Real Time Ray Tracing for Augmented Reality

Conference Paper · May 2012

DOI: 10.1109/SVR.2012.8

Artur Lira dos Santos
Federal University of Pernambuco
8 PUBLICATIONS **58** CITATIONS
SEE PROFILE

Jorge Lindoso
Federal University of Pernambuco
4 PUBLICATIONS **22** CITATIONS
SEE PROFILE

Veronica Teichrieb
Federal University of Pernambuco
154 PUBLICATIONS **739** CITATIONS
SEE PROFILE

Real Time Ray Tracing for Augmented Reality

Artur Lira dos Santos, Diego Lemos, Jorge Eduardo Falcão Lindoso, Veronica Teichrieb
Voxar Labs, Informatics Center
Federal University of Pernambuco
Recife, Brazil
{als3, dlam, jefl, vt}@cin.ufpe.br

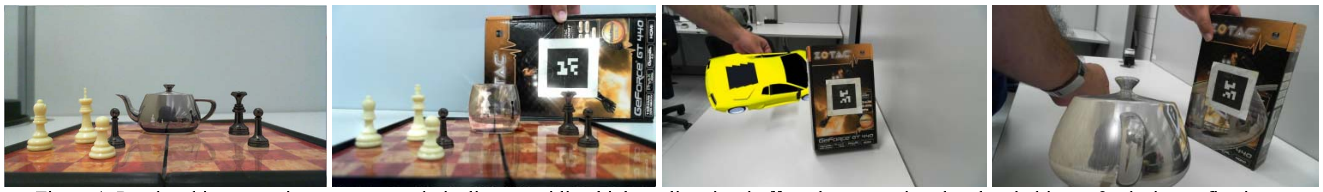


Figure 1. Rendered images using our proposed pipeline, providing high quality visual effects between virtual and real objects. Occlusion, reflections, refractions, soft shadows, custom shaders and lens effects are supported in real time.

Abstract—**This paper introduces a novel graphics rendering pipeline applied to augmented reality, based on a real time ray tracing paradigm. Ray tracing techniques process pixels independently from each other, allowing an easy integration with image-based tracking techniques, contrary to traditional projection-based rasterization graphics systems, e.g. OpenGL. Therefore, by associating our highly optimized ray tracer with an augmented reality framework, the proposed pipeline is capable to provide high quality rendering with real time interaction between virtual and real objects, such as occlusions, soft shadows, custom shaders, reflections and self-reflections, some of these features only available in our rendering pipeline. As proof of concept, we present a case study with the ARToolKitPlus library and the Microsoft Kinect hardware, both integrated in our pipeline. Furthermore, we show the performance and visual results in high definition of the novel pipeline on modern graphics cards, presenting occlusion and recursive reflection effects between virtual and real objects without the latter ones needing to be previously modeled when using Kinect. Furthermore, an adaptive soft shadow sampling algorithm for ray tracing is presented, generating high quality shadows in real time for most scenes.**

Keywords—Ray tracing; augmented reality; real time; occlusion; reflection; refraction; kinect.

I. Introduction

Ray tracing techniques have been used as an efficient way to generate detailed images based on optical physics phenomena simulation since their introduction by the work of A. Appel [1] in 1968. As computing techniques in general, ray tracing has its advantages and drawbacks. While its rendering results can achieve more accurate simulation of the real world than traditional rasterization techniques, the computational power demanded for ray tracing made its using unfeasible during many decades for real time applications, while rasterized approaches offered high performance at expense of accuracy. Therefore, well-established real time graphics pipelines such as OpenGL and Direct3D use rasterization as main technique for rendering.

However, since GPU pipelines became highly programmable with recent architectures such as NVIDIA CUDA [2], they are used as a coprocessor for executing highly parallel algorithms, including ray tracing in real time[3][4][5]. This work exploits this architecture to implement our ray tracing pipeline for application on augmented reality, named RT^{2}AR.

RT^{2}AR processes pixels independently from each other, allowing an easy integration with image-based tracking techniques, contrary to rasterization. Furthermore, AR scenes can benefit from this pipeline since in this case most of pixels come directly from the RGB camera, reducing the number of pixels with high ray tracing costs.

In order to achieve near photorealistic effects on our pipeline, some real object required to be previously modeled, allowing virtual and real objects interactions. On the other hand, a Kinect's depth information based approach was also implemented, avoiding the need for pre-modeling of real objects. Finally, a new soft shadow technique is proposed, being an optimization of a path tracing shadowing technique.

II. Related Work

There are some interesting published studies about photorealism in augmented reality, despite the fact that it's a recent research topic. The majority of these publications are restricted to a specific problem, such as occlusion, illumination, shadowing and Microsoft Kinect usage for augmented reality purposes. Another important topic related to our work refers to ray tracing techniques applied in real time rendering. Therefore, the following subsections describe briefly these related research topics.

A. Photorealism in Augmented Reality

Fournier et al. [6] proposed a method to coherently insert virtual objects in a real scene calculating the approximation of the reflectance, geometry and light sources from the real world. Finally, virtual objects are rendered using an expensive ray casting technique.

Another work that deals with global illumination is from Gibson and Murta [7], who proposed a solution based on environment maps to integrating synthetic objects into background photographs at interactive rates. As result they got some consistent virtual shadows in a real scene.

Another important work was developed by Jacobs et al. [8] that presented a solution to provide consistent shadows between virtual and real objects of a scene. Their solution consists of three steps: shadow detection, shadow projection and shadow generation. However, this solution requires informing manually the geometry of the considered real objects and supports only one light source. To solve this problem, Sato et al. [9] developed a way to acquire such information automatically from a pair of images using an omni-directional stereo algorithm. However, in order to coherently insert shadows, the local scene have to be manually modeled, because the acquired geometry is imprecise and contains only information about the distant scene.

The median-cut algorithm was proposed by Debevec [10] as a method to extract geometric light sources from the environment map, including their direction, color and intensity. This information provides the necessary reconstruction of real light sources to render drop shadows of a virtual object onto the real scene.

Heidrich et al. [11] presented techniques for realistic shading and lighting using computer graphics hardware. They discussed multipass methods for high quality local illumination using physically-based reflection models, as well as techniques for the interactive visualization of non-diffuse global illumination solutions. The results are combined with normal mapping for increasing the visual complexity of rendered images. They obtain a good frame rate, however their method is based on environment maps, so the real scene must be static, because they have to pre-capture the environment map images.

Another important characteristic of realistic scenes is the reflection between the objects. Uranish et al. [12] proposed a method for rendering an inter-reflection between a marker and a glossy floor in augmented reality. This method has some limitations, such as: the marker has to be placed on a flat surface, only one white light source is supported and the marker has to be a black and white pattern.

Most of the previous works about photorealism focus on a particular realistic effect. Pessoa et al. [13] proposed a solution that comprises lots of realistic techniques and effects into a single system, called RPR-SORS. RPR-SORS uses different shaders, OpenGL and Direct3D rasterization as basis rendering techniques.

B. *Real Time Ray Tracing*

Ray tracing has some advantages compared to other rendering methods when dealing with augmented reality systems. Taking this into account, Scheer et al. [14] proposed a method to generate shadows in augmented reality scenarios using ray tracing. In order to achieve this, the method requires that the geometry of the real scene is previously modeled. Their approach obtained realistic shadows, however they have low-frame rate which is critical for augmented reality scenes.

To solve the problem of ray tracing performance for interactive applications, Santos et al. [3] describe in detail the study, comparison and hardware implementation of many kD-Tree traversal algorithms. They proposed two new algorithms based on their analysis, aiming performance improvement.

C. *Kinect in Augmented Reality*

Franke et al. [15] presented a framework to include depth sensing devices into X3D in order to enhance visual fidelity of X3D mixed reality applications. In their work, they explain how to calibrate the depth and RGB image data in a meaningful way through calibration for devices that do not already come with pre-calibrated sensors. They also proposed a technique to make the occlusion of virtual objects by the real scene and to implement a shadow mapping approach to generate shadows using the depth information.

Izadi et al. [16] proposed a method to reconstruct, geometrically precise, 3D models from the real scene in real-time, using only the depth data from Kinect. In their system, an user holding a standard Kinect camera can move within any indoor space, and reconstruct a 3D model of the physical scene within seconds. The system continuously tracks the 6 degrees-of-freedom pose of the camera and fuses new viewpoints of the scene into a global surface-based representation. In addition, some ray traced rendering effects can be calculated in real time to create shadows, lighting and reflections. However, their pipeline takes advantage of ray tracing only in the blending stage between the virtual and real scene obtained as an implicit surface, without using a complete ray tracing technique for all the stages. For instance, virtual objects are rendered using a traditional rasterization pipeline. A complete real time ray tracing pipeline is the key difference of our work in this case.

III. Ray Tracing Pipeline Applied to Augmented Reality

The ray tracing algorithm presents the characteristic of being easily and efficiently parallelized, since each ray/pixel computation is independent, meaning that parts of the image can be processed separately. This pixel independence makes the ray tracing approach a good candidate for integration with computer vision algorithms required by augmented reality applications, such as visual tracking techniques. In addition, with ray tracing, there is no need for using Differential Rendering [17] to determine visibility and shadows between real and virtual objects. Furthermore, there is no computation of environment maps for every reflective object, since the ray tracing algorithm can itself handle the reflections independently for each pixel.

The simplicity of merging AR features in a ray tracing pipeline, combined to a plethora of near-realistic visual effects from ray tracing, makes this hybrid system an appropriate rendering pipeline to achieve high visual quality in AR systems. The set of features available in our pipeline is further detailed in IV, while the next subsections briefly describe the RT^2 and RT^{2}AR rendering pipelines proposed in our work.

A. *RT^2*

Real Time Ray Tracer or simply RT^2 [18] refers to our novel graphics rendering API, with the purpose of rendering virtual 3D scenes in real time. The main difference between the RT^2 and standard graphics pipelines (such as OpenGL and Direct3D) resides in the main techniques used. While both OpenGL and Direct3D use rasterization as the core technique for rendering, RT^2 uses ray tracing to obtain most of the visual effects. Another important difference is that OpenGL and Direct3D usually require multiple passes over the pipeline stages to obtain a final image with advanced visual effects. On the contrary, the RT^2 implements a single pass pipeline that consists in an iterative per-pixel ray traversal and shading. Over a GPU pipeline, the single pass approach is generally more efficient in terms of memory bandwidth, since there's no need for additional memory transfers from the input and output data for each pass. For example, to obtain shadows with OpenGL is necessary to pass the scene through the pipeline for every light source considered in the scene, while the RT^2 can obtain the same or even better visual results without the need for multiple scene passes. The same occurs with reflections; when using OpenGL, one environment map of the virtual scene has to be computed for every visible 3D object (one rendering pass per reflective object). RT^2 can obtain better visual results including correct inter-reflections and self-reflections in a single pass, independently of the number of visible objects in the scene.

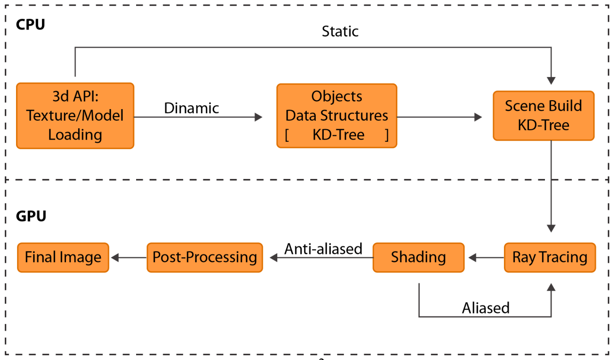


Figure 2. RT^2 pipeline.

To achieve real time frame rates, the RT^2 ray tracing core runs on GPU [3]. It uses auxiliary data structures such as kD-Trees [4], BIHs [19] and BVHs [20] that are implemented in CUDA, reaching speedups of at least one order of magnitude when compared to the CPU approach. Furthermore, the RT^2 supports anti-aliasing [21], anisotropic texture filtering, geometry instancing and custom shaders. Furthermore, it has multi-GPU capability, being highly parallelizable and consequently highly scalable in future generation GPUs. Figure 2 shows the different stages of the RT^2 pipeline. See [18] for detailed information of the pipeline implementation.

B. RT^2AR

The RT^2AR architecture [18], meaning the extended RT^2 for Augmented Reality purposes, is composed of several layers. The top layer refers to the RT^2AR API, a public interface which provides all available functionalities to the user, namely scene configuration, XML instancing and binding of objects to markers and camera (RGB or Kinect) intrinsic parameters configuration. RT^2AR uses the markers' positions and orientations data to allow objects' transformations which are performed modifying a ray's properties. This layer is supported by the Controller, responsible for the management of the AR scene. The Controller contains two libraries for camera management: Microsoft Kinect SDK [22], and OpenCV [23]. It also contains the ARToolkitPlus [24] library, responsible for the management of the markers. The last layer refers to the RT^2 library, responsible for loading the 3D scene and ray tracing simulation.

On ray tracing, there is no interdependence between neighbor pixels computations as exists in rasterization. The fine-grained modularity of the ray tracing algorithm, on which each pixel has its own computation flow, makes it simple to add new capabilities such as user-defined geometric primitives and custom shaders, including different types of shadowing techniques.

This modularity is an advantage of ray tracing for AR when compared to a rasterization-based approach, since there's no need to treat every visual effect over each object as a result of many rendered images, like a combination of environment and shadow maps with complex differential rendering techniques, frequently present in most of the previous works [13][17].

Therefore, adding AR components in the RT^2 pipeline was straightforward. The desired visual effects could be done in a simpler manner, since each screen ray tries independently to determine its final color based on the intersections and bounces the ray will do through the scene.

It is important to notice that on RT^2AR, visual effects such as occlusion, reflection, refraction and shadowing are results of the same steps: the process of finding the nearest object that intersects a ray, followed by a shading computation. The differences to obtain these effects are present only in the shading algorithm, one specific for each effect.

Thus, adding AR features inside the RT^2 was a simple matter of creating certain types of primitives that represent the real objects in the scene and also adding proper shaders to support different kinds of cameras and real objects effects, initially not supported in RT^2.

To help understanding these concepts, Figure 14 shows the GPU-based RT^2AR pipeline as a flow diagram. The yellow rectangle covers the new AR stages added into the RT^2. The flow starts with the primary or screen ray and has three sequential iterative stages:

1. *Search for Nearest Ray-Object Intersection*: the input ray is passed to the kD-Tree Ray Traversal algorithm to find the nearest object intersection. Then, if the Kinect is enabled, it is used to determine if the nearest object is a real object represented by the z-depth value from Kinect's depth map. Else, it goes to the next stage.
2. *Shading*: This stage determines the color contribution of the current ray to its pixel. The input is the intersection information retrieved from the previous stage. Depending on the type of object intersected, a different set of shaders will be executed. This stage is also responsible for the

shadow ray traversals. Most of the AR additions in the pipeline occurred in this stage, since it's during the shading process that the camera's pixel contribution is done, as well as the influence of real objects over virtual ones and vice-versa.

3. *Secondary Ray Generation*: If the material of the intersected object is reflective or refractive, new rays must be created and traced through the pipeline. The input of this stage is the intersection type, the normal of the intersection and the type of bounce (reflection or refraction) the ray can do. The original ray is then transformed into others that become input rays of the first stage.

To obtain a high level of realism, it is important to make virtual objects interact with the real scene, so we can perceive the virtual object as naturally immersed as possible into the real environment. To provide and support this, the concept of a "ghost" object is used in this work, in a similar way to "phantom" objects [13]. In RT^{2}AR they are the real objects associated to markers, that were previously modeled having a virtual representation in the system.

If the primary or screen ray intersects a ghost object or the Kinect depth information, the incoming pixel color from the RGB camera is used for shading, since this pixel already represents the real object intersected. In this case, no special virtual shading is necessary. However, for secondary rays, ghost objects execute all the available shaders, making possible the interaction between real and virtual objects, and the generation of reflection, refraction and shadow effects considering these objects. This dual behavior of the ghost object is shown in Figure 14, where its intersection in Stage I creates two new paths in Stage II, based only on the ray's type.

IV. Implemented Techniques

In AR scenes, visual realism is a very important issue. Therefore, we intended to build a consistent way to render such scenes, where virtual objects are intuitively blended into the real scene, creating a pipeline that supports visual effects with the highest possible quality considering the requirement of real time performance and nowadays hardware constraints.

In this section, those visual effects are further explained, considering implementation details regarding the GPU-based RT^{2}AR pipeline, the interaction between real and virtual objects, and also the usage of different types of cameras.

A. Occlusion

The main problem associated with occlusion or visibility is the separation of what's visible from what's hidden. In AR scenes this separation becomes more complex since it's a merging of two scenes, one real and another virtual. Rasterization-based techniques handle this merging process through multiple passes in the GPU pipeline, using techniques such as Differential Rendering [17] to determine the final image with correct occlusion verification between real and virtual objects. The RT^{2}AR pipeline solves naturally the visibility problem using the ray traversal algorithm, since the nearest intersected object point by the primary ray traversal is exactly the visible point in the final image. The nearest intersection can be either with a virtual object or a real one, since in the traversal they are treated equally. Therefore, there's no need to use Differential Rendering on RT^{2}AR.

For real objects, the occlusion effect can be trivially achieved since the ray tracer has a virtual representation of its ghost object. As shown in Figure 14, we cast a primary ray, and if this primary ray intersects a ghost object, a real object is closer to the camera, leading to the use of the pixels that came from the capture device. Otherwise, if the primary ray intersects a virtual object, this virtual object is closer to the camera, occluding any other objects. Figure 3 illustrates the result of these different situations.

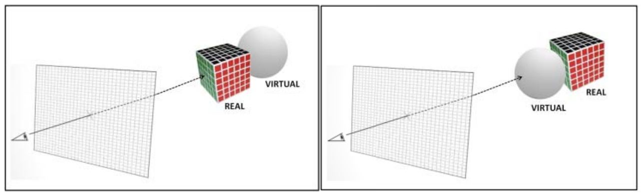


Figure 3. Real and virtual objects occlusion cases.

In addition, our work used the Kinect for tracking the camera distance from real objects, such as walls and small objects, avoiding the need for previously modeled versions of these objects. Through the depth map provided by the Kinect, it was possible to retrieve the distance information of the nearest intersection of each pixel from the camera screen. With this depth value, we can perform a comparison of this distance with the depth of the intersection from the ray traversal of the scene. Therefore, when a primary ray intersects a virtual object, we compare this distance to the depth map buffer to see which is closer to the camera; if the camera image pixel is closer, we use the pixel that came from the camera. Otherwise, we shade the virtual object.

It's important to explain that the depth information provided by the Kinect needs to be transformed into a spatial distance to the pixel, since the original depth refers to a parallel distance from the camera plane, not from the pixel to the point of intersection. To calculate a 3D position of the Kinect pixel according to RT^2's format, where the real distance is a multiple of the norm of primary ray, since the depth is the basis of a right triangle, where the hypotenuse is the real distance, represented by the primary ray.

B. Reflection

Inter-reflections among objects are a common visual effect observed in real world that may be used in augmented scenes to enhance realism. The RT^2 requires that every object material has a RGB reflectance image map defining each reflective region of objects' surface. This way, the material defines if and how the object is reflective.

Reflections between objects happen when a ray intersects an object, bounces to a different direction and consequently intersects another surface. This effect can be simulated in a simple way because the ray tracer does this automatically due to its functioning nature. According to our proposed pipeline (see Figure 14), first the primary ray is generated and then the traversal is performed. The ray intersects an object and the shaders of this object are executed. After that, the ray is transformed into a reflected secondary ray and goes back to the first stage of the pipeline. Then, the reflection ray intersects another object, so the shade contribution from the second object is calculated and added to the first one.

If the primary ray intersects a virtual object and then its secondary ray hits a real object, a reflection of the real object will appear on the surface of the virtual one. In a similar way, if at first a ghost object was intersected and then the reflected ray hits a virtual one, this virtual object will appear reflected on the surface of the real object. Therefore, the inter-reflections between virtual and real objects depend only on the intersections orders of the primary and secondary rays. Both of these intersection orders are illustrated in Figure 4.

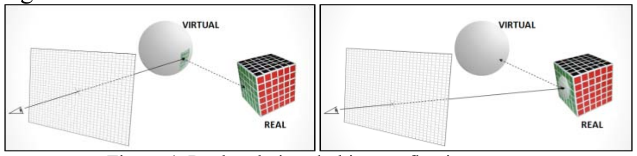


Figure 4. Real and virtual objects reflection cases.

The Kinect depth information was also used to achieve reflections of a real markerless object on a virtual one. However, the contrary way (virtual object reflected on a real one) wasn't possible, since the Kinect doesn't provide normal information of the real world surface. The normal vector is necessary to create the reflected ray. This explains why the pipeline of the Figure 14 doesn't provide a path to Stage III when the closest ray intersection was the Kinect's depth information.

When handling secondary rays, it's not possible to directly use Kinect's depth value from the current pixel index to determine the closest intersection. These rays, when projected onto the screen space, generally pass through many different pixels, while the primary ray passes only through its pixel on the screen, as shown in Figure 5.

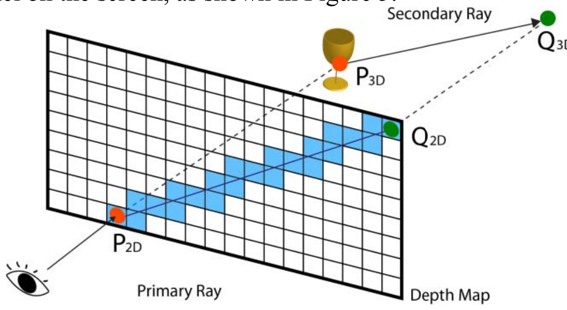


Figure 5. Depth map traversal scheme.

Therefore, to determine the closest depth intersection of a secondary ray, a new traversal method is proposed in this work. It's based on Bresenham's [25] line rasterization algorithm, with the difference that we evaluate both ceil and floor functions of the next step, and need to compute the z-depth of the current ray position during the traversal. The traversal starts at the first valid pixel (P_{2D} in Figure 5), and finishes when finds an intersection or the last visited pixel (Q_{2D} in Figure 5) is out of screen. The pseudo code of this algorithm is shown in Figure **6**.

Algorithm 2 - Depth Map Ray Traversal

```
1: procedure depthMapRayTraversal(Ray secondaryRay, Real intersectionDistance)
2:   Point p1 = secondaryRay.origin;
3:   Point p2 = secondaryRay.origin + secondaryRay.direction * camera.far;
4:   projectPointToScreenSpace(p1);
5:   projectPointToScreenSpace(p2);
6:   if(p1.x > p2.x)
7:     swap(p1,p2);
8:   end if
9:   (p1,p2) = clipToPlane(camera,p1,p2);
10:  Real m = (p2.y-p1.y)/(p2.x-p1.x);
11:  Real deltaY = p2.y - p1.y;
12:  Real deltaZ = p2.invZ() - p1.invZ();
13:  Real zFactor = deltaZ / deltaY;
14:  for(Integer x = p1.x;x ≤ p2.x and x < camera.width;x++)
15:    Real y = p1.y + (m*(x - p1.x));
16:    Integer yScreen = floor(y + 0.5);
17:    DepthPixel dPixel = (x,yScreen);
18:    Intersection intersection = checkIntersectionWithDepthPixel(secondaryRay,dPixel,p1,p2,zFactor);
19:    if(intersection.type == hit and intersection.t < intersectionDistance)
20:      intersectionDistance = intersection.t;
21:      return;
22:    end if
23: end procedure
```

Figure 6. Depth map ray traversal algorithm.

C. *Ambient Environment Map*

To simulate an ambient environment mapping effect using RT^2AR, we just need to create a bounding sphere around the entire scene. This sphere is mapped with the previously generated texture from the ambient. So, when a virtual or real object is reflective, we generate the secondary reflection ray, and calculate its traversal, as explained in the previous subsection. If this ray intersects a virtual or a ghost object we do the reflection normally; otherwise, the ray goes to the space and will intersect the bounding sphere. Thus, the whole environment mapping process is reduced to the simple act of ray trace reflections.

D. *Refraction*

For refraction simulation, every object material was defined with a refraction factor for each color channel, a global factor that attenuates the channel factors and a refraction index to simulate the Snell law [26]. In this work, the refraction effect was done using two approaches: planar refraction, where the environment's current refraction index is always constant and the ray deviates when intercepts an object's surface plane, and volumetric refraction, where the environment's current refraction may change depending on the ray's current medium location, such as a secondary ray inside of an object, for example.

The RT^{2}AR creates refraction and reflection using the same set of techniques. The only difference resides in secondary ray generation, when distinct ray transformations are used to create the refracted or reflected rays. This similarity to reflection is another advantage of RT^{2}AR, since other approaches require treating refractions and reflections with a different set of techniques, turning the process more complex.

E. Shadowing

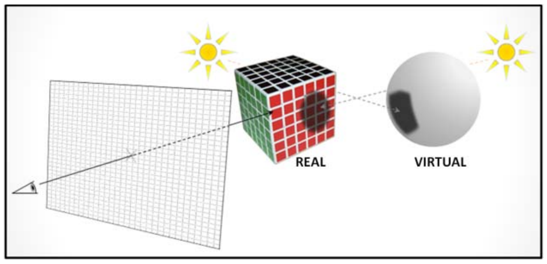


Figure 7. Real and virtual objects shadowing.

Shadows provide an important depth perception. RT^{2}AR simulates shadows in a similar way to reflections. The basic difference is that instead of creating a reflection ray, we create a light ray in the shading stage, which starts in the object position and goes in direction to the lights, giving a shadow attenuation factor (color subtraction) if the ray intersects an object (see Figure 7). As with reflection, we may have three kinds of situation: virtual objects generating shadows on other virtual objects, real objects generating shadows on virtual objects, and virtual objects generating shadows on real ones. It's important to say that in our pipeline, the shadow shading process is part of the object shading. When the pipeline is shading an object (particularly in diffuse and specular components), we cast a shadow ray for each light, to get the shadow contribution for all lights.

According to Figure 14, first the primary ray is generated and the traversal of this ray is done. The ray intersects an object and the custom shaders of this object are executed. During the shade, a ray to each light is cast, so when these rays intersect an object, the light contribution is subtracted (shadow) and the pixel is realistically rendered as it had a shadow.

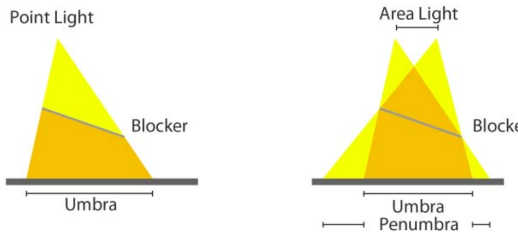


Figure 8. Hard (left) and soft (right) shadow effects.

Hard shadow techniques are capable of rendering shadows that have only one dark region, called umbra, while soft shadow approaches try to synthetize shadows with two distinct types of regions, umbra and penumbra, as shown in Figure 8. The penumbra region ranges from a maximal to a minimal value of darkness.

We developed a path tracing based [27] algorithm to cast soft shadows. Firstly, instead of using point lights, we considered the lights as spherical lights (see Figure 9). Therefore, one light has many point lights distributed over the surface of the sphere. The position of the light gives us the center of the sphere and is our first sample of light. The other samples are generated as a uniform distribution of the sphere's surface region.

Having these samples, when we are shading the shadow, we cast a shadow ray for each sample of light. If the ray intersects an object, this sample is a shadow sample, and otherwise it's a light sample. After that, we have the amount of samples that are light or shadow, so we get the percentage of shadow attenuation of this light for this pixel. It's important to say that more samples provide better visual effects with the drawback of higher computational cost and therefore less performance.

To obtain higher performance, we developed an approach to minimize the number of casted shadows. The main idea is to take four extreme points from the visible semi sphere. Verifying the shadow ray cast to the center point of the sphere we can obtain the plane normal that cuts the sphere into two hemi spheres. With the plane normal we can get the vectors $\vec{\boldsymbol{v}}$ and $\vec{\boldsymbol{u}}$ orthogonal to the plane normal. So, using the normal vector, $\vec{\boldsymbol{u}}$ and $\vec{\boldsymbol{v}}$, we define five initial samples (including the center point) to cast the shadow rays, as shown in Figure 9.

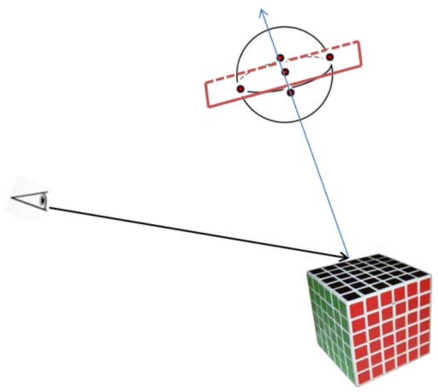


Figure 9. Optimized soft shadow technique using 5 initial samples.

After casting the shadow ray to each of these five samples, we are able to know if they are light or shadow samples. If all of them are light samples, we don't have shadows; otherwise if all of them are shadow samples we have hard shadow at this point. In the case some of them are light and the others are shadow samples, we have a soft shadow pixel. In this case, we cast a shadow ray to all other samples to get the soft shadow factor of that pixel. Due to the fact that we simulate multiple point lights for one spherical light, the result gives the perception of multiple lights illuminating the scene. To smooth even more the gradient between pixels in soft shadows, a mean filter is applied to these pixels in a post-processing step of the RT^2 pipeline (see Figure 2).

F. Lens Effects

To enhance the visual quality of rendered images, two lens effects were added to RT^{2}AR, in the post-processing step of the RT^2 presented in Figure 2, which are bloom and glare. The implementations of both effects were based on the work of Pessoa et al. [13]. The bloom camera-effect occurs when an object is directly observed against a light source and a phenomenon occurs which is the light exceeding the object edge bounds, obfuscating the observer. The effect implementation consists of post processing passes in the pipeline. First, the luminance map with a certain threshold is extracted from the final scene and successive Gaussian filters are applied to the luminance map. An optimization of the post processing passes is to downscale the luminance map image to reduce the operations amount on GPU and apply a bilinear access on the filtered image and sum the color, with a weight to the corresponding pixel.

The glare camera-effect is an effect which occurs due to a phenomenon where light sources or intense reflections cause light rays scattering. It's perceived as a set of radial edges around the light source. This implementation also used the post processing approach where is extracted a luminance map with a threshold little higher than used with the bloom effect and it's used as the algorithm input. The algorithm consists in a number of passes n and a number of samples s to generate a radial ray where each pixel will be affected by the neighbor's pixels according to the following equation:

$$Image(x, y) = \big(x + (b * s), y + (b * s)\big), \quad (1)$$

where b is the multiplication factor calculated as follows:

$$b = 4^{(n-1)}. \quad (2)$$

Each neighbor pixel that sums with the currently processed pixel will be affected by a weight which is defined according to the following equation:

$$weight(s) = a^{(b*s)}, \quad (3)$$

where a is the attenuation factor that varies between 0.85 and 0.95. The Equation 1 exemplifies a ray generation in a specific direction. For more ray directions generation we simply invert the signals that sum x and/or y.

V. RESULTS

This section shows the visual results achieved using our novel pipeline. Since real time performance is an essential requirement, we present timing results as well. To assist the analysis, different scenarios were created as case study, showing the influence of each visual effect implemented.

The results were collected using an Intel Core i7 3.06 GHz CPU with 12GB of RAM and a NVIDIA GeForce GTX 580, running Microsoft Windows 7 Professional 64-bit and using CUDA 4.1 toolkit. Two RGB cameras of different resolutions were used: a 1280x720 pixels webcam and Kinect's 640x480 pixels camera.

With the intent to ease comparisons between previous and future related publications, the 3D models used in the tests were taken from public repositories, such as the Dragon model (1.1 million triangles) from the Stanford 3D Scanning Repository [28], the Lamborghini model (500k triangles) taken from the Scifi-Meshes repository [29], and generic geometric forms, as boxes and spheres, and a standard teapot model with 14k triangles.

A. Visual Effects on Virtual and Tracked Real Objects

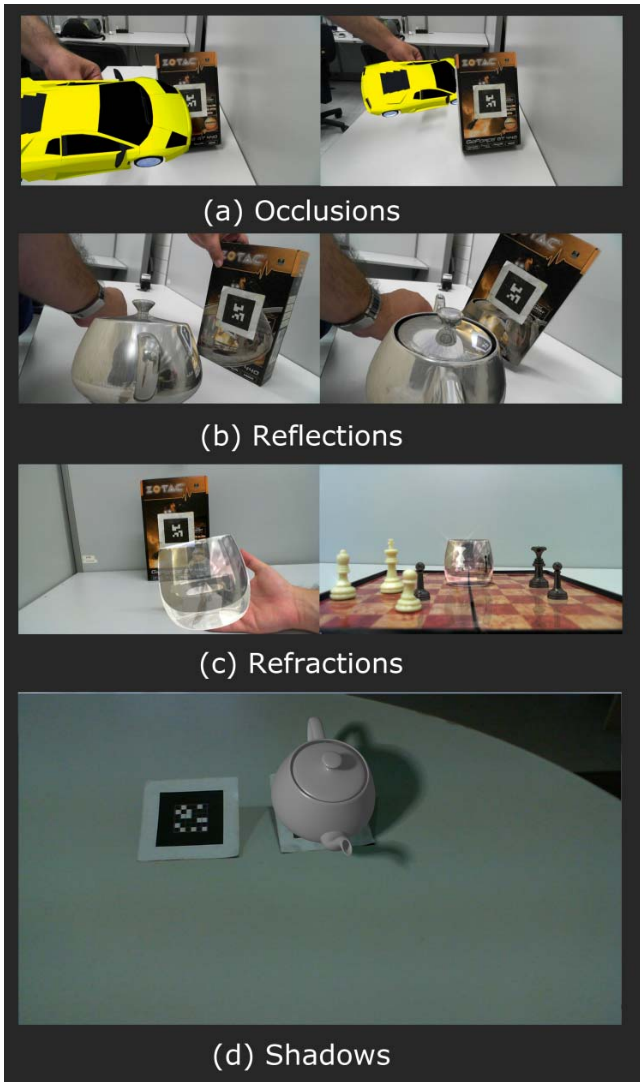


Figure 10. Visual effects on virtual and tracked real objects.

Figure 10 shows the visual effects supported by our pipeline applied to virtual and real objects tracked by markers. The two first images display occlusion cases occurring between two different entities: the virtual Lamborghini and a real box with a marker. It's important to notice the visually correct silhouette between these objects, a consequence of the correct usage of camera's intrinsic configuration on RT^2 virtual camera.

The next images (Figure 10b) show reflection effects between the real box and a virtual teapot, since both objects have reflective materials. However, the teapot has a metallic material, while the box a reflectance map that defines no reflections over the marker region, since the real paper marker doesn't have reflective properties. Thus, the final result is a correct absence of reflections on the pixels that show the marker's surface. It's important to notice that the RT^{2}AR pipeline is also capable of creating recursive inter-reflections, such as the visualization of the box inside the reflection of the teapot on the box itself. Furthermore, RT^{2}AR is capable of self-reflections, such as those present on the teapot's surface. Those are important features of our novel pipeline, since over a rasterization pipeline they usually require expensive complex techniques [13] with multiple passes.

Refraction effects are shown in Figure 10c. The left image exhibits a virtual glass refracting the box, deflecting its natural silhouette. The right image shows a chessboard with some chess pieces. The white pieces are real objects, while the black ones are virtual. The chessboard is reflecting both kinds of objects, and the virtual glass surface reflects the pieces and refracts real and virtual objects behind it. The image also shows the results of the bloom and glare lens effects, visible on the top of the glass and on the leftmost chess piece.

Finally, Figure 10d shows the visual result of soft shadows created by a virtual teapot over a real table. The shadowing effect has a high quality result, being one important feature of the RT^{2}AR pipeline.

B. *Visual Effects on Virtual and Depth-Mapped Real Objects*

The Figure 11 shows all obtained effects using the Kinect depth map information, avoiding the need for previously modeled versions of the real objects. The two first images represent occlusion cases between a virtual teapot object and real objects (such as the box), represented by the depth information from each pixel. On the left image, some artifacts are visible, caused by the imprecision of depth information provided by Kinect on silhouettes of objects. However, most pixels have a correct information, allowing the removal of markers to achieve occlusion effects.

Reflections and refractions effects using Kinect are shown in Figure 11b and c. Although the fiducial marker on the box (it was not used for tracking), only depth data were considered to achieve the simulation. Refractions showed a more compelling visual result than reflections, since it usually converges objects in foreground, available as depth information from Kinect. On the other hand, reflections usually require information coming from the surrounding environment. This information is not available, since the depth map came from a planar, non-fisheye camera.

Shadowing results are shown in Figure 11d. The top image shows a synthetized shadow of the real box over a virtual sphere. The bottom images show the virtual Stanford Dragon casting shadows over the wall and a box. Notice the correct distortion of the shadow projection over the box in relation to the wall.

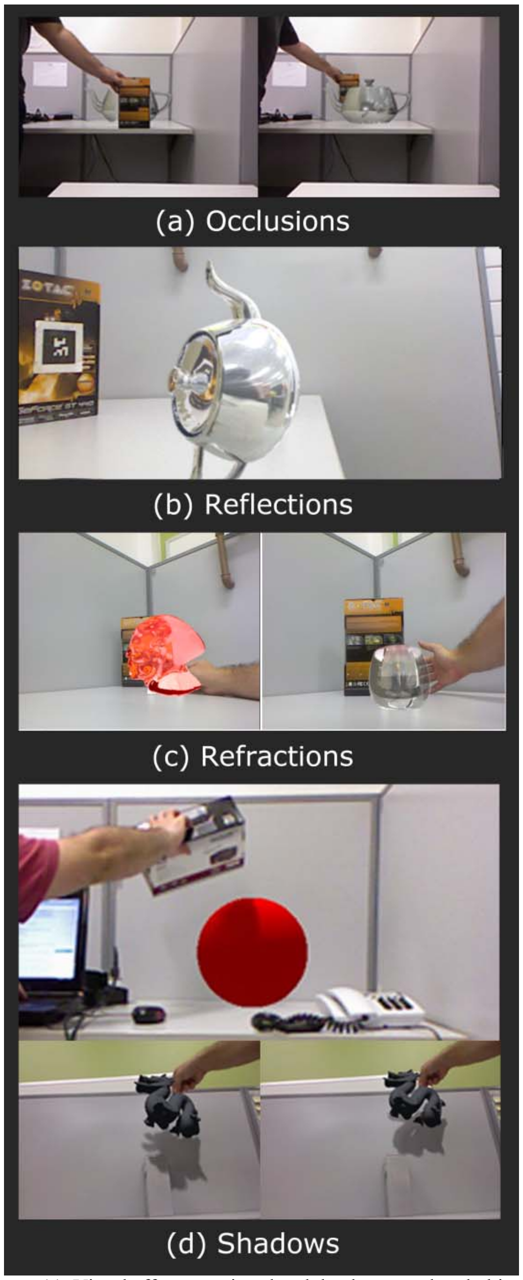


Figure 11. Visual effects on virtual and depth-mapped real objects.

C. *Shadows Implementations*

This subsection shows some visual and numeric results of the different shadow techniques supported by the RT2AR. Figure 12a shows the scene rendered at 108 FPS without shadows, while the Figure 12b shows a scene rendered at 84 FPS with hard shadows, showing a drop of approximately 28% in performance caused by the addition of hard shadow. The Figure 12c shows a path traced soft shadow with 20 samples per pixel without any kind of optimization, leading to a poor 7 FPS performance. Figure 12d shows our optimized soft shadow technique, reaching 31 FPS for a maximum of 20 samples per pixel. The images of Figure 12e show our optimized soft shadow algorithm plus the smooth filtering, achieving performance of 30 FPS for 20 samples and 20 FPS for 50 samples. Therefore, our optimized algorithm delivers high quality soft shadows with real-time performance.

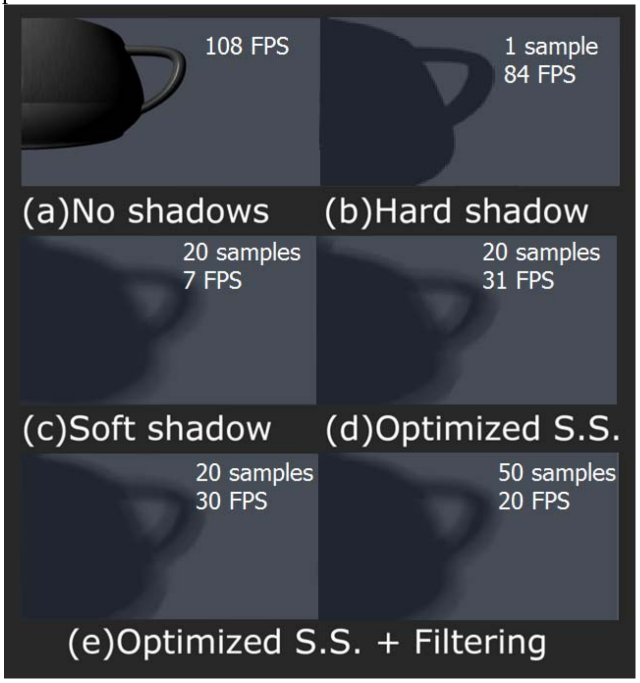
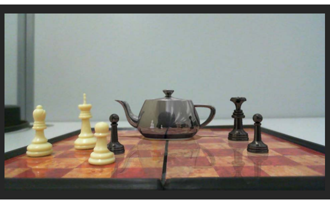


Figure 12. Shadow visual and performance results.

D. *Performance Measurements*

Table 1. Performance results of the RT^{2}AR pipeline.



	Primary ray only	3 secondary rays	5 secondary rays
Without shadows	74 FPS	60 FPS	60 FPS
With shadows	40 FPS	32 FPS	32 FPS

The Table 1 shows performance results of the chessboard scenario with a reflective teapot. Without shadows and obtaining high quality reflection effects it had a drop of performance of 19%, from 74 FPS to 60 FPS. Beyond the high visual quality, this is an important result, since rasterized techniques [13] show much higher performance penalties when the number of objects increases, while our pipeline doesn't have this problem. It's important to notice that for this scenario a maximum of 5 secondary rays is equivalent to 3 secondary rays, meaning that all the reflection and refraction effects finished at a traced ray of fourth level. Adding our optimized soft shadow algorithm increased by approximately 54% the rendering costs. Since it's an advanced effect, it's an acceptable performance loss if the objective is obtaining real time photorealistic effects.

Figure 13 shows two images with a different number of primitives. The left image shows a teapot with 14280 triangles while the right one exhibits two Stanford Dragons with 1.1 million triangles each, plus a teapot, giving the approximated amount of 2.2 million triangles, 153 times more than the first image. The left image was rendered at 95 FPS and the right one at 112 FPS. Both for rasterized and ray-traced based techniques, the image on the right would be rendered at much slower frame rate than the left one, since these techniques are usually linear in number of primitives. However, ray tracing performance with an optimized acceleration structure, which we used on the proposed work, is more dependent on the number of shaded pixels. This is a key feature of ray tracing for augmented reality, since in many cases the pixels will be rendered from the RGB camera, and therefore only some will have a high ray tracing cost.

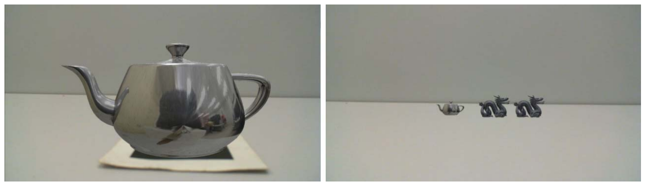


Figure 13. Primitives vs Pixel performance.

VI. Conclusion and Future Work

RT^{2}AR showed itself as a promising hybrid pipeline for real time near-photorealistic rendering of augmented reality scenes. The visual effects between virtual and real objects supported by this pipeline are: occlusions, reflections (including inter and self-reflections), refractions and shadowing. Although soft shadowing shows severe performance penalty, our optimized technique proposed shows high quality real time results.

There are some improvements that can be done in our pipeline, such as: ambient occlusion considering real and virtual objects; depth of field using kinect's depth information; normal surface estimation for real objects using Kinect, allowing generation of secondary rays from depth data.

Acknowledgment

The authors would like to thank CNPq and Propesq UFPE for partially funding this research project.

References

[1] A. Appel, "Some techniques for shading machine renderings of solids," in Proceedings of the April 30–May 2, 1968, spring joint computer conference, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 37–45.

[2] NVIDIA, NVIDIA CUDA Programming 4.1, 2012. [Online]. Available: http://www.nvidia.com/object/cuda

[3] A. Santos, J. Teixeira, T. Farias, V. Teichrieb, and J. Kelner, “Understanding the efficiency of kd-tree ray traversal techniques over a gpgpu architecture,” International Journal of Parallel Programming, pp. 1–22, 10.1007/s10766- 011-0186-1.

[4] A. L. dos Santos, J. M. X. N. Teixeira, T. S. M. C. de Farias, V. Teichrieb, and J. Kelner, “kd-tree traversal implementations for ray tracing on massive multiprocessors: A comparative study,” in Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing, ser. SBAC-PAD ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 41–48.

[5] T. Aila and S. Laine, “Understanding the efficiency of ray traversal on gpus,” in Proc. High-Performance Graphics 2009, 2009, pp. 145–149.

[6] A. Fournier, A. S. Gunawan, and C. Romanzin, “Common illumination between real and computer generated scenes,” in Proceedings of Graphics Interface ’93, Toronto, ON, Canada, May 1993, pp. 254–262.

[7] S. Gibson and A. Murta, “Interactive rendering with realworld illumination,” in Proceedings of the Eurographics Workshop on Rendering Techniques 2000. London, UK, UK: Springer-Verlag, 2000, pp. 365–376.

[8] K. Jacobs, J.-D. Nahmias, C. Angus, A. Reche, C. Loscos, and A. Steed, “Automatic generation of consistent shadows for augmented reality,” in Proceedings of Graphics Interface 2005, ser. GI ’05. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2005, pp. 113–120.

[9] I. Sato, Y. Sato, and K. Ikeuchi, “Acquiring a radiance distribution to superimpose virtual objects onto a real scene,” IEEE Transactions on Visualization and Computer Graphics, vol. 5, no. 1, pp. 1–12, 1999.

[10] P. Debevec, “A median cut algorithm for light probe sampling,” in ACM SIGGRAPH 2008 classes, ser. SIGGRAPH ’08. New York, NY, USA: ACM, 2008, pp. 33:1–33:3.

[11] W. Heidrich and H.-P. Seidel, “Realistic, hardware accelerated shading and lighting,” in Proceedings of the 26th annual conference on Computer graphics and interactive techniques, ser. SIGGRAPH ’99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 171–178.

[12] Y. Uranishi, A. Ihara, H. Sasaki, Y. Manabe, and K. Chihara, “Real-time representation of inter-reflection for cubic marker,” in Proceedings of the 2009 8th IEEE International Symposium on Mixed and Augmented Reality, ser. ISMAR ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 217–218.

[13] S. A. Pessoa, G. d. S. Moura, J. P. S. d. M. Lima, V. Teichrieb, and J. Kelner, “Virtual reality in brazil 2011: Rpr-sors: Real-time photorealistic rendering of synthetic objects into real scenes,” Comput. Graph., vol. 36, no. 2, pp. 50–69, Apr. 2012.

[14] F. Scheer, O. Abert, and S. Müller, "Towards Using Realistic Ray Tracing in Augmented Reality Applications with Natural Lighting" 4º workshop "Virtuelle und erweiterte realität" der gi-fachgruppe VR/AR 15 July 2007 (2007-07-15).

[15] T. Franke, S. Kahn, M. Olbrich, and Y. Jung, “Enhancing realism of mixed reality applications through real-time depth-imaging devices in x3d,” in Proceedings of the 16th International Conference on 3D Web Technology, ser. Web3D ’11. New York, NY, USA: ACM, 2011, pp. 71–79.

[16] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon, “Kinectfusion: Real-time dense surface mapping and tracking,” in Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality, ser. ISMAR ’11. Washington, USA.

[17] P. Debevec, “Rendering synthetic objects into real scenes:bridging traditional and image-based graphics with global illumination and high dynamic range photography,” in ACM SIGGRAPH 2008 classes, ser. SIGGRAPH ’08. New York, NY, USA: ACM, 2008, pp. 32:1–32:10.

[18] D. Lemos, “Um Pipeline para Renderização Fotorrealística de Tempo Real com Ray Tracing para Realidade Aumentada”, Master dissertation, Federal University of Pernambuco, 2012.

[19] C. Wächter and A. Keller, “Instant ray tracing: The bounding interval hierarchy,” in Eurographics Workshop/ Symposium on Rendering, T. Akenine-M¨oller and W. Heidrich, Eds. Nicosia, Cyprus: Eurographics Association, 2006, pp. 139–149.

[20] T. L. Kay and J. T. Kajiya, “Ray tracing complex scenes,” in Proceedings of the 13th annual conference on Computer graphics and interactive techniques, ser. SIGGRAPH ’86. New York, NY, USA: ACM, 1986, pp. 269–278.

[21] J. M. X. Teixeira, E. S. Albuquerque, A. L. dos Santos, V. Teichrieb, and J. Kelner, “Improving ray tracing antialiasing performance through image gradient analysis,” Computing Systems, Symposium on, vol. 0, pp. 144–151,2010.

[22] Microsoft Corporation, “Kinect SDK for Windows,” 2012.

[23] G. Bradski, “The OpenCV Library,” 2012.

[24] R. M. Freeman, S. J. Julier, and A. J. Steed, “Augmented Reality Toolkit Plus (ArtoolKitPlus),” 2012.

[25] J. E. Bresenham, “Algorithm for computer control of a digital plotter,” IBM Syst. J., vol. 4, no. 1, pp. 25–30, Mar. 1965.

[26] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, Computer graphics (2nd ed. in C): principles and practice. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[27] J. T. Kajiya, “The rendering equation,” in Proceedings of the 13th annual conference on Computer graphics and interactive techniques, ser. SIGGRAPH ’86. New York, USA: ACM, 1986, pp. 143–150.

[28] “The stanford 3d scanning repository.” [Online]. Available: http://graphics.stanford.edu/data/3Dscanrep/

[29] “Scifi-meshes.” [Online]. Available: http://www.scifi-meshes.com/forums/

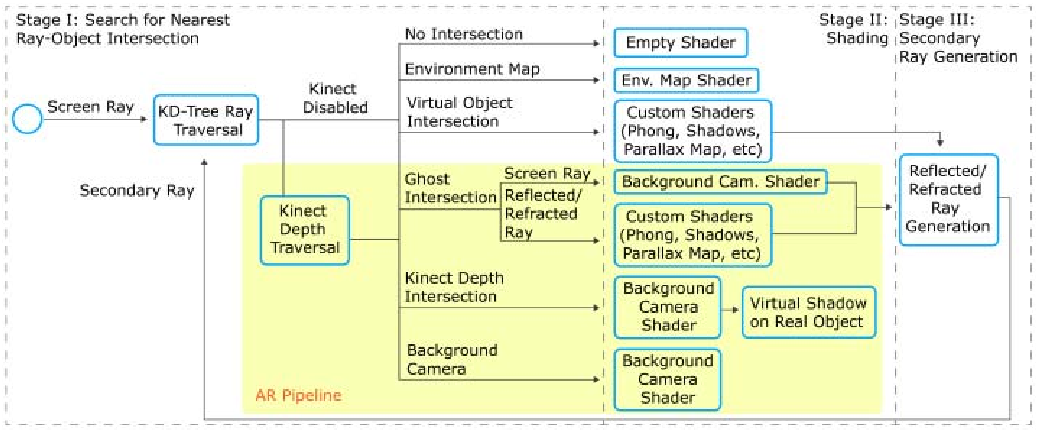


Figure 14. RT^{2}AR pipeline. Yellow rectangle represents AR sub-pipeline.