UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Lidiane Teixeira Pereira

ARRay-Tracing: A middleware to integrate real-time ray tracing and augmented reality

Juiz de Fora
2021

Lidiane Teixeira Pereira

ARRay-Tracing: A middleware to integrate real-time ray tracing and augmented reality

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientador: Jairo Francisco de Souza
Coorientador: Rodrigo Luis de Souza da Silva

Juiz de Fora
2021

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Pereira, Lidiane Teixeira.
ARRay-Tracing: A middleware to integrate
real-time ray tracing and augmented reality / Lidiane Teixeira Pereira.
– 2021.
46 f. : il.

Orientador: Jairo Francisco de Souza
Coorientador: Rodrigo Luis de Souza da Silva
Dissertação (Mestrado) – Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação, 2021.

1. Realidade aumentada. 2. Ray tracing. 3. Fotorrealismo. 4. Iluminação global. I. Souza, Jairo Francisco de, orient. II. Silva, Rodrigo Luis de Souza da, coorient. III. Título.

Lidiane Teixeira Pereira

ARRay-Tracing: A middleware to integrate real-time ray tracing and augmented reality

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 24 de setembro de 2021

AGRADECIMENTOS

Agradeço à minha mãe, pelo constante incentivo aos meus estudos e por fazer tudo que estava ao seu alcance para que eu completasse a minha formação acadêmica.

À minha irmã por ser meu exemplo de coragem e persistência e pelo apoio incondicional em todos os momentos.

Aos meus queridos amigos e amigas que estiveram ao meu lado durante todo o processo, compartilharam das angústias e vibraram comigo a cada conquista. Especialmente aqueles que me apoiaram e contribuíram de forma direta para a conclusão desse trabalho!

Aos colegas do PGCC por partilharem seus conhecimentos e tornarem essa caminhada mais leve e divertida.

Ao professor Jairo e ao professor Rodrigo pela orientação.

Aos professores e professoras do DCC por todos os conhecimentos transmitidos.

Aos funcionários e funcionárias do ICE por toda a ajuda e assistência prestada. Em especial à Sarah pela paciência e atenção no atendimento de todas as solicitações.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior pelo financiamento que foi de extrema importância para viabilizar realização desse trabalho.

“We’re flooding people with information. We need to feed it through a processor. A human must turn information into intelligence or knowledge. We’ve tended to forget that no computer will ever ask a new question.”

(Grace Hopper)

RESUMO

Nos últimos anos percebemos o aumento e a popularização de aplicações de realidade aumentada. Entretanto, é comum que esses sistemas apresentem uma considerável discrepância visual entre elementos reais e virtuais, o que acarreta na falta de realismo, sendo esse um dos motivos que desencorajam o uso desse tipo de aplicação. Algoritmos baseados em física, como o *ray tracing*, geram renderizações com um alto grau de fotorrealismo e estão se popularizando após o recente desenvolvimento de aceleradores de hardware. Alguns trabalhos presentes na literatura combinam essas duas tecnologias, realidade aumentada e *ray tracing*, de forma rígida e sem modularização, tornando a solução dependente de *frameworks* específicos. Neste trabalho, propomos um *middleware* para integrar realidade aumentada e *ray tracing* em tempo real, através do mapeamento das coordenadas da câmera no framework de *ray tracing* e de um processo de composição de imagens. Além disso, o *middleware* funciona de forma modularizada, permitindo ao usuário escolher entre bibliotecas e frameworks existentes, os que melhor se adequem às necessidades e competências do usuário. Construímos uma aplicação utilizando o middleware ARRay-Tracing para integrar dois *frameworks* de *ray tracing*, o Optix e o VKRay, a um *framework* de realidade aumentada, o artoolkitX, obtendo em tempo real reflexões e refrações mais realistas. Após a avaliação, concluímos que o ARRay-Tracing possibilita a integração modular sem acrescentar sobrecarga de processamento à aplicação, atingindo a taxa de 30 quadros por segundo para a renderização de cenas de baixa complexidade.

Palavras-chave: Realidade aumentada. Ray tracing. Fotorrealismo. Iluminação global.

ABSTRACT

In recent years we saw the increase and popularization of augmented reality applications. However, in these systems, it is common to perceive a visual discrepancy between real and virtual elements, which leads to a lack of realism, that is one of the reasons that discourage the use of this type of application. Physically-based algorithms, like ray tracing, can generate renderings with a high degree of photorealism and are becoming popular after the recent development of hardware accelerators. Some works in literature combine these two technologies, augmented reality and ray tracing, but rigidly, without modularization, making the solutions dependent on specific frameworks. In this work, we propose a middleware to integrate augmented reality and real-time ray tracing by mapping the camera's coordinates into the ray tracing framework, and performing an image composition process. Besides, the middleware works in a modularized way, allowing to choose between existing libraries and frameworks those that better fulfill the user's needs and expertise. We build an application using the ARRay-Tracing middleware to integrate two ray tracing frameworks, the Optix and the VKRay, to an augmented reality framework, the artoolkitX, obtaining in real-time more realistic reflections and refractions. After the evaluation, we concluded that our middleware enables the modular integration without adding processing overhead to the application, achieving the frame rate of 30 fps for rendering less complex scenes.

Keywords: Augmented reality. Ray tracing. Photorealism. Global illumination.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

API	Application Programming Interface
AR	Augmented Reality
AS	Acceleration Structure
GPU	Graphics Processing Unit
ISMAR	International Symposium on Mixed and Augmented Reality
SBT	Shader Binding Table
SLR	Systematic Literature Review
VR	Virtual Reality

SUMMARY

1 INTRODUCTION

In recent years, the advancement of information and communication technologies has propitiated the popularization and consolidation of Augmented Reality (AR). Either in the academic and commercial areas, the interest in this technology has been increasing. As pointed by Hounsell, Tori & Kirner (2020), this increase can be confirmed by the growing number of publications concerning this technology in leading journals in the related field.

Despite the popularization of augmented reality, some old problems related to this technology persist, such as photorealistic rendering, as pointed out by Azuma et al. (2001). The lack of realism in rendering can cause a substantial negative impact on visualization in several types of applications in which, ideally, virtual objects should be indistinguishable from real objects. The field addressing this problem is renderization in augmented reality, which is also growing recently. Only at International Symposium on Mixed and augmented reality (ISMAR), works on this topic corresponded to 12.5% of the total papers between 2008 and 2017 against 1.9% in the previous decade (KIM et al., 2018). However, there is a small number of works addressing photorealism in AR, unlike photorealism in Virtual Reality (VR), such as the work by Yu et al. (2012) that assesses the impact of visual realism in terms of presence in a virtual environment or the work by Lee et al. (2013) that studies the impact of the rendering quality of the virtual environment when performing simulated tasks within it.

Many of the techniques aiming to increase the quality of the rendered image in AR are based on rasterization and, sometimes, fail to correctly simulate the reflection of the light on specular or transparent surfaces. Physically-based rendering techniques such as ray tracing and path tracing, on the other hand, can produce results with a high level of visual realism for this kind of surface and have been widely used by the film industry for offline rendering (KELLER et al., 2015). Due to its computational cost, only after the development of graphics cards equipped with hardware acceleration and modern ray tracing frameworks, this technology has become viable for real-time rendering and popular in the gaming industry. Nowadays, these frameworks can even generate images in Full HD at 60 fps (KELLER et al., 2019).

The contribution of this work is the proposition of ARRay-Tracing, a middleware to facilitate the integration of augmented reality frameworks and modern ray tracing frameworks to obtain photorealism in AR applications. In this work, it is possible to simulate the real environment through a skybox built using real images as textures. In order to verify the ease of integration, we generate examples of the integration between two ray tracing frameworks, Optix and VKRay, and an AR framework, the artoolkitX.

The middleware maps the camera coordinates of the AR framework to the camera coordinates of the ray tracing framework by extracting these coordinates from the transformation matrix; it also performs the composition of real and virtual images within the alpha blending of OpenGL. In our tests, we used an AR framework based on fiducial markers. However, other AR frameworks can be used, since the type of marker, fiducial or natural, does not change how we extract coordinates from the camera's transformation matrix. The geometric reconstruction of the real environment and the lighting conditions detection of the real environment are not included in the scope of this work and, so far, are not supported by ARRay-Tracing. Thus, in the examples presented in this work, the geometries and light sources were defined arbitrarily.

1.1 PROBLEM DEFINITION

According to the definition by Ferwerda (2003), in Computer Graphics, an image is photorealistic if generated by a computer but indistinguishable from a photograph of the same scene. This definition, however, assumes that a photograph is realistic without supporting this assumption. Still, according to Ferwerda (2003), it is possible to consider the concept of photometrically realistic to define a photorealistic image. Photometry is the measurement of light energy perceived by the human eye. Thus, a photometrically realistic image must produce the same visual response that the original scene would produce in the human eye.

The problem addressed in this work is the lack of photorealism in AR applications. To better understand this problem, initially, we conducted a Systematic Literature Review (SLR), on which one of the research questions was "What is considered photorealism in the context of augmented and mixed reality?"(PEREIRA; JUNIOR; SILVA, 2021). Despite seeming a trivial question, we aimed to verify the existence of a closed definition with minimum requirements about photorealism in mixed and augmented reality contexts.

According to Schwandt & Broll (2016), photorealism is a visual representation where realistic interactions between surfaces and light are perceived, and close and medium-range reflections are essential for a realistic representation of virtual objects in this context. Also, according to Jacobs & Loscos (2006), the represented scene must have a consistent shadow configuration, its virtual objects must look natural, and the illumination of these virtual objects needs to resemble the illumination of the real objects. Conforming to Agusanto et al. (2003), a common illumination for virtual and real objects gives consistent looks between them but also inter-reflections and shadows. Finally, Pessoa et al. (2010) pointed out that the scene is visually compelling, although not necessarily physically correct.

By the descriptions, it is possible to perceive that there is no commonly accepted closed definition of photorealism in the context of mixed and augmented reality. However, some points are common among the descriptions and were perceived as a goal in most of the papers analyzed in the review, such as visual coherence in the interaction between light, real and virtual objects. A manner to increase this coherence is enabling well-looking shadows, reflections, and refractions in the applications.

1.2 OBJECTIVES

The main objective of this work is to facilitate the inclusion of high visual quality shadows, reflection, and refraction in AR applications to increase their photorealism and improve the user experience. To accomplish this, we propose the middleware ARRay-Tracing to integrate modern augmented reality and ray tracing frameworks.

A secondary objective is to carry out this integration modularly to allow the user to choose among the existing frameworks the one that best suits their needs and expertise. In this way, to verify the modularity of the proposed solution, we performed a qualitative and quantitative analysis using the ARRay-Tracing middleware with two different ray tracing frameworks.

1.3 RESEARCH QUESTION

Since AR applications render virtual elements in real-time, it was impossible to integrate ray tracing with AR until recently because the processing was very time-consuming and made real-time unfeasible. However, in recent years, with the development of Graphics Processing Units (GPUs) with greater computational power, mainly since the launch of the Turing architecture[1] by NVIDIA in 2018, new frameworks emerged, allowing the use of real-time ray tracing.

With the possibility of using ray tracing for real-time rendering in several graphic applications, it also became possible to integrate it with AR frameworks. Through ray tracing algorithms, virtual elements could be rendered more photorealistic and similar to real elements. So, this work aims to answer if a middleware would facilitate the integration of ray tracing with AR applications using modern frameworks, like Optix and VKRay.

1.4 RELATED WORK

Over the years, a wide range of techniques has been used to increase the visual realism of augmented reality applications. In this section, we highlight the works that employ this method and are closer to our approach.

Santos et al. (2012) presented a method for interactive illumination of virtual objects under time-varying lighting in augmented reality applications and introduced a graphics rendering pipeline based on a real-time ray tracing paradigm. Using a ray tracer previously developed by some of the authors in association with an augmented reality framework, the proposed pipeline was capable of providing high-quality rendering with real-time interaction between virtual and real objects, such as occlusions, soft shadows, and reflections. The tests presented in this work show that their proposed solution got a maximum value of 74 fps for the simplest scenario, where there was no shadow and using a single primary ray.

[1] <https://www.nvidia.com/pt-br/geforce/turing/>; accessed on 29/08/2021

With the insertion of the shadows, the value obtained for the same scenario decreased to 40 fps.

Kán & Kaufmann (2013) proposed an algorithm based on irradiance caching and Differential Rendering to produce an augmented reality application for realistic interior design. Their algorithm runs at interactive frame rates and produces a high-quality result of diffuse light transport. Also, the proposal combines ray tracing and rasterization to achieve high-quality results, preserving interactivity. The authors used the Optix framework and ray tracing to calculate direct illumination by the camera. The tests presented in the work show that their solution achieved 31 fps.

Schwandt & Broll (2016) presented a solution to get close and medium-range glossy reflections in smooth surfaces for mixed reality applications. They used a combination of Ray Tracing with a partial 3D reconstruction of the environment obtained through an RGB-D camera. They also created an environment map with the RGB image from the same RGB-D camera. As a limitation of the work, they point out that the approach does not support estimating arbitrary light sources. So, they assumed the light as a white area illuminating the scene from the top. Also, there was no material properties estimation for the real objects in the scene. The solution was coded in C++ using a graphics engine. developed in his research group. The OpenGL 4.5 was used as graphics library and the OpenCV for image processing. The final system achieved 10 fps.

Alhakamy & Tuceryan (2019) presented a system that captures the existing physical scene illumination, applying it to the rendering of virtual objects added to a scene for augmented reality applications. With a 360 camera, the system estimates the direct light position. It also estimates the indirect illumination and builds an environment map. Assuming previously known geometry of the real environment, they used GLSL shaders to render visually coherent virtual elements in the scene. Better results were achieved for shadows than for reflections and refractions. The system uses Vuforia as AR framework and Unity for rendering, achieving 60 fps.

Our work proposes the real-time integration between ray tracing and augmented reality through the ARRay-Tracing middleware. Unlike other works in the literature, this integration is performed in a modular way. This way, it allows the use of modern augmented reality and ray tracing frameworks, choosing the ones that best suit the application. In order to test the integration, we created an AR application to visualize scenes with shadows, reflections, and refractions using two ray tracing frameworks, Optix[2] and VKRay[3], combined with the artoolkitX[4] framework. Using the ARRay-Tracing, it was possible to achieve a frame rate of around 30 fps for scenes with less complex geometry and materials. The frame rate decreased to around 6 fps for more complex scenes, either using the middleware or only ray tracing framework, without AR integration.

[2] <https://developer.nvidia.com/optix>; accessed on 29/08/2021

[3] <https://developer.nvidia.com/rtx/raytracing/vkray>; accessed on 29/08/2021

[4] <http://www.artoolkitx.org/>; accessed on 29/08/2021

1.5 OUTLINE

This work is organized as follows: Chapter 2 presents the concepts that support the construction of the middleware. Chapter 3 presents the ARRay-Tracing middleware and workflow, while Chapter 4 presents the results obtained using the ARRay-Tracing. Finally, Chapter 5 and Chapter 6 discuss the experimental results and our concluding remarks, respectively.

2 FUNDAMENTALS

This chapter briefly presents basic concepts about the theoretical foundations involved in ARRay-Tracing middleware to better contextualize our proposal.

2.1 AUGMENTED REALITY

Augmented reality is a technology and methodology that supplements reality by inserting virtual elements into the real environment combining them with real elements (AZUMA et al., 2001). On the Reality-Virtuality Continuum proposed by Milgram & Kishino (1994), it is situated as a part of Mixed Reality in which most of the elements that make up the environment are real elements, as shown by Figure 1. Thus, even with virtual elements, the user still has the most robust perception of experiencing the real environment in which he is inserted, different from Virtual Reality, in which the user experiences an environment predominantly virtual.

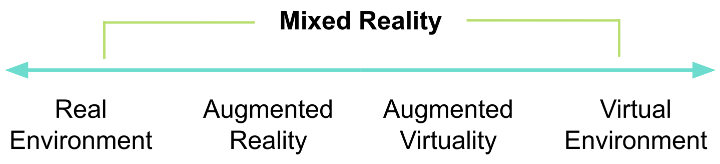


Figure 1 – Reality-Virtuality Continuum.

Source: adapted from Milgram & Kishino (1994).

Augmented reality is not just limited to augment sight. It can also be applied to augment hearing, touching, and smelling. A definition that comprises any AR system, regardless of the sense involved, was presented by Azuma (1997). According to his definition, an AR system combines real and virtual elements in the real environment, runs interactively and in real-time, and registers real and virtual objects to each other in three dimensions. Despite AR being capable of involving several senses, AR systems that augment sight are more common in practice. Additionally, according to Azuma, AR systems can be classified into two large groups in terms of technology for combining real and virtual elements: optical see-through and video see-through.

In optical see-through, the combination of virtual and real images occurs by using an optical combiner placed in front of the user's eyes (AZUMA, 1997). This optical combiner is a display made from a translucent material, so the user sees the real world through it, acquiring the real image. The image combination is made by simply presenting the virtual images overlapped in this optical combiner. The most recent example of this type of technology is Microsoft's Hololens 2[5]. In video see-through, differently, the display is not translucent. The real-world image is obtained through a video camera, then combined with the virtual image by a graphical framework, and, finally, shown in a display in front of the user. When this display is a monitor, the system is classified as a monitor-based system. Also in the survey, Azuma (1997) pointed out that the most basic way to combine real and virtual elements in monitor-based systems was to remove the background from the virtual image and superimpose this image on the image captured by the camera. This image composition technique is still used today in many leading augmented reality frameworks and was considered a starting point for our work.

2.2 TRACKING SYSTEMS

An AR system must ensure correct registration between real and virtual elements. That is, they must be correctly aligned with each other, with spatial coherence about position and orientation, so that in the final combined image, the real and virtual elements appear to coexist in the same environment (AZUMA, 1997). Tracking methods are used to enable correct registration. In tracking is defined a reference system in the real world from which virtual objects will be aligned. Also, it is possible to obtain by this reference the position and orientation from which the observer views the scene (OLIVEIRA, 2018).

According to Billinghurst, Clark & Lee (2015), different methods can be used for tracking, and they can be grouped into magnetic tracking, vision based tracking, inertial tracking, GPS tracking, and hybrid tracking. Due to the low hardware requirements demanded, combined with the advancement of the computing power of mobile devices such as modern smartphones that already include good cameras and screens, vision based tracking has become popular in recent years. This type of tracking uses data captured by an optical sensor and, according to the sensor used, the methods are classified into three groups: infrared sensors, visible light sensors, and 3D structure sensors. Considering that good quality cameras are quite affordable today, including those integrated into smartphones, tablets, and notebooks, visible light sensors are more commonly used. Furthermore, the same camera can be used to acquire the real world image for composition and as an optical sensor for tracking.

This work employs visual based tracking using a webcam as an optical sensor. More specifically, fiducial tracking is used. The tracking target inserted in the scene is the fiducial marker, constituted of rigid geometric shapes arranged in a pattern known by the software and with a unique identifier. The main advantage of fiducial markers is their low cost, as they can be printed on paper or even displayed on a secondary screen, and their recognition system only requires a standard camera adequately calibrated (ZORZAL; SILVA, 2020).

[5] <https://www.microsoft.com/en-us/hololens>; accessed on 29/08/2021

2.3 IMAGE COMPOSITING

Digital image compositing is a method of combining two or more source images into a final image. This combination is made by evaluating the contribution of each pixel from each original image. Let c_A, c_B, and c_O be the color components of images A, B, and the *output* of the composition. The Equation 2.1 gives the composition of images A and B, where F_A and F_B are the fractions of images A and B that prevail in the final composition, whose value is given according to the binary operation performed between the two source images. For the B *over* A operation, for example, F_A equals $1 - \alpha B$, and F_B equals 1 (PORTER; DUFF, 1984).

$$c_O = c_A F_A + c_B F_B \tag{2.1}$$

When blending two source images, it's more useful to be able to select an object or subregion from image A and a subregion from image B to produce the output composite. To distinguish an arbitrary subregion in an image, we can vary the weighting function for each pixel in the image. As pointed by McReynolds & Blythe (2005), the OpenGL provides this weighted per-pixel merge capability through the framebuffer blending operation. In framebuffer blending the weights are stored as part of the images themselves. Since the alpha values are most often used as the weights than R, G or B values, framebuffer blending is often called alpha-blending. In OpenGL, to composite a geometry against a background image, we can load the background image into the framebuffer, then render the geometry on top of it with blending enabled. So, we set the source and destination blend factors to GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA, respectively, and assign 1.0 for the α of the pixels corresponding to the geometry when they are sent to the OpenGL pipeline.

2.4 RAY TRACING

According to Glassner, ray tracing is an image synthesis technique whereby a 2D image of a 3D world is created (GLASSNER, 1989). Unlike rasterization, which is an object-order technique where for each object in the world, the rasterization finds and updates all the pixels influenced by it, and draws them accordingly, ray tracing is image-order rendering. Thus, a ray tracing algorithm performs an iteration over the pixels and, for each pixel, finds all the objects that influence it and computes their value.

The origin of the technique goes back to the proposal presented by Appel (1968), who introduced the shooting of rays in a scene to calculate pixel colors, a process known as ray casting. However, the work presented by Whitted (1979) was responsible for further popularizing the technique by introducing the ray tracing algorithm. The ray tracing

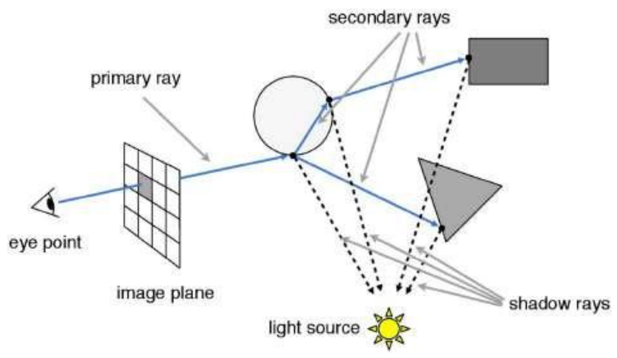


Figure 2 – Recursive ray tracing schema.

Source: Computer Graphics and Imaging course of UC Berkeley.[6]

algorithm added recursion to the ray cast process, allowing to cast secondary rays to treat shadows, reflections, and refractions to obtain a more accurate pixel color.

Despite having a simple logic, ray tracing algorithms have a high computational cost, which is why their use in real-time applications was impractical for many years. As pointed by Marschner & Shirley (2018), a basic ray tracing algorithm has three main parts: (i) ray generation, where the observer's point of view is defined, and from which the ray will be shot towards the pixels in the scene; (ii) the computation of the ray intersections, through which the first geometry of the scene reached by the ray is obtained and secondary rays shooting; and (iii) shading, where a lighting model is used to compute the pixel color based on the result of the intersections. Algorithm 1, adapted from Hughes et al. (2014), presents the basic structure of a simple ray tracer algorithm, and Figure 2 illustrates a ray traversal.

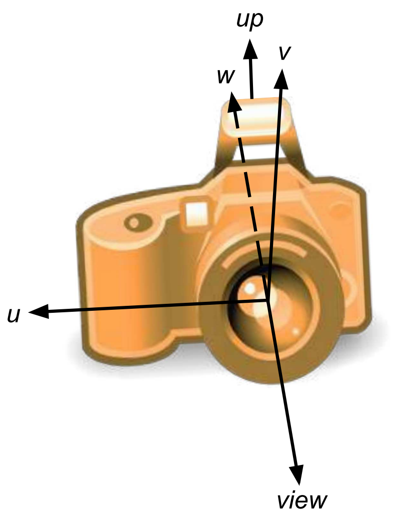


Figure 3 – Camera coordinates.

The ray's origin is calculated using camera coordinates. In computer graphics, we

[6] <https://cs184.eecs.berkeley.edu/sp18/lecture/ray-intro/slide_015>; accessed on 22/11/2021

Algorithm 1 Recursive ray tracing (HUGHES et al., 2014)

```
for each pixel ($x$, $y$) do
    $R$ = ray from eyepoint $E$ through pixel
    image[$x$, $y$] = raytrace($R$)
end for
procedure RAYTRACE($R$)
    $P$ = raycast($R$)
    return lightFrom($P$, $R$)
end procedure
procedure LIGHTFROM($P$, $R$)
    $color$ = emitted light from $P$ in direction opposite $R$
    for each light source $S$ do
        if $S$ is visible from $P$ then
            $contribution$ = light from $S$ scattered at $P$ in direction opposite $R$
            $color$ += $contribution$
        end if
    end for
    if scattering at $P$ is specular then
        $R_{new}$ = reflected or refracted ray at $P$
        $color$ += raytrace($R_{new}$)
    end if
    return $color$
end procedure
```

can define a camera from three coordinates: (i) the eye, which determines the location where the camera is positioned; (ii) the view, or look at, direction in which it points; and (iii) the up, which determines the direction that points upwards from the camera. Let $\vec{t}$ be the vector in the up direction, perpendicular to $\vec{w}$, and $-\vec{w}$ be the vector in the view direction. We obtain the coordinates of the origin from which the rays are shooted using the Equations 2.2 and 2.3, as shown in Figure 3.

$$\vec{u} = -\vec{w} \times \vec{t} \tag{2.2}$$

$$\vec{v} = -\vec{w} \times \vec{u} \tag{2.3}$$

2.5 RAY TRACING FRAMEWORKS WORKFLOW

As presented by Usher (2021), modern ray tracing Application Programming Interfaces (APIs) like Optix and VKRay implement the three steps of a basic ray tracer using a Shader Binding Table (SBT) structure, a set of shader functions handlers, and the respective parameters of those functions. The execution of a function in the SBT is determined by the existence or not of an intersection between a ray and a geometry, besides the knowledge of the intersected geometry. There is also a set of parameters defined in the application that contribute to determining the function. Figure 4 presents the pipeline of ray tracing by using the SBT.

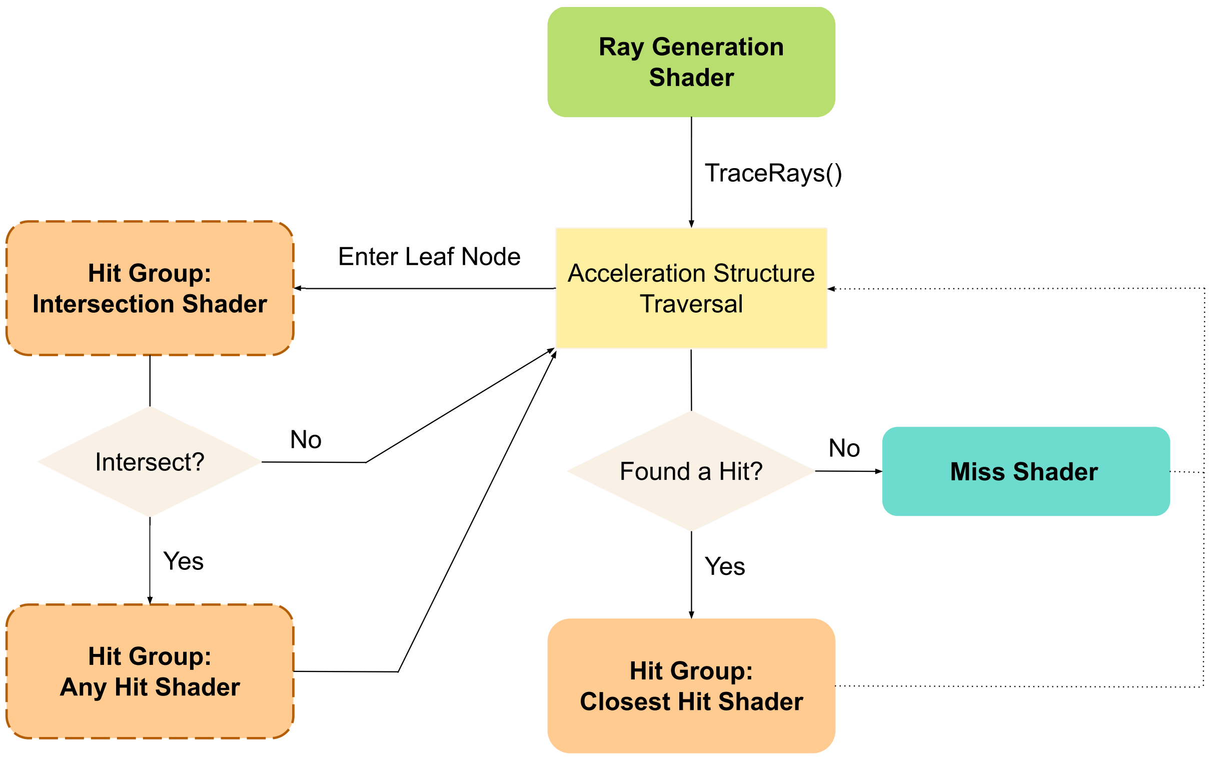

Ray Generation Shader
TraceRays()
Hit Group: Intersection Shader
Enter Leaf Node
Acceleration Structure Traversal
Intersect?
No
Yes
Found a Hit?
No
Miss Shader
Yes
Hit Group: Any Hit Shader
Hit Group: Closest Hit Shader

Figure 4 – Workflow of the shaders in the SBT during a ray traversal.

Source: adapted from Usher (2021).

The idea behind the SBT usage is that it is not known which object the ray will hit at the beginning of a ray traversal. So, it is necessary to have access to the entire scene, usually encapsulated in an Acceleration Structure (AS), and to all shaders during the rendering process, since any of them can be invoked. That is, there is not previously bind between geometry and a set of shaders like in rasterization.

The shader records are divided into three groups: (i) generation, (ii) hit group, and (iii) miss.

(i) generation: contains the shader function responsible for generating and tracing the rays into the scene. It works as an entry point in the rendering process. As long it generates the ray, it has information about the camera's coordinates. Generally, this shader is also responsible for writing the final result of the computation into the frame buffer;

(ii) hit group: contains three shaders functions, (ii.a) intersection, (ii.b) any hit, and (ii.c) closest hit. These shaders are used together to process the interaction between ray and geometry and share the same parameters;

- (ii.a) the intersection is optional since some kinds of intersections, like ray-triangle, are verified in hardware. It is used to compute intersections with custom geometry;
- (ii.b) also optional, any hit is used whenever an intersection is found. It is commonly used to filtering intersections using cutout textures;
- (ii.c) the closest hit is called when a ray traversal ends, and an intersection was found during the traversal. It can trace additional rays or perform the shading by computing the pixel's color based on an illumination model and returning it to its caller;

(iii) miss: contains the shader function called when no intersections are found toward a ray traversal. It can be used to attribute a solid color to the scene's background pixels or identify unoccluded rays in some applications.

The ray generation shader initially computes the ray origin and direction and calls the TraceRays function, which performs the traversal through the AS. Following, the hit group takes care of the intersections. The intersection is verified by testing the ray against the primitives in AS's leaf nodes. When using non-triangle primitives, this test is performed by the intersection shader. If an intersection is found during traversal and the any-hit shader is enabled, it is called. When the ray traversal ends, if any intersection was found, the closest-hit shader is called. At this point, closest-hit can compute the shading and return to TraceRays or trace additional rays. In case of no intersection has been found, the miss shader is called and returns to TraceRays.

3 THE ARRAY-TRACING MIDDLEWARE

We proposed the ARRay-Tracing as a middleware to integrate AR frameworks with ray tracing frameworks in a modularized way. That is, it allows users to choose which framework to use, according to their demand. This modularity is possible based on basic premises of AR frameworks and ray tracing rendering. The first, the premise that every AR system and every ray tracing system define a virtual camera with the same set of coordinates, and so, it is possible to map the coordinates of the AR framework camera to the ray tracer camera coordinates. And second, the premise that a video-based AR framework uses an image overlay to combine virtual and real elements in a scene. Therefore, it is possible to combine the final ray tracer image with the video frame obtained by the physical camera using the same technique.

The ARRay-Tracing module is responsible for two central operations: mapping the camera coordinates from the augmented reality framework to the ray tracing framework, and the image composition. The geometric reconstruction of the real environment and the lighting conditions detection of the real environment are not included in the scope of this work and, so far, are not supported by ARRay-Tracing. Figure 5 shows the middleware workflow.

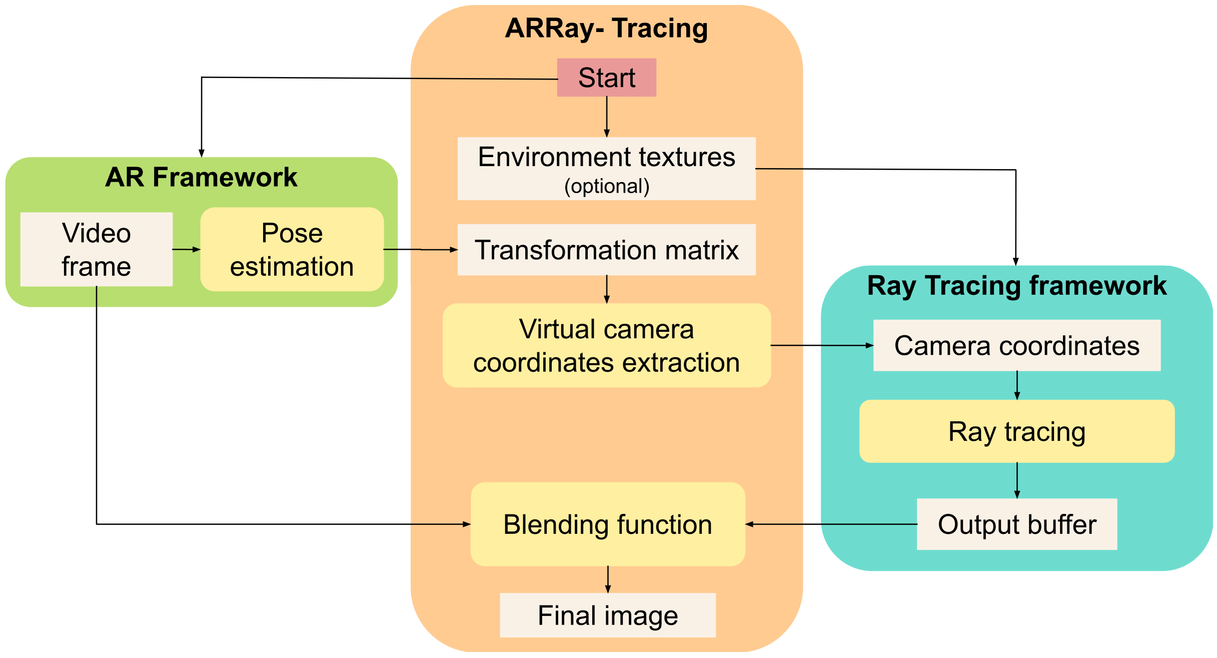


Figure 5 – The ARRay-Tracing workflow.

In ARRay-Tracing middleware, the first step to integrate the two frameworks is mapping the AR camera to the ray tracing camera. Fiducial marker-based augmented reality frameworks estimate the real camera position and orientation (pose), processing the video frames to get the marker position and orientation. Frameworks based on the

ARToolkit, like the artoolkitX used in this work, give us the camera pose, in the marker's coordinate system, by the inversion of the marker's transformation matrix, resulting in the camera pose matrix. In this work, we used an AR framework based on fiducial markers. However, other AR frameworks can be used, as the type of marker, fiducial or natural, does not change how we extract coordinates from the camera's transformation matrix. Let MT be a 4×4 homogeneous matrix of the marker transformation obtained by the AR framework in the marker's coordinate system, in a format like shown by Equation 3.1.

$$MT = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \tag{3.1}$$

Sometimes, the coordinate system of the marker may not be the same as the world coordinate system in ray tracing frameworks. Figure 6 shows the marker coordinate system and the world coordinate systems for the two ray tracing frameworks covered by this work. It is possible to observe that the marker coordinate system from AR framework is the same used in the world of the Optix framework but differs from VKRay framework.

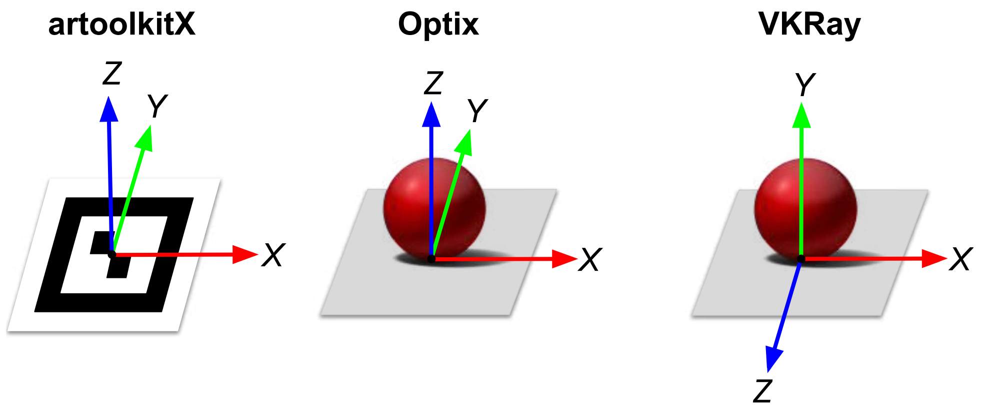
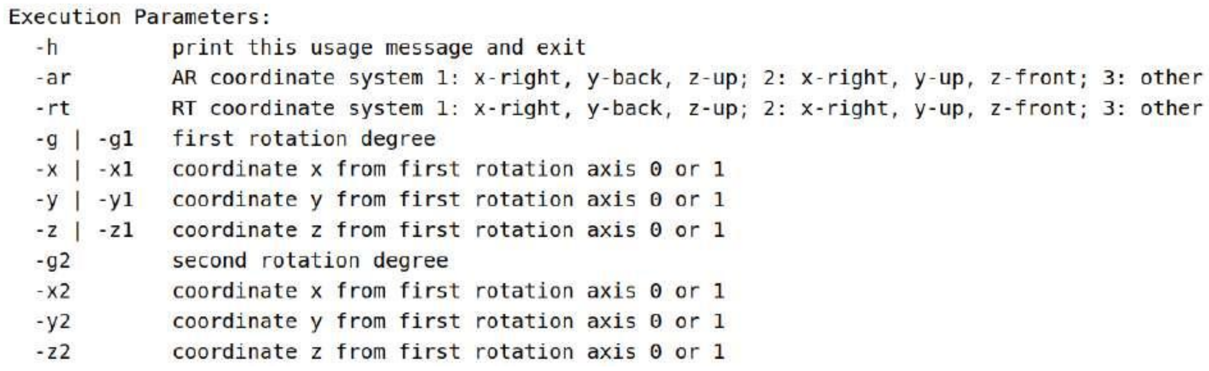


Figure 6 – The coordinate systems of the marker in artoolkitX framework, and of the world in Optix and VKRay frameworks.

As the position and orientation of the camera are defined in the ray tracing frameworks considering their world coordinates, if the coordinate systems are different, we first convert the MT matrix to the ray tracing framework coordinate system and then invert this matrix to get the camera pose matrix CP. To accomplish this, we proposed the use of predefined profiles and rotation parameters, considering that with two rotations, at maximum, the coordinate systems can be matched. When running the code, the user has the option to pass as parameter two profiles ids, one for AR framework and one for ray tracing framework; the first rotation degree followed by the coordinates of rotation axis; and, the second rotation degree followed by the coordinates of second rotation axis. Thus, when the user informs the rotation parameters, the middleware calculates the rotation matrix by which the camera pose matrix CP is multiplied. Figure 7 shows the usage help message with the explanation about the parameters. When using Optix, no conversion is required. When using the VKRay, we apply a rotation of 90 deg around the x-axis passing the respective parameters *"-ar 1 -rt 2 -g 90 -x 1 -y 0 -z 0"* in the execution.



```
Execution Parameters:
  -h          print this usage message and exit
  -ar         AR coordinate system 1: x-right, y-back, z-up; 2: x-right, y-up, z-front; 3: other
  -rt         RT coordinate system 1: x-right, y-back, z-up; 2: x-right, y-up, z-front; 3: other
  -g | -g1    first rotation degree
  -x | -x1    coordinate x from first rotation axis 0 or 1
  -y | -y1    coordinate y from first rotation axis 0 or 1
  -z | -z1    coordinate z from first rotation axis 0 or 1
  -g2         second rotation degree
  -x2         coordinate x from first rotation axis 0 or 1
  -y2         coordinate y from first rotation axis 0 or 1
  -z2         coordinate z from first rotation axis 0 or 1
```

Figure 7 – The usage help message explaining the profile's parameters.

After obtaining the MT matrix in the ray tracing world coordinate system, inverting this matrix we get the camera pose matrix CP as shown by the Equation 3.2.

$$MT^{-1} = CP = \begin{bmatrix} c_0 & c_4 & c_8 & c_{12} \\ c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \end{bmatrix} \tag{3.2}$$

From CP matrix, the ARRay-Tracing extracts the camera coordinates (eye, look at, and up) from AR framework, according to Equations 3.3, 3.4, and 3.5.

$$eye = (c_{12}, c_{13}, c_{14}) \tag{3.3}$$

$$look\_at = (c_{12}, c_{13}, c_{14}) - (c_8, c_9, c_{10}) \tag{3.4}$$

$$up = (c_4, c_5, c_6) \tag{3.5}$$

Ray tracing frameworks define their cameras using these same three coordinates eye, look at and up. Therefore, ARRay-Tracing assigns to these three variables the values extracted from CP matrix. Into the ray tracing framework, this values are passed to ray generation shader, which uses them to calculate the rays origin conforming Figures 3 and 2. Then, the ray traversal is performed and generates the output buffer. This process occurs following the workflow presented in Section 2.5. The ray traversal is computed and the ray generation shader writes the pixels color result in the output buffer according to the returned values by closest hit and miss shader. In this step, we assign zero to the alpha component of all pixels that did not intersect any ray, that is, we assign 0 to the fourth component of pixel color in the miss shader. This way, we get a transparent image in regions where there are no models to render.

The synchronization between the frameworks is done using a semaphore since the execution takes place in the same thread, and the framebuffer is accessed to render the image captured by the AR framework and to render the image generated by the ray tracing framework. The ray tracing framework only starts the ray tracing after the assignment of camera coordinate values, and ARRay-Tracing obtains these values after the AR framework captures and processes a video frame. Finally, the middleware only performs the image composition after the complete rendering of the framebuffer by the ray tracer.

Finally, ARRay-Tracing combines the video frame with the output image of the ray tracer through the OpenGL's alpha blending using as blend factors GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA, generating the final image to be rendered on the screen. In this process, a final image is composed by two source images, A and B, where A is the real-world image captured by the camera, and B is the virtual image generated by a ray tracing framework. This composition uses the B *over* A operation, as the Equation 3.6, where c_A, c_B, and c_O are the color components of images A, B, and the *output*, and $[i, j]$ represents each image's pixels.

$$c_O[i, j] = c_A[i, j] * (1 - \alpha B) + c_B[i, j] * 1 \tag{3.6}$$

The middleware allows modularity since any frameworks that fit the following restrictions can be used: in the AR framework, it is possible to extract the camera coordinates (eye, look at, and up) and the video frame; in the ray tracing framework, it is possible to get the output buffer with an alpha value equal to zero for pixels that do not correspond to any geometry present in the scene.

3.1 TIME COMPLEXITY

The AR frameworks based on ARToolkit, like artoolkitX, execute a detection algorithm having a complexity order of $O(M)$, where M is the number of markers being tracked (GOMES et al., 2007). In our test scenario, we used only one marker.

A naive ray tracing implementation has a complexity order of $O(WHN)$, where W is the width and H is the height in pixels of the rendered image, and N is the number of objects, or polygons in a scene. Modern ray tracing frameworks such as Optix and VKRay in turn use structures of type Bounding Volume Hierarchies to optimize the interceptions calculation. In this approach, instead of enclosing each object in a box, a group of objects, sufficiently close to each other, are bounded by a single box. So, the complexity order is $O(WHM)$, where $M < N$ is the number of boxes (MARTINEZ, 2006).

The ARRay-Tracing middleware integrates an AR framework and a ray tracing framework mapping the camera coordinates and performing an image composition. The camera mapping is an operation where the coordinates are extracted from the camera pose matrix according to Equations 3.3, 3.4, and 3.5, and assigned to their corresponding variables in ray tracing framework. So, this operation has a complexity order of $O(1)$. The image composition consists of, for each pixel of the image, performing a linear combination of the values of two source pixels. So, this operation has a complexity order of $O(WH)$, where W is the width and H is the height in pixels of the rendered image. Therefore, the ARRay-Tracing middleware has the complexity order of the image composition, that is, $O(WH)$. Thus, the AR framework and ray tracing framework have dominance in complexity order.

4 RESULTS

The tests we performed aimed to make two primary checks: (i) the proposed solution allows modularity, and (ii) the middleware does not add significant processing overhead to the application. To check the first item, we created two versions of a simple 3D model visualization application using different frameworks. Both tests use artoolkitX as an AR framework, and for the ray tracing framework, the first uses the Optix, and the second uses the VKRay.

We chose artoolkitX as the AR framework because it is a continuation of the classic ARToolkit, and which works very well with fiducial markers. Optix was identified as a possible ray tracing framework for use in our application through a review of photorealism trends in mixed reality (PEREIRA; JUNIOR; SILVA, 2021). Like artoolkitX, it works with the OpenGL graphical API[7], which would facilitate the integration process. The VKRay, in turn, works with the Vulkan graphical API[8]. However, it is possible to establish interoperability between this framework and the OpenGL by using specific extensions. Therefore, we chose to switch between two modern real-time ray tracing frameworks to test the modularity of the application.

The implementation is similar for both versions. We mapped the camera coordinates extracted via the artoolkitX to the camera's coordinates of the Optix and the VKRay. We then performed ray tracing on each of the tools according to the examples available in each SDK. For the final composition of the images, we used the OpenGL's alpha blending in both versions because the artoolkitX and the Optix render the final image through it and, with the interoperability extension, we were able to create a texture in the OpenGL from the VKRay's output buffer.

We performed the tests on a desktop with an AMD Ryzen 5 3600 processor; 16GB DDDR4 3600Mhz RAM; GTX 1060 6Gb, graphic driver version 455.46.04; Microsoft LifeCam Cinema webcam, and Ubuntu 18.04.5 LTS operating system.

4.1 QUALITATIVE ANALYSIS

For the tests, we selected some models from the Stanford repository to be visualized with different materials: Stanford Bunny, Lucy, Dragon, and Happy Buddha[9]. We also used the Utah Teapot model, and built a scene using cuboids and a sphere. Table 1 shows the number of faces of the models used.. We only used materials and shaders available in the examples and tutorials of the two SDKs, therefore we did not compare the visual results obtained in the two versions of the application since the visual results are different

[7] <https://www.opengl.org/>; accessed on 29/08/2021
[8] <https://www.vulkan.org/>; accessed on 29/08/2021
[9] <http://graphics.stanford.edu/data/3Dscanrep/>; accessed on 29/08/2021

for each framework. We create repositories on GitHub containing the source code for generating the samples with Optix[10] and VKRay[11].

Table 1 – Face counting of the models used.

Model	**Faces**
Bunny	4.968
Utah Teapot	126.048
Lucy	448.880
Dragon	871.306
Happy Buddha	1.087.451

The complexity of rendered scenes depends on the computational cost of shading for each material type and is proportional to the number of pixels the geometry occupies in the render window. Specular materials have a low computational cost in shading, while reflective or refractive materials have a higher computational cost. We initially rendered less complex scenes in both frameworks to assess whether a satisfactory frame rate for real-time, around 30 fps, was achieved. Later, only for the Optix framework, scenes of greater complexity were generated to evaluate the interaction between real and virtual elements, without the obligation to obtain a frame rate greater than or equal to 30 fps.

The examples using VKRay were created by modifying the *vk_ray_tracing__advance_NV* example shown in Figure 8. This example includes diffuse material used for the floor, specular material used for bisons, and reflective material, used for the mirror plane. However, the translucent material used in the sphere results in a dotted surface. Therefore it was not used in our examples. When run on our hardware, the original example achieves a very high frame rate, around 2000 fps, when the camera is stationary. However, when any camera movement is performed, whether zooming, rotating, or translating, this value drops drastically, reaching 20 fps depending on the geometry and materials viewed via the viewport, which is expected as depending on the viewport the number of intersections being calculated increases.

The initial scenes, shown in Figures 9, 10, 12, and 13, were rendered in an 800×600 pixels window. In the closest hit shader, the alpha value of 1.0 was assigned to all pixels that had some intersection and matched some geometry. Therefore, during the image composition step, the framebuffer generated by the ray tracer had only two alpha values, equal to 1.0, for the pixels with the virtual elements' geometry, and an alpha equal to 0, assigned in the miss shader, for the pixels that did not correspond to any geometry.

Figures 9 and 10 show four scenes rendered using VKRay. In all VKRay scenes, lighting is provided by a point light source with varying intensity and white color. In Figure 9a,

[10] <https://github.com/lidianetpereira/ARRay-Tracing-OptiX-Samples.git>; accessed on 30/11/2021

[11] <https://github.com/lidianetpereira/ARRay-Tracing-VKRay-Samples.git>; accessed on 30/11/2021

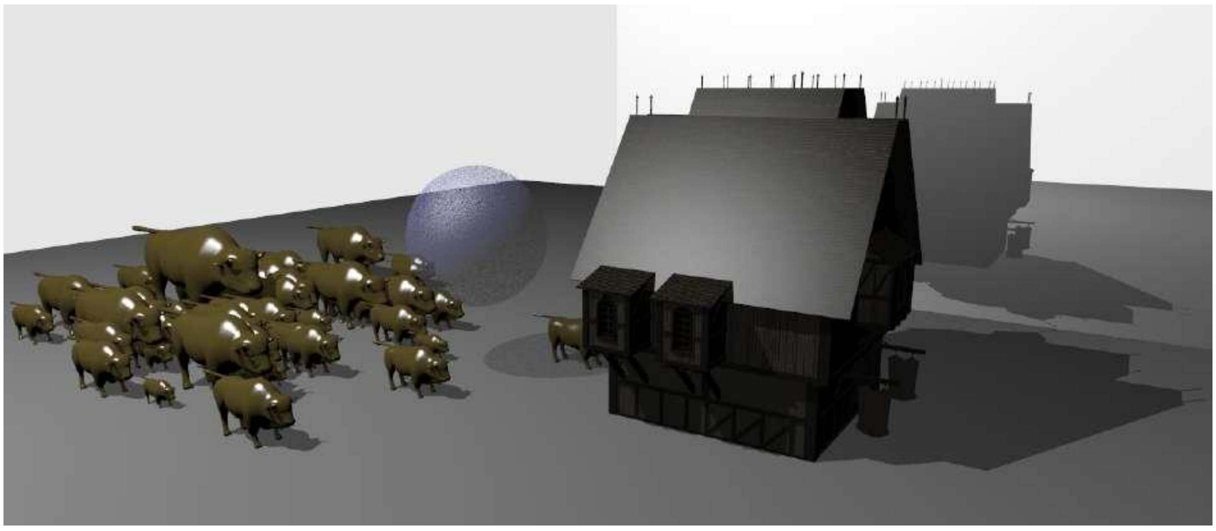


Figure 8 – The original *vk_ray_tracing__advance_NV* that was modified to build the examples using VKRay.

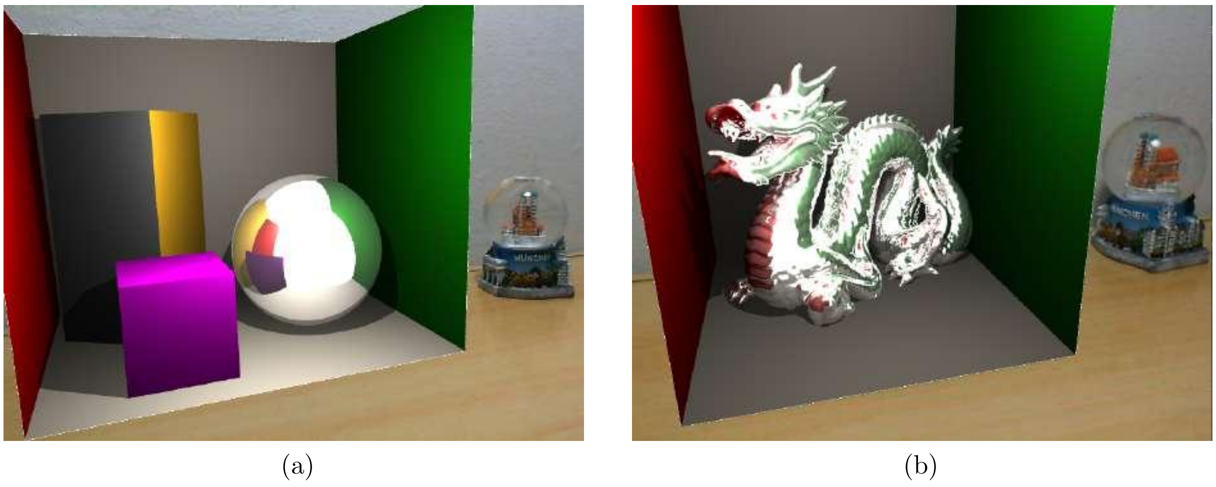


Figure 9 – Cuboids and sphere, and Dragon models rendered using the ARRay-Tracing with the VKRay framework.

the light is translated 100 mm in Y axis and 80 mm in Z axis in relation to the model and its intensity is 10000. While in Figure 9b, the light is translated 120 mm in Y axis and 60 mm in Z axis in relation to the model and its intensity is 6000. For the scene in Figures 9a and 9b, the diffuse material was used for cuboids and the surroundings, which simulates a Cornell box, while the reflective material was used for the sphere and the Dragon model. In the sphere, it is possible to see the reflection of the surrounding elements and the reflection of a white area that corresponds to the background color of the empty part of the world. On the Dragon model, we can also observe the reflection of the surrounding surfaces and the white area corresponding to the empty part of the world, just like in the sphere from Figure 9a. It is important to emphasize that the Cornell box was inserted in the scenes so that the reflection of the surrounding elements could be perceived, since in these examples

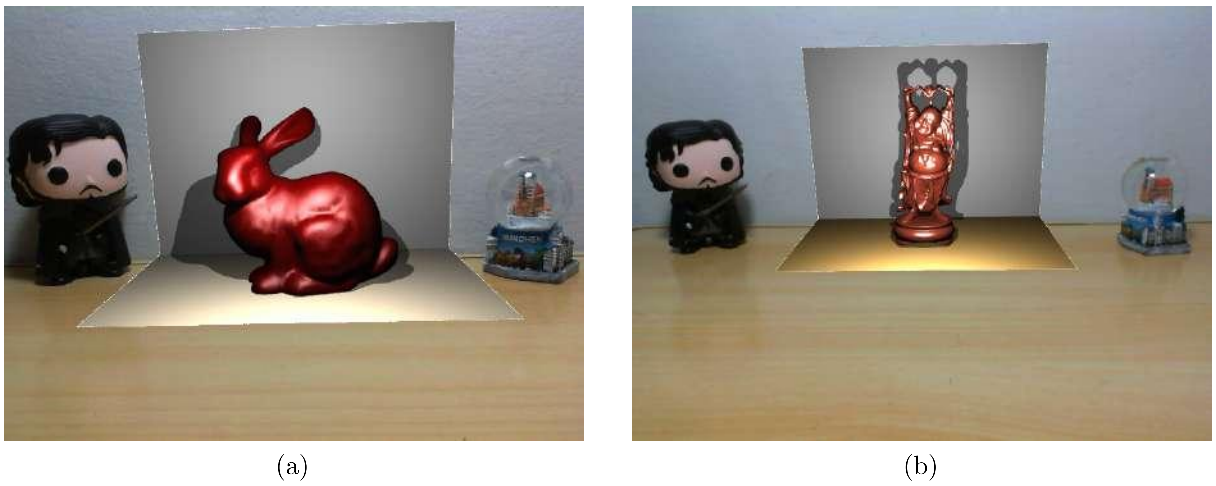


Figure 10 – Bunny and Happy Buddah models rendered using a specular material. the models were rendered using the ARRay-Tracing with the VKRay framework.

there is no interaction between real and virtual elements. Without the surrounding planes, the sphere and the Dragon would only show the reflection of a large white area.

For the scene in Figure 10, we placed two planes to simulate the geometry of the real environment and enable the visualization of the shadows. In both planes, diffuse material was used. For the Bunny and Happy Buddha model, specular materials were used. In Figure 10a, the light is translated 100 mm on Y axis and 90 mm on Z axis in relation to the model and its intensity is 8000. While in Figure 10b, the light is translated 100 mm on Y axis and 100 mm on Z axis in relation to the model and its intensity is 5000. Considering that in VKRay, the point light was positioned to approximate the real lighting position, in 10a, it is possible to notice that shadows were cast in similar directions for the Bunny and the Jon Snow toy figure.

The examples using Optix were created based on two examples from the SDK. Diffuse, specular, and reflective materials were extracted from the *optixWhitted* example shown in Figure 11a. The solid glass material, in turn, was extracted from the *optixTutorial* example shown in Figure 11b. In the used hardware, both examples achieve a frame rate around 60 fps when the camera is stationary. For the *optixTutorial*, this value remains the same even when the camera is moving. Differently, in *optixWhitted* the frame rate drops during camera movements, reaching up to 30 fps depending on the geometry covered by the viewport.

Figures 12, and 13 show four scenes rendered using Optix. All four scenes are illuminated by a white point light of variable intensity, and ambient light with 0.2 of intensity. In Figure 12a, a diffuse material was used in the planes forming the Cornell box and also in the cuboids. The sphere was rendered using the reflexive material, and it is possible to see reflected on its surface the surrounding elements, including the empty world in white. In Figure 12b, the Bunny model was rendered using a specular material. The planes

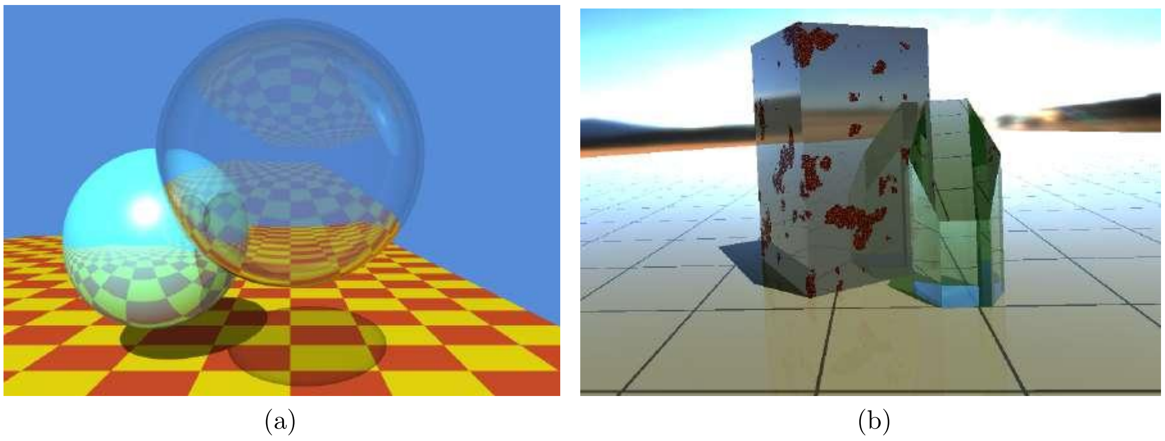


Figure 11 – The original examples from the Optix SDK. In (a) the *optixWhitted* and in (b) the *optixTutorial*.

Constants	**Dragon**	**Lucy**
cutoff_color	0.34f, 0.55f, 0.85f	0.55f, 0.55f, 0.55f
fresnel_exponent	1.0f	0.9f
refraction_index	1.4f	1.5f
refraction_maxdepth	50	20
reflection_maxdepth	50	20

Table 2 – Different constants in glass material for Dragon and Lucy models in Figure 13.

replicating the real wall and table were rendered using a diffuse material. In Figure 12a, the light is translated -80 mm on Y axis and 100 mm on Z axis in relation to the model and its intensity is 0.8. While in Figure 12b, the light is translated -90 mm on Y axis and 100 mm on Z axis in relation to the model and its intensity is 0.5.

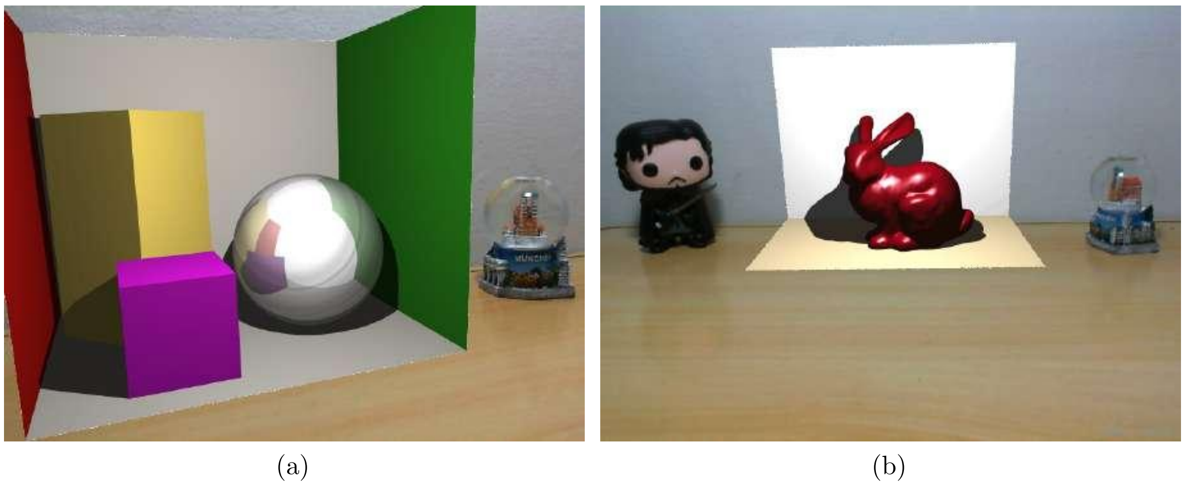


Figure 12 – Cuboids and sphere, and Bunny models rendered using the ARRay-Tracing with the Optix framework.

The Dragon and the Lucy in Figure 13 were rendered using solid glass materials. The differences between their materials are the constant values shown in Table 2. In Figure 13a, the light is translated -60 mm on Y axis and 120 mm on Z axis in relation to the model and its intensity is 0.6. While in Figure 13b, the light is translated -90 mm on Y axis and 100 mm on Z axis in relation to the model and its intensity is 0.5. The Dragon is a denser model and consequently, it is not possible to visualize the refraction in most of its body. It can only be seen in the less dense parts, such as the horns. The reflection, in turn, is noticeable throughout the entire length of the model. Lucy, in turn, is a less dense model facilitating the visualization of refraction, especially in the part of the wings. As it is a less dense model using solid glass material, it can also be seen that the shadow cast by the model is lighter than the shadows cast by solid material models, such as the Bunny and the cuboids in Figure 12. In Figures 12a and 13a, the Cornell box was inserted in the scenes so that the reflection of the surrounding elements could be perceived. Without the surrounding planes, the sphere and the Dragon would only show the reflection of a large white area. And in Figures 12b and 13b the planes were inserted to simulate the real geometry and enable visualization of shadows. Also, in Figure 13b, the background plane is important to enable the visualization of refraction in Lucy's wings.

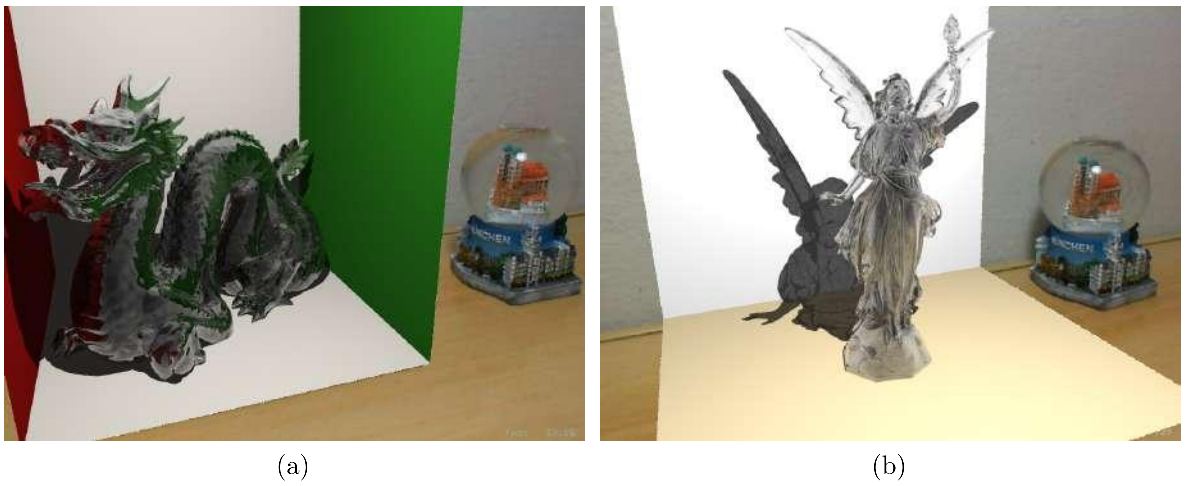

(a) (b)

Figure 13 – Dragon and Lucy models rendered using materials which simulates glass. Both models were rendered using the ARRay-Tracing with the Optix framework.

When we integrate ray tracing into an augmented reality application, two types of materials become a challenge, reflective and refractive. In order to obtain a photorealistic visualization on the surfaces of objects made of reflective materials, we must be able to see the reflection of real objects around them. For refractive objects, we must perceive the elements behind them through their surface.

In order to be able to perceive the interaction between real and virtual elements, it is necessary to replicate, at some level, the real environment in the virtual environment. In this work, we proposed to insert a skybox in the virtual world whose surfaces were textured with images of the real environment. Thus, the proposed solution applies to a

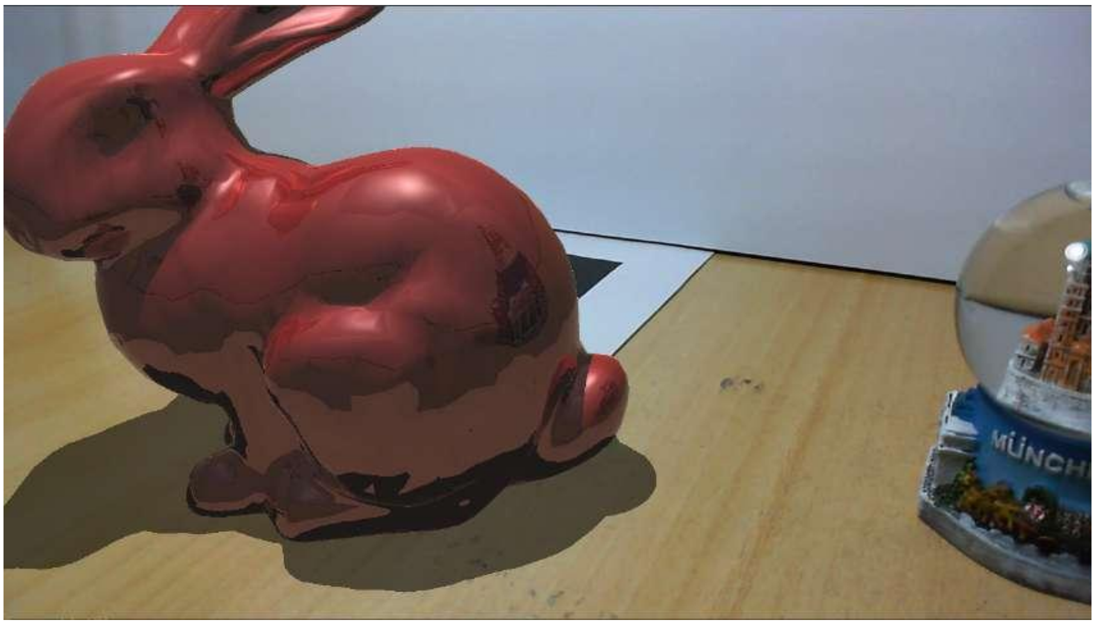


Figure 14 – Bunny model rendered with a reflexive material. It is possible to see the snow globe reflected in it’s surface.

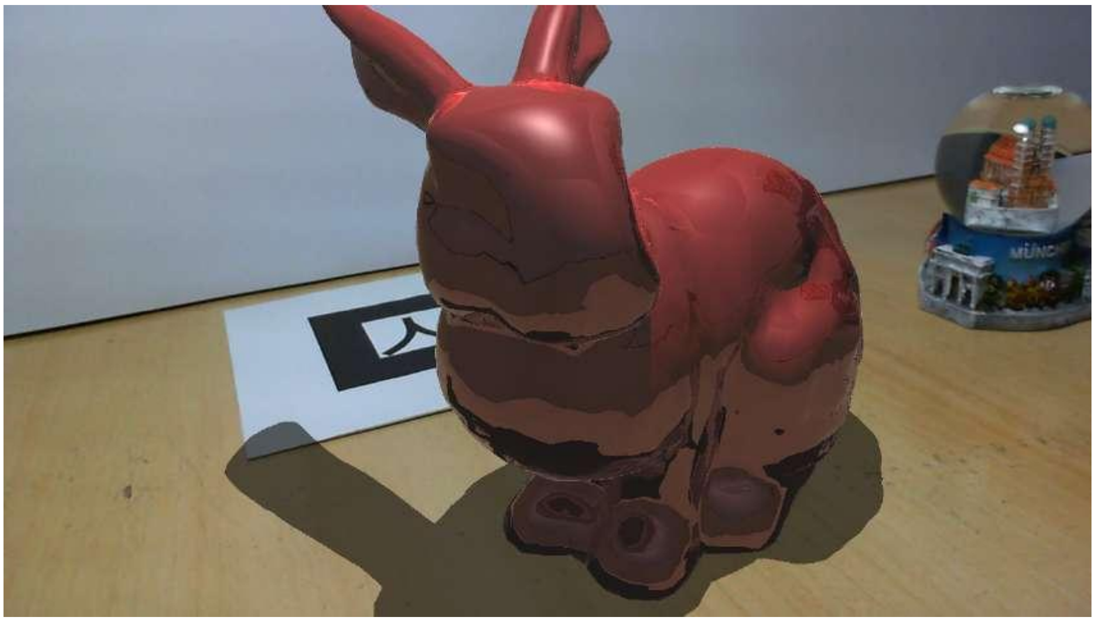


Figure 15 – Bunny model seen by another point of view.

controlled environment, where the positioning of the real elements and the images of the environment are previously known.

Using the Optix framework, we applied this solution to render the five scenes shown by Figures 14, 15, 17, 18, 19, e 20. To obtain only the virtual geometry of interest in the final image, we used the any hit shader to check if the pixel represented a shadow region on a skybox’s area during the ray traversal. If so, we assign an alpha value between 0 and 1.0 to that pixel. For the remaining pixels of the skybox, corresponding to a region not being shadowed by other geometry, we assigned a value of 0 to alpha in the closest hit

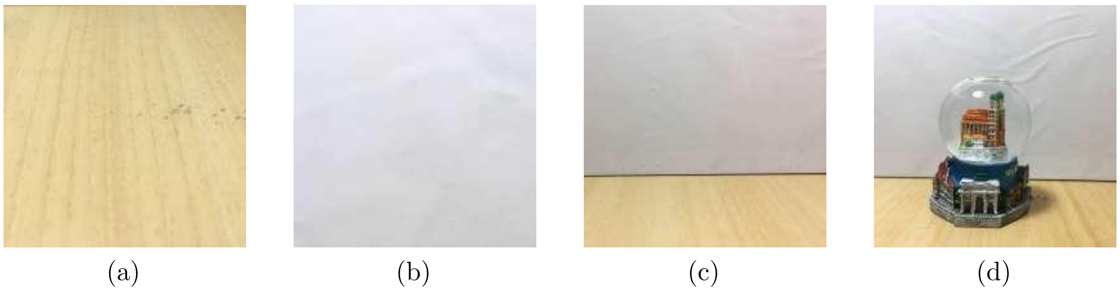


Figure 16 – Textures of floor, ceiling, empty side walls, and snow globe in front a wall, respectively.

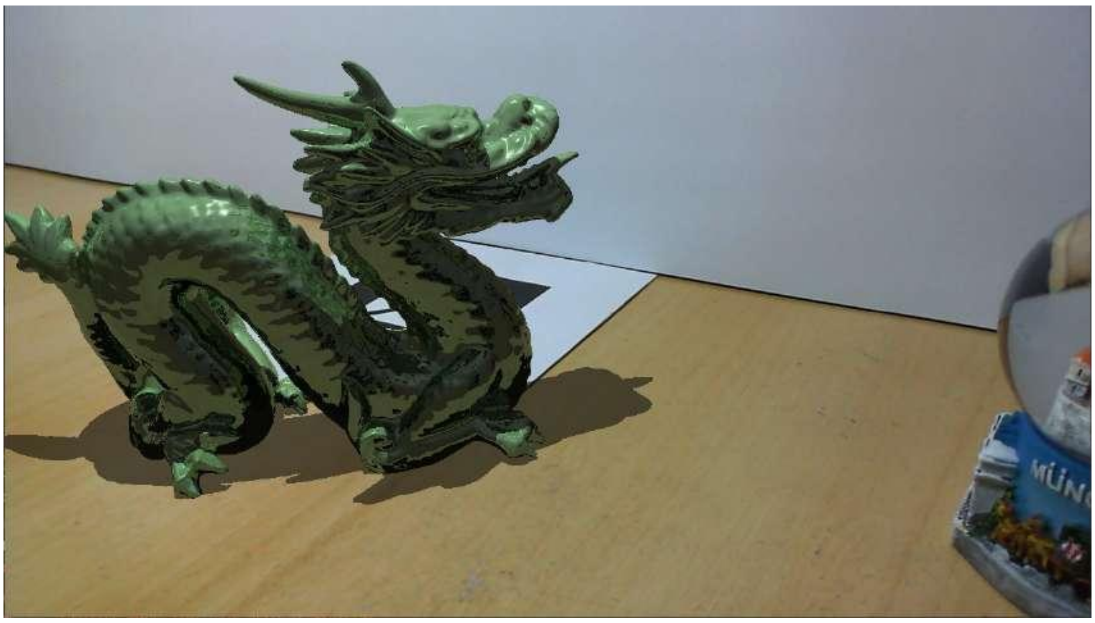


Figure 17 – Dragon model rendered with a reflexive material. Since it does not have a smooth surface, we see a blurred reflected snow globe on it's surface, mainly a blue region.

shader. For pixels referring to the geometry of interest was assigned an alpha equals to 1.0. Thus, in the framebuffer generated by the ray tracer, three distinct alpha values were presented, the value 0 for pixels that did not correspond to the geometry of interest, nor the shadow region; a value between 0 and 1.0 for pixels that matched a shadow region, we used 0.6 for opaque geometry shadow and 0.3 for translucent geometry shadow; and the value 1.0 for pixels corresponding to the geometry of interest.

In the generation of the scenes presented in Figures 14, 15, 17, 18, 19, and 20, we inserted in the virtual world a cube measuring 220mm on each face, positioned around the geometry of interest. Each face was textured with a previously captured image of the environment. The textures are shown in Figure 16. For all scenes, lighting is done by a white ambient light of intensity 0.8 and a point light with the same x and y coordinates as the model but translated 218mm in Z axis in relation to the model's base. The scenes were rendered in a window of 1280 $\times$ 720 pixels. All models were translated in relation to the fiducial marker for a better visualization of the shadows.

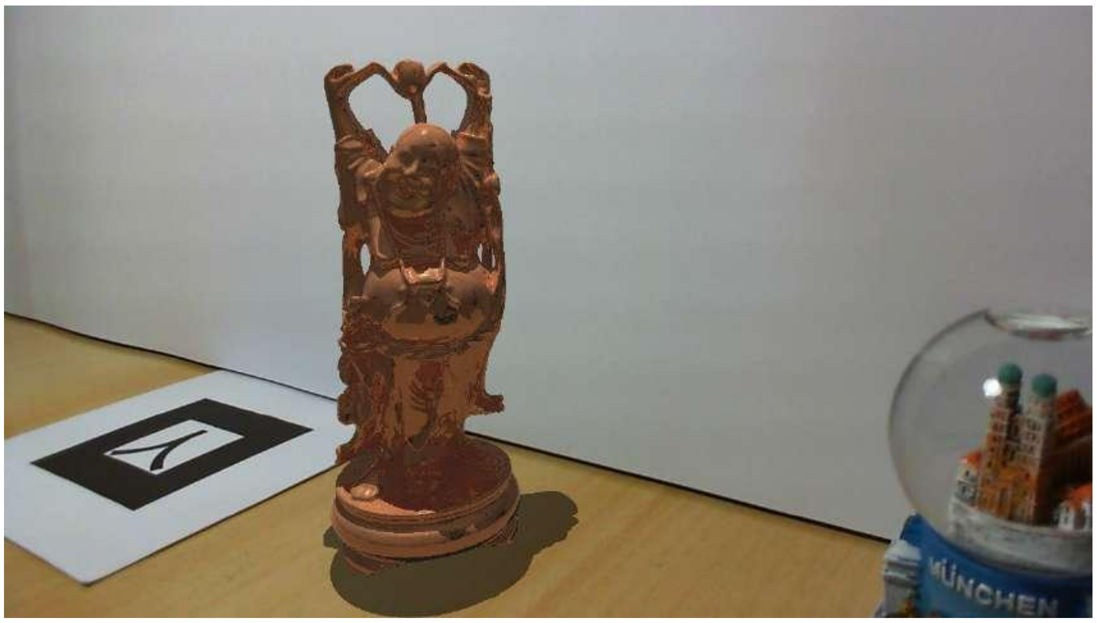


Figure 18 – Happy Buddah model rendered with a reflective surface. The reflection of the snow globe can be perceived by small blue areas along its surface.

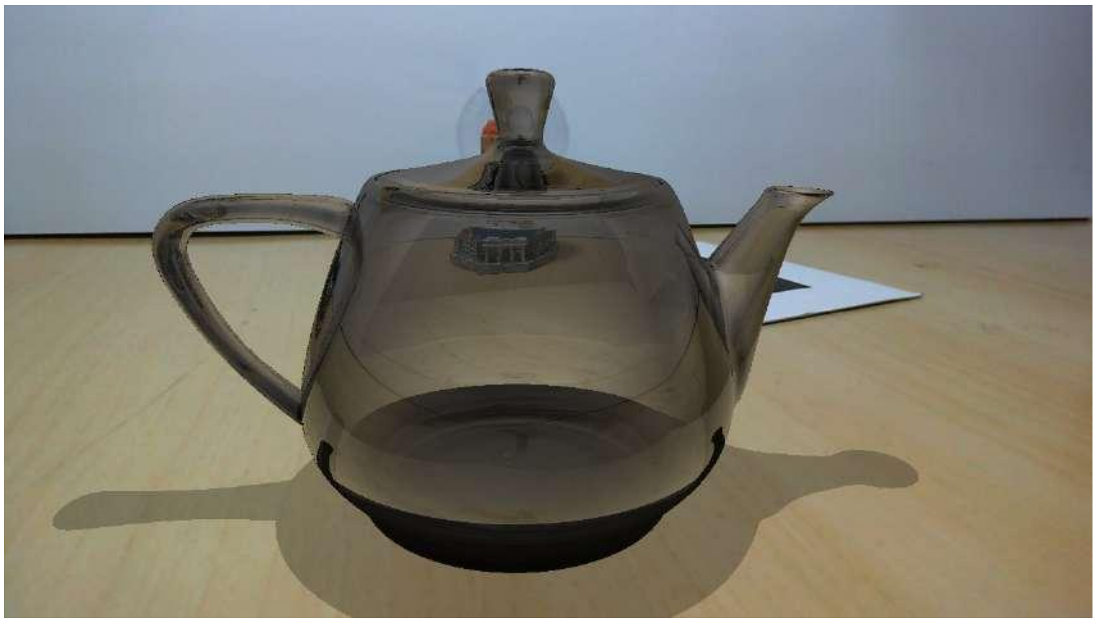


Figure 19 – Utah teapot model rendered with a glass material. In the scene we can see the snow globe texture through the teapot body. The texture position in skybox matches the real snow globe position.

4.2 QUANTITATIVE ANALYSIS

We observed that using the ARRay-Tracing, the maximum frame rate was 30 fps for all scenes. This maximum value is due to the limitation in the camera's capture capacity. For the pixel format used by artoolkitX, the camera we used in the tests captures a maximum of 30 fps. Some scenes that rendered more complex models with specific materials of bigger computational cost calculation, like the reflective material in VKRay or the glass material

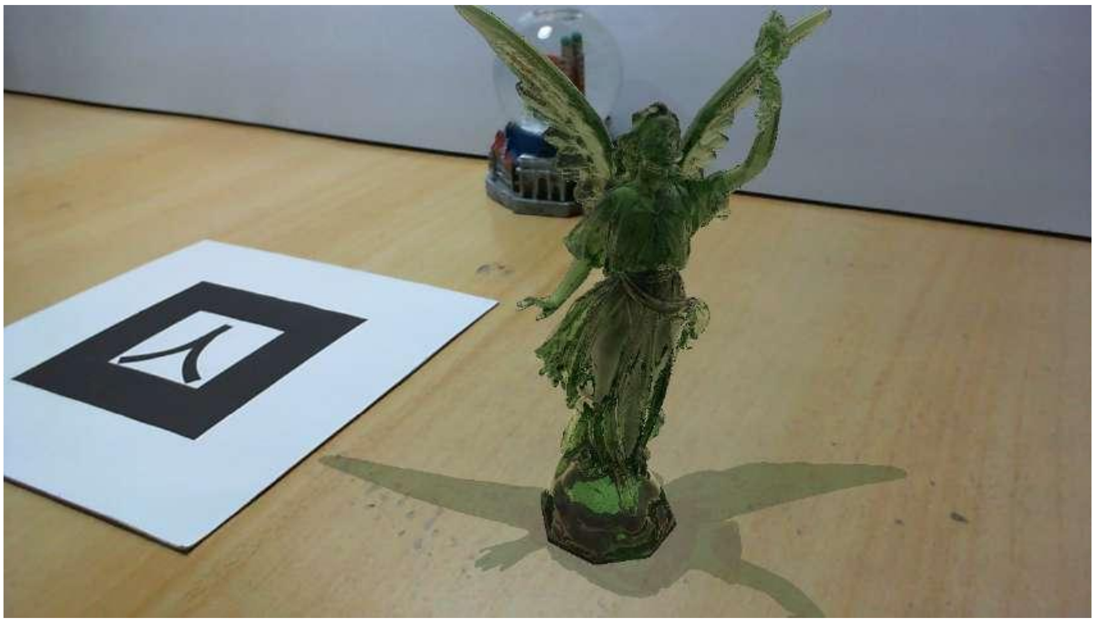


Figure 20 – Lucy model rendered with a glass material. It is possible to perceive the refraction of the snow globe through the colors in Lucy's left wing. The texture position in skybox matches the real snow globe position.

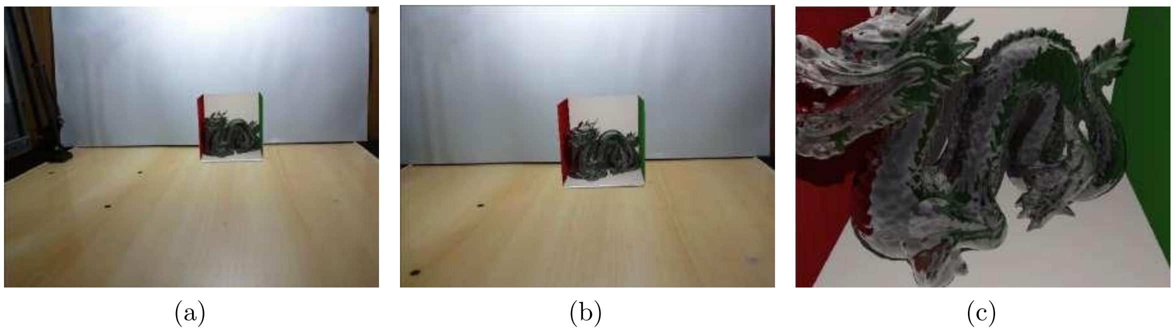


Figure 21 – Dragon with glass material rendered in Optix. In (a), (b), and (c), the viewport from the beginning, middle and end of camera's trajectory.

in Optix, had a higher frame rate when the camera was away from the object, decreasing as the camera approached the object, as expected, since the number of intersections calculations increases as the camera moves closer to the models and the amount of detail increases.

In order to verify with practical examples that the impacts of the camera capture limitation is lower than ray tracing when rending more complex scenes, we tested bringing the camera closer to the marker and recording the frame rate during this approximation. In these tests, we initially recorded the camera's position and orientation using the ARRay-Tracing version considering one record per frame, and then consumed this data using Optix and VKRay versions to repeat the same camera movement. Figures 21 and 22 show the viewport for the beginning, middle and end of camera's trajectory for tested Optix scenes, while Figure 23 shows the measured frame rate for these scenes.

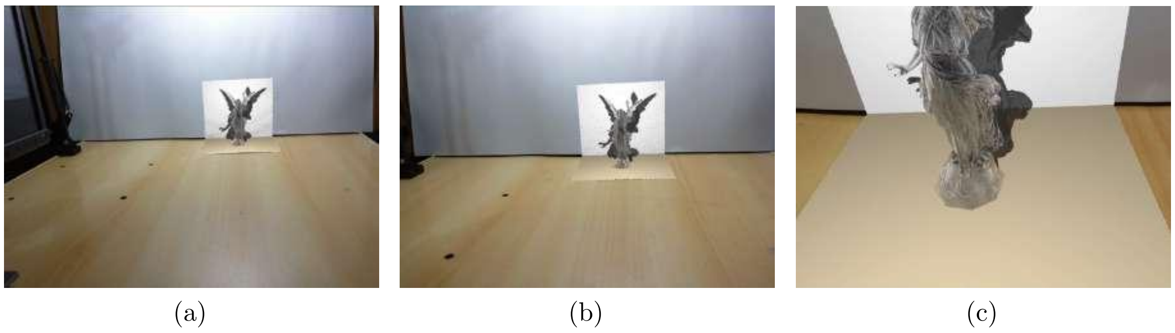


Figure 22 – Lucy with glass material rendered in Optix. In (a), (b), and (c), the viewport from the beginning, middle and end of camera's trajectory.

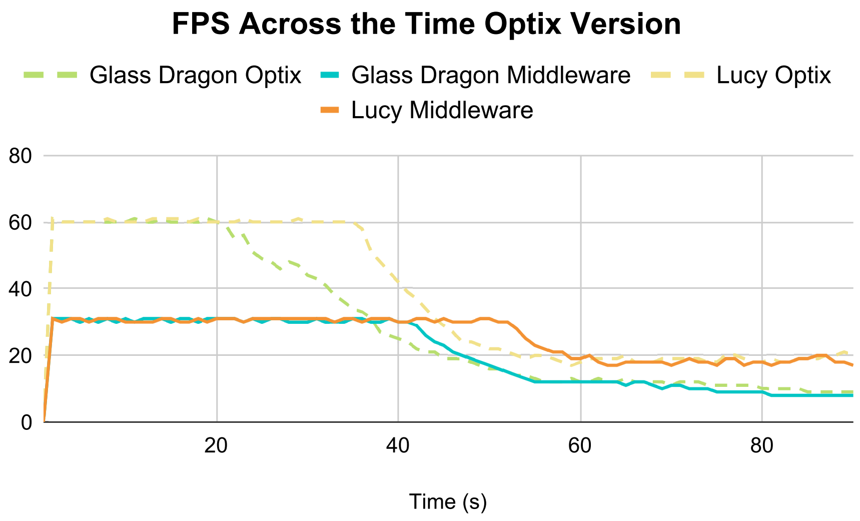


Figure 23 – Frame rate across the time Optix version.

Figure 23 shows that the application with pure Optix initially reaches a higher frame rate, around 60 fps, while ARRay-Tracing is limited to 30 fps for both scenes. As the camera approaches the rendered model, the frame rate drops in both versions and starts to tend towards the same value for each scene.

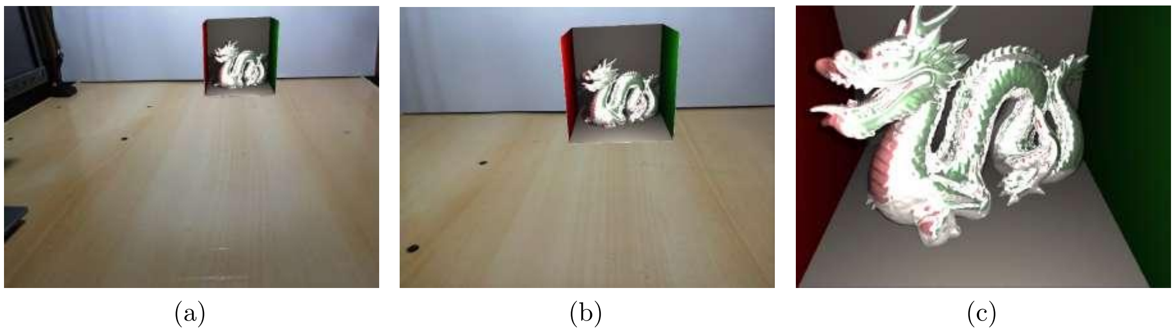


Figure 24 – Metallic Dragon rendered in VKRay. In (a), (b), and (c), the viewport from the beginning, middle and end of camera's trajectory.

Figures 24 and 25 show the viewport for the beginning, middle and end of camera's

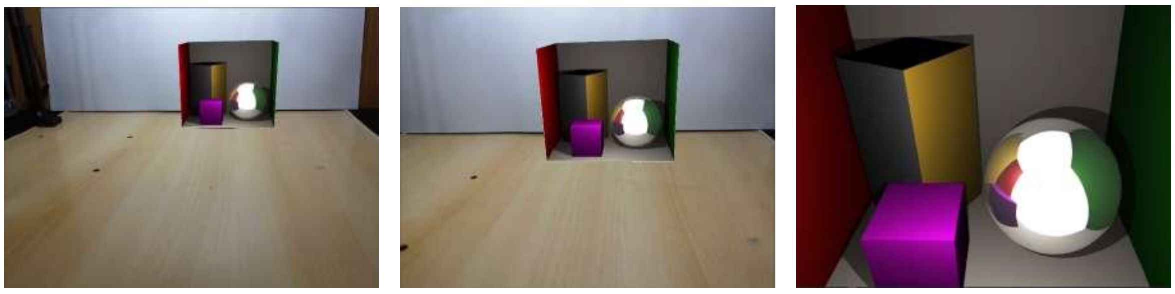

(a) (b) (c)

Figure 25 – Cuboids and sphere rendered in VKRay. In (a), (b), and (c), the viewport from the beginning, middle and end of camera's trajectory.

trajectory, while 26 shows the measured frame rate for VKRay scenes.

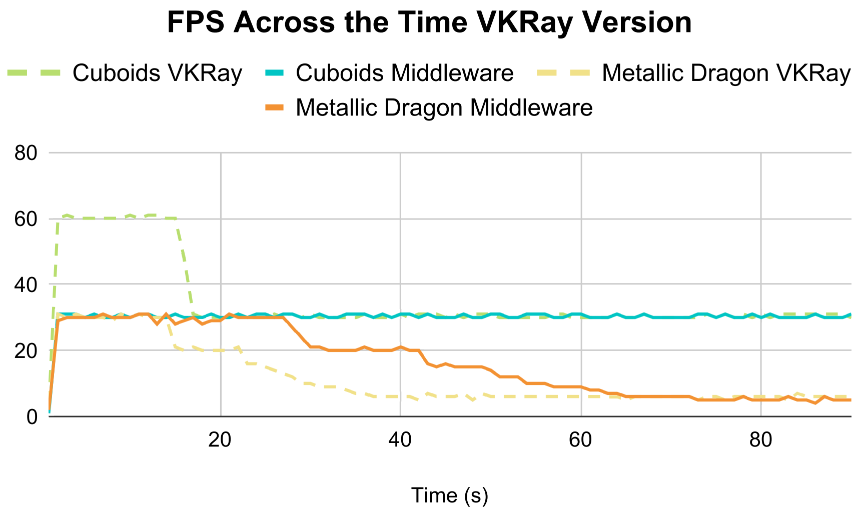


Figure 26 – Frame rate across the time VKRay version.

In Figure 26, two slightly different scenarios are shown. In the first scene, the cuboids, the frame rate starts higher on VKRay, around 60 fps, while the ARRay-Tracing is limited to 30, as a result of the limitation of the camera that captures at most 30 frames per second. However, being a less computationally costly rendering scene, the frame rate with the camera near the models is still relatively high, staying close to 30 fps in VKRay and holding constant in ARRay-Tracing. In the second scene, the metallic dragon, as it is computationally more expensive, the frame rate is already limited to 30 fps for both VKRay and ARRay-Tracing. As the camera approaches the model, the frame rate similarly decreases and converges to the same value for both cases.

By the tests, we observed that when rendering more complex scenes, the camera's capture limitation doesn't impact the frame rate, as ray tracing, which has the highest order of complexity, results in a frame rate lower than 30 fps on the tested hardware.

For each rendered scene, we also measured the frame rate with the camera in a stationary position. In some cases, initially, we placed the camera farthest from the marker, with a viewport similar to Figure 12b, and then we made a new measurement with the camera positioned very close to the marker, with a viewport similar to Figure 13. The fps corresponds to the average value obtained over a measurement interval of 3 minutes. Tables 3 and 4 show the fps values for the Optix and VKRay versions, respectively. Comparing the values for lower complexity scenes, as cuboids and sphere, and Bunny, we can see that the performance is equivalent using Optix and VKRay as ray tracing framework.

Table 3 – Rendered scenes and its fps for Optix version.

	CS	**B**	**HB**	**GL**	**CGL**	**GD**	**CGD**
Array-Tracing	29.99	30.0	29.99	29.99	14.20	29.99	10.16
Optix	60.0	59.99	59.97	51.21	14.43	37.81	10.13

Scenes: CS - Diffuse cuboids and metallic sphere, B - Specular Bunny, HB - Metallic Happy Buddha into a Cornell Box, GL - Glass Lucy, CGL - Close Glass Lucy, GD - Glass Dragon, CGD - Close Glass Dragon.

Table 4 – Rendered scenes and its fps for VKRay version.

	CS	**B**	**HB**	**HBF**	**MD**	**CMD**
ARRay-Tracing	30.19	30.15	30.07	30.32	15.16	5.73
VKRay	59.79	59.75	29.99	59.51	15.01	5.98

Scenes: CS - Diffuse cuboids and metallic sphere, B - Specular Bunny, HB - Metallic Happy Buddha into a Cornell Box, HBF - Metallic Happy Buddha on the floor, MD - Metallic Dragon, CMD - Close Metallic Dragon.

5 DISCUSSION

Through the tests we performed, it was possible to conclude that ARRay-Tracing facilitates enabling photorealism in augmented reality applications through the integration with ray tracing in a modularized way. It was also possible to notice that the tests with ARRay-Tracing reach frame rate values very close to those using only the Optix and VKRay frameworks, considering only the range between 0 and 30 fps. Regarding more extensive frame rates ranges, the ARRay-Tracing remains limited to 30 fps while frameworks manage to reach higher values. This frame rate limitation is due to camera limitations combined with the AR framework settings we used, not caused by the middleware.

For less complex scenes, ARRay-Tracing reached a value of 30 fps, which is considered real-time. For more complex scenes, the value tends to be lower. However, it is noteworthy that the test cases were built based on the existing examples in each framework's SDKs, not as commercial applications, and possibly, some implementation optimizations would help to increase the frame rate. Furthermore, the tests ran in an environment equipped with a graphics card that does not have hardware acceleration capabilities for ray tracing, contributing to the obtained values being lower.

Considering that the proposal presented in this work uses visual based tracking, specifically fiducial tracking, it is subject to the limitations of this type of tracking system. That is, under inadequate lighting conditions or when the marker is partially occluded, tracking may not work, and consequently, virtual elements will not be correctly aligned in the scene. A potential solution to mitigate this problem would be to replace the use of fiducial tracking with Natural Feature Tracking(NEUMANN; YOU, 1999), which was not supported by the version of artoolkitX used.

Until now, ray tracing frameworks do not contemplate the rendering of dynamic scenes containing animated models, only the geometric transformations of these elements. Due to its characteristic of building the Acceleration Structure before and outside the ray tracing loop, its use is only suitable for rigid geometry. The rendering of dynamic geometry whose shape varies with time demands the reconstruction of the AS with each new shape, causing a loss of performance and making its use in real-time unfeasible.

Finally, it is also important to mention that the middleware does not include any method for recognizing the lighting conditions of the physical environment, nor for reconstructing its geometry. The lighting used in the tests was defined arbitrarily in the ray tracing frameworks. Therefore, it is possible to notice a discrepancy in the lights and shadows between real and virtual objects in the final image.

6 CONCLUSION

In this work, we proposed the ARRay-Tracing, a middleware to integrate augmented reality and ray tracing frameworks in a modularized way. As the middleware works with camera mapping and image composition, any framework can be used with it as long as, for AR, it is possible to obtain the camera's coordinates: eye, look at, and up; and for ray tracing, it is possible to manipulate the image generated by the method execution.

We also present an evaluation of middleware using two different ray tracing frameworks, Optix and VKRay, integrated with the artoolkitX augmented reality framework. We could conclude that ARRay-Tracing does not add processing overhead to the application through the evaluation comparing the use of middleware to the use of only ray tracing frameworks, in Figures 23 and 26, as was expected since ray tracing has the biggest complexity order in the whole process. The frame rate values obtained are inversely proportional to the complexity of the rendered models and materials. Due to the camera used and the artoolkitX settings in the test environment, we identified the 30 fps limit for applications using the middleware. This maximum value is likewise obtained in applications using only the AR framework and is also considered the threshold to classify an application as real-time.

Through this work, we achieved and photorealistic visual effects in AR applications using the ARRay-Tracing middleware for controlled environment applications, where it is possible to simulate the real environment through a skybox built using real images as textures. For more complex scenes where the frame rate is less than 10 fps, AR's real-time interactivity principle is violated. Therefore, integration is satisfactory only in applications with less complex geometries and materials. Also, ARRay-Tracing did not include methods for reconstructing the geometry and lighting of the real environment, thus, geometry positioning and lightning conditions are arbitrarily defined. Consequently, real and virtual elements seem visually different, and there is no proper occlusion between them. Many AR applications are developed for the mobile platform, but since real-time ray tracing is unfeasible in this platform, ARRay-Tracing was proposed for desktop-based applications.

As future work, it would be interesting to add the integration with methods to reconstruct geometry and lightning of the physical environment to the middleware. This way, it would be possible to recreate the same lighting conditions in ray tracing frameworks, and the virtual models would be visually more similar to real objects.

REFERENCES

AGUSANTO, Kusuma; LI, Li; CHUANGUI, Zhu; SING, Ng Wan. Photorealistic rendering for augmented reality using environment illumination. In: IEEE. **The Second IEEE and ACM International Symposium on Mixed and Augmented Reality, 2003. Proceedings.** [S.l.], 2003. p. 208–216.

ALHAKAMY, A'aeshah; TUCERYAN, Mihran. Cubemap360: Interactive global illumination for augmented reality in dynamic environment. In: **2019 SoutheastCon**. [S.l.: s.n.], 2019. p. 1–8.

APPEL, Arthur. Some techniques for shading machine renderings of solids. In: **Proceedings of the April 30–May 2, 1968, spring joint computer conference**. [S.l.: s.n.], 1968. p. 37–45.

AZUMA, Ronald T. A Survey of Augmented Reality. **Presence: Teleoperators and Virtual Environments**, v. 6, n. 4, p. 355–385, 08 1997.

AZUMA, Ronald T.; BAILLOT, Yohan; BEHRINGER, Reinhold; FEINER, Steven K.; JULIER, Simon; MACINTYRE, Blair. Recent advances in augmented reality. **IEEE Computer Graphics and Applications**, v. 21, n. 6, p. 34–47, 2001.

BILLINGHURST, Mark; CLARK, Adrian; LEE, Gun. A survey of augmented reality. **Foundations and Trends in Human-Computer Interaction**, v. 8, n. 2-3, p. 73–272, 2015.

FERWERDA, James A. Three varieties of realism in computer graphics. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. **Human Vision and Electronic Imaging VIII**. [S.l.], 2003. v. 5007, p. 290–297.

GLASSNER, Andrew S. **An introduction to ray tracing**. [S.l.]: Morgan Kaufmann, 1989.

GOMES, Wnêiton; CAMILO, Celso; LIMA, Leonardo; CARDOSO, Alexandre; JR, Edgard Lamounier; YAMANAKA, Keiji. Artificial neural networks to recognize artoolkit markers. In: **International Conference on Artificial Intelligence and Pattern Recognition**. [S.l.: s.n.], 2007. p. 464–469.

HOUNSELL, Marcelo da Silva; TORI, Romero; KIRNER, Claudio. Realidade aumentada. In: TORI, Romero; HOUNSELL, Marcelo da Silva (Ed.). **Introdução a Realidade Virtual e Aumentada**. Porto Alegre: Editora SBC, 2020. p. 30–59.

HUGHES, John F; DAM, Andries Van; MCGUIRE, Morgan; FOLEY, James D; SKLAR, David; FEINER, Steven K; AKELEY, Kurt. **Computer graphics: principles and practice**. [S.l.]: Pearson Education, 2014.

JACOBS, Katrien; LOSCOS, Céline. Classification of illumination methods for mixed reality. In: WILEY ONLINE LIBRARY. **Computer Graphics Forum**. [S.l.], 2006. v. 25, n. 1, p. 29–51.

KÁN, Peter; KAUFMANN, Hannes. Differential irradiance caching for fast high-quality light transport between virtual and real worlds. In: IEEE. **2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)**. [S.l.], 2013. p. 133–141.

KELLER, Alexander.; FASCIONE, Luca; FAJARDO, M.; GEORGIEV, Iliyan; CHRISTENSEN, Per H.; HANIKA, Johannes; EISENACHER, Christian; NICHOLS, Greg Boyd. The path tracing revolution in the movie industry. In: **ACM SIGGRAPH 2015 Courses**. New York, NY, USA: Association for Computing Machinery, 2015. (SIGGRAPH '15). ISBN 9781450336345.

KELLER, Alexander; VIITANEN, Timo; BARRÉ-BRISEBOIS, Colin; SCHIED, Christoph; MCGUIRE, Morgan. Are we done with ray tracing? In: **ACM SIGGRAPH 2019 Courses**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGGRAPH '19). ISBN 9781450363075.

KIM, Kangsoo; BILLINGHURST, Mark; BRUDER, Gerd; DUH, Henry Been-Lirn; WELCH, Gregory F. Revisiting trends in augmented reality research: A review of the 2nd decade of ismar (2008–2017). **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 11, p. 2947–2962, 2018.

LEE, Cha; RINCON, Gustavo A.; MEYER, Greg; HÖLLERER, Tobias; BOWMAN, Doug A. The effects of visual realism on search tasks in mixed reality simulation. **IEEE Transactions on Visualization and Computer Graphics**, v. 19, n. 4, p. 547–556, 2013.

MARSCHNER, Steve; SHIRLEY, Peter. **Fundamentals of computer graphics**. [S.l.]: CRC Press, 2018.

MARTINEZ, Héctor Antonio Villa. **Accelerating algorithms for Ray Tracing**. [S.l.], 2006.

MCREYNOLDS, Tom; BLYTHE, David. **Advanced graphics programming using OpenGL**. [S.l.]: Elsevier, 2005.

MILGRAM, Paul; KISHINO, Fumio. A taxonomy of mixed reality visual displays. **IEICE TRANSACTIONS on Information and Systems**, The Institute of Electronics, Information and Communication Engineers, v. 77, n. 12, p. 1321–1329, 1994.

NEUMANN, U.; YOU, S. Natural feature tracking for augmented reality. **IEEE Transactions on Multimedia**, v. 1, n. 1, p. 53–64, 1999.

OLIVEIRA, Douglas Coelho Braga de. **Dynamic-Object-Aware Simultaneous Localization and Mapping for Augmented Reality Applications**. 66 p. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de Juiz de Fora, Juiz de Fora, 2018.

PEREIRA, Lidiane Teixeira; JUNIOR, Wellingston Cataldo Roberti; SILVA, Rodrigo Luis de Souza da. Photorealism in mixed reality: A systematic literature review. **International Journal of Virtual Reality**, v. 21, n. 1, p. 15–29, 2021.

PESSOA, Saulo; MOURA, Guilherme; LIMA, João; TEICHRIEB, Veronica; KELNER, Judith. Photorealistic rendering for augmented reality: A global illumination and brdf solution. In: IEEE. **2010 IEEE Virtual Reality Conference (VR)**. [S.l.], 2010. p. 3–10.

PORTER, Thomas; DUFF, Tom. Compositing digital images. In: **Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: Association for Computing Machinery, 1984. (SIGGRAPH '84), p. 253–259. ISBN 0897911385.

SANTOS, Artur Lira dos; LEMOS, Diego; LINDOSO, Jorge Eduardo Falcão; TEICHRIEB, Veronica. Real time ray tracing for augmented reality. In: IEEE. **2012 14th Symposium on Virtual and Augmented Reality**. [S.l.], 2012. p. 131–140.

SCHWANDT, Tobias; BROLL, Wolfgang. A single camera image based approach for glossy reflections in mixed reality applications. In: IEEE. **2016 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)**. [S.l.], 2016. p. 37–43.

USHER, Will. The shader binding table demystified. In: MARRS, Adam; SHIRLEY, Peter; WALD, Ingo (Ed.). **Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX**. Berkeley, CA: Apress, 2021. p. 193–211. ISBN 978-1-4842-7185-8.

WHITTED, Turner. An improved illumination model for shaded display. In: **Proceedings of the 6th annual conference on Computer graphics and interactive techniques**. [S.l.: s.n.], 1979. p. 14.

YU, Insu; MORTENSEN, Jesper; KHANNA, Pankaj; SPANLANG, Bernhard; SLATER, Mel. Visual realism enhances realistic response in an immersive virtual environment - part 2. **IEEE Computer Graphics and Applications**, v. 32, n. 6, p. 36–45, 2012.

ZORZAL, Ezequiel R.; SILVA, Rodrigo L. S. Software. In: TORI, Romero; HOUNSELL, Marcelo da Silva (Ed.). **Introdução a Realidade Virtual e Aumentada**. Porto Alegre: Editora SBC, 2020. p. 103–111.