Real-Time Ray Tracing with Spherically Projected Object Data*

Bridget Makena Winn, Reed Garmsen, Irene Humer[0000−0003−2647−4813], and Christian Eckhardt[0000−0001−7721−2763]

California Polytechnic State University, San Luis Obispo, California
Email corresponding author: ceckhard@calpoly.edu

Abstract. As raytracing becomes feasible in regards to computational costs for real-time applications, new challenges emerge to achieve sufficient quality. To aim for an acceptable framerate, the amount of consecutive rays is strongly reduced to keep the workload on the GPU low, but sophisticated approaches for denoising are required. One of the major bottlenecks is finding the ray intersection with the geometry. In this work, we present a fast alternative by pre-computing a spherical projection of an object and reduce the cost of intersection-testing independent of the vertex count by projecting the object onto a circumscribed sphere. Further on, we test our Spherical Projection Approximation (SPA) by implementing it into a DirectX Raytracing (DXR) framework and comparing framerates and outcome quality for indirect light with DXR's native triangle intersection for various dense objects. We found, that our approach not only hails comparable quality in representing the indirect light, but is also significantly faster and consequently provides a raytracing alternative to achieve real-time capabilities for complex scenes.

Keywords: real-time raytracing · DiretX raytracing · intersection shaders.

1 Introduction

In modern real-time Computer Graphics, the trend to enhance render quality, in specific light bleeding and reflections for real time applications is still ongoing. While techniques such as direct lighting, pre-computed light maps and more sophisticated methods like ambient occlusion are still broadly standard, other techniques that support real-time global illumination already exists, accompanied with certain limitations. Common examples are Voxel Cone Global Illumination (VXGI)[1], Light Propagation Volumes (LPV)[2] and Reflective Shadow Mapping (RSM)[3]. However, VXGI and LPV quanitize the involved geometry, making it challenging to find a reasonable balance between quality, memory usage and frame-time and suffer under artifacts due to the voxelization, especially for moving objects. RSM is a post-processing technique and as such difficult to implement off-screen light-information effectively. Raytracing[4], one

* Supported by organization x.

of the oldest rendering technique, was hardly considered to be used in real-time applications due to its heavy computational workload. According to the render equation[5], respectively the Bidirectional Reflectance Distribution Function (BRDF)[6], theoretically every primary ray (cast from the camera view) intersecting with geometry casts an infinite number of secondary rays, invoking even more rays on each consecutive hit. Modern render-programs strongly reduces the ray count leading to a noisy outcome, with a need to be post-processed with competent denoising algorithms.

Recently, new graphic card generations (NVidia Titan V, 20XX) are designed to hardware-accelerate real-time ray-tracing. Supporting that, DirectX Raytracing (DXR)[7] is a Microsoft developed API based on DirectX 12. The DXR shader table and state objects allow the GPU to spawn rays and shaders in parallel [11] and its acceleration structure for efficiently finding the correct intersection-triangle per ray even allows multi-object ray intersections [12]. Several games implemented DXR ray-tracing into their render pipeline utilizing a single secondary reflection ray, improving the quality of reflective surfaces. Using more than one secondary ray has still a significant impact on the real-time capabilities. State-of-the-art ongoing research investigates effective GPU denoise algorithms aiming to produce high quality results with just a small set of secondary rays. Further on, studies have been done to accelerate shadow computations in DXR [10]. Noteworthy, there are several other, less efficient methods to approximate object color and position during raytracing, with and without precomputation [13] [14].

In this work, we present a different approach to reduce the workload for real-time raytracing with DXR. The DXR programmable interface allows the coding of intersection-shaders, which are called to evaluate if a ray hits an object. Since objects in modern 3D graphics consist of up to several hundreds of thousands of triangles, a highly effective ray-triangle intersection testing as well as a fast hierarchical data structure for finding the right triangle is intrinsic for DXR, but still a bottleneck regarding real-time. Our approach reduces the complexity of any object by projecting it to a circumscribed sphere. Hence, all triangle testing per object is reduced to one ray-sphere intersection. For this publication, we present the implementation as well as the qualitatively and quantitatively results comparing objects of different vertex count.

2 Methodology

2.1 Testing environment

We utilized DXR to run real-time raytracing on NVidia Titan V graphics card (10XX series) (+ model, CPU, GPU). We used the OpenGL graphics API to generate textures on a MacBook Air 2015. Code was developed on Visual Studio 2019.

2.2 Direct-X 12 Raytracing Overview

To achieve global illumination in real-time, we use DirectX Raytracing (DXR with DirectX 12), an expert-level graphics API that takes advantage of driver and hardware support for raytracing on the GPU.

In the DXR pipeline, we implement an additional *intersection shader* to define behavior between a ray and a sphere geometric primitive. Further on, we also implement a *closest hit shader* to process the hit, after DXR determined the closest object in case a ray intersects multiple objects. When the ray intersects a geometric primitive, DXR chooses the corresponding shader to intersect the geometric type and uses the closest hit shader to accumulate light.

2.3 Spherical Model Projection Overview

The raytracing algorithm sends rays from a camera into the scene and shades pixels based on ray-object intersections and several bounces. It is used recursively to generate global illumination. During global illumination, every primary ray that intersects an object must send secondary rays to collect global light information. Depending on implementation, each secondary ray may invoke tertiary rays and so on. Each ray sent into the scene may intersect objects of different face density. Consequently, the performance depends on the number of rays, as well as the scene objects' vertex count. The algorithm presents challenges when implemented in real-time. In our implementation, we are concerned only with primary and secondary rays, as a minimum requirement to approximate global illumination.

DXR geometry acceleration structure is driver defined and works on an NVidia card with bounding volume hierarchy to increase computational efficiency to find the correct triangle for the ray intersection. However, depending on the complexity of the object, approaching real-time capabilities with reflection rays as well as diffuse light rays is challenging.

In our approach, we contain an object in a bounding sphere and project its complex shape to the sphere's surface. Consequently, we reduce every ray intersection to one test per object, without further searching in a bounding volume hierarchy for the specific triangle. Our implementation aims to achieve a diffuse light quality that is compareble to traditional raytracing.

During raytracing, each primary ray may hit an object containing several triangles. Secondary rays hit bounding spheres instead of hitting the enclosed object. Once a ray intersects a bounding sphere, we sample precomputed spherical textures to approximate the color, position, and normal of the contained object at the hit point. We use the values from the spherical texture to approximate diffuse light contributing to global illumination.

We precompute spherical textures for each object in our real-time scene. To project a 3D object onto the surface of a sphere, we developed a program to scan the object's surface properties and store depth, surface normal, and non-shaded texture color in textures. A camera follows a spherical path around the object and outputs information to each texture. As a result, the output quality of our

approach depends on the complexity of the object's shape. The projected depth is the perpendicular distance from the sphere surface to the object, and the depth consistently entails the point closest to the sphere surface. Consequently, surface points behind this will be neglected, see Fig. 1.

In this work, we investigate and show the quality performance under these constraints. The inside of e.g. doughnut-shaped objects will not be visible, although its contribution to secondary rays is less prominent than the outside surface. Thus the spherical intersection algorithm allows developers to further divide an object into separate parts, which are represented by spheres.

Our algorithm aims to work with static as well as with animated objects. Simple static objects may be contained in one sphere. Thus to obtain global illumination with animated objects, each sub-mesh associated with an animation-bone can be contained in its own sphere.

During raytracing, secondary rays intersect bounding spheres to approximate contributing diffuse light. After a ray hits the bounding sphere, we perform a subsequent collision test to determine if the ray intersects the projected object contained in the sphere. For that, we construct a test ray $\boldsymbol{n}$ from the center of the object S_{center}, perpendicular to the incoming ray $\boldsymbol{r}$, see Fig. 1 (a). If the length d_2 of $\boldsymbol{n}$ is less than the distance d_1 from S_{center} to the geometry surface along $\boldsymbol{n}$, we detect a hit, see Fig. 1 (b). d_1 is stored in the pre-computed spherical texture on uv-coordinates derived from $\boldsymbol{n}$ on the sphere's surface.

As can be seen in Fig. 1 (b), rays such as $\boldsymbol{r_2}$ are incorrectly detected as miss if only considering one perpendicular test ray $\boldsymbol{n}$. Our solution is to increase the number of test rays, see Fig. 1 (c): $\boldsymbol{r_1}$ trajectory through the sphere is smaller than $\boldsymbol{r_2}$ and its minimum distance to S_{center} is greater, consequently less test rays are necessary to cover the same area or the object. Thus, the amount of test rays is proportional to this minimum distance and yield at a value between 1 and a set maximum of 50 test rays. We use a boot object to test our algorithm with a non-uniform shape, see Fig. 2. We use a texture emphasizing contrasting colors for both side of the boot, in order to evaluate the quality outcome of color bleeding. Due to our model projection, the boot tip takes up a very small portion of the spherical texture, is a known limitation of SPA and the boot is intentionally chosen for comparing our approach to traditional raytracing.

2.4 Spherical Texture Generation

We generate two spherical textures of a resolution of 640×480 with four bytes per pixel. The first texture stores the unshaded texture color in the red, green, and blue channel as well as scaled depth in the alpha channel. The other texture stores the surface normals in the red, green, and blue channel and sets the alpha channel to 1.

For each pixel of the spherical textures, we rotate the camera view around the centered object along a spherical trajectory and render the object into a frame buffer. The view target is set to the center of the object. The middle pixel of each frame buffer is stored into the spherical textures with uv-coordinates depending on the camera position relative to the object. Further on, the distance of the

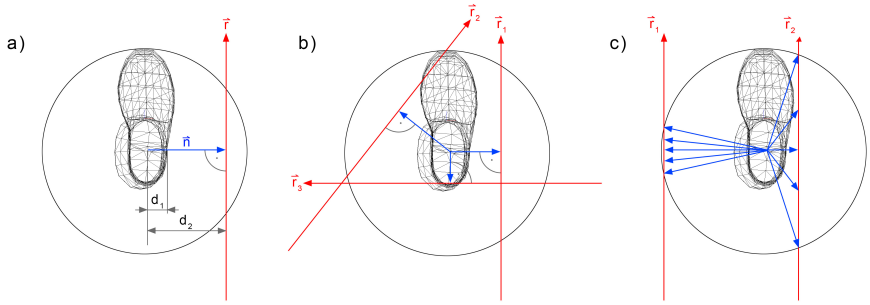

Fig. 1: Spherical Projection Approximation algorithm: (a) Ray $\boldsymbol{r}$ hitting the sphere; a perpendicular test-ray $\boldsymbol{n}$ with the length d_2 is constructed from the minimum distance point on $\boldsymbol{r}$ towards the sphere center S_{center} and is tested against the stored depth information d_1 to determine if a hit occurred. (b) Several cases of hit ($\boldsymbol{r_3}$) and miss ($\boldsymbol{r_1}, \boldsymbol{r_2}$). Evidently, $\boldsymbol{r_2}$ should hit the boot but hails a negative result with one perpendicular test ray. (c) Several test rays are generated between the two enter/exit points of the secant ray towards S_{center}.

middle pixel is divided by the sphere's radius to normalize its value, suiting to be stored as (color) byte. From the fragment shader, the non-shaded texture color as well as the normalized distance is stored into one of the spherical textures, while the surface normal is stored into the other one. Later on, to retrieve the real distance, the sphere's radius must be known.

Finally, we bind our spherical color/depth and normal textures to DXR to run our SPA approach in real-time.

2.5 SPA Integration in DXR

In DXR, we implement two shaders: The *intersection shader* defines our ray-sphere intersection and overwrites DXR's intrinsic triangle-intersection. The *sphere closest hit shader* calculates diffuse lighting after a ray-sphere intersection.

DXR allows the implementation of *intersection shaders*, so that rays can intersect user-defined geometry [9]. We implement a sphere intersection shader to define ray behavior with our bounding spheres. On a positively detected hit with the bounding sphere, our algorithm further tests if the contained object has been hit, see Fig. 1.

Since the depth values of the spherical texture are normalized, for each test ray, as described in section 2.3, it's length needs to be divided by the sphere's radius in order to compare it's value with the depth value of the spherical texture. To access the spherical textures data, we calculate the uv-coordinates based on the test ray direction as well as through the model-matrix of the object.

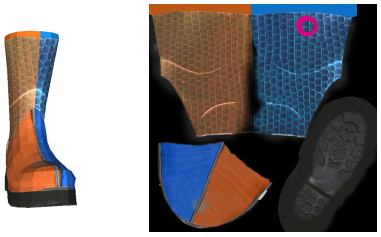

Fig. 2: We are using a simple boot object, as it represents a simple, yet non-spherical object as test-mesh with different vertex count: 326, 1317, 5233, 20865, 83329, 333057, and 1131713. The texture splits the boot into an orange and a blue half.

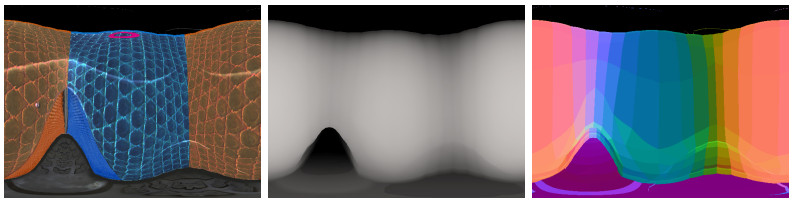

Fig. 3: Spherical texture generation with a resolution of 640×480: (left) texture color of the boot object spherically scanned. (Middle) the distance from the boot to the sphere surface is stored as alpha-value by dividing it by the sphere-radius. (Right) Normals are scanned spherically, no tangent space is needed.

If the length of the test ray is less than or equal to the sample depth, we consider this an object hit. Else, if all test rays miss, we consider this an object miss.

After DXR determines which type of object primitive was intersected closest, the corresponding *closest hit shader* is in charge of diffuse shading at the hit point. Here, the non-shaded texture color as well as the surface normal is loaded from the two spherical textures in order to shade the pixel.

3 Results and Discussion

In raytracing, primary rays are reflected by the objects and dependent on the objects' material yield to the reflection and to indirect light. Reflections are usually performed with one secondary ray, while diffuse light requires collecting colors from different directions, which is computationally expensive. To reduce the workload on the GPU, only a small number of secondary rays is sent out and leads to a strong noise in the outcome. To aim for high quality results, a sophisticated denoise step needs to be performed [16]. Nevertheless, to better

illustrate the outcome of our approach, we do not perform denoising, but rather keep the result from the ray intersection.

In Fig. 4, we present shadow ray casting using DXR triangle intersection versus SPA to demonstrate the accuracy of our approach. One has to note, we would not recommend using SPA for shadows, since primary rays are used and the difference in computational cost is marginal. To display the shadows, the boot is positioned against a white wall in Fig. 4, with DXR triangle intersection ray-casted shadow (right) compared with the SPA shadow (left). The boot is lit from the front, and global illumination has been disabled to study the shadow only. The SPA algorithm is able to produce shadows that show distinct sections of the boot, including the heel underneath. The SPA shadow is also warped towards the poles of the sphere due to the spherical approximation: The distance projected onto the spheres' surface is measured by a point on the sphere to the spheres' center, which in the case of the cylindrical shape of the upper parts of the boot leads to an increasing steepness towards the pole. Counteraction to this warping is part of a further investigation and entails different shapes, such as cylinders and boxes depending on the base object.

Fig. 4: Shadow mapping by ray intersection to illustrate the effectiveness of the SPA (left) compared to raytracing (left). The detail of the heel is clearly visible.

We created images using test ray counts of 1, 3, 5 and 10. When compared with results yield from raytracing, our images most accurately recreate color bleeding with test ray counts 5 and above. Our algorithm is able to reproduce distinct colors from the boot faces. In Fig. 5 one can see a comparison between raytracing Fig. 5 (a) and our spherical projection approximation Fig. 5 (b). The boot is placed close to the edge between two adjunct, white walls. We are using 10 test rays with a boot being perpendicular orientated towards the left wall. The boot texture is most prominent on both sides and the tip, where it is most illuminated from the top-positioned light source. Consequently, indirect light will be collected from the frontal area of the boot rather than from the back, as can be seen in both images Fig. 5 (a) and (b) by the dark gap between the orange and blue color areas. Notably is the overall less pronounced indirect color bleeding in the SPA. This is due to the color-distribution on the spherical texture, see Fig. 3, were the color-intense parts of the boot-tip are inhabiting less space as a

side effect of our approach. However, the color distribution pattern is noticeable comparable. A different angle in the same setting is displayed in Fig. 5 (c) and (d) with flipped colors on the boot, where the tip is oriented towards the edge between the walls. Besides having a weaker color bleeding analog to the previous image, the lack of color close to the tip for the SPA image is evident. Here, the SPA sphere already contacts the walls surface, and is the reason for a less intense color concentration at the boots' tip. To further test the light distribution, we rotated the boot in order to increase the direct illuminated surface area of the boot for both methods and compared them in Fig. 5 (e) and (f), and Fig. 5 (g) and (h). In Fig. 5 (e) and (f), both different colored sides of the boot are contributing to the indirect light on the wall.

Due to the spherical projection, the boot tip takes up a small portion of the spherical output texture. As a result, our algorithm does not reproduce as much color bleeding from the boot tip. Fig. 6 demonstrates the boot object with raytracing (a) and (c) and SPA (b) and (d), lit from above in the Sponza scene, taking a boot texture that yields the boot half blue and half orange. In Fig. 6, the DXR raytracing color bleeding produces blue and orange colors on the ground plane as well as on the surroundings. Compared with SPA Fig. 6 (b), both methods achieves the same color bleeding distribution from the boot tip. However, the effect from the spherical texture in SPA, storing less pixels from the boot tip as discussed in Fig. 5 is clearly visible in the difference of color intensity. One will also notice that the scene in Fig. 6 (b) lacks shadows from the Sponza objects onto itself; our pipeline uses SPA for both secondary bounces and shadow rays of the boot, while primary rays on the Sponza object are turned off. However the pipeline could be altered to include shadows from triangle intersections. In Fig. 6 (c) and (d), we rotated the camera to face the boot tip and rotated the boot down by 20◦ in order to see the top of the tip from the front view. This angle further demonstrates the color bleeding distribution and amount of color from the boot tip for both methods, DXR triangle intersection and SPA.

We have run the SPA algorithm on identically shaped boots of differing vertex count (326 up to 1131713 vertices), produced by the loop-subdivision surface algorithm. Figure 7 displays the frames per second (FPS) for each boot model. To adequately measure the FPS, we perform SPA with primary and secondary ray intersections, and compare it with DXR ray-triangle primary and secondary intersections.

Fig. 7 (a) displays the boot vertex count versus FPS for 8 secondary samples per primary ray. For each SPA curve, we use different amounts of test rays, see Fig. 1. The (green), (blue), (orange), and (grey) lines represent our SPA algorithm with test ray density of 1, 3, 5, and 10 respectively. The yellow line represents raytracing using DXR ray-triangle intersection. Test ray counts of 1, 3, 5, and 10 each begin at an FPS of 55.4, 49.7, 45.8, and 42.5 respectively for the boot with 326 vertices, see Tab. 1-7. Evidently, the maximum gain in FPS for using the SPA yields at boots with approximately 80K vertices, where we find a FPS rate of 56 compared with DXR triangle intersection of 29 FPS. It is clearly visible, that the DXR triangle intersection is strongly dependent on the vertex

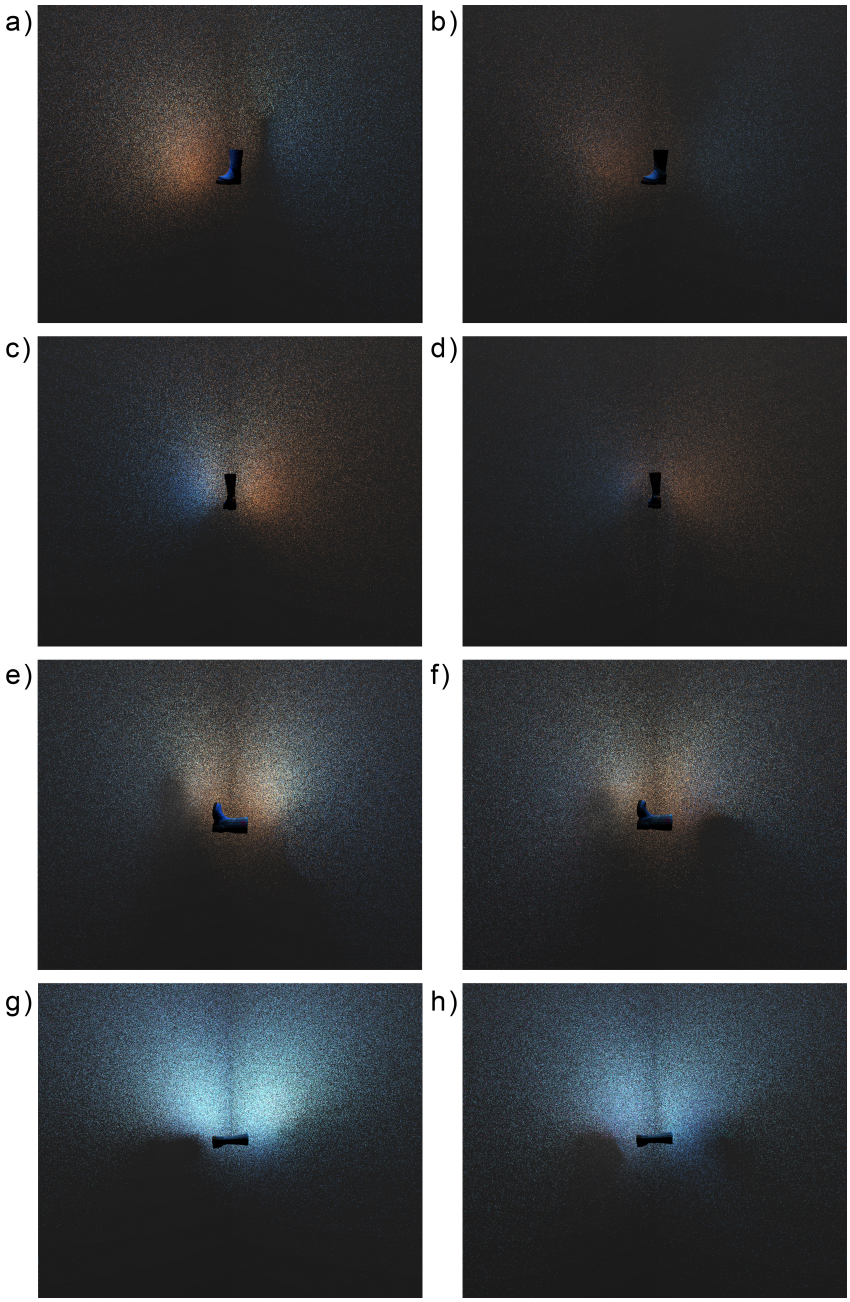

Fig. 5: Color bleeding of the boot under different angles with raytracing (left column) vs SPA (right column) against a corner of a white wall. The light-source is at the top of the boot.

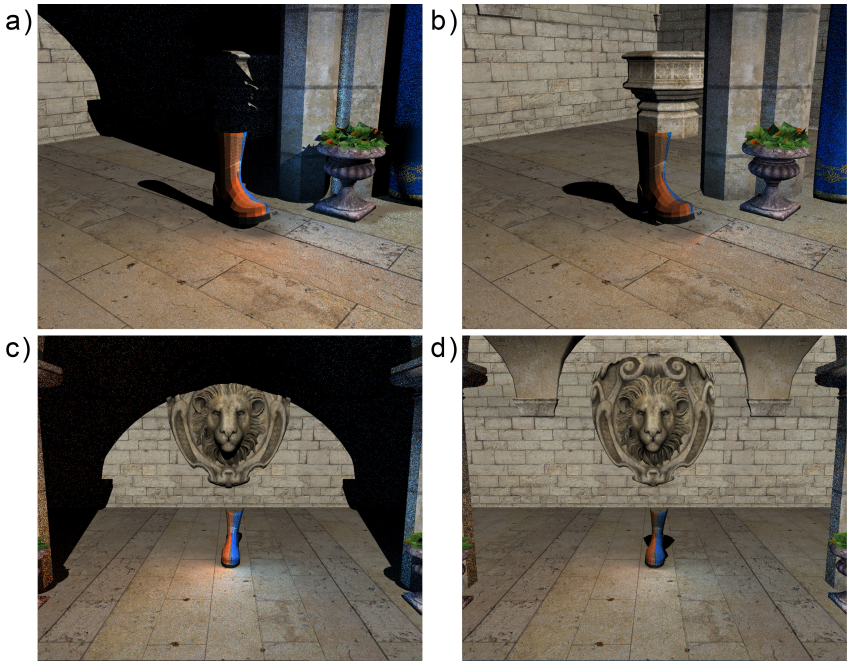

Fig. 6: Comparing color bleeding of the boot in the Sponza scene: (a) and (c) raytracing, (b) and (d) SPA. The shadow mapping for the Sponza objects SPA-image is turned off due to using primary and secondary rays with SPA only.

count, as it decreases steadily with increasing vertex density. We anticipated this effect due to the ray-triangle intersection pipeline, in which every ray may intersect one of millions of triangles per object. Further increasing the vertices leads to a dependency of the SPA as well, which is due to the vertex processing stage becoming a dominant factor in the shader pipeline. Fig. 7 (b) displays the vertex count and FPS for 32 recursive samples. Test ray densities of 1, 3, 5, and 10 begin at FPS of 16.8, 14.5, 13.2, and 12.2 respectively for the smallest boot. The highest relative gain in FPS is observed again at objects with 80K vertices comparing one step SPA with DXR ray-triangle intersection.

Fig. 7 involves all boots for Fig. 7 (c) 128 and Fig. 7 (d) 256 recursive samples respectively. In these figures, there are some fluctuation in FPS due to a steadily decreasing FPS rate, making it difficult to get accurate FPS results, since we need to measure over a longer period of time. In Fig. 7 (c) we see the vertex count and FPS for 128 recursive samples. For this many secondary samples, we do not observe a dependency of vertex count for the SPA, since the GPU is mostly occupied with solving ray-intersections compared with vertex processing. All data points can be read from the Tab. 1 through 7.

test rays	samples 8	32	128	256
0	128.8	43.6	13.3	6.7
1	55.4	16.8	4.5	2.3
3	49.7	14.5	3.8	1.7
5	45.8	13.2	3.3	1.6
10	42.5	12.2	2.9	1.4
ray	42.1	12.3	3.3	1.6

Table 1: 326 vertices

samples 8	32	128	256
130.5	49.5	12.5	6.1
55.7	16.9	4.4	2.2
47.6	14.6	3.4	1.8
46.8	13	3.2	1.4
41.6	11.8	2.8	1.3
42.3	12.8	3.1	1.5

Table 2: 1317 vertices

samples 8	32	128	256
127	45.3	12.6	5.8
56.2	17.1	4.4	2.3
46.2	13.8	3.7	1.7
43.8	12.7	6.7	1.4
40.7	11.6	3.3	1.4
35.8	10.7	2.4	1.3

Table 2: 5233 vertices

test rays	samples 8	32	128	256
0	127.2	45.8	12.4	6.2
1	56.7	17.4	4.2	2.3
3	47.8	14.2	3.6	1.7
5	44.3	13.1	3.1	1.6
10	41.3	11.9	3.2	1.5
ray	32.9	9.5	2.4	1.1

Table 1: 20865 vertices

samples 8	32	128	256
121.1	44.7	13.5	6.6
56.6	18.3	4.8	2.1
48.4	14.6	3.8	1.8
44.6	13.8	3.4	1.6
41.4	12.3	2.9	1.5
28.8	8.5	2.2	0.9

Table 2: 83329 vertices

samples 8	32	128	256
98.5	42.6	13.4	6.4
51.2	17.7	4.8	2.2
44.0	14.7	3.6	1.9
38.8	12.9	3.4	1.3
38.1	12.3	2.7	1.3
24.9	7.4	1.9	0.3

Table 2: 333057 vertices

test rays	samples 8	32	128	256
0	48.6	31.1	11.7	6.3
1	38.5	15.9	4.8	2.1
3	33.6	13.3	3.7	1.8
5	32.2	12.2	3.4	1.9
10	30.2	11.3	3.2	1.7
ray	20.9	6.9	1.6	0.2

Table 7: 1331713 vertices

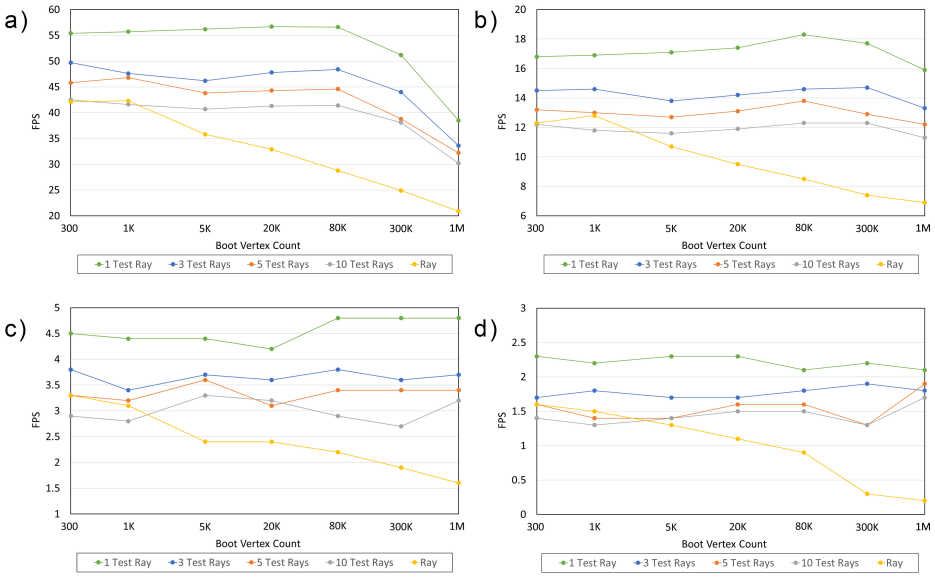

Fig. 7: Boot vertex count versus FPS for (a) 8, (b) 32, (c) 128 and (d) 256 samples samples using different amount of test rays for SPA.

4 Conclusion and Future Work

In this work, we presented a spherical projection approximation (SPA) algorithm. By spherically scanning complex objects and storing their diffuse color, normals, and depth-to-sphere-surface in textures, and substitute the objects with its bounding sphere for ray-collision detection, we were able to reduce all triangle intersection tests per object to one sphere intersection. Further on, we tested our algorithm compared with DXR triangle intersection raytracing for feasibility regarding computational workload as well as quality outcome and found striking results for both aspects. In a best case scenario for high density objects (80K+ vertices), our algorithm yields up to double the FPS rate compared with DXR. Taking shadow mapping as a test case, SPA holds a sufficient degree of detail, yet due to the spherical approach, objects are warped towards the sphere's poles.

Future work entails different approximation objects such as cylinders and boxes depending on the base objects' shape.

References

1. Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, Elmar Eisemann: Interactive Indirect Illumination Using Voxel Cone Tracing. Computer Graphics Forum (Proceedings of Pacific Graphics 2011), Volume 30, Number 7 - September 2011

2. Anton Kaplanyan and Carsten Dachsbacher: Cascaded light propagation volumes for real-time indirect illumination. In Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D '10). ACM, New York, NY, USA, 99-107. https://doi.org/http://dx.doi.org/10.1145/1730804.1730821
3. Carsten Dachsbacher and Marc Stamminger: Reflective shadow maps.In Proceedings of the 2005 symposium on Interactive 3D graphics and games (I3D '05). ACM, New York, NY, USA, 203-231. https://doi.org/http://dx.doi.org/10.1145/1053427.1053460
4. Arthur Appel: Some techniques for shading machine renderings of solids. In Proceedings of the April 30–May 2, 1968, spring joint computer conference (AFIPS '68 (Spring)). ACM, New York, NY, USA, 37-45. https://doi.org/http://dx.doi.org/10.1145/1468075.1468082
5. James T. Kajiya: The rendering equation. In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH '86), David C. Evans and Russell J. Athay (Eds.). ACM, New York, NY, USA, 143-150. https://doi.org/http://dx.doi.org/10.1145/15922.15902
6. Fred E. Nicodemus: Directional Reflectance and Emissivity of an Opaque Surface. Appl. Opt. 4, 767-775 (1965).
7. Microsoft DirectX Raytracing `https://blogs.msdn.microsoft.com/directx/2018/03/19/announcing-microsoft-directx-raytracing/` Last accessed 15 July 2019
8. SIGGRAPH 2018 NVIDIA talk, `http://intro-to-dxr.cwyman.org/presentations/IntroDXR_RaytracingShaders.pdf` Last accessed 15 July 2019
9. SIGGRAPH 2018 NVIDIA talk, `https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial-Part-2` Last accessed 15 July 2019
10. Boksansky J., Wimmer M., Bittner J. (2019) Ray Traced Shadows: Maintaining Real-Time Frame Rates. In: Haines E., Akenine-Mller T. (eds) Ray Tracing Gems. Apress, Berkeley, CA, `https://link.springer.com/content/pdf/10.1007%2F978-1-4842-4427-2_13.pdf`
11. Wyman C., Marrs A. (2019) Introduction to DirectX Raytracing. In: Haines E., Akenine-Mller T. (eds) Ray Tracing Gems. Apress, Berkeley, CA, `https://link.springer.com/content/pdf/10.1007%2F978-1-4842-4427-2_3.pdf`
12. Gribble C. (2019) Multi-Hit Ray Tracing in DXR. In: Haines E., Akenine-Mller T. (eds) Ray Tracing Gems. Apress, Berkeley, CA, `https://link.springer.com/content/pdf/10.1007%2F978-1-4842-4427-2_9.pdf`
13. Akenine-Mller T., Nilsson J., Andersson M., Barr-Brisebois C., Toth R., Karras T. (2019) Texture Level of Detail Strategies for Real-Time Ray Tracing. In: Haines E., Akenine-Mller T. (eds) Ray Tracing Gems. Apress, Berkeley, CA, `https://link.springer.com/content/pdf/10.1007%2F978-1-4842-4427-2_20.pdf`
14. SzirmayKalos, L. , Aszdi, B. , Laznyi, I. and Premecz, M. (2005), Approximate RayTracing on the GPU with Distance Impostors. Computer Graphics Forum, 24: 695-704. doi:10.1111/j.1467-8659.2005.0m894.x, `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2005.0m894.x`
15. Barr-Brisebois C. et al. (2019) Hybrid Rendering for Real-Time Ray Tracing. In: Haines E., Akenine-Mller T. (eds) Ray Tracing Gems. Apress, Berkeley, CA, `https://link.springer.com/content/pdf/10.1007%2F978-1-4842-4427-2_25.pdf`
16. Liu E., Llamas I., Caada J., Kelly P. (2019) Cinematic Rendering in UE4 with Real-Time Ray Tracing and Denoising. In: Haines E., Akenine-Mller T. (eds) Ray Tracing Gems. Apress, Berkeley, CA, `https://link.springer.com/content/pdf/10.1007%2F978-1-4842-4427-2_19.pdf`