Dynamic per Object Ray Caching Textures for Real-Time Ray Tracing

Christian F. Ruff, Esteban W. G. Clua, and Leandro A. F. Fernandes
Instituto de Computação, Universidade Federal Fluminense (UFF)
CEP 24210-240 Niterói, RJ, Brazil
Email: {cruff, esteban, laffernandes}@ic.uff.br

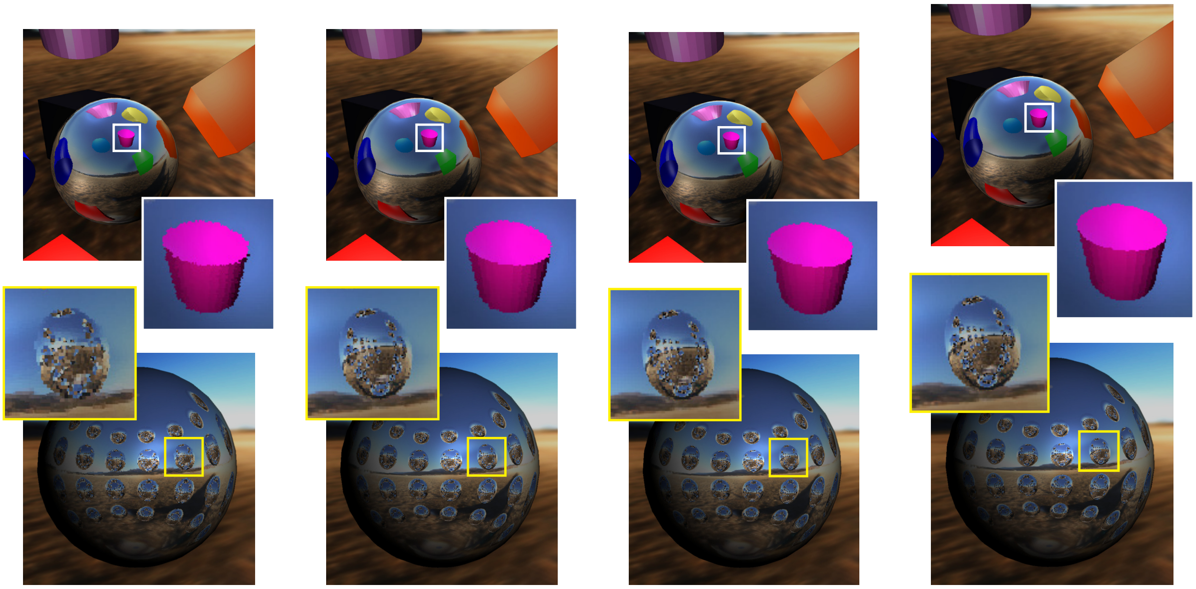

(a) 128 × 128 (b) 256 × 256 (c) 512 × 512 (d) Conventional Ray Tracing

Fig. 1. Images produced by ray tracing two scenes while using the proposed ray-caching textures for real-time rendering (a)-(c), and without our caching mechanism (d). The quality of the resulting images increases as the resolution of the proposed caching structures increases. Notice that caching textures having 512 × 512 texels (c) generated images equivalent to the standard ray tracing (d). As can be seen in Table I for the images on the bottom, our approach has the advantage of producing results like (a)-(c) at 42 fps, while (d) is rendered at 12 fps.

***Abstract*—Ray tracing allows the rendering of scenes with very complex light interactions. It is based on the idea that reflection, refraction and shadows can be modeled by recursively following the path that light takes as it bounces through an environment. However, despite its conceptual simplicity, tracing rays is a computationally intensive task. Also, optimizing memory management to increase efficiency is hard since coherent access in 3D space would not generate coalescent memory patterns. We present a new caching-like strategy suitable for real-time ray tracing which is capable to store data generated in previous frames in such a way that coherent memory access is achieved while data is reused by subsequent frames. By storing light bounce results of previously traced rays in a cubemap attached to each scene object, we show that it is possible to explore the efficient memory sampling mechanism provided by the graphics hardware to increase frame rate. Our approach is suitable for static scenes and may prevent deep interactions of rays with the scene as well as enable synchronous computation of rays in parallelized architectures, and it can be easily integrated to any existing ray tracing solutions.**

***Keywords*-ray tracing; cache memory; cubemap; real time.**

I. Introduction

Computer-generated images having outstanding quality and unsurpassed realism have been created through the use of ray tracing [1]. It is based on the idea of tracing the path of individual light rays from the eye into the scene, and calculating the effect of its interaction with the environment. The recursive task of tracing a ray consists of traversing the 3-dimensional space until the ray hits an object and generates up to three new types of ray leaving the intersection point: a reflected ray continues on in the mirror-reflection direction from a shiny surface; a transmitted ray travels through transparent materials; and a shadow ray is used to test if a surface is visible to a light source. At each ray-surface interaction, an intensity is computed, added to the color of the pixel related to the ray, and some energy is lost. The creation of reflected and transmitted rays stops when the computed intensity becomes less than a certain threshold. While the technique is capable of producing a high degree of visual realism, the large number of rays, intersections and recursive calls brings a large computational cost. For this reason, ray tracing has been best suited for applications that do not require real-time or interactive frame rates, such as still images creation and cinematography visual effects.

Ray tracing can be trivially parallelized in a naïve way. However, the incoherence of the direction of neighboring rays difficult coalescent memory and data arrangement. Also, the independency of the rays' computation naturally leads to different efforts to calculate paths, and usually result in idle threads. Since rays related to neighbor pixels travel through different levels of the data structure used to arrange the objects (e.g., octree), the algorithm is prone to high degree of kernel disparities, making the optimization of the code into GPU architectures not trivial. The existence of efficient ray-caching mechanisms to ensure memory coherence and avoid inter-frame redundant rays computation could really increase the performance of ray tracing algorithms by making different rays shot towards the same object to return synchronously, leaving no threads in idle state, and maximizing the efficiency of the GPU.

We present an efficient technique that manages caching textures of rays. Our approach is capable of storing the reflex information computed by tracing rays in previous frames and reusing it while producing the next frames of the sequence. The main idea is to assign to each reflective object a cubemap to store the reflection color calculated by rays leaving the object in a given direction. Cached information is dynamically completed and updated in render time and used in subsequent frames to avoid tracing redundant reflected and refracted rays. So, before casting new rays to the environment, the technique tries to fetch from the caching textures the reflex information in the ray's direction. If the information is available, the algorithm returns the color and the recursive tracing stops. When the reflex information is not available the ray tracing proceeds normally and the resulting color is used to update the caching textures. Since retrieving information from the caching textures is more efficient than tracing rays into the environment, the proposed technique leads to better frame rates. In the first rendered frames, the conventional tracing of rays will be executed quite more often because the caching textures lookups will not return valid information. However, as the cache memory gets dynamically filled, the efficient memory sampling will gradually replace the computationally intensive work of bounce rays through an environment. Fig. 1 (bottom) shows a scene renderer using the proposed technique (Fig. 1a to Fig. 1c) at 42 fps. Without our caching strategy the same scene (Fig. 1d) was rendered at 12 fps.

The central contribution of this work is the dynamic construction and usage of cubemaps attached to scene objects to store render-time generated data to be used in subsequent frames of ray traced frame sequences. This simple but effective strategy prevents the creation of redundant rays and deep interactions with the environment. Our approach allows synchronous computation of rays cast to the same object, maximizing the efficiency of the GPU. The approach also improves memory coalescence by converting the task of traverse the scene into the simpler problem of fetching coalescent regions of textures as in rasterization-based reflection mapping rendering. We have implemented and tested the solution in order to demonstrate that our approach works well in static scenes with convex objects and is suitable for real-time rendering in ray tracing systems (see Table I).

TABLE I
PERFORMANCE COMPARISON BETWEEN A RAY TRACING SYSTEM RUNNING WITH AND WITHOUT OUR RAY CACHING STRATEGY

Additional Reflective Objects	Hybrid Ray Tracing		Speed Up
	Standard	With Our Approach	
0	68 fps	125 fps	$\sim 84\%$
1	63 fps	120 fps	$\sim 90\%$
2	59 fps	117 fps	$\sim 98\%$
4	49 fps	98 fps	$\sim 100\%$
8	38 fps	85 fps	$\sim 124\%$
16	23 fps	63 fps	$\sim 174\%$
32	12 fps	42 fps	$\sim 250\%$

The remaining of the paper is organized as follows: Section II presents related work. It discusses techniques for optimizing ray tracing, previous attempts to achieve coherent access to 3-dimensional space and memory, and the local storage of render data for computing lightning effects. Section III describes the proposed approach, whose results, advantages, and limitations are discussed in Section IV. Section V concludes the paper with some observations and directions for future exploration.

II. Related Work

Many techniques have been created for making the ray tracing algorithm less computationally intensive. Spatial data structures have been widely used to optimize the interaction between traced rays and scene objects. Glassner [2] proposed the traversal of an octree while computing the intersection of rays with the environment. Foley and Sugerman [3] were the first to demonstrate k-d tree traversal algorithms suitable for the programmable rendering pipeline of GPUs, and to integrate it into a streaming ray tracer. Complete GPU implementations of ray tracers that use an octree as acceleration structure were also developed in the CUDA architecture [4]. Interactive ray tracing of moderate-sized animated scenes was achieved by Wald et al. [5] by traversing frustum-bounded packets of coherent rays through uniform grids. Regular grids and quadtree structures were also investigated by van Reeth et al. [6]. The ability to efficiently rebuild the grid on every frame enabled the treatment of fully dynamic scenes that are typically challenging for k-d tree or octree-based architectures. However, state-of-art general purpose ray tracing engines such as OptiX [7] uses more sophisticated acceleration algorithms. Among them, the Split Bounding Volume Hierarchy (SBVH) algorithm [8] have been used because its data structure is simple to construct, has low memory footprint, allows refitting in animations, and works well with packet tracing techniques. In the same way, the Linear Bounding Volume

Hierarchy (LBVH) algorithm [9] have recently been attracting increasing attention because of its construction speed capacity.

Unfortunately, efficient spatial data structures may not guarantee coherent memory access. In the literature it is possible to find several ray tracers developed with this goal. Pharr et al. [10] have described algorithms that use caching and lazy creation of texture and geometry to manage scene complexity. In order to improve cache performance, they increase locality of reference by dynamically reordering the rendering computation based on the contents of the conventional cache. Aila and Karras [11] proposed a massively parallel hardware architecture based on hierarchical treelet subdivision of the acceleration structure and repeated queuing and postponing rays to reduce cache pressure. As a result, the authors reduced the total memory bandwidth. In a recent work, Yang et al. [12] presented efficient data and task management schemes designed for GPU-based ray tracing. Their approach uses fuzzy spatial analysis, a two-level ray sorting method, and a ray bucket structure to reorganize ray data. By doing so, the threads can be scheduled in order to achieve coherent access to geometry and to reduce memory bandwidth. Our approach uses the spatial data structures provided by NVIDIA OptiX to handle geometry, and no data reordering is necessary to achieve coherent access to rays cached in previous frames. We explore the existing texture mapping functionalities of the GPU to store, sample, and filter cached rays with the same memory-management advantages of using cubemap textures in rasterization-based rendering.

The idea of storing information as textures and use it for computing lightning effects is widely spread. Blinn and Newell [13] presented an extension of the ideas of Catmull [14] in the areas of texture simulation and lightning models, developing the environment mapping. It can be considered even nowadays an efficient image-based lightning technique for approximating the look of a reflective surface by means of a precomputed texture image. This texture is used to store the image of the distant static environment surrounding the rendered object. Miller and Hoffman [15] extended Blinn and Newell's work to cover a wider class of reflectance models. Panoramic images of environments were used as illumination maps that are blurred and transformed to create reflection maps. Miller and Hoffman were the first to use perspective projection onto cube faces. A couple of years later, Greene [16] created the well-known cubemapping.

The storage of render data on the surface of objects was explored by Ward [17], who has described a physically-based rendering system to create the radiance effect on objects. The information of the radiance was stored in structures to be later combined with the objects in the scene. Jensen [18] developed the photon mapping technique that makes possible to efficiently simulate global illumination in complex scenes. It can simulate caustics, diffuse inter-reflections, and participating media like clouds and smoke. The information of these effects are stored in temporary data structures to be used later to generate the final image. In contrast to these techniques, our approach writes traced rays in the proposed cached memory. Also, the data is generated and updated in render time.

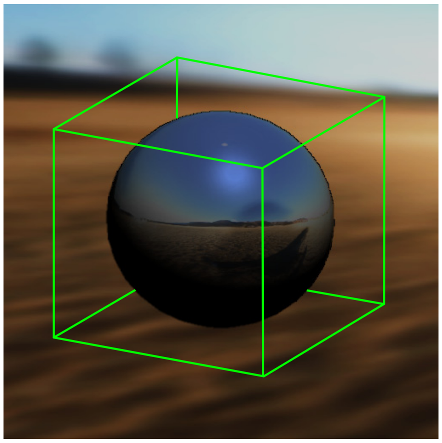


Fig. 2. Axis-aligned cubic box assigned to a reflective sphere in object space during the loading of the scene. The faces of the box coincide to the faces of the caching cube.

III. Per Object Ray Caching

The approach uses 2-dimensional caching textures to store the color of recursive rays calculated by a ray tracer. Each reflective object of the scene generates a particular set of six caching textures, which we call a *caching cube* (Section III-A). For each frame, before tracing a ray leaving the object, the algorithm verifies in the caching textures of the object whether the color information for that ray is available. If the color is not available, the ray tracer calculates the ray's color using the conventional tracing technique, and it updates the caching cube with the new color information (Section III-B). However, when updated color information is available in the caching cube, the algorithm returns the stored color and the ray tracing does not need to be evaluated for that ray (Section III-C). It is easy to realize that the more the camera moves around the scene, the more complete will be the cached information.

The process of consulting a texel at the caching cube is much faster than computing the color information through a ray tracing procedure. Since in the first frames the ray tracing algorithm will be executed quite more often than the simpler texture lookup, it is supposed that when a new reflective object appears in the scene there will be a decrease at the frame rate. However, as ray tracing fills the textures of the object's caching cube, the fetching procedure returns valid results so that the number of ray tracing calls reduces. Stochastically, new reflective objects will appear at the camera frustum in a well distributed manner, so that while new objects will require ray tracing calls, older ones will mostly use cached rays.

A. Caching Cube Setup

For each reflective or refractive object in the scene we define an axis-aligned cubic box. The object and its box are children of the object's transformation in the scene graph hierarchy, so that any transformation applied will affect both. Also, both object and box are centered on the origin of the local coordinate system. Fig. 2 shows a representation of the cubic box around a reflective sphere. The faces of the box are oriented according to the cubemap texture convention, i.e., the faces' normals pointing inside the cube. In our implementation, the cubic box is not created as a piece of geometry. It is only a convention about the geometrical relation between the object and its respective caching cube.

The caching cube setup consists on creating six $RGBA$ 2-dimensional textures, one for each face of the cube. The RGB channels of the textures will store the reflection information of the object they belong. We use the A-channel of the textures as flag to indicate the presence of updated ray information. According to our convention, $A = 0$ means outdated data and $A = 1$ means updated data. All texels are initially marked as outdated. Each face of the cubic box is related to a face of the caching cube. It is easy to see that our technique uses more memory than the common ray tracing. The greater the number of reflective objects in the scene, the greater the memory used to store these textures. The use of the caching cube is a trade-off between speed and memory in favor of speed. However, we remark that these textures remain at the GPU memory, which are abundant in modern architectures. In Section IV we discuss the frame rates and the quality of images produced by conventional ray tracing and by the use of caching cubes having different resolutions.

B. Caching the Reflection Information

Once the caching cube is created, it is ready to store reflex information. The caching procedure is performed along with the ray tracing. For each ray leaving the object, we compute the direction of the reflected (1) or refracted (2) ray using standard formulas [19]:

$$\vec{r} = \vec{i} + 2\,\vec{n}\,\cos\theta_i \tag{1}$$

$$\vec{t} = \eta\,\vec{i} + \left(\eta\,\cos\theta_i - \sqrt{\cos^2\theta_t}\right)\,\vec{n} \tag{2}$$

where $\vec{i}$ is the direction of the unit incident ray pointing to the surface, and $\vec{n}$ is the unit surface normal. θ_i and θ_t are, respectively, the angles of incidence (3) and transmission (4), leading to:

$$\cos\theta_i = -\left(\vec{i}\cdot\vec{n}\right) \tag{3}$$

$$\cos^2\theta_t = 1 - \eta^2\,\left(1 - \cos^2\theta_i\right) \tag{4}$$

In (2), $\eta = \eta_1/\eta_2$, where η_1 and η_2 are the indices of refraction of each medium. Notice that we assume rays leaving the object while updating the caching cube. Therefore, we also assume that the first refraction and internal reflections are already solved for transmitted rays.

We update the caching cube only for rays whose stored information is outdated (i.e., A-channel equals to *zero*). The current state of cached information is retrieved for a given ray $\vec{r}$ by fetching the texture position related to the intersection between the cubic box and a ray r_s traced from the center of the box in the reflection direction $\vec{s}$ ($\vec{s}$ is equal to $\vec{r}$ (1) or $\vec{t}$ (2) depending of the type of ray leaving the object). Based on the intersection point, we calculate which of the six textures of the caching cube has to be used to store the reflection information. Details are described later in this section. In case of outdated data, the standard ray tracing procedure is evaluated for $\vec{r}$ leaving the surface of the object in order to compute its color.

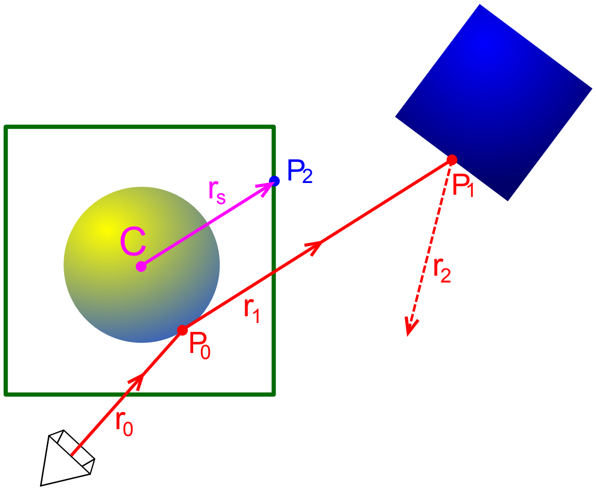


Fig. 3. The path of a primary ray traced from the camera to the scene is defined by its interactions with scene objects. In this 2-dimensional example, the primary ray r_0 intersects a circular object at point P_0, producing the reflected ray r_1 which, in turn, hits a squared object at point P_1. The resulting reflected ray r_2 keeps interacting with the scene until some stop condition be achieved. With our approach, the creation of r_1 is prevented if the caching square (the green box) has updated color information at point P_2, defined by the intersection between one of the faces of the box and the ray r_s is casted from the center C in the direction of r_1.

Fig. 3 shows the process of storing reflection rays in a simplified 2-dimensional view of a complete ray tracing procedure. First, the ray r_0 is traced from the camera to the scene and hits a circular object at point P_0. In turn, the reflected ray r_1 is created and is shot into the environment, hitting a squared object at point P_1. The recursive ray tracing continues until the ray hits a non-reflective object or another stop condition is achieved. In this example, the reflex color to be stored is computed as the intensities accumulated along r_k, for $k > 1$. The location in which the reflex color is stored in the caching square is computed from the intersection of the ray r_s leaving the center C of the square with the same direction than r_1. In Fig. 3, r_s hits the squared box at P_2. In our implementation, all rays traveling from the camera (red arrows in Fig. 3) only intercept renderable objects in the scene. Therefore, cubic boxes do not affect the rays directly. In practice, rays traveling from the center of their respective reflective objects (like the purple arrow in Fig. 3) are not handled by the ray tracer.

The texture that stores the current ray's color (RGB) and state (A) is selected based on the largest magnitude coordinate direction of $\vec{s}$. The texture coordinates to be used in the selected texture are computed as follows:

$$t_u = \frac{1}{2}\left(\frac{t'_u}{|m|} + 1\right),\ \text{and}\quad t_v = \frac{1}{2}\left(\frac{t'_v}{|m|} + 1\right) \tag{5}$$

where m is the coordinate value related to the major axis direction of $\vec{s}$ in object space, and t'_u and t'_v are defined

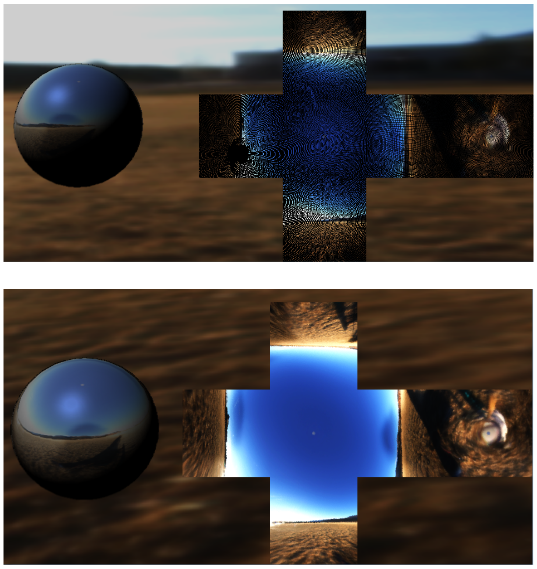


Fig. 4. Color information generated for the caching cube of a reflective sphere after executing the first frame of the application (top) and after moving the camera around the object (bottom).

according to the OpenGL's convention. We assume nearest-neighbor interpolation while sampling rays from the caching textures because the writing operation is also performed on exact texels location.

We have defined the texture selection and the computation of texture coordinates to be consistent with DirectX's and OpenGL's cubemap arrangements. However, it is important to comment that we had to implement those operations as part of our programs because the ray tracing engine adopted in this work (NVIDIA OptiX 2.6) does not support native cubemaps.

Fig. 4 (top) shows the result of caching the ray tracing reflection at the textures of the caching cube. Black texels represent outdated information. After moving the camera and complete near 30 degrees around the object all texels of the caching cube were filled with updated information (bottom). It is important to comment that reflection and shadowing for the current object are computed separately, so that the caching textures will never be affected by shadow rays.

C. Using the Cached Values

The process of retrieving information from the caching cube is performed every time a ray hits the object and produces a reflected or a transmitted ray whose direction ($\vec{s}$) is defined, by (1) or (2), respectively. The 2-dimensional texture of the caching cube and the texture coordinates in which the ray information resides are computed using (5). When the stored information is valid (i.e., A-channel equals to *one*), the recursive ray tracing of the ray leaving the object is prevented and the cached information is combined with the shadow ray in order to compose the final intensity of the incoming ray.

When several primary rays hit objects whose caching cubes are up-to-date, the creation and tracing of all secondary rays are avoided and replaced by the simpler texture fetching procedure. Thus, in GPU-based ray tracing architectures, it is clear that threads related to primary rays will not be idle for too long, maximizing the efficiency of the GPU.

IV. Results and Discussion

We have implemented our technique using C++ with OpenGL, NVIDIA CUDA 4.0, and NVIDIA OptiX 2.6. We have used the Assimp library to load the 3-dimensional scenes, and the DevIL library to load the environment maps. The experiments were performed on a 3.40 GHz Intel Core i7-2600K machine with 8 GB of RAM, and with a NVIDIA GTX680 graphics card having 2048 MB of memory. Microsoft Windows 7 (64-bit) was used as operating system.

The proposed ray caching strategy can be integrated to any existing ray tracing solution. In our experiments we have integrated at an hybrid raster and ray tracing framework developed by Sabino et al. [20], which is totally executed at the GPU, and implemented using the same programming languages adopted by our solution (except for the use of GLSL as shading language, which it is not required by our approach). Sabinos's et al. framework uses the raster deferred shading technique [21], [22] in order to prevent the computation of primary rays hitting diffuse objects in the scene. In a subsequent step, the hybrid solution applies conventional ray tracing to compute and add visual effects such as reflection and transmission for pixels neglected in the previous step. The last stage of this technique consists on the composition of the images created by the raster and by the ray tracing stages.

A. Visual Quality and Performance

The visual quality of reflections and refractions is dependent of the resolution of the textures used as faces of the caching cubes. Fig. 1 presents the quality comparison among the render of two scenes. The first one (Fig. 1, top) is comprised by 10 objects that have diffuse materials and a reflective sphere facing the camera. The second one is comprised by 33 reflective spheres. From the left to the right, the first three pairs of images were produced using the proposed ray caching mechanism, while the last pair was generated by the standard (hybrid) ray tracing. The resolution of the 2-dimensional caching textures used in those examples are, respectively, 128×128, 256×256, and 512×512 pixels. The resolution of the final images is 1024×1024 pixels. Closer looks on the resulting renders are presented in detail. Notice that in Fig. 1a the edges of the objects are clearly bumpy. In Fig. 1b, the texture size was increased, leading to slightly better results. By comparing the detailed view of Figs. 1c and 1d one can see that the use of 512×512 caching textures produced results that are visually equivalent to those produced by ray tracing.

The performance of the proposed approach was evaluated by comparing the frame rates of the standard implementation of the framework developed by Sabino et al. [20] against the same framework enhanced with our ray caching mechanism. The measurements were made using the scenes presented in Fig. 5. All those scenes are comprised by a main reflective sphere facing the camera and by a set of reflective spheres placed behind

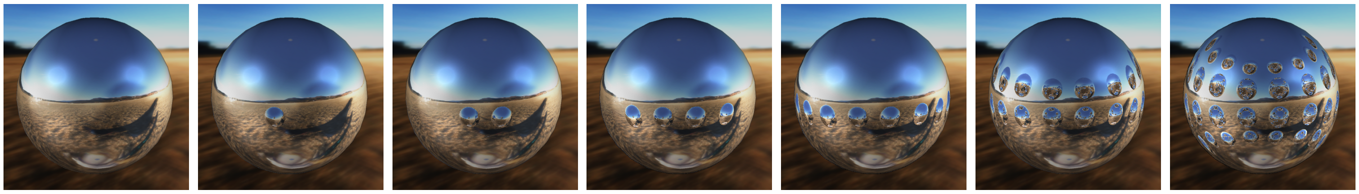


Fig. 5. Scenes used in the performance comparison between our approach implemented as part of an existing ray tracing framework [20] and the standard implementation of the same framework. The complexity of the scenes varies from the left to the right. The first one is comprised by a single reflective sphere facing the camera. The other scenes include, respectively, 1, 2, 4, 8, 16, and 32 additional reflective spheres placed behind the camera. They can be observed from their reflex in the main object. The resulting frames rates are presented in Table I and Fig. 6.

the observer. This setup makes the extra spheres visible only from their reflex in the main sphere. From the left to the right, the sets of additional objects are comprised by, respectively, 0, 1, 2, 4, 8, 16 and 32 spheres. The material of the main sphere implements the proposed ray caching mechanism. The material of the additional objects uses conventional ray tracing. The resolution of the final images is 1024×1024 pixels. The frame rates were taken after the caching textures have been filled. The update rates at different caching resolutions is discussed in Section IV-B. Table I presents the performance of the compared implementations (columns 2 and 3), and the speed up achieved by the use of caching textures having resolutions of 512×512 pixels (column 4). Each scene in Fig. 5 is related to a row of Table I. Notice that the greater the number of reflective objects in the scene, the greater the speed up with our method. This happens because, in contrast to our approach, the computational cost of the ray tracing technique is proportional to the number of bounces performed by the rays, which is dependent of the number of objects. Fig. 6 complements Table I by showing that the relative performance between the use of ray caching structures and the standard implementation has an approximately linear relationship to the number of additional objects is the scene. Table II presents the frame rate comparison of the conventional ray tracing algorithm with and without our technique. The scene used for this test contains a group of spheres with our technique, arranged like a grid in front of the camera. As expected, the speed up was smaller than with the hybrid technique. In the hybrid approach, the primary rays are shot by the raterization-based graphics pipeline. The actual ray tracing is performed only for secondary and high order rays. So, every ray in the hydrid approach will be replaced by our caching technique. On the other hand, in the conventional ray tracing, all the rays are traced in the GPU, and many of them may hit a diffuse object, or not hit an object at all. These rays will not be replaced by our caching technique, giving no speed up at all.

We present only the performance achieved by using caching textures having the same resolution (512×512), because after the caching cube has been filled, the frame rates achieved with different resolutions of the caching textures are the same.

B. Update Rate

The update rates of caching cubes having different resolutions were measured by placing a spherical reflective object in front of the camera and panning the camera around it with nine regular steps of $10/3$ degrees each. The number of outdated texels were recorded every step before the rays being cast into the scene. The results are presented in Fig. 7, regarding caching textures with, respectively, 128×128, 256×256, and 512×512 pixels.

TABLE II
PERFORMANCE COMPARISON BETWEEN A CONVENTIONAL RAY TRACING SYSTEM RUNNING WITH AND WITHOUT OUR RAY CACHING STRATEGY

Additional Reflective Objects	Conventional Ray Tracing		Speed Up
	Standard	With Our Approach	
1	56.7 fps	57.8 fps	$\sim 2\%$
2	52.0 fps	55.6 fps	$\sim 7\%$
4	44.7 fps	51.8 fps	$\sim 15\%$
8	32.8 fps	43.5 fps	$\sim 32\%$
16	17.1 fps	33.4 fps	$\sim 95\%$
32	8.0 fps	21.5 fps	$\sim 168\%$

In all three cases, the caching cubes were initialized with invalid information (100% at the beginning of frame 1). According to Fig. 7a, for 128×128, after the first frame only 3% of the texels had invalid information. After the end of the seventh frame the caching textures achieved almost 100% of valid data. For the texture of 256×256 pixels, the convergence rate is virtually the same than for the previous example. After the first frame, 31% of the texels had invalid information, but after the seventh frame the caching textures are almost filled. For the case depicted in Fig. 7c, as expected, the convergence rate was slightly slower than the previous

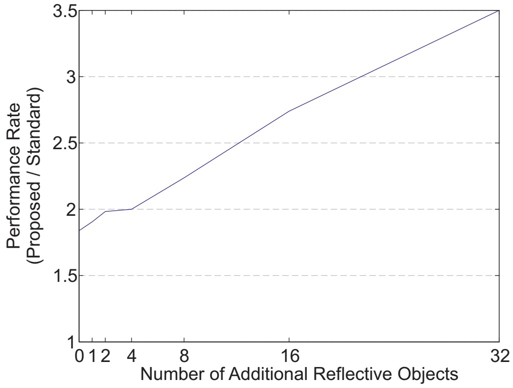


Fig. 6. The relative performance achieved by using the proposed caching scheme and without using ray caching increases linearly with respect to the number of objects in the scene. See Table I.

cases. After the first frame, 73% of the texels had invalid information, but after 8 frames, only 1% of the texels were not updated yet. These tests show that the caching cubes can be quickly filled, even if they have high resolutions. A quick update is a desirable feature that promotes the reuse of stored information.

C. Limitations and Future Work

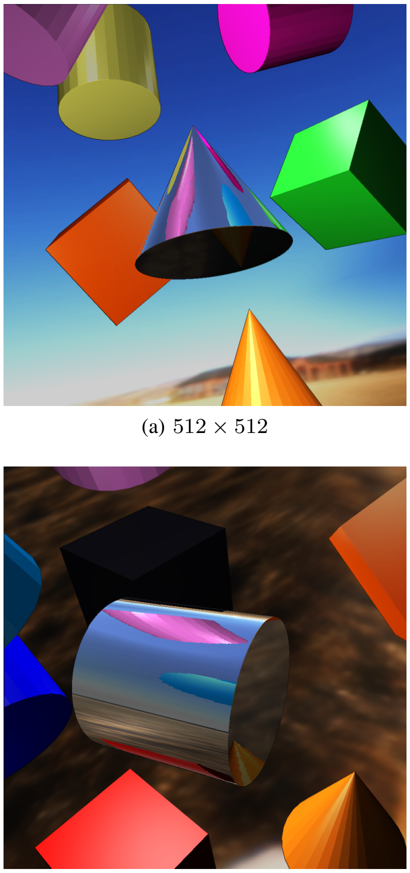
The proposed approach is tailored to static scenes. Since we are storing previous ray tracing information of the scene around a certain object, it is easy to see that any movement on the scene objects would generate inconsistencies on cached data. One possible solution for that problem would be flush the information on the caching cube of objects that can be "seen" by moving elements. That could be managed by a visibility-graph-like structure that connects all the objects that have reflex information of each other. So, when a given object moves, just the caching data of objects that are connected to that particular entity need to be marked as outdated.

Our technique explores cubemap-like texture addressing mechanisms available in current graphics hardware. Cubemaps model skyboxes that always appears infinitely distant from the object. As a result, the visual appearance of smooth surfaces reflecting or refracting close objects may lack parallax effect. A promising area of future work is the extension of relief textures [23] to cubemap environments in order to produce local reflections with seamless transitions between objects. Another possible extension is to use not only rays leaving, but also the rays hitting the object to populate the caching memory. This is possible due to the bidirectional nature of reflectance distribution functions [19]. By doing so, we believe that the update rate of cached rays (Fig.7) may increase significantly. Finally, we also pretend to implement dynamic changes at the resolution, according with the distance and projected size of the reflective object.

The technique is also tailored to be used with convex objects and auto-reflection features. Figs. 1 and 8 show that when the reflective object is convex, the resulting reflex are plausible and comparable to those produced by conventional ray tracing solutions. However, as can be seen in Fig. 9, a concave object will generate concurrent rays that hit different objects of the scene. As a result, some cached rays fetched from the caching cube may be inconsistent at a particular view point. See the impulsive noise at the detailed views of Fig. 9. A possible solution to this issue is the use of relief mapping of non-height-field surface details [24] as caching structure. Another solution would be breaking the object into convex parts.

V. Conclusion

We have presented an efficient cache strategy that improves the locality of storage of rays computed in previous frames and the locality of cached rays' retrieval in subsequent frames rendered by ray tracing algorithms. The approach uses 2-dimensional textures attached to scene objects in a cubic arrangement as cache structure. Outdated data in a given direction of the caching cube is replaced in runtime by the color of rays casted from the object's surface to the environment in that direction. When the cached data is up-to-date, the proposed approach replaces the computationally expensive bouncing of rays leaving the object by the texture sampling mechanism of the GPU. Our local cache system has also enhanced the coherent memory access of the first interaction of rays related to neighbor image pixels and thus has improved performance in that sense. We demonstrate the effectiveness of the proposed technique by implementing it as part of a real-time ray tracing rendering system. Our results show increases ranging from 84% to 250% in the frame rate of scenes comprised by several reflecting objects.



(a) 512×512

(b) 512×512

Fig. 8. Images (a) and (b) show a cone and a cylinder whose material implements the proposed ray caching cube for real-time ray tracing. In both cases, the resolution of the caching textures was set to 512×512 pixels.

Improving performance by gathering related rays together in an object-driven cache memory is a powerful technique. We believe that this idea will lead to further benefits in the development of real-time ray tracing architectures for video games and virtual reality.

Acknowledgment

Christian was sponsored by a CAPES fellowship. Esteban and Leandro received grants from FAPERJ (processes E-26/102.290/2009 and E-26/112.468/2012). The authors would like to thank NVIDIA for the donation GPUs

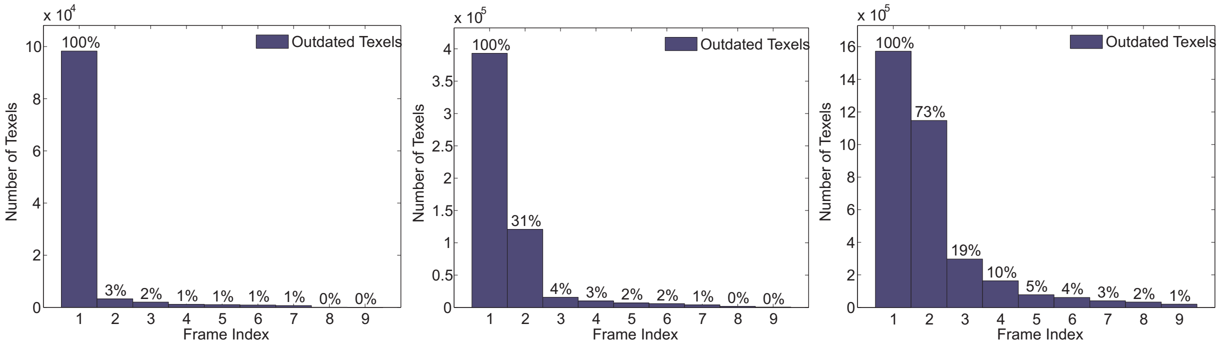
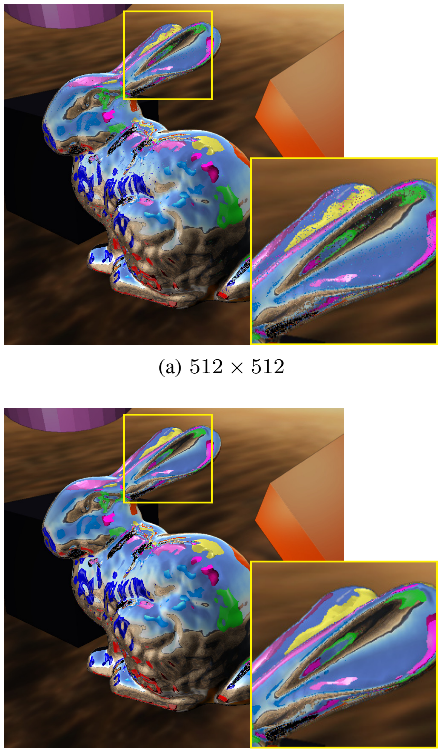


Fig. 7. The amount of outdated texels in the caching textures before each frame of a sequence of nine panning steps of $10/3$ degrees each.



(a) 512×512

(b) Conventional Ray Tracing

Fig. 9. A convex object rendered using 512×512 caching textures (a), and conventional ray tracing (b). For this kind of object, concurrent rays leaving the object may have different targets.

and OptiX team for additional support. We thank T. L. Sabino for additional support with the hybrid raster ray-tracing renderer [20] used in this work.

References

[1] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[2] A. S. Glassner, "Space subdivision for fast ray tracing," *IEEE Comput. Graph. Appl.*, vol. 4, no. 10, pp. 15–24, 1984.

[3] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," in *Proc. of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2005, pp. 15–22.

[4] D. C. Barboza and E. W. G. Clua, "GPU-based data structure for a parallel ray tracing illumination algorithm," in *Proc. of Brazilian Symposium on Games and Digital Entertainment*, 2011, pp. 11–16.

[5] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," in *Proc. of ACM SIGGRAPH*, 2006, pp. 485–493.

[6] F. van Reeth, P. Monsieurs, P. Bekaert, and E. Flerackers, "Ray tracing optimization utilizing projective methods," in *Proc. of Comput. Graphics International*, 1996, pp. 47–53.

[7] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: a general purpose ray tracing engine," in *Proc. of ACM SIGGRAPH*, 2010, p. Article No. 66.

[8] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proc. of High-Performance Graphics*, 2009, pp. 7–13.

[9] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," in *Proc. of Eurographics*, 2009, pp. 375–384.

[10] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, "Rendering complex scenes with memory-coherent ray tracing," in *Proc. of ACM SIGGRAPH*, 1997, pp. 101–108.

[11] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays," in *Proc. of the Conf. on High Performance Graphics*, 2010, pp. 113–122.

[12] X. Yang, D. Xu, and L. Zhao, "Efficient data management for incoherent ray tracing," *Appl. Soft. Comput.*, vol. 13, no. 1, pp. 1–8, 2013.

[13] J. F. Blinn and M. E. Newell, "Texture and reflection in computer generated images," *Commun. ACM*, vol. 19, no. 10, pp. 542–547, 1976.

[14] E. E. Catmull, "A subdivision algorithm for computer display of curved surfaces." Ph.D. dissertation, University of Utah, 1974.

[15] G. S. Miller and C. R. Hoffman, "Illumination and reflection maps: simulated objects in simulated and real environments," in *Proc. of ACM SIGGRAPH*, 1984.

[16] N. Greene, "Environment mapping and other applications of world projections," *IEEE Comput. Graph. Appl.*, vol. 6, no. 11, pp. 21–29, 1986.

[17] G. J. Ward, "The radiance lighting simulation and rendering system," in *Proc. of ACM SIGGRAPH*, 1994, pp. 459–472.

[18] H. W. Jensen, *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001.

[19] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2004.

[20] T. L. Sabino, P. Andrade, E. W. G. Clua, A. Montenegro, and P. Pagliosa, "A hybrid GPU rasterized and ray traced rendering pipeline for real time rendering of per pixel effects," in *Entertainment Computing – ICEC*, ser. LNCS. Springer, 2012, vol. 7522, pp. 292–305.

[21] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: a VLSI system for high performance graphics," in *Proc. of ACM SIGGRAPH*, 1988, pp. 21–30.

[22] T. Saito and T. Takahashi, "Comprehensible rendering of 3-D shapes," in *Proc. of ACM SIGGRAPH*, 1990, pp. 197–206.

[23] F. Policarpo, M. M. Oliveira, and J. L. D. Comba, "Real-time relief mapping on arbitrary polygonal surfaces," in *Proc. of ACM I3D*, 2005, pp. 155–162.

[24] F. Policarpo and M. M. Oliveira, "Relief mapping of non-height-field surface details," in *Proc. of ACM I3D*, 2006, pp. 52–62.