One of the criteria to decide on when using our technique is: "what interpolation function to use?". In theory, that choice should influence the quality of the final render, and you might want to choose the most accurate interpolation method, to get the best results. In practice, however, we observe that this is not the case, as most interpolation functions need a lot of computation in order to choose the non-linear parameters, for results that are only slightly better than linear interpolation.

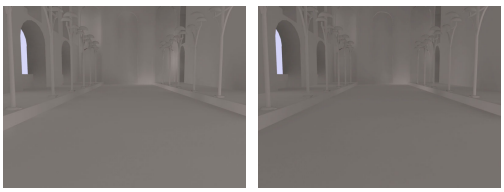
The main problem with most interpolation methods is that they need to run on the whole dataset to compute the parameters needed. In our work, that means that most images must be preprocessed to generate the interpolation function. Linear interpolation, however, only needs two images to compute a third one, which is the easiest function, albeit not very accurate.

Accuracy, however, is not really an issue with our dataset. As explained in section 2, we weeded out all the high frequency data from our images to only keep the low-frequency indirect illumination. That means that indirect illumination only present small changes between two frames, and so interpolating them should not provide major artifacts. Furthermore, if we use the Just Noticeable Difference as a threshold, then the interpolation should present no noticeable artifacts. Figure 8 shows results for such an interpolation graphically. A perceptual difference test on these images shows that no pixel is visibly different, although the arithmetic difference between the two images shows some minor interpolation errors.

4.3 Quality loss induced by PCA

In section 3.2, we described the method of reducing the dimensionality of 180 frames into only ten vectors. The 121'st frame in the sequence has been restored and contains 99.6 % of the variance of the original frame. However, as it might be seen from a the image of the perceptual difference in 7b and as a closer inspection of the frames reveals, differences in color nuances and transitions appears. This is apparent at the back wall of the church and on the floor where some transitions

from light to darker grey is smoother in the original than the restored frame and other transitions does not appear. Further differences occur in the lamps when a hardly visible, small pocket of white in the distant lamp is converted to grey. A blow-up of this region is shown in the third frame of Figure 7.



(a) Original frame (b) Interpolated frame

Figure 8: Comparison of a random frame and its interpolated counterpart.

In order to acquire a more vivid representation of the frame with more transitions and nuances imminent, the number of vectors used for compressing the frames is doubled to 20. The overall improvement of the sequence may be small however, as the variance only increases to 99.83%, a relative improvement of approximately 0.23 %. This small improvement in the total variance captured is visible on the blown-up frame on figure 7d as the original, white ray of light on the distant lamp now is apparent. Additionally, with this increase in the number of eigenvectors, there are no longer a perceptual difference between the precomputed and the estimated frames.

5 CONCLUSION

In conclusion, we see that our algorithm gives good results, and allows using total indirect illumination in an interactive context. If combined with a texture atlas to provide the indirect illumination, it allows for a camera to move freely within boundaries, and if programmed carefully, dynamic objects could also be added with only minor errors. Furthermore, selecting a varying amount of frames across the sequence, and compressing them with an algorithm such as PCA, allows getting similar results while greatly reducing the amount of data to store and process.

The main area of improvement of our algorithm, that could be the topic of another research work, is the way the indirect illumination textures are precomputed. Finding a way to have indirect illumination images precomputed as fast as possible might be interesting in most applications.

REFERENCES

[Chr99] Per H. Christensen. Faster photon map global illumination. *Journal of graphics tools*, 4/3:1–10, 1999.

[DRP09] Samuel Delepoulle, Christophe Renaud, and Philippe Preux. Light source storage and interpolation for global illumination: A neural solution. *Studies in Computational Intelligence*, 240/2009, 2009.

[NJTK95] Eihachiro Nakamae, Guofang Jiao, Katsumi Tadamura, and Fujiwa Kato. A model of skylight and calculation of its illuminance. *Image analysis applications and computer graphics*, 1024/1995:304–312, 1995.

[NPG03] Mangesh Nijasure, Sumanta Pattanaik, and Vineet Goel. Interactive global illumination in dynamic environments using commodity graphics hardware. In *11th Pacific Conference on Computer graphics and Applications*, pages 450–454, 2003.

[RGKS08] Tobias Ritschel, Thorsten Grosch, Jan Kautz, and Hans-Peter Seidel. Interactive global illumination based on coherent surface shadow maps. In *Proceedings of graphics interface 2008*, 2008.

[RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 2009.

[SJ10] Melissa K. Stern and James H. Johnson. Just noticeable difference. *Corsini Encyclopedia of Psychology*, pages 1–2, 2010.

[WKB+02] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray tracing. In *EGRW '02 Proceedings of the 13th Eurographics workshop on Rendering*, 2002.

[YCS+04] Hector Yee, Scott Corley, Tobias Sauerwein, Jeff Breidenbach, Chris Foster, and Jim Tilander. Perceptual image diff. http://pdiff.sourceforge.net/, 2004.

[YN04] Yangli Hector Yee and Anna Newman. A perceptual metric for production testing. In *SIGGRAPH '04 ACM SIGGRAPH 2004 Sketches*, 2004.