Stereoscopic Ray-Tracing for VR

Faster Image Generation through Reprojection

MICHAL PISZCZEK, University of Washington

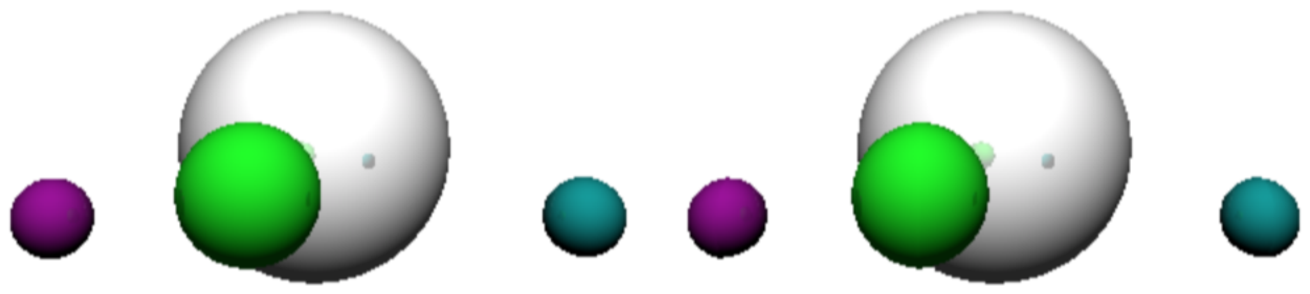

Fig. 1. Ray-Traced left and right eye images of the same scene. The difference in perspective is most visible when looking at the reflection of the green sphere in the white sphere. The right image was partially re-projected based on the left. Distortions resulting from the re-projection are visible as drag on the turquoise and purple spheres at the extremes of the scene.

We explore Stereoscopic Ray-Tracing, with the aim of doing it efficiently for the purposes of VR. Specifically, we aim to generate one of the two images necessary for stereoscopic ray-traced rendering approximately, based on the image already ray-traced for the other eye. For this purpose we re-implement and evaluate an algorithm from [Adelson 1993] that re-projects pixels from an already generated left-eye image into a right eye-image, only re-tracing when it is necessary. Additionally, in order to implement this algorithm and evaluate it, we develop a small CPU based ray-tracer that runs in the browser, based on an open source project [MacWright 2019]. We show that this re-projection algorithm significantly reduces the cost of generating one of the two images necessary, but the quality degradation may prohibit this method from being useful in VR. We propose Machine Learning as a tool for future work to correct the distortions generated by re-projection.

1 INTRODUCTION

Ray-Tracing is a powerful technique for generating high-quality realistic looking images. It works by tracing an imaginary line from the camera through pixels in the screen, and into the scene itself to see what objects are in the trace's path and thus what color the pixels should be. The depth of a ray-trace is how many times each ray is reflected within the scene, incorporating different color and lighting values from all the surfaces it impacts into the final pixel color [MacWright 2019]. The high cost of this technique comes from all the reflections that must be calculated throughout the trace, depending on the depth. Even fully obscured objects can contribute to the final image as reflections in another object, so it is difficult to save computation as one could through culling in rasterisation.

With respect to VR, ray-tracing offers the exciting potential for very realistic and immersive renderings that better reflect our reality. Unfortunately, due to the real-time nature of VR rendering and the need to render a separate image for each eye, ray-tracing is difficult to incorporate into VR systems. VR rendering must be fast, responsive and consistent to present a compelling experience. Not only can lag and dropped frames break user immersion, they can also make one feel extremely uncomfortable. With its high computational cost, now doubled by the need for two images, ray-tracing makes it difficult to sustain the necessary frame rate, at high enough a quality, to offer as consistent a VR experience as rasterization can.

Despite the difficulty of these challenges, the promise of fully immersive, ray-traced VR is so high that industry and academia are rapidly pushing forward to solve them. One such example is hardware vendor NVidia, who has recently developed a new platform to aid in the acceleration of ray-tracing [V.V. Sanzharov 2019]. In

Author's address: Michal Piszczek, michalp@cs.washington.edu, University of Washington.

contrast, the approach we will explore to enabling efficient VR ray-tracing is a software solution. A large part of the cost of ray-tracing in VR comes from having to do it twice per frame, to generate the appropriate image for each eye. A technique to make such stereoscopic ray-tracing more efficient through approximate generation of one of the required images has existed for some time [Adelson 1993]. This method works by first fully ray-tracing the left eye image for a scene, and then re-projecting a large part of those pixels into the right eye-image such that only pixels that weren't re-projected have to be re-traced.

To evaluate the effectiveness of this approach, we constructed a CPU-based ray-tracer, in Javascript, capable of running in the browser. Our implementation was based on an existing open source one [MacWright 2019]. We then implemented the stereoscopic ray-tracing algorithm from [Adelson 1993] into our ray-tracer. Upon evaluation, we noticed a marked decrease in the number of ray-traces necessary to generate the right-eye image. Though, while gaining in computational effectiveness, we also noticed some visual issues with the re-projected right-eye image. Overall, we found that out-of-the-box, this method is not suitable for VR due to the degradation in image quality, and suggest future work involving Machine Learning for image correction to make this a viable approach to VR ray-tracing.

1.1 Contributions

Our primary contributions are:

- The development of a CPU-based, real-time stereoscopic ray-tracer, capable of running in the browser and useful as a platform for experimentation.
- The implementation of the efficient stereoscopic ray-tracing algorithm from [Adelson 1993] into our ray-tracer.
- The evaluation of the algorithm from [Adelson 1993] in terms of its potential for use in real-time VR ray-tracing.

2 RELATED WORK

Effective VR ray-tracing is not a new idea and is being approached from many directions, including purpose-built hardware, deep-learning, and classical algorithms like the one we will explore.

2.1 Hardware and Deep Learning

Platforms such as NVidia's RTX system are purpose built for advancing the state of the art with respect to ray-tracing [Vk 2019]. With a special architecture that is capable of executing ray-tracing and deep-learning operations hyper-efficiently, these devices can produce realistic ray-traced images in real-time faster than ever before. The deep learning component particular allows for noisy and lower-resolution ray-traced renderings to be enhanced, in real-time, into much higher fidelity images. These improvements in general ray-traced image generation help realize the dream of VR ray-tracing, as any speed-up in generating one ray-traced image in terms of rendering time helps doubly so when the task is to generate two ray-traced images. Simply put, such raw power moves us significantly closer to VR ray-tracing at an acceptable frame-rate.

2.2 Classical Algorithms

Classical approaches to efficient stereoscopic ray-tracing include the one we will be exploring in this paper [Adelson 1993]. Such approaches aim to exploit the large degree of similarity between the left-eye and right-eye images to effectively "cheat" the rendering of one of them. One view (in our approach, the left-eye's) is fully ray-traced as normal, and is then used to approximately generate the other view. Such an approach of course has the potential to introduce massive savings in terms of rendering, depending on how much of the second image can be "cheated". If such savings are high enough, the overall cost of creating both images needed for a stereoscopic rendering can be close to the cost of just generating one image, effectively halving the computational cost of a frame. This of course would help increase frame rates, enabling effective ray-tracing for VR.

3 METHOD

To evaluate the algorithm from [Adelson 1993], we first implemented a stereoscopic ray-tracer, and then modified it to use the algorithm for approximate generation of the right-eye image. We then proceeded to evaluate the algorithm by qualitatively assessing the accuracy of the images generated by the re-projection algorithm (compared to a fully ray-traced image), and by quantitatively measuring the gains in performance.

3.1 Re-projection Algorithm

The re-projection algorithm used is taken directly from [Adelson 1993], where it is referred to as the "visible surface algorithm for creating stereo pairs of ray-traced images". The algorithm operates on the principle that, given the coordinates of a point in the left eye's projection plane, that point can be re-projected into the right eye's projection plane with some simple math.

We reproduced the algorithm almost exactly as it is presented in the original paper. As in [Adelson 1993], we assume a left-handed coordinated system with the view plane located at $z = 0$. We also assume two cameras, located at $(-e/2, 0, -d)$ and $(e/2, 0, -d)$ where e is the interpupillary distance and d is the camera's depth offset relative to the view plane. Then, given the point (x_{pl}, y_{pl}) in the left eye's projection plane, we can determine the position of this point in the right eye's projection plane using this equation,

$$x_{pr} = x_{pl} + e(z_p/(d + z_p))$$

where z_p is the depth of the point to be projected in the scene. Note that the y-coordinates of a point are the same across both views as the algorithm assumes the cameras, or "eyes" are only horizontally displaced relative to each other. This assumption does not appear to be limiting in terms of our desire to use this method for VR stereoscopic ray tracing, as the two cameras in VR are typically also only horizontally offset.

The full algorithm, reproduced from [Adelson 1993] incorporating this re-projection equation is present on the next page. To the best of our understanding, we implemented precisely this algorithm in our ray-tracer.

Algorithm 1 Visible surface algorithm for creating stereo pairs of ray-traced images reproduced from [Adelson 1993].

```
for each scan-line do
    let oldx = $-1$
    for each pixel $i$ in the left eye-image do
        Compute intersect of ray through $i$ and scene, $(i_x, i_y, i_z)$
        Calculate color C for left-eye view
        Let $j = i_x + e(i_z/(d + i_z))$
        if $j <$ width of screen then
            mark j in left image as able to re-use C
        end if
        if $j \geq$ width of screen then
            set $j = M$
        end if
        if $j -$ oldx $> 1$ then
            for each pixel k in right image between oldx and j do
                mark k in left image as unable to be re-projected
            end for
        end if
    end for
    let oldx = $j$
end for
```

4 IMPLEMENTATION DETAILS

In order to implement the algorithm from [Adelson 1993] we required a modifiable ray-tracer. There are plenty of open source ray-tracers available, but many are complicated and have far more features than necessary for our purpose of evaluating an algorithm. The complexity of these pre-built solutions, in terms of the time it would take a ray-tracing novice to understand them enough to modify them, made them unattractive. Thus, we wrote our own based heavily on the most minimal open source implementation we could find [MacWright 2019]. Our finished ray-tracer functions as a ray-tracer typically does, by projecting virtual rays through the screen into the scene to calculate the colors of pixels on the screen based on what each ray encounters as it reflects around in the scene. The mathematics of this are well established and can be referenced here [MacWright 2019].

As with our reference implementation, our ray-tracer is built in Javascript, drawing its two images (one for each eye) in separate HTML5 Canvas elements horizontally aligned on a web page. The simplicity of this approach makes it easy to modify and portable, running in any web browser supporting HTML5 Canvas elements. A brief attempt was made to port the ray-tracing code into a GPU shader, for increased performance, but it was decided that this would unnecessarily complicate implementation of the re-projection algorithm.

In terms of ray-tracing capabilities, our ray-tracer only handles sphere intersections, and only goes to depth 3 (though it can be configured to do more reflections). The scene is kept entirely static except for one revolving sphere which serves to display the real-time nature of the rendering. These simplifications were made to ease implementation, effectively producing the minimum viable real-time stereoscopic ray-tracer capable of taking advantage of the

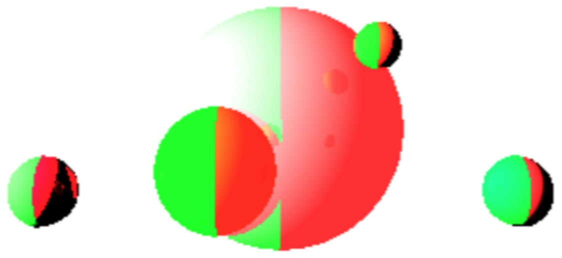
Fig. 2. The re-projected right-eye image with modified colors to show the re-projection algorithm in action. The green pixels, as well as most of the white space, have been re-projected. Only the red regions required re-tracing.

re-projection algorithm. The two cameras in the scene (representing a viewer's left and right eyes) were placed far apart enough for the difference in their perspectives to be visible across both generated images. Additionally, unlike in our reference implementation, the ray tracing code itself was written to surface the location of intersections to the top-level rendering loop as this information is required by the re-projection algorithm which resides there. Finally, instrumentation was written to track the number of ray-traces executed during the rendering of each image (the fully ray-traced left-eye image and the partially re-projected right-eye image).

5 EVALUATION OF RESULTS

To evaluate the re-projection algorithm, we considered both the computational performance achieved, as well as the quality of the resulting images.

5.1 Performance

To measure computational performance we look at relative rendering costs, in terms of ray traces needed, for each image. Our instrumentation code reveals that, with the algorithm in place, drawing the left-eye image takes approximately 87% of all ray-traces per frame, and only the remaining 13% percent are used to render the right-eye image. Clearly, the re-projected right-eye image needs much less tracing, and is thus much cheaper to produce. Put another way, the right eye-image costs approximately 85% less ray traces to produce than the left-eye image, given that the left-eye image has already been produced. The additional cost of the re-projection algorithm itself, when compared to the number of ray-traces it saves, is negligible.

5.2 Image Quality

There were several noticeable issues with the re-projected right-eye image. Firstly, objects at the extremes (such as the turquoise and purple spheres) experienced a significant amount of drag, no longer appearing as perfect spheres. Additionally, when the light source was not head on, but far off to the left or right of the scene, the boundary between re-projected and re-traced areas in the right-eye

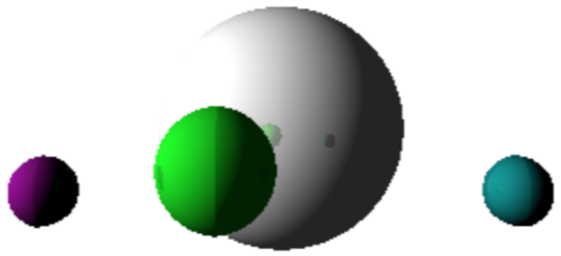

Fig. 3. Ray-Traced left and right eye images of the same scene. The difference in perspective is most visible when looking at the reflection of the green sphere in the white sphere. The right image was partially re-projected based on the left. Distortions resulting from the re-projection are visible as drag on the turquoise and purple spheres at the extremes of the scene.

image became obvious due to the lighting differences. This is visible most clearly on the green and turquoise spheres in the figure above.

Depending on the desired application, these errors may be admissible if increased performance is more important than image quality. The trade-off is quite favorable if that is the case, as based on our example, the re-projected right-eye image, while costing 85% less to compute, certainly didn't appear to look 85% worse, subjectively speaking. Of course, our scene is quite simple. Given more complex scenes the image degradation may be far more acute. For VR specifically, the image degradation, even with our simple scene, may make this method a poor choice. Such lightning errors and object drag as visible in our figures would be very noticeably in a VR environment, and would break immersion and possibly induce discomfort.

6 DISCUSSION OF BENEFITS AND LIMITATIONS

The benefit of this method is clearly the reduction in computation one gains by not having to fully ray-trace one of the two images necessary for stereoscopic rendering. The limitations of this method lie in the quality of the re-projected image. Out-of-the-box, it is only suitable for domains in which some visual errors are acceptable. Due to the discomfort experienced by users in VR when scenes are incorrectly rendered, this makes the method difficult to recommend for that domain. But for other situations, where a big jump in performance for a relatively small loss in quality is acceptable, this method would be effective.

7 FUTURE WORK

As previously stated, the key limiting factor preventing this method from being useful in VR ray-tracing is the degradation of image quality in the re-projected right-eye image that results from the approximate nature of the process. We propose that future work, involving Machine Learning and Computer Vision could tackle this issue. Platforms such as NVidia's RTX already involve deep-learning based rendering enhancements for reducing noise in ray-traced images where the number of rays is intentionally kept low for performance, and for up-scaling cheaper to create low resolution renderings in general [Vk 2019]. Potentially, such post-rendering image-correction could be applied to the re-projected right-eye images generated by our approach to "fix-up" the visual issues resulting from the re-projection. Given a large enough data set of fully ray-traced right-eye images, and re-projected right eye images, a model could be trained to apply corrections to the re-projections, generating an image more like what a full trace would create while still avoiding the cost of a full trace.

8 CONCLUSION

Overall, we find the method from [Adelson 1993] to be quite effective in terms of making stereoscopic ray-tracing more efficient. One of the two images required can be produced at a fraction of the typical cost, significantly reducing the computational load compared to fully ray-tracing two images. Despite the performance increases, the applicability of this method to VR ray-tracing is questionable due to the degradation in image quality of the re-projected image. Further work, perhaps involving Machine Learning, is needed to explore ways of "fixing-up" the re-projected image's quality to make this method truly suitable for real-time VR ray-tracing.

ACKNOWLEDGMENTS

Thank you to the UW CSE 490v Winter 2020 team (Douglas Lanman, Ethan Gordon and Kirit Narain) for the help and resources they provided throughout this project and for a great course in general.

REFERENCES

Hodges L.F. Adelson, S.J. 1993. Stereoscopic ray-tracing. *The Visual Computer* 10 (1993), 127–144. https://doi.org/10.1007/BF01900903

Tom MacWright. 2019. Literate RayTracer. https://tmcw.github.io/literate-raytracer/

Anirudh Vk. 2019. NVIDIA's Real-Time Ray Tracing Technique and AI-powered RTX Technology Explained. (2019). https://analyticsindiamag.com/nvidias-real-time-ray-tracing-ai-powered-rtx-explained/

V.A. Frolov A. G. Voloboy V.V. Sanzharov, A.I. Gorbonosov. 2019. Examination of the Nvidia RTX. *CEUR Workshop Proceedings* (2019). http://ceur-ws.org/Vol-2485/paper3.pdf