A FLEXIBLE REAL-TIME FRAMEWORK FOR PRE-CALCULATED GLOBAL ILLUMINATION SOLUTIONS

Markus Lipp, Stefan Maierhofer and Robert F. Tobler
VRVis Research Center
Donau-City-Str. 1/3, A-1220 Wien

Keywords: real-time rendering, shader programming, global illumination.

Abstract: A framework for real-time application of view-independent pre-calculated global-illumination solutions, retaining the ability to adjust the intensity of light-sources after pre-calculation, is described. High dynamic range scenes are fully supported. The framework is highly flexible both in terms of light-source numbers and target hardware: both high-end graphics-cards and older models are supported. Furthermore it is orthogonal to the global-illumination solution method and the chosen tonemapping operator, and therefore easy to implement into existing applications. Implementation details to both minimize memory footprint and maximize performance and flexibility are given. The performance of this framework has been evaluated in the context of an existing CAD application.

1 INTRODUCTION

Global Illumination (*GI*) Solutions for computer graphics are inherently expensive to calculate, as the complex interaction of light with the scene has to be accounted for. Therefore direct application to Real-time Graphics is only possible if restrictions on the scene are made. An often used approach is to define the scene and light sources, as well as the light intensities, as static, and precalculate a global illumination solution. If the precalculations are view independent, it is possible to apply the precalculated GI solution in Realtime to the scene. The obvious drawback of this approach is the inflexibility: After the precalculations there is no way to change parameters of the scene without doing the expensive precalculations again.

A straightforward extension to view independent GI precalculations that lifts the restriction of static light intensities is done the following way: The GI precalculations are not executed for all lights at once, instead they are done for subsets of all lights, called light-groups. This creates one solution of the GI problem for every light-group. Those separated solutions can then be combined in real-time with scaling factors applied to each GI solution, enabling flexible light-intensities adjustments.

The main focus of this paper is to describe a framework to allow this extension to be used in a highly flexible and efficient fashion: There is no restriction on the number of light-groups, and there are only a few restrictions on the target graphics-hardware. Techniques to reduce the memory footprint are described, as well as implementation and optimization details for various graphics-hardware.

1.1 Previous Work

Three main research-areas act as a base this framework: The first area, view independent GI, describes how the precalculations are done as input for the framework. Many methods for view independent GI preprocessing have been proposed, with Photonmapping (Jensen, 1996) being the method of choice as base for this work. Every GI method that allows the output of the precalculated data to textures can be used as base for this framework. Therefore the choice of the GI solution orthogonal to the described framework, as long as the mentioned restriction holds.

The second area are methods of real-time rendering. To be more specific, high dynamic range (HDR) texture storage and rendering, as well as programmable hardware shading are used in the framework. HDR texture storage is required to store the results of the GI pre-process. Only recent graphics-hardware allow to use the straightforward way: store the results with floating point precision. Older hardware

Lipp M., Maierhofer S. and F. Tobler R. (2006).
A FLEXIBLE REAL-TIME FRAMEWORK FOR PRE-CALCULATED GLOBAL ILLUMINATION SOLUTIONS.
In *Proceedings of the First International Conference on Computer Graphics Theory and Applications*, pages 335-341
DOI: 10.5220/0001358303350341

may use other methods: store in rgbe (Ward, 1991), or store textures with multi-exposure levels (Cohen et al., 2001). Note that these two methods are not implemented in the framework, only floating point an byte textures are supported at the moment. Programmable shading describes a new paradigm of real-time computer-graphics: Instead of a configurable fixed function pipeline, parts of the pipeline are replaced by programmable shaders (Mark et al., 2003). Shaders are extensively used in the framework, as the only way to enable flexible precalculated GI combinations.

The third area is the process of tonemapping: Physically plausible GI methods produce images with an high dynamic range of the output luminance. This high dynamic range has to be compressed to fit the dynamic range of monitors. This process is called tonemapping, refer to (Artusi, 2004) for a good overview. Note that the described framework is orthogonal to the problem of tonemapping, as this framework creates source images to be used with tonemapping. Therefore tonemapping is not in the scope of this paper, however in Section 2.4.2 a speed-up technique applicable to all luminance-based tonemapping operators is described.

1.2 Organization of this Paper

This paper is structured in the following way: In Section 2 the basics and the parts that make up the framework are described. Section 3 shows performance evaluations and screenshots obtained using the framework. Section 4 serves as a conclusion.

2 METHODS

In this section we will at first describe the extension to view independent GI precalculation in more detail. Then an overview of the framework is provided in Section 2.2, followed by a discussion of hardware specific issues in Section 2.3. Finally, Section 2.4 will provide details on implementation of specific parts of the framework.

2.1 Method Details

The basic idea of the extension is simple: At first, the GI precalculation are not created for the scene with all light sources, instead they are calculated for each light source (or group of light sources). After this step multiple preprocessed GI data is now present for the scene, one data set for each light source group. This data can then be combined in Realtime, with arbitrary scaling applied to either GI solution set. This way the intensities of every group of lights can be changed in real time without expensive precalculations.

There are some points to note. For this method to be not only perceptually correct but physically plausible, the light source interactions in the scene have to be linear and thus separable for later combination. Furthermore the combination of the multiple GI data has to be performed in a linear space. This implies following restriction on the format of the GI data: The data must be stored either directly in a linear space, or it must be stored in a way that can be decompressed to a linear space for realtime combination. As actual scenes may have a dynamic range of up to 1:400000 (Drago et al., 2003), the only way to directly store the data in a linear space is by using floating point numbers.

2.2 Framework Overview

Figure 1 displays the components of the framework. The framework takes n light-source descriptions as input, and further splits them up into $m \leq n$ user defined light-groups. An external GI solver is now called for each light-group, this solver then creates a GI-solution set for each light-group. After all GI-solutions are calculated, there are two ways to apply these solutions to the scene in real-time.

The first method is the fully hardware accelerated path: Here a shader is generated based on parameters determined during light-source splitup, as described in Section 2.4.2. This shader is then applied during scene-rendering and combines all GI-solutions. Afterwards tonemapping is performed on the combined result.

The second method uses a software-fallback to combine the solutions, the combined solution can then be used during scene rendering. This approach is described in Section 2.4.3.

2.3 Hardware Considerations

The linearity requirement described in Section 2.1 combined with the high dynamic range of actual scenes, is the reason why this approach only works straightforward on current graphics hardware, and can not directly implemented into older hardware. To explain why this requirement poses a problem for older graphics hardware, let us be more specific on how view independent GI methods can be implemented using current graphics hardware.

GI Implementation Let us assume a GI solution that creates textures containing the lighting information for the scene. These textures, called lightmaps, contain the precalculated GI solution. Following the previously described approach, multiple textures must be created, one set for every light group. Those textures are used as source textures for a realtime frag-

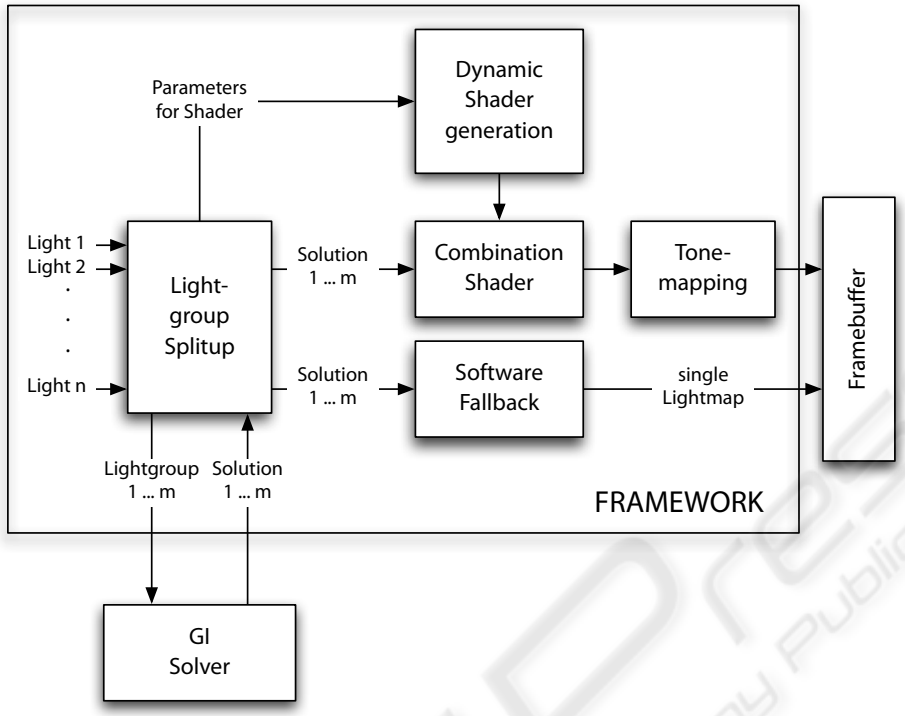

Figure 1: The components of the presented framework.

ment shader. The used fragment shader combines the textures in realtime and applies tonemapping at the end.

As textures are used for storage, the textures must be able to contain the high dynamic range for a correct linear combination later on. This restrains these textures to be in floating point format. Therefore limitations of graphics hardware considering floating point texture formats have to be analyzed. Table 1 shows the capabilities regarding floating point texturing on some generations of graphics hardware, and the corresponding OpenGL Extensions that are supported.

Floating Point Support As can be seen in this table, only NVidia GeforceFX and Ati R300 series and later series are able to handle floating point textures. Further it can be seen that GeforceFX series cards need the floating point textures to be provided via the NV_float_buffer, unlike the other cards that can handle the ATI_texture_float extension. This extension mismatch can only be resolved by providing different render paths for these cards, as described in section 2.4.2.

Compatibility However, we want most graphics cards, and not only the latest series to be able to run the framework. Therefore two mechanisms are integrated into the framework: The first mechanism is a software fallback for combination of the lightmap textures, as described in section 2.4.3. This fallback runs on every hardware capable of texturing. Note that the software combination has to performed only when the light intensities are changed, not on every frame.

The second mechanism is to allow the use of 8 Bit per channel textures for lightmaps. It is not possible to use these textures in a naive way for most scenes, as they only offer a dynamic range of $1:255$. Therefore values have to be packed into those textures in a nonlinear manner during lightmap creation, and later during combination in a fragment shader those values must be unpacked again into a linear space. Arbitrary functions are applicable during packing, as long as they are reversible in an easy way for unpacking in the fragment shader.

The exponential function

$$x_1 = f(x) = 1 - e^{-x}$$

with its inverse

$$x = f_1(x_1) = -ln(1 - x_1), x_1 \in [0, 1[$$

is used in this implementation. With this function a dynamic range of $1 : 1413$ is achievable, as is shown: The highest value $x \neq 1$ representable by 8Bit is $x = 254/255 = 0.9961$, this results in $f_1(254/255) = 5.541$ after unpacking. Therefore a dynamic range of $max/min = 5.541/(1/255) = 1412.955$ is achieved.

Table 1: Floating point capabilities of graphics-hardware. Sources: (nvi, 2005), (ati, 2005).

Vendor	Graphics Processor	Floating point capability	supported via OpenGL Extension
NVidia	Geforce 6 Series	Full Support	ARB_texture_float, ATI_texture_float
NVidia	Geforce FX Series	No Bilinear Filter	NV_float_buffer
NVidia	Earlier Series	No floating point support	-
ATI	R300 Series	No Bilinear Filter	ATI_texture_float
ATI	Earlier Series	No floating point support	-

To sum it up, if a scene does not exceed the dynamic range of $1:1413$ a exponentially packed 8Bit representation is sufficient. Otherwise the software-fallback must be used for correct results, or floating point textures have to be used.

Note that other possibilities to get high dynamic range without floating point textures have been proposed (Ward, 1991) (Cohen et al., 2001). Those methods could also be integrated into the framework, however they are no implemented at this moment.

2.4 Implementation Details

To actually create an efficient implementation of the described framework, three points must be considered. At first, the memory requirements have to be optimized, as described in Section 2.4.1. Secondly, the fragment shader used for recombination has to be flexible and efficient. This is discussed in Section 2.4.2. And lastly, a Software-Fallback must be provided in order to run this framework on older graphics-hardware. Section 2.4.3 describes this step.

2.4.1 Memory Footprint

To actually make the described method feasible, considerations to reduce the memory footprint have to be taken. The memory requirement is a possible problem, as multiple solutions of the GI have to be stored. If every light has its own lightmap the memory footprint may be too large.

Light-Groups In order to reduce the amount of required GI solutions, Lights can be sorted into groups, and every group gets its own lightmap set. Of course, the ability to adjust light-intensity is thereby reduced to the whole light-group, the separate lights in this group can not be controlled anymore. It is up to the user to decide whether lights should be combined to group. It may be best practice do combine semantically equivalent lights (e.g. all ceiling lights) to one group.

Luminance Packing Another method to reduce memory footprint is to calculate one channel luminance maps of three channel RGB-Lightmaps. Four of those luminance maps can then be packed to a single RGBA Lightmap. The unpacking is then performed in realtime by a fragment program. Therefore the fragment program takes each component of a packed RGBA-Lightmap and multiplies it with the accumulated light source-colour of the corresponding light group. The light source-colour is provided to the fragment program via constant registers. It is important to note that effects like colour bleeding can not be reproduced this way, as colour information is restricted to the light source colours only. Again, it is up to the user do decide whether colour-bleeding is important for the specific light group.

Lightmap Precision Reducing the precision of the lightmap textures from 32 Bit floating point to 16 Bit floating point (half) is another way to decrease the memory requirements. The half format is probably sufficient for most scenes. If the dynamic range does not exceed $1:1413$, further reduction to 8Bit precision is possible, as described in Section 2.3.

2.4.2 Fragment Shader

Multiple steps are performed in the combination fragment shader: At first, the value of each lightmap at the current texture coordinates is fetched. For packed lightmaps each component of this value is then multiplied with the corresponding light source colour. Byte textures are unpacked to a linear space. Every value is then multiplied with a scaling factor provided via constant registers. All scaled values are then summed up.

This sum is then used for the tonemapping operator. At the moment the tonemapping operator is directly implemented in the fragment shader. We have chosen *Adaptive Logarithmic Mapping* (Drago et al., 2003) as the tonemapping operator. As the tonemapping problem is orthogonal to the lightmap mixing problem, any tonemapping operator could be used here.

Optimization Note that the chosen tonemapping operator is luminance based: Given an input luminance value l_{in}, an output luminance value l_{out} is calculated. Therefore we need to calculate the luminance value l_{in} of our RGB colour-values c_{rgb} at first. Using the `ITU-R BT.601-4` definition of

luminance, this is done using a simple dot-product $l_{in} = c_{rgb} \circ (0.299, 0.587, 0.114)$. We then calculate l_{out} using the tonemapping operator. There is no need in colour space conversions to get the tonemapped color t_{rgb}, the scalar product $t_{rgb} = c_{rgb} \cdot l_{out}/l_{in}$ is sufficient, as shown below:

We want to show that the luminance of $t_{rgb} = c_{rgb} \cdot l_{out}/l_{in}$ equals the luminance l_{out}. The luminance of t_{rgb} is defined as $l_t = t_{rgb} \circ (0.299, 0.587, 0.114) = t_r \cdot 0.299 + t_g \cdot 0.587 + t_b \cdot 0.114$. Substituting $t_{rgb} = c_{rgb} \cdot l_{out}/l_{in}$ into this formula results in $l_t = c_r \cdot l_{out}/l_{in} \cdot 0.299 + c_g \cdot l_{out}/l_{in} \cdot 0.587 + c_b \cdot l_{out}/l_{in} \cdot 0.114$. This simplifies to $l_t = l_{out}/l_{in} \cdot (c_r \cdot 0.299 + c_g \cdot 0.587 + c_b \cdot 0.114) = \frac{l_{out}}{l_{in}} \cdot l_{in} = l_{out}$.

Dynamic Shader Generation The source lightmaps used in the fragment shader are flexible in two ways: The number of lightmaps, and the type of each lightmap. Each lightmap may either be packed or unpacked, and may be in floating point or 8Bit format. Additionally the OpenGL extension to be used for floating point-support varies depending on the graphics hardware, as described in section 2.3.

Therefore the fragment shader used to combine the textures must adapt to these parameters. In our implementation this is achieved trough a configurable `lua` (Ierusalimschy, 2003) script: The script is configured with the number of lightmaps for each possible type. The type of OpenGL extension used toggles where the textures should be fetched from: On the one hand, `NV_float_buffer` requires textures to be bound to `NV_texture_rectangle`, they must be sampled via `texRECT` in cg, and additionally the texture coordinates must be scaled to absolute pixel positions. On the other hand, `ATI_texture_float` textures must be bound to `TEXTURE_2D` and fetched via `tex2D`.

Using this configurations, the script then generates a fragment shader using appropriate predefined textblock-connections. This fragment shader is then bound during runtime.

Note that recent developments allow the generation of dynamic shaders to be done via interfaces in Cg (Mark et al., 2003). However, at the moment of implementation the graphics-card driver support for this feature was still in an early version. Therefore we restrained from using these interfaces, however in near future it may be feasible to use those interfaces used instead of lua-scripting.

2.4.3 Software fall-back for Combination

To make this framework compatible to older graphics hardware, a fallback method for all hardware capable of texturing is required. This method takes all lightmaps of the GI solutions and combines them on the CPU. All the steps that are done in the shader (including tonemapping), as described previously, are now done on the CPU. A new set of lightmap textures is now created for storage of the combined result. This set is now used for simple textured rendering.

The combination step just has to be performed every time the intensity values of the light-groups change, not for every frame. As pointed out in the results section, an combination step takes less than a minute, and is thus much faster than a complete recalculation of the GI solution, which can require a few hours.

3 RESULTS

The described framework was implemented for a commercial CAD package. To conduct performance tests, a bath-room scene is used. This scene features 4 lights. The test system has the following specifications: Pentium 4 1.7GHz, 1Gb RAM, GeforceFX 5950 Ultra.

Following test-cases where used:

Test-Case 1 Every light is in the same light-group, therefore no separate intensity-adjustment is possible. This effectively bypasses the described framework. Just tonemapping is performed every frame. Note that the tonemapping-operator is explicitly specified in a fragment shader, using a lookup-texture instead would improve performance.

Test-Case 2 Every light has its own light-group. 4 times as much texture memory is therefore needed compared to test-case 1. Combination and tonemapping are performed every frame, therefore separate intensity adjustment is possible in real-time.

Test-Case 3 Every light has its own light-group, however these light-groups are now packed into a single RGBA-Texture using the method described in 2.4.1. The texture memory requirement is now the same as for the first test-case. Combination and tonemapping are performed every frame, therefore separate intensity adjustment is possible in real-time. Figure 2 shows a screen-shot for this test-case. The blocky appearence in this screenshot is caused by the lack of linear-filter support for floating-point textures on the GeforceFX.

Test-Case 4 Every light has its own light-group, however the software-fallback is enabled for combination. One combination step for this scene takes approximately 30 seconds. This step has to be performed everytime a light-group intensity changes. Tonemapping is performed during the combination step, therefore 8Bit textures are sufficient for the

resulting combined texture. This reduces the required texture memory to 0.5. Figure 3 shows a screen-shot for this test-case.

Table 2 summarizes the performance results for those test-cases. As can be seen, the performance difference between un-adjustable lights and adjustable lights is 6Fps at most. We think this slight drop in framerate is worth the gained flexibility. Software fall-back is even faster than un-adjustable lights, as no tonemapping is performed per frame, and faster 8Bit textures can be used. To sum it up, software fall-back is a good compromise between performance and flexibility, while the hardware-accelerated-mode allows instant adjustment of light-intensities on current graphics-hardware without loosing too much performance. Therefore the described framework scales well on different graphics-hardware, and can thus be considered usable for actual application on a wide range of graphics-hardware.

4 CONCLUSION

We have described a framework that allows flexible adjustment of light-source intensities without expensive recalculation of the GI solution. Limitations of graphics-cards regarding this framework have been observed, and implementation details for different generations of graphics-hardware were discussed. The results show that this framework is feasible on a broad range of graphics-cards, as the increased flexibility only inflicts a small hit in performance.

REFERENCES

(2005). *NVIDIA GPU Programming Guide*. NVIDIA Corporation.

(2005). *Radeon 9500/9600/9700/9800 OpenGL Programming and Optimization Guide*. ATI Corporation.

Artusi, A. (2004). *Real Time Tone Mapping*. PhD thesis, TU Vienna University of Technology.

Cohen, J., Tchou, C., Hawkins, T., and Debevec, P. E. (2001). Real-time high dynamic range texture mapping. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 313–320, London, UK. Springer-Verlag.

Drago, F., Myszkowski, K., Annen, T., and Chiba, N. (2003). Adaptive logarithmic mapping for displaying high contrast scenes. In Brunet, P. and Fellner, D. W., editors, *Proc. of EUROGRAPHICS 2003*, volume 22 of *Computer Graphics Forum*, pages 419–426, Granada, Spain. Blackwell.

Ierusalimschy, R. (2003). *Programming in Lua*. Published by Lua.org.

Jensen, H. W. (1996). Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK. Springer-Verlag.

Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J. (2003). Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907.

Ward, G. (1991). Real pixels. In Arvo, J., editor, *Graphics Gems II*, pages 80–83. Academic Press.

Table 2: Performance evaluations.

Number of Lightgroups	Type of Lightgroup-Textures	rel. Texture Memory usage	Fps
1, framework bypassed	1x 16 Bit RGBA Float	1.0	28
4	4x 16 Bit RGBA Float	4.0	25
4	1x Packed 16 Bit RGBA Float	1.0	22
4, software fallback	1x 8 Bit RGBA	0.5	36

Figure 2: Hardware accelerated combination (No bilinear filter as GeforceFX does not support this for float-textures), combination every frame - 25fps.

Figure 3: Software fallback for combination, one combination at beginning - 36fps.