Real-Time Multiply Recursive Reflections and Refractions using Hybrid Rendering

Per Ganestam
Lund University

Michael Doggett
Lund University

2014

Abstract

We present a new method for real-time rendering of multiple recursions of reflections and refractions. The method uses the strengths of real-time ray tracing for objects close to the camera, by storing them in a per frame constructed *bounding volume hierarchy* (BVH). For objects further from the camera, rasterization is used to create G-Buffers which store an image based representation of the scene outside the near objects. Rays that exit the BVH continue tracing in the G-Buffers' perspective space using ray marching, and can even be reflected back into the BVH. Our hybrid renderer is to our knowledge the first method to merge real-time ray tracing techniques with image based rendering to achieve smooth transitions from accurately ray traced foreground objects to image based representations in the background. We are able to achieve more complex reflections and refractions than existing screen space techniques, and offer reflections by off screen objects. Our results demonstrate that our algorithm is capable of rendering multiple bounce reflections and refractions, for scenes with millions of triangles, at 720p resolution and above 30 FPS.

1 Introduction

Reflective and refractive objects are an important component of reality found in ray traced imagery, but rarely found in real-time rendering. When these objects do appear in real-time rendering, they typically only demonstrate a single bounce using rendered or pre-rendered environment maps. These reflective and refractive objects can relay to the viewer information about the composition of the scene, such as what is hiding behind an object, or what can be seen through refractive objects.

For example, in modern real-time games, being able to see movement behind the player in a mirror would add to the gameplay experience. Also, real-time reflections are listed by Andersson [1] as a major challenge for real-time rendering. Modern real-time rendering scenes have a high triangle count, and most triangle meshes are highly detailed. Storing this data can take a large amount of memory, and computing complex visibility with reflections is quite challenging, if all geometry is taken into account.

The contribution of this paper is a general framework that enables real-time reflection and refraction rays to traverse multiple bounces, while running on graphics hardware. Our method enables complex reflections and refractions with multiple recursions to be computed within a region near the camera, and more limited interaction in the remainder of the scene. We use a hybrid approach that starts with a rasterization of the scene to compute primary ray hit points. We also generate a cube map of G-Buffers, that store depth, color, normal and material, creating an image based representation of the scene from the camera's view point. In the area close to the camera, a bounding volume hierarchy is constructed every frame to enable fully deformable objects. For the primary ray hit points that require further tracing, rays are traced into the BVH, and the G-Buffers. To trace rays in the G-Buffer we present a new approach to ray marching in the scalable, geometry insensitive G-Buffer that represents an entire scene. Rays traced into the image-based G-Buffer that intersect with reflective objects, can spawn new rays that trace back into the BVH volume or into other G-Buffers.

The different types of rays that are traced are illustrated in Figure 1. An important objective of our system is to integrate complex viewing rays into large scenes, e.g. where complex foreground reflections and refractions are integrated with non-reflective, faster to render, backgrounds.

2 Related Work

Real-Time reflections and refractions have long been possible using environment maps and graphics hardware using the technique introduced by Blinn and Newell [2]. This method is limited to a single bounce and so immediately loses much of the realism that reflections and refractions provide. True photo realism requires more complex rendering of visibility to capture realistic reflections and refractions. Realistic reflections and refractions can be achieved using ray tracing [24], but ray tracing needs to be integrated carefully into real-time systems in order to ensure high performance.

Real-time rendering in games has used environment mapped reflection and refraction for many years. Recent game engines such as Unreal Engine [3] supports features such as billboard reflections, where imposters are used to improve the accuracy of reflections.

More realistic screen space reflections are created in CryEngine 3 [19] using ray marching in screen space to create accurate reflections, but are limited to objects which appear in the view frustum. Recent research on using non-pinhole cameras for reflections is presented by Rosen [16].

Wyman [25] makes real-time refraction look more realistic by also representing a second surface. Sun et al. [20] present a technique for simulating light transmission through refractive objects using the GPU, but at much lower frame rates than we target in this work.

Our approach also captures the surrounding scene by rendering a cube map, similar in nature to image based methods such as presented by McMillan [11] where images are warped to create the final rendering. Hakura and Snyder [4] present a hybrid system that uses environment maps and ray tracing to generate realistic reflections and refractions. Their system creates multiple environment maps from different directions and distances from the reflective or refractive object in a preprocessing stage that would take significantly more time than our method, which creates maps of the entire scene's environment using the rasterization pipeline, which is significantly faster, even without modern graphics hardware. Our algorithm also improves accuracy by tracing rays at each pixel.

Cube maps have been used extensively in real-time rendering to capture lighting. Games, such as Half-Life 2 [12], assign individual cube maps to each object to create local lighting. Sebastien et al. [18] present techniques for recent games that solve the parallax issues present in the cube map technique. Szirmay-Kalos et al. [21] render a cube depth map, similar to our G-Buffer Cube Map used here, and uses it for parallax corrected access to lighting in the environment map. Their approximation to the intersection point is calculated using the point where the cube maps where created from, and the currently intersected point. In our work we use similar cube and depth maps, but use ray marching to compute accurate intersection points and trace secondary rays from those points, rays that also traverse the depth map to enable much more accurate visibility from our cube map. Knecht et al. [8] also use G-Buffers to capture illumination and relight reflective and refractive objects.

In recent work, Mara et al. [10] use a two-layer deep G-Buffer to achieve low frequency lighting effects. They also show how to use their deep G-Buffer to compute mirror reflections. Although their two-layer G-Buffer captures objects hidden to the viewer (such as objects behind a wall), the reflected object still has to reside within the view frustum. Our approach offers reflective rays in any direction, even opposite to the view direction, and with a higher quality if the reflected object is located within the BVH region.

We use a screen space approach based on deferred shading [17] to find the hit points of the primary rays and then use real-time ray tracing to trace secondary rays through the foreground objects. There is a great deal of work in real-time ray tracing which will not be reviewed here that includes data-structure construction such as BVHs and kD-trees [27] and optimizations of ray tracing performance, but this paper focuses on combining ray tracing with image based rendering to compute complex reflection and refraction effects. Recent work on real-time ray tracing on GPUs [6], using voxelization and A-Buffers to create a representation of the scene, is capable of global illumination at interactive rates using a full GPU pipeline. The algorithm presented targets higher frame rates and

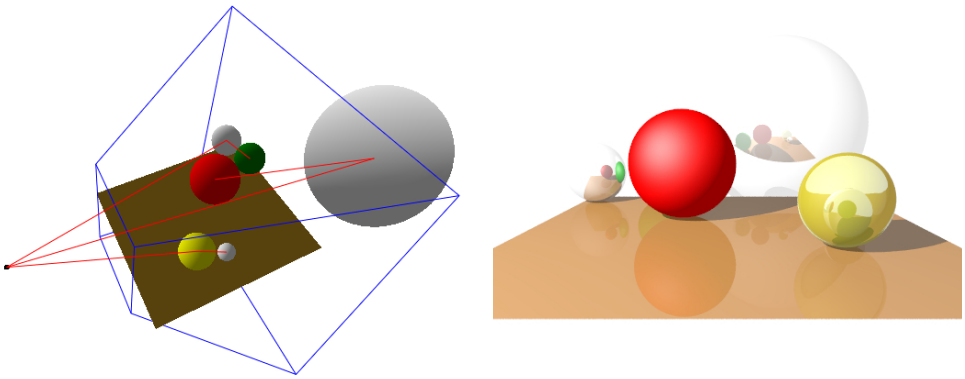

Figure 1: A scene showing the type of complex reflections possible in real-time using our system. The left image shows the camera and it's viewing frustum from a distance and the right shows the camera image. On the left three reflection and refraction ray paths are shown as red lines. The top ray reflects off a mirror sphere and reveals a green sphere that is occluded in screen space. The middle reflection bounces off a large mirror sphere that is behind our BVH region close to the camera, but the viewing ray is still traced back into the BVH to show the back of the red sphere. The lower ray traces through a refractive yellow sphere and is then reflected back into the yellow sphere. All these complex ray paths are not possible with existing screen space techniques.

resolutions in order to fit more easily into modern real-time rendering engines.

Interactive global illumination attempts to accurately model the interaction of light and matter by rendering frames in less than a second. Ritschel et al. [15] survey the current state of the field and we take a few highlights from that area that are related to our current work.

Screen space techniques improve upon basic environment mapping by using the representation of objects in the scene in screen space, but still screen space does not handle objects occluded from the view point or objects outside the view frustum. Reinbothe et al. [14] voxelize the entire scene so that ambient occlusion can be more accurately calculated in screen space.

Thiedemann et al. [23] also voxelize the scene to avoid illumination errors. Ritschel et al. [15] point out that interactive global illumination approaches approximate the geometry and lighting in the scene to reduce the complexity, because low frequency representations are sufficient for lighting. For accurate reflections and refractions, accurate geometry is required and for this we use real-time BVH construction. Recent improvements in accurate indirect illumination using BRDFs [26], demonstrate future directions for improving the image quality of our work, but since their performance is limited, they are beyond the scope of this paper.

3 Algorithm

Our rendering algorithm is based on a hybridization of existing rasterization and ray tracing techniques. It balances visual quality and performance in such a way that multiple bounce reflections and refractions of complex scenes are possible in real-time on current graphics hardware.

Distant geometry in the scene is represented by image-based maps that reduce scene complexity, but still allow rays to recursively reflect and even refract if the refraction goes to a sky box. The maps are a set of six G-buffers arranged in a cube map style. Geometry close to the view camera is represented in a BVH that is traversed with a real-time ray tracer. In order to facilitate fully dynamic scenes in real-time, a full rebuild of the BVH around objects close to the camera is performed each frame. An overview of the partitioning of the scene geometry is shown in Figure 2.

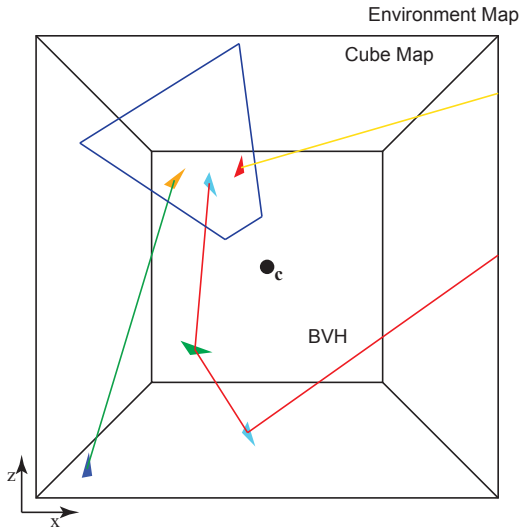

Figure 2: An X-Z 2D diagram of our rendering setup shows the different regions we break the scene into and how they are represented for rendering. Objects near to the camera (**c**) are inside a BVH. Objects further away are rendered into 6 G-Buffers that are stored as a Cube Map. Objects beyond the Cube Map will be represented by the typical sky box in an outer Environment Map. The Cube Map faces are rendered with the camera at the center point **c**. A blue trapezoid shows the view frustum where three paths are traced from the primary view G-Buffer into the BVH and intersect with objects in the BVH (red ray), objects in the Cube Map (green ray) and going through two Cube map faces (yellow ray). The red ray shows an example where a ray can trace in and out of the Cube Map and BVH regions.

Each frame of our algorithm performs the following steps:

1. Rasterize primary visibility
2. Render a G-buffer cube map
3. Build the BVH of geometry near the view camera
4. Perform primary shading and generate secondary rays
5. Recursively traverse the BVH and G-buffer cube map

These steps are further explained in the following sections.

3.1 Primary Visibility

The first pass of the algorithm is the same as the first step of a standard deferred renderer. Primary visibility depth values and normals are stored in 2D textures that are used as a G-buffer. The only variation with our G-buffer compared to one commonly used in deferred rendering methods is that a material index rather than a specularity value is stored in the alpha component of the normal texture. These material indices are later used as material identifiers by the ray tracer.

3.2 The Cube Map

The cube map in our algorithm is an image-based data structure that is updated once every frame. The cube map is used to reduce scene complexity for ray tracing by rasterizing distant geometry. It stores the same kind of G-buffers as the primary visibility pass does, but instead of having one G-buffer, an individual G-buffer is stored per cube face. The cube origin is set to the view camera's world space position and the near planes of the cube map cameras define the boundary between BVH ray traversal and image-based ray marching. To avoid issues when a ray travels parallel to the diagonal planes of the view frustums of the cube map camera faces, the cube map cameras' field of view (FOV) is slightly wider than 90 degrees. In our implementation the FOV is set to 90.2 degrees (Figure 3). By widening the FOV, a ray existing in one of the cube's diagonal planes will always belong

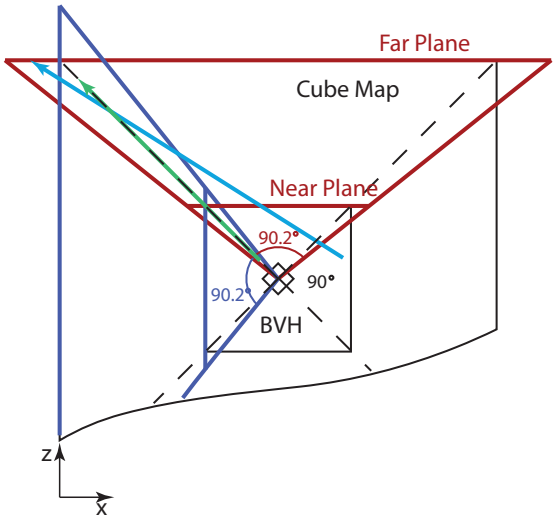

Figure 3: It is always possible to get a ray that goes along the diagonal planes (green ray) that separate the 90 degrees frustums of the cube map's faces. A drawback of having the view camera centered in the cube is that this is a frequent event. In our approach we widen the FOV of the cube map cameras so that the green ray always belongs to one or the other cube map side, and once the side to traverse is picked, the ray will continue in that side and not risk repeatedly switching sides. The blue ray displays a case where the ray first only belongs to the red frustum and then enters the overlapped region. Entering the overlapped region doesn't mean that the ray will switch side. The blue ray continues in its current side for as long as it is within the red frustum. A ray aligned with the diagonal plane of the red view frustum would only be able to switch to the blue frustum once, since the new switching planes would then be the blue view frustum's diagonal planes.

to at least one of the cube sides when entering the cube map. It is not important which side a ray belongs to, the first side a ray is tested positively against is chosen for ray marching.

3.3 BVH Construction

Our BVH implementation builds an LBVH, the linear BVH approach by Lauterbach et al. [9], where tree construction is reduced to a sorting problem. Parallelism is further improved by applying a tree and axis aligned bounding box (AABB) construction algorithm similar to the one by Karras [7].

Only geometry residing inside the cube defined by the cube map cameras' near planes is represented in the BVH. Before the BVH is constructed, a per object culling pass is performed. If an object's AABB overlaps with the cube then all triangles of that object are transformed to world space and a second culling pass is executed. The second pass performs per triangle culling. This is motivated since any ray that leaves the BVH boundary will enter the cube map and not continue in the BVH, even if the BVH contained triangles from partially overlapping objects. By culling triangles from objects partially overlapping the BVH region, no time is wasted on building a tree that includes triangles that would never be intersected anyway. Per triangle culling also enables a more balanced BVH of the triangles that actually are inside the BVH region. Since a lot of geometry is represented in the cube map, the BVH becomes much smaller, with the benefits of both reduced construction time and faster ray traversal.

By introducing a small overlap of the BVH and the cube map the possibility of a gap at the boundary due to precision errors is removed. In the overlap, an intersection occurs either in the map or in the BVH.

It is also possible for objects to have individual precomputed BVHs, rather than a per-frame full BVH rebuild, and that rays intersecting an object are transformed to the local frame of the object before continuing traversal. However, this method only works when applying rigid body transformations. Our method is capable of handling any types of transformations, as an example, procedurally animated meshes.

3.4 Ray Tracing: BVH and Cube Map Traversal

Our ray tracer approximates Whitted ray tracing by letting rays recursively traverse through the two different data structures, the BVH and the cube map. In fact, the cube map can itself be considered an approximation of a BVH. And as it is an image based data structure, once rasterized, ray traversal time in the cube map is constant in relation to scene complexity. The size of the BVH and cube map can be chosen arbitrarily and as the BVH is increased in size, to cover a larger part of the scene, our method converges towards a complete Whitted ray tracer.

Before ray tracing begins, the view camera G-Buffer is used to compute shading of the primary intersections and to determine whether a visible object should generate secondary rays or not. A spawned recursive ray continues to traverse the scene until it hits a diffuse surface, exits through the far planes of the cube map (and intersects a sky box in an outer environment map), or the maximum recursion depth is surpassed. The first recursion of a ray may start either in the BVH or in the cube map. Where it starts is simply decided by testing if the origin of the ray is inside the BVH bounding box or not. A recursive ray is initially defined in world space as

$$r_w(t) = o_w + d_w t, \;\; \{o_w, d_w\} \in \mathbf{R}^3, \; |d_w| = 1, \; t > 0.$$

If a ray currently traversing the BVH doesn't intersect any geometry, then it is instead sent to intersect with the world space representation of the cube map cameras' near planes. Given an intersection in the cube map side $i \in [1,6]$ at the parameter value $t_i > 0$, the two points

$$p_0 = r_w(t_i) \;\; and \;\; p_1 = r_w(t_i + \varepsilon)$$

can be computed along the ray, where the offset $\varepsilon > 0$ is a small value that extends the ray slightly into the cube map side. By multiplying the points p_0 and p_1 with the view projection matrix M_i given by the camera that corresponds to cube map side i, the transformed points

$$p_0' = M_i p_0 \;\; and \;\; p_1' = M_i p_1,$$

are computed and further used to define the map ray used

for ray marching as:

$$r_m(t) = \underbrace{p'_0}_{o_m} + \underbrace{\frac{p'_1 - p'_0}{|p'_1 - p'_0|}}_{d_m} t$$

with the components of o_m and d_m in the range $[-1, 1]$.

Rays that leave the BVH and enter the cube map use a similar ray marching technique to that of per-pixel displacement mapping [5] and Parallax Occlusion Mapping [22]. An important difference is that where previous ray marching techniques expect an orthogonal height map to traverse, in our method, each face of the cube map is represented in perspective. A second difference, and an result of the perspective projection, is that a ray can exit one face of the cube map but still be an active ray, and enter a neighbouring side of the cube map. A ray can also bounce between the cube map and the BVH and back again, as many times as the recursive traversal needs before reaching one of the terminating conditions.

If the map ray, r_m, currently traversing the cube side i doesn't intersect anything in the map, then there are five alternatives to exit the cube map sides for continued traversal and one alternative which would terminate the ray. If the z-component $r_{mz} > 1$, then the ray exits through the far plane of the cube and is sent to intersect the sky box as a final traversal step. If the ray traverses in the negative z-direction and $r_{mz} < -1$, then traversal continues in the BVH, using the original ray r_w. The other four alternatives are when the ray exits through the x- or y-axis and continues in the cube map side j. Given the side i and the map exit condition it is possible to directly pick the side j to traverse. Before the ray can continue in cube side j, r_m is transformed back to world space and further transformed to the cube side j's space using the jth view projection matrix. The matrices from any side i to any other possible side j can be precomputed to speedup the transition from one cube side to another.

The sampling rate n while traversing a cube map side depends on the angle between the normal N_w of the intersected plane and the ray direction r_w and is computed in a similar way as it is done by Tatarchuk [22], $n = n_{min} + N_w \cdot r_w(n_{max} - n_{min})$. However, once the ray is transformed to normalized device coordinates (from r_w to r_m) the transformed normal will always be directed along the z-axis and $N_m \cdot r_m$ simply becomes the z-component of r_m.

To avoid stretching artifacts when a ray that is close to parallel to the current cube map side intersects an object, and since the cube map only stores one layer of depth values, objects represented in the cube map can be considered to be thin or thick. The thickness value of an object is proportional to the amount of stretching permitted by that object when intersected in the cube map. If a ray, currently traversing the cube map, intersects an object that is considered thin, instead of stretching the object, the ray simply misses and continues directly to the sky box. Whether an object is thin or thick is a per material property which can be chosen arbitrarily by an artist. The thickness value ranges between 0 and 1, where a thickness of 0 represents a perfectly thin object and a thickness of 1 represents an object that stretches to the far plane. Smaller moving objects seem to visually benefit from being considered rather thin, and static objects, such as walls (which shouldn't let rays pass behind them anyway), should preferably be considered thick. Refractive objects in the cube map are always considered thin and once intersected, ray traversal is cancelled and the refracted ray is sent to the sky box.

3.5 Shadows and Deferred Rendering

It would be possible to compute accurate ray traced shadows inside the BVH, complying to the restriction that the light sources also reside within the BVH. If the light sources and thus possible occluders are positioned outside the extent of the BVH it is no longer possible to guarantee correct shadows using conventional ray traced shadow rays. However, since our method is highly compatible with the deferred rendering pipeline, it is straight forward to incorporate shadow maps, or any other effect or post processing filter commonly used with deferred rendering, to our method. Computing shadows using shadow maps has an insignificant impact on rendering performance.

4 Results

We implemented our method using OpenGL and CUDA 5 on a 32-bit Windows 7 PC with an Intel Core i7 and an Nvidia GeForce 680, and tested it on several scenes. For the larger San Miguel scene, we used 64-bit Windows and an Nvidia Quadra K5000 to generate the full BVH

ray traced images. Figure 4 shows two images from each of our test scenes in comparison to an accurately rendered image, using a BVH for the entire scene, and a colored coded image that shows the size of the BVH near the camera. The San Miguel model is from PBRT [13]. The San Miguel scene uses only per triangle frustum culling, since per object frustum culling is not possible because of its file format. We would expect much better performance if per object culling was implemented as it is for the other scenes, even so, our method still manages to render The San Miguel scene at an average speed of 10 frames per second.

Figure 5 shows the frame time for rendering a 200 frame sequence in the Sponza-Buddha-Bunny model. Each frame is broken down into the CUDA kernels that are used for rendering each frame. The breakdown shows that the ray tracing kernel is the dominant part of rendering, with the cube map rasterization of the scene taking little of the overall rendering time. The results show that on average our algorithm is four times faster than using a BVH for the entire scene.

The Chess scene is considered a pathological case when it comes to computing approximated reflections. This because of its many reflective convex objects (288 chess pieces and 9 spheres) where many of them reflect and interreflect each other. Yet, our method accurately computes reflections nearby the view camera and successfully approximates reflections far away. This can be compared to what is possible in, as an example, unreal engine [3], where reflections of dynamic objects only are achieved in screen space (and only one recursion is possible), and off screen reflections have to be pre-computed and stored in reflection environment maps. Since only static objects are visible in the reflection environment maps, but all objects in the Chess scene are dynamic, no reflections at all would be possible from the reflection environment maps. Figure 6 shows the performance of our method for the Chess scene compared to the full BVH version. The results show a similar characteristic for both methods as the rendering time is dominated by ray tracing for both our method and the BVH version. But our method always shows significant improvement in performance due to the use of the cube maps for storing the scene.

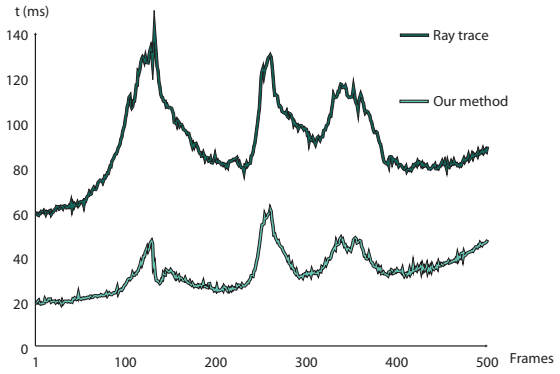

Figure 6: Performance comparison for the Chess scene between our method and using a full BVH for the scene. Both methods rasterize primary visibility. Our method always out performs full BVH ray tracing by being at least twice as fast and up to 4 times faster.

To fairly compare the performance of our method versus a full BVH ray tracer, we have chosen to rasterize primary visibility in both methods, which brings the full BVH ray tracer closer to real-time performance. Even so, our method always performs better than a full BVH ray tracer.

4.1 Limitations

While accurate reflections and refractions are achieved inside the BVH, this is not possible in the cube map. The G-Buffers only represent a single depth value without any thickness. So objects that have some thickness, details that are behind the front or objects that are hidden behind this depth value are not represented in the depth map and appear missing in some rays. This results in some artifacts, but our BVH close to the camera ensures that the artifacts are in distant geometry and are only present for secondary rays. Even with this limitation, the resulting images in real-time applications have a more realistic look when rendering multiply recursive reflections and refractions.

The first row of figure 7 presents a minor artifact in the Chess scene where the reflected chess piece on the board is slightly stretched (dark pixels to the right of the chess piece) due to its thickness value and that the reflected

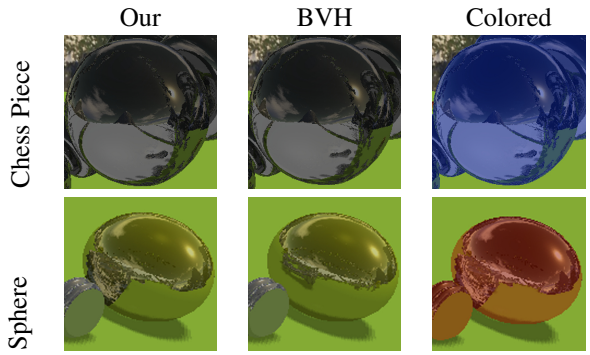

Figure 7: In the first row a minor stretching artifact is visible in the reflected chess piece on the board (visible as darker pixels to the right of the chess piece). The second row display an artifact where an object (chess piece) is covering a reflective object (sphere) in the cube map and thus important scene information between the two objects is lost. Neither of the two artifacts can take place inside the BVH and so only occurs in the background where the G-buffer cube map is used to store scene information. The magnified regions can be located by red boxes in the Chess scene images in figure 4.

rays' origins aren't shared with the cube map cameras' origin.

Another artifact, also displayed in figure 7, is when an object is covering a reflective object in the cube map. This artifact has a lower probability to appear near the camera (and can't appear inside the BVH) than further away, due to the increased possibility of having objects covering each other in the cube map the further away they are represented. A reflected ray can detect that it is behind another object, but there is no information about what to intersect, and as a fallback, the ray is sent to do a look-up in the sky box.

Rendering performance is greatly affected by the type of materials in the scene and also by the size of the BVH. If a scene contains many reflective and refractive materials, performance is naturally reduced due to the high recursive ray count. The reduced performance in scenes containing a lot of reflective materials can be mitigated by adapting the size of the BVH, thus a trade off between performance and image quality is made.

Figure 8: In our method, refractive objects residing in the cube map cannot truly refract incoming rays due to the lack of information behind it, and as a fallback method a typical real-time refraction method is used, where rays simply does a look-up in the cube map. The magnified region presented can be located as red squares in the results image (figure 4) of San Miguel.

A carefully sized BVH is in some cases vital to minimize the presence of possible artifacts using our method. One artifact that would be too interfering had it not been pushed to the background by a, for this scene, suitably sized BVH is displayed in figure 8. The refractive objects (water pitcher and glasses) in the back of the San Miguel scene are only stored in the cube map, and thus no information about what is behind them exists. Instead of computing accurate refractions, a typical real-time refraction approximation is used where the objects are considered thin and rays simply refract only once and do a look-up in the cube map.

The G-Buffers require a reasonable amount of memory. If needed, the G-Buffers may be rendered at a lower resolution in order to reduce memory usage. However, a reduced G-Buffer resolution would also affect image quality.

5 Conclusion

We have presented an approach for rendering multiple bounce reflections and refractions in real-time using rasterization and ray tracing on modern graphics hardware. Our technique is capable of rendering objects typically not seen in previous real-time screen based techniques at real-time rates of between 30 and 60 FPS for 720p images. Since the BVH can be arbitrarily sized, our technique is highly customizable to scene or performance requirements.

Since our approach is highly compatible with current rendering approaches, such as deferred rendering, we hope it will impact future applications and enable new types of interactions and improved visibility in real-time rendering.

Supplemental Materials

A paper web page with additional resources can be found at:
http://fileadmin.cs.lth.se/graphics/research/papers/2014/r5/.
At the web page we present videos of our test scenes rendered with our method, full BVH ground truth videos, and videos visualizing the extents of BVHs and cube maps.

Acknowledgements

To ELLIIT and Intel Visual Computing Institiute for funding. Thanks to TurboSquid artist cjx3711 for the chess piece models.

References

[1] Andersson, J.: Five Major Challenges in Real-Time Rendering. In: Beyond Programmable Shading course, SIGGRAPH (2012)

[2] Blinn, J.F., Newell, M.E.: Texture and reflection in computer generated images. Commun. ACM **19**(10), 542–547 (1976)

[3] Epic: Epic games unreal engine. URL `http://www.unrealengine.com`

[4] Hakura, Z.S., Snyder, J.M.: Realistic reflections and refractions on graphics hardware with hybrid rendering and layered environment maps. In: Proceedings of the 12th Eurographics Workshop on Rendering Techniques, pp. 289–300 (2001)

[5] Hirche, J., Ehlert, A., Guthe, S., Doggett, M.: Hardware accelerated per-pixel displacement mapping. In: Proceedings of Graphics Interface 2004, GI '04, pp. 153–158 (2004)

[6] Hu, W., Huang, Y., Zhang, F., Yuan, G., Li, W.: Ray tracing via GPU rasterization. The Visual Computer **30**(6-8), 697–706 (2014)

[7] Karras, T.: Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In: High-Performance Graphics, pp. 33–37 (2012)

[8] Knecht, M., Traxler, C., Winklhofer, C., Wimmer, M.: Reflective and Refractive Objects for Mixed Reality. IEEE Trans. Vis. Comput. Graph. **19**(4), 576–582 (2013)

[9] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D.P., Manocha, D.: Fast bvh construction on gpus. Comput. Graph. Forum **28**(2), 375–384 (2009)

[10] Mara, M., McGuire, M., Luebke, D.: Lighting Deep G-Buffers: Single-Pass, Layered Depth Images with Minimum Separation Applied to Indirect Illumination. Tech. Rep. NVR-2013-004, NVIDIA Corporation (2013)

[11] McMillan, L., Bishop, G.: Plenoptic modeling: an image-based rendering system. In: Proceedings of SIGGRAPH 95, Annual Conference Series, pp. 39–46 (1995)

[12] McTaggart, G.: Half-life 2 shading. In: Direct3D Tutorial, GDC (2004)

[13] Pharr, M., Humphreys, G.: Physically Based Rendering: From Theory to Implementation, 2nd ed. Morgan Kaufmann Publishers Inc. (2010)

[14] Reinbothe, C., Boubekeur, T., Alexa, M.: Hybrid ambient occlusion. EUROGRAPHICS 2009 Areas Papers (2009)

[15] Ritschel, T., Dachsbacher, C., Grosch, T., Kautz, J.: The state of the art in interactive global illumination. Computer Graphics Forum **31**(1), 160–188 (2012)

[16] Rosen, P., Popescu, V., Hayward, K., Wyman, C.: Nonpinhole Approximations for Interactive Rendering. Computer Graphics and Applications, IEEE **31**(6), 68 –83 (2011)

[17] Saito, T., Takahashi, T.: Comprehensible rendering of 3-D shapes. In: Computer Graphics, vol. 24, pp. 197–206 (1990)

[18] Sébastien, L., Zanuttini, A.: Local image-based lighting with parallax-corrected cubemaps. In: ACM SIGGRAPH 2012 Talks, pp. 36:1–36:1 (2012)

[19] Sousa, T., Kasyan, N., Schulz, N.: Secrets of CryENGINE 3 graphics technology. In: ACM SIGGRAPH 2011 Courses, Advances in Real-Time Rendering in 3D Graphics and Games (2011)

[20] Sun, X., Zhou, K., Stollnitz, E., Shi, J., Guo, B.: Interactive Relighting of Dynamic Refractive Objects. ACM Transactions on Graphics **27**(3), 35:1–35:9 (2008)

[21] Szirmay-Kalos, L., Aszódi, B., Lazányi, I., Premecz, M.: Approximate ray-tracing on the gpu with distance impostors. Computer Graphics Forum **24**(3) (2005)

[22] Tatarchuk, N.: Dynamic parallax occlusion mapping with approximate soft shadows. In: Proceedings of the 2006 symposium on Interactive 3D graphics and games, I3D '06, pp. 63–69 (2006)

[23] Thiedemann, S., Henrich, N., Grosch, T., Müller, S.: Voxel-based global illumination. In: Symposium on Interactive 3D Graphics and Games, I3D '11, pp. 103–110 (2011)

[24] Whitted, T.: An improved illumination model for shaded display. Commun. ACM **23**(6), 343–349 (1980)

[25] Wyman, C.: An approximate image-space approach for interactive refraction. ACM Trans. Graph. **24**(3), 1050–1053 (2005)

[26] Xu, K., Cao, Y.P., Ma, L.Q., Dong, Z., Wang, R., Hu, S.M.: A Practical Algorithm for Rendering Interreflections with All-frequency BRDFs. ACM Transactions on Graphics **33**(1), 10:1–10:16 (2014)

[27] Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time KD-tree Construction on Graphics Hardware. ACM Transactions on Graphics **27**(5), 126:1–126:11 (2008)

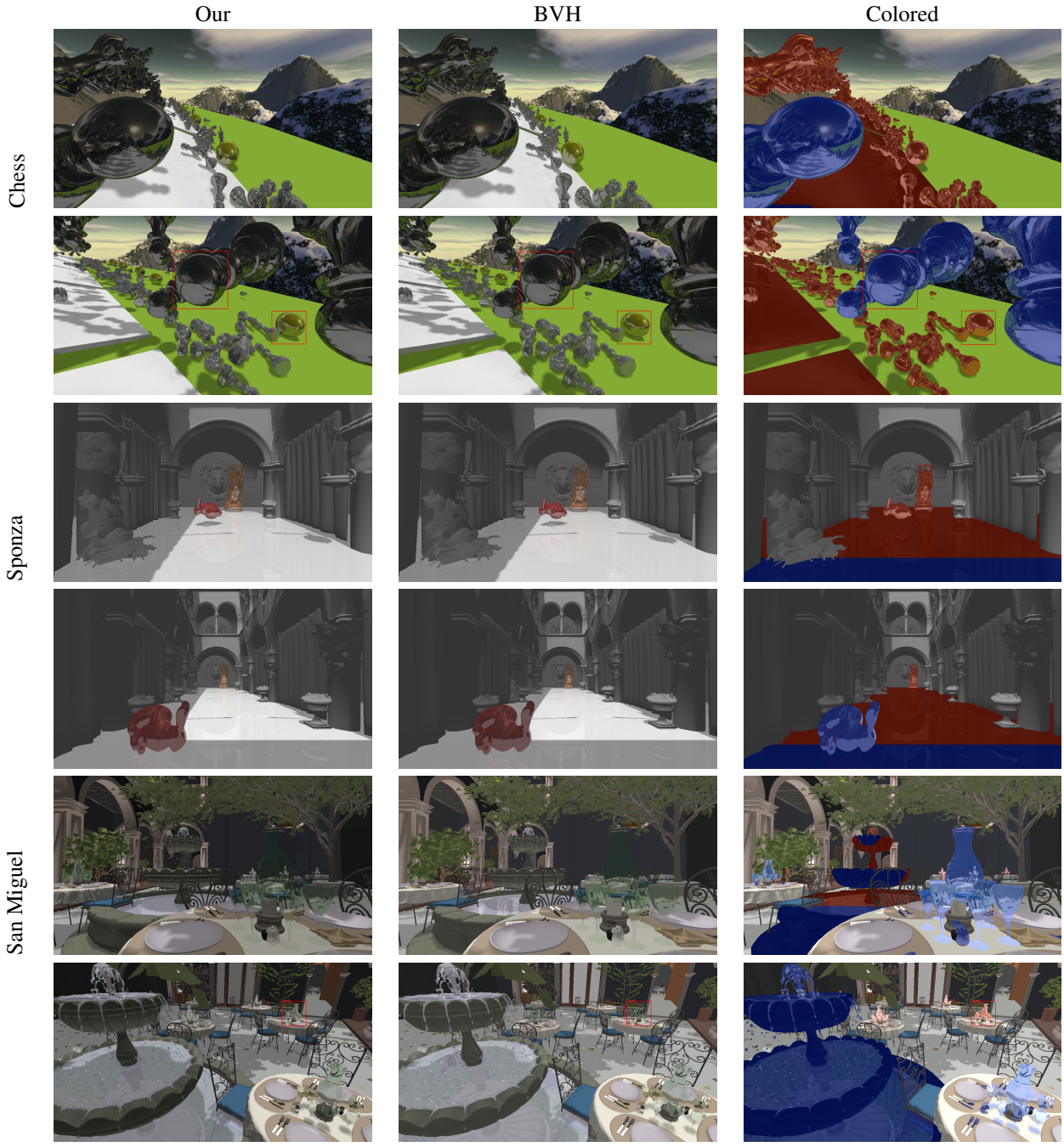

Figure 4: Three test scenes rendered from left to right with our algorithm (Our), using the BVH only (BVH), and with geometry inside the BVH colored blue and geometry in the cube map colored red (Colored). For the colored image, only pixels that contain reflective material that starts a ray are colored according to which area of the scene the ray is started in. The number of triangles for each scene is Chess 2,149,944, Sponza Buddha Bunny 1,354,743 and San Miguel is 10,500,551.

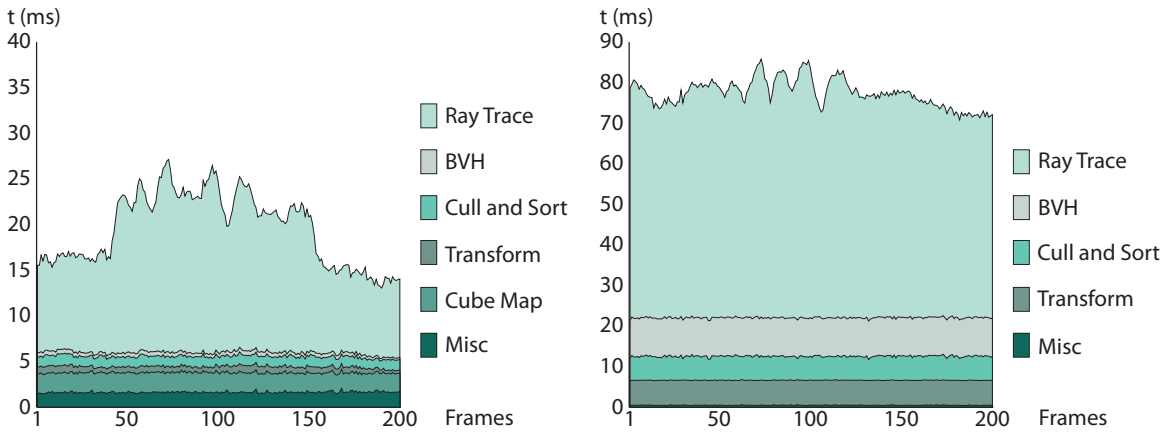

Figure 5: Rendering time breakdown of the 200 frames Sponza-Buddha-Bunny animation. Our method to the left compared to full ray tracing to the right. The scene is rendered with a maximum of four ray recursions. The improved performance is mostly due to the improved ray traversal in the cube map over the cost of BVH ray tracing. For this scene the BVH has been optimized to give high image quality and good performance. But since the size of the BVH used is chosen arbitrarily, performance is dependent on this trade-off, and the scene.