Universiteit Utrecht

MSc Project

Video-Based Scene and Material Editing

Author:
Thijs Zumbrink BSc

Supervisor:
Dr. Robby T. Tan

Student number: 3040682
Thesis number: ICA-3040682
Date: May 8, 2013

Abstract

The technique presented in this report alters the appearance of objects in video by substituting the original material for another, synthetic, material. Inspired by methods of Khan et al. and Karsch et al., the object is tracked and re-rendered using global illumination data obtained from the input video, assisted by a brief user annotation. A model of the environment is constructed geometrically, onto which textures from the input sequence are projected. This leads to an approximate environment that provides indirect lighting.

A major part of this system is identical to inserting objects into video. An implementation is shown that allows physically correct interaction of the inserted object with its environment; the object is shaded from the correct directions, casts shadows onto the environment and can even block out light sources, reducing the overall brightness of the result.

The result is a system for synthetic object insertion or replacement into video, requiring no access to the physical scene, which works on low-quality recorded footage. Additionally, a contribution is presented on the subject of Additive Differential Rendering, a technique composing rendered objects into original footage, where the render time can be drastically decreased. Another contribution is shown on the subject of Exemplar-Based Image Inpainting, enabling the use of more image content to fill a region.

Contents

1 Introduction **1**
- 1.1 Summary . . . 1
- 1.2 Report outline . . . 2
- 1.3 Contributions . . . 2

2 Related work **3**
- 2.1 Relevant concepts . . . 3
  - 2.1.1 The pinhole camera model . . . 4
  - 2.1.2 Coordinate systems . . . 4

3 Project description **9**
- 3.1 Project description . . . 10
- 3.2 Goals . . . 10

4 Basic implementation of the program **11**
- 4.1 Tracking . . . 13
  - 4.1.1 Color features . . . 13
  - 4.1.2 Post processing . . . 14
  - 4.1.3 Disk fitting . . . 14
- 4.2 Shape recovery . . . 16
- 4.3 Image inpainting . . . 17
  - 4.3.1 Theory . . . 17
  - 4.3.2 Data term . . . 19
  - 4.3.3 Patch matching . . . 20
  - 4.3.4 Patch look-up . . . 20
- 4.4 Environment mapping . . . 23
- 4.5 Rendering . . . 26
  - 4.5.1 The rendering equation . . . 26
  - 4.5.2 Material properties . . . 27
  - 4.5.3 Renderer implementation . . . 27
  - 4.5.4 Shader implementation . . . 28
  - 4.5.5 BRDF implementation . . . 31
- 4.6 Results . . . 33

5 Improvement of the environment and rendering system **35**
- 5.1 Rendering Synthetic Objects into Legacy Photographs . . . 36
  - 5.1.1 Scene annotation . . . 37
  - 5.1.2 Texture mapping . . . 38

5.1.3 Light parameter estimation 39
5.1.4 Rendering 41
5.1.5 Results 46
5.2 Integration into the video-based pipeline 48
5.2.1 Environment Mapping 48
5.2.2 Camera Tracking 49
5.2.3 Evaluation and discussion 50

6 Conclusion **52**
6.1 Future work 52

Bibliography **56**

A Evaluated videos **57**

Chapter 1

Introduction

Material Editing in real-life video footage is an attractive problem in the Computer Vision field. The goal is to plausibly alter the material of a given object in a digitized visual scene. It enables us to create videos that would be expensive, unfeasible or even impossible using real objects. For a long time, this required either labor-intensive manual modeling of the virtual environment or physical access to the scene to capture illumination data. The ideal automatic program automatically gets a good idea on the shape of the target object and the appearance of the surrounding environment. Having acquired this information, the object can be re-rendered using different material properties.

An image is the simple representation of a scene, but complex things happen when we record one. Light is emitted from a light source and interacts with the scene. Some of the light is reflected towards the camera and hits a light-sensitive area, where it is absorbed. After possibly some adjustments by the camera, the image is recorded. When we look at the image, our brain forms a mental reconstruction of the scene. In this application of computer vision however, we usually do not have the scene description available, all we have is the visualization. We try to reverse these complex events proceeding image formation, to again form a reconstruction of the scene. Or we can interpret the image in a way that is similar to how the brain interprets images. In practice, we must apply a combination of these two approaches if we want to reconstruct the scene, which makes it a versatile challenge.

1.1 Summary

This report contains the theoretical information, implementation details and evaluation on the topic of *Video-Based Material Editing.* An MSc project was performed as an educational assignment for Game and Media Technology, a master program at Utrecht University, in the field of Computer Vision.

In computer vision we continually aim to improve the ability of the computer to "see" in the sense that it interprets visual scenes and extracts meaningful information. With the advance of computing power over the years, more and more techniques, some of which have been developed decades ago, become feasible to execute and give astonishing results. The ultimate goal of computer vision is not only to emulate human vision, but to go *beyond.* And researchers worldwide are progressing.

One of the techniques is Image-Based Material Editing. With very little information about the scene, the software recognizes the shape of an object and the details of the surroundings in a single image, only to change the material of the object with a completely different material or range of materials. In the process, information about the scene is used to create a plausible image, computing reflections and even plausible transparency. An experimentation project has previously been performed to investigate this technique.

The goal of this project is to extend this method to a video-based version: to alter the appearance of a target object in a video stream, while keeping the number of required input parameters to a minimum. To my knowledge, such a technique does not exist yet.

1.2 Report outline

The outline of this report is as follows: in Chapter 2, a quick overview is given on the work previously done in this field. The problem is defined in Chapter 3, where a pipeline and brief overview of the proposed technique is given. The planned steps of the project are also laid out, detailing the steps planned to progress the project, of which the execution is described in Chapters 4 and 5. Finally, Chapter 6 concludes by reflecting on the project.

1.3 Contributions

The main contribution of this project is an implementation of a video-based version of the technique described in [KHFH11], which renders synthetic objects into images. This results in a system where synthetic objects are inserted into video, requiring no access to the scene, one single LDR video input, and relatively little user experience. Furthermore, there is no need of specialized equipment. For example, all but the first videos shown in Appendix A were recorded with a cellphone camera and are quite shaky.

More in-depth to this specific technique, a contribution has been made to drastically reduce render time making the image-based render technique of Karsch et al. feasible on consumer-grade hardware and opening the way for video rendering. This is achieved by carefully observing the object composition method and adjusting the characteristics of the renderer accordingly. Details are shown in Section 5.1.4. To give a rough indication of the impact: rendering a 200-frame video takes about 30 hours on a consumer CPU, which would have been well over a month without this change. Furthermore, the theory behind this contribution is applicable to other techniques that use the *Additive Differential Rendering* method of Debevec.[Deb98]

In the field of image inpainting, a change has been made to the technique of [CPT04] that allows the use of image patches from the entire image, instead of only patches that are located near the region that is being inpainted. The details of this change are explained in Section 4.3.4.

Chapter 2

Related work

Rendering objects using real-world reflection data began with Blinn and Newell [BN76] showing how texture maps could be used not only to texture an object, but also to use them as illumination data suitable for reflections. This essentially lead to an approach that provides global illumination data to a synthetic object using a single texture map. This map is specific to a certain point in space, however errors in the reflection go unnoticed up to a certain degree.[OCS05] Debevec[Deb98, Deb02] uses a light probe to capture all global illumination in one photograph by placing a mirror-like ball into the scene and photographing it, leading to the technique called *Image-Based Lighting*. This reduces graphics artists' work but obviously needs physical access to the scene. Furthermore, the photograph of the probe should be captured with High-Dynamic Range (HDR)[DM97] to prevent intensity information loss due to tone mapping. By allowing an approximation of the illumination map, the technique of Khan et al.[KRFB06] does not need physical access to the scene. The illumination is taken from the input photograph by mapping the center region of the image onto two hemispheres. Karsch et al.[KHFH11] model the environment geometrically and project the input photograph onto the model. They alleviate the need for HDR photography and also allow light sources to be out of view, requiring a brief user annotation of the scene.

One of the prime inspirations for this project is *Image-Based Material Editing* by Khan et al.[KRFB06]: a technique that plausibly alters the appearance of an existing object in the scene. From only the HDR input photograph and a mask of the object, the shape is automatically detected using Shape from Shading[Hor70]. Ambiguities in the depth aspect of the shape are allowed because the camera angle does not change. Using a combination of image inpainting and basic environment mapping, the appearance of the object can be altered even to having transparent or reflective materials. Finally, the 3D representation of the object is re-rendered in the location of the original object, using image-based lighting.

2.1 Relevant concepts

Below, some relevant concepts to computer graphics are explained. Firstly, the pinhole camera model is widely used as a simple basis for 3D to 2D projection.

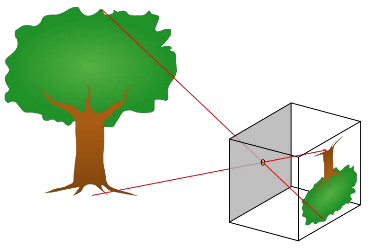


Figure 2.1: The pinhole camera model in a nutshell. We call the distance between the pinhole and the image plane the *focal length* f.

Secondly, commonly used coordinate systems and the relations between them are illustrated.

2.1.1 The pinhole camera model

Image theory typically features the *pinhole camera* model for the purpose of explaining projection of the scene onto the image plane. This is a simple model in which the light of a scene passes through a tiny (infinitesimal) hole and falls onto the image plane inverted, where it is recorded as intensity, shown in Figure 2.1.1. Instead of working with this inverted image, for convenience we often imagine a virtual projection plane in front of the camera which is not inverted, as shown in Figure 2.2(d). It should be noted that the pinhole camera model differs from actual cameras which have lenses and larger apertures. However, the pinhole camera model leads to simple coordinate system transformations which makes its use attractive.

The world coordinates of a certain point in the scene are called $\vec{X}_w = [X_w, Y_w, Z_w]^T$, and the image plane coordinates of the corresponding point $\vec{x}_c = [x_c, y_c]^T$:

$$\vec{x}_c = \begin{bmatrix} x_c \\ y_c \end{bmatrix} = \frac{f}{Z_w} \begin{bmatrix} X_w \\ Y_w \end{bmatrix} \tag{2.1}$$

with f being the camera's focal length. This manner of projection is called perspective projection [Hor86], as distances between points on the image plane become relatively smaller for scene points with a z-coordinate far from the observer.

2.1.2 Coordinate systems

Detailed below is an outline of the coordinate systems used in my programs. There are five different coordinate systems with corresponding transformations, accounting for 3D geometry modeling and viewing. Refer to Figure 2.2 for each of the coordinate systems.

Model Every scene model renders itself locally in $\mathbb{R}^3$. The axes follow the right-handed system convention.[1] This system is rotation, translation and

[1]Point the fingers of your right hand in the direction from x to y and stretch your thumb out; the thumb now points in the z direction. So when x points right and y points up, z points toward the viewer.

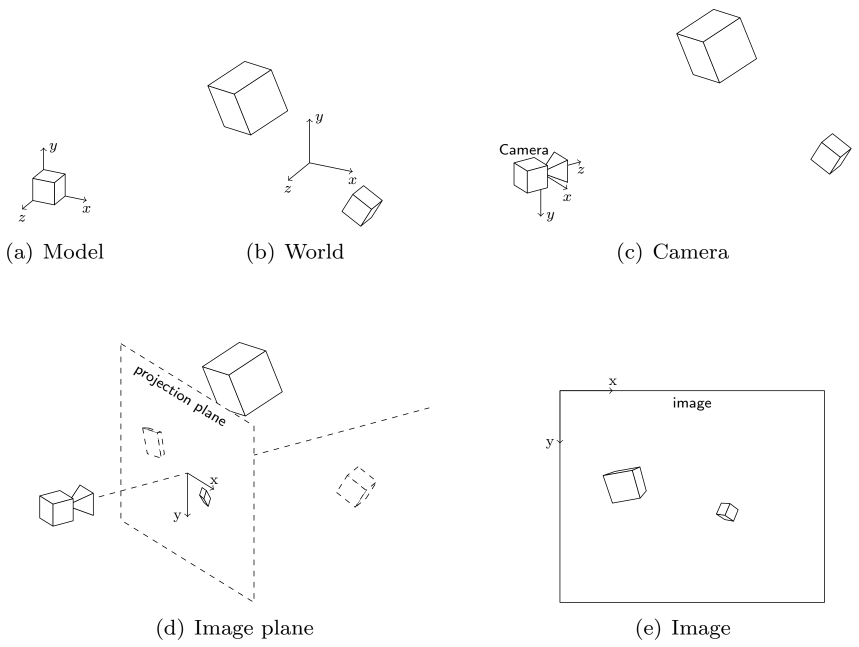


Figure 2.2: (a) The model coordinate system in which the model is defined independently of scene dimensions. (b) The world coordinate system with two models inserted having different scale, rotation and translation parameters. (c) A camera is added to the scene and the scene contents are transformed into the camera coordinate system. (d) The image plane contains the projections of the camera coordinate system. This is a perspective projection towards the camera's optical center. (e) The image coordinate system in which the final representation of the scene will be.

scale invariant; scene objects are placed into the world coordinate system by scaling, rotating and translating them according to object properties. This is commonly referred to as the *model transformation.*

The transformation from model ($\vec{X}_m = [X_m, Y_m, Z_m]^T$) to world ($\vec{X}_w = [X_w, Y_w, Z_w]^T$) is as follows:

$$\vec{X}_w = tRS\vec{X}_m = [R|t]S\vec{X}_m \tag{2.2}$$

or in matrix form, using homogeneous coordinates:

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{pmatrix} = \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \tag{2.3}$$

where t is the translation matrix, R is the rotation matrix and S is the scaling matrix. Coordinates are handled as homogeneous coordinates.[2]

While the implemented software exclusively use right-handed coordinate systems, this convention is not universally used. For example, OpenGL used a left-handed coordinate system to specify models. Simply inverting one axis (I invert z) with a scaling operation circumvents this problem.

World The world coordinate system ($\mathbb{R}^3$) is essential in the sense that it enables us to perform computations involving multiple models, light sources and the camera, such as rendering the entire scene or performing other global calculations. When the camera moves, we do not need to recalculate every object's position in the world. Every scene object, which is an instance of a model, is placed into the scene by transforming it according to the rotation, translation and scale parameters of that object. Cameras also have rotation and translation parameters to place them into the scene, usually obtained by performing external camera calibration.

Camera The camera coordinate system ($\mathbb{R}^3$), also known as *eye space*, has the x axis pointing right, the y axis pointing down and the z axis pointing into the scene.[3]

Imagine we would place the camera into world coordinates. We would apply the model transformation using the camera's translation and rotation properties. However, since we want to view the scene through the camera, we must transform the entire scene into a system with the origin at the camera and oriented however the camera is oriented. To do this, we do not apply the model transformation to the camera but rather apply its inverse to the scene. This is commonly referred to as the *view transformation.* The transformation from ($\vec{X}_w = [X_w, Y_w, Z_w]^T$) world to ($\vec{X}_c = [X_c, Y_c, Z_c]^T$) camera thus becomes:

$$\vec{X}_c = [R|t]^{-1}\vec{X}_w = R^{-1}t^{-1}\vec{X}_w \tag{2.4}$$

[2]When we want to transform a 3-dimensional point $\vec{X} = [x, y, z]^T$, we add a fourth coordinate: $\vec{X'} = [x, y, z, 1]^T$. This enables us to perform translations through matrix multiplication, which would be impossible otherwise.

[3]Multiple implementations exist for right-handed camera coordinate systems. The other usual implementation is to point x right, y up and positive z away from the viewing direction. The image plane then lies at $z = -f$ instead of $z = f$ where f is the camera's focal distance.

where t is the translation of the camera center and R is the camera rotation, together called the extrinsic camera parameters. Or in matrix form, using homogeneous coordinates:

$$\begin{pmatrix} r'_{11} & r'_{12} & r'_{13} & t'_x \\ r'_{21} & r'_{22} & r'_{23} & t'_y \\ r'_{31} & r'_{32} & r'_{33} & t'_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} = \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} \tag{2.5}$$

In the program, R is determined by aligning the axes of the world coordinate system to the scene geometry; the x and y axes are aligned to two vanishing lines while constraining them to being perpendicular. z is then perpendicular to x and y by means of the cross product, following the right-handed convention. The vanishing points alignment is further described in Section 5.1.1.

Image plane The image plane is identical to the camera coordinate system, with the exception that every point is projected onto the plane $z = f$, the camera's focal distance. Therefore this system is conceptually in $\mathbb{R}^2$, although the transformation below results in a three-dimensional point with $z = f$.

To transform a point from the camera ($\vec{X}_c = [X_c, Y_c, Z_c]^T$) to the image plane ($\vec{x}_c = [x_c, y_c, z_c]^T$):

$$\vec{x}_c = f \cdot \frac{\vec{X}_c}{Z_c} \tag{2.6}$$

or in matrix form using homogeneous coordinates:

$$\begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & f & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} = \begin{pmatrix} fX_c \\ fY_c \\ fZ_c \\ Z_c \end{pmatrix} \propto \begin{pmatrix} x_c \\ y_c \\ f \\ 1 \end{pmatrix} \tag{2.7}$$

This transformation is a direct consequence of using the pinhole camera model. It is commonly referred to as the *projection transformation*. The division by Z_c to homogenize the resulting coordinate leads to the perspective property of the result, where points that are far away appear relatively closer to each other than points near the camera.

Image The image coordinate system is used to discretize the image plane into pixels. We start counting from the top left, meaning that x counts to the right and y counts downward. The image is in $[0 \dots w-1][0 \dots h-1] \subset \mathbb{N}^2$ where (w, h) are the dimensions of the image.

To transform the image plane ($\vec{x}_c = [x_c, y_c]^T$) to the image ($\vec{x}_{im} = [x_{im}, y_{im}]^T$):

$$\vec{x}_{im} = S\vec{x}_c + \vec{P} \tag{2.8}$$

or in matrix form, using homogeneous coordinates:

$$\begin{pmatrix} S_x & 0 & P_x \\ 0 & S_y & P_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_c \\ y_c \\ 1 \end{pmatrix} = \begin{pmatrix} x_{im} \\ y_{im} \\ 1 \end{pmatrix} \tag{2.9}$$

where $\vec{P}$ is the center of projection in pixel coordinates and S is a scaling matrix multiplying the x coordinate with w_{im}/w_c and the y coordinate with h_{im}/h_c (the ratio between visible image plane dimensions and image dimensions). This is commonly referred to as the *viewport transformation*.

Chapter 3

Project description

The main problem in material editing is re-rendering an object from the scene using different material properties. There are several sub-problems to be solved to successfully achieve this. To be able to re-render an object, we need to know the shape of the object for accurate shading and the appearance of the environment for accurate reflections and transparency. Therefore, two important sub-problems are *shape recovery* and *environment mapping*. Furthermore, we need an *inpainting* method to remove the object from the scene prior to mapping the environment, and we need *object boundary tracking* to be able to locate the target object in subsequent frames. The basic pipeline is as follows:

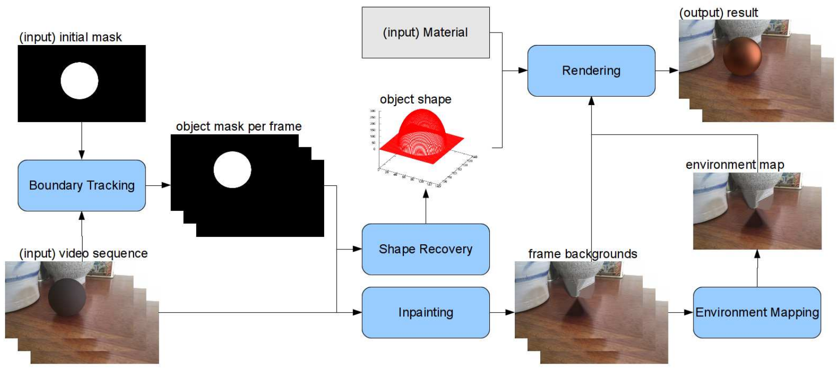


Figure 3.1: The pipeline for the proposed Video-Based Material Editing technique, which will be explained further in Chapter 4.

Note that the inputs to the system are few: the object mask in the first frame, the desired material and of course the video data. The focus will lie on off-line processing, i.e. all data is known at the start of the process.

3.1 Project description

The proposed technique will be designed and implemented as part of the master's project for the Game and Media Technology master's program at Utrecht University. The 45 ECTS project, resulting in a master thesis, is required to graduate. The project is supervised by dr. R. T. Tan, who specializes in physics-based computer vision and machine learning.

To solve the sub-problems of this technique, existing techniques and/or software will be used if available, or new methods of obtaining the desired results will be explored otherwise.

The first step is to create a pipeline with all components, providing outputs for all sub-problems and consequently for the main problem. These results will be very basic as the first implementation of components will be rudimentary. After creating this first version, sub-problems that require additional attention will be chosen and improved by using different techniques or improving current techniques.

The project will be concluded with a thesis document describing all aspects of the project, and a final presentation.

3.2 Goals

Step	Goal
1	General pipeline with rudimentary components
1.1	Research knowledge for components
1.2	Create implementations for component
2	Improvement of component(s)
2.1	Choose component that needs improvement
2.2	Research knowledge for that component
2.3	Refine implementation
3	Thesis
4	Final presentation

Chapter 4

Basic implementation of the program

For the first implementation of the pipeline, a lot of components will be implemented in a basic form. Therefore a certain number of assumptions are made on the input to the program. Most notably the tracking stage and shape recovery stage are primitive. The basic descriptions of the stages are as follows:

Tracking INPUT: A video and a mask of the object in the first frame. The target object is assumed to be of uniform and easily distinguishable color. Furthermore, it should be shaped as a sphere, for the tracking phase will track a disk on the video. OUTPUT: A mask of the tracked object per frame, i.e. the *foreground* mask.

Shape recovery INPUT: The input video along with a mask of the object for each frame. In conjunction with the assumption in the tracking stage, the resulting shape will always be a perfect sphere on the tracked position. The target object should not be obstructed. The additional input, original video data, can later be used for more elaborate shape recovery techniques, such as Shape from Shading or Structure from Motion. OUTPUT: a depth map per frame which denotes the surface elevation of the object for each pixel within the object mask.

Image inpainting INPUT: The input video along with a mask per frame of the area that needs to be removed and replaced with inpainted data. The object mask will be used here to remove the object. OUTPUT: A video from which the object has been removed, i.e. the *background*.

Environment mapping INPUT: A video in which the environment is visible. The result from the image inpainting stage will be used. During rendering of a certain frame, the background for that frame will be used in a simple image-based lighting technique. OUTPUT: An environment object in which directions can be queried to collect irradiance.

Rendering INPUT: A video containing shape data, a backgrounds video and an environment object. The shape and environment are everything needed to re-render the object with a new appearance. The backgrounds are used to insert the synthetic object into. OUTPUT: The final video in which the apparent material of the object has been altered.

The idea for the general pipeline of Video-Based Material Editing stems from my experience with *Image*-Based Material Editing. I previously performed an experimentation project on the subject, following [KRFB06] as the main technique to study. As for the various stages in this pipeline: the implementations of the image inpainting, environment mapping and rendering stages are following various papers on these subjects. The tracking and shape recovery stages are simpler and have a "homebrew" implementation. The separate stages are further explained in the sections below.

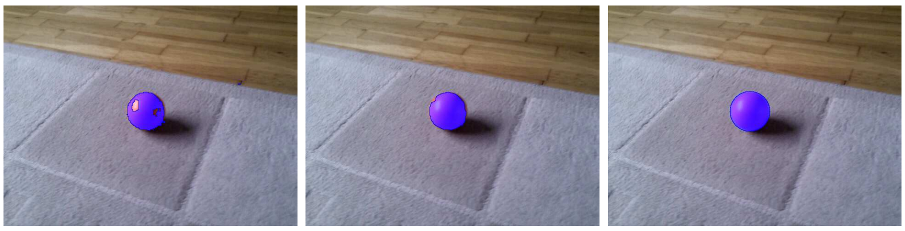

(a) Raw color histogram tracking (b) Refined with post processing (c) Refined with post processing and disk fitting

Figure 4.1: Visualization of the tracking result. (a): the color histogram values are used in the calculation of f which is then thresholded. (b): post processing is applied to remove outliers and fill in holes. (c): the disk fitting procedure uses a disk-shaped template to refine the result.

4.1 Tracking

The implemented tracking method evaluates pixels according to their by color features. Per frame the pixels are given a value f ranging from 0 to 1, denoting how likely that pixel is to belong to the foreground, i.e. the object to track. When f reaches a value over a threshold T, the pixel is determined to belong to the foreground and is added to the object mask used in the rest of the pipeline. The value $T = 0.5$ was empirically chosen.

The tracking result and effect of post processing and *disk fitting*, as explained in the following sections, are shown in Figure 4.1.

4.1.1 Color features

To keep track of color features, two histograms are constructed, H_{fg} for foreground pixels and H_{bg} for background pixels. These histograms are two-dimensional, each bin addressed by hue and saturation. The histograms are populated with pixel values from the first frame, then scaled to have the highest value in the histograms equal to 1.

f is determined as follows:

$$\begin{aligned} f &= 0 && \text{if } h_{\text{fg}} = 0 \text{ and } h_{\text{bg}} = 0 \\ f &= h_{\text{fg}}/(h_{\text{fg}} + h_{\text{bg}}) && \text{otherwise} \end{aligned} \tag{4.1}$$

where h_{fg} is the value of H_{fg} in the hue-saturation bin corresponding to the pixel in question, and h_{bg} is defined similarly. The resulting value of f will range from 0 to 1.

From the use of this method, the need arises to assume easily distinguishable object color. If the object color is sufficiently different from the background, this method will produce acceptable results. Note that highlights (meaning fully saturated, or white pixels) can cause problems.

4.1.2 Post processing

The mask generated constructed is far from optimal. The post processing phase refines it by applying several basic image processing techniques to remove small artifacts. We use two morphological operations: *opening* and *closing* which are defined as erosion followed by dilation for opening, and the same operations but in reverse for closing.[Ser82] Erosion and dilation take a binary image and a *structuring element*[1] as inputs. Erosion "shrinks" the mask in such a way that a mask pixel is erased when we overlay the structuring element and notice that the pixels under the structuring element are not all part of the masked pixels. Dilation is the dual operation: a non-mask pixel is added to the mask when we overlay the structuring element and notice that there is a masked pixel within the structuring element.

To give a more formal definition of erosion:

$$\begin{aligned} A \ominus B &= \{z \in E | B_z \subseteq A\} \\ \text{with } B_z &= \{b + z | b \in B\}, \forall z \in E \end{aligned} \tag{4.2}$$

where A is the input image, B is the structuring element and E is the euclidean space. The structuring element B is translated through E by translation vector z, resulting in B_z. For all translations where B_z is a not a subset of A, we exclude the pixel z from the output image.

The dilation operation can be defined as the Minkowski Sum:

$$A \oplus B = \{a + b | a \in A, b \in B\} \tag{4.3}$$

Finally, we define opening and closing, respectively:

$$\begin{aligned} A \circ B &= (A \ominus B) \oplus B \qquad \text{and} \\ A \bullet B &= (A \oplus B) \ominus B \end{aligned} \tag{4.4}$$

During post processing of the computed mask, opening is used to remove any small mask artifacts, while closing is used to fill in small holes in the mask and irregularities in the mask's boundary. After these operations, a flood-fill operation is performed on the mask to fill in any remaining interior holes larger than the structuring element. The flood-fill operation is done on the pixel $(0, 0)$ in the image, so it is assumed that the object is not present there. It will fill all areas that are on the exterior of the mask, so negating it will give us only the interior holes. The interior holes are then added to the mask by means of a union.

4.1.3 Disk fitting

Finally we improve the mask by assuming that the tracked object is a sphere. This is a procedure trying to estimate the optimal position of a template, in our case a disk, in the mask. After the post processing step, we take the median of x and y values of mask pixels to use as the center of our initial disk. The diameter of that disk is the average of the initial width and height, which are computed as

$$w = \max_{(x,y) \in \text{ mask}} x - \min_{(x,y) \in \text{ mask}} x \tag{4.5}$$

[1]I used a 9×9 square for opening and a 13×13 square for closing.

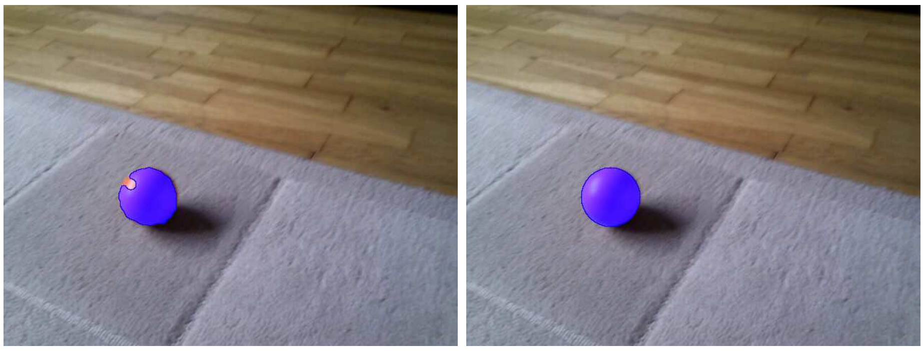

(a) Mask with visible “dent”　　(b) Mask enhanced by disk fitting

Figure 4.2: Visualization of the disk fitting result. (a): the mask before disk fitting. (b): the mask after disk fitting. The disk was found by hill climbing from the initial guess until we encountered a (local) optimum. The disk is optimal in the sense that most disk-pixels belong to the original mask and most non-disk pixels lie outside of the original mask.

and the height similarly.

After constructing the initial disk, we perform a hill-climbing method to improve the fit. At each iteration, we explore six alternatives: $x \pm 1$, $y \pm 1$ and $r \pm 1$, keeping the alternative that provides us with the highest quality measure. When we arrive at a (local) optimum, we stop.

To calculate the quality of a disk, we count how many pixels of the mask are inside the disk and how many are erroneously outside:

$$\begin{aligned}
\text{quality}(\text{pixel } p) &= 1 && \text{if } p \in \text{ mask and } p \in \text{ disk}\\
\text{quality}(\text{pixel } p) &= 1 && \text{if } p \notin \text{ mask and } p \notin \text{ disk}\\
\text{quality}(\text{pixel } p) &= -1 && \text{if } p \in \text{ mask and } p \notin \text{ disk}\\
\text{quality}(\text{pixel } p) &= -1 && \text{if } p \notin \text{ mask and } p \in \text{ disk}\\
\text{quality}(\text{disk}) &= \sum_{p \in \text{ image}} \text{quality}(p) && \qquad (4.6)
\end{aligned}$$

This method gives good results even when we have considerable “dents” or “tails” in the mask, the effect is shown in Figure 4.2.

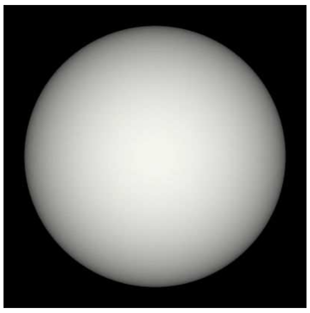


Figure 4.3: An example of the depth map for a sphere.

4.2 Shape recovery

With the assumption that the target object is a sphere, this stage is exceptionally simple. For portability reasons, the disk-fitting procedure is executed again on the object mask, resulting in a center $\vec{c} = [x_c, y_c]^T$ and a radius r.

Since $x^2 + y^2 + z^2 = r^2$ for spheres, we derive $z = \sqrt{r^2 - x^2 - y^2}$. In the depth map we set the pixel to the value:

$$[x + x_c, y + y_c]^T = 255 * \frac{z}{r} \tag{4.7}$$

This yields a maximum value (a white pixel) for the part of the sphere closest to the observer and a minimum value (a black pixel) for parts that have the most distance or are not part of the object. The values that are not part of the object have no meaning in the depth map and will be ignored because they are not within the object mask.

Shape look-up is the process of retrieving the three-dimensional surface normal $\vec{n}$ for a certain pixel $\vec{x} = [x, y]^T$. The normal is constructed from the gradients in x and y direction. The reason we do not simply return the surface normal from a mathematical sphere is because the depth map may encode other shapes as well.

Let $p = \frac{\partial z_{x,y}}{\partial x}$ and $q = \frac{\partial z_{x,y}}{\partial y}$ be the first partial derivatives of the surface, with respect to x and y respectively. In discrete terms, as our values are stored in pixels, this means: $p = z_{x,y} - z_{x-1,y}$ and $p = z_{x,y} - z_{x,y-1}$. Then the vectors $\vec{r}_x = [1, 0, p]^T$ and $\vec{r}_y = [0, 1, q]^T$ are both parallel to the tangent plane at $\vec{x}$. The normal vector thus becomes:

$$\vec{n} = \vec{r}_x \times \vec{r}_y = [-p, -q, 1]^T \tag{4.8}$$

We then normalize this vector to obtain the unit normal.[Hor86]

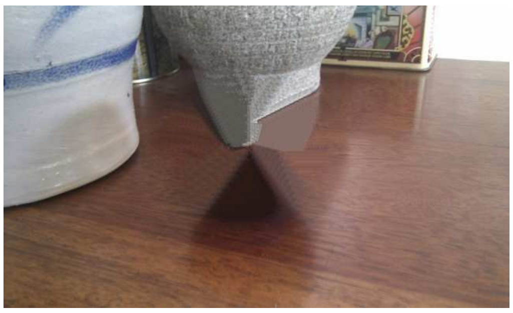


Figure 4.4: A photograph where a sphere has been removed and the resulting gap inpainted using the implemented technique of exemplar-based image inpainting.

4.3 Image inpainting

While techniques exist for video inpainting, I feel that an image-based approach is sufficient for the purpose of this project. Inpainted imagery will only be used for an environment map as the basis for reflections. As [OCS05] have shown, the human visual system is quite generous with errors in illumination. Furthermore, image-based inpainting technique shown in this section was already implemented, interfacing with the data structures used in this project.

Image inpainting is the process of filling in an unknown region of an image by substituting that region with plausible content. The technique is used to erase an object in an image, after which the created gap is filled in with data that is supposed to be the background texture behind the object. Various techniques have been published, and exemplar-based methods which fill the region by taking small patches from elsewhere in the image, have been gaining popularity lately.[KSW10] Therefore, we will explore one of these techniques which fills in large regions of an image, focusing on correctly propagating texture in the region.

4.3.1 Theory

This particular technique, called *Exemplar-based image inpainting* and published in [CPT04], is based on two separate classes of problems: "texture synthesis" algorithms that generate large textures based on sample textures, and "inpainting" techniques that fill small gaps in images. The presented algorithm takes advantages from both these approaches in order to propagate structure along large gaps in the image.

An important aspect of their approach is the preservation of structure in the image. Structured texture in the known region of the image are propagated into the unknown region by assigning a structure score to areas of the image. Structured areas are more likely to be filled in before unstructured areas. The authors note that the resulting filling order is important for the quality of the result. Along with this structure score, image areas are also assigned a confidence score; the confidence of every pixel is maintained, representing how sure we are that the data in that pixel is correct. This confidence gradually shrinks as we propagate pixels into the unknown region, and possibly propagate those pixel values over and over again.

In the algorithm, the aforementioned image areas are square patches of pixels. The size of these patches is an input parameter, and can generally be larger for coarser textures. As the algorithm copies texture patch-by-patch, larger patches result in lower running time as texture is propagated in larger amounts, reducing the amount of checks that need to be done in between this copy process. The algorithm is outlined in Procedure 1.

Procedure 1 Outline of the image inpainting algorithm.

```
Input: Image $I$, target region $\Omega$
Output: The image formed by completing $I$ to the extent that $\Omega$ has been completely filled.
while $\Omega \neq \emptyset$ do
    Identify the fill front $\delta\Omega^t$
    Compute priorities $P(p)$, $\forall p \in \delta\Omega^t$
    $\Psi_{\hat{p}} \leftarrow$ patch around point $\hat{p}$, where $\hat{p}$ has the highest priority.
    Find exemplar $\Psi_{\hat{q}} \in \{I \setminus \Omega\}$ that minimizes the distance $d(\Psi_{\hat{p}}, \Psi_{\hat{q}})$
    Copy image data from $\Psi_{\hat{q}}$ to $\Psi_{\hat{p}} \cap \Omega$
    Update confidence values $C(p)$, $\forall p \in \Psi_{\hat{p}} \cap \Omega$
end while
```

The priority of a patch is the multiplication of the confidence term and the data term:

$$P(p) = C(p)D(p) \tag{4.9}$$

The confidence term $C(p)$ is measured at a certain point p. In the calculation for this value, we look at the patch Ψ_p, the patch centered at p. The more we know about this patch, the higher the confidence term becomes. This makes it easy to fill in patches that are already largely known. Out confidence in the patch points also plays a role. The value of this term is simply the sum of all confidence values of pixels inside Ψ_p, divided by the area of the patch:

$$C(p) = \frac{\sum_{q \in \Psi_p \cap (I \setminus \Omega)} C(q)}{|\Psi_p|} \tag{4.10}$$

Where $C(q)$ is the confidence stored in point q, as a result of either initialization, or because q was previously filled in by the algorithm. After we fill in a patch, we store the confidence of all filled pixels to the confidence $C(p)$ calculated with this formula. Confidence values of all $p \in (I \setminus \Omega)$ are initialized to 1, while those of $p \in \Omega$ are initialized to 0.

The data term $D(p)$ is used to increase the priority of structured patches. Therefore, this term will be higher when there is a stronger structure present at the point p. Any gradients present in the patch influence in this term. The term is defined as follows, clarified by Figure 4.3.1:

$$D(p) = \frac{|\nabla I_p^{\perp} \cdot \vec{n}_p|}{\alpha} \tag{4.11}$$

where α is a normalization factor, set to 255 for typical images.

The important notion here is that structure is propagated up to the point that we are not really confident about the validity of those pixels anymore, in

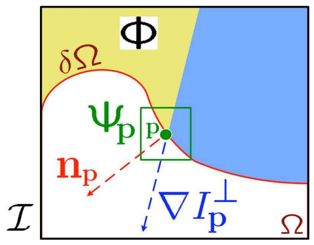


Figure 4.5: Notation diagram from [CPT04]. $\vec{n}_p$ is the normal to the contour of $\delta\Omega$. $\nabla I_p^\perp$ is the image isophote (the "direction" and intensity of the gradient) at point p.

which case we continue filling in less-structured patches. This is different from previous approaches such as e.g. the Onion Peel algorithm, which fills the image in a clockwise order instead of determining priority values.

4.3.2 Data term

In the computation of the data term, the factors $\nabla I_p^\perp$ and $\vec{n}_p$ are used. We will look into how these factors are computed.

The image isophote, $\nabla I_p^\perp$, is the direction perpendicular to the strongest gradient in the patch. Or in other words: the direction of the most dominant line in the patch. We should aim for this direction to have a large angle, perpendicular at best, with the fill-front normal in that patch, so we know that this patch contains structure that we can easily propagate.

Procedure 2 Obtain the most dominant direction in a given patch.

```
**Input:** Image intensities $I(x,y)$, patch center $[x_{c_p}, y_{c_p}]^T$, patch radius $R$
**Output:** Image isophote $\nabla I_p^\perp$ for patch $p$
$\hat{m}^2 \leftarrow -1$
**for all** $[x,y]^T \in [x_{c_p} \pm R/3] \times [y_{c_p} \pm R/3]$ **do**
  $I_x \leftarrow I(x,y) - I(x-1,y)$
  $I_y \leftarrow I(x,y) - I(x,y-1)$
  $m^2 \leftarrow I_x^2 + I_y^2$
  **if** $m^2 > \hat{m}^2$ **then**
    $\hat{m}^2 \leftarrow m^2$
    $\hat{I}_x \leftarrow I_x$
    $\hat{I}_y \leftarrow I_y$
  **end if**
**end for**
$\nabla I_p^\perp \leftarrow [-\hat{I}_y, \hat{I}_x]^T$
```

In the implementation, shown in Procedure 2, only the center region of the patch is inspected in order to prevent strong structure far from the center from influencing the data term. Also, this saves computation time slightly. For each pixel in the examined area, the horizontal and vertical gradients are computed in a trivial fashion, and combined into a gradient vector. The magnitude of that vector is registered. After examining all pixels, we take the gradient vector with the largest magnitude. This gradient vector is then rotated 90 degrees so we can use it in the data term computation.

As for the fill-front normal $\vec{n}_p$, we construct this term by adding together a number of vectors obtained from pixels in the patch, as shown in Procedure 3. As we work with filled and unfilled space, we can now look at the patch as being a binary image. Assume that unfilled pixels are 0 and filled pixels are 1. Call this value $f_{x,y}$. We know that the center of the patch that we are working with lies on the fill-front, as we only compute priorities for those patches. Therefore, we could simply add up every vector from the center of the patch to all unfilled pixels, normalize it, and we obtain a fill-front normal vector.

Procedure 3 Determine the fill-front normal for a patch containing binary pixels.

```
Input: Information about filled pixels $f_{x,y}$, patch center $[x_{c_p}, y_{c_p}]^T$, patch radius $R$
Output: Fill-front normal $\vec{n}_p$ for patch $p$
$\vec{n} \leftarrow [0,0]^T$
for all $[x,y]^T \in [x_{c_p} \pm R] \times [y_{c_p} \pm R]$ do
  if $f_{x,y} = 0$ then
    $\vec{n} \leftarrow \vec{n} + [x - x_{c_p}, y - y_{c_p}]^T$
  end if
end for
$\vec{n}_p = \vec{n}/|\vec{n}|$
```

4.3.3 Patch matching

Patch matching is the process of determining how similar two patches are. For this, we employ a distance metric that compares the colors of the two patches. For simplicity, we simply take the L_2 distance between the two patches, as if each patch is a large vector with $3N^2$ values; 3 color channels for N pixels. We use this operation for matching an exemplar patch to a *query patch*, so naturally the query patch contains unfilled pixels. The distance for those unfilled pixels is set to zero.

The authors of [CPT04] use almost the same approach: they take the sum of squared differences of all corresponding pixel pairs. However, they do not use the RGB color model but the CIE Lab color model, as this model respects perceptual uniformity. This remains a possible improvement to the implementation.

4.3.4 Patch look-up

In the original algorithm, the authors look in the neighborhood of the query patch for suitable exemplars. Looking in the whole image would be too time consuming, as this step is performed a lot of times throughout the execution of the program. There are however data structures that can facilitate the look-up procedure of patches to be able to use the entire image data as exemplar patches. In [KZN08], several candidate structures are explored, each being based on a binary space partitioning tree such as the kd-Tree. The problem

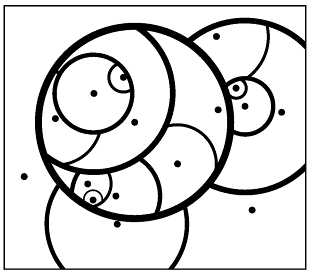


Figure 4.6: A visualization of a *vp*-Tree containing two-dimensional points. Each circle partitions the points belonging to that sub tree into two subsets. Image from [KZN08]

of finding similar patches can be formulated as finding nearest neighbors in a high-dimensional space.[2] The advantages of such binary space partitions are weighed in the paper, and the *vp*-Tree, or vantage point tree, is suited for this task. Additional information on this data structure can be found in [Yia93].

The *vp*-Tree partitions a space into two subspaces: point near the pivot and points far from the pivot. The pivot is chosen randomly from the input set. Then, a threshold distance is calculated, being the median of all distances of points in the space to the pivot. By taking the median, we get a balanced tree, as we toughly split the set in half each step. After the split, we recursively partition the two subsets. The construction method is illustrated in Procedure 4.

Procedure 4 Constructing the *vp*-Tree.

```
**Input:** Point collection $P$
**Output:** A vp-Tree $T$ containing the points in $P$
$T.root \leftarrow$ random point in $P$
$T.\mu \leftarrow$ median of $d(p, T.root), \forall p \in \{P \setminus T.root\}$
$T.left \leftarrow$ new vp-Tree with points $\{p \in \{P \setminus T.root\} | 0 < d(p, T.root) < T.\mu\}$
$T.right \leftarrow$ new vp-Tree with points $\{p \in \{P \setminus T.root\} | d(p, T.root) \geq T.\mu\}$
$T.low \leftarrow \min_{p \in \{P \setminus T.root\}} d(p, T.root)$
$T.high \leftarrow \max_{p \in \{P \setminus T.root\}} d(p, T.root)$
```

The $0 < d(p, T.root)$ constraint removes identical patches from the tree, which can save a lot of space and computation time when working with synthetic images.

The query method looks in both sub trees, but prunes them before recursion. When we are querying a patch and our best found match so far has a distance of τ to the query point. With the information stored in the tree nodes, we can prove that recursion into certain sub trees is futile, as we cannot find a distance closer to τ, so we can prune those sub trees. The search is explained in Procedure 5.

The *vp*-Tree takes some time to construct, but it gives us the advantage of being able to inspect the entire image contents when looking for candidates. However, there is a fundamental problem with patch look-up data structures

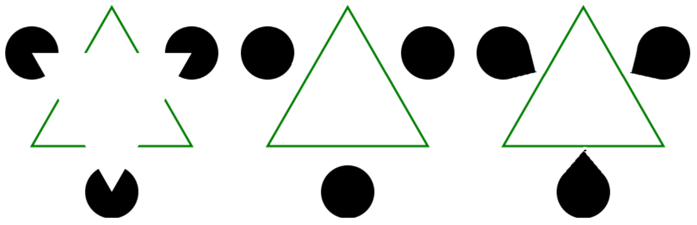
[2]If we take 9x9 patches for example, we get $3 \cdot 9^2 = 243$-dimensional points.

Procedure 5 Searching the *vp*-Tree.

```
**Input:** vp-Tree $T$, query point $q$, best match $\hat{T}$ so far and its distance $\tau$
**Output:** The vp-Tree $\hat{T}$ which has a root that is closest to $q$, and its distance $\tau$
if $T$ is empty then
    return $(\hat{T}, \tau)$
else
    $d \leftarrow d(q, T.root)$
    if $d < \tau$ then
        $\tau \leftarrow d$
        $\hat{T} \leftarrow T$
    end if
    if $T.left.low - \tau < d < T.left.high + \tau$ then
        $(\hat{T}, \tau) \leftarrow Search(T.left, q, \hat{T}, \tau)$
    end if
    if $T.right.low - \tau < d < T.right.high + \tau$ then
        $(\hat{T}, \tau) \leftarrow Search(T.right, q, \hat{T}, \tau)$
    end if
    return $(\hat{T}, \tau)$
end if
```

that was only identified after the implementation, and which is also explained in [KSW10]: the query point q contains unknown values. Or in other words: q has a variable dimensionality. This makes use of the data structure less optimal as we cannot prune the tree anymore without getting answers that are very far from the optimal answer. As a workaround, the query patch is first *auto-completed* by fixing the gradient in unknown pixels at zero. Then, the first 100 matches are obtained from the tree, after which the best is selected by computing distances to the unmodified q which contains the unknown pixels.

The result in Figure 4.7(b) is generated using a look-up method facilitated by the *vp*-Tree data structure. One of the advantages of this method is that we can have every possible patch in the image as an exemplar patch, instead of only the immediate neighborhood. Therefore, the three disks can be made round. Should we only consider nearby patches as candidate exemplars, the disks would turn into droplet shapes such as in Figure 4.7(c).



(a) Input image (b) Result image with vp-Tree (c) Result image without vp-Tree

Figure 4.7: The results of inpainting on a triangle which is completed by the algorithm. (b) shows the improved result over (c), which is similar to that of the original authors.

4.4 Environment mapping

As the input to our program is limited, we have limited knowledge about the environment around the scene. We have the intensity values of the pixels in the image to work with. How do we use this information to construct a useful environment map, responsible for plausible reflections of the scene? [KRFB06] use a very simple method that produces decent results without requiring any additional input.

The general idea, illustrated in Figure 4.4, is to construct a unit sphere shell by taking a large disk from the input image and protruding it to form a hemisphere. We simply protrude this disk in two directions, toward and away from the viewer, to obtain two hemispheres; together they form a full spherical environment mapping.

We choose the center of the disk to be the center of the image $[x_c, y_c]^T$, and the radius to be $R = min(w, h)/2$. This ensures that we take the disk as large as possible. For each pixel $[x, y]^T$ in the disk, we construct a normalized coordinate pair:

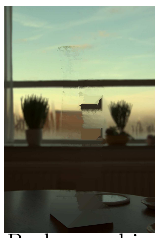
$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \frac{1}{R}\left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_c \\ y_c \end{bmatrix}\right) \tag{4.12}$$

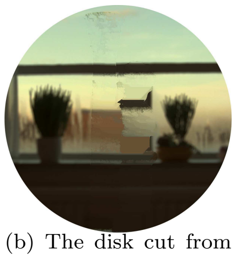
Any pixel on the disk can then be mapped onto a direction in the hemisphere by the following equation:

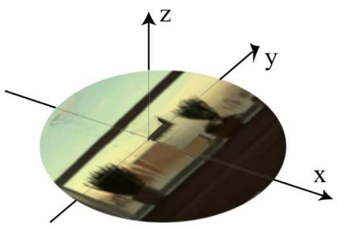
$$\vec{d} = \begin{bmatrix} x_n \\ y_n \\ 1 - \sqrt{x_n^2 + y_n^2} \end{bmatrix} \tag{4.13}$$

to which any incident light will be related as ω_i in Equation 4.15 in the next section. If we invert these equations, we get a function that maps a direction $[x_n, y_n, z_n]^T$ onto a pixel in the background image.:

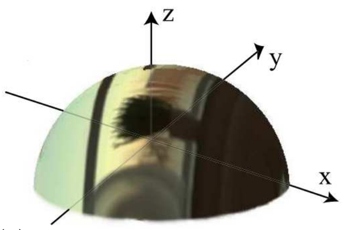
$$\begin{bmatrix} x \\ y \end{bmatrix} = R\begin{bmatrix} x_n \\ y_n \end{bmatrix} + \begin{bmatrix} x_c \\ y_c \end{bmatrix} \tag{4.14}$$


(a) Background image


(b) The disk cut from the image



(c) The disk placed in a 3D environment



(d) The disk is protruded to form a hemisphere

Figure 4.8: The construction of the environment map explained in four steps. Note that the result in (d) will be duplicated in the opposite direction to form the whole spherical environment. Images from [KRFB06].

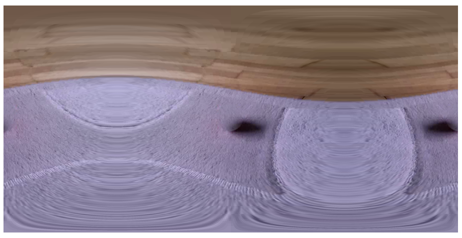
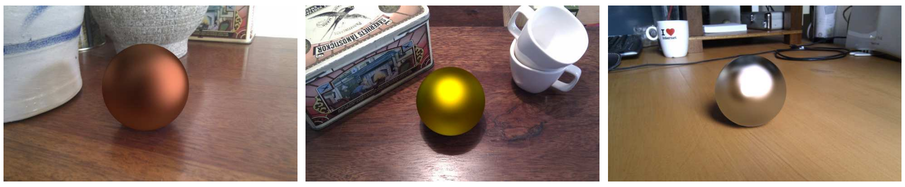


Figure 4.9: A cylindrical map projection of the environment. The dark part in the center of the image is the part of the environment in front of the viewer. The other dark part is situated behind the viewer.

Bi-linear filtering can be applied to improve the quality of the result, which is only useful for specular surfaces. From the last equation we can easily see that this is an orthographic projection. Other spherical projections may be utilized that do not distort the environment. However, the distortion of the environment is hardly noticeable; the authors note that the results will be plausible when observed indirectly.

It should be noted that there has been an improvement to the image-based illumination modeling techniques since this implementation was made. See [LE10] for more information on creating location-dependent spherical illumination modeling from a single image.



(a) Copper (b) Gold (c) Mirror

Figure 4.10: Render results using the Cook-Torrance shader. In addition to the image-based reflections, a light source was added to the environment in a plausible direction. The used material parameters are guessed, therefore they do not look too realistic. The influence of the *roughness* parameter is apparent between (a) where it is high and (c) where it is low.

4.5 Rendering

To obtain the results of Image-Based Material Editing, we need a rendering system. This system needs to take care of multiple aspects which together should yield a plausibly realistic result.[3] Firstly, we will explore a rendering equation that determines shading on surface points using a physical approach. Secondly we will see how to model material appearance by looking into the empirical Phong model and the physically inspired Cook-Torrance model.

4.5.1 The rendering equation

The rendering equation models how a point on a surface radiates light and reflects incident light, therefore it defines how we should render the surface.[Kaj86] Each surface point receives incident light from all directions in the hemisphere around the surface normal. Light coming from a direction nearly perpendicular to the surface leads to a larger irradiance than light coming from nearly parallel to the surface. Therefore, we should attenuate the intensity by taking into account the angle under which it hits the surface. This leads to a $\cos\theta_i$ factor. We can render the object by evaluating the rendering equation for every pixel where the surface is visible:

$$L(x, y) = L_e + \int_\Omega f_r(\omega_i, \omega_o) L_i(\omega_i) \cos\theta_i d\omega_i \tag{4.15}$$

In this equation, $L(x, y)$ is the synthesized radiance and consequently the intensity in pixel $[x, y]^T$. L_e is the light emitted by the material, which is set to 0 in the program, but can be set to any ambient intensity value if desired. Ω denotes the hemisphere around the surface normal visible in the pixel. ω_i and ω_o are directions of incident and reflected light respectively, parametrized

[3]The words “plausibly realistic” are used, as a perfectly realistic result is impossible to obtain. For example: we cannot accurately know about all light sources in the scene, so we should model light sources according to some user parameters. Also, we have limited knowledge about the environment in the input image, so we utilize an approximation of the environment that looks plausible, but is far from physically correct.

as $\omega = (\phi, \theta)$. f_r is the BRDF (see Section 4.5.2) of a certain material that is selected for rendering. $L_i(\omega_i)$ is the light intensity from the given direction, which is attenuated using its cosine due to the angle of incident light.

In the program, the hemisphere around the surface normal is simply sampled with a set number of sampling points; L_i comes from an approximation of the environment. f_r, the BRDF, is dependent on the material that we would like to render.

4.5.2 Material properties

If we want to render realistic looking objects, we need to know how to accurately render it using its material properties. In particular, we need to know how the material interacts with light. This is modeled in a function known as the *Bidirectional Reflectance Distribution Function* or BRDF[Nic65], and is directly used in the rendering equation. It is closely related to a reflectance map.

The BRDF is a four-dimensional function with parameters incoming and outgoing direction $\omega_i = (\phi_i, \theta_i)$ and $\omega_o = (\phi_o, \theta_o)$. The function returns the fraction of light from the incoming direction that can be observed from the outgoing direction:

$$f_r(\omega_i, \omega_o) = \frac{dL_r(\omega_o)}{dE_i(\omega_i)} = \frac{dL_r(\omega_o)}{L_i(\omega_i)\cos\theta_i d\omega_i} \tag{4.16}$$

in which L is radiance and E is irradiance. Physical BRDFs have the following three properties:

positivity: $f_r(\omega_i, \omega_o) \geq 0$

obeying Helmholtz reciprocity: $f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i)$

conservation of energy: $\forall \omega_i, \int_\Omega f_r(\omega_i, \omega_o) \cos\theta_o d\omega_o \leq 1$

The BRDF reflects natural properties of materials and therefore helps us render objects that look plausible. We can render materials with special reflective properties such as feathers or certain kinds of crystals, giving us an advantage over simpler shading models such as the Phong shading model. To obtain physically correct results, accurately recorded or well modeled BRDFs can be used. The recording of a BRDF usually happens by holding a surface of the material in a stand, lighting and recording it from different angles using an illuminant and camera respectively. By recording a sphere, many data points can be captured in a single image. The recorded data may be used directly, or converted into a mathematical model that approximates the recorded data.[GCG+05]

4.5.3 Renderer implementation

The renderer takes as arguments an Image, a Shader and a Shape. Its task is to render the Shape, using the Shader, onto the background Image. The location and dimensions of the Shape are known, and the Shape can be queried for any surface point and returns the surface normal in that point. The steps taken are described in Procedure 6.

The program iterates all pixels in which the given shape is visible. The surface normal is known from the *Shape* object. The normal should point

Procedure 6 Rendering a shape into an image.

```
**Input:** $Image$, $Shape$, $Shader$
**Output:** A visual representation of $Shape$ rendered on $Image$ using $Shader$
for all $[x, y]^T \in Shape$ do
  $\vec{N} \leftarrow Shape.getNormal(x, y)$
  if $\vec{N}$ points towards viewer then
    $[x_{im}, y_{im}]^T \leftarrow [x, y]^T + Shape.position$
    $\vec{V} \leftarrow [x_{im} - w_{im}/2, y_{im} - h_{im}/2, -z]^T$
    $I \leftarrow Shader.shadePixel(\vec{N}, \vec{V})$
    Fill the pixel at $[x_{im}, y_{im}]^T$ with color $I$
  end if
end for
```

towards the viewer (i.e. $z < 0$) otherwise it is ignored, as a form of "back face culling".

A viewer direction vector $\vec{V}$ is constructed, assuming that the viewer position is in the center of the image and with a certain z value as a distance to the image plane. w_{im} and h_{im} denote the width and height of the image, respectively. V contains the direction from the image point at $[x_{im}, y_{im}]^T$ towards the viewer.

The actual coloring of the pixel is delegated to the given *Shader* object, which accepts a normal and viewer direction and returns a color.

4.5.4 Shader implementation

There are multiple ways to implement a shader. Therefore, the *Shader* class is an interface to multiple specific implementations. Each of the implementations have the interface in common, which means that they have a method called *shadePixel* which takes a surface normal and viewer direction, and returns a color. The shader collection currently consists of the LightShader, AreaLightShader, ImageShader and CombinationShader.

The LightShader (Procedure 7) is a simple shader that takes a light source and a BRDF to do its calculations. It uses colored light at infinite distance to illuminate the surface.

Procedure 7 LightShader

```
**Input:** $BRDF$, light source direction $\vec{L}$, light source color $\vec{I_L}$, surface normal $\vec{N}$, viewer direction $\vec{V}$
**Output:** The color of the surface, illuminated from $\vec{L}$ and seen from $\vec{V}$
$I_b \leftarrow BRDF.evaluate(\vec{L}, \vec{V}, \vec{N})$
$I \leftarrow max(\vec{N} \cdot \vec{L}, 0)(I_{L,r}I_{B,r}, I_{L,g}I_{B,g}, I_{L,b}I_{B,b})$
return $I$
```

The AreaLightShader (Procedure 8) is an improved version of the LightShader, adding the property of *spread* to the light source. Where the LightShader uses a point light source, the AreaLightShader has area, giving both a more natural and a softer, more pleasing look to the result. It discretely samples points on a unit sphere, which are treated as incoming light directions. We sample ϕ from 0 to 2π in 47 steps and we sample z from -1 to 1 in 23 steps, although these numbers vary depending on the roughness of the used material; shinier surfaces need higher sampling density. We construct the incoming light vector $\vec{L}$ as $[\cos\phi\sqrt{1-z^2}, \sin\phi\sqrt{1-z^2}, z]^T$. When the dot product of $\vec{L}$ and $\vec{L_0}$ ($\vec{L_0}$ being the vector pointing to the center of the light source) is higher than a certain value corresponding to the *spread* of the light source, then $\vec{L}$ apparently lies close enough in the direction of light source and we consider $\vec{L}$ a contributing light direction.

Procedure 8 AreaLightShader

```
Input: $BRDF$, light source direction $\vec{L_0}$, light source spread $s$, light source color $\vec{I_L}$, surface normal $\vec{N}$, viewer direction $\vec{V}$
Output: The color of the surface, illuminated from the light source at $\vec{L_0}$ and seen from $\vec{V}$
$I \leftarrow$ black
$n_{samp} \leftarrow 0$
for $\phi = 0 \rightarrow 2\pi$ do
  for $z = -1 \rightarrow 1$ do
    $[x, y]^T \leftarrow [\cos\phi\sqrt{1-z^2}, \sin\phi\sqrt{1-z^2}]^T$
    $\vec{L} \leftarrow [x, y, z]^T$
    if $\vec{L} \cdot \vec{L_0} \geq (1-s)$ then
      $n_{samp} \leftarrow n_{samp} + 1$
      if $\vec{L} \cdot \vec{N} > 0$ then
        $I_B \leftarrow BRDF.evaluate(\vec{L}, \vec{V}, \vec{N})$
        $I \leftarrow I + (\vec{N} \cdot \vec{L})(I_{L,r}I_{B,r}, I_{L,g}I_{B,g}, I_{L,b}I_{B,b})$
      end if
    end if
  end for
end for
return $I/n_{samp}$
```

The ImageShader (Procedure 9) is different from the previous shaders in the respect that is does not take a light source as an input, but rather uses the image values around the object as light sources. An Environment object is constructed using the method described in Section 4.4. The shader evaluates Kajiya's rendering equation. (Equation 4.15) The sampling of directions in the unit sphere is done in a similar fashion as is the case with the AreaLightShader; we sample ϕ from 0 to 2π and we sample z from -1 to 1. The incoming light vector $\vec{L} = [\cos\phi\sqrt{1-z^2}, \sin\phi\sqrt{1-z^2}, z]^T$ should lie in the hemisphere of all incoming light directions (we simply check $\vec{L} \cdot \vec{N} > 0$), then we consider this a direction from which light illuminates our surface. The function $env.color(\vec{L})$ is a reference to Equation 4.14, performing the texture look-up for the direction vector $\vec{L}$. The rest of the rendering equation can be solved trivially:

Lastly, the CombinationShader (Procedure 10) is a meta-shader. It takes multiple Shader objects as parameters, each with an associated weight. When evaluating this Shader, it evaluates all registered Shaders and returns a weighted average of the result. Using this Shader, we can render an object using the image-based environment with the ImageShader, while also taking into account a light source such as the sun with the LightShader.

Procedure 9 ImageShader

```
**Input:** $BRDF$, Environment $env$, surface normal $\vec{N}$, viewer direction $\vec{V}$
**Output:** The color of the surface, illuminated by $env$ and seen from $\vec{V}$
$I \leftarrow$ black
$n_{samp} \leftarrow 0$
**for** $\phi = 0 \rightarrow 2\pi$ **do**
  **for** $z = -1 \rightarrow 1$ **do**
    $[x, y]^T \leftarrow [\cos\phi\sqrt{1-z^2}, \sin\phi\sqrt{1-z^2}]^T$
    $\vec{L} \leftarrow [x, y, z]^T$
    **if** $\vec{L} \cdot \vec{N} > 0$ **then**
      $I_L \leftarrow env.color(\vec{L})$
      $I_B \leftarrow BRDF.evaluate(\vec{L}, \vec{V}, \vec{N})$
      $I \leftarrow I + (\vec{N} \cdot \vec{L})(I_{L,r}I_{B,r}, I_{L,g}I_{B,g}, I_{L,b}I_{B,b})$
      $n_{samp} \leftarrow n_{samp} + 1$
    **end if**
  **end for**
**end for**
**return** $I/n_{samp}$
```

Procedure 10 CombinationShader

```
**Input:** Shaders $S$, weights $\vec{W} = \{w_1 \dots w_n\}$, surface normal $\vec{N}$, viewer direction $\vec{V}$
**Output:** The color of the surface seen from $\vec{V}$
$I \leftarrow$ black
**for all** $s \in S$ **do**
  $I_s \leftarrow s.shadePixel(\vec{N}, \vec{V})$
  $I \leftarrow I + w_s I_s$
**end for**
**return** $I$
```

4.5.5 BRDF implementation

As with the shaders, the implementation also supports different implementations of the BRDF, adhering to a standard interface. The BRDFs all have a method *evaluate* which takes the vectors $\vec{L}$, $\vec{V}$ and $\vec{N}$ and returns a color corresponding to the surface illuminated from $\vec{L}$ and observed from $\vec{V}$. In this respect it differs slightly from the BRDFs as explained in Section 4.5.2, where the BRDF is a function $f_r(\omega_i, \omega_o)$ with only two parameters. ω_i and ω_o are similar to $\vec{L}$ and $\vec{V}$, only in a local coordinate frame with the surface (and thus $\vec{N}$) in a fixed position. Instead of doing this coordinate transformation, $\vec{N}$ is given as a third parameter to the BRDF. There are three BRDFs implemented:

The PhongBRDF uses the Phong illumination model, which is an empirical model. Depending on the material properties, the color value is the sum of several terms. The first term is the ambient term, which denotes light that radiates from the object. The second term is the diffuse term, which uses incoming light direction and surface normal. The third term is the specular term which depends on incoming light direction, surface normal and viewer direction. It generates specular highlights onto the surface. This type of BRDF is given a Material object in addition to the parameters it receives from the Shader.

The formula used to determine color values is:

$$I_p = k_a i_a + \sum_{m \in lights} (k_d(\vec{L}_m \cdot \vec{N}) i_{m,d} + k_s(\vec{R}_m \cdot \vec{V})^{\alpha} i_{m,s}) \tag{4.17}$$

in which k_a, k_d and k_s are respectively the ambient, diffuse and specular weight constants. i_a is the static ambient color value of the material, whereas $i_{m,d}$ and $i_{m,s}$ are the colors of the material while illuminated by light m, which can be different for diffuse and specular reflection. α is the shininess constant, meaning highlights become smaller when this value is larger. $\vec{L}_m$ is the direction from the surface point towards light source m, $\vec{R}_m$ is the reflection of $\vec{L}_m$ in the surface normal $\vec{N}$. ($\vec{R}_m = 2(\vec{L}_m \cdot \vec{N})\vec{N} - \vec{L}_m$)

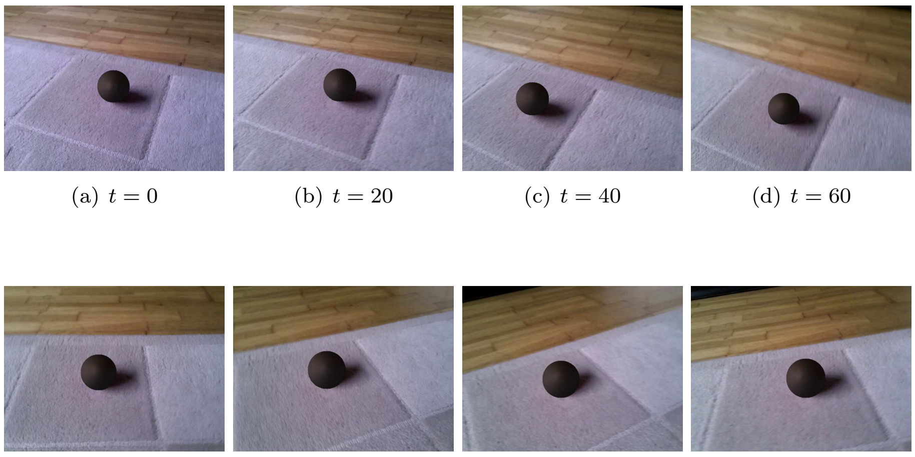
The second implemented BRDF, the Cook-Torrance reflectance model presented in [CT81], is a more physically correct model. It is primarily intended for use with metallic materials, although non-metals may also be rendered. Roughness or shininess can be adjusted in the material properties, and the model even accounts for color shift near the edge of highlights by realistically incorporating reflectance curves of materials. Without going into too much detail—the model is trivial to implement, especially with the help of [EHK+07]—I will describe the terms that together make up the model.

The model is based around the notion that metallic surfaces consist of micro facets. The roughness of the material determines how these facets are oriented in the model; a smooth surface has flat facets which are aligned, while a rough surface has standing facets that form a sawtooth shape. The *geometric* term attenuates the reflected light as facets partially mask and shadow each other. Parts of the surface of the facets receive no light because of self-shadowing, and parts of the reflected light does not reach the observer as the path is blocked by other facets.

The roughness term determines the spread of reflection of a light ray. A rough surface will produce a wider spread than a smooth surface, while a perfect mirror reflects a light ray in only one direction. The Beckmann distribution [BS63] can be used to model a wide range of materials. It is unfortunately complex and therefore costly to evaluate, leading to an approximation in the implementation.

The Fresnel term accounts for the fact that reflected light has no constant color. The exact color of the reflection is dependent on the reflection direction. Ideally, we would have a reflectance curve of every material available, which tells us exactly the relation between incoming light wavelength, viewer angle and reflected light wavelength. For simplicity, we compute this reflectance using an approximation presented by [Sch94].

Finally, the MatteBRDF is the simplest. This BRDF returns a constant value under all conditions, representing perfect Lambertian surfaces. The constant can be given in addition to the parameters that the BRDF receives from the Shader.


(a) $t = 0$ (b) $t = 20$ (c) $t = 40$ (d) $t = 60$

(e) $t = 80$ (f) $t = 100$ (g) $t = 120$ (h) $t = 140$

Figure 4.11: Results of the described pipeline. The original orange ball has been replaced by a synthetic sphere.

4.6 Results

Figure 4.11 shows the results on a short video sequence. The Phong BRDF was used for its simplicity which results in fast computation. An estimated area light source was added to the left of the scene with a fixed direction relative to the camera. The used Shaders are AreaLightShader and ImageShader. Because the light source is located in a fixed direction, the AreaLightShader contribution is more or less constant, only varying with the dimension and position of the sphere. The ImageShader also varies little as the scene is overall rather dull. When inspecting the render closely we can recognize faint influences from the carpet and wooden floor, this would be more apparent with different material properties.

This result on a ten second video was generated in roughly six minutes on a 2.8GHz dual-core machine. A breakdown of processing time into different sub-components:

Component	**Time**
Tracking	47s
Shape Recovery	7s
Inpainting	4s
Rendering[4]	5m9s

Do note that optimizations have not been focused on, therefore the program could be significantly faster. Possible optimizations include:

[4]Includes environment mapping as the computations on the environment are done during look-up by the ImageShader.

Multi-threading The implementation is completely single-threaded for simplicity. However, most components of the pipeline operate on a single frame which makes the application very suitable for parallel processing. Also by far the most time-consuming process is rendering, which in itself is suitable for parallelization. For best results the GPU should be used.

Data management Intermediate results are stored on the hard disk, the reason being that all stages are executed one after another and each stage processes every frame of the video. With different processing order, such as a sequence of stages per frame, the intermediate data size would be $\mathcal{O}(1)$ and could be stored in memory, avoiding expensive disk operations.

Micro optimizations Using a profiling tool such as *callgrind* can give detailed insights into time consumption per part of the program. It can report processor operations and estimated time consumption per method or even per line of code. The most obvious time consumers were optimized (for example by *function inlining* and changing $y \leftarrow x^3$ to $y \leftarrow x \cdot x \cdot x$) but there is still room for improvement. Optimal cache usage is also a subject which was only barely explored. Most image data access is performed in the same order as it is stored, but cache usage analysis (with tools such as *cachegrind*) could lead to more optimization.

While the drawbacks of a fixed lighting direction may not be apparent in a sequence of still images, it is more noticeable in a video result. Therefore the environment mapping (including more accurate modeling of light sources) would be an obvious improvement to the pipeline.

Chapter 5

Improvement of the environment and rendering system

The environment mapping subsystem was chosen to be improved because the simplistic image-based lighting approach is insufficient. Image-based lighting works best using high-dynamic range (HDR) images, for the simple reason that some information is lost during tone-mapping of low-dynamic range (LDR) images, especially in the important bright areas. To make the system work with LDR images and still produce a pleasing result, a directional light source was added with a direction relative to the camera instead of the scene. In moving video, the resulting error is easily noticeable.

This calls for a more elaborate scene model in which camera motion can be expressed as motion relative to the scene. The goal is to model light sources in the scene so that we can have accurate scene lighting independent of camera motion. Karsch et al. [KHFH11] provide a method that models the environment without requiring physical access to the scene. The presented implementation is based on this technique. As an added benefit, the rendering system will be greatly improved, enabling us to render detailed models with various materials.

The input to this image-based system consists of a single LDR image and an annotation of the camera orientation in the form of two vanishing points. Furthermore the position and size of the room, which is modeled as a box, are entered, and light sources are annotated by drawing polygons onto the image. After processing stages to determine room albedo and light parameters, the model of the scene is used to create a new render containing a synthetic object. Finally, the render of the synthetic object and its environmental interaction is composited back into the input image.

The first section describes the image-based approach. In the second section the transition to a video-based application will be explained.

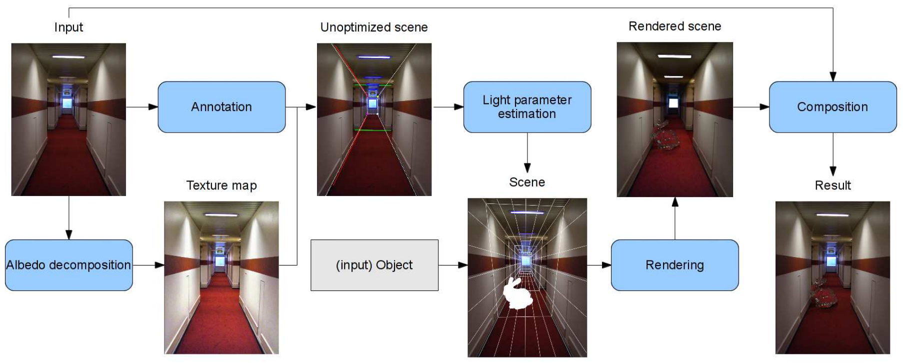


Figure 5.1: The pipeline for the environment system.

5.1 Rendering Synthetic Objects into Legacy Photographs

To achieve our goal of inserting an object into a photograph, we will create a synthetic render from which we take the object and its environmental interaction such as shadows and reflections, and composite them back into the original photograph. To create such a render, generally four things are required: a camera, scene geometry, material properties and light sources. The stages for this program are shown in Figure 5.1. An annotation stage will lead to the camera properties, scene geometry and light source locations, while an albedo decomposition method constructs a diffuse texture map to use as an approximation of the scene materials. An additional stage estimates the light source parameters such as power, giving everything needed to render the scene.

However, the scene will look far from realistic for the reason that we use a lot of approximations. Therefore, we use a composition stage. Not shown in the pipeline overview is the fact that we create two renders: one with and one *without* the object. The difference between these images is what we would like to transfer onto the original. We also create an object mask that indicates pixels where the synthetic object is present. This is needed because object pixels need to be copied directly from the object render, while for non-object pixels we copy the difference between the renders.

There are a few differences between this implementation and the original. First of all, the albedo decomposition is simplistic: this implementation just performs the Retinex[LM+71] algorithm to obtain the texture maps, while the original implementation computes an irradiance map based on textures and scene geometry, and uses that to estimate the scene albedo based on the Lambertian assumption.[1]

Second, the original implementation supports multiple types of geometry:

[1]This approach was tried, as explained in Section 5.1.2, but it would take extremely long to obtain a noise-free irradiance map. This is based on an experimental implementation of an irradiance shader in the simplistic rendering system described in Section 4.5.3.

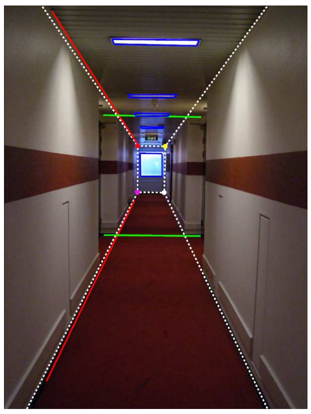


Figure 5.2: User annotation. Red and green lines are sets of vanishing lines, leading to vanishing points used for camera orientation and focal length. Blue polygons are light sources. The white dashed wireframe box is the model of the room. The user positioned and scaled the box using the keyboard.

the room, the synthetic object(s) and any number of protruding surfaces or occluding models. A protruding surface is a 2D planar polygon parallel to the floor of the room, with an added height vector to make it a 3D object. For simplicity only the room and inserted objects are supported. Protruding surfaces and occluding models would add additional complexity to the video-based approach and were not implemented for that reason. While inserting (non-occluded) objects into images, the results are quite satisfactory although the objects would have to be supported by a protruding surface.

Another difference from the original implementation is the light parameter estimation. It uses a similar approach, namely to render the scene and adjust light parameters to minimize an objective function, but it uses a different objective function.

Finally, the logic behind the composition stage is exploited to greatly improve rendering time. While the original authors rendered their images as 500 to 1000 samples per pixel, this number has been brought down to 32 or sometimes fewer. This makes it feasible to use the software on consumer-grade hardware with reasonable rendering times. This is made possible by a small change, for which the reason is explained in Section 5.1.4.

5.1.1 Scene annotation

The technique assumes an interior scene which is roughly shaped like a box. Although the technique performs well on scenes that totally do not fit this description, a textured, box-shaped room is the internal representation of the scene. The user annotation consists of camera orientation and focal length, translation, room size and the position of light sources. These are entered via a user annotation interface shown in Figure 5.2.

To know the orientation of the box with respect to the camera, the user annotates two vanishing points. The points should be chosen such that the respective vanishing lines leading to them are perpendicular in three-dimensional space.

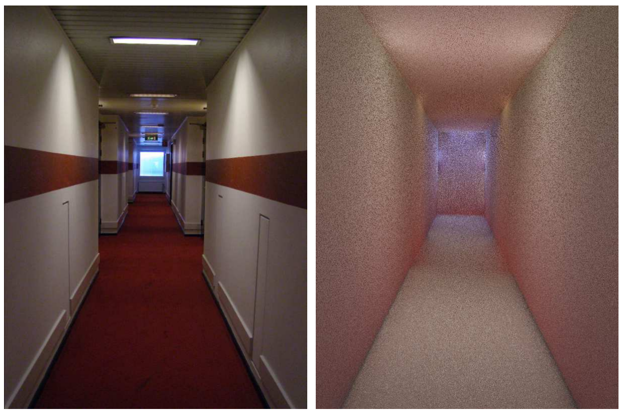


Figure 5.3: Irradiance map. At every pixel of the room, the irradiance is computed by sampling the hemisphere around that point in three dimensional space. This gives a representation of the amount of indirect light received by the point visible at that pixel. For example, the floor receives most light from the walls, yielding white, while the lower part of the walls are more red because of the carpet. The noise can be reduced by increasing render time.

Using this assumption, we obtain the camera's focal length and subsequently the orientation of the camera. The used technique is detailed in [GMMB00].

Once the camera has been oriented properly, the user manipulates the position and scale of the box using the keyboard. Six keys are assigned to position and six keys to scale. Once the preview of the box lines up with the input image visually, the annotation of the camera and box are complete.

Finally, the user draws polygons onto the image that denote the light sources. The polygons are then projected onto the box, leading to a simplification that light sources are flat and situated onto the walls. The pixels inside the polygons are inspected to give light sources their color. It should be noted that this can be erroneous because the input is LDR and therefore tone-mapped.

5.1.2 Texture mapping

To obtain a reasonable representation of the scene albedo from the input image, the Retinex algorithm is performed. This algorithm works on an image and also returns an image. The original authors construct the albedo map not by using Retinex, but by using knowledge of the scene geometry. They create an irradiance map measuring the amount of indirect light at every scene point. Using the irradiance map together with the assumption that the room materials are Lambertian, they obtain a better estimate for the albedo. This method was implemented, for which the results are shown in Figure 5.3, but it is too intensive to compute the irradiance map on the used hardware[2], therefore the approach was discarded and replaced with Retinex.

Once we have a good estimation of the albedo of the room, we need to prepare it for re-rendering. The goal is to construct texture maps that can be

[2]The implementation is single-threaded and was executed on a 2.8GHz CPU.

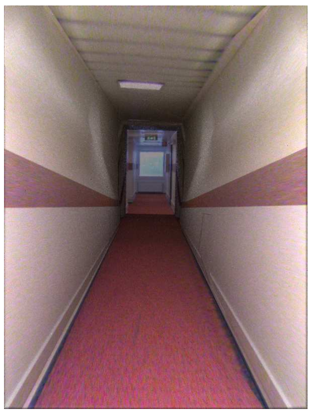


Figure 5.4: Incorrect texture mapping obtained by using the input image directly as a *uv*-map. Note: the colors in this image are inaccurate, because of an unrelated problem with texture gamma at the time, which is explained in Section 5.1.4.

used as *uv*-maps in the rendering pipeline. My initial idea was to project the corners of the box onto the image plane to get their *uv* parameters, and use the input image directly as a texture map. The result is apparent in Figure 5.4.

The issue with this approach is that the texture map should be a frontal orthographic view of the surface. There are two workarounds for this problem: to divide the polygons into a large number of smaller triangles, or to use a different kind of texture mapping interpolation.

The preferable solution to this problem is to rectify the texture of each face of the box separately, then use the rectangular texture maps during rendering. To rectify an image, The four corners of the box face are obtained in image coordinates and applied an homography to map them to $(0, 0), (w, 0), (0, h), (w, h)$ respectively. w and h are chosen to be 1000 pixels which is sufficient for the application. The result is shown in Figure 5.5.

5.1.3 Light parameter estimation

In order to obtain plausible illumination for the scene, light parameters must be estimated. The most influential property of light sources is light power, as a very dark or very bright render will not integrate into the photograph well, especially when the rendered objects are transparent or reflective. The reason for this is that synthetic object pixels are copied directly onto the input by the compositor, as explained in the next section. To a lesser extent, parameters such as light location and color could be optimized, but those are kept at their initial values in the current implementation.

Light power is estimated using a hill-climbing approach. The power is increased or decreased to match the average render intensity to intensity of the input image. In every iteration, the light power is multiplied by an inverse sigmoid factor obtained from the error in the intensity:

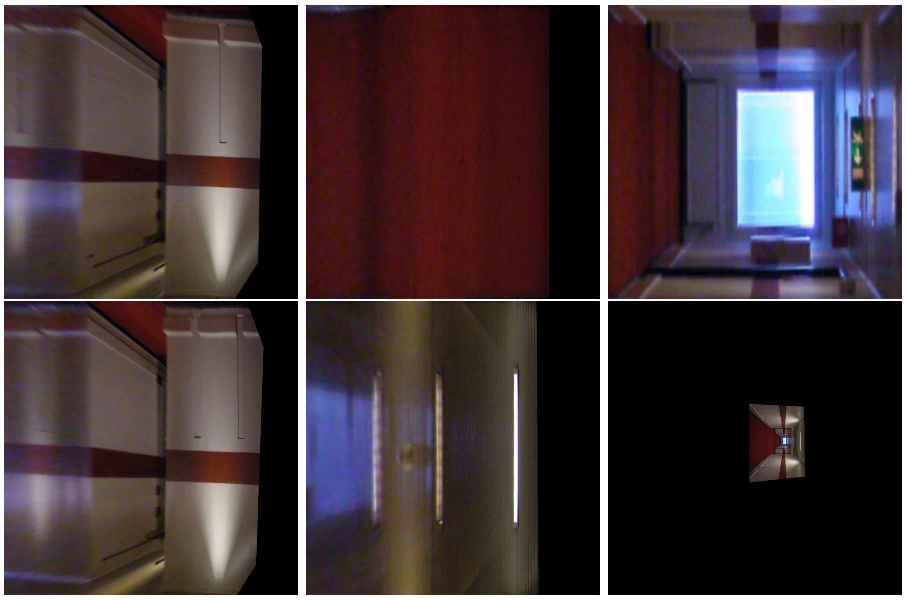


Figure 5.5: The texture maps for each side of the box. From left to right, top to bottom: left wall, floor, wall at the back, right wall, ceiling, wall behind the camera. The last one results in an inverted view of the scene because the points are behind the camera, but still projected onto the image plane by dividing by their depth. Black parts in the texture maps are regions outside of the input image, causing problems with reflections. An inpainting method could be used to fill them in.

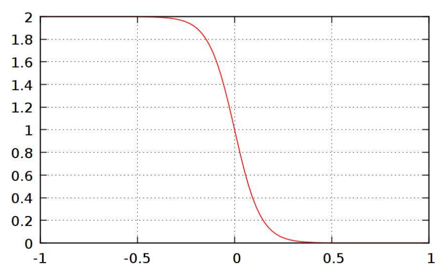


Figure 5.6: Visualization of Equation 5.1: the adjustment factor for light power, as a function of average intensity error.

$$P' = P \cdot 2 \cdot \left(1 - \frac{1}{1 + e^{-error \cdot 15}}\right) \tag{5.1}$$

in which *error* is the difference in average intensity of the render and the original image, which is between -1 and 1. The optimization stops when the absolute error is below a threshold (0.001) or when a maximum number of iterations (5) has been reached. These limits, as well as the coefficient in the exponent, are chosen empirically. All lights in the scene are simultaneously affected by this operation.

To speed up computation, the renders for this procedure are produced on a lower resolution as the final image. Also, the number of samples per pixel is lowered and inter-reflections are not computed, yielding only direct light.

5.1.4 Rendering

The rendering stage makes use of the open-source LuxRender[3] software. While the original authors generate LuxRender scene files for the artist to insert objects into, this implementation uses the LuxRender 0.9 C++ API. This led to some implementation difficulties, but it allows to integrate LuxRender work into the pipeline in order to perform video processing without user intervention. A small number of adjustments to the source code were required, such as a query method to check whether the system has initialized and whether rendering has been finished. Also several tweaks had to be made to the build files in order to build on the used Linux distribution.

With the camera properties, scene geometry, materials and lights known, all data is present for rendering. Some renders are shown in Figure 5.7. During the composition stage, explained later, we focus on extracting the rendered objects and inserting them into the original photograph, greatly enhancing the quality of the result.

Implementation details

To render an image, we create a render instance and post a render job to it. We assign one thread to the render so it will render in the background while the pipeline handles other tasks. The various LuxRender settings are as follows:

[3] `http://www.luxrender.net/`

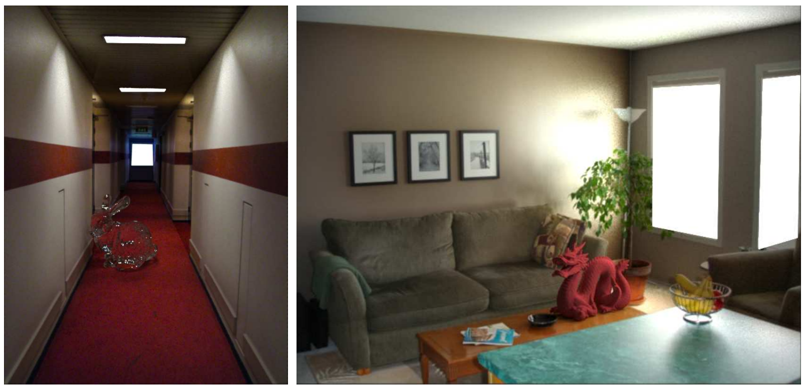


Figure 5.7: Outputs of the rendering stage.

Renderer is set to *sampler*, which renders using the CPU only.

Sampler is set to *lowdiscrepancy*. The sampler chooses how and where to pick a pixel to render. It is required to use a “dumb” sampler, i.e. one that samples each pixel reliably, as opposed to prioritizing certain pixels based on the scene contents. When using a “smart” sampler, we will run into problems during composition. Using this sampler also ensures predictable running times.

VolumeIntegrator is set to *multi* for scenes requiring volumetric effects. This is only enabled when one or more models in the scene have a transparent or translucent material.

SurfaceIntegrator is set to *exphotonmap*. The SurfaceIntegrator computes the irradiance of a point by interacting with light sources and materials. This setting is very influential on the quality of the result, especially when using reflecting or refracting materials. This particular integrator produces very smooth results while *bidirectional*, which is commonly used, produces noisy reflections and refractions when rendering only a low number of samples.

PixelFilter is set to *gaussian*. A pixelfilter is needed to collect sampled points on the image plane and assign values to individual pixels. A sample is typically assigned to the pixel it lies in, but it can also influence neighboring pixels. The gaussian filter applies a gaussian kernel to produce a smooth result, at the cost of slightly more computation time. The radius is the gaussian kernel is 2 pixels.

Accelerator is set to *qbvh*. This applies a data structure to the scene geometry in order to accelerate look-up.

Film is set to *fleximage*. An important setting of the film is *haltspp*, instructing the renderer to stop after reaching a certain number of samples per pixel. 32 was used in this implementation. Another notable setting is tone-mapping: we need to apply identical tone-mapping to our renders if we want to render movies. The tone-mapping kernel is set to *linear* with parameters *sensitivity* 320, *f-stop* 2.8 and *gamma* 2.2. Furthermore, *outlierrejection* is used with $k = 5$ to avoid "fireflies", i.e. very bright artifacts on the render.

As for the textures of the room, each face of the bounding box is assigned one of the rectified textures. It should be noted that the LuxRender manual ensures a default gamma of 2.2, but this is not respected. Therefore, the gamma of the texture maps (using the *imagemap* texture) should be manually set to 2.2. Omitting this setting will result in "washed out" texture maps.

Compositing

After the image has been rendered, call it I_{obj}, the rendered object needs to be composited into the original photograph, I_{input}. A mask M is generated with value 1 where a synthetic object is present, value 0 where no object is present, and a value in between for pixels where the object is partially visible, in order to achieve sub-pixel accuracy. The mask is obtained by projecting the object geometry onto the image plane, storing the projection to an image with high resolution as a means of super-sampling, then downscaling to the original resolution. The chosen resolution is 8 times the original resolution in both dimensions, yielding 64 times the number of pixels. Lower might suffice, but the operation is very fast compared to the actual rendering. The object projection is implemented in a *SimpleRenderer*, which was already implemented to obtain debug images that do not warrant the use of more sophisticated software such as OpenGL or LuxRender. Without regard to depth, every object triangle is transformed to image plane space and the pixels contained in the triangles are set to 1.

Apart from the object, we also want to composite the influence that the object has on its environment. To make this possible, we create an additional high-quality render of the scene, but without the synthetic object. We call it I_{noObj}. The difference $I_{obj} - I_{noObj}$ represents the environmental effect and can be composited by simple addition. This approach is called *additive differential rendering* by [Deb98]:

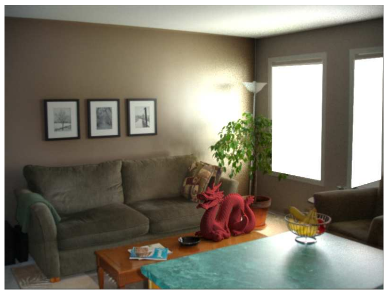
$$I_{result} = M \odot I_{obj} + (1 - M) \odot (I_{input} + I_{obj} - I_{noObj}) \tag{5.2}$$

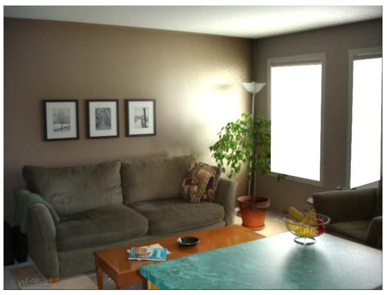
where $\odot$ is the Hadamard product. See Figure 5.8 for sample images.

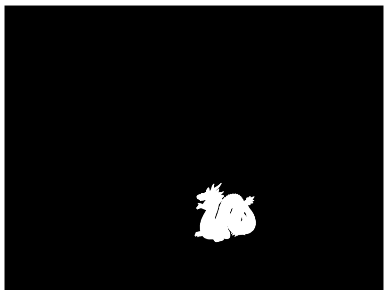
This method has one vulnerability which leads to a dramatic impact on the result: noise. LuxRender samples points randomly during rendering, which introduces noise in the rendered image. The noise pattern between I_{obj} and I_{noObj} is different, which leads to a noise pattern in $I_{obj} - I_{noObj}$ with potentially twice the amplitude of its components, all across the image. Karsch et al. work around this problem by increasing the number of samples per pixel, which naturally increases render time. They used between 500 and 1000 samples per pixel and rendered the images using a render server cluster. Using current consumer-grade hardware, this approach leads to render times of approximately five to ten hours.

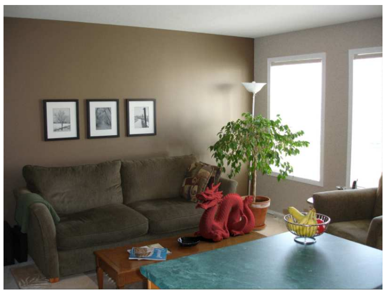
By making a slight adjustment, the number of samples per pixel can be brought down to 32, making it feasible to run on lower-end hardware and also

(a) I_{input}


(b) I_{obj}


(c) I_{noObj}


(d) M


(e) I_{result}

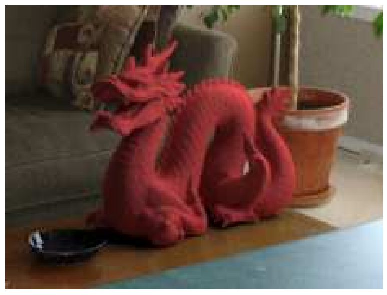

(f) Closeup of I_{result}

Figure 5.8: Sample images involved in the compositing stage.

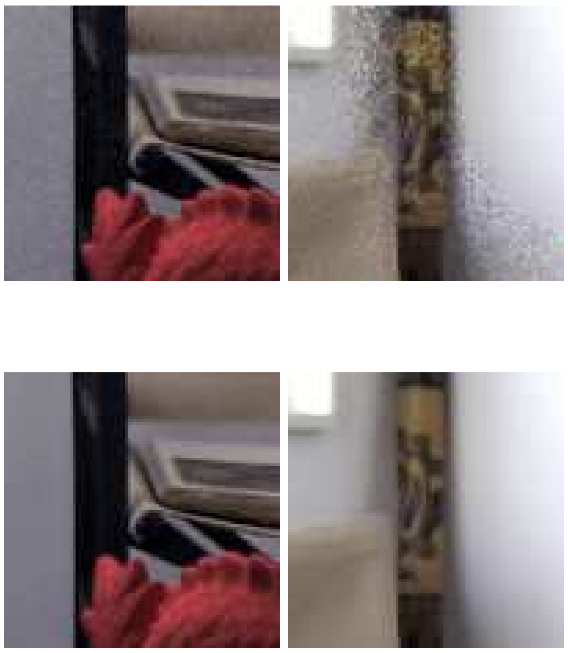


Figure 5.9: Comparison of I_{result} for inconsistent and consistent noise patterns. In the top row no attention is given to the distribution of samples by the renderer. The end result is noisy even in areas where no object (red figure) is present. In the bottom row the noise pattern is consistent between I_{obj} and I_{noobj}, which is canceled out in the subtraction. The images were rendered at 32 samples per pixel.

making it feasible to render video later in the project. Instead of letting LuxRender choose the seed for its random number generator, the seed should be set to the same value *for the renders of* I_{obj} *and* I_{noObj}. This ensures a consistent noise pattern for the renders, leading to noise cancellation in $I_{obj} - I_{noObj}$. This increase in quality enables us to render far fewer samples per pixel. A static seed also ensures that we have a consistent noise pattern between frames in a rendered video: a desirable feature in animation rendering. The result is shown in Figure 5.9.

This adjustment imposes one practical constraint however: we can only render an image using one thread. That means that hardware, multi-core and network rendering is out of the question. The reason for this is presumably a bugged or incomplete implementation of pseudo random number distribution over multiple concurrent processes. This might be solved by changing the way LuxRender chooses its samples or distributes the random numbers in a multi-threaded rendering situation. This might have been changed in newer versions of LuxRender. However, by executing the render jobs for I_{obj} and I_{noObj} on separate concurrent threads, we can still take advantage of a dual-core processor. Rendering is well parallelizable in the case of animation rendering; each frame in the animation can be rendered using up to two threads.

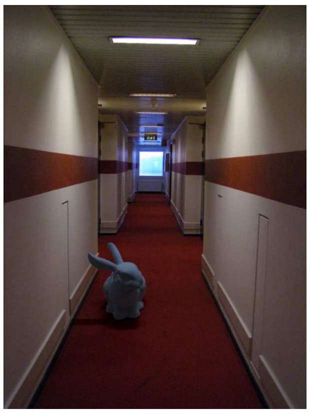
5.1.5 Results

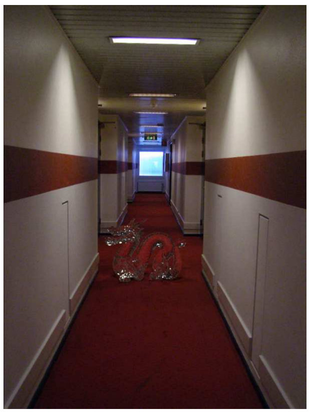
Figure 5.10 shows results of the pipeline discussed in this section.

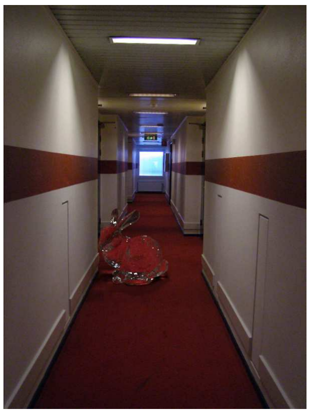
The images were rendered on a desktop PC with a 2.8GHz AMD Athlon dual-core processor. Rendering an image, 603 by 804 pixels, containing two glass models from input to end result takes about 20 minutes, at 32 samples per pixel. When using materials that do not require volumetric rendering, the render time is about 15 minutes.

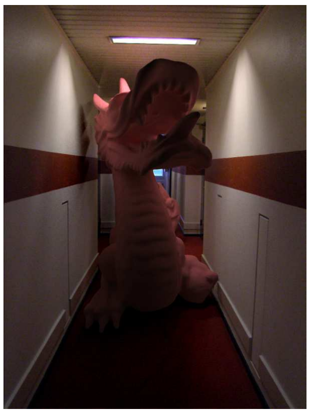
The models used in this project are taken from the Stanford 3D Scanning Repository[4]. Particular publications from which the used models were made available are [TL94] and [CL96]. The Thai Statue was provided by XYZ RGB Inc.

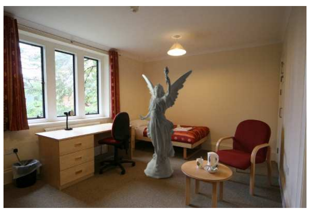
[4]`http://graphics.stanford.edu/data/3Dscanrep/`


(a) Matte bunny, 69,451 faces


(b) Glass dragon, 871,414 faces


(c) Ice bunny. The index of refraction is lower, leading to fewer specular highlights than is the case with glass.


(d) Large matte dragon. Note that the shadow on the left wall follows the original lighting. The light in the back casts shadows, but the light in the front eliminates them.


(e) Matte angel, 525,814 faces

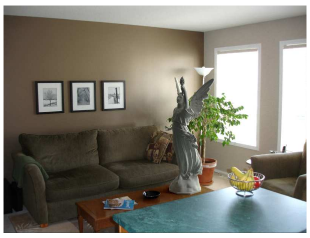

(f) Matte angel

Figure 5.10: Results of rendering synthetic objects into legacy photographs.

5.2 Integration into the video-based pipeline

In order to convert the aforementioned rendering pipeline to render synthetic objects into video, an additional piece of information is needed: accurate knowledge of camera parameters over time. The inserted object has a fixed position in the virtual world; it should also appear to have a fixed position in the end result. Inaccurate camera tracking will have a big impact on the plausibility of the result when we expect the object to have a fixed position.

This section will detail the transition from image to animation rendering by explaining the consequences for environment mapping and camera tracking. The section will be concluded by showing the results and a qualitative evaluation thereof.

5.2.1 Environment Mapping

The first frame of the input video is presented to the user to annotate the environment. This frame does not necessarily have to be in the final result; the user can determine a range of frames to process. This makes it convenient to start a video from a convenient position in the environment, before possibly filming parts of the environment that would be very difficult to annotate in just one frame.

As is the case with the image-based approach, the user annotation results in, among other data, the camera's focal length. It is assumed that the focal length of the camera remains constant throughout the video. Theoretically this constraint is not very strong, but practically the focal length of the camera is used to guide the camera tracking system, which can handle a variable focal length, in order to obtain more accurate results.

While the environment of the scene remains constant, and therefore in theory also our model of the environment, illumination conditions can change throughout the video. For example, illumination from outside can change due to clouds, or the camera's auto-exposure setting can change depending on the brightness of the image. While this should theoretically not influence the decomposed albedo maps, in practice the result is improved by recalculating the texture maps (as detailed in Section 5.1.2) in every frame. This gives a visually consistent result for transparent and reflective objects[5], even without recomputing light source power or virtual camera exposure. The difference in albedo maps is illustrated in Figure 5.11.

The problem of changing illumination conditions opens up the idea of improved albedo decomposition by solving for every available frame. Another possible improvement to the environment mapping, left for a follow-up project, is to fill in unknown areas of the texture maps with data obtained from past or future frames. This would improve the quality of rendered reflective objects, as the part behind the camera is always unknown in the current implementation.

[5]Recall that environmental influence of the object's presence, such as shadows, are composed into the original image differentially, yielding a good result in most illumination conditions. However, object pixels are directly copied onto the original image. This means that when rendering for example a glass object, it is very important that the virtual world behind the object, visible through the glass, appears consistent with the input image.

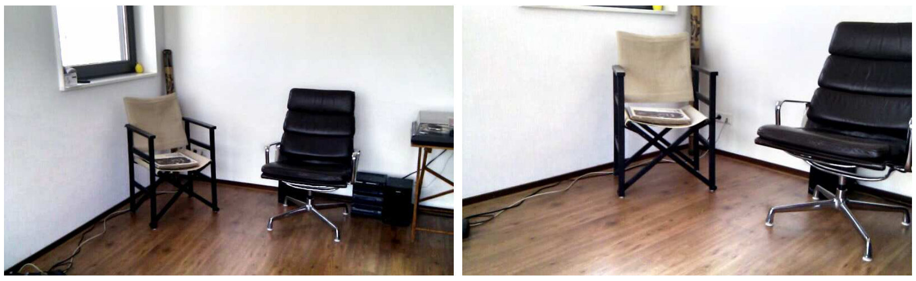


Figure 5.11: The albedo maps in two different frames. Obtained by performing the Retinex algorithm separately on a frame-by-frame basis. While albedo should ideally remain constant for a particular scene, differences can be large, e.g. on the left wall and on the floor.

5.2.2 Camera Tracking

Beginning to work on the topic of camera tracking, also known as extrinsic camera calibration, several different approaches were tried that seemed simple, logical and hypothetically correct. The very first try was a homebrew approach, detecting key points with SIFT[Low04], projecting them onto the virtual world in the first frame, and optimizing camera orientation and translation to minimize the re-projection error every in subsequent frames. Problems soon started to become apparent, for example when the camera would move away onto other parts of the scene. The second approach tried to amend this by minimizing re-projection error based on key points from the previous frame. This approach, along with hybrid mixes, suffered from cumulative errors. Furthermore, overall the results would contain high-frequency errors, resulting in a very shaky camera while it should in fact be smooth. An approach using `cvFindExtrinsicCameraParams2` from the OpenCV library also did not prove useful, presumably because it needs planar data. Other tried approaches include decomposing the essential matrix[6] into a rotation matrix and translation direction, according to [Har92], [ATLB04]. However, the results were also not satisfactory, partly because they depended on a primitive matching of SIFT key points.

The problem of camera tracking in the pipeline has finally been solved by using *Automatic Camera Tracking System* (ACTS).[ZQA+06] This software can extract the wanted camera extrinsics, along with other data such as focal length, camera intrinsics, interesting key points and even dense depth recovery. It does so while typically needing little to no user-defined parameters. It detects either SIFT[Low04] or KLT[LK+81, TK91] features (based on user preference) in every frame, filters them based on an epipolar constraint, then chains nearby features in neighboring frames together. Such a chain corresponds to a 3D point. The length of the chains is thresholded to determine whether the chain is used in camera motion reconstruction.

In the experiments it became apparent that the calculation can in some cases yield unreasonable results or even become stuck indefinitely. Upon closer

[6]The *essential matrix*, closely related to the *fundamental matrix*, contains the relation between two different views of the same scene

inspection this happens when not constraining the camera focal length, in which case the software tries to determine it automatically. Since we know the camera focal length in advance, by means of user annotation of the scene, we can constrain the focal length to this constant value, giving a higher chance of a successful tracking result.

While the program is free to use for academic purposes, the source code is not released and the executable is available for Windows only. Therefore it was not tightly integrated into the rest of the program. The user must therefore perform camera tracking using ACTS, then copy some generated output files over and load them into the pipeline.

To use the detected extrinsic camera parameters, a conversion must be made. While the rotation matrix can be used directly, the translation is only correct up to a scale factor. Ideally, this factor is constant for every point in the scene. It is computed by taking 2D features from ACTS, projecting them back onto the virtual scene, measuring the depth and dividing that by the corresponding reported depth of the 3D ACTS point:

$$s = \frac{1}{N} \sum_{i=1}^{N} \frac{\|backproject(feature_{i_{2D}})\|}{\|feature_{i_{3D}}\|} \tag{5.3}$$

s can then be used to multiply the translation vectors. The complete camera transformation thus becomes:

$$\mathbf{T} = [\mathbf{R}_{acts} | s \cdot \vec{t}_{acts}] \cdot \mathbf{T}_0 \tag{5.4}$$

with $\mathbf{T}_0$ representing the initial camera pose obtained from the annotation stage.

5.2.3 Evaluation and discussion

A qualitative evaluation was performed in the form of an on-line survey. The participants were presented six rendered videos which are shown in Appendix A in the given order. For each video, the participant was asked to give three grades from 1 to 10: one for overall realism/plausibility, one for lighting accuracy and one for object placement accuracy. The goal of this distinction is to gain separate ratings for lighting interactions and camera tracking. Optionally the participants could also leave comments for each video. In total, 19 people filled out the survey. The average grades are shown below.

Video	**Overall**	**Lighting**	**Placement**
Angel in synthetic room	7.00	7.79	7.16
Dragon in corridor	7.42	7.58	6.95
Bunny in room	7.37	7.79	6.89
Dragon in room	7.74	8.16	7.32
Thai statue in corridor	6.11	6.74	5.53
Thai statue in hall	6.37	6.89	5.89
Average	7.00	7.49	6.62

For the first video, *Angel in synthetic room*, the most common comment is that shadows seem to be missing. Indeed, the intensity difference caused by the shadow is very small, and the shadow falls mostly onto the dark blue floor tile. Very notable is that the camera tracking on this scene is practically perfect, and the object indeed appears to remain in place constantly. However, the average grade given for placement is only 7.16, suggesting that the wording of the *placement* question was poorly chosen. None of the comments mention object instability.

The *Dragon in corridor* video received relatively little feedback. One comment says that the position is off a bit, another comment says that the material of the dragon (dark red *matte*) is unrealistic.

On the contrary, *Bunny in room* received largely negative feedback on the stability of the bunny in the scene; participants correctly commented that the bunny *hovers* above the ground.

Dragon in room received the highest grades overall. Only one participant commented on the hover problem, while the camera parameters were the same as those used for the previous video. Presumably the error is less noticeable for larger objects. Two comments said that the dark part of the dragon looks too dark, which is true because there is no synthetic light source there while the real scene has a large window on that side. Two comments said that the insertion is practically indistinguishable from reality.

As for *Thai statue in corridor* and *Thai statue in hall*, complaints were mainly about instability of the object placement. These videos were recorded with quite some shaking which impacted the quality of the camera tracking results.

Chapter 6

Conclusion

The implementation of a rudimentary pipeline to perform Video-Based Material Editing is shown, which served as a motivation for the more widely applicable improvement made in artificial object rendering into video. The resulting system is physics-based and requires little input which can have low quality. The integration of the rendering improvement into the VBME pipeline is trivial.

The system inserting objects into video can perform really well depending on the circumstances. As the user survey has shown, some viewers were fooled by the realism of the results, however, certain results were less convincing. This is mainly dependent on the accuracy of camera tracking and the conversion of extrinsic camera parameters into the scene model. A significant reduction in render time has been achieved by carefully choosing which samples to evaluate during rendering. This reduced the render time by a factor 32.

6.1 Future work

The next suggested improvement for the VBME pipeline is automatic 3D shape recovery to obtain the target object's shape. The VideoTrace technique could be used, in which an object is interactively modeled from a video sequence with the help of user annotation.[vdHDT+07] The dense depth recovery from ACTS could also be used for this. This technique could be extended to obtain the shape of the environment as well, which has been done previously in [ZQA+06].

One vulnerability of the current implementation is the accuracy of the camera tracking which is essential for a plausible animation. While ACTS performs reasonably well, the integration of camera tracking results into the rest of the pipeline could be improved. As it stands right now, the annotator must be very careful to annotate the floor correctly; modeling the floor too high or too low will appear to "sway" the object as the camera translates perpendicular to the view axis. Furthermore the calculation of the scale factor in Equation 5.3 could use work. Perhaps it is worth looking into proper weights or the selection of certain features to use in the calculation.

Currently the environment out of view is not taken into account; during each frame a new texture map is computed and projected onto the scene. Reuse of texture data from previous and future frames could prove helpful, along with auto-completion of regions of the box that never appear in view. If a larger

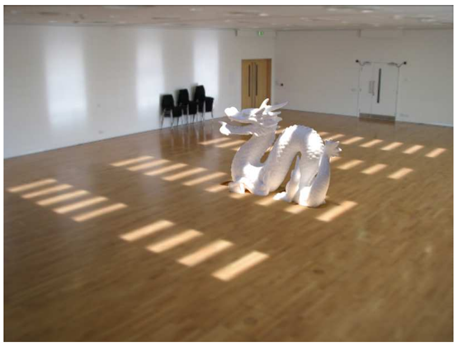


Figure 6.1: Light shafts are handled in the image-based technique with the help of relatively little user annotation. This allows for more complex interaction between the environment and the synthetic object. Image from [KHFH11].

portion of the box is textured, rendering of reflecting materials would be more realistic. Another suggestion for future projects is to exploit all available video data for the albedo decomposition, so that we obtain a more accurate model of the materials in the scene.

Light shafts in the video are not taken into account as they are in the image-based approach of Karsch et al. The addition of a video-based version of this technique would increase realism in the case that the synthetic object is placed into a shaft of light with a possibly complex shape. An example is shown in Figure 6.1.

Bibliography

[ATLB04] Benjamin Albouy, Sylvie Treuillet, Yves Lucas, and Dimitar Birov. Fundamental matrix estimation revisited through a global 3d reconstruction framework. In *Advanced concepts for intelligent vision systems*, pages 185–192, 2004.

[BN76] James F Blinn and Martin E Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[BS63] P. Beckmann and A. Spizzichino. The scattering of electromagnetic waves from rough surface. 1963.

[CL96] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312. ACM, 1996.

[CPT04] A. Criminisi, P. Pérez, and K. Toyama. Region filling and object removal by exemplar-based image inpainting. *Image Processing, IEEE Transactions on*, 13(9):1200–1212, 2004.

[CT81] R.L. Cook and K.E. Torrance. A reflectance model for computer graphics. In *ACM SIGGRAPH Computer Graphics*, volume 15, pages 307–316. ACM, 1981.

[Deb98] P. 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.

[Deb02] Paul Debevec. Image-based lighting. *IEEE Computer Graphics and Applications*, 22(2):26–34, 2002.

[DM97] Paul E. Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. *SIGGRAPH 97*, August 1997.

[EHK$^+$07] W. Engel, J. Hoxley, R. Kornmann, N. Suni, and J. Zink. *Programming vertex, geometry and pixel shaders.* Charles River Media, 2007.

[GCG$^+$05] Johannes Günther, Tongbo Chen, Michael Goesele, Ingo Wald, and Hans-Peter Seidel. Efficient acquisition and realistic rendering of car paint. In Günther Greiner, Joachim Hornegger, Heinrich Niemann, and Marc Stamminger, editors, *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005*, pages 487–494. Akademische Verlagsgesellschaft Aka GmbH, November 2005.

[GMMB00] E. Guillou, D. Meneveaux, E. Maisel, and K. Bouatouch. Using vanishing points for camera calibration and coarse 3d reconstruction from a single image. *The Visual Computer*, 16(7):396–410, 2000.

[Har92] Richard Hartley. Estimation of relative camera positions for uncalibrated cameras. In *Computer Vision—ECCV'92*, pages 579–587. Springer, 1992.

[Hor70] B.K.P. Horn. Shape from shading: A method for obtaining the shape of a smooth opaque object from one view. 1970.

[Hor86] B. Horn. *Robot vision.* The MIT Press, 1986.

[Kaj86] J.T. Kajiya. The rendering equation. *ACM SIGGRAPH Computer Graphics*, 20(4):143–150, 1986.

[KHFH11] Kevin Karsch, Varsha Hedau, David Forsyth, and Derek Hoiem. Rendering synthetic objects into legacy photographs. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 157:1–157:12, New York, NY, USA, 2011. ACM.

[KRFB06] E.A. Khan, E. Reinhard, R.W. Fleming, and H.H. Bülthoff. Image-based material editing. *ACM Transactions on Graphics (TOG)*, 25(3):654–663, 2006.

[KSW10] T.H. Kwok, H. Sheung, and C.C.L. Wang. Fast query for exemplar-based image completion. *Image Processing, IEEE Transactions on*, 19(12):3106–3115, 2010.

[KZN08] N. Kumar, L. Zhang, and S. Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? *Computer Vision–ECCV 2008*, pages 364–378, 2008.

[LE10] Jean-François Lalonde and Alexei A. Efros. Synthesizing environment maps from a single image. Technical Report CMU-RI-TR-10-24, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, July 2010.

[LK$^+$81] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence*, 1981.

[LM$^+$71] E.H. Land, J.J. McCann, et al. Lightness and retinex theory. *Journal of the Optical society of America*, 61(1):1–11, 1971.

[Low04] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[Nic65] F.E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied Optics*, 4(7):767–773, 1965.

[OCS05] Y. Ostrovsky, P. Cavanagh, and P. Sinha. Perceiving illumination inconsistencies in scenes. *Perception*, 34(11):1301, 2005.

[Sch94] C. Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

[Ser82] J. Serra. *Image analysis and mathematical morphology.* London.: Academic Press.[Review by Fensen, EB in: J. Microsc. 131 (1983) 258.] Review article General article, Technique Microscopy Staining, Mathematics, Cell size (PMBD, 185707888), 1982.

[TK91] Carlo Tomasi and Takeo Kanade. *Detection and tracking of point features.* School of Computer Science, Carnegie Mellon Univ., 1991.

[TL94] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 311–318. ACM, 1994.

[vdHDT$^+$07] Anton van den Hengel, Anthony Dick, Thorsten Thormählen, Ben Ward, and Philip HS Torr. Videotrace: rapid interactive scene modelling from video. In *ACM Transactions on Graphics (TOG)*, volume 26, page 86. ACM, 2007.

[Yia93] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

[ZQA$^+$06] Guofeng Zhang, Xueying Qin, Xiaobo An, Wei Chen, and Hujun Bao. As-consistent-as-possible compositing of virtual objects and video sequences. *Computer Animation and Virtual Worlds*, 17(3-4):305–314, 2006.

Appendix A

Evaluated videos

These videos, all approximately ten seconds long, were used in the qualitative evaluation detailed in Section 5.2.3.

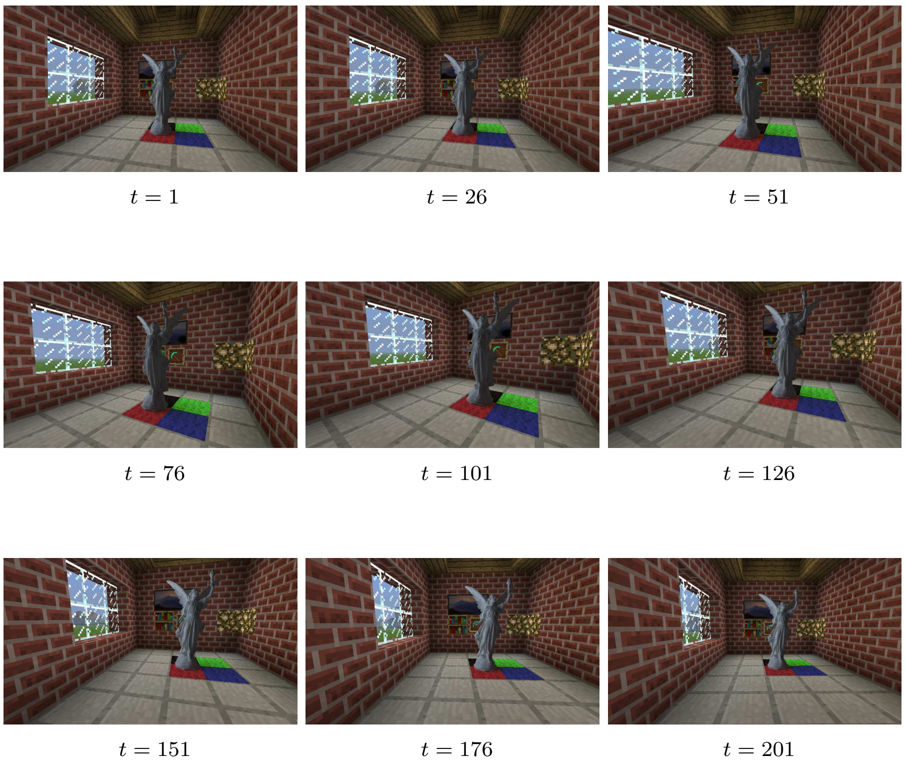
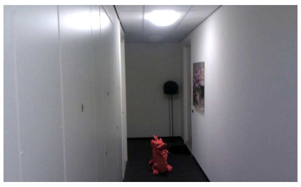
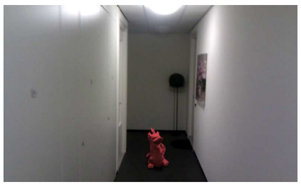


Figure A.1: Angel statue inserted into synthetic scene, recorded in the game *Minecraft*.

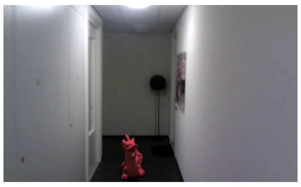
$t = 1$



$t = 25$



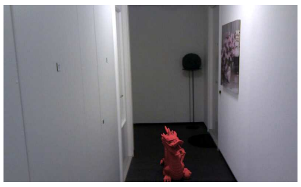
$t = 49$



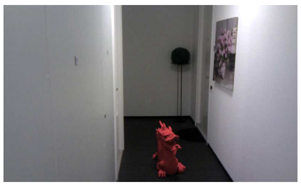
$t = 73$

$t = 97$

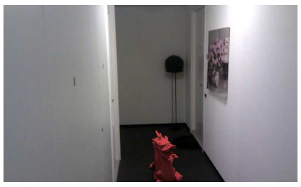
$t = 121$



$t = 145$



$t = 169$



$t = 192$

Figure A.2: Dragon statue inserted into corridor.

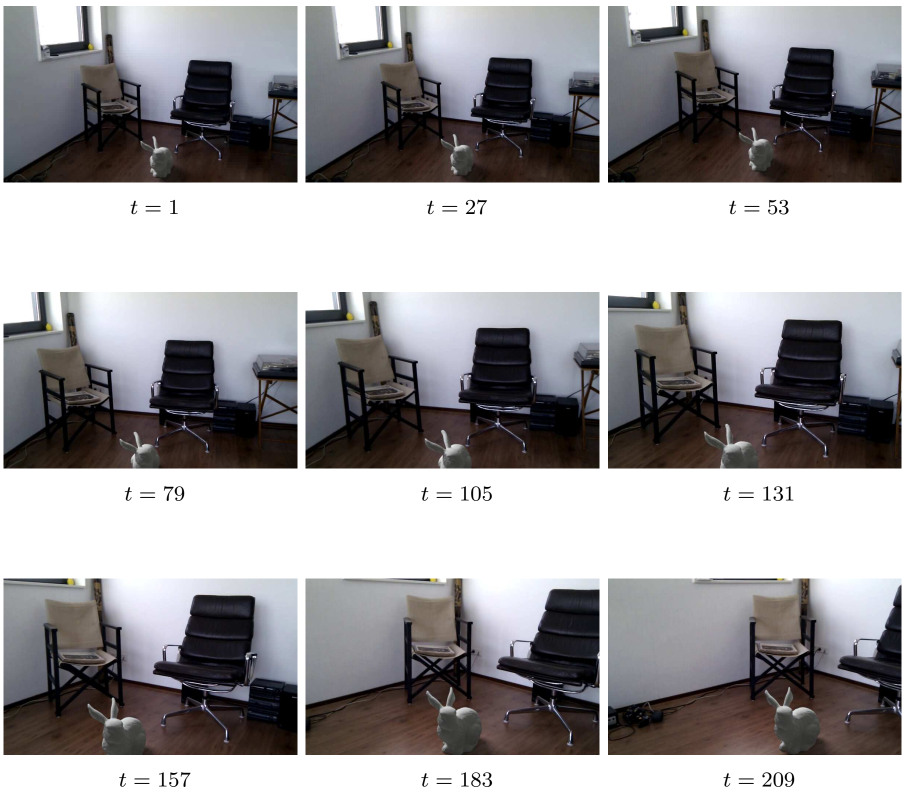


Figure A.3: Bunny statue inserted into room.

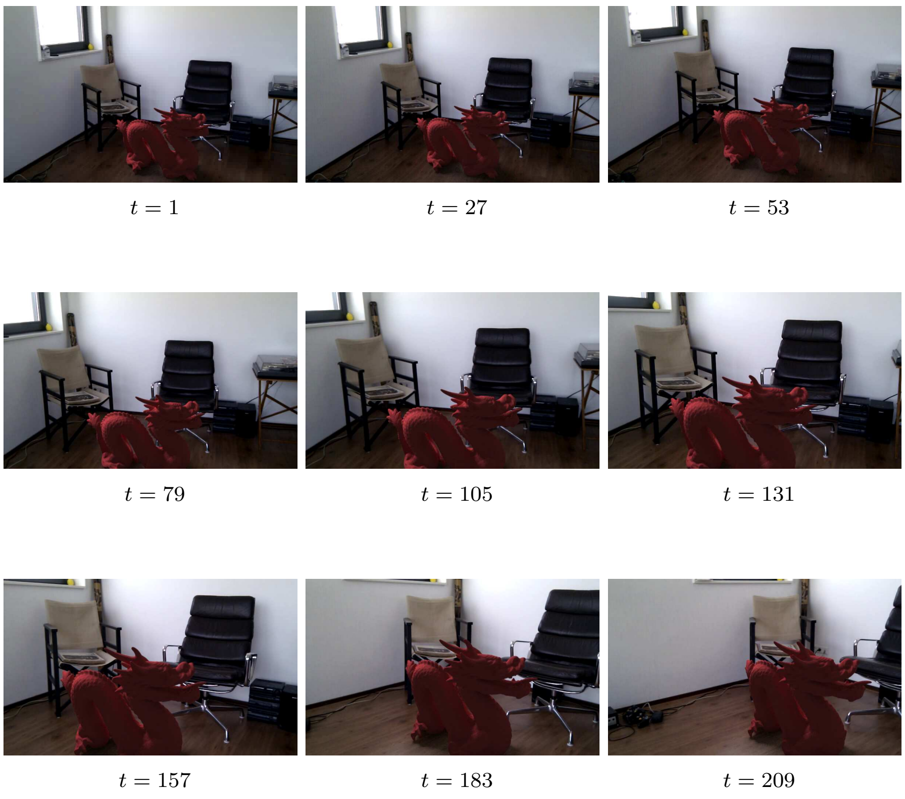


Figure A.4: Dragon inserted into room. The used footage and camera parameters were equal to those of the previous result.

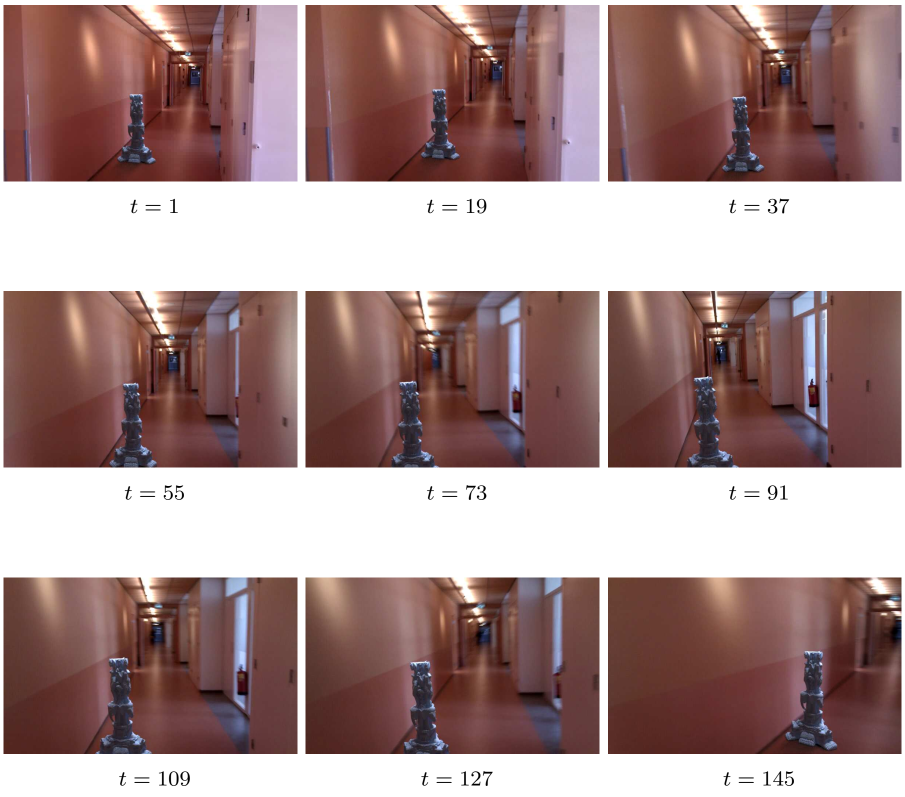


Figure A.5: Thai statue inserted into corridor.

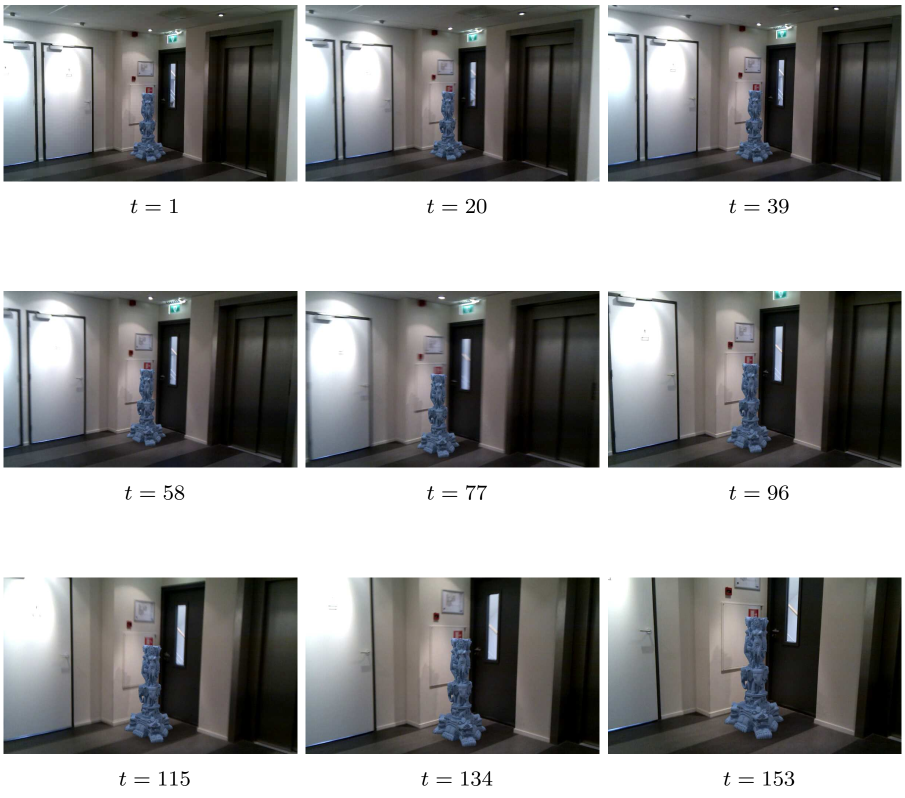


Figure A.6: Thai statue inserted into hall.