Error Metrics for Smart Image Refinement

Julian Amann | Matthäus G. Chajdas | Rüdiger Westermann

CG/Vis Group, CS Departement
Technische Universität München
Boltzmannstrasse 3
85748 Garching, Germany

amannj@in.tum.de | chajdas@tum.de | westermann@tum.de

ABSTRACT

Scanline rasterization is still the dominating approach in real-time rendering. For performance reasons, real-time ray tracing is only used in special applications. However, ray tracing computes better shadows, reflections, refractions, depth-of-field and various other visual effects, which are hard to achieve with a scanline rasterizer. A hybrid rendering approach benefits from the high performance of a rasterizer and the quality of a ray tracer. In this work, a GPU-based hybrid rasterization and ray tracing system that supports reflections, depth-of-field and shadows is introduced. The system estimates the quality improvement that a ray tracer could achieve in comparison to a rasterization based approach. Afterwards, regions of the rasterized image with a high estimated quality improvement index are refined by ray tracing.

Keywords

hybrid rendering, reflection error metric, depth-of-field error metric

1 INTRODUCTION

Nowadays, rasterization based graphic pipelines dominate real-time 3D computer graphics, because graphics hardware is highly optimized for this rendering algorithm. Many years of research have been spent on developing massive parallel processing units that are able to process complete pixel quads in a fast way by exploiting frame buffer locality and coherence [KMS10].

Even though rasterization has a lot of advantages, it also has some limitations. It performs well evaluating local illumination models, however there are problems with global effects like reflections. Because rasterization is limited to local illumination models, it is hard to compute physically correct reflections of the environment. For that reason, approximations like environment maps [Gre86] are used, which can result in visually plausible reflections.

Shadows are also a very challenging problem for rasterization. Although there are numerous shadow mapping and shadow volume techniques, they all have some inherent problems that arise from the local view of shading in the rasterization process. For example, most shadow mapping techniques suffer from the well-known biasing problems or have shadow mapping artifacts due to a too small shadow map resolution [ESAW11]. The number of annually appearing publications to shadow topics proves that the generation of shadows is still a challenging subject.

Besides reflections and shadows, it is also hard to simulate a correct thin-lens camera with rasterization. Most depth-of-field techniques which are rasterization based compute the circle of confusion and then just blur the image with the computed radius, which results in an incorrect depth-of-field effect [Pot81].

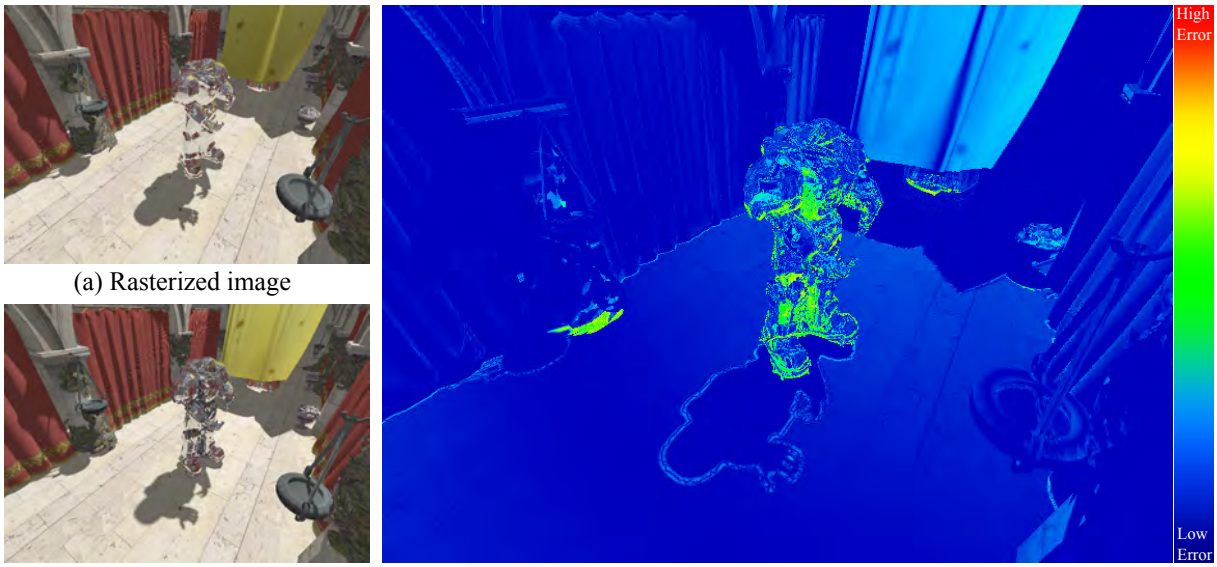
Secondary effects, like shadows or reflections are hard to achieve with a rasterization approach. A ray tracer can natively handle shadows and multiple reflections just by sending additional secondary rays. With a ray tracer it is not hard to simulate these effects. A thin-lens camera model can also be easily implemented in a ray tracer to get a nice depth-of-field effect.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2 MOTIVATION

Because ray tracing is computationally very expensive, rasterization is still used for games today. Another reason is that under some circumstances a rasterizer can produce exactly the same image as a ray tracer could. Figure 1 compares a scene rendered with ray tracing to a scene rendered with rasterization.

Remarkable is the fact that the rasterized image took about 7 ms to render with precomputed environment maps on commodity hardware (NVIDIA GeForce GTX



(a) Rasterized image

(b) Ray traced image

(c) Difference image

Figure 1: A difference image of scene (a) that has been rendered with rasterization (b) and ray tracing (c). The temperature scale on the left shows how to interpret the colors of the difference image. Blue colors mean a low difference and red colors mean a high difference. For example high differences can be seen in the reflecting object in the center of scene.

460) at a resolution of 762 x 538 pixels with no scene graph optimizations. However, the ray traced image with the same image quality and resolution and a highly optimized scene graph took roughly 976 ms to render (the performance measurements were made with the render system described in section 7).

Being given the choice of those techniques, one has to weigh the speed up against the high image quality. In order to profit from the advantages of rasterization and ray tracing, a hybrid approach is preferable. For example, a hybrid technique can just use ray tracing to compute reflections while the rest of the scene can be conservatively rasterized. This helps to get high quality reflections at the cost of only ray tracing the parts of the scene that have reflecting materials.

Ideally, one would estimate beforehand how big the difference between the rasterized and ray traced image is. This difference (see Figure 1c) could be used as a hint to find out where it is most appropriate to send some rays to improve the image quality.

3 CONTRIBUTIONS

This paper describes an error metric for reflections, shadow mapping and depth-of-field, which estimates the expected pixel error between the rasterized and the ray traced scene. The error metric is a heuristic one, which yields an expected but not an absolutely correct error value. During the rasterization of the scene, the error value is calculated and stored in an additional render target. After a first approximation of the current scene by rasterization, the error value of each pixel can be used to find out where it is most appropriate to refine the rasterized image by ray tracing.

This paper also presents a scheduling strategy that determines in which order the parts of the image are getting refined via ray tracing, so the scene converges quickly. This means that parts of the scene with high error get ray traced first and more often than parts of the scene with a low error. This helps to prevent wasting a lot of computation time on parts of the image where no difference or only a slight difference between the ray traced and the rasterized version is observable.

Furthermore, the implementation of a hybrid rasterization and ray tracing framework that is based on Microsoft Direct3D 11, NVIDIA CUDA Toolkit and NVIDIA OptiX 2.5[1] ray tracing engine is described, which demonstrates the feasibility of the described smart image refinement system. The system is called smart, because it refines the image according to the error estimate.

4 RELATED WORK

In the following section related work is briefly presented. Contributions made by other researchers reach from simple hybrid rendering systems to sophisticated perceptually-based techniques.

[1] NVIDIA OptiX is a freely available low level ray tracing engine that runs entirely on the GPU. Currently OpitX is only supported by NVIDIA GPUs. Similar as Direct3D or OpenGL provides an interface to an abstract rasterizer which can be used to implement various rasterization-based algorithms, OptiX provides an interface to an abstract ray tracer.

Perceptually-based techniques try to shortcut the render process by computing a perceptually indistinguishable solution instead of a fully converged one. In [YPG01], a perceptually-based technique is described that calculates a spatio–temporal error tolerance map. The computation of the error map takes a few seconds and is targeted at offline renderers. Each pixel of the error map indicates how much effort should be spent on the respective pixel. The error value is determined by harnessing knowledge about the human visual system and a model which predicts visual attention. For example, the eye is less sensitive to areas with high spatial frequency patterns or movements. Alongside with a prediction on where the viewer directs his or her attention, an estimate is computed that describes how important a pixel will be. This estimate is saved in the error map and is used during the render process to spend more time on important regions of the image. The paper's authors achieved on their test rendering system a $6\times$ to $8\times$ speedup.

[Cab10] uses a simple error metric. The metric consists of only a binary decision if ray tracing or rasterization is to be used. If the rasterizer renders a triangle with a transparent or reflecting material, a flag is set in a ray casting buffer. Afterwards all pixels marked with the flag get ray traced and combined with the rasterized image. They use a CPU-based ray tracer.

In [KBM10], a hybrid approach is shown that combines shadow mapping and ray tracing to render shadows. In a direct render pass and from a small number of shadow maps that are used to approximate an area light source by several point lights, a shadow refinement mask is derived. The mask is used to identify the penumbra region of an area light source. A pixel is classified as inside the penumbra when it cannot be seen from all point lights. Afterwards, the penumbra pixels are dilated by a 5×5 structuring element. The dilated region is then rendered by a CPU-based ray tracer to compute accurate shadows.

5 ERROR METRICS

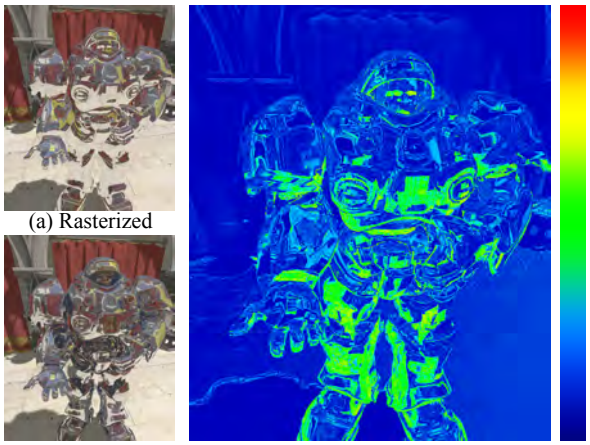
This section describes different error metrics for reflections, depth-of-field and soft shadows. The presented error metrics are used by the smart image refinement system to find out in which regions refinement by ray tracing is most appropriate. An error value is estimated for each pixel and is stored in an additional render target. The error metric is used as a heuristic that indicates how likely the calculated pixel is wrong in comparison to a pure ray traced version of the scene.

A high error value indicates that the approximation by the rasterizer contains a high error, whereas a small error value indicates low errors in the approximation by the rasterizer. The error value is just based on heuristics, which means that in certain circumstances, a high error value refers to a pixel that has only a small real or no approximation error at all compared to the ray traced scene. Conservative error metrics were chosen, so no pixels get estimated as correctly approximated, even if they are not correct.

Each error metric is normalized, which means it generates an error value in the range of $[0,1]$.

Reflections

Reflections can be approximated by environment maps in a rasterization based environment. Figure 2 compares reflections rendered by rasterization to reflections rendered by a recursive ray tracer.



(a) Rasterized

(b) Ray traced

(c) Difference

Figure 2: The rasterized image (a) approximates reflection with an environment of the scene. Figure (b) shows the same scene rendered with a recursive ray tracer. The difference image (c) visualizes the difference between Figure a and b. A red value indicates a high difference and a blue value a small difference.

As can be seen from the difference image, the approximated rasterized image is far from perfect. It contains several regions with wrong reflections.

The simplest to think of error heuristic is one that just makes a binary decision, depending on the criterion if a fragment is part of a reflection object or not [Cab10]. Assuming it is part of the reflecting material, the error metric returns 1, in all other cases it returns 0:

$$E_{reflection_1} = \begin{cases} 1 & \text{: reflecting material} \\ 0 & \text{: else} \end{cases}$$

The previous classification is a very simple one. A more sophisticated error metric can be derived, if we try to understand why the approximation of an environment is wrong. Figure 3 shows two major problems of environment mapping.

Put the case that we want to rasterize a reflecting sphere with the help of environment mapping. For a point P

on the sphere (see Figure 3) we need to look up the current reflection from the environment map. For the look up in the environment texture, we first need to compute the reflection vector. The look up vector is then transformed to texture coordinates which are afterwards used to access the reflection map. The problem with this approach is that the environment map has usually been rendered from the center of the reflecting object. This means, we get the color that is seen from the center of the environment map towards the direction of the reflected vector. Instead of this color, the correct color would be the color that can be seen from the point P towards the direction of the reflected vector. Figure 3 illustrates this. From point P, the reflection direction points toward the colored sphere (r_2). So we expect to see a colored sphere in the reflection of P. But in fact, the environment map technique uses the center of the environment map to look up the color of the reflected object. Looking from the center of the environment map into the direction of the reflection vector, a yellow cube can be seen instead of a colored sphere.

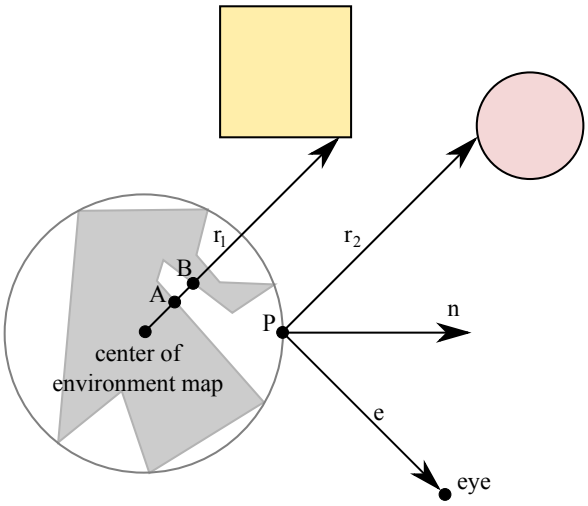


Figure 3: Environment mapping has two major problems: First of all each reflection is computed as if the reflecting point (P) would lie in the center of the environment map. Also it cannot handle self-reflections, which leads to incorrect shading results at point A.

The environment map technique would result in a correct shading, if the shaded point is located directly in the environment map center. For points that only have a very small distance to the environment map center, this approximation works as well. But for points with a further distance to the environment map center, the approximation by an environment map gets worse.

From this observation, a simple heuristic can be derived: The further a point is away from the environment map center, the more likely an environment map technique results in an incorrect approximation. This means that the distance between a point (p) of a reflecting object and the environment map center (c) needs to incorporate in the approximating environment map error metric:

$$E_{reflection_2} = \begin{cases} \frac{distance(p,c)}{maxDistance} & \text{: reflecting material} \\ 0 & \text{: else} \end{cases}$$

Another error metric can be deduced from the incident vector and reflection vector, as Figure 4 illustrates. Assuming that there is a scene with a reflecting sphere where the center of the environment map has been placed at the center of the sphere, this would mean that the environment map has been rendered from the center of the sphere. Looking at the reflecting sphere in a way that the incident ray (the ray from the eye point to a point in the scene) is hitting directly the center of the sphere, as this is the case for the incident vector I_1, the look up in the environment map will yield the correct reflection color.

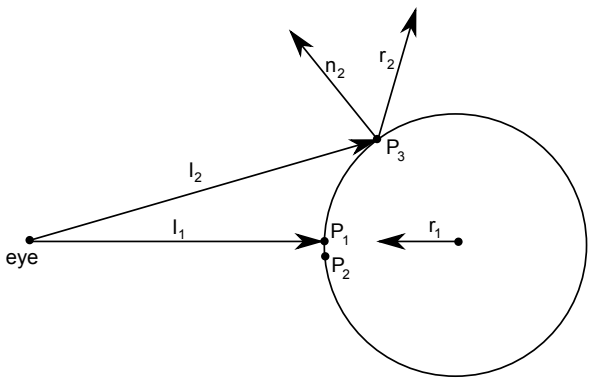


Figure 4: Incident and corresponding reflected vectors

The returned reflection color is correct because the given incident vector I_1 leads to the reflection vector r_1, which means we want to know what can be seen from the intersection point P_1 into the direction r_1. But this is exactly the same question as what can be seen from the center of the environment map into the direction of r_1. If we look from the eye point into the direction of a point that is near to P_1 like point P_2, the incident vector narrowly misses a hit with the center of the environment map, but the looked up direction in the corresponding environment map is approximately not too far from being correct.

It seems that for small angles between the incident and the reflection vector, the approximated reflection vector is almost correct, but for bigger angles like the angle between incident vector I_2 and r_2 it gets worse. From this property, the following error heuristic can be derived:

$$E_{reflection_3} = \begin{cases} \langle -i, r \rangle & \text{: reflecting material} \\ 0 & \text{: else} \end{cases}$$

It is assumed the reflection vector r and the incident vector (vector from eye point to shaded point) i in the

above equation are normalized. The angle between the vector r and $-i$ is always in the range $[0°, 90°]$. Since the angle can be in the range $[0°, 180°)$ the dot product fails for greater than 90° angles. To circumvent this problem instead of considering the reflected vector the normal can be considered which leads to the following equation:

$$E_{reflection_4} = \begin{cases} \langle -i, n \rangle & \text{: reflecting material} \\ 0 & \text{: else} \end{cases}$$

This works because the angle between the incident and reflected vector is directly proportional to angle between the reflected and the incident vector. The angle between the negative incident vector and the normal can never exceed 90°.

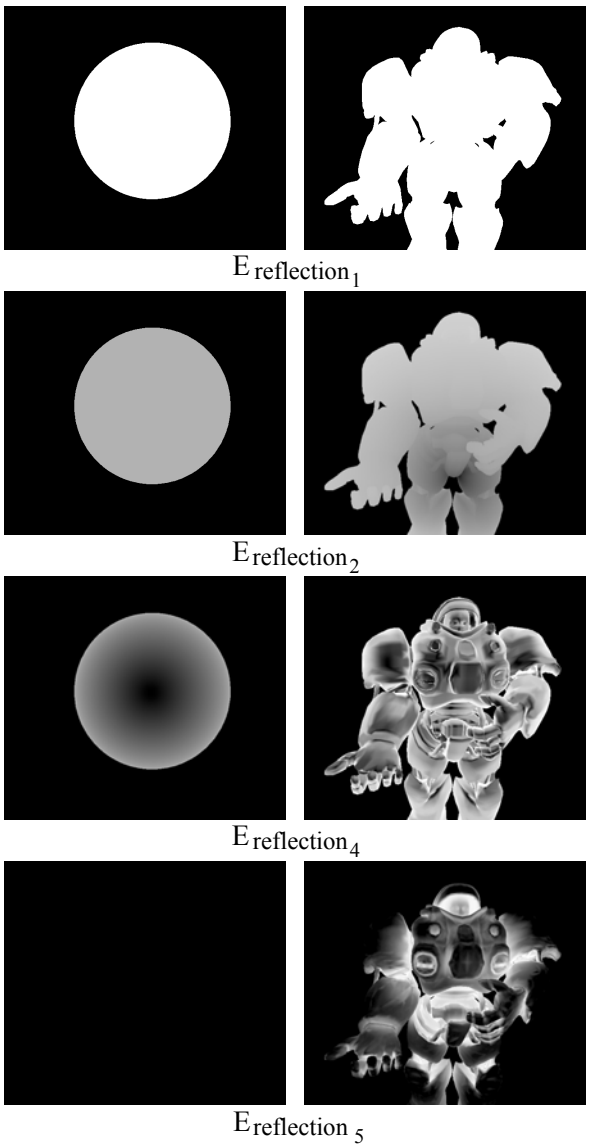


Figure 5: Displays the different reflection error metrics applied to a scene with sphere (left) and a scene with a more complex shape (right)

Another not yet considered problem of the error metric that is also related with environment maps are self-reflections. Concave objects are displayed incorrectly by an environment map technique. Figure 3 shows the reason for this. Assuming we want to compute the reflection of the point A in Figure 3 given the reflection vector r_1. In a look up in the environment, the yellow color from the yellow cube is returned. However in fact the reflection ray intersects the reflecting object itself (a so-called self-reflection) in point B and despite of this, the yellow color from the environment map is nonsense. Self-reflections can probably not be handled by environment maps. We can take care of this in our error metric by using an ambient occlusion map. The ambient occlusion map can be interpreted as a description of the curvature of the reflecting object. This information can be directly used in a heuristic that estimates the possibility of self-reflections:

$$E_{reflection_5} = \begin{cases} k_a(p) & \text{: reflecting material} \\ 0 & \text{: else} \end{cases}$$

$k_a(p)$ refers here to the ambient occlusion term.

Figure 5 shows the different error metrics applied to two sample scenes.

Depth-of-Field

Most rasterization based depth-of-field techniques are image based and use only the depth buffer and color buffer to compute a depth-of-field effect. Thereby, the information about the scene is lost. In a ray tracer, the lens is sampled at different locations. Each sample on the lens has a different view of the current scene. In the rasterizer approach, we have only one view at the scene from the center of the lens. This can lead to missing objects, because in some cases from some points on the lens, objects can be seen that cannot be seen from the center of the lens.

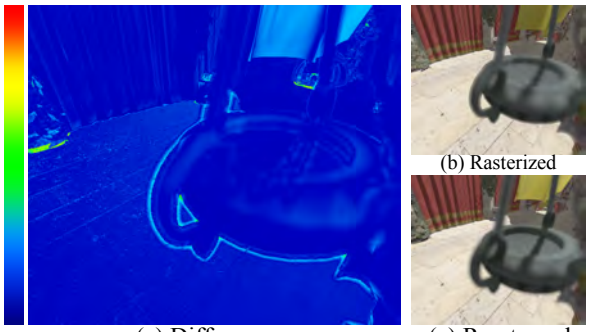
Another problem of most depth-of-field techniques is color leaking. Color leaking can be seen around sharp edges that are in focus in which other blurry objects from the background bleed into [LU08]. The other way around, objects in the foreground can bleed into objects in the background. Figure 6 shows this effect.

As demonstrated in Figure 7, rasterization based depth-of-field have problems in regions with high depth discontinuities. This knowledge can be exploited to construct an error metric for the depth-of-field.

To find depth discontinuities, an edge filter, like the Sobel filter, can be applied. A problem with this approach is that the founded edges have to be dilated by a structuring element, since the artifacts do not only occur at the identified edge pixels, but also in the neighborhood of the edge pixel according the circle of confusion. The

(a) Rasterized (b) Raytraced

Figure 6: A blurry foreground object bleeds into the focused background object.



(b) Rasterized

(a) Difference (c) Ray traced

Figure 7: Difference image of scene (with applied depth of field effect) (a) that has been rendered with rasterization (b) and ray tracing (c). Regions with high depth discontinuities are problematic for rasterization based rendering techniques.

maximal radius of the circle of confusion C_{max} for the dilation can be determined for a point p by the following equation with image distance V_P, focal length F and aperture number n (z_p is the distance between the point p and the image plane):

$$C_{max}(p) = max(C(z_p), C_\infty)$$

$$C_\infty = \lim_{z \to \infty} C(z) = |F - V_p| \frac{1}{n}$$

For simplicity reasons, we use a quad shape structuring element in our implementation to approximate the circle of confusion. Figure 8 shows the error metric for depth-of-field.

The Error metric for depth of field can be expressed as:

$$E_{dof} = \begin{cases} 1 & : \text{Vicinity of a depth discontinuity} \\ 0 & : \text{else} \end{cases}$$

The described error metric is not absolutely conservative, which means that errors can also occur in regions

(a) Rasterized (b) Dilated

Figure 8: After the depth discontinues have been marked, they need to be dilated according to the circle of confusion.

that were not classified with an error value of 0. However, it can give a smart image refinement system a good hint where to start the refinement process.

Shadows

In [GBP06], an algorithm has been described that can be modified to estimate where the penumbra of a given area source light will be projected onto the scene. In Figure 9, the estimated penumbra region is shown in green color.

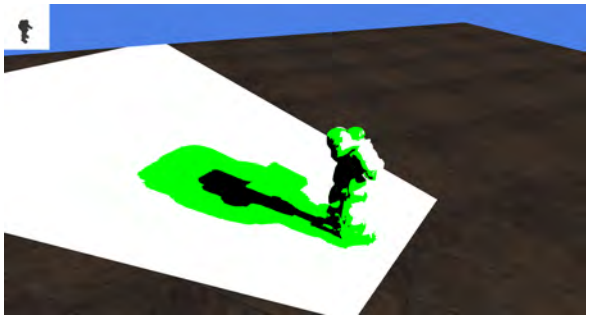


Figure 9: The estimated penumbra region is shown in green color.

The algorithm described in [GBP06] makes a conservative estimation of the penumbra region, and is therefore perfectly appropriate to be used as an error metric. The difference between the shadow generated by the rasterization system and the ray tracer is only notable in the penumbra region. In umbra regions and in regions where the area light source is not occluded by other objects (so that the scene is fully lit by the light source) no difference between the rasterizer and ray tracer is noticeable (see Figure 1 - only the penumbra region needs to be refined).

In [Mic07], an improved algorithm is presented which can estimate a tighter, conservative penumbra estimation than the algorithm described by [GBP06]. Even though it requires more memory, the tighter estimation reduces the amount of pixels that have to be refined, resulting in an overall improved performance.

The corresponding error metric for soft shadow is therefore quite simple:

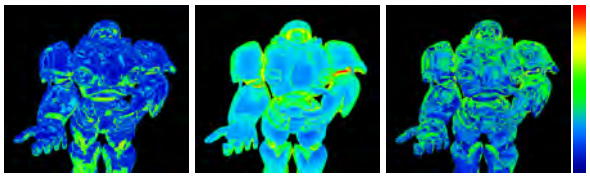
$$E_{shadow} = \begin{cases} 1 & \text{: Pixel resides in penumbra region} \\ 0 & \text{: else} \end{cases}$$

Combination of Error Metrics

The different error metrics can be combined in multiple ways. A naive idea is to calculate an averaged sum:

$$E_{combined_1} = \frac{\sum_{i=1}^{n} E_i(p)}{n}$$

Figure 10 shows the quality of the average sum metric.



(a) (b) (c)

Figure 10: Quality of averaged sum metric. (a) shows the real error, (b) the estimated error and (c) the difference image.

For a better estimation, a more complex combination is required. A simple, yet effective approach is to use a linear combination of the different error metrics (i.e. $E_{reflection_i}$, E_{dof}, E_{shadow}) and let the smart image refinement system automatically determine the coefficients λ_i by rendering multiple views of a scene with the goal of minimizing the difference between the estimated and the real error:

$$E_{combined_2} = \sum_{i=1}^{n} \lambda_i E_i(p)$$

The determination of the factors λ_i is done as a pre-process for each different scene. In this pre-process a certain number of screenshots from the scene is taken (with pure rasterization and pure ray tracing). Then a random tuple of λ_i coefficients is chosen and the combined metric $E_{combined_2}$ is then compared with the real error. We repeat this step multiple times and choose the λ_i coefficients which result in the best approximation for all test images.

6 SCHEDULING STRATEGY

This section describes how the error value is used to direct the rendering process of the smart image refinement system.

First the scene is rasterized by the rasterization system. During rasterization, an error value is also computed as described in the previous section about error metrics. After the rasterization pass, the color buffer and the error color buffer are filled. Now post-processing effects are applied to the previously rendered scenery. The post-processing result is written to the post-process color buffer; after this, the post-process error buffer is computed. Then the error buffer and the post-process error buffer get composed in a combined error buffer. For each pixel, an error value is computed and stored in the combined error buffer. After composing the error buffers, the next step is to sort the pixels. The error buffer also stores, besides the error value, the position for each pixel. The position is needed to find out to which pixel a corresponding error value belongs to after reordering them according their error values. After sorting the error pixels, they are gathered in the request buffer. Additionally to the position, a sample count value is also stored in the request buffer for each pixel that determines how many rays should be traced for the corresponding pixel. The sample count is determined by an estimation pass that fills the request buffer. After the request buffer is filled, it is handed over to the ray tracing system. The ray tracing system reads the first entry from the request buffer and samples the corresponding pixel according to the sample count. The ray tracer proceeds this process for a user-defined maximum number of pixels. After the maximum number is reached, the ray tracing process stops and the smart image refinement system continues with blending the current ray traced image with the computed rasterized image. The blending factor of each pixel depends on the total number of samples that were computed for the corresponding pixel by this time. Figure 11 gives an overview of this process. This process is repeated until the whole image is refined.

7 IMPLEMENTATION

For the implementation of the smart image refinement system Direct3D 11, CUDA 4.1 and OptiX 2.5 ([Ste10]) have been used. Direct3D is used for the rasterization part and analogously OptiX is used for the ray tracing part. Thus all rendering is done on the GPU. Optionally the system can perform pure rasterization with Direct3D or pure ray tracing with OptiX. In the case of pure ray tracing Direct3D is needed only to show the generated OptiX output buffer. Pure ray tracing and rasterization is used for comparison purposes like the time measurements in section 2 or in table 1.

The ray tracing subsystem uses a SBVH acceleration structure [SFD09] provided by OptiX to accelerate ray-triangle intersections. The rasterizer subsystem renders without any scene graph optimizations in a brute force manner.

A pixel shader is used to write the error values during rasterization to an additional render target (the error buffer). Some error values can be only determined after post-processing so there is an additional error buffer (post-process error buffer) which stores the error values determined during applying post-processing effects like depth-of-field. The combined error buffer which contains the unsorted error values is shared with CUDA.

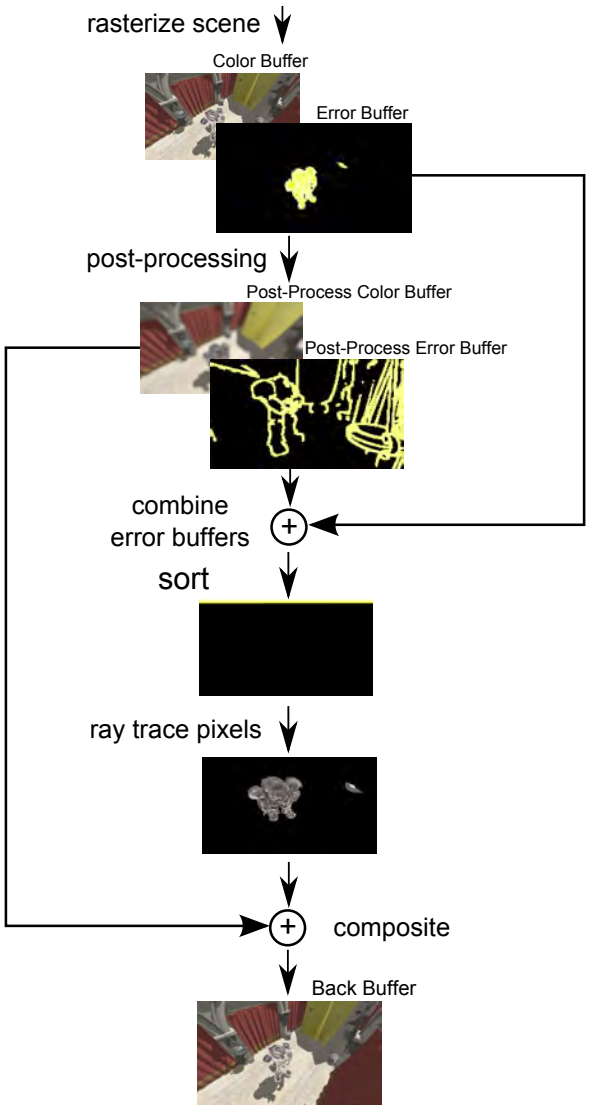


Figure 11: Overview of the smart image refinement system. During rasterization an error buffer is generated. The error buffer is sorted to give the ray tracing subsystem information where refinement makes most sense.

Thrust [HB10], a CUDA library is then used to sort all error values. The error values are encoded in such a way that the radix sort algorithm of Thrust can be used. After sorting the pixel values according their error values a full screen quad is drawn with the size of the sorted error buffer. In this step all pixel are gathered in a request buffer which is implemented as an structured buffer. The request buffer is list with the pixels that need to be refined. Since the error buffer has been sorted the list is also sorted according to the error value.

8 RESULTS

Table 1 shows a performance comparison of pure ray tracing and the prototypically implemented smart image refinement system.

Resolution	PRT in ms	SIR in ms	Error pixel (%)
800×600	440	406	188023 (39)
800×600	312	230	100934 (21)
800×600	203	79	20158 (0.4)
800×600	145	45	4181 (0.01)
1024×768	640	587	290542 (36)
1024×768	305	185	88063 (11)
1024×768	238	84	32889 (4)
1024×768	201	52	32889 (1)
1920×1080	1510	1463	805368 (39)
1920×1080	1107	901	499369 (24)
1920×1080	639	243	145239 (7)
1920×1080	484	113	44140 (2)

Table 1: Performance comparison of pure ray tracing (PRT) and smart image refinement (SIR). In the SIR implementation, one primary ray is traced for each error pixel. Also a shadow and reflection ray is cast per intersection. This is done recursively for reflections three times.

As can be seen from Table 1 smart image refinement is faster than pure ray tracing and at the same time it has the same image quality, provided that conservative error metrics are used. All measurements in this section were made with a NVIDIA GeForce GTX 560 Ti. As a test scene, the extended Atrium Sponza Palace scene that was originally created by Marko Dabrovic and extended by Frank Meinl has been chosen.

The sorting only has to be performed when the scene or camera orientation/position changes. In the implementation of the smart image refinement system a user-defined number of rays are always traced. For instance, the tracing of 8192 rays and the composition of the ray traced and rasterized image takes about 30 ms, depending on the current scene on camera view. This makes it possible to show the user first results after a short render time. Something that has be taken into consideration as well is the fact that in a pure ray traced based approach, the same number of samples is computed for each pixel, no matter if refinement makes sense for the corresponding pixel.

The performance of the smart image refinement system drops with higher error pixel rates. The major reason for this is that resources (e.g. error buffer, request buffer) have to be copied between Direct3D 11, CUDA and OptiX because they cannot be directly shared (some API extensions to improve the interoperability between OptiX, CUDA and Direct3D could avoid these copying steps). For example to hand over the error pixels that should be refined by OptiX, a request buffer has to be filled with the sorted data from a CUDA buffer. The CUDA buffer cannot be directly accessed by OptiX. The data has to be copied first.

9 CONCLUSIONS AND FUTURE WORK

In this work, a GPU-based hybrid rasterization and ray tracing system was presented that is able to direct the render processes to regions with high relevance. Regions with a high error are getting refined first and more often than regions with a small error value. This helps to converge fast to a stable image and avoids at the same time the waste of computing time in regions that do not need any refinement.

There is some scope for improvements of the described error metrics.

Besides reflections, shadows and depth-of-field, it would also be interesting to see how other effects like ambient occlusion (AO) or refractions can be integrated into a smart image refinement system. In the case of AO, a screen based ambient occlusion technique can be employed in the rasterizer to compute a fast approximation of an occlusion term.

Another interesting aspect that has not been considered in this work is global illumination. Global illumination could be approximated with light propagation volumes and refined with a more sophisticated ray tracing technique like path tracing.

There are several real-time perceptually based approaches like [CKC03] which try to cut down rendering time by focusing on important parts. These ideas can be combined with our approach.

10 REFERENCES

[Cab10] Cabeleira Joǎo. Combining Rasterization and Ray Tracing Techniques to Approximate Global Illumination in Real-Time. http://www.voltaico.net/files/article.pdf, 2010.

[CKC03] Cater, K., Chalmers, A., and Ward, G. Detail to attention: exploiting visual tasks for selective rendering. Proceedings of the 14th Eurographics workshop on Rendering, Eurographics Association, 270-280, 2003.

[ESAW11] Eisemann, E.; Schwarz, M.; Assarsson, U. & Wimmer, M., Real-Time Shadows A. K. Peters, Ltd., 2011.

[GBP06] Gaël Guennebaud, Loïc Barthe, and Mathias Paulin. Real-time soft shadow mapping by backprojection. In Eurographics Symposium on Rendering (EGSR), Nicosia, Cyprus, 26/06/2006-28/06/2006, pages 227-234. Eurographics, 2006.

[Gre86] Ned Greene. Environment mapping and other applications of world projections. IEEE Comput. Graph. Appl., 6:21-29, November 1986.

[HB10] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, Version 1.3.0. http://www.meganewtons.com/, 2010.

[KBM10] Erik Knauer, Jakob Bärz, and Stefan Müller. A hybrid approach to interactive global illumination and soft shadows. Vis. Comput., 26(6-8):565-574, 2010.

[KMS10] Jan Kautz Kenny Mitchell, Christian Oberholzer and Peter-Pike Sloan. Bridging Ray and Raster Processing on GPUs. High-Performance Graphics 2010 Poster, 2010.

[LU08] Per Lönroth and Mattias Unger. Advanced Real-time Post-Processing using GPGPU techniques, Technical Report, No. 2008-06-11, Linköping University, 2008.

[Mic07] Michael Schwarz and Marc Stamminger. Bitmask soft shadows. Computer Graphics Forum, Vol. 26, No. 3, pages 515-524, 2007.

[Pot81] Potmesil, Michael and Chakravarty, Indranil. A lens and aperture camera model for synthetic image generation. In SIGGRAPH 81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques, pages 297-305, New York, NY, USA, 1981. ACM.

[SFD09] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In Proceedings of the Conference on High Performance Graphics 2009, HPG 09, pages 7-13, New York, NY, USA, 2009. ACM.

[Ste10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison and Martin Stich. OptiX: A General Purpose Ray Tracing Engine. ACM Transactions on Graphics, August 2010.

[YPG01] Hector Yee, Sumanita Pattanaik, and Donald P. Greenberg. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. ACM Trans. Graph., 20:39-65, January 2001.