

Combining Rasterization and Ray Tracing Techniques to Approximate Global Illumination in Real-Time

João Cabeleira*
Instituto Superior Técnico



Abstract

This paper introduces a 3D rendering engine that combines rasterization and ray tracing techniques to approximate global illumination in real-time. Namely, direct and indirect diffuse illumination are generated with state of the art illumination techniques processed on the GPU, including a new sky lighting technique that we propose to simulate realistic ambient lighting for outdoor environments. The illumination is then complemented with realistic reflections and refractions generated by an hybrid real-time ray-tracer that performs the ray tracing step on the CPU and the lighting on the GPU.

Keywords: global illumination, rasterization, ray tracing, real-time, sky lighting

1. Introduction

Achieving realistic illumination in real-time is a particularly desirable feature for applications like videogames. However, achieving this goal is difficult because global illumination remains too complex for real-time rendering. For this reason, dynamic illumination in videogames is still almost exclusively generated with local-illumination models processed as rasterization on the GPU.

*Email: jcabeleira@hotmail.com

Recently, several techniques were developed to approximate global illumination effects on the GPU. In general, the idea behind these techniques is the adaptation of classical global illumination algorithms to make them execute as rasterization processes on the GPU.

On the other hand, global illumination effects like sharp reflections are difficult to obtain using rasterization rendering. In these cases, ray tracing is much better suited.

In this paper we introduce a rendering engine called Serenity, that combines state of the art rasterization and ray tracing illumination techniques to approximate global illumination. Additionally, we also present and employ a new sky lighting technique that provides realistic ambient lighting for outdoor environments.

The idea behind the engine is that very advanced and useful illumination techniques already exist for simulating lighting, each one with their own advantages and drawbacks, and which could be combined to overcome each other limitations.

For instance, the current local illumination models and shadowing mapping techniques are very well suited for simulating direct lighting effects and can even match the realism of direct lighting generated by production renderers. On the other hand, techniques like the *light propagation volumes* and *screen space ambient occlusion* can approximate indirect lighting effects very efficiently on the GPU.

Ray tracing can also be used to generate reflections and refractions effects exclusively for the surfaces where they are needed. The problem is that combining ray tracing with the other rasterization based effects is not trivial due to the differences between the two rendering techniques. The solution we propose for this problem consists in using *deferred rendering* as a unification method for the two techniques.

2. Previous Work

Local Illumination In computer graphics, the BRDFs of the most common reflective materials are often approximated using analytic local illumination models like the Phong and Oren-Nayar models. Due to their efficiency and quality, local-illumination models became extensively used to generate dynamic illumination in real-time, even though they only provide a rough approximation to lighting.

Real-time Global Illumination Some of the most advanced and realistic lighting techniques currently used in real-time were achieved by adapting classical global-illumination algorithms to allow them to execute on the GPU. For instance, a recently proposed technique called *light propagation volumes* [1] [2], approximates the *instant radiosity* technique [2] on the GPU using a discretization of the scene lighting to detach the light count from the rendering complexity. Ray tracing also had a particular strong influence on the development of other innovative techniques that are already extensively used in videogames, like screen space ambient occlusion, screen space global illumination, depth-of-field, atmospheric scattering, etc.

Real-time Ray Tracing In recent years, the field of real-time ray tracing has been subject to enormous attention and research. Ingo Wald presented an important contribution to this field in his PhD thesis which laid the foundations for real-time ray tracing and which remains one of the main references in this field [4]. From this point on, several real-time ray tracers appeared, namely the adaptations made by Daniel Pohl to the Quake series videogames to replace their rasterization based graphics by ray traced ones [5]. And more recently, Jacco Bikker also developed a real-time ray tracing engine that was already used to create several academic videogames [6].

3. Lighting Architecture

The illumination is split by the engine into three main components: direct illumination, indirect diffuse illumination and ray traced illumination; see Fig. 1. The direct illumination component is processed completely on the GPU using the phong illumination model and shadow mapping techniques. Indirect illumination is also processed on the GPU using three different techniques: the proposed sky lighting technique, *light propagation volumes* and *screen space ambient occlusion*. The sky lighting and *light propagation volumes* techniques are described in separate in this document.

On the other hand, ray traced illumination is first processed on the CPU, where ray tracing is performed to find which portions of the scene are visible through a reflection and/or refraction, the result of this process is a G-buffer that is then sent to the GPU for lighting. The lighting of the ray traced effects is performed using the same direct and indirect lighting components used to render the scene. The purpose of this method is to maintain all the lighting processing on the GPU, even for ray traced effects, to keep it fast and ensure visual coherency.

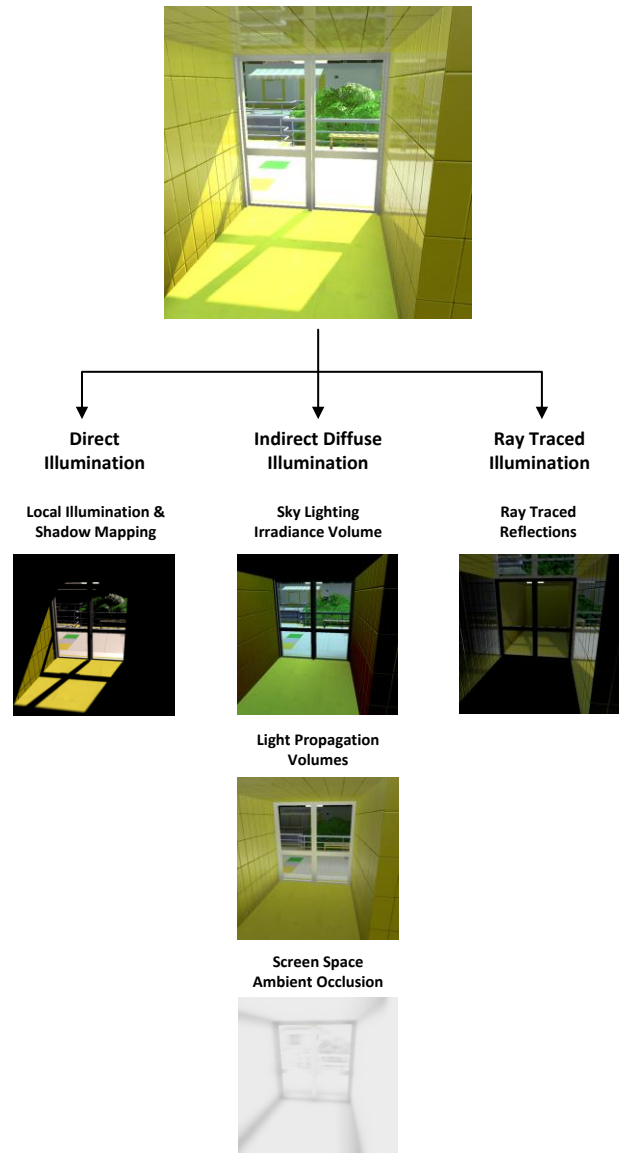


Fig. 1: Lighting architecture

4. High Dynamic Range and Linear Space Lighting

High dynamic range is a crucial feature of the Serenity engine because most of its lighting comes from natural light sources, like the sun and the sky, which can vary from very bright day light conditions to very dark night time. Therefore, it is important to accurately represent this wide range of illumination conditions in order to achieve realistic lighting. Our approach to high dynamic range is

trivial so we will not delve into its details. In essence, all the lighting is processed and stored in high precision buffers and then outputted to the screen using the Reinhard tone mapping operator [7].

Another important requirement for generating physically correct lighting is to ensure that the lighting is processed in linear space. This is important because when simulating lighting in computer graphics there are several subtleties that turn it into a non linear process and which must be considered.

Namely, when an image is displayed in a typical monitor the color is transformed to non linear space, following a function called a *gamma* curve. To counter this effect, the image must be transformed by the inverse of the *gamma* curve before being displayed, known as called *gamma correction*. Moreover, the textures that are used for rendering are also encoded with *gamma correction* and therefore in non linear space which also turns lighting into a non linear process because the textures are used to represent the albedo of surfaces.

Therefore, to generate lighting in linear space it is necessary to perform two steps: at loading time all albedo textures must be converted to linear space and every rendered image must be gamma corrected before being outputted to the screen. Current graphics rendering hardware and APIs already provide the required functionality to do this, by allowing loading textures in sRGB space and converting output color to sRGB space, which makes this process straightforward. The result is depicted in Fig. 2, where it is shown a comparison between non linear and linear lighting.

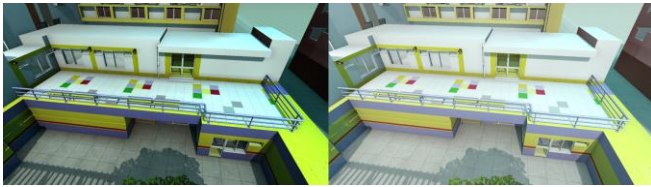


Fig. 2: Non linear vs linear space lighting

5. Atmospheric Scattering

The atmosphere has an important influence on earth's illumination which should be considered in order to obtain realistic lighting when rendering outdoor environments.

In the Serenity engine, the atmosphere affects three points of the rendering:

- the scattered sun light originates the sky
- the color of the direct sun light varies with the amount of light that is scattered away by the atmosphere
- the sky acts as a secondary light source of the scene, known as sky lighting

To simulate these effects, we employ a real-time atmospheric model that generates atmospheric scattering effects for clear sky conditions completely on the GPU [8] [9].

Despite running in real-time, this model is computationally expensive, so to reduce its impact on performance we assume that the sky depends only on the time of day

(atmospheric conditions are ignored because the model is limited to clear sky) which allows to generate the sky once and store it for future use. This way, the atmospheric scattering simulation only has to be performed again to update the sky conditions when the time of day changes significantly.

The storage of this process is performed by mapping the sky dome onto a 2D surface, using paraboloid mapping to preserve the details near the horizon, and rendering the atmospheric scattering simulation to a texture called the *sky map*. Since the sky color varies so smoothly, a 256x256 texture is usually enough to capture and store the sky color accurately; see Fig. 3. Once generated, the *sky map* texture can then be applied to the sky dome to provide color to the sky or used as a data source for the sky lighting effect.

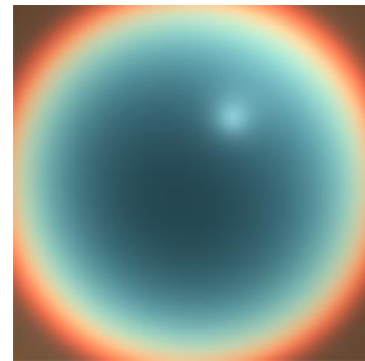


Fig. 3: Example of a sky map

6. Direct Lighting

Direct lighting is simulated by the Serenity Engine using the rendering techniques that are currently used by videogames to generate dynamic lighting. Namely, this lighting is generated with the Phong illumination model combined with shadow mapping.

The main difference is that we use the *percentage-closer soft shadows* (PCSS) [7] which is a shadow mapping technique that provides more realism than other techniques since it generates soft shadows with variable penumbra width. To demonstrate the contribution of this technique, Fig. 4 shows a comparison between different shadow mapping techniques, starting by sharp shadows on the left, fixed penumbra width shadows on the middle, and variable penumbra width shadows achieved through the PCSS technique on the right.



Fig. 4: Comparison between shadow mapping techniques

Special attention was given to simulate the sun accurately because it is the main light source in the real world. The most important aspect of our simulation is that the color of sun light is calculated using the engine's atmospheric

scattering model to simulate the amount of scattering and absorption that the light suffers before it reaches the earth. This allows for the color to vary in a very realistic way when simulating changes in the time of day.

Additionally, shadows are also generated for this light source by combining the *cascaded shadow mapping* technique [8] with the *percentage-closer soft shadows* that were previously mentioned.

Fig. 5 shows the result of direct lighting; notice that no constant ambient term was used due to the fact that indirect lighting will be added afterwards by the sky lighting and *light propagation volumes* techniques.

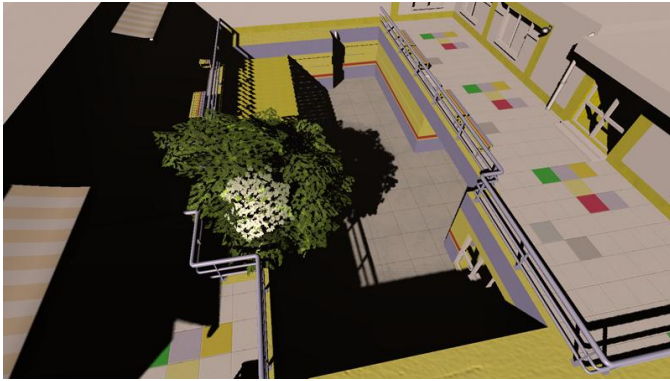


Fig. 5: Direct lighting

7. Sky Lighting

When rendering outdoor scenes or indoor scenes with access to outdoors (e.g. open doors or windows), it is important to simulate the illumination that comes from the sky. This kind of illumination is particularly noticeable in shadowed areas, since the light that reaches them comes almost exclusively from the sky.

In real-time rendering, this illumination is often faked by giving a blue tone (or any other color depending on the sky conditions) to the constant ambient lighting term, which is both physically and visually inaccurate.

To simulate this lighting in a more accurate way, we propose a new technique that runs completely on the GPU, called *sky lighting irradiance volume*. This technique generates an irradiance volume of the sky lighting that reaches the scene while taking into account the blocking of light caused by large objects. The volume is then used for rendering the scene, by sampling it per pixel to obtain interpolated irradiance of the scene.

The idea behind the use of an irradiance volume [9] [10] to represent sky lighting is the fact that sky lighting is in general a smooth effect, which means that it can be processed at a reduced resolution and interpolated as needed.

Hence, the irradiance volume represents a set of evenly distributed points on the scene where the irradiance is calculated. We follow the ideas of the *light propagation volumes* and represent the irradiance volume with 3D textures to allow the volume to be easily generated and sampled on the GPU.

The irradiance at each point is stored as 3-band spherical harmonics, to represent colored irradiance 3 spherical harmonics are used for each point, which amounts to 27

spherical harmonics coefficients for. The storage of these coefficients is distributed by six RGBA textures and one RGB texture, which amounts to the necessary 27 texture color components per point needed for the storage.

Since sky lighting is a smooth effect, the details of the occluders that block sky light are not very relevant, hence their shapes can be represented by bounding shapes.

To make the ray intersection tests perform efficiently on the GPU, we support a single and simple bounding shape called *occlusion quad*. The *occlusion quad* shape was designed with the purpose of providing an occluder representation versatile enough for most scenes that would also be efficient to test for intersections on the GPU.

Fig. 6 shows that the *occlusion quad* is simply a plane defined by a center position, two vectors that point toward its up and right sides, and two scalars that represent the length of each vector. This simple parameterization allows the use of a simple intersection test routine that requires few calculations and conditionals which is important to obtain good performance on the GPU.

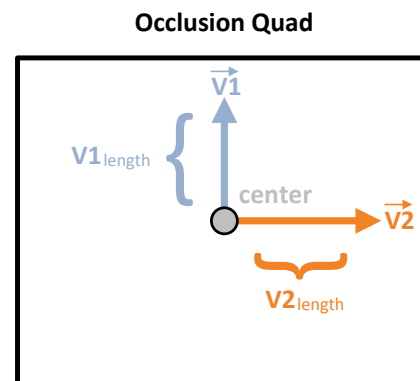


Fig. 6: Occlusion quad layout

Although being limited to planes may seem too restrictive, in practice they are quite versatile in representing most of the large occluders seen on videogames since these tend to be buildings or any other kind of fairly geometric objects.

Fig. 7 shows the placement of *occlusion quads* in the reference scene. On the left it is shown an editor view of the scene and on the right it is showed the corresponding *occlusion quads*. Their main limitation is the fact that it is difficult to make them fit irregular geometry like terrain.

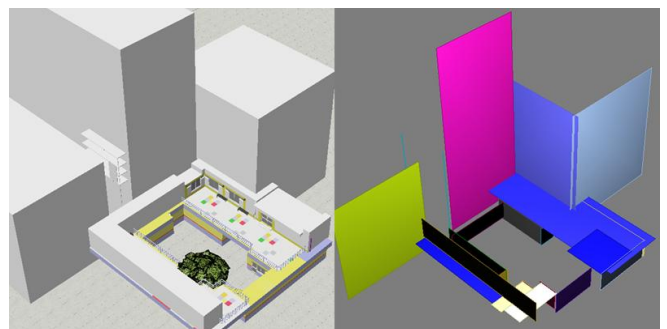


Fig. 7: Editing occlusion quads

The process of generating the *irradiance volume* is done layer by layer, where each layer is generated separately and where each processed pixel represents an irradiance point. At each point, the irradiance is calculated by

sampling the sky color from the *sky map* using a quasi Monte Carlo method and projecting each sample to the *spherical harmonics* basis. For each sample, this process works as follows:

- A ray is created that starts from the position of the irradiance point in world space coordinates and points in the sample direction.
- The ray is then checked for intersection against the *occlusion quads*. If no intersection is found then the *sky map* texture is sampled to obtain the sky color/light that comes from that direction and this color is projected onto the *spherical harmonic* irradiance of the point.
- To finish, the *spherical harmonic* coefficients are outputted for storage on the textures that compose the *irradiance volume*.

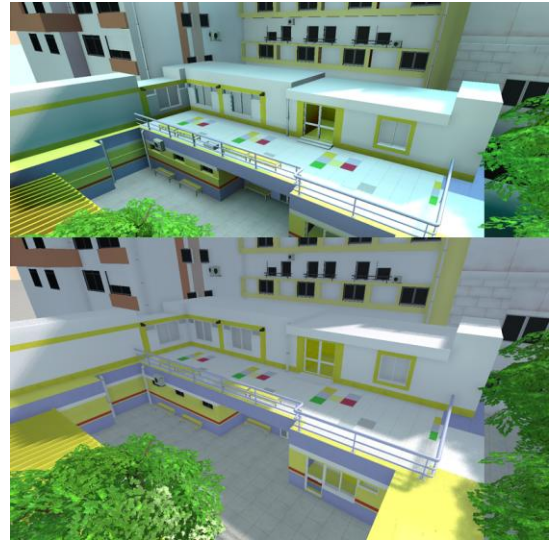
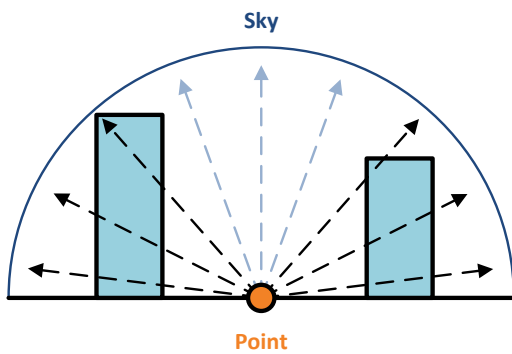


Fig. 8: Comparison between sky lighting (top) and constant ambient term (bottom)



The *sky map* has an important role in this process because it avoids performing repeated expensive evaluations of the atmospheric scattering model.

Nevertheless, sky lighting remains too expensive to compute in real-time. Therefore, we distribute the processing across several frames which is straightforward to do since the processing of the volume is already performed on a per layer basis. Hence, all that must be done is to define how many layers should be generated per frame.

In general, generating a single layer per frame provides a good compromise between the overall time needed to update the whole volume and its impact on the duration of the frame. For the $32 \times 16 \times 23$ *irradiance volume* used for the reference scene, which is composed by 16 layers, the whole volume is computed in only 16 frames without a dramatic impact on the frame rate.

Fig. 8 shows the contribution of the proposed sky lighting technique in comparison to the use of a constant ambient term. Notice how the constant ambient term makes the scene look flat while the sky lighting effect preserves the shapes of the objects and provides realistic lighting because each wall receives lighting from different parts of the sky.

The fact that the *sky lighting irradiance volume* is generated dynamically allows for dynamic time of day changes. This is depicted in Fig. 9, which shows the changes of day illumination, starting by a morning sunrise, followed by noon and finished by an afternoon sunset. All these changes were computed almost instantaneously.



Fig. 9: Time of day changes with sky lighting

8. Light Propagation Volumes

One of the most important global illumination effects, that is required to achieve visual realism, is the inter-reflection of diffuse lighting between surfaces. Simulating this kind of lighting is usually too expensive for real-time rendering. However, a recently proposed technique, called *light propagation volumes*, provides a very efficient approximation of this kind of lighting that runs completely on the GPU.

This technique was implemented in the Serenity engine and provided excellent results. As depicted in Fig. 10, the technique can provide very realistic lighting, particularly for interiors. Moreover, the *light propagation volumes* can also provide glossy reflections which is a very useful feature that increased the range of materials that can be rendered by the engine.

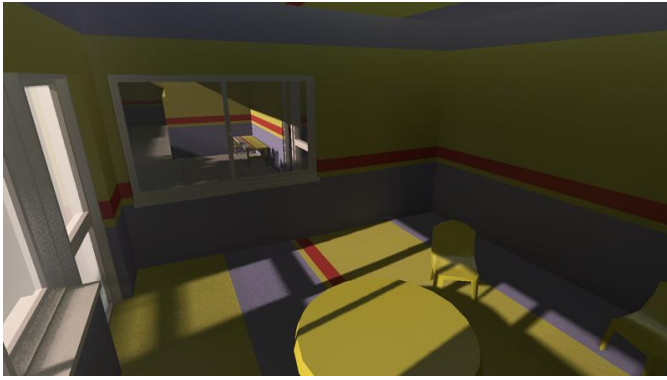


Fig. 10: Contribution of the light propagation volumes

This technique also proven to be a useful complement to our sky lighting technique. Since sky lighting comes exclusively from above, surfaces that are oriented downwards tend to become too dark because they cannot receive sky lighting. However, in the real world these surfaces are rarely dark because they are also lit by light reflected from nearby surfaces. Hence, to obtain realistic lighting it is necessary to inject the reflected sky lighting into the *light propagation volumes*. This is done by using the G-buffer as a reflective shadow map to generate a set of sky lighting VPLs which are then injected. The result of this process is depicted in Fig. 11 where on the left it is shown how outdoor lighting looks when only sky lighting is used, and on the right it is show the result of combining sky lighting with *light propagation volumes*.

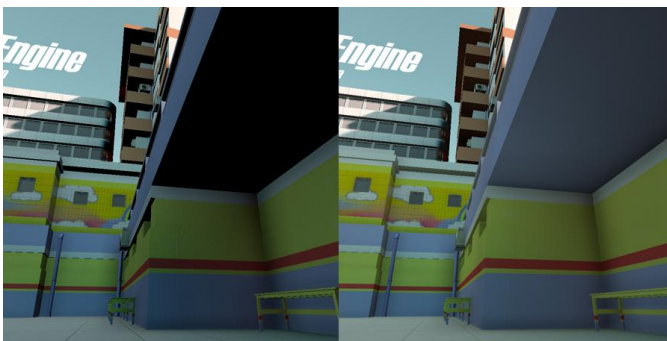


Fig. 11: Using light propagation volumes to complement sky lighting.

9. Ray Traced Illumination

The lighting components presented so far focused on providing only diffuse global illumination. Even though the *light propagation volumes* can also provide glossy reflections, the proposed solution still lacks the support for sharp reflections and refractions that are important to simulate many manmade materials like glass and polished metal, and also natural effects like water.

Even though these effects can be approximated with rasterization based techniques on the GPU, these are not versatile enough to fit every possible situation. Therefore, the ray tracing algorithm was integrated into the Serenity engine to take advantage of its superior versatility for generating accurate reflections and refractions effects for almost every possible situation.

However, ray tracing is very different from rasterization and much more performance expensive, hence the challenge is to make ray traced illumination to run in real-

time and to integrate it seamlessly with the rest of the lighting.

The solution we propose for this problem assumes that it is possible to render the scene from any view with deferred rendering, whether the scene is viewed directly or from a reflection/refraction, as long as there is a G-buffer filled with the necessary attributes of the scene.

The main difference from classic deferred rendering is that the G-buffer is not filled by a rasterization process but by a ray tracing one. Hence, the purpose of the ray tracer is solely to simulate the paths of reflected and refracted view rays and to obtain the scene attributes at their intersections with the scene.

Although GPU ray tracing is becoming increasingly available, we opted to perform ray tracing on the CPU instead. This is due to the fact that the GPU is already extensively used by the engine to process the other lighting components while the CPU spends most of the time idling, waiting for the GPU to finish processing. Hence, we opted to take advantage of this underused processing power to generate the ray traced lighting in parallel while the GPU processes the other lighting components. Moreover, implementing ray tracing for the CPU is much more straightforward than for the GPU and has already been proven to be suitable for videogames while GPU ray tracing has not.

The core of the ray tracer is highly based on Wald's work [4]. The scene geometry is partitioned and stored in a kd-tree that is built using the Surface Area Heuristic to optimize the split positions. The main disadvantage of using kd-trees is that they are difficult to update in real-time so our ray tracing implementation becomes limited to static geometry.

Besides the geometry, the ray tracer also needs access to the textures that define the materials properties of the surfaces which is done by loading the textures to system memory. This requires special care because as stated in the "High Dynamic Range and Linear Space Lighting" section, all textures that represent the albedo of surfaces must be converted from sRGB space to linear space to allow lighting to be processed in linear space. Since there is no automatic conversion available like when loading them to the GPU, the conversion must be performed manually. Additionally, mipmap levels are also generated for each texture to allow sampling from them with mipmap filtering.

The first step in our ray traced lighting technique is to generate the reflected and refracted view rays that will traverse the scene. This is done in the GPU by rendering the reflective/refractive surfaces with a shader that calculates the per-pixel view vectors and applies the corresponding optical distortions to them. The result of this process is stored in a buffer called the *ray casting buffer* that represents the reflection and refraction rays that were generated for each pixel and where each ray is represented by its origin and direction. However, since both reflection and refraction rays share the same origin it is only necessary to store it once.

Fig. 12 shows a rendering of the scene that highlights in red the areas where reflections are required and therefore where the ray casting buffer is filled, and Fig. 13 shows the

contents of the corresponding *ray casting buffer*, where on the left it is shown the origins of the rays, on the middle the reflection vectors and on the right the refraction vectors.

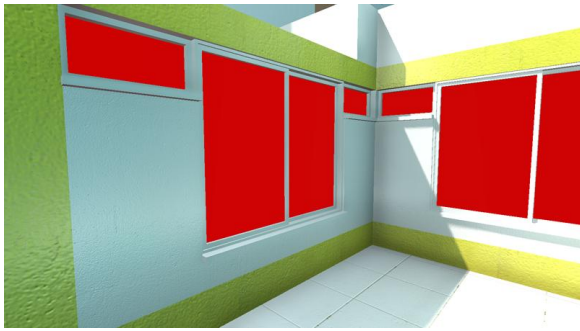


Fig. 12: Areas of the scene where ray tracing is performed

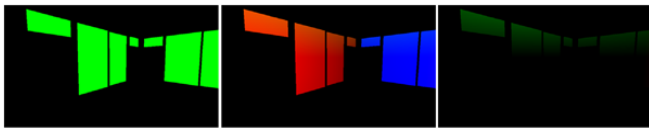


Fig. 13: Contents of ray casting buffer

The *ray casting buffer* is then downloaded to system memory to be accessed by the ray tracer. At this point, the ray tracer extracts the ray data for each pixel, builds the ray and traverses the scene. For each ray that intersects the scene, the attributes of the intersected surface are extracted from the surfaces material textures and stored on G-buffers. The ray tracer uses two distinct G-buffers, one for reflections and another for refractions. The sampling from textures is performed with *mipmapping* to avoid noise in the final image, using ray differentials to calculate the amount of texel compression per pixel.

Fig. 14 depicts the contents the G-buffer used for reflections after being filled by the ray tracer, where on the left it is shown its color attributes and on the right its normal attributes.

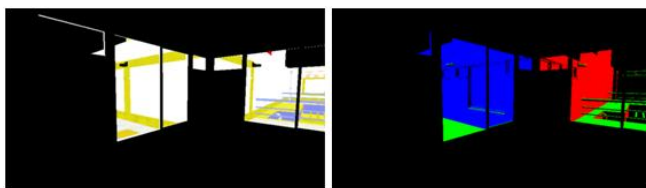


Fig. 14: Contents of ray traced G-buffer

The G-buffers are then uploaded to the GPU so the other lighting components can be used to generate the lighting for the reflected and refracted views of the scene. This is only possible because these components were developed as deferred rendering passes to make them fit any kind of visualization of the scene represented by a G-buffer.

Therefore, applying this illumination to a ray traced view is only a matter of setting the corresponding *ray traced G-buffer* as a data source of the deferred rendering pipeline. Instead of outputting the result of this process to the screen, the result is instead stored in a color buffer, called *ray traced effect buffer*, which allows to perform additional operations and provides a more flexible way of combining the ray traced lighting with the scene. Fig. 15 shows the

contents of a *ray traced effect buffer* used for storing the reflection of the scene.



Fig. 15: Contents of the ray traced effect buffer used for reflections

However, in its original state the deferred rendering pipeline is not completely well suited for applying lighting to ray traced views due to some differences that must be considered between rendering directly and indirectly views.

Namely, the *cascaded shadow maps* technique is not suitable for providing sun light shadows for ray traced views because it focus its coverage on the view frustum. When reflections and refractions come into play, areas of the scene that are outside the view frustum may become visible through those effects but not covered by shadows; see Fig. 16.

To solve this problem, we devised a simplistic solution called *overlapped shadow maps*, which is a generalization of the CSM technique that provides omnidirectional coverage; depicted in Fig. 17. Although this solution is not optimal, since the shadow maps overlap each other wasting some of their coverage potential, our experiments demonstrated that using only 3 shadow maps with 512x512 resolution each provides good visual results with a minimal impact on performance.

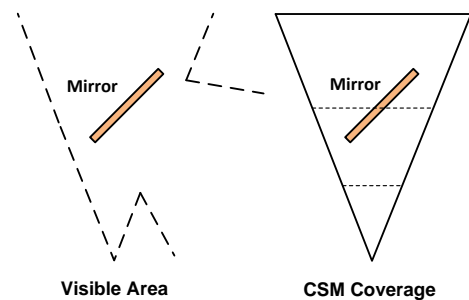
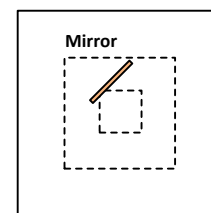


Fig. 16: CSM limitation when dealing with reflections



OSM Coverage

Fig. 17: Overlapped Shadow Maps

Another consideration needed when applying lighting to ray traced views is that *screen space ambient occlusion* (SSAO) may provide unpredictable results and therefore

must be excluded. This happens because SSAO relies on obtaining data about the scene that surrounds a given visible point by sampling neighboring pixels from the G-buffer. However, this is only valid if the view rays are coherent. If the rays are heavily distorted, like when looking at a bumpy reflective surface, then the neighbor pixels may contain information about points of the scene that are unrelated to the point being processed, which may result in visual artifacts.

Combining the ray traced effects with the final image is done by rendering all reflective and refractive surfaces in a forward pass. How the reflections and refractions are blended together to simulate a particular material depends exclusively on the shader used to render the surface. In general, the shader samples the reflections and refractions from the respective *ray traced effect buffers* and combines them using any kind of blending effect, usually a Fresnel reflectance term.

Fig. 18 shows the result of this process, on the left image shows how glass looks before reflections are applied to it and on the right shows the glass featuring reflections. Ray traced refractions were not necessary to simulate transparency, only traditional blending, because common glass does not cause a significant refraction effect. Fig. 19 also shows the use of ray traced reflections to simulate highly reflective materials.



Fig. 18: Using ray traced reflections to simulate glass



Fig. 19: Using ray traced reflections to simulate mirror materials

The main limitation of our ray tracing system is its inability to handle dynamic geometry because it is too expensive to update the kd-tree in real-time. A possible solution to this problem could be the use of a second acceleration structure dedicated exclusively to dynamic geometry, like a BVH tree. This has already been done in the past and proven to be efficient enough [6] [14].

Another important limitation is the fact that the ray tracer cannot generate recursive effects. Therefore, only primary reflections and refractions are supported, which can become noticeable in some situations like the one in Fig. 20, where the reflection on the door shows the window on the right as an opaque object. In practice, this is not a major drawback since higher order reflections and

refractions are not very common in the real world nor very relevant in visual terms.



Fig. 20: Limitations of non recursive ray tracing

10. Scheduling

An important advantage brought by the fact that ray tracing is performed on the CPU is that it can be processed while the GPU is busy rendering the other lighting components, thus amortizing the performance impact of ray tracing.

Obtaining a good parallelism between the CPU and the GPU requires a careful planning of the whole rendering process because the rendering processes of the CPU and the GPU are dependent on each other. In particular, this interdependency is caused by the two synchronization events where the CPU and GPU transfer data to each other: the transfer of the *ray casting buffer* to system memory and the upload of the *ray traced G-buffers* to video memory.

Each of these transfers forces a pipeline flush that stalls the CPU until all pending rendering commands have been processed by the GPU. Hence, achieving a good parallelism requires a careful placement of these transfer events to minimize the stalling effect and so the scheduling of the engine was designed to perform the rendering as follows:

- The first rendering operation performed by the engine is the ray casting process. The idea is to generate and transfer the *ray casting buffer* as soon as possible, before the GPU is commanded to perform any other expensive rendering operations which could make the CPU stall an unnecessary long time.
- Then, the CPU issues the commands that will render both direct and indirect lighting on the GPU. Issuing these commands does not cause any performance impact on the CPU side since they return immediately.
- Once the GPU is busy processing the other lighting components, the CPU starts the ray tracing process that fills the *ray traced G-buffers* with data. At this point, both the CPU and the GPU are running completely in parallel performing the most expensive rendering operations of the lighting solution.
- The engine must then wait for the CPU to complete the ray tracing process by waiting for all ray tracing

threads to finish. Once they do, the *ray traced G-buffers* are transferred to the GPU.

- After the *ray traced G-buffers* have been uploaded, the CPU issues rendering commands to perform their lighting and to combine them with the rest of the scene lighting.

Fig. 21 shows a graphical view of this scheduling as a time line. Notice how a good deal of processing time is spent in parallel processing by the CPU and the GPU.

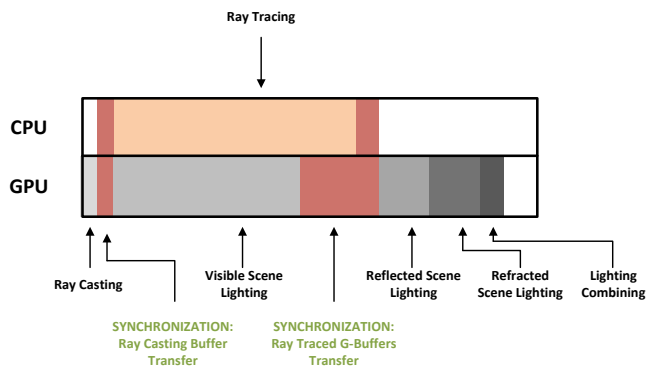


Fig. 21: Scheduling of the rendering

Results and Discussion

In general, the lighting solution we presented can provide realistic illumination effects for both outdoor and indoor environments. In the case of outdoors, this realism is mainly due to our new sky lighting technique; see Fig. 22. For indoor environments, the light propagation volumes presented the most important contribute to the lighting due to their capability for generating both diffuse and glossy reflections; Fig. 23.

The ray traced illumination proven to be very useful for both indoor and outdoor environments, particularly for generating reflections; see Fig. 23. On the other hand, although the engine trivially supports refractions, in practice they are rarely useful.



Fig. 22: Rendering of outdoor environment



Fig. 23: Rendering of indoor environment

Even though ray traced effects are very performance expensive, in practice they do not have a dramatic impact on performance because they are only generated for the areas of the image where they are needed.

Fig. 24 represents a typical case scenario of this fact. To demonstrate the performance of the engine in this situation, we measured the time taken to render this frame and the corresponding timings of processing each lighting step are provided in Table 1. For this measurement we did not account for the time taken to generate the sky lighting volume for the total frame time since this step is excluded from rendering as soon as the volume is completely built. Notice that the total frame time (42.6 ms) is much lower than the sum of the individual timings (65.1 ms - not counting with sky lighting update) because the engine performs the CPU ray tracing process in parallel to the previous lighting steps that are processed on the GPU. To demonstrate the benefits of this parallelism, we modified this experiment to force lighting to execute in serial instead. In this situation, the total frame time increased to 68 milliseconds, which is very similar to the sum of the individual timings, and amounted to an increase of 26 milliseconds in comparison to parallel processing which reduced the frame rate from 23 to only 14 frames per second.



Fig. 24: Typical usage scenario

Lighting Process	Duration (ms)
Auxiliary Steps	
Light Propagation Volumes Update (3 cascades)	5.0
Main Lighting	
Direct Lighting (Phong + CSM + PCSS)	6.5
Sky Lighting	2.4
Light Propagation Volumes Rendering (diffuse + glossy reflections)	6.8
Screen Space Ambient Occlusion	6.0
Ray Traced Reflections	
CPU Ray Tracing	36
Direct Lighting (Phong + OSM + PCSS)	0.8
Sky Lighting	0.6
Light Propagation Volumes Rendering (diffuse + glossy reflections)	1.0
Total Frame Time (without sky lighting update)	42.6 (23 FPS)

Table 1: Timings for rendering the scene

11. Conclusions and Future Work

In this paper, we presented a versatile method for combining state of the art rasterization techniques with ray traced effects on current consumer hardware.

Even though this method may not yet be suitable for current videogames, since it consumes most of the available processing power from the hardware leaving insufficient power for other elements that compose a videogame, the fact that it runs in real-time suggests that it may become suitable in the near future.

However, before that happens several limitations should be overcome. To begin with, the ray tracing system should be improved to support dynamic geometry and to increase the range of materials supported. The *sky lighting irradiance volume* must also be improved to handle arbitrarily sized scenes by following the camera like a *light propagation volume*. Currently the volume is fixed because it takes about one second to generate it, which makes it difficult to account for sudden changes in the camera position.

12. References

- [1] Anton Kaplanyan, "Light Propagation Volumes in CryEngine 3" *Advances in Real-Time Rendering in 3D Graphics and Games Course – SIGGRAPH*, 2009.
- [2] Anton Kaplanyan and Carsten Dachsbacher, "Cascaded Light Propagation Volumes for Real-Time Indirect Illumination" *Proceeding of the 2010 Symposium on Interactive 3D Graphics and Games*, 2010.
- [3] Alexander Keller, "Instant Radiosity" *Proceedings of*

the 24th annual conference on Computer graphics and interactive techniques, pp. 49 - 56, 1997.

- [4] Ingo Wald, "Realtime Ray Tracing and Interactive Global Illumination".
- [5] Daniel Pohl, "Quake Wars Gets Ray Traced" 2009.
- [6] Jacco Bikker, "Real-time Ray Tracing through the Eyes of a Game Developer" *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, 2007.
- [7] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda, "Photographic Tone Reproduction for Digital Images" *ACM Transactions on Graphics-Proceedings of ACM SIGGRAPH*, vol. 21, no. 3, pp. 267 - 276, 2002.
- [8] Ralf Stokholm Nielsen, "Real Time Rendering of Atmospheric Scattering Effects for Flight Simulators" 2003.
- [9] Sean O'Neil, "Accurate Atmospheric Scattering" *GPU Gems 2*, 2005.
- [10] Randima Fernando, "Percentage-Closer Soft Shadows" *International Conference on Computer Graphics and Interactive Techniques*, 2005.
- [11] Rouslan Dimitrov, "Cascaded Shadow Maps".
- [12] Natalya Tatarchuk, "Irradiance Volumes for Games" *Game Developers Conference Europe*, 2005.
- [13] Gene Greger, "The Irradiance Volume" 1996.
- [14] Stephan Reiter, "Real-time Ray Tracing of Dynamic Scenes" 2008.