FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Merging Ray Tracing and Rasterization in Mixed Reality

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Adam Celarek
Matrikelnummer 0926881

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Mag.rer.soc.oec. Martin Knecht, Bakk.techn.

Wien, 20.11.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Merging Ray Tracing and Rasterization in Mixed Reality

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Adam Celarek
Registration Number 0926881

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Mag.rer.soc.oec. Martin Knecht, Bakk.techn.

Vienna, 20.11.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Adam Celarek
Talererstrasse 4, 6322 Kirchbichl

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

In mixed reality, virtual objects are inserted into a video stream of a real environment. This technique can be used for many applications including marketing, simulations and cultural heritage. Therefore it is important that the images look plausible. Many applications also have real time constraints.

With traditional rasterization it is difficult to create realistic reflections and refractions. In ray tracing on the other hand this is a trivial task, but rendering is slow. The solution described in this work uses the graphics card for speeding up ray tracing. Additionally it employs a rasterizer for diffuse surfaces and only traces rays if there is a reflective or refractive surface visible.

This works by creating a ray tracing mask using the fast rasterizer in a first step. It holds true for reflective or refractive surfaces and false otherwise. Then all diffuse objects are drawn using the rasterizer. Finally rays are traced on each pixel which is masked as reflective or refractive surface by the ray tracing mask. These rays produce secondary rays which can hit a diffuse surface eventually. In this case the ray tracer takes over the shading.

Results show, that our hybrid rendering method allows high quality reflections and refractions while still having interactive frame rates in mixed reality scenarios.

Contents

CHAPTER 1

Introduction

In mixed reality virtual objects are placed into video streams and then the result is presented to the viewer. Possible fields of application include visualisations for product marketing or museums and simulations for any kind of training. The goal is to make it as realistic as possible. Ideally the viewer would not discover any difference between virtual objects and real ones. However, there are many challenges to achieve that:

- Lighting and shadows
  Virtual objects need to be illuminated in a similar way as the real ones. This is difficult to achieve, not only because of rendering limitations, but also because it is hard, if not impossible, to get a complete model of the real lighting situation. Good approximations can be achieved by cameras recording the environment also known as Inside-Out methods [23] [14], photographs of chrome spheres known as Outside-In methods [1] [5], or annotations made by the user [11].

- Geometry and materials
  Information about the real geometry and its materials is needed. As with light, a complete model is not possible until now. Usually simplified versions of the real geometries and materials are used in conjunction with differential rendering. This achieves good approximations [5]. In this way, good approximations can be achieved.

- Position of the camera and other objects
  To place the virtual objects in the real scene, the position of the camera must be known. There are mainly two tracking methods, marker and SLAM (Simultaneous Localization and Mapping) based tracking. Both have the problem of jitter and naturally it is possible to loose the position. Printed 2d BCH markers need to be placed in the scene and therefore will be visible. SLAM based approaches try to extract the feature points from the camera image [4]to track the movement of the camera.

- Missing information
  The challenges in the first three points lead to missing informations. This is especially true for reflecting and refracting surfaces.

- Realistic rendering of virtual objects
  Real time global illumination algorithms are needed to render virtual objects in a plausible way. Only for still images off-line rendering can be used [7].

1.1 Problem statement

In mixed reality diffuse and specular objects need to be displayed. For diffuse surfaces a traditional rasterizer is fast and produces good results. But for specular objects like mirrors or glass it is complicated to implement plausible effects. Especially when it comes to multiple reflections or refractions through more than one object, a raster implementation will get more and more complex. A ray tracer is slower but refractions and reflections are more realistic and the implementation is very straight forward by using recursion.

The goal of this thesis is to implement reflective and refractive effects using a real-time ray tracer and merge the outcome with the result produced by the rasterizer. In this way the ray tracer will only need to render the specular surfaces. Both, ray tracing and rasterization will be done on the graphics card because we want to achieve real-time frame rates.

We will evaluate the performance of our method compared to

- not rendering specular objects at all,
- rendering everything with ray tracing
- and rendering with an existing rasterizer.

The image quality will also be compared to the existing rasterization implementation.

CHAPTER 2

Related Work

2.1 Differential Rendering

A challenge in mixed reality is to blend the virtual objects into the real camera image. Real lights should cast light on the virtual objects, as well as virtual objects should cast shadows onto real ones. This is often done by differential rendering described by Debevec [5]. Two images are rendered, one containing illumination effects of real and virtual (I_m) objects and the other containing only illumination effects of real ones (I_r). Composition is done by adding the difference of real and mixed rendered image to the camera image I_c, shown in Equation 2.1.

$$I_f = I_c + I_m - I_r \tag{2.1}$$

2.2 Instant Radiosity

Instant radiosity [13] uses Virtual Point Lights (VPLs) to approximate global illumination in virtual reality. These VPLs are placed at surfaces which are exposed to incoming light, introducing one indirect light bounce. Since it is too time consuming to calculate standard shadow maps for each VPL every frame, a method called imperfect shadow maps (ISM) introduced by Ritschel et al. [22] is used. This allows to create hundreds of ISMs at real-time frame rates.

2.3 Differential Instant Radiosity

Knecht et al. [14] combines instant radiosity and differential rendering to achieve plausible lighting effects in a mixed reality scenario. A camera with fish-eye lens is used to capture the environment and lighting information is extracted from the image. Their framework is the basis for this work and will be described more thoroughly in Chapter 5.

2.4 Ray Tracing

Whitted et al. [25] described how to trace rays from the camera backwards into a 3d graphics scene. When the ray hits a surface, the point will be shaded by computing terms for reflection, refraction, diffuse lights and an ambient term. In case of reflection or refraction secondary rays will be sent in reflection and refraction direction, respectively, calling the shading algorithm recursively. The diffuse lighting term sends rays towards all light sources. If the ray hits an object, the light source is obscured by that object and therefore the point lies in a shadow.

The simplest method for finding the intersection point of a ray and an object would be to iterate over all objects and their triangles and check, if the ray intersects with the triangle. Since this is computational intensive, more sophisticated algorithms have to be used. Kay and Kajiya [12] described a method called bounding volume tree and Foley and Sugerman [6] described kd-tree acceleration structures for GPU ray-tracers. Both methods speed up computation time dramatically.

To speed up ray tracing, parallel computation can be used. With modern processors having multiple cores it is possible to achieve real-time frame rates on single workstations [2]. Modern graphics cards have a specialized and even more parallel architecture. Since they are programmable for instance with CUDA [18] they are predestined to speed up ray tracing. NVIDIA implemented specifically for this purpose the framework OptiX [19] which is utilised in this work.

Differential Rendering can also be applied to ray tracing. Kán and Kaufmann. [9] found a method to avoid doing the expensive intersection test two times for mixed and real only image. Every ray has a type (mixed or real) and a payload (mixed colour and real colour). The camera shoots only mixed rays, if a mixed ray hits a real object both colour values will be filled, if it hits a virtual object only the virtual colour will be filled and the ray continues as a real one. Real rays can only hit real objects. Shadow rays work similar. They have a mixed or real type and two attenuation values in their payload. Real shadow rays are emitted only from real rays and do not consider virtual objects.

2.5 Differential Photon Mapping

Grosch [7] described a method of adding virtual objects without differential rendering. He uses photon maps [8] that can save positive and negative light. Negative light represents shadows from virtual objects and positive light caustics or colour bleeding. Normal ray tracing is used to get the final image. If the ray hits a real diffuse surface, positive and negative photons are added to the camera image value. Virtual diffuse surfaces are rendered normally by using the gathered photons. Reflective or refractive objects produce secondary rays that are traced until they hit a point of a diffuse surface. If the point belongs to a virtual object it is rendered normally. Otherwise the algorithm tries to get a back projection of the pixel in the camera image and again adds positive and negative photons. If that fails, an estimation of the real surface or the environment map is used. The resulting images are very realistic but it is an off-line approach.

CHAPTER 3

The OptiX Framework

Ray tracing can produce very realistic computer generated images but has high computation costs. For a very long time this computation was done on the CPU. With modern multiprocessor and multi-core systems it is possible to achieve interactive frame rates [2]. However modern GPUs with their parallel architecture outpace CPUs in cases where parallel programming can be used [18], which is the case for ray tracing.

OptiX is a framework for ray tracing on the graphics card based on CUDA. The user supplies only ray generating, primitive intersection testing and shading programs as well as all needed data. OptiX implements control of the computation threads, data copy between graphics card and host, import of buffers and textures from Direct3D. Several speed up algorithms for intersection search and auxiliary libraries, for instance vector maths or refraction and reflection functions are also included.

3.1 CUDA

With the start of programmable GPUs early attempts were made to use them for general computations, however this was still very restricted as the graphics pipeline had to be used [16] [18]. The basic work flow of these systems was as follows:

1. upload input data to the graphics hardware as a texture
2. upload a fragment shader program doing the computation
3. render a screen quad into a buffer using this fragment shader
4. retrieve the result by reading back the buffer into host memory

More general frameworks were invented, they abstracted the graphics pipeline to provide a more convenient programming model [3]. Graphics card manufacturers saw the trend towards general purpose GPUs (GPGPU) and supported the development. Several GPGPU programming

systems were released [18]. One of them is CUDA, a C like high level language developed by NVIDIA. There is also a non-proprietary alternative called OpenCL which is very similar to CUDA. However it was shown, that CUDA performs better than OpenCL on appropriate hardware [10].

Parallelism in CUDA is implemented by running many parallel threads. The code run by one thread is called a kernel. It has a thread id which can have up to 3 dimensions. This thread id is used to select the domain in the data that the current thread is processing and in the output to prevent write conflicts between threads. Threads are organised into blocks which can also have up to 3 dimensions. All threads in one block will be executed on one GPU core. Due to hardware restrictions of such a core the number of threads per block is limited. The exact limit depends on the hardware generation, but a number of 256 threads is common.

Blocks can be executed in any order or parallel on several cores, so the data must be independent. There are mechanisms to sync threads and share memory inside blocks. The memory is organised in a hierarchy with per-thread local memory, per-block shared memory and global memory [17].

CUDA defines only a few extensions to the C language for defining the kernels. The host code is interleaved with the device code, Listing 3.1 shows an example. Two vectors with N elements are added up. Inside the device code the current thread id can be accessed by using threadIdx.

Listing 3.1: CUDA code sample taken from [17].

```
//Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    ...
    //Kernel invocation with N threads per block, 1 block used
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

3.2 OptiX

OptiX is a general purpose ray tracing framework and it is not limited to computer graphics. Other fields of application are collision detection, sound propagation and more.

The CPU only provides all needed data and the actual work is done on the graphics card. Figure 3.1 shows the work flow. Similar to OpenGL and Direct3D every stage (ray generation, intersection and shading) of the ray tracing pipeline is customizable. This is done through loading appropriate programs, similar to vertex and fragment shaders in traditional rasterization. In OptiX these are called host and device parts. The host part is a C-library but also contains a thin wrapper for C++. The kernels of the device part are programmed using CUDA.

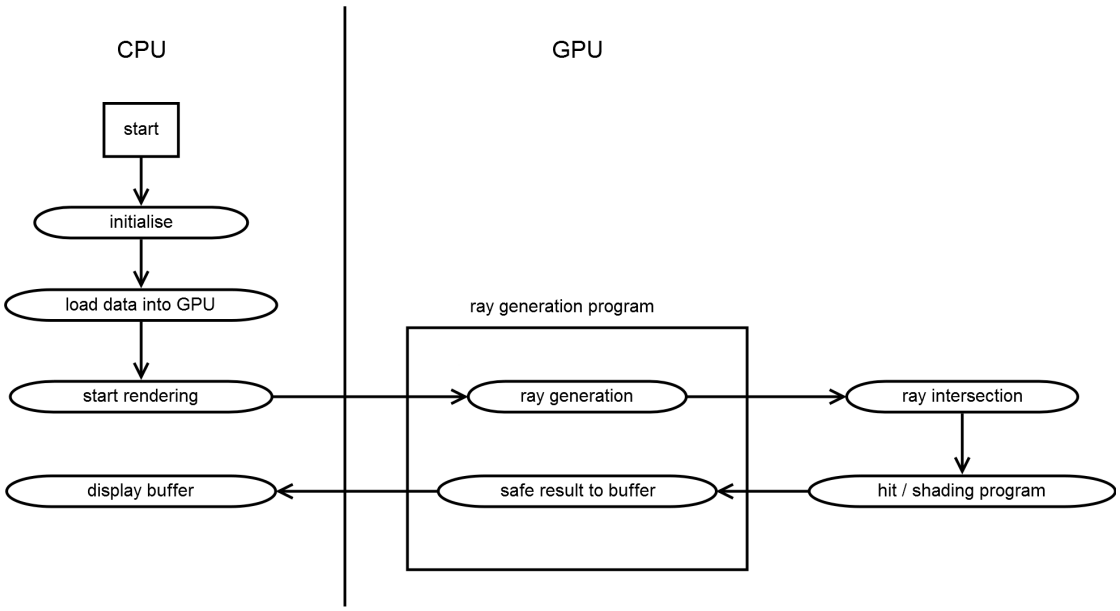

Figure 3.1: Diagram showing the work flow of an OptiX program.

Host Part

The host code needs to set up the context which includes the following points:

- create the context object
- load geometric data, create a scene graph and select an acceleration structure
- create materials
- load ray generation, intersection, hit and other programs
- load textures, materials, transformation matrices and other data

After setting up the context, objects are created and attached to the scene graph. An object consists of geometry and a material. Unlike in traditional raster graphic systems geometry is not limited to triangle meshes. It is only necessary to provide an intersection and bounding box program. Variables can be bound to the geometry, for instance vertex and index buffers, and then used in the programs. Objects can be grouped and transformed by using special nodes. Acceleration structures are attached to group and geometry nodes.

Materials are assigned to objects in the scene graph and these materials hold variables describing the properties. It is possible to have several ray types, with every type having a different program. The link between the ray type and the hit program is established within the material definition 3.2.

Listing 3.2: Setting up a material

```
optix::Material material = m_context->createMaterial();
material->setClosestHitProgram( RAYTYPE_RADIANCE, radianceProg );
material->setAnyHitProgram( RAYTYPE_ANYHITSHADOW, shadowProg );
material["diffColor"]->set3fv( diffColor );
material["specIntensity"]->setFloat( specIntensity );
material["textureSampler"]->setTextureSampler(textureSampler);
..
```

The ray generation program is attached directly to the context. After everything is set up, rendering is started by calling the launch method.

Device Programs

The first program that we describe is the ray generation program. Similar to the example in the CUDA section it can access the current launch index. Usually the launch index corresponds to the pixel on the screen but it is also possible to launch one or three dimensional contexts. With the screen position the ray generation program determines the starting point and direction of the ray and then traces it. Every ray has associated data which is saved in a customizable data structure called ray pay load. Listing 3.3 shows a simple example but usually perspective would be considered.

Listing 3.3: Example of ray generation program

```
RT_PROGRAM void rayGeneration()
{
  float2 d = (make_float2(launch_index)/make_float2(launch_dim)) * 2 - 1;
  float4 screenPt = make_float4(d.x, d.y, 0, 1);

  optix::Ray ray;
  //make_Ray(origin, direction, ray type, min, max distance)
  ray = optix::make_Ray(screenPt, make_float3(0,0,1), 0, MIN, MAX);

  PerRayData_radiance prd;
  rtTrace(top_object, ray, prd);
  output_buffer[launch_index] = prd.result;
}
```

For determining the intersection point with an object and creating an acceleration structure two programs are needed:

- Intersection with primitives, for instance with a triangle or a sphere
  When loading an object, the number of primitives must be declared. These programs get the current primitive index. For triangle meshes it would look up the triangle in the index and vertex buffer.

- Bounding box for primitives
  It calculates the bounding box for the given primitive index. This bounding box will be used for the acceleration structure.

If a ray hits a surface, the intersection program computes the normal and shading coordinates and control is forwarded to one of the two types of hit programs

- any hit
  Any of the objects on the way of the ray is hit, not necessarily the first one. It is possible to ignore the hit and finally traversing all objects on the way of the ray.

- closest hit
  All objects are tested and the closest is selected. It is not possible to ignore the hit. These programs are used for shading.

It is possible to trace secondary rays in the same way as in the ray generation program. This can be used in case of reflection and refraction.

There are also three other kinds of programs:

- Miss programs are called, if a ray does not hit an object at all. This can be used for environment maps.

- Exception programs are optional.

- Selector programs can be used for instance to implement level-of-detail.

CHAPTER 4

Method

In our method we want to use ray tracing for reflective/refractive surfaces and traditional GPU rasterization for the rest.

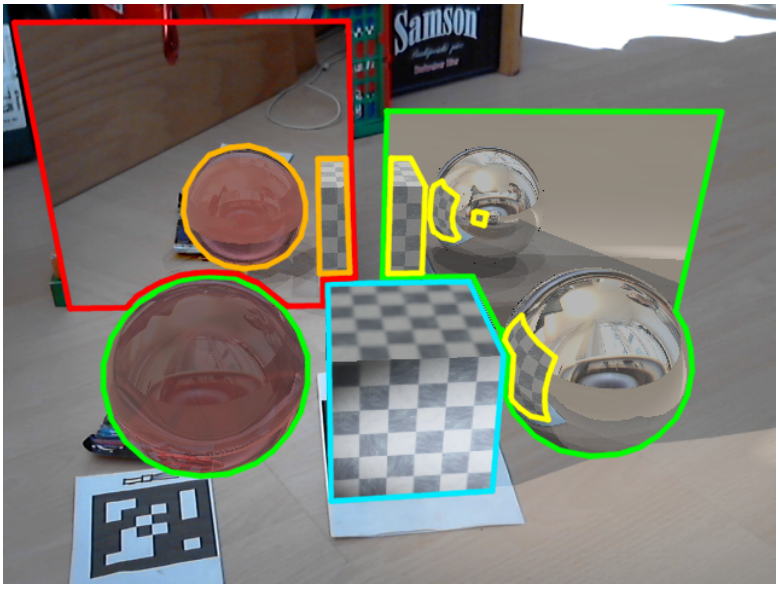

Figure 4.1: Resulting image with annotations

Figure 4.1 shows an example with annotations. It uses the following colour coding:

red = real mirror
orange = reflection of virtual objects in the real mirror
green = virtual refracting and reflecting objects
blue = diffuse virtual object rendered by the RESHADE framework
yellow = diffuse virtual object rendered by the OptiX part

Figure 4.2 shows in a simplified way, how the application works. After initialisation and position update the first thing is to find out in which part of the image the ray tracer should be used. To achieve that, we create a mask in a separate rasterization render pass which is masking all reflective or refractive surfaces. By shooting rays only for masked pixels, the primary rays will always hit reflective/refractive surfaces, but reflected or refracted secondary rays can also reach diffuse surfaces. Only in this case a diffuse surfaces will be shaded by the ray tracer and not by the rasterizer. After that the rasterizer draws all diffuse objects into a separate buffer and composition is done.

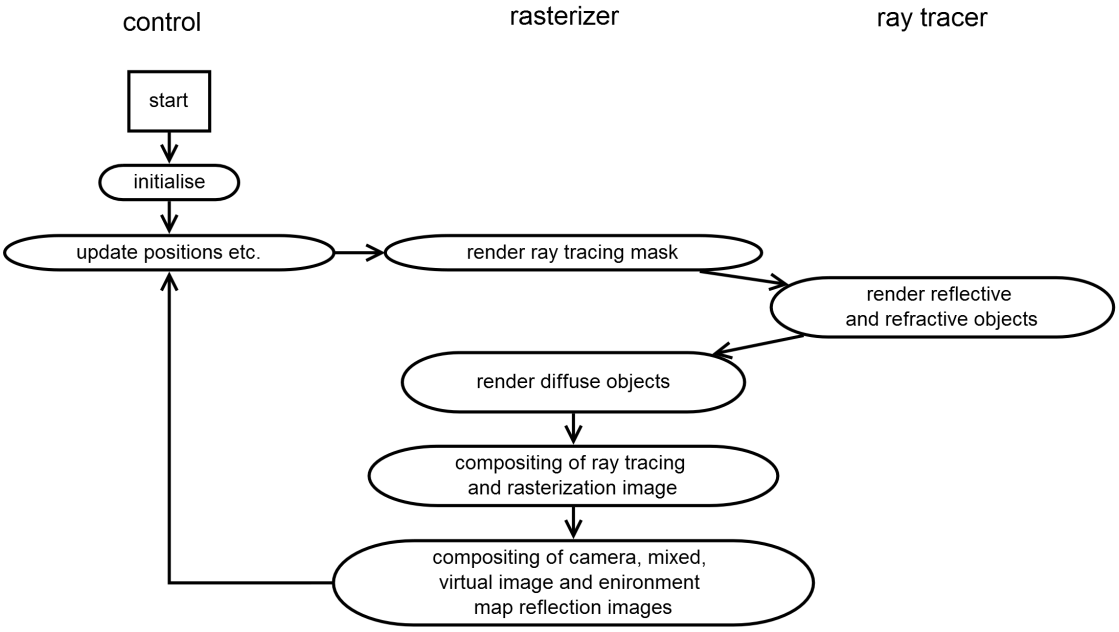

Figure 4.2: Diagram showing the basic workflow of the application.

4.1 Compositing of the Buffers

Composition is a two stage process in our method. First rasterized and ray traced buffers are merged. Then the resulting mixed and final buffers are composited together with the camera image and reflections of the environment map into the final image. Reflections of the environment map are put into a separate buffer as they shouldn't be tone mapped, see Section 4.4 for details. There is an overview of the composition process in Figure 4.3.

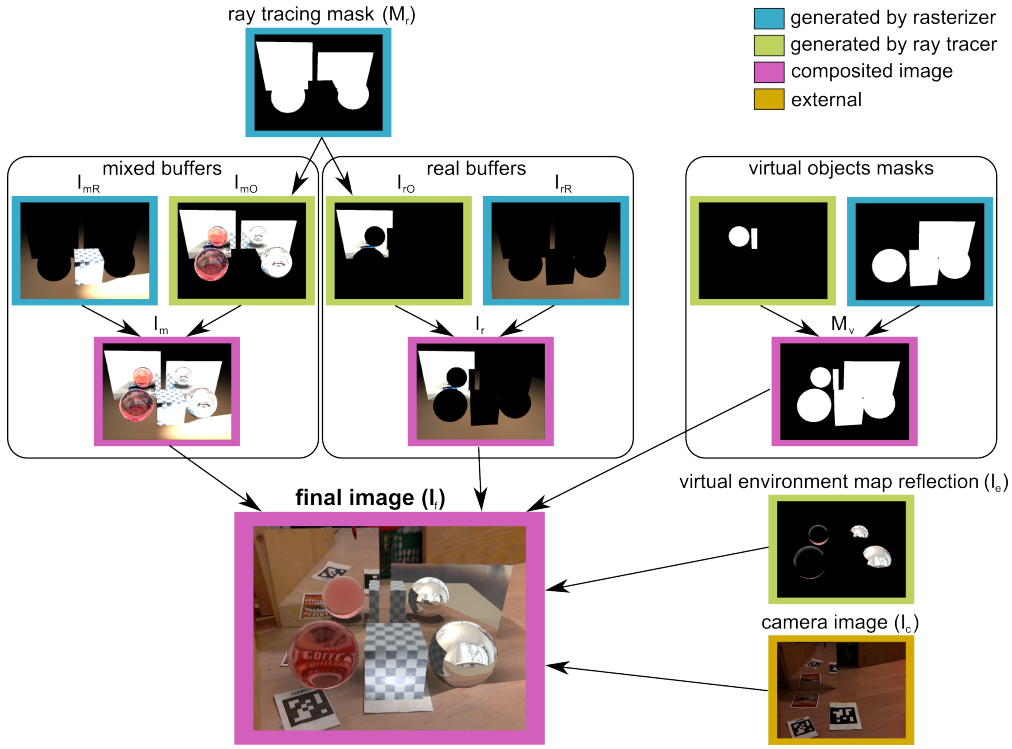

Figure 4.3: This figure shows the outline of buffer composition.

OptiX Part and RESHADE Framework Buffers

Compositing of the ray traced images and the images coming from our RESHADE framework is done by just adding them together, shown in Equations 4.1 and 4.2. The virtual environment reflections buffer does not need compositing as it can be used directly.

$$I_r = I_{rO} + I_{rR} \tag{4.1}$$

$$I_m = I_{mO} + I_{mR} \tag{4.2}$$

where
I_r and I_m are the composed real and mixed buffers used in Equation 4.5,
I_{rO} and I_{mO} are the real and mixed buffer from the ray tracer,
I_{rR} and I_{mR} are the real and mixed buffer from the rasterizer

Mixed, Real and Camera Image

The RESHADE framework uses tone mapping [14], so before the image can be merged, mixed and real buffer need to be tone mapped 4.3 4.4, indicated by $TM()$. Then the difference of mixed and real image is added to the camera image (I_c), if the virtual objects mask (M_v) shows no virtual objects. Otherwise just the mixed image is used 4.5. This is needed, because virtual object pixels should replace camera pixels. Without the mask the camera image would blend into the virtual object. The mask is created by the rasterizer in a separate render pass and extended by the ray tracer for virtual objects that are reflected in real mirrors. See Chapter 4 for details. The virtual reflections of the environment map (I_e) are also added to the final image if M_v is 1. We extended the equations from Karsch et al. [11] and Knecht et al. [14].

$$I_m = TM(I_m) \tag{4.3}$$

$$I_r = TM(I_r) \tag{4.4}$$

$$I_f = M_v * I_m + (1 - M_v) * (I_c + I_m - I_r) + M_v * I_e \tag{4.5}$$

4.2 Ray Tracing

Our ray tracer works like described by Whitted [25]. Figure 4.4 shows how rays are traced recursively starting from the camera. On every surface point hit, Equation 4.6 will be evaluated. It was taken from Whitted [25] but we added a specular term described by Phong [20].

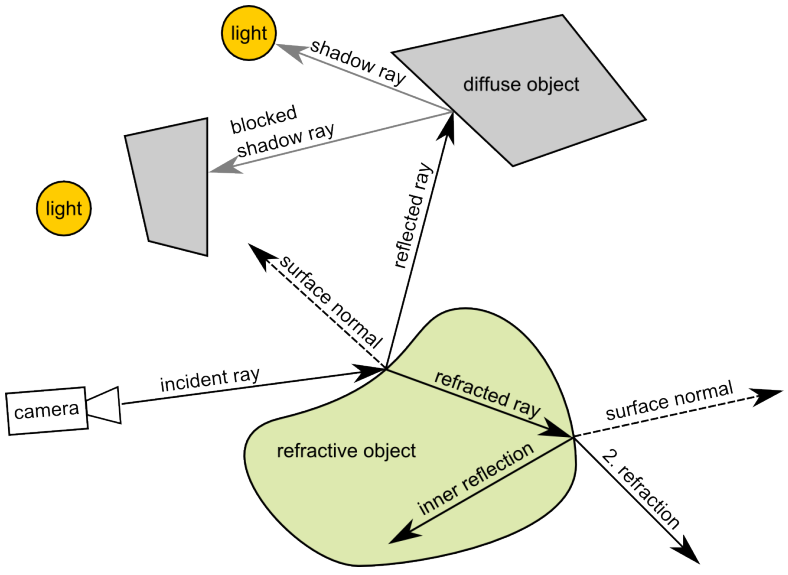

Figure 4.4: This figure outlines how Whitted ray tracing works.

$$I = I_a + \sum_{j=1}^{j=ls} (k_d \bar{N} \cdot \bar{L_j} + k_{ds} \bar{L_j} (\bar{R} \cdot \bar{V})^n) + k_s S + k_t T \tag{4.6}$$

where
I = light intensity,
I_a = reflection due to ambient light,
k_d = diffuse reflection constant,
$\bar{N}$ = unit surface normal,
$\bar{L_j}$ = the vector in the direction of the jth light source (in our case including shadow attenuation),
k_s = specular reflection coefficient,
S = the intensity of light incident from the $\bar{R}$ (reflection) direction,
k_t = the transmission coefficient,
T = the intensity of light from $\bar{P}$ (refraction) direction,
k_{ds} = specular reflection coefficient for not diffuse reflecting surfaces
$\bar{R}$ = reflection direction
$\bar{V}$ = view direction, negative ray direction
n = specular index

We separately calculate the colour of mixed and real light. For diffuse shading points the difference is only in $\bar{L_j}$ because the shadow rays can have different attenuation. I_a, k_d and k_{ds} are set to zero in our reflective materials. We only trace shadow rays for diffuse materials. S and T are obtained by tracing secondary reflection and refraction rays. k_s and k_t are obtained by an approximation of the Fresnel term found by Schlick [24] as shown in Equation 4.7 and 4.8.

$$k_s = R_m + (1 - R_m)(1 - cos(\theta))^5 \tag{4.7}$$

$$k_t = 1 - k_s \tag{4.8}$$

where
R_m = reflection when $\bar{N}$ is 90 degrees (minimal reflection),
θ = angle between $\bar{N}$ and ray direction

4.3 Differential Ray Tracing

While ray tracing, we save two types of colour in the per ray data structure (ray payload), mixed colour and real colour. The advantage is that we do not need to render two times and the intersection test is done only once per ray. This was introduced by Kán and Kaufmann [9]. We use only mixed rays though. If a ray hits a virtual object, we do not trace a real ray, because the virtual objects mask (M_v) prevents the use of the real part in this case. See Section 4.1 for details.

Shadow rays work similar. They also have two separate attenuation values and are traced until both values are zero or the light is hit. Our ray payload looks like shown in Listing 4.1.

Listing 4.1: Ray payload structures

```
struct PerRayData_shadow
{
  float3 mixedAttenuation;
  float3 realAttenuation;
};

struct PerRayData_radiance
{
  float4 mixedResult;
  float4 realResult;
  float3 virtualEnvMapReflection;
  float importance;          //to prevent endless bounces inside objects
  int depth;                 //recursion depth
  int hitVirtualObjectOnTheWay;
};
```

If the ray tracer encounters a virtual object in its recursion, it changes *hitVirtualObjectOnTheWay* from zero to one. This is later added to the mask M_v, shown in Equation 4.9. In compositing this will indicate that the mixed image should be used, see Section 4.1 for details.

$$M_v = M_v | hitVirtualObjectOnTheWay \tag{4.9}$$

4.4 Finding Information for Reflecting or Refracting Surfaces

When doing refractions and reflections secondary rays can hit a diffuse surface. If this surface is virtual it will be shaded straight away. But if it hits a real surface then a more complex approach will produce better results.

The idea is to use color information from the see-through video frame for real surfaces. Note that this is only right if we assume diffuse real surfaces. We have the shading position (P_{sh}) in world coordinates. To get the position in the video frame (P_v) we simply apply the view-projection matrix (m_c, see Equation 4.10) and check if the resulting point (P_c) lies inside the view frustum between -1 and 1. In this case we transform into screen space (P_v, see Equation 4.11) and retrieve the pixel from the video frame. With this information it is possible to perform differential rendering again.

$$P_c = m_c \times P_{sh} \tag{4.10}$$

$$P_v = \left(P_c + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right) \cdot 0.5 \tag{4.11}$$

If the point does not fall into the view frustum, we have to use the shading result of the modelled real surface. The quality will depend on the model.

Another possibility is that the point will not hit any surface. In this case we use the environment map. Since the environment map is already in low dynamic range it should not be tone mapped. To do that we extended the per ray data structure 4.1 from Kán and Kaufmann [9] and

added another buffer (I_e) with virtual reflections of the environment map. This separation allows to selectively only tone map the high dynamic range parts of the image.

CHAPTER 5

Implementation

The RESHADE framework [15] [14] is the basis of this work. It provides a working mixed reality program based on a traditional Direct3D renderer. We used it for the following purposes:

- rendering rasterized parts of the images
- merging of the ray traced and rasterized parts
- render pass for creating ray tracing mask
- tone mapping
- reading and generation of triangle meshes, textures etc.
- reading camera and fish eye camera (if any) data
- control the library for tracking bar codes and updating positions
- building a scene tree
- extracting lighting information from the environment image
- configuration through an XML file

The second major technology used is OptiX [19], it is described in Chapter 3. Because the RESHADE framework is a Microsoft C# .net project and OptiX uses C++, we needed to implement a wrapper. All of the needed OptiX host code for initialisation, data transfer and update is programmed in C++. Wrapping is done by a slim class that uses Microsoft's C++/CLI bindings. In the RESHADE framework we can import this class and use it. Transferring data arrays and pointers needs marshalling and $unsafe\{\}$, because managed context does not allow access to the heap.

5.1 Detailed Approach

The first thing that happens in connection with the OptiX part is to import the scene tree including geometric data, materials, textures, lights and camera data from the RESHADE framework. Texture data is shared with Direct3D.

Then the RESHADE framework renders all diffuse objects into two buffers, mixed (I_{mR}) and real (I_{rR}) (see Equations 4.1 and 4.2 as well as Figure 5.1). It also fills a third buffer, which is a mask (M_v) containing 1 for virtual objects and 0 otherwise. This mask can be seen in Figure 5.2 on the left side. In a separate render pass another mask is created (M_r) which contains 1 for specular objects and 0 otherwise as seen in Figure 5.2 on the right side.

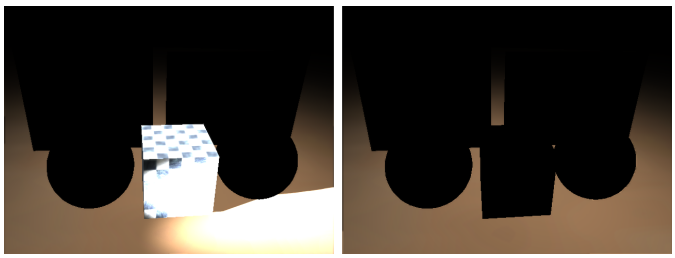

Figure 5.1: Left: Mixed buffer without reflective or refractive objects (I_{mR}). Right: Real buffer without reflective or refractive objects (I_{rR}).

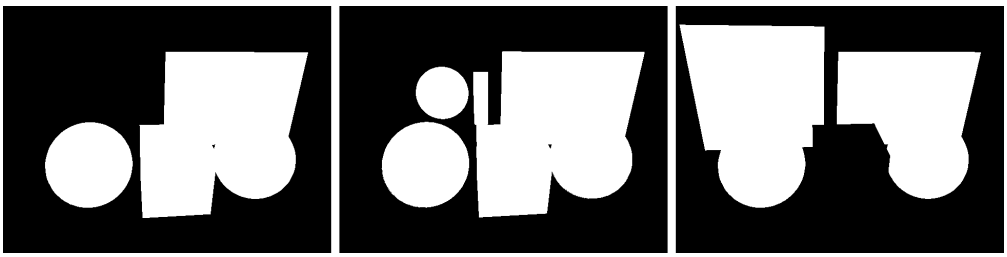

Figure 5.2: Left: Virtual objects mask M_v created by the RESHADE framework. Middle: M_v extended by the OptiX part. Reflected virtual objects are added (hitVirtualObjectOnTheWay). Right: Ray tracing mask (M_r), white parts will be traced.

The OptiX program for shooting rays into the scene uses the M_r mask (Figure 5.2 on the right) along with the camera's view projection matrix. If the mask is 1, it simply unprojects the pixel position to get the ray direction. Initially all pixels that are computed in OptiX are set to 0 in M_v.

Now there are two possibilities:

1. The ray hits a virtual refracting / reflecting object.
   In this case M_v will be set to 1. This happens with the green objects in Figure 4.1.

2. The ray hits a real refracting / reflecting object.
   Secondary rays will be traced. Only if they eventually hit a virtual object, the mask will be set to 1. In the example the real mirror is marked with red and the reflected virtual objects with orange.

The resulting M_v is a combination from the RESHADE framework and OptiX part, shown in 5.2 in the middle.

The ray tracer creates three buffers as described in Chapter 4. Mixed (I_{mO}) and real (I_{rO}) buffers are merged with the rasterized image coming from the RESHADE framework, and the virtual environment map reflection buffer (I_e) is used directly in the last, differential, composition stage. The merged buffers can be seen in Figure 5.3.

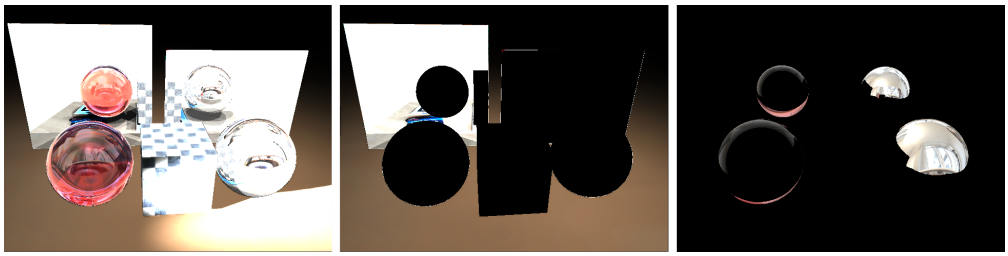

Figure 5.3: Left: Mixed buffer including reflective and refractive objects (I_m). Middle: Real buffer including reflective or refractive objects (I_r). Right: Virtual environment map reflections (I_e).

5.2 Environment Illumination

The RESHADE framework generates more than 100 virtual point lights from the environment map, which are later used for shading with imperfect shadow maps as visibility test [14]. Using this amount of light sources in a ray tracing method is not possible due to real-time constraints. For that reason the number of lights has to be reduced while keeping a good enough approximation. Due to performance we did not use any clustering algorithm. Instead we put the lights into 8 spatial bins and calculate the weighted average of the position and sum up the luminosity.

Listing 5.1: Code listing showing the algorithm to reduce the amount of environment lights.

```
for(int i=0; i < envLightCount; i++) {
  int sector = 0;
  float3 position = envLightArray[i].position;
  float3 color = envLightArray[i].color;

  if(position.x > 0) sector = 4;
  if(position.z > 0) sector += 2;
  if(abs(position.y) > abs(position.x) + abs(position.z))
    sector += 1;
```

```
    //less influence on position from dark lights
    newEnvLights[sector].position += position * luminance(color);
    newEnvLights[sector].color += color;
}

for(int i=0; i < 8; i++)
    lights[i].position = normalize(lights[i].position)*20.0f;
```

This is done by a simple and fast algorithm (Listing 5.1) that is executed on the GPU. The environment lights are placed on a sphere around the centre by the RESHADE framework, we do the same for simplicity.

5.3 Compositing

The last step is to composite the whole image. This is done in two Direct3D fragment shader programs operating on screen quads according to the equations shown in Chapter 4.1. The OptiX buffers I_{rO}, I_{mO} and I_e are transferred to Direct3D using interoperability functions provided by OptiX. By using the interoperability functions the transfer to main memory and back on the graphics card is avoided.

CHAPTER 6

Results

We compared our combined ray tracing differential instant radiosity solution with an existing solution implemented using traditional rasterization. The comparison was done on 3 different scenes shown in Figure 6.1. There is a refractive and a reflective Stanford bunny and a scene showing complex reflections.

Figure 6.1: Comparison was done on these 3 scenes

Full size pictures can be found in appendix A.

6.1 Test Setup

The performance tests where made on two different PCs.

1. AMD Phenom II X4 CPU, 4 GiB RAM and a NVIDIA GeForce GTX 550 Ti with 1 GiB of video memory.

2. Intel Core2 Quad CPU, 8 GiB RAM and a NVIDIA GeForce GTX 580 with 1.5 GiB of video memory.

We took a static image as the environment map and a webcam for capturing the see-through video.

The Stanford bunny has 69666 triangles and the scene with the two spheres approximately 4006. We also rendered the ray tracing parts of the bunny scenes with 4x and 9x Super-Sampled Anti-Aliasing (SSAA) (shooting 4 and 9 rays per pixel then taking the average) for comparison.

6.2 Description of the Result

Stanford bunny made from glass

The rasterized and ray traced bunnies have the same refraction index but the result is different (Figure 6.2). This is due to approximations made by the rasterizer which is implemented with a method described by Wyman [26]. This method uses a rough guess on the exit point of the ray and it does not consider inner reflection.

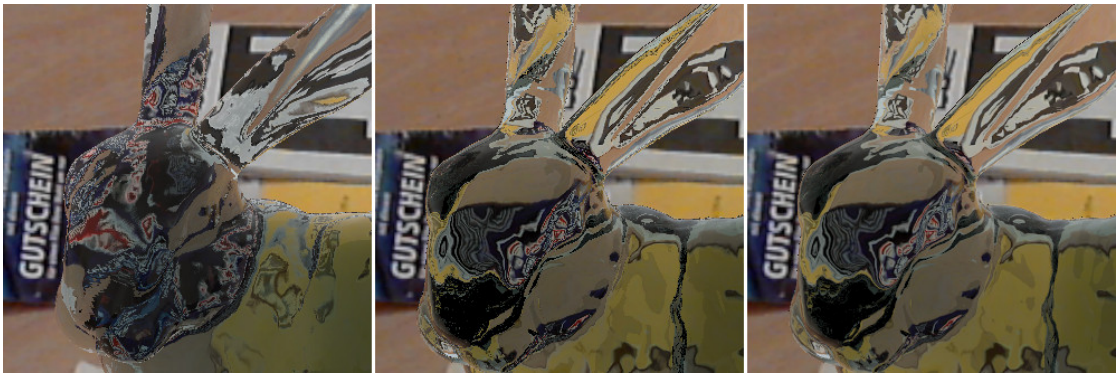

Figure 6.2: Left: Rasterization. Middle: Ray tracing without anti-aliasing. Right: Ray tracing with 9x SSAA.

On the middle image there are strong artefacts in the area of the eye and the ears. These come from rays being refracted into a slightly different direction and because of that having a completely different colour. With higher refraction indices these artefacts increase. Strong edges also produce strong artefacts. This can be solved by sampling in a higher resolution as shown in the right picture.

Another artefact which is present in all three images comes from the low resolution triangle model. The normals are interpolated between the triangle edges so that the model looks smooth. As a result the shading point does not correspond to the perceived surface as can be seen in Figure 6.3. This can be seen on the snout between tip and eye.

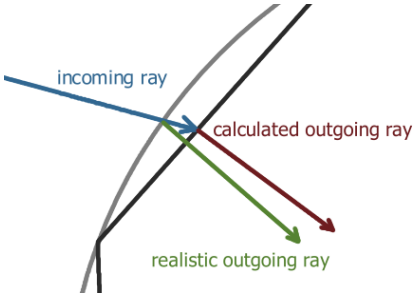

Figure 6.3: Because normals are interpolated the perceived surface (grey) does not correspond to the geometric one (black).

Mirroring Stanford bunny

The reflecting bunnies show very strong artefacts at the snout, see Figure 6.4. These are due to missing information. As described in Chapter 4 information in mirrors is retrieved either from the see-through video frame, rendering of real objects or an environment map. Due to different rendering of real objects the artefacts in rasterized and ray traced images are different.

Figure 6.4: Left: Rasterization. Middle: Ray tracing without anti-aliasing. Right: Ray tracing with 9x SSAA.

Also the mirroring bunnies show aliasing artefacts. The reasons are similar to the refractive ones.

Complex Scene

This scene contains a real and a virtual mirror at the back and two reflecting spheres in the foreground. The ray tracer on the right image in Figure 6.5 shows reflections of the spheres in the mirrors as well as recursive reflections of the right sphere. The rasterizer implementation in the RESHADE framework includes second order reflections as described by Popescu et al. [21]. It is currently limited to a recursion depth of 2, the marble cube seems distorted in the mirror and reflected refractions are not supported. However this implementation is still work in progress.

Figure 6.5: Left: Rasterizer. Right: Ray tracing without anti-aliasing.

On the right mirror are strong artefacts in both implementations. It is possible to see where the reflection of the see-through video frame ends and the rendered real objects and environment map are used instead.

6.3 Benchmarks

The measurements in Table 6.1 show that in some cases the ray tracer has a slightly better performance than the pure rasterizer. But if the triangle count is great and complex inner reflections are computed then ray tracing is much slower. To get good results for the glass bunny we needed to use anti-aliasing and this made the frame rate even lower. However it is possible to reduce recursion depth and get better performance by the cost of refraction quality. Doing so also reduces aliasing effects.

The results show that our method is faster than using ray tracing only for the whole image and that specular objects are very costly to render in a realistic way.

6.4 Limitations

One limitation is speed in scenes with many transparent or mirroring objects and polygons. This is the same for all ray tracing methods.

Another limitation is the different rendering of diffuse objects in ray tracer and rasterizer. The picture in Figure 6.5 on the right shows a different rendering of the marble box in the mirrors. Also the shadows are inconsistent because the rasterizer uses instant radiosity as a solution for global illumination and the ray tracer uses only a limited set of light sources, see Listing 5.1.

A general problem with mirroring objects in mixed reality is missing information. This can be seen very clearly in Figure 6.5 on the right mirror in both implementations.

Since the stack on the graphics card is smaller than the stack for a traditional host programs, a limitation specific to our method is a smaller recursion depth.

Scene	Resulution	PC (see 6.1)	Renderer	FPS	Image
glass bunny no AA	1280 x 1024	1	rasterizer	4.2	A.1
glass bunny no AA	1280 x 1024	1	our method	3.1	A.2
glass bunny 4x SSAA	1280 x 1024	1	our method	0.8	A.3
glass bunny 9x SSAA	1280 x 1024	1	our method	0.5	A.4
glass bunny no AA	800 x 600	1	rasterizer	9.9	A.1 $_{4)}$
glass bunny no AA	800 x 600	1	our method	6.4	A.2 $_{4)}$
mirroring bunny no AA	1280 x 1024	1	rasterizer	7.2	A.5
mirroring bunny no AA	1280 x 1024	1	our method	8.1	A.6
mirroring bunny 9x SSAA	1280 x 1024	1	our method	2.3	A.7
complex scene no AA	800 x 600	1	rasterizer	10.9	A.8
complex scene no AA	800 x 600	1	rasterizer$_{1)}$	10.2	A.9
complex scene no AA	800 x 600	1	our method	11.1	A.10
complex scene no AA	1900 x 1100	1	rasterizer$_{1)}$	1.4	A.9 $_{4)}$
complex scene no AA	1900 x 1100	1	our method	1.7	A.10 $_{4)}$
complex scene no AA	800 x 600	2	rasterizer	17.2	A.11
complex scene no AA	800 x 600	2	our method	22.4	A.12
complex scene	800 x 600	1	diffuse objects$_{3)}$	25.9	A.13
complex scene	800 x 600	1	ray tracer $_{2)}$	6.5	A.14

Table 6.1: Benchmarks of the two different rendering systems. $_{1)}$ including second order reflections $_{2)}$ ray tracer for the whole image $_{3)}$ specular objects were replaced with diffuse ones $_{4)}$ exactly the same scene but different resolution

CHAPTER 7

Conclusion

We have presented a method to mix traditional rasterization with ray tracing in mixed reality applications and we have shown that it is possible to achieve interactive frame rates for certain scenarios. Our ray tracing implementation generates two images, one containing only real objects and the other real and virtual ones, which are later merged with the video frame using the method of differential rendering. Our implementation computes the real and mixed images in a high dynamic range. Before merging the images are reduced by a tone mapping algorithm. With ray tracing it is possible to create realistic looking images of virtual refracting and reflecting objects in mixed reality. It is trivial to implement recursive reflections and refractions.

Using back-projection to get the colour of the see-through video frame greatly improves the quality of refractions and reflections but there are still artefacts if the video frame does not cover all of the reflection.

Anti-aliasing improves greatly the quality of complex refractions and reflections.

7.1 Future Work

It is possible to use ISMs 2.2 in the ray tracer to make the rendering of diffuse surfaces more similar to the rasterizer. Ideally the shading algorithms of ray tracer and rasterizer should be exactly the same.

Anti-aliasing can be implemented more efficient by loosing only little quality as described for instance by Whitted [25]. He is using adaptive super-sampling which means that he only sends more than one ray per pixel if this is necessary due to geometry or textures.

The problem of missing information in mirroring objects can be reduced by adding information. Using a fish eye camera instead of the webcam and cutting out only a part in the centre for the video image would extend the area of possible back-projection.

APPENDIX A

Images

In this appendix we present the full size images from our test scenes.

A.1 Glass Stanford bunny

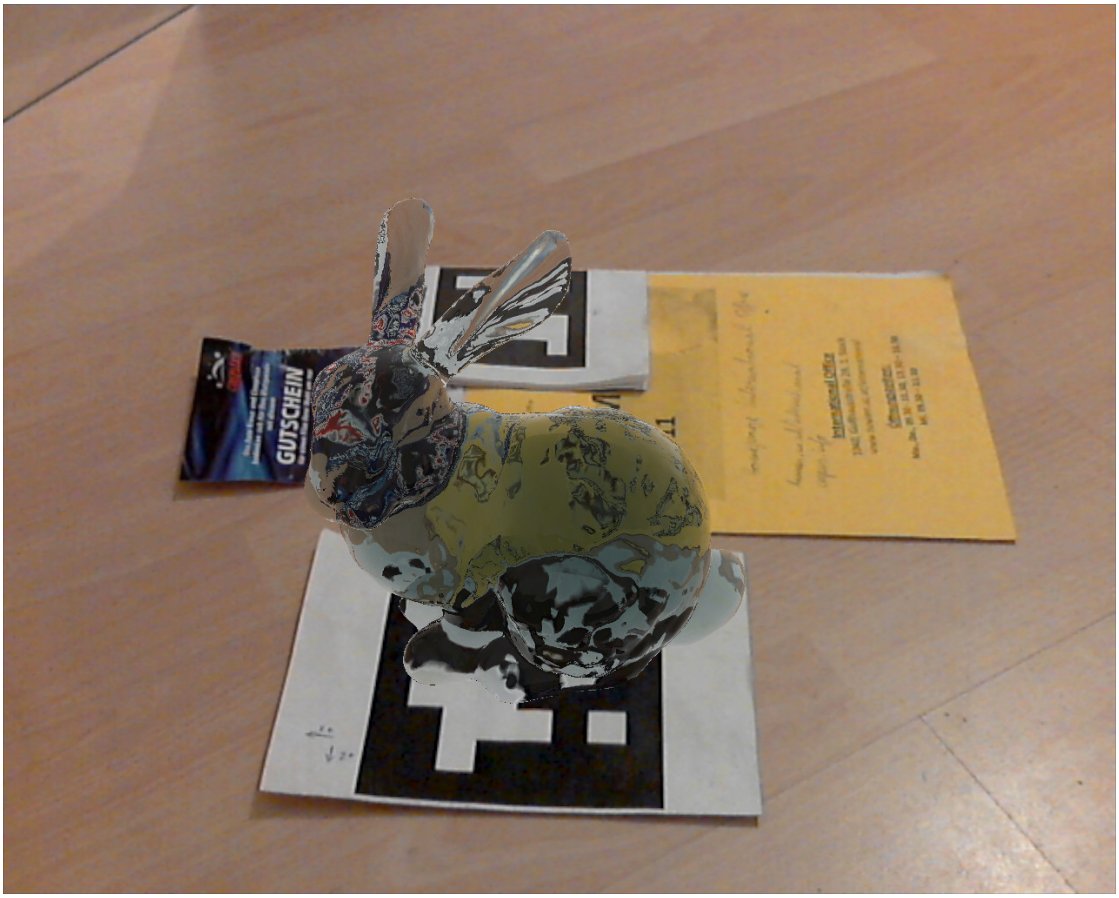

Figure A.1: Rendered with rasterization, 1280 x 1024, no AA, 4.2 fps

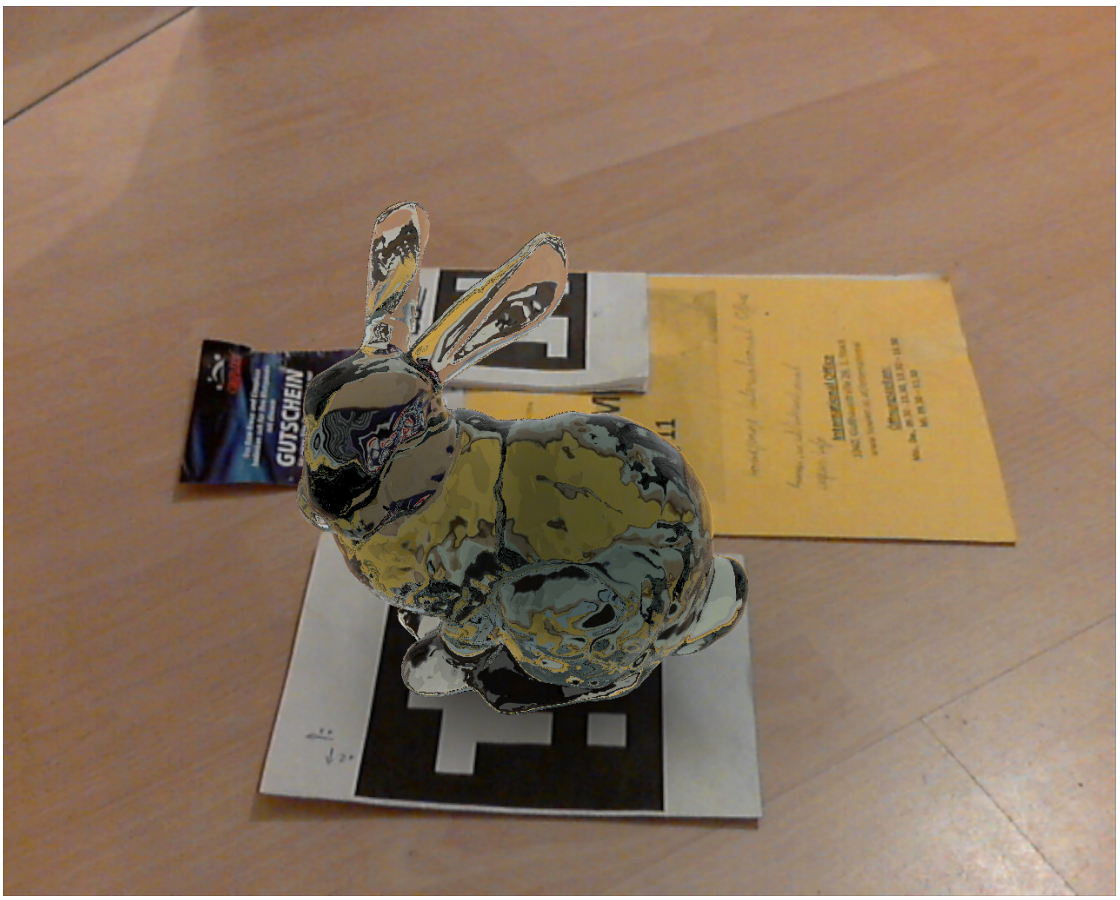

Figure A.2: Rendered with ray tracing, 1280 x 1024, no AA, 3.1 fps

Figure A.3: Rendered with ray tracing, 1280 x 1024, 4x SSAA, 0.8 fps

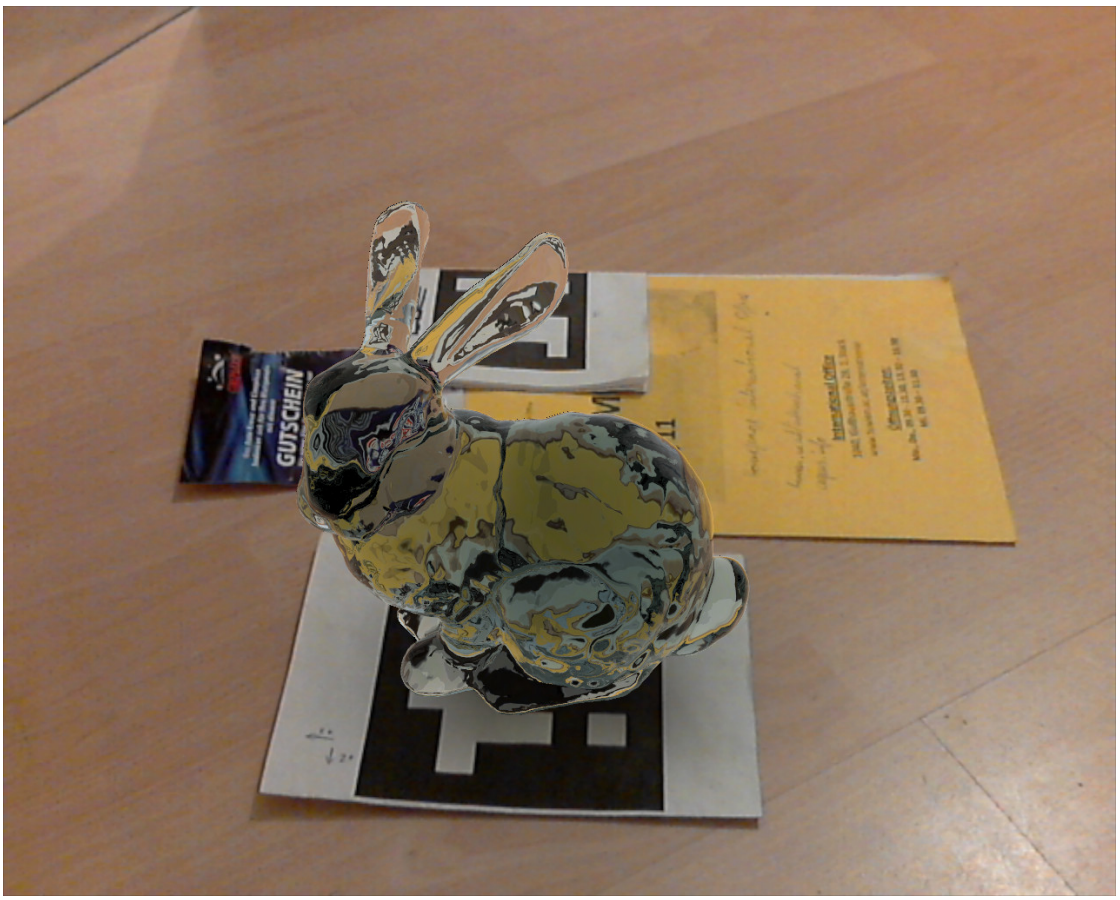

Figure A.4: Rendered with ray tracing, 1280 x 1024, 9x SSAA, 0.5 fps

A.2 Reflective Stanford bunny

Figure A.5: Rendered with rasterization, 1280 x 1024, no AA, 7.2 fps

Figure A.6: Rendered with ray tracing, 1280 x 1024, no AA, 8.1 fps

Figure A.7: Rendered with ray tracing, 1280 x 1024, 9x SSAA, 2.3 fps

A.3 Scene with complex reflections

Figure A.8: Rendered with rasterization, 800 x 600, no AA, 10.9 fps

Figure A.9: Rendered with rasterization, 800 x 600, no AA, 10.2 fps

Figure A.10: Rendered with ray tracing, 800 x 600, no AA, 11.1 fps

Figure A.11: Rendered with rasterization on PC 2 (see Chapter 6.1), 800 x 600, no AA, 17.2 fps

Figure A.12: Rendered with ray tracing on PC 2 (see Chapter 6.1), 800 x 600, no AA, 22.4 fps

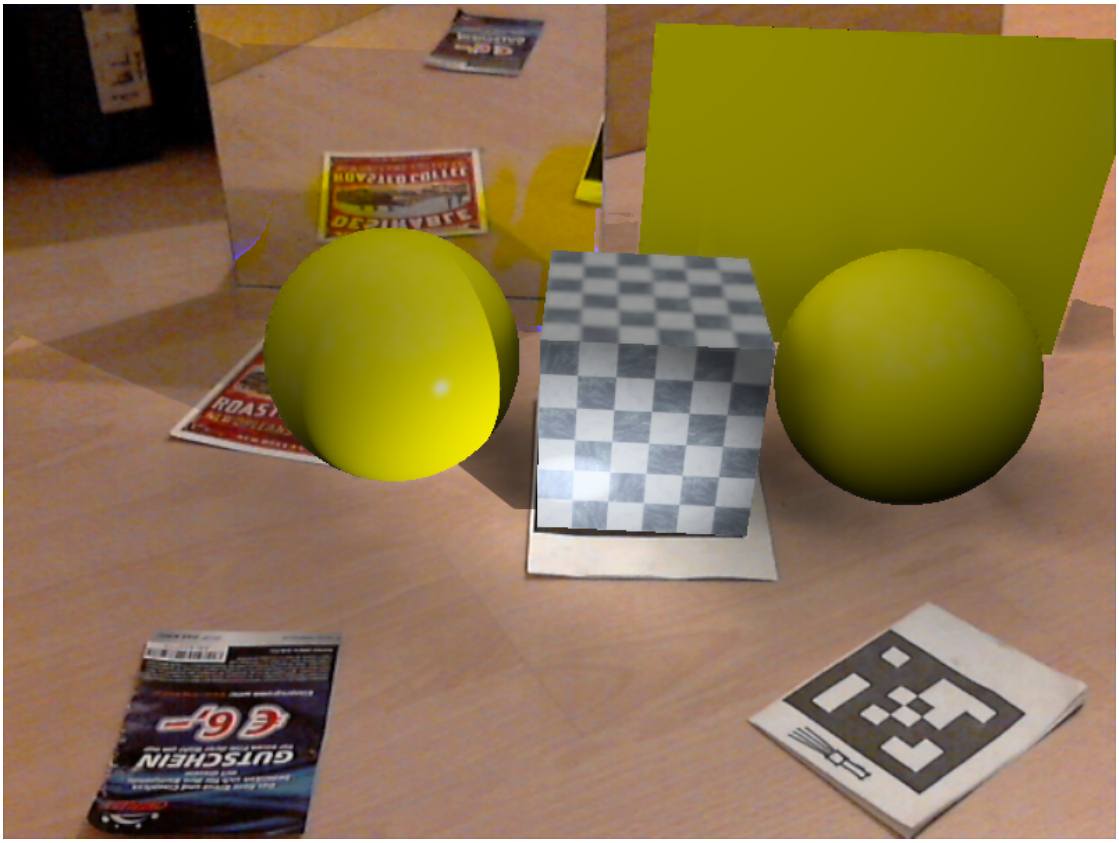

Figure A.13: Rendered with rasterizer, diffuse objects instead of reflective and refractive ones, 800 x 600, no AA, 25.9 fps.

Figure A.14: Rendered whole image with ray tracer, 800 x 600, no AA, 6.5 fps.

Bibliography

[1] Kusuma Agusanto, Li Li, Zhu Chuangui, and Ng Wan Sing. Photorealistic rendering for augmented reality using environment illumination. In *Mixed and Augmented Reality, 2003. Proceedings. The Second IEEE and ACM International Symposium on*, pages 208–216. IEEE, 2003.

[2] James Bigler, Abe Stephens, and Steven G. Parker. Design for Parallel Interactive Ray Tracing Systems Design for Parallel Interactive Ray Tracing Systems. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 187–196. IEEE, 2006.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786, 2004.

[4] Andrew I. Comport, Éric Marchand, and François Chaumette. A real-time tracker for markerless augmented reality. In *Proceedings of the 2nd IEEE/ACM International Symposium on Mixed and Augmented Reality*, volume 03, pages 36–45. IEEE Computer Society, 2003.

[5] Paul Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 189–198. ACM, 1998.

[6] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, number 6, pages 15–22. ACM, 2005.

[7] Thorsten Grosch. Differential Photon Mapping-Consistent Augmentation of Photographs with Correction of all Light Paths. In *Eurographics*, pages 53–56. Eurographics Association, 2005.

[8] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd. Natick, MA, USA, 2001.

[9] Peter Kán and Hannes Kaufmann. Physically-Based Depth of Field in Augmented Reality. In *Eurographics 2012-Short Papers*, pages 89–92. The Eurographics Association, 2012.

[10] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, (1), 2010.

[11] Kevin Karsch, Varsha Hedau, David Forsyth, and Derek Hoiem. Rendering synthetic objects into legacy photographs. *Proceedings of the 2011 SIGGRAPH Asia Conference, ser. SA*, 11:157, 2011.

[12] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *ACM SIGGRAPH Computer Graphics*, 20(4):269–278, August 1986.

[13] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, New York, New York, USA, 1997. ACM Press.

[14] Martin Knecht, Christoph Traxler, Oliver Mattausch, Werner Purgathofer, and Michael Wimmer. Differential Instant Radiosity for mixed reality. In *Mixed and Augmented Reality (ISMAR), 2010 9th IEEE International Symposium on*, pages 99–107. IEEE, October 2010.

[15] Martin Knecht, Christoph Traxler, Oliver Mattausch, and Michael Wimmer. Reciprocal shading for mixed reality, 2010.

[16] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[17] C Nvidia. NVIDIA CUDA programming guide. 2011.

[18] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879 – 899, 2008.

[19] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)*, 29(4):66, July 2010.

[20] Bui Tuong (Utah University) Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[21] Voicu Popescu, Chunhui Mei, Jordan Dauble, and Elisha Sacks. Reflected-Scene Impostors for Realistic Reflections at Interactive Rates. *Computer Graphics Forum*, 25(3):313–322, September 2006.

[22] Tobias Ritschel, Thorsten Grosch, Min H. Kim, Hans-Peter Seidel, Dachsbacher, Carsten, and Jan Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Transactions on Graphics (TOG)*, 27(5):129, 2008.

[23] Imari Sato, Yoichi Sato, and Katsushi Ikeuchi. Acquiring a radiance distribution to superimpose virtual objects onto a real scene. *Visualization and Computer Graphics, IEEE Transactions on*, 5(1):1–12, 1999.

[24] Christophe Schlick. An Inexpensive BRDF Model for Physically-based Rendering. *Computer graphics forum*, 13(3):233–246, 1994.

[25] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[26] Chris Wyman. Interactive image-space refraction of nearby geometry. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia - GRAPHITE '05*, page 205, New York, New York, USA, 2005. ACM Press.