Examensarbete
LITH-ITN-MT-EX--05/049--SE

Rendering Realistic Augmented Objects Using a Image Based Lighting Approach

Johan Karlsson
Mikael Selegård

2005-06-10

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköpings Universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LITH-ITN-MT-EX--05/049--SE

Rendering Realistic Augmented Objects Using a Image Based Lighting Approach

Examensarbete utfört i medieteknik
vid Linköpings Tekniska Högskola, Campus Norrköping

Johan Karlsson
Mikael Selegård

Handledare Mark Ollila
Examinator Mark Ollila

Norrköping 2005-06-10

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2005-06-10

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _____

Rapporttyp
Report category

Examensarbete
☐ B-uppsats
☐ C-uppsats
☒ D-uppsats

☐ _____

ISBN

ISRN **LITH-ITN-MT-EX--05/049--SE**

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version
http://www.ep.liu.se/exjobb/itn/2005/mt/049/

Titel
Title

Rendering Realistic Augmented Objects Using a Image Based Lighting Approach

Författare
Author

Johan Karlsson, Mikael Selegård

Sammanfattning
Abstract

Augmented Reality (AR), the combination of real and virtual worlds, is a growing area in computer graphics. Until now, most of the focus has been on placing synthetic objects in the right position with regards to the real world, and to explor the possibilities of human interaction within the two worlds. This thesis presents the fact that virtual objects must not only be placed correctly but also lit truthfully in order to achieve a good degree of immersion. Conventional rendering techniques such as ray-tracing and radiosity requires intensive calculations and preparations for satisfying results. Hence, they are less usable for AR that demands calculations to perform in real time. We present a framework for rendering the synthetic objects using captured lighting conditions in real time. We use improved standard techniques for shadowing and lighting that we adapt for the use in a dynamic AR system as well as recent techniques of image based lighting.

Nyckelord
Keyword

IBL,AR,HDR,Augmented,Shadows

Copyright

Abstract

Augmented Reality (AR), the combination of real and virtual worlds, is a growing area in computer graphics. Until now, most of the focus has been on placing synthetic objects in the right position with regards to the real world, and to explor the possibilities of human interaction within the two worlds. This thesis presents the fact that virtual objects must not only be placed correctly but also lit truthfully in order to achieve a good degree of immersion. Conventional rendering techniques such as ray-tracing and radiosity requires intensive calculations and preparations for satisfying results. Hence, they are less usable for AR that demands calculations to perform in real time. We present a framework for rendering the synthetic objects using captured lighting conditions in real time. We use improved standard techniques for shadowing and lighting that we adapt for the use in a dynamic AR system as well as recent techniques of image based lighting.

Table of Figures

Table of Tables

Table of Contents

Acknowledgement

First of all, we truly want to thank Prof. Mark Billinghurst for giving us the great opportunity to do our Master Thesis at the Human Interface Technology Laboratory New Zeeland. The time at the lab was highly inspiring and enjoyable both socially and educationally.

We also want to express our gratitude to the following people:

Our advisor Prof. Richard Lobb for giving us invaluable advices and broad knowledge in the field of computer graphics, and also for his enthusiasm and the inspiration he gave us.

Dr. Mukundan for his support with mathematical and OpenGL related matters.

Dr. Raphael Grasset for all his initial help with the ARToolkit and the matrix transformation mayhem we encountered.

Dr. Michael Haller for his great help and contribution to our framework and his CG expertise.

Dr. Mark Ollila for being our supervisor and making it possible for us to continue the development of our project in Sweden and setting up the collaboration with HIT Lab NZ.

Thanks also to Föreningssparbanken Alfastiftelsen whose scholarship made this trip possible for us.

Thanks to all our friends at HITLab and especially the guys in our room: Michael Siggelkow, Felix löw and Herschi, but also to Phil Lamb, Marcel, Matt Keir and Nathan Gardiner.

Finally a great thanks to our social mentor Anna Lee Mason for all her help and support.

1. Introduction

1.1. Motivation

After some time of exploring current possibilities with Augmented Reality (AR) we realized that in all the different demonstrations and applications available, one lacked a true immersive feeling. We determined that the most crucial features for this lack were:

- Bad tracking data
    - Missing information (object disappear)
    - Unstable tracking results (object shiver)
- Poor blending
    - Not the same light setup
    - No shadows between worlds
    - Objects looked as if they were floating around on top of the world
- Lack of occlusion
- Visual quality
    - Poor video quality (When using Video AR)
    - 3D and video resolution disparity

The bad tracking data was a well documented and explored area of research. The inaccuracy could be reduced by introducing more exact but thereby vastly more expensive tracking equipment. The lack of occlusion was a problem that would require more information from the worlds than solely the relation between them. We would have needed to know for instance the geometry of the real world to be able to calculate possible occlusions. The problem with poor blending would also require more information than just the relation. In order to obtain a correct blending we needed to know more about the current illumination in our world. This was the motivation for studying an Image Based Lighting approach within AR. With a higher immersive feeling a user can obtain a more natural interaction with the augmented scene.

1.2. Problem

Is it possible to construct a framework for rendering synthetic objects using the actual lighting conditions in a room in real time? Is it possible to generate correct looking shadows from this information to help improve the depth cues in an AR application? How can a system like this be done with standard computer hardware and without to much system knowledge from a possible end user?

1.3. Purpose

There are certain areas in AR where realism is essential and where our method could vastly improve the graphical realism and thereby the immersive feeling. It could also simplify user interaction through a better understanding of the world.

Improved immersion and interaction:

- Improved graphical blending due to the use of a global lighting model.
- Improved depth cues due to the addition of shadows – easier to get an understanding of where the objects are placed in correspondence to the real world.

There are several areas in AR where improved realism could be beneficial:

- Systems for placing realistic looking objects into any real environment
- System for placing architecture
- Art
- City planning
- Museums
- Entertainment

But there are also other areas where the improved depth cues might lead to smoother user interaction with the objects.

1.4. HIT Lab NZ

Human Interface Technology Laboratory New Zealand

The HIT Lab NZ is a leading-edge human-computer interface research centre hosted at the University of Canterbury , Christchurch New Zealand. It is a joint venture between the University of Washington, the University of Canterbury and the Canterbury Development Corporation. The mission is to empower people through the invention, development, transition and commercialization of technologies that unlock the power of human intelligence. Their motto is consequently "**Unlocking the power of Human Intelligence**". Its goals of developing interfaces for human interaction with computers are shared with the world-leading HIT Lab US based at the University of Washington, Seattle. Together they strain to create new breakthrough technologies, in a broad area, to:

- Enhance human capabilities
- Vanquish human limitations
- Increase the flexibility and utility of industry's existing and imaginary products

Some of the technologies currently being developed at the Lab include 3D panoramic displays, virtual and augmented reality, voice and behavior recognition and intuitive aural and tactile feedback. The lab also has a research collaboration with Norrköping Visualization and Interaction Studio (NVIS) in the field of Mobile AR. These new technological innovations will intentionally increase human capabilities by accelerating people's ability to learn, create and communicate. Technologies developed in the Lab are thought to be utilized in areas such as education, medicine, scientific visualization, telecommunications and entertainment. Staff and students can work on their own initiatives or on industry and faculty-driven projects, all of which have the potential to result in real commercial products.

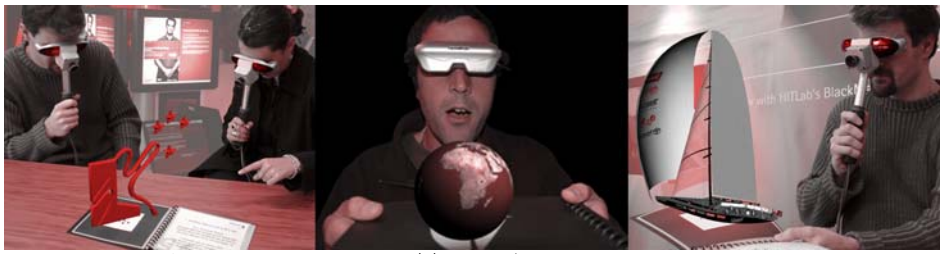


Figure 1 – Some projects from HIT Lab NZ

2. Background

2.1. Augmented Reality

Augmented reality is a technique for adding virtual objects (computer generated) to the user's view of reality. It supplements reality instead of completely replacing it as in virtual reality (VR). According to Azuma et al. [Azuma97] [Azuma01], an augmented reality system has the following characteristics:

- Combines real and virtual objects in a real environment
- Runs interactively in real time
- Registers (aligns) real and virtual objects with each other

The definition is not restricted to any particular techniques or methods and nor to which senses it applies. AR can potentially be applied to all the different senses like hearing, touch and smell and is not at all restricted to a visual experience. They also define the reality-virtuality continuum (Figure 2) where AR is one of the parts of mixed reality. Augmented reality is defined between the real environment and the augmented virtuality, in which real objects are added to a virtual world. At the right end of the band we have the virtual environment (VR) where worlds and objects are both virtual.

Figure 2 - The reality-virtuality continuum

AR is a relatively new field of research and there are still many and complex problems to solve before the real breakthrough, but recently a lot of progress have occurred.
The field will probably soon become established since its potential looks tremendously promising in a variety of different areas. The development mainly concentrates on enhancing perception and interaction with the real world and improving productivity in real world tasks. Movies like Star Trek and Star Wars early mentioned the idea of AR by introducing the hologram. In the movies the holograms were used mainly for presenting complex data in 3D and to enhance cooperation in mission planning. This former science fiction is actually exactly how AR today is being used.

Some areas of the field today are:

- Medical applications
- Military applications
- Educational applications
- Gaming applications
- Path planning
- Cooperation applications

2.1.1. Displays

There are several possibilities for merging real and virtual environments and we will briefly mention the most widely used techniques.

We can classify the different methods into the following three areas:

- Head mounted displays (HMD)
- Handheld displays
- Projective displays

Head mounted displays (HMD)

All the HMD:s are attached to the user's head providing information for the eyes. There are two methods today that are mostly used, optical see-through and video see-through.
The video based method uses opaque glasses where a captured video stream is displayed in the background of the virtual content. This also requires a video camera attached to the device for the demanded video stream. The optical method provides the AR overlay through a transparent display.

Figure 3 – Typical HMD Display using video see-through

There are still a lot of problems with the HMD and even if the development is in progress the major problems seems to remain. Ideally the displays would be no larger than an ordinary pair of eyeglasses, without annoying cables and heavy parts. Many of the displays today are far too heavy and awkward for serious use under longer periods of time. Other problems are:

- Insufficient Brightness
- Insufficient Resolution
- Insufficient Field of view
- Insufficient Contrast

These factors all contribute to the fact that it is still hard to make the blending between the worlds look perfectly natural. In the video see-through method we also have the parallax error, since the placements of the cameras are not perfectly aligned with the eyes. This makes the view slightly different from the view that our eyes would have registered and this small inaccuracy can make it hard to adapt to the display. There has also been some research on virtual retinal display, where a low power laser draws the AR graphics directly on the retina. This might give the field new possibilities when we can obtain a higher brightness, contrast and depth of field. The question is how accepted it will become to let laser beams sweep the eyes repeatedly in a daily use.

Handheld displays

The handheld is a fast growing market for AR systems. Since PDA and modern mobile phones all seems to be equipped with better cameras and advanced graphics processors the possibilities are on the rise. With an attached camera it is easy to provide video see-through based augmentations on the LCD screen. This is a very natural step for AR into the big commercial area. Of course the immersive feeling in a handheld is not the same as with a HMD since you only see the augmented world on the small screen. Many applications don't really require full immersion and can work really well on the handheld, like:

- Path finding
- Gaming applications
- Educational applications

Figure 4 – A mobile AR system

Projection displays

In this area the virtual content are directly projected on the real world. This is often done using a fixed projector and there is no requirement for special eyewear. This can be extended to use several projectors for augmentation in all directions. Systems like this have existed for a while in the VR area, known as Cave Automatic Virtual Environment (CAVE). In this approach the user stands in a room where all the walls are being projected by several aligned projectors, and the same approach are being used for augmented reality rooms. There are also approaches where the user wears a head worn projector which can be carried around freely in the world. This approach is more dynamic than the fixed projector ditto, but often results in heavy equipment and the use of HMD would then often be preferred. The main areas for projection displays are when several people use the same system at once and where cooperation might be simplified by a shared view.

2.1.2. Registration

To be able to know where to draw our virtual objects in respect to the world we need to know the relation between the worlds. This tracking is a crucial part for AR and is known as registration. What we need from our registration are position and orientation of some known part of the real world. There exist several tracking systems with different strengths and weaknesses and they are all suitable for different environments and tasks. Different tracking devices have different degree-of-freedom (DOF). For a position in a 3D room we need three DOF (x,y,z) and for the orientation we require additional three DOF. To be able to track any free moving accurately in our world this means we need a 6-DOF tracker.

Some examples of tracking devices for AR are:

- Video based tracking
- Magnetic tracking
- Ultra sound tracking
- Laser tracking
- Gyros
- Global Positioning System (GPS)

The video based tracking can be done using any digital camera attached to a computer while the other methods require often expensive special equipment. In some situations some systems aren't good enough and have to be used in combination with other tracking systems. The combination of different systems is called hybrid tracking and is becoming more frequently used. Combining a GPS with an inertial setup is a common way to get a wide range and good accuracy at the same time. It is also necessary for some methods to be combined to get enough world information, since all systems aren't 6-DOF.

Figure 5 – A hybrid system for outdoor AR using GPS and Inertial systems.

2.2. ARToolKit

ARToolKit is a software library for developing AR applications. The toolkit uses the computer vision idea for tracking the relation between the real and virtual worlds. This chapter will describe how this tracking is achieved and explain some of the essential mathematical ideas and concepts.

The toolkit was initially developed by M. Billinhurst (Director of HIT Lab NZ) and Hirokazu Kato [Billinhurst 99]. ARToolKit is free for use in non-commercial applications and is distributed as open-source under the GPL license. The current version supports both video and optical see-through augmented reality and works more or less on most existing platforms.

The requirements are:

- Computer
- Webcam (USB,USB2,Fire Wire)
- DirectShow compatible graphics card

Figure 6 – Typical use of ARToolkit

2.2.1. How it works

As we mentioned one of the most crucial part of AR is the registration. ARToolKit uses a computer vision based system that looks for pre specified markers in the world. These markers must therefore be placed in the world in order to obtain any usable information from the system. Before analyzing the details of the system we present an overview of a complete AR solution running on AR toolkit. Figure 7 describes the logic of the system schematically.

Main steps in ARToolKit:

1. Prepare system for AR
2. Acquire video image
3. Find markers in scene
4. Calculate position and rotations (store in a transformation matrix)
5. Identify markers (if several markers are being used)
6. Position and orient objects (with applied transformations)
7. Render objects in overlay to real image (aligned to our real world)
    - Redo point 2-7 for every frame to obtain a correct and real time augmented reality solution.

ARToolkit's responsibility is to analyze our image and produce the correct transformation matrix between camera and marker frame. This matrix can be used in any application and graphic API, and the rendering part is actually not a task for the toolkit itself.

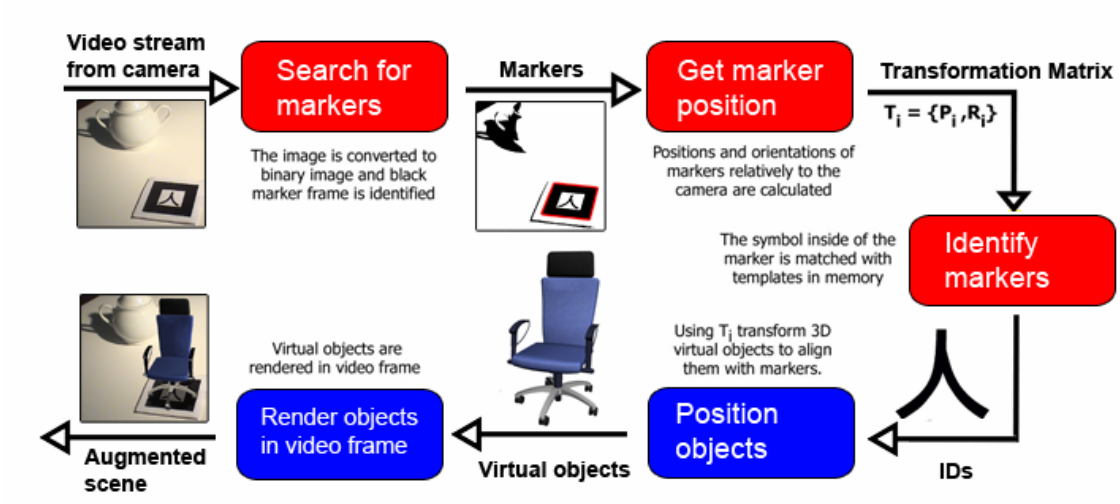


Figure 7 –The pipeline of ARToolkit

2.2.2. Initialization

In order to start an AR session a few things has to be set up. First of all we need to calibrate our camera and adapt the camera's white balance to the current lighting conditions. Since we use our video stream as the source for tracking, the picture has to be well adjusted in order to find the markers in a convenient way. Incorrect settings might lead to problems in distinguishing the marker from the rest of the scene, which will result in failure. Malfunction will force us to stop rendering the synthetic objects, since the current alignment between the worlds are unknown. Some cameras are equipped with automatic settings for the white balance but that will result in a video frame that alters light settings continuously during operation. These fluctuations can result in a lower immersive feeling and should be avoided. If the lighting in the room is rather intense it can also be a problem with reflections and glare spots on the marker. This complicates the tracking process and should be avoided as well. A simple solution is to construct the markers of non-reflective materials, like velvet fabric.

It is also important to have the marker in range of the camera. The larger the marker the further away the pattern can be detected and consequently the size of the tracked world. Table 1 displays some typical ranges for square markers of different sizes. These results were gathered by making marker patterns of different sizes. Placing them perpendicular to the camera and translating the camera back until tracking failed, gave a rough idea about realistic viewing distances.

Pattern Size (cm)	Usable Range (cm)
7	*40.6*
8.9	*63.5*
10.8	*86.5*
18.7	*127*

Table 1 – Usable ranges for different Pattern sizes

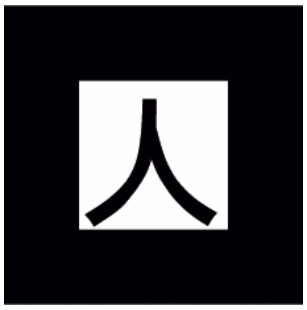
The marker can be tracked even further away, but the values can be seen as a recommendation if good tracking results are a requirement. This range is also affected by pattern complexity. Simpler patterns give better tracking results than advanced dittos. Patterns with large black and white regions (low frequency patterns) are the most effective. This test was done using a web cam with a resolution of 640*480 pixels. The higher resolution of the input picture the better tracking results can be achieved. Since we are using image processing on the video and we have to do it in real time higher resolution isn't the optimal solution. The image based calculations are already consuming a huge amount of computational power in every frame. A resolution of 1024*768 would drop the frame rate noticeably. It is also a

matter of compression. Older cameras transmitting via standard USB must compress the stream in order to obtain fast transmission. Using USB2 or FireWire doesn't limit us in bandwidth and less compression needs to be done. The compression itself doesn't influence the cost of the image based calculations, only the quality of the result.

The virtual objects will only appear when the marker is in view. This may limit the size or movement of the virtual object in respect of the marker. What it also means is that the marker can't in any way be occluded by any real objects. Then the tracking will fail. Figure 8 demonstrates an example where the marker is occluded by a real object.

Figure 8 – Marker occluded by pen

Before an augmented session we have to let our system know how our current markers look like. A typical marker is made of a black frame on a white piece of paper. In the middle of this frame we have a white area in which a black shape is placed. The shape is for making the different markers unique from each other and for the system to know how the marker is oriented. The frame in black is shaped as a square and the reason for this is that it is a convenient shape to trace since it is made of four corners. To be able to decide a position in 3D space we need at least three points. To avoid errors the toolkit tracks four points (each corner) and selects the three with least error probability for geometrical calculation. See figure 9 for a typical AR marker layout.



Different markers are stored in the system during a pre process set up and are stored as binary files. When this is done the system is ready to be fed with streaming video.

Figure 9 – Typical marker layout

2.2.3. Image processing

When the system is fed with the streaming video information from the camera, every image frame is treated individually. For every frame the following sequence of operations must be completed for a complete registration. Most parts of this chapter is received from [Vial03].

1. Grab image I
2. Construct binary version of I
3. Extract connected components
4. Extract contours of connected components
5. Reject all contours that doesn't fulfill the rules of a square
6. Sub-pixel recovery of the corners' coordinates in I
7. Calculate transformation matrix from the given coordinates

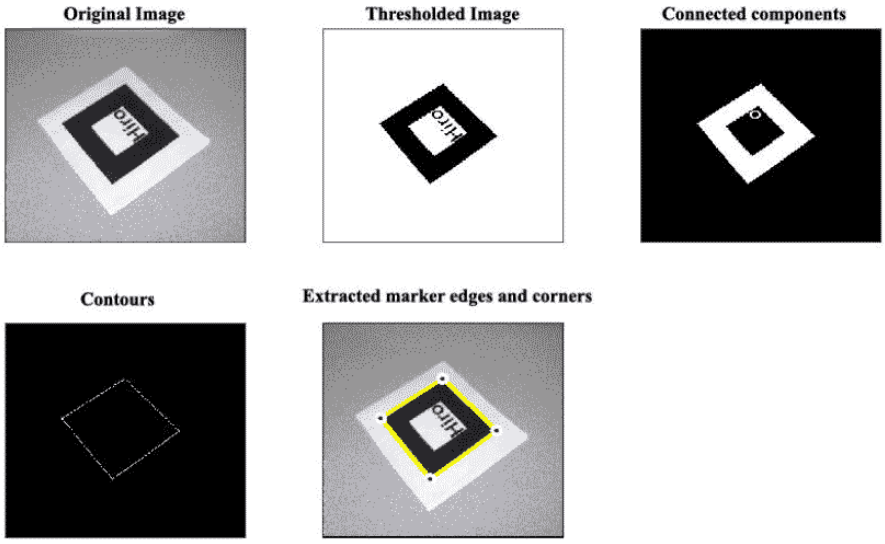


Figure 10 – Basic steps of ARToolkit's video based tracking method

Construction of the binary image

To start with the image is turned into a binary version (black or white), which will separate the dark parts of the image from the light ones. The global threshold value T, corresponds to a place in the histogram (pixel value) where the light pixels will be separated from the dark. A pixel in the original image whose grey value is less than T will be represented by 0 and a gray value greater than T will be represented by 255.
This T value can be changed depending on different lighting aspects. Using a global value on an entire image makes the process fast but it suffers from low robustness. Under uneven illumination the method might lack in result, but a more adaptive process would be more consuming.

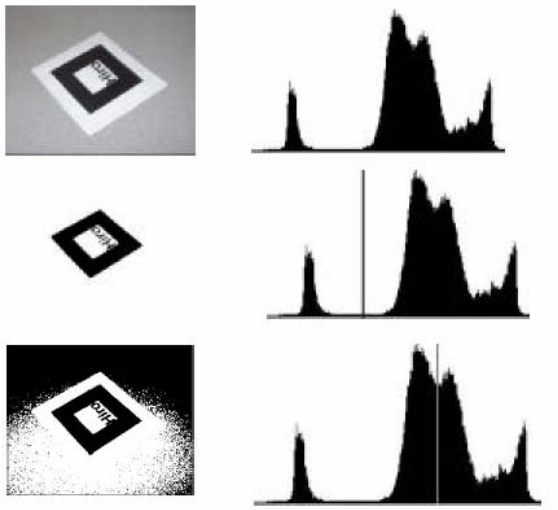


Figure 11 – Construction of binary image with different threshold values

Extract connected components

This step decides whether a region is a connected component or not. The system scans the entire image and where a black value is found a label L is assigned. This label will have a new value if none of the current pixel's neighbors (connected components) already have an assigned value. If any of the neighbors already have a label the current pixel receives the same label.

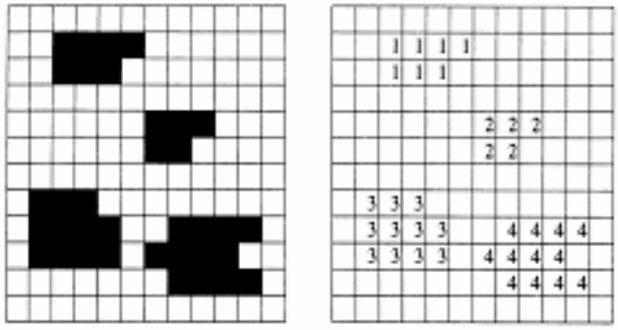


Figure 12 – Extracted connected components and assigned labels

Extract contours of connected components

Once the connected components are identified we can easily identify their contours. This is represented as a chain of pixels. To identify one contour pixel we scan for pixels whose neighborhood includes at least one exterior pixel (value 0). This chain of pixels can then efficiently be stored as a coded direction chain code. With an identified starting point we simply code the direction of the following pixel in the chain. It gives us eight choices of direction and an efficient representation.

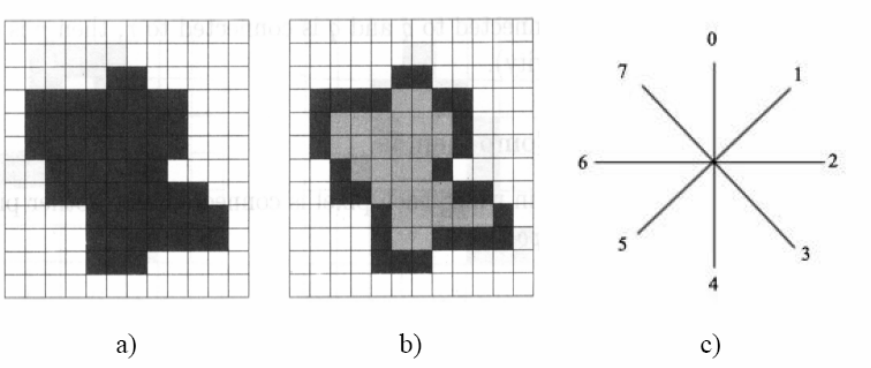


Figure 13 – a) Connected region b) contours c) possible directions in chain code

Reject all contours that doesn't fulfill the rules of a square

The latter step often gives us a lot of regions of interest but we want to exclude areas that won't likely be projections of our markers. Different selective methods are used to discriminate between good and bad areas. A good region should not be too small (noise exclusion), not too big and have exactly four corners. If these requirements can't be fulfilled the region is rejected from the list of possible detections.

To find a corner the following steps are processed recursively.

- Fit line through two points on contour
- Find point with maximum distance from this line
- If the distance is greater than threshold we have a corner

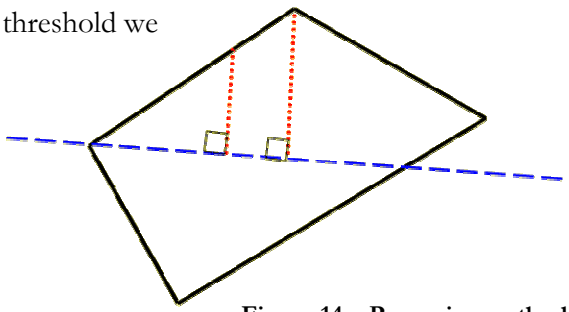


Figure 14 – Recursive method for finding corners

Sub-pixel recovery of the corners' coordinates in I

When the markers are found we want to make sure that the corners' coordinates are as accurate as possible. The coordinates are extremely important since they are the source of defining the correspondence between the virtual and real camera. We have to calculate the sub-pixel coordinates, which is defined as the intersection of the contour lines. We calculate how to most accurately represent the lines by looking at all the pixels in the contour. The least error representation is calculated for all contours in the square and the corner points of these lines are established. When this process is finished we have four coordinates for every marker in the image plane, which describes the position of the corners. Since we know the size of the marker in the real world we also know the 3D position of these markers relative to the marker centre. With this information we can now start to calculate the mathematical relation between the two worlds.

Calculate transformation matrix

For the current frame we now have four correspondences of coplanar points in the object and image frame. The transformation calculated to map these four points is called a homography. It represents the projection of the marker relative to the camera. Let H represent our homography as a 3x3 matrix, then the coordinates in the image frame (u, v) and Object frame (X^w, Y^w) can be presented as:

$$\begin{pmatrix} u_i \\ v_i \\ 1 \end{pmatrix} = \mathrm{H} \begin{pmatrix} X_i^w \\ Y_i^w \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} X_i^w \\ Y_i^w \\ 1 \end{pmatrix}$$

So each of our point correspondence between the frames generates three equations for the elements of H.

$u_i = h_{11}X_i^w + h_{12}Y_i^w + h_{13}$
$v_i = h_{21}X_i^w + h_{22}Y_i^w + h_{23}$
$1 = h_{31}X_i^w + h_{32}Y_i^w + h_{33}$

With four know correspondences the parameters in our homography can be calculated numerically. The result gives us a set of parameters that is called a set of extrinsic parameters. The result should be seen as a rigid transformation. It is convenient to represent this result as a 3D rotation matrix and a 3D translation vector. The rotation matrix for mapping the axis to each other and the translation vector to align the two origos. See figure 15 for a graphical understanding of the two systems. Figure 16 also describes how the image frame is related between these.

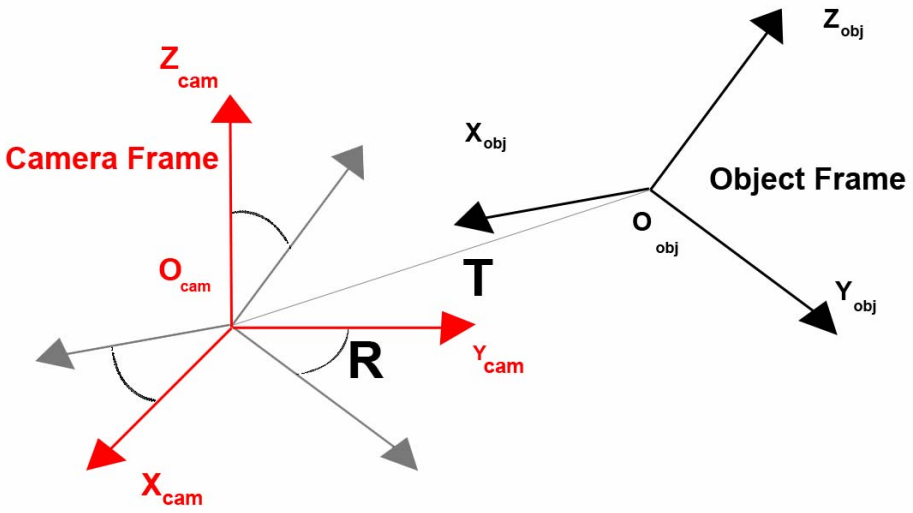


Figure 15 – Relation between Camera Frame and Object Frame can be found by calculation of Translation and Rotation

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \qquad T = [t_x\ t_y\ t_z]^T$$

We can now easily obtain the relation between the coordinates of any point in the Object and Camera frames:

$$P^c_i = R\ P^o_i + T \qquad \text{(I)}$$

This can be represented as the following 3x4 matrix:

$$M_{ext} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} \qquad \text{(II)}$$

Any given coordinate in our object frame can now be presented in camera frame by:

$$P^c_i = M_{ezt}\ P^o_i \qquad \text{(III)}$$

This matrix is recalculated in every frame and this is the real output from AR toolkit. To be able to draw objects on our marker we can use this matrix as the modelViewMatrix in OpenGL.

2.2.4. Intrinsic paramaters

Our graphic API also needs some information about the camera settings, to be able to draw the synthetic objects aligned with the world. We need to know how the camera projects the 3D world into 2D. A camera (real and virtual) can project an image of the world differently depending on focal length and lens distortion. We need to know how the camera that we are using as input for ARToolkit is calibrated. This calibration step can be done in advance and there is no need for updating this information during run time. Simple utilities exist for calibrating the camera in ARToolkit and when this is done we can construct our camera projection matrix needed to render our graphics. See fig 16 for a complete understanding of the relation between the camera frame, marker frame and the projected image.

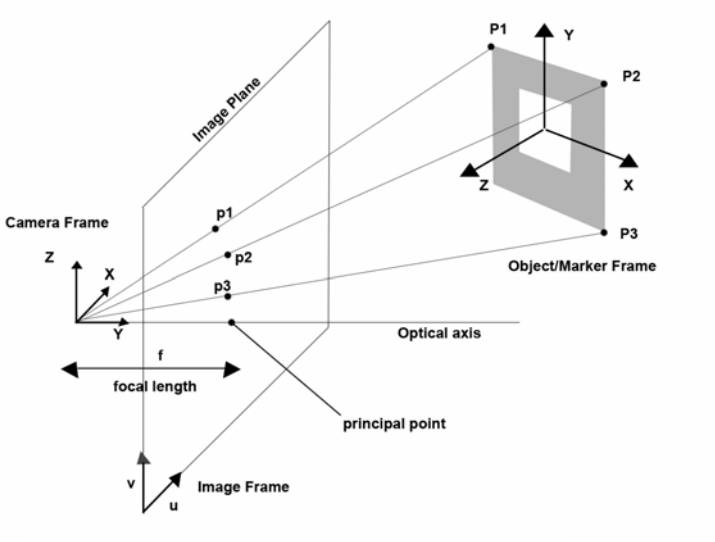


Figure 16 – Final relation depends on focal length, principal point and distortion of the camera lens

Since this is not a central part of our project we will only show how the result of this calibration step will look like.

Resulting matrix $M_{inr} = \begin{pmatrix} fs_x & 0 & u_0 & 0 \\ 0 & fs_y & v0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

Here (f) is depending on our focal length, (u,v) are depending on lens distortion and (s) on the current principal point.

Final Result from ARToolkit

Camera Projection Matrix = $M_{inr} * M_{ezt}$

$$= \begin{pmatrix} fs_x & 0 & u_0 & 0 \\ 0 & fs_y & v0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix}$$

M_{inr} = camera calibration parameters
M_{ezt} = camera transformation matrix

In OpenGL we now use the M_{inr} as input for the GL_PROJECTION and M_{ezt} as input for our GL_MODELVIEW. After that we can draw any 3D object augmented with our scene correctly in every frame.

2.3. Image-based Lighting

2.3.1. Introduction

Image-based lighting (IBL) is the process of illuminating scenes or objects (real or synthetic) with images from the real world containing light information. IBL is closely related to image-based modeling, where 3D geometric structures from an image are derived and also to image-based rendering where the rendered appearance of a scene is produced from its appearance in images. The result of a successful use of IBL is a realistic integration between real and synthetic objects in a real world environment. Paul Debevec, USC Institute for Creative Technologies, has done a tremendous work in this big area of research and has contributed with both advanced theories and easy to use applications for users' first experience to the field. [Debevec98] [Debevec97] [Debevec96]

The basic steps in IBL usage are:

- Capture real-world illumination using an omni directional, high dynamic range, capture system.
- Map the illumination onto an environmental representation.
- Place synthetic objects in world space.
- Simulate the light coming from the environment on synthetic objects.

Figure 17 – A microscope, modeled by Gary Butcher in 3D Studio Max rendered using Marcos Fajardo's Arnold system. Scene illuminated by light captured in a kitchen.

2.3.2. Capturing light information

The first step of IBL is acquiring a measurement of real-world illumination. The result of this is stored as a light probe image. It is a photographically obtained image of the world with two exclusive properties. First, they are omni directional; meaning that for every direction in the world there is a pixel in the image representing that direction. Also, their pixel values are linearly proportional to the amount of light in the real world. There are several methods of obtaining omni directional images. The simplest way is to use a standard camera and take photographs of a mirrored ball placed in the world. The unique property of a mirrored ball is that it actually reflects the entire environment surrounding it, not just the hemisphere in front of it. The outer regions of the sphere are actually reflecting the back half of the world. See figure 18 for a higher understanding of a mirrored sphere's reflective nature. There is only one small spot right behind the sphere that we can't see in its reflection.

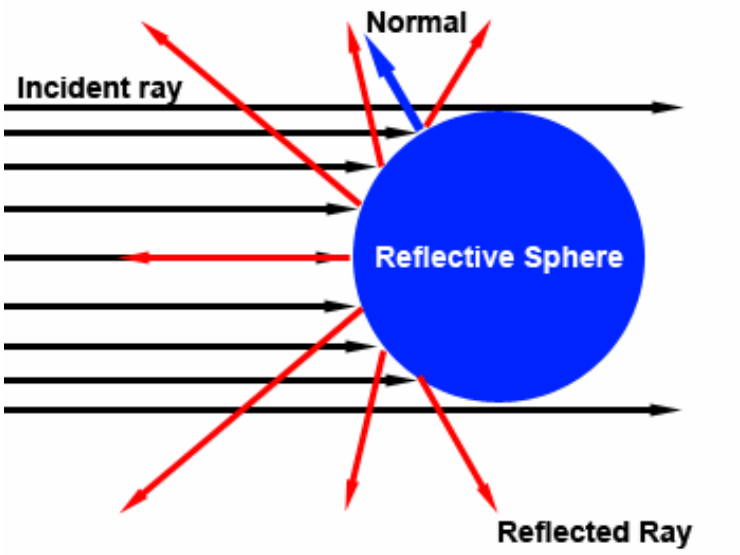


Figure 18 – Figure describing reflection angles from sphere

The mirrored ball is a great starting point in many ways. It is cheap since it only requires a standard camera and a reflective sphere. A reflective sphere can probably be found in the family's Christmas decoration collection. It is also very convenient as the image already is a correct sphere map. This map can easily be implemented in OpenGL as a sphere reflection map for testing purposes. One drawback of the sphere is the resolution of the background information. As seen in figure 19, the background information is stored in an area much more compressed than the front hemisphere. To obtain better resolution from the back hemisphere as well we would need to take additional photos from other angels. Another drawback is that the

image of the mirrored ball will also include the camera and rig in its reflection. By acquiring several photos we can eliminate the camera in the picture by combining the different pictures. A different method to acquire the data is to take several photos in diverse directions and then stitch them together. Using a fisheye lens is a good way to cover a particularly large area in a single shot, using two of these images is enough for a complete environmental light probe. Rotating cameras also exist that can scan across a 360° field and produce light probes in one sweep, but are often expensive and requires intense data flow possibilities.

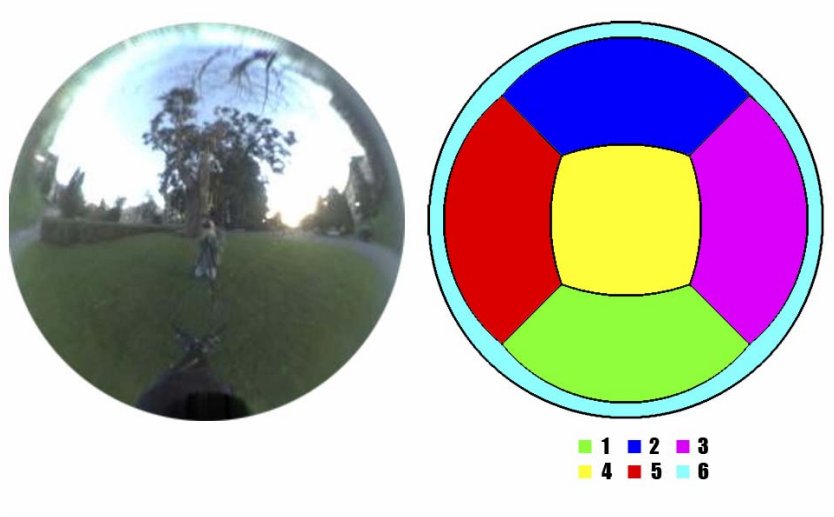


Figure 19 – Different areas of the sphere reflection
1-bottom - 2-top - 3-right - 4-front - 5-left - 6-back

2.3.3. High Definition Range Imaging

Accurately recording the light in a scene is hard because of the high dynamic range that natural scenes typically exhibit. The intensity of a light source might be from two to six orders of magnitude larger than the intensity of the non-emissive parts of the world. For a perfect light solution it is important to register both the concentrated areas of light sources as well as the large areas of indirect light from the environment. A standard digital image only represents a small fraction of this dynamic range – the ratio between the dimmest and the brightest regions represented. When a part of our scene is too bright, the current pixel will be saturated to the maximum value (usually 255) and equally for all the dark parts that will be represented by zero. This means that no matter how bright or dark our parts of the image are they can't be represented with more than our limits. (0->255). This means that our pixel values aren't really proportional to the light levels in the scene. To obtain truly proportional images P. Debevec [Debevec97] proposes a technique to represent the full dynamic range as a radiance map (High Definition Range Image). These maps are derived from a series of images with varying exposure levels (figure 20). A linear response composite image is then formed that covers the entire range of illumination values in the scene.

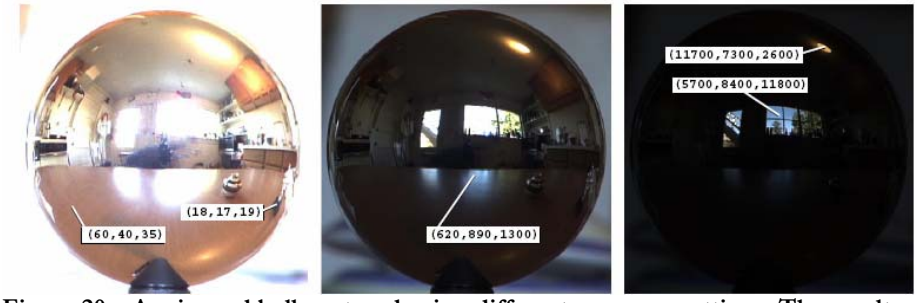


Figure 20 – A mirrored ball captured using different exposure settings. The result tells us about direction, color and intensity of all forms of incident light (Image from [Debevec98])

The HDR image is stored as a single-precision floating point number in RGB, allowing the full range of light to be presented.

2.3.4. Scene Reconstruction

In the field of IBL it is often required to have some knowledge of the 3D geometry of the real environment.
The major reasons for a 3D geometric model are:

- Resolving occlusions between real and virtual objects
- Collision detection between background scene and synthetic objects
- Constructing an correct illumination environment to shade our objects
- Render shadows on real world cast by virtual objects

The quality and need might differ depending on project conditions. There are several approaches of reconstructing the geometry with different advantages and disadvantages.

Automatic

- Dense stereo matching
  - Inexpensive
  - Requires multiple images and decent scene textures
  - Dependent on good image quality.

- Laser scanning
  - Expensive hardware

- Structured light projection
  - Complicated setup

Semi-automatic

- Primitive-based modelling [Debevec96] [Gibson04]
  - Effective for simply environments
  - Pre-defined shapes (fast)

In [Gibson04] Gibson concludes the advantages and disadvantages of the different types of methods as seen in table 2.
As seen all the automatic methods gives very accurate data, but with an immense cloud of unstructured geometry data. This leads to complications in calculations of light and shadows and the less accurate but better structured semi-automatic reconstruction method is more suitable for AR.

Table 2 – Automatic versus Semi-Automatic reconstruction methods

Automatic	Semi-Automatic
Accurate geometry	Less accuracy
Relatively fast and easy to use	More labour-intensive
Unstructured geometry	Good scene structure
Hole and occlusion problems	Can apply "user-knowledge"
Requires scene "texture"	Works without scene "texture"
Requires more than one image	Works with single image

2.3.5. Map the Illumination onto an environmental representation

With the illumination information the last step is to map the radiance information to an environmental representation. This representation can be stored differently depending on the area of usage and the accuracy of the scene. The complexity involved in modeling the physical behavior of light by explicitly tracing light rays in a scene representation has led to alternative methods. In a case where the known scene geometry is restricted, the use of an alternative representation, called environmental mapping [Blinn 76] is often used. In practice, environment mapping applies a special texture map that contains an image of the scene surrounding an object, to the object itself. The result approximates the appearance of a reflective surface, close enough to fool the eye, without incurring any of the complex computations involved in ray tracing. The value of the reflected pixel is simply calculated by finding the corresponding pixel in the map that the current reflecting vector points at (figure 21).

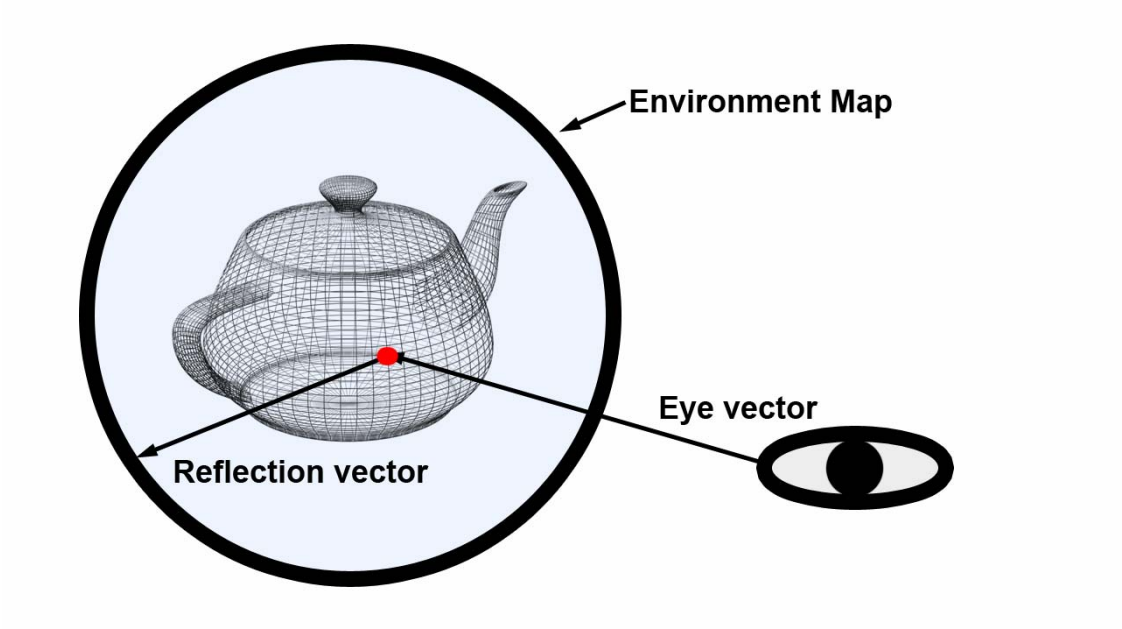


Figure 21 - object placed in an environmental representation

There are some different types of environment mapping in the graphics industry. In the beginning of the mapping era, spherical mapping was most common. A sphere map is a 2D representation of the full 360-degree view of the scene surrounding an object. Unfortunately, a spherical representation meant a lot of problems and restrictions for the developers. While sphere mapping could produce satisfactory reflections under exactly the right conditions, it was limited in a changing environment. It suffered from distortion problems, viewpoint dependency and singularities. Mapping normal rectangular images to the inside of a sphere led to complications for the artists. Singularities are mathematical discontinuities

that occur with sphere mapping because the point behind the object is represented by the entire outer ring of the spherical map. One point defined by several sources. (figure 22)

Figure 22 – The backside of a reflecting object is faulty due to the singularity problem (image from NVIDA)

The singularity problem is something that exists even during the capture of the light probe. The information from the back is singular and no real information actually exists. Due to early restrictions in implementation, hardware manufacturers and developers soon began to use cubical environment mapping instead. Here the shape of the map was changed to a six-sided cube where linear mapping are allowed in all directions to the six planar maps. Each face of the cubic environment map covers a 90-degree field of view in the horizontal and vertical vision.(figure 23)The resulting reflection therefore doesn't undergo the warping or damaging singularities associated with a sphere map.

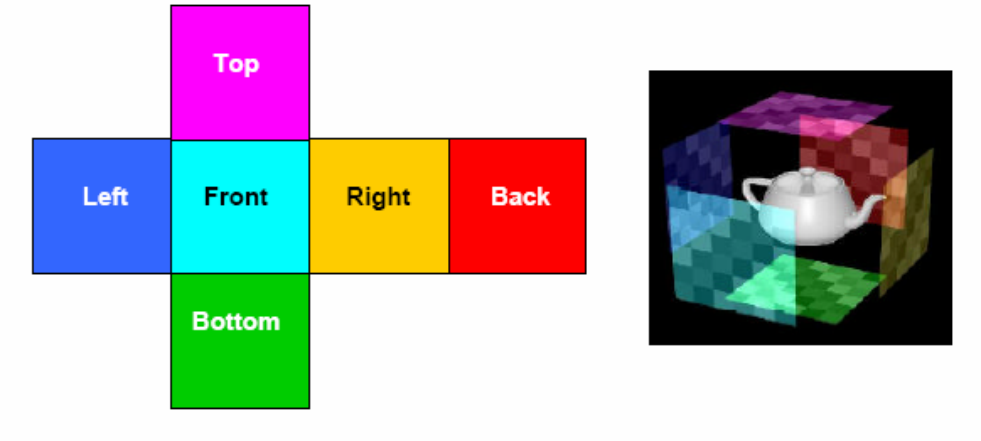


Figure 23 – A cube environment is created by stitching six projective textures

Where sphere maps usually need to be recalculated for any change in viewpoint the cube maps are totally independent. This means that there is no need for an update of the map if the surroundings are static and only the camera moves. These methods where both initially developed for the use in computer games where the maps where supposed to be updated frequently from the surrounding environments in the game. In this area the cube map was a more appealing method for developers and therefore the support of this method became top priority in modern graphics hardware.

2.4. Computer graphics

2.4.1. Global Illumination

If you have seen a computer rendering from the last couple of years that looked so real that you thought it was a photo there is a big possibility that it was rendered with Global Illumination (GI). This concept tries to overcome some of the problems associated with Direct Illumination; the method used in for example OpenGL. In a GI approach the light is modeled in a more physically correct way taking both direct and indirect lighting caused by diffuse reflections into account. Images rendered using global illumination algorithms are often considered to be more photorealistic than images rendered using local illumination algorithms. However, they are also much slower and more computationally expensive to create. Examples of GI solutions are radiosity and photon mapping. We will discuss radiosity in (2.7.2).

Figure 24 – Global illumination rendering (image from http://www.studiopc.com)

2.4.2. Real time lighting models

A lighting model describes the way a systems calculates the interaction between objects, materials and lights in a scene. The first method that dealt with non-diffuse surfaces was introduced by Phong [Phong73] in 1973. It was not based on physics but was instead derived from physical observations. Phong made experiments trying to isolate the most important properties that decide how a material reacts to light. For example Phong observed that for very shiny surfaces the specular highlight was small and the intensity fell off rapidly, while for rough surfaces it was larger and fell off more slowly. The visual results of Phongs studies combined were however very convincing and the model is widely used in 3D graphics today. It is also the model that we decided to focus on and extend in our project since it is the de facto standard in 3D graphics for real time. The Phong lighting model is for example used by standard OpenGL in order to calculate the color values for vertexes. Phong's lighting model is a local illumination model, which implies that only direct reflections are taken into account meaning that light that bounces off more than one surface before reaching the eye is not accounted for. This also means that the method doesn't handle shadows automatically; these instead have to be derived and rendered with other techniques. While this may not be perfectly realistic or convenient, it allows the lighting to be computed efficiently in real time. To properly handle indirect lighting, a global illumination method such as radiosity is required, which is much more computationally expensive.

In Phongs' model the color of a pixel is expressed as a linear combination of an ambient, a diffuse and a specular term.

Final color = Ambient term (A) + diffuse term (D) + specular term (S)

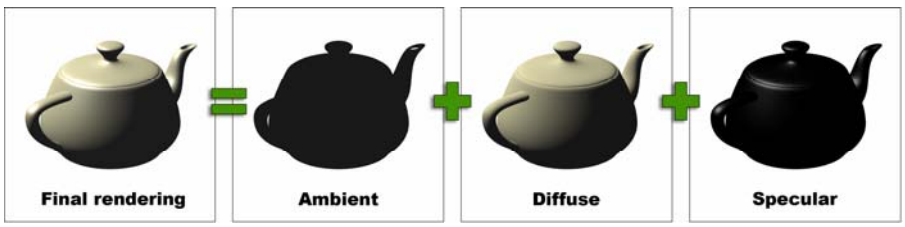


Figure 25 – Phong's lighting model

Ambient light

This is a constant amount of light that gets added to the scene and can be thought of as the background light. Its goal is to imitate the contribution of indirect reflections, which can normally only be accounted for using global illumination solutions. The ambient term is used mainly to keep shadows from turning completely black, which would look unrealistic. The ambient term is simply a combination of the ambient components of the light source (A_l) and the surface material (A_m).

$$A = A_l \times A_m \qquad \text{(I)}$$

Diffuse light

This is the part of the light that is independent of the view vector because it is reflected equally into all directions. Its intensity is proportional to the angle between the light direction and the normal at which the light hits the surface but it is independent of the viewer's position in the scene. The intensity is also proportional to the material's diffuse reflection coefficient as well as the lights diffuse intensity. This is known as Lambertian reflection and can be expressed as:

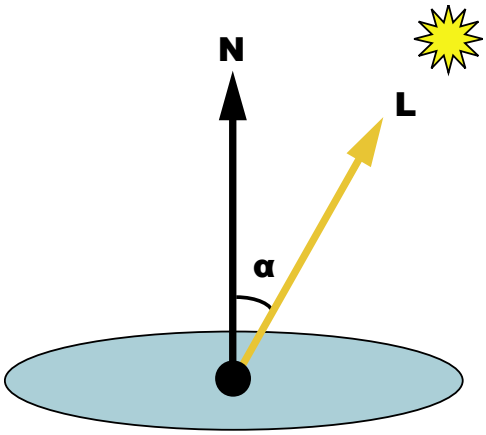


Figure 26 – Diffuse light model

$$D = D_l \times D_m \times \max(L \bullet N, 0) \qquad \text{(II)}$$

D_l – The light's diffuse intensity
D_m – The material's diffuse coefficient.
N – The normal of the surface
L – The normalized light vector from the point being shaded to the light source.

The dot product of L and N will return the cosine of the angle between the two vectors. If they are equal, the dot product is one. If they are perpendicular, the value is zero. If the two vectors point away from each

other, the value becomes negative and must be set to zero. This is why the max() function is needed in order to prevent ending up with negative light which has no meaning in the model.

Specular light

Specular light is light that reflects in a particular direction depending on the view angle. Because of this, specular light is dependent on the viewer's position. Its intensity is proportional to the cosine between the light reflection vector R and the view vector V. The specular light is what creates the highlights that make an object look shiny. The reflection vector R represents the direction the incoming light would be reflected in if the surface were a perfect mirror and is calculated as follows:

$$R = 2(N \bullet L)N - L \qquad \text{(III)}$$

N – The normal of the surface
L – The normalized light vector from the point being shaded to the light source.

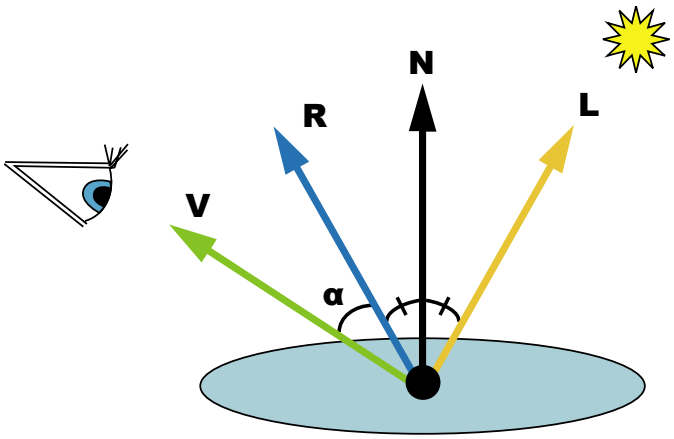


Figure 27 – Specular light model

When we have the reflection vector we can then calculate the specular term as follows:

$$S = S_l \times S_m \times \max((R \bullet V), 0)^{\,n} \qquad \text{(IV)}$$

S_l – The light's specular intensity
S_m – The material's specular coefficient.
R – The light reflection vector
V – The normalized view vector from the point being shaded to the camera
n – The specular exponent

The larger the angle between R and V, the lower the specular term will be and the less noticeable the specular highlighting effect. The exponent *n* on

the dot product term is called the specular exponent of the surface and represents the materials shininess. Higher values of n lead to smaller, sharper highlights, whereas lower values result in large and soft highlights. Regardless of the exponent used, the function is always zero when the angle between the two vectors is 90 degrees and one when the angle is zero.

Jim Blinn [Blinn77] came up with an alternative way to calculate the specular term, which is less computational expensive, eliminating the expensive reflection vector calculations. The difference is that Blinn introduced a half-angle vector, which is a vector halfway between the light vector and the view vector.

The half-angle vector H can simply be calculated as:

$$H = (L + V)/2$$

L – The normalized light vector from the point being shaded to the light source.
V – The normalized view vector from the point being shaded to the camera

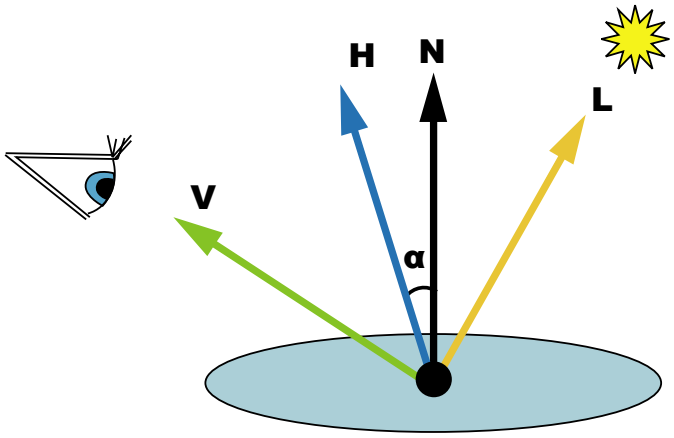


Figure 28 – Specular light model using Blinn-Phong

The specular term using Blinn's method known as Blinn-Phong is then calculated as follows:

$$S = S_l \times S_m \times \max((N \bullet H), 0)^{n}$$

S_l – The light's specular intensity
S_m – The material's specular coefficient.
N – The normal of the surface
H – The half angle vector
n – The specular exponent

Phong and Gourad shading

Phong shading [Phong73] is a technique that is often used together with the Phong lighting model to shade the polygons of a model. The two different techniques are often mixed up because of their similar names. In Phong shading the shading is done by interpolating the vertex normals across the surface of a polygon, and evaluating the Phong lighting model at each pixel. This gives pixel precision to the lighting model which looks very good but is computational expensive to do. In OpenGL however the Phong shading method is not used for shading the result instead a simpler interpolation technique called Gouraud shading [Gouraud71] is used. Here the Phong lighting model is only evaluated at the vertexes and the results are then linearly interpolated across the whole polygon. A big problem with the Gouraud shading algorithm is that the specular highlights doesn't look so convincing and can sometimes disappear. They will also fade in and out in intensity if the object or light sources are moving during an animation.

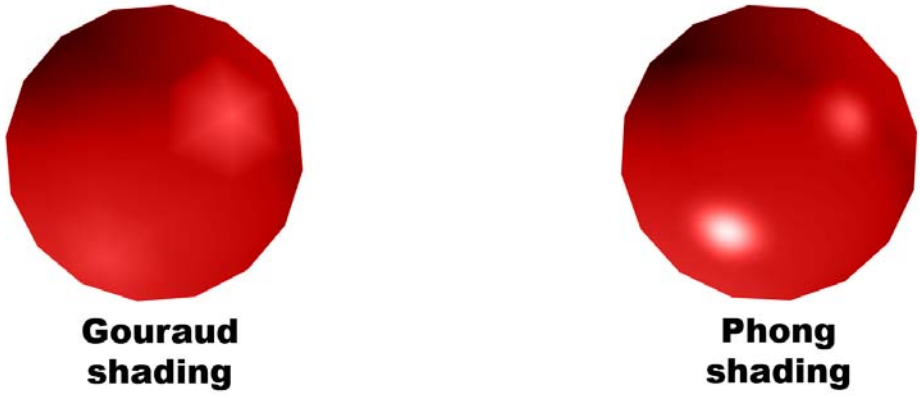


Figure 29- Gouraud shading versus Phong shading

2.4.3. OpenGL Geometry Pipeline

Throughout this report we will talk about methods and calculations performed in different spaces in the geometry pipeline. For an overview of the different existing spaces and transformations matrixes used to alter between them, see figure 30. For more information about frames and transformations the reader is referred to chapter 5 "The graphics pipeline" in [Watt99] or any similar source in 3D computer graphics.

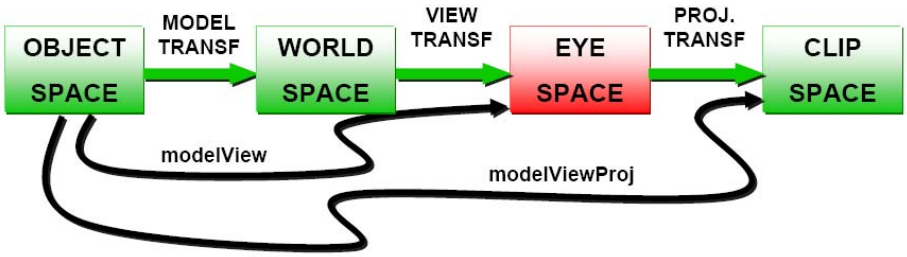


Figure 30 – OpenGLGeometry Pipelinte

2.4.4. OpenGL Stencil Buffers

The stencil buffer is a buffer that is used in OpenGL to do different types of masking operations. The operation is similar to a real stencil in the way that you can use it to control which parts of the rendered screen, i.e. the frame buffer, gets updated. For example; every pixel for which the corresponding stencil buffer bit is set will not be updated when rendering the scene.
The stencil buffer is controlled by the stencil test, stencil function and stencil operation. It usually occupies one bit per pixel that stores the extra masking information for that pixel. This information can not be seen directly on the screen. Instead it will get updated during the rendering to the frame buffer, if stencil test is enabled.
The stencil function controls whether a pixel is discarded or not by the stencil test, and the stencil operation determines how the stencil buffer is updated as a result of that test. This might be used, for example, to implement reflections by ensuring that the reflected image is constrained only to a particular area, such as the mirror and not the wall on to which it is fixed. However, it can also be used for much more complicated effects: For example in using shadow volumes, the stencil buffer is updated in a complicated way to indicate whether a point is in or out of shadow.

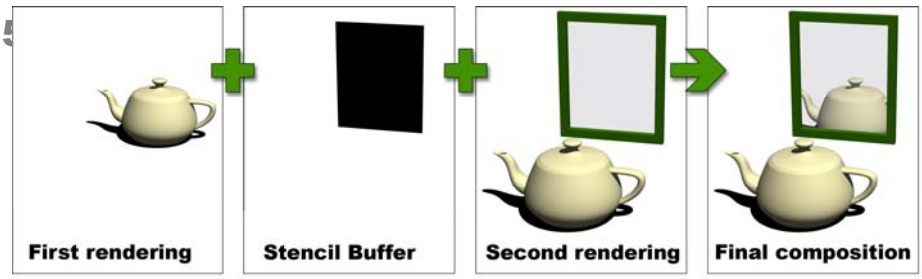


Figure 31 – Stencil buffer example

GPU Programming

In the last ten years the evolution of the graphics hardware has been even higher than the one of CPUs. Today graphics processor performance increases at approximately three times the rate of microprocessors. In addition to becoming more powerful and much cheaper, the graphics hardware has also become far more flexible and the function sets has rapidly increased. The evolution of consumer graphics cards in the latest years has introduced the GPU (Graphical Processing Unit) as a new mainstream programmable processor for fast parallel calculations. The evaluations has mainly been driven by the demands of the gaming industry but now people from other areas are starting to see the great possibilities this new hardware has made possible. As mentioned the GPU is targeted towards handling graphics and is therefore very fast at transformations, coloring, texturing and shading operations. However the parallelism is also highly suitable for other tasks and people are now starting to use the GPU as a general purpose vector processing unit for heavy calculations and simulations such as physical simulations and fluid dynamics. In this thesis the focus will be on real time graphics and the use of the GPU in its main area.

It was not until very recently that high level real time shading languages started to pop up on the market. Before that, programming the GPU was very hard and had to be done either by configuring fixed-function pipelines by setting states such as the texture-combining modes or using assembler. These programs became very hardware specific making it hard to use the code on several different GPUs. This was also the main reason why real time shader programming didn't take of before the introduction of the high level languages. Shader languages for non real time operation have existed for over twenty years and the most famous one being the Render Man Interface Standard developed by Pixar Studios [Hanrahan90] in order to program high quality shaders for films and commercials. The former has also been an important model for the real time high level shading languages we se today. The benefits of such a language are many:

- The code becomes much easier to write and to understand.
- The portability between different kinds of hardware is greatly increased.
- The low level code optimization can be handled by the compiler.
- The debugging and development becomes much easier.
- It is easier to modify and build on existing shaders.

There are mainly three different high level shading languages on the market today.

- C for graphics (Cg) produced by NVIDIA.
- High Level Shading language (HLSL) produced by Microsoft and a part of DirectX 9.
- OpenGL Shading Language (GLSL) that is part of OpenGL 2.0.

GLSL was not available when the project started and HLSL and Cg are almost identical given that they have been developed together by NVIDIA and Microsoft. Since the chosen platform for the system was OpenGL, Cg became the language of choice.

2.5.1. Cg

The syntax of the language is based on C which makes it easy to pick up for people that are used to C/C++ and java. It can handle functions, conditionals and flow control such as *if, else, while* and *for*. The language has built in optimized functions for vertex and matrix operations such as multiplication, square rot and dot product as well as built in graphical functions for texture handling etc. In order to handle different kind of hardware the concept of profiles is introduced. Since not all GPU:s support the same functions, a Cg profile defines a subset of the full Cg language that is supported on a particular hardware platform or API.

The programming model of a GPU is very different from that of the CPU. This is because of the fact that a CPU is a single unit that is sequential in nature. This means that one command or calculation is done at one variable at a time. The GPU:s of today on the other hand consists of two different programmable units, the vertex processor and the fragment processor, and several other non-programmable units that are linked together by data-flows, se figure 32. Since the pipeline can work on parallel data, calculations on streams of vertexes or fragments can be done simultaneously.

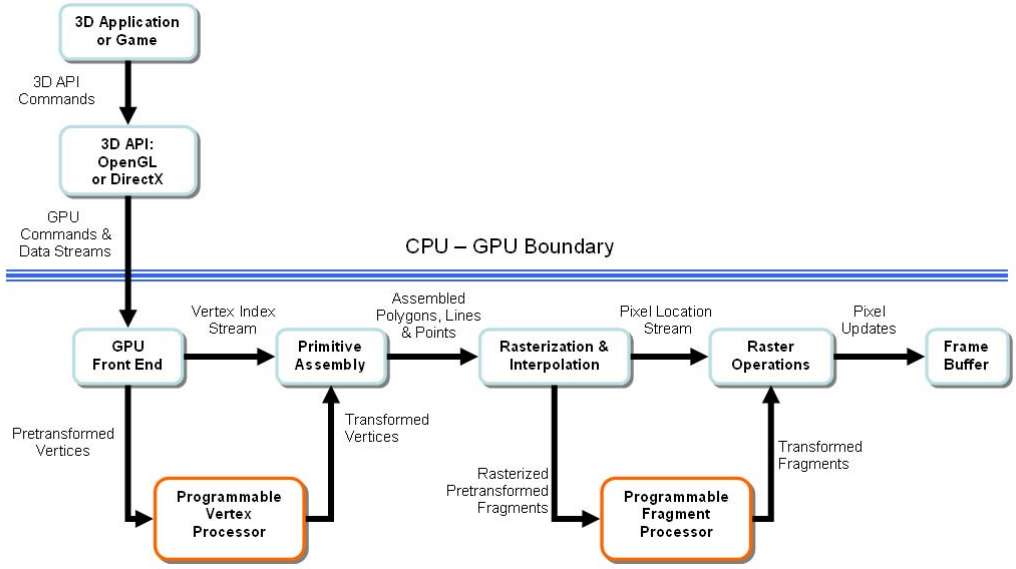


Figure 32 – The use of GPU programming in the normal 3D graphic pipeline

In Cg it is possible to write programs both for the vertex and the fragment processor. And they are referred to as vertex programs/shaders and fragment programs/shaders, respectively. Fragment programs are also known as pixel programs or pixel shaders. Often the vertex and fragment program together are simply called a shader. The normal CG pipeline can

be seen in figure 33. In order to simplify the understanding of the pipeline a typical example of the usage for rendering objects by means of vertex and fragment shaders is illustrated below.

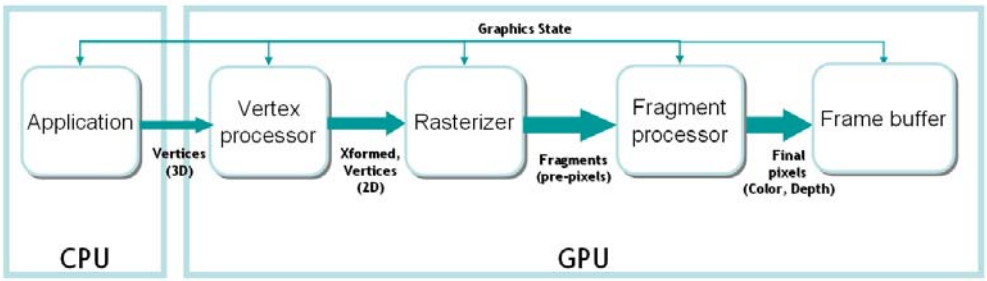


Figure 33 – CG:s graphic pipeline

The different states in the pipeline:

1. **Main 3D Application**
   All the main logic of the application is handled here, e.g. program logic, data handling, user interaction, AI etc. This part of the program is run entirely on the CPU. The only thing that is sent to the GPU is the vertexes that are to be rendered along with their properties and textures.

2. **Vertex Processor**
   Here all the transformations of the vertexes and there properties are handled (e.g. model-space positions, normals, texture coordinates.). Finally the projection from world space to image space is calculated giving the 2D positions of the vertexes. The processor is also often used for computing per-vertex lighting and vertex animations.

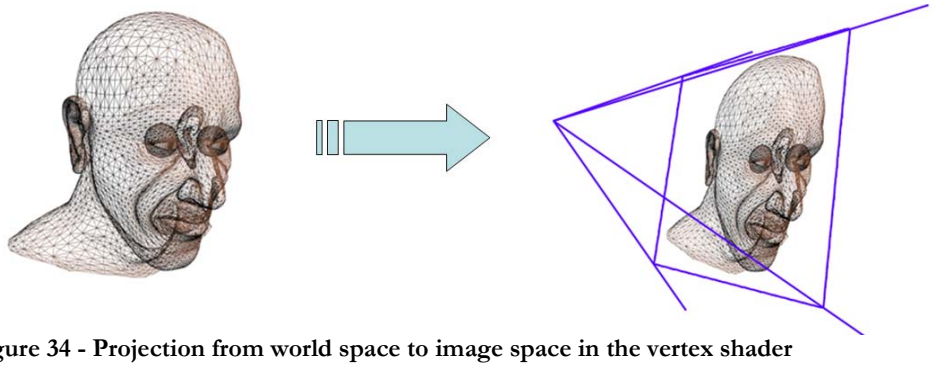


Figure 34 - Projection from world space to image space in the vertex shader geometry (picture from developer.nvidia.com)

3. **Rasterizer**
   This works the same as in OpenGL and is not yet programmable in today's hardware. The geometric pixel representations are rasterized into image based fragments by interpolating between vertexes.

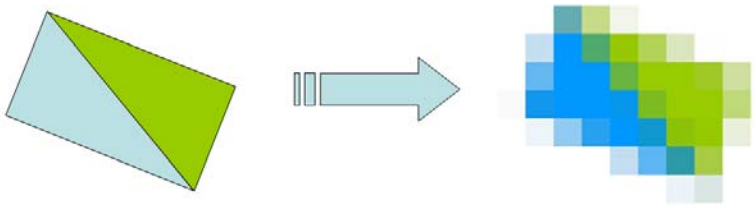


Figure 35 – Rasterization of geometry (picture from developer.nvidia.com)

4. **Fragment processor**
   Here the final colors and depth of the pixels are computed using passed in properties from the vertex shader and from the main application. This is also were the texturing is handled. The final output of the pixel is often a combination between different texture maps as well as different calculations such as lighting.

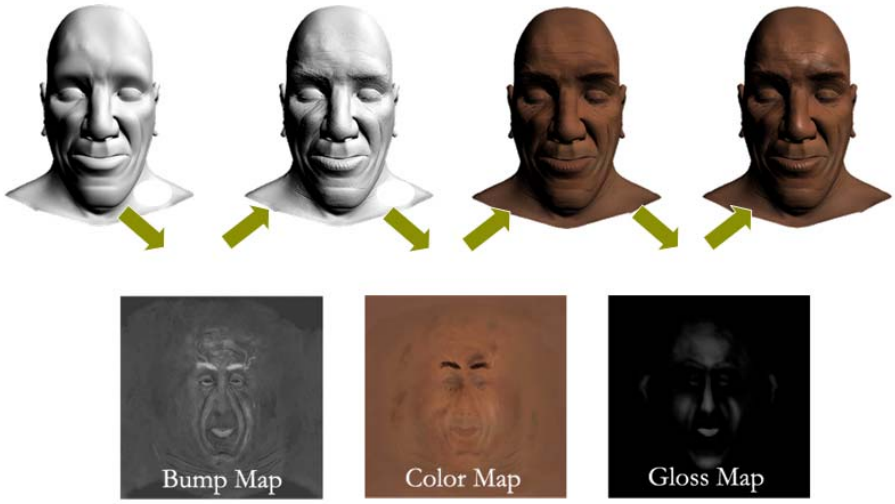


Figure 36 – Shading an object using textures in the fragment shader geometry (picture from developer.nvidia.com)

Cg code can be compiled into GPU assembly code, either at runtime or before. To actually run the shader much code has to be written in the OpenGL environment handling initialization, debugging, passing of parameters, cleaning up etc.

2.6. Shadows

A very important aspect of objects in a 3D environment is their shadows. Shadows can make a scene appear more realistic but more importantly they give a lot of information about the objects in the scene.

Figure 37 – Shadows help us to better understand the position, size and orientation of objects in the world

Shadows help us to better understand the position, size and orientation of objects in the world. For example without a shadow on the ground it is very hard to estimate the distance from an object to the floor. The shadows also provide a good aid in understanding the geometry of the casting object as well as the receiving objects.
The physiological aspect of shadows has long been researched and one of the first scientists who started to analyze them and to understand their importance was Leonardo Da Vinci, focusing on paintings and static images.

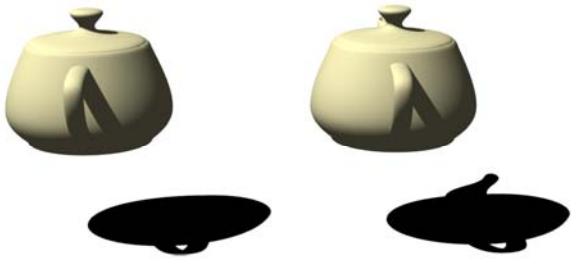


Figure 38 – The shadows provide a good aid in understanding the geometry of the casting objects

Since all real objects cast and receive shadows, this is also a highly desired property for virtual objects in a computer generated world. Without shadows the virtual objects would look totally out of place. They would then be easy to distinguish from the real world, thereby reducing the illusion of immersion that is highly desired in Augmented Reality solutions.

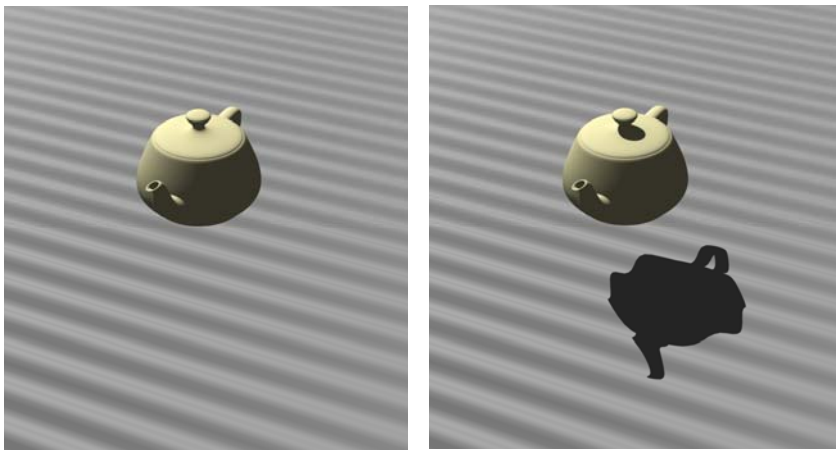


Figure 39 - The shadows help us to understand the geometry of the receiving objects

A shadow is defined as an area that is only partially irradiated or illuminated due to blockage of light by an opaque object [Mat99]. This means that a face of an object is in shadow if it is not lit by all the light sources in the scene. The object that is causing the shadow is called the occluder and the object lying in shadow is called a receiver. Since the lighting models used by OpenGL and Direct3D are local they do not directly support shadows except to the very limited extent that a surface facing away from a particular light source get no diffuse or specular contribution from that light, called an Attached shadow. This basic shadowing effect only takes the orientation of the surface compared to the light source into consideration. As an example it would not be possible for a table to cast a shadow on the ground using only this technique. Since the ground would always be facing the light it would always be lit. In order to handle Cast shadows, which are shadows cast on objects because of the blocking of illumination, more advanced techniques must be used. These special techniques can then be combined with the rest of the rendered scene. Self-shadows are a specific case of cast shadows that occur when the shadow of an object is projected onto itself. This means that the occluder and the receiver are the same object. Not all algorithms can handle self-shadows.

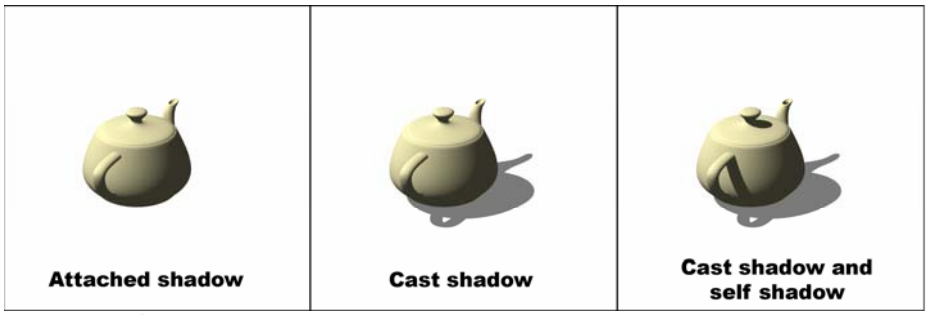


Figure 40 – Different types of shadows

An object lighten from a single point light source will generate a sharp shadow in a nearly sole ambient color i.e. a hard shadow. Although this type of shadows are usually enough to approximate the interaction of the sun on a clear day or a single light bulb in a room without windows, they are not enough for more complex scenes. However since they are the most easily generated type of shadows and often also the basis for more complex methods we will start by examining techniques for generating these. We will then move on to soft shadows i.e. shadows that have a smoother look with a gradient transition from the darkest to the lightest spot in the shadow.

2.6.1. Hard Shadow rendering techniques

Today's existing techniques for rendering computer generated hard shadows can be divided into four different categories:

- projection shadows
- shadow maps
- shadow volumes
- ray-tracing

It is often fairly computationally demanding to generate shadows and much data needs to be processed. It is also common to compromise the mathematical correctness of the shadows to be able to compute them in real-time.

These techniques only generate sharp shadows in their normal operation. A pixel is either rendered in full shadow or not in shadow. However this doesn't necessarily mean that a pixel in full shadow is totally black. A simple often used extension is to render the scene an extra time with only an ambient light value in order to decide the color of the pixel in shadow. Often this extra rendering pass is done first and only the pixels displaying objects that are lighten by the light sources are rendered afterwards. The results are then mixed together using a stencil buffer. If the objects in shadow were to be rendered totally black the shadows would look extremely unrealistic.

Projective shadows

First introduced by Blinn in 1988 [Blinn88], the technique is original and simple but can only cast shadows on planar surfaces. This means that the method is only applicable in certain situations for example when casting shadows on a flat floor or wall. The goal is to generate a new polygonal object to resemble the shadow by projecting the objects geometry on a plane with a transformation matrix. Given the equation of a 3D plane and the homogenous position of the light, a 4x4 matrix called the planar projected shadow matrix can be constructed. The matrix projects 3D polygons onto the specified plane based on the light source position. The polygonal object transformed by the planar projected shadow matrix will still be a 3D object after the projection but with no height and placed on the 3D plane, which is the effect that is aimed for.

Extensions and Improvements
The method has some annoying errors and shortcomings that have to be taken into consideration. The most obvious is of course the fact that it can only be used for planar objects. Another problem is that because of precision error the projected polygons can be rendered beneath the receiver surface and the plane will show through the shadow. A solution to this problem is to add some bias to the plane so that the shadows are always drawn in front of the plane. Unfortunately, setting the bias is not so easy – too little bias will not remove all the artifacts and too much could cover other objects.

Another problem is that the occluders can be projected outside the receiving object because they are projected to planes and not to polygons. This can be solved using the stencil buffer while rendering the shadows. The last big problem with the method is that it will generate incorrect shadows if the casting object is behind the light source, called anti shadows, or behind the receiving plane which is called false-shadows as discovered by Blinn. A solution to the problem is to use extra clipping planes or only use the technique in scenes were these problems don't occur. Another solution suggested by Herf et al [Herf97]. is to use 3D to 3D projections instead of 3D to 2D as in Blinn's method. This transformation maps the false shadows casting objects out of the camera view. In this way they exclude the objects, which are behind the plane or behind the source of light that causes the improper shadows in the original method.

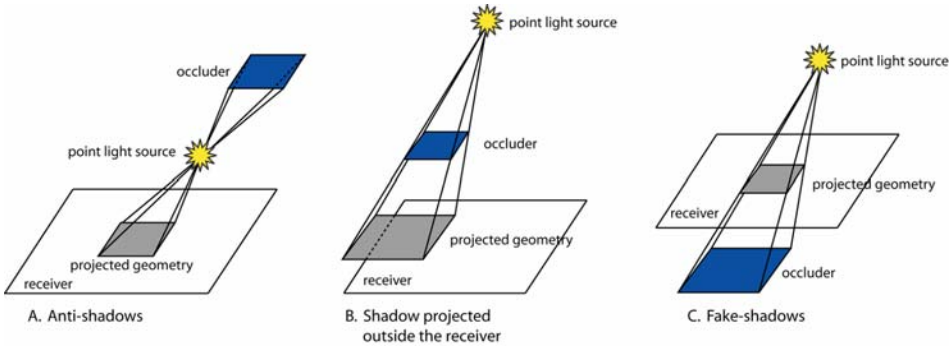


Figure 41 – Known problems with projective shadows

The left picture shows what happens if the casting object is behind the light source called an anti shadow.
The center image shows a projected shadow that is projected outside of the receiving object. The right image shadows the case when the casting object is behind the receiving plane which is called false-shadows.

Advantages

- Casts a shadow onto a plane with good-looking results
- Very fast for projecting shadows in small scenes with only one ground plane

Disadvantages

- Don't allow concave objects to cast shadows on themselves or objects to cast shadows on each other.
- Can generate incorrect shadows if the casting object is behind the light source, called anti-shadows, or behind the receiving plane, called false-shadows.
- Because of precision errors the projected polygons can be rendered beneath the receiver surface and the plane will show through the shadow.

Shadow Volumes

In this approach the shadows that are generated are entirely geometrical. In the method the goal is to the find the volume that a given set of occluders generate from a specific light source by finding the silhouette of the given occluders along the light direction and extruding these in the direction of the light. A three dimensional volume can be constructed that vaguely resembles a pyramid. The idea is then to use this pyramid as a simple technique to defining which objects are in shadow and which are not for the specific occluder and light source. All the objects inside the shadow volume are in shadow and should be lighted only by an ambient color while the other objects are not affected by the specific occluder. The shadow volume approach was first introduced by Crow et al [Crow77]. Heidmann [Heidmann91] was then the first to implemented Crow's algorithm on graphics hardware using the stencil buffer and we will start with explaining this original zpass method. Thereafter we will discuss advantages and disadvantages as well as some improvements that are commonly used.

The shadow volume zpass algorithm is divided into several steps.

- Before anything else can be calculated, the silhouette edges must be found. A simple technique that is sufficient for the method is to choose all edges that share one face that is turned towards the light source and one that faces in the opposite direction. This is called the visibility test and can easily be done by calculating the dot product between each face normal and the light source. More edges than necessary will be generated.

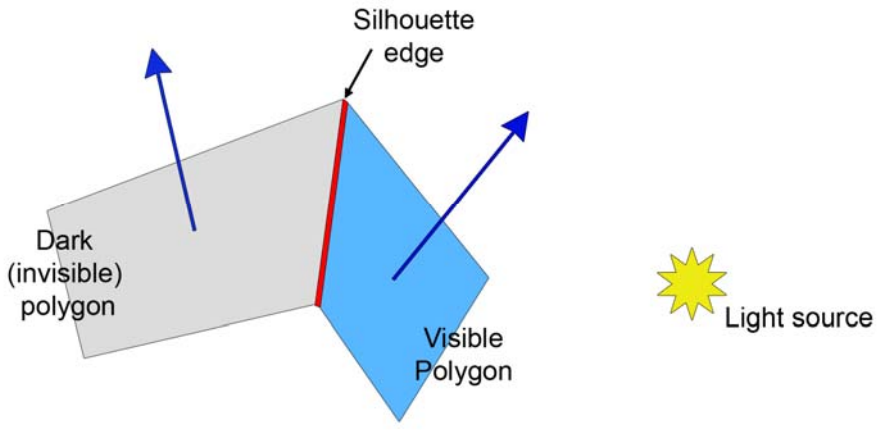


Figure 42 – Finding silhouette edges

- Next the planes building up the volume are constructed by extruding the found faces in the direction away from the light source. This can be done either to a finite or an infinite distance depending on implementation method.

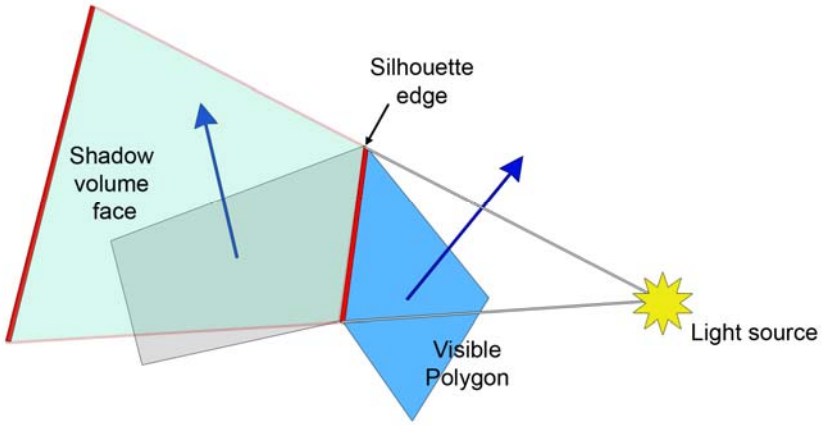


Figure 43 – Extruding faces to create a volume

- When we render the scene we know that all objects inside the volume are in shadow. Since the volume created can be very complex we still need a simple way of knowing if we are inside or outside the volume when we are rendering a specific pixel. The solution is to count the number of faces of the shadow volume that we are crossing when we are traveling from the object rendered to the camera. Every face facing the camera is counted as plus one and every face facing the opposite direction is counted as minus one. As long as the count is none zero we know that we still are in the shadow volume and the pixels should be rendered as if in shadow. When implementing this step a stencil buffer is used to keep track of the count of traversed faces. If a ray from the eye goes through both a front and a back face, then it intersects the shadow volume an even number of times so the point is lit; if it pierces only a front face but not a back face (or vice versa), then it intersects the volume an odd number of times, so the point is in shadow.

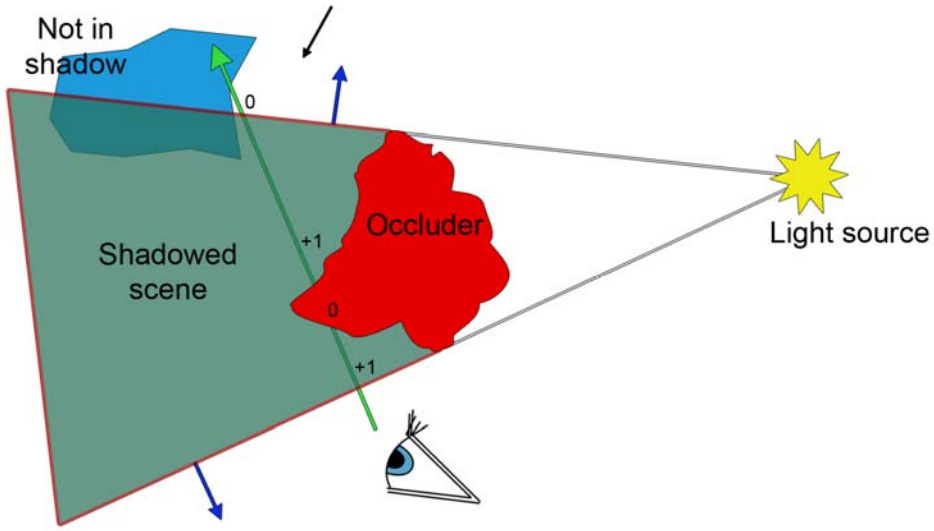


Figure 44 – We need a simple way of knowing if we are inside or outside the volume when we are rendering a specific pixel.

In order to properly render a scene shaded with volume shadows the following render passes has to take place:

- Render the scene with full shading but only ambient lightning i.e. the lighting that the objects in shadow will have. Find he edges of the occluding objects and generate the shadow volume.
- Render the front faces of the shadow volume to the stencil buffer incrementing it for each face. Here we can turn of all shading and use flat shading to speed up the process because we are only interested in Boolean values for the stencil buffer.
- Render the back faces to the stencil buffer decrementing as we go. The same simplifications as on the previous step can be made here.
- Render all objects a forth time with full lightning this time leaving out all the pixels that has a positive value in the stencil buffer.

Extensions and Improvements

In order to speed up this technique one has to concentrate on optimizing the amount of rendered polygons because of the big fill rate that they consume. An improvement from Everitt et al. [Everitt02] is to constrain the shadow volume and only concentrate on rendering what is really necessary. For this they use different software and hardware methods such as scissoring. In order to solve an error in the original method they also use the stencil buffer in a different way, known as the zfail approach. The normal zpass method usually works fine except for the single case when the viewpoint is in the shadow volume. Then the counting of faces gives the wrong result. The solution is to use the zpass technique but backwards; first rendering the back faces incrementing the buffer, and then the front faces

decrementing. One is then left with information about the intersections in the stencil buffer.
A different improvement is to use a pre-calculated BSP tree as suggested by Batagelo et al [Batagelo99]. to reduce the number of polygons, but the method doesn't work well with dynamic lights or many moving objects.
In order to speed up the rendering much effort has also been put into the research of hardware optimizations. McCool[McCool99] has suggested a method that computes the silhouette edges by first rendering a shadow map, but this method requires that some information is read back from the depth buffer which is also costly. Brabec et al [Brabec03]. on the other hand have suggested a method that uses programmable hardware to calculate the silhouette edges which works faster than the original approach. With some of the newest extensions to OpenGL it is also possible to do the rendering of the stencil buffer tests for shadow volumes in a single pass thereby reducing the number of passes required for the method which is a great timesaver.

Advantages

- Handles self-shadowing
- Works for omni directional light sources
- Renders shadows with pixel precision
- Can handle and render correct shadows in very complicated scenes for example when the light source is inside a complicated object that the light should shine through.

Disadvantages

- The numerous large polygons that make up the shadow volume has to be rendered which requires a large amount of fill rate from the graphic card thereby leaving less processing power to the rendering of the rest of the scene.
- At least two rendering passes are required usually three to four to render a complete shaded scene.
- Computational time is related to the complexity of the occluding objects
- Requires that the silhouettes of the occluders are determent
- The objects rendered needs to have meshes without any cracks, because otherwise the light will leak through the object from certain angles. A crack is a hole in the mesh that occurs when not all edges in the object are surrounded by faces.

Shadow Maps

This technique was first discovered by L Williams et al.[Williams78] in 1978 and marked an important step in the evolution of shadow algorithms. It is a method often used for high-end offline rendering for movies and television. It is for example implemented in Pixar's Renderman and 3DS Max and has been used on major films such as Toy Story. But as we shall se, apart from being a rather fast technique it has some annoying drawbacks that have to be taken into consideration when doing real time applications. The shadow map algorithm consists of two major steps. First the scene is rendered as seen from the light source into a Z-buffer commonly known as a shadow map. Here the z values of the nearest pixels are stored for later use. Next when the scene is rendered from the cameras point of view, the points on the objects generating the actual screen pixels are transformed into the light sources coordinate system and compared to the values in the shadow map. If the distance from the object to the light is greater than the value stored in the shadow map, we know that the specific point is in shadow. A single shadow map is sufficient for directional lights and spot lights; omni directional point lights require 6 shadow maps to cover all directions.

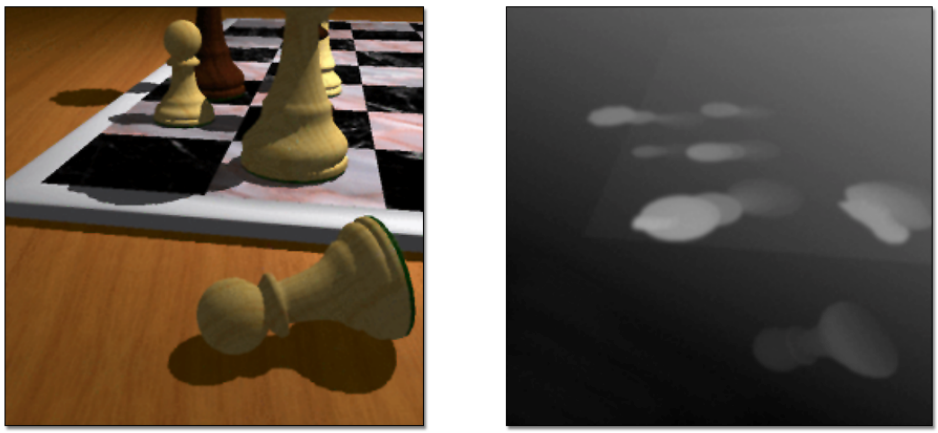


Figure 45 – Shadow map rendering and shadow map depth texture (picture from http://www.devmaster.net)

Because the method works in image space, compared to object space for projective and shadow volumes, several advantages can be found but also several disadvantages. A big advantage is the fact that all objects that can be rendered, also can be made to cast shadows so the objects doesn't have to be made of polygons. This is especially useful for casting shadows from objects that get their shape from masking textures. Take for example a tree that is made of a simple plane that always faces the camera, this is called a billboard. The billboard is then textured with an image of a tree and a texture mask defines the actual shape of the object. If an object space model would be used to render the shadows from

this object the result would be a rectangle while if we used shadow maps the shadow rendered would look correct.
The biggest disadvantage is that the method often generates shadows with visible pixilation or alias problems. This is because of the image based nature of the method and is highly visible at shadow borders when the projection shadow transform magnifies the shadow map. What happens is that the projection matrix needs to transform the shadow map depending on the position of the camera, occluders and light which sometimes magnifies the texel values. This problem is worst when the light source is far away from the viewer or in scenes with big depth ranges, where nearby shadows need high resolution, whereas a low resolution would be enough for the distant shadows. One solution is then to resize the shadow map so that it is two to four times larger than the screen resolution, but even though the aliasing then becomes less noticeable, it is still often visible and the processing load increases as well as the texture memory consumption. Often one has to find a balance between aliasing artifacts and processing power and it is often impossible to get rid of all the pixilation problems. Some other techniques exist to solve these problems such as Perspective Shadow Maps and Percentage Close Filtering but they suffer from their own problems as will be discussed in the extensions and improvements section.

Another problem that the method shares with projective shadows is that because of precision error the projected shadow can be rendered incorrectly. The problem here is that because of the precision error in the shadow map sometimes the shadows that are suppose to be cast on receiving objects also can be projected onto the occluders, known as erroneous self-shadowing. A solution to this problem is to add a bias to the z-value in the shadow map. Unfortunately, setting the bias is not so easy – too little bias will make the object self-shadow, and too much will move he shadow away from the occluder making it look unrealistic.

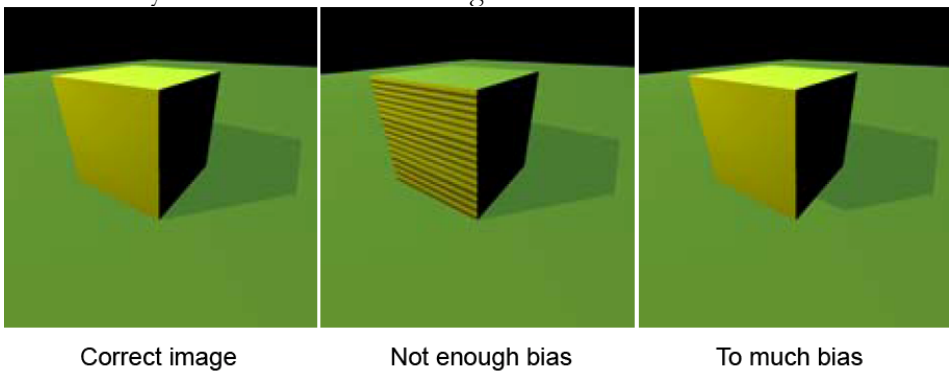


Figure 46 – Setting the right bias level (picture from http://www.devmaster.net)

Advantages

- No knowledge or processing of the scene geometry is required, since shadow mapping is an image space technique.
- Only a single texture is required to hold shadowing information for each light; the stencil buffer is not used.
- Avoids the high fill requirement of shadow volumes

Disadvantages

- Aliasing, especially when using small shadow maps.
- The scene geometry must be rendered once per light in order to generate the shadow map for a spotlight, and more times for an omni directional light.
- Because of precision error in the shadow map the projected shadows can be rendered incorrectly and a bias value has to be conveyed.

Improvements

If the surfaces on the objects are to close to each other sampling problems can occur, where the surfaces casts shadows on themselves. A solution to this is to offset the values in the shadow map buffer by a small bias value as suggested by Reeves at al. [Reeves87]. The biggest problem with shadow maps as discussed before is the aliasing problems. A solution to the problem is the Percentage Close Filtering (PCF) introduced by Reeves et al. 1987[Reeves87], were multiple shadow map comparisons are made on a pixel level and then averaged together. The method is hardware intensive and frequently used in offline software rendering packages. Two similar solutions from Tadamura et al. [Tadamura01] and Fernando et al. [Fernando01] are to change shadow map resolution by using multiple shadow maps of different resolution. Tadamura et al. uses multiple shadow maps in fixed sizes with varying resolution to reduce aliasing for outdoor scenes. Fernando et al on the other hand replaces the depth map by an adaptive, hierarchical representation that is continuously updated. However, by using multiple shadow maps, several rendering passes are required and none of the methods is easy to map to graphics hardware. Another elegant solution that also has been implemented in hardware is to compute the shadow maps in perspective space, thereby storing more information in parts of the map that are closer to the eye were the problems are most noticeable. This technique has been developed by Stamminger et al. [Stamminger02]. However it still doesn't completely solve the aliasing problems and it doesn't work in all situations.

Ray-tracing

In this technique introduced by Appel in 1968 [Appel68] rays are cast from the light sources, reflected and refracted numerous times in the scene when they interact with different objects and will finally arrive at the view plane give the resulting color of every pixel in the scene. Effects such as reflections, refraction and shadows, which are difficult to simulate in many other algorithms, follow naturally from the ray tracing algorithm. This is the optimal method for rendering correct looking hard shadows and is often used in offline rendering when speed is not the main concern. However the technique is still to computational intensive to be considered for any serious real-time shadow generation. Because of this the technique will not be explained in greater detail.

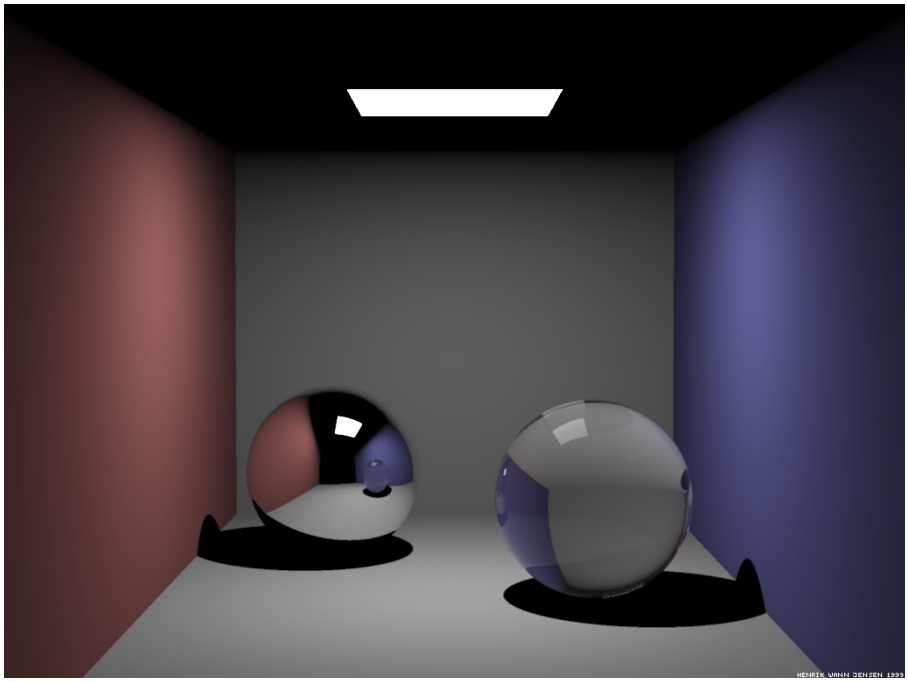


Figure 47 – Simple ray-raced scene (picture from http://graphics.ucsd.edu/~henrik/)

Advantages

- Gives very good looking and correct hard shadows.

Disadvantages

- Is to computational intensive to be considered for real time shadow generation in complex scenes and with complex objects.

2.6.2. Soft shadows rendering techniques

Most often a shadow will have a smoother look with a gradient transition from the darkest to the lightest spot in the shadow. The light sources are then often seen as areas or volumes emitting light, from were sample points can be taken. A common way of classifying the parts of a soft shadow is to say that a point on the shadow surface that receives light from only some of the light sample points in the environment is called penumbra. While a point that doesn't receive light from any light source is called umbra. The penumbra part of a shadow will fade the intensity down to zero from the umbra. The amount of intensity on a given point in the penumbra depends on how much of the light source can be seen from that point. Several techniques exists for generating soft shadows and they are all fairly complex and computer intensive. In many methods the regions of the umbra and penumbra are only estimated and then the rest of the shadow is filled in. This can result in incorrect looking shadows but which often is not noticeable during animations.

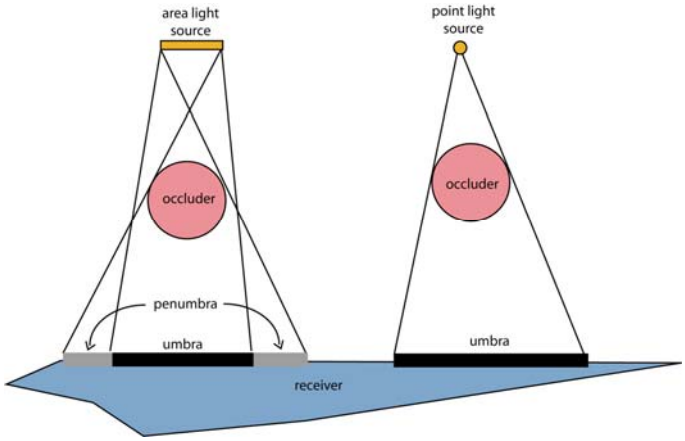


Figure 48 – Area light source with penumbra versus point light source

Ambient occlusion

Ambient occlusion is a lighting technique used to give models a global illumination-like effect. The technique was introduced at SIGGRAPH 2002 by Industrial Light and Magic and has been used by effect studios for many years as a way to get better visual results [Pharr04]. A simple way to look at ambient occlusion is to se it as a more advanced form of an ambient lighting term than the one used in pong shading. The problem with pong shading that is used in OpenGL is that since only local illumination is considered the objects generally look to dark and an ambient scalar value has to be added in order to uniformly lighten the scene. This will of course also lighten cracks and contact surfaces on objects which are not desired. The solution comes in the form of ambient occlusion where the ambient values of the objects surface points are calculated depending on their position on the objects. For example a point under a table will be darker than a point on top of the same table in general. The objects rendered with ambient occlusion will look like as if they were lit from the entire hemisphere (rather than a point light) which is close to the truth in a room with many lights and windows or a cloudy outdoor scene. A set of samples are collected from the hemisphere above a certain surface point on the object, and an "ambient occlusion" scalar value is computed based on how many samples that were occluded by the object itself. This scalar value is simply a ratio of how much ambient light a specific surface point would be likely to receive.

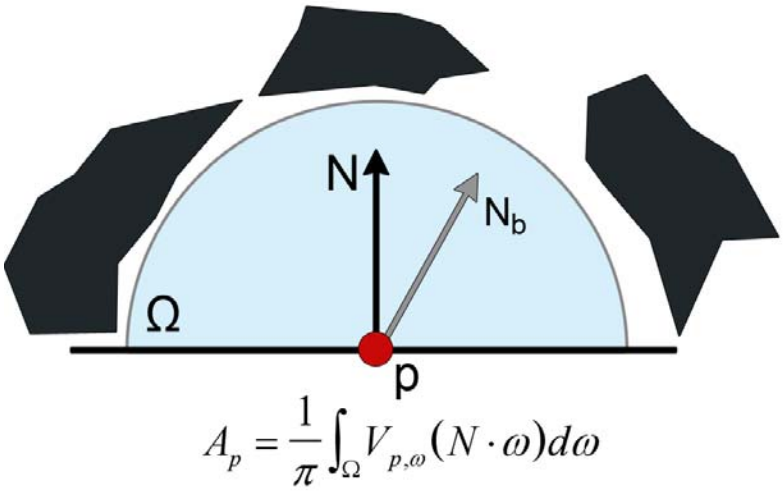


Figure 49 – Ambient occlusion

The ambient occlusion A_p at a point p on a surface with normal N can be computed by integrating the visibility over the hemisphere Ω around p. Where $V_{p,\omega}$ is the visibility function at p which is zero if p is occluded in the direction ω and one elsewhere. The calculations can be done beforehand if the objects are static but has to be done in real time or interpolated between key frames if they are animated. Several methods exist for

evaluating the integral from ray tracing to depth map based approaches. An extension is to also calculate the average light direction vector known as the bent normal vector N_b. This vector for every point will tell from were the point gets most of its lighting contribution and can be used together with an environment map to tell were in the map the point should look when coloring the object.

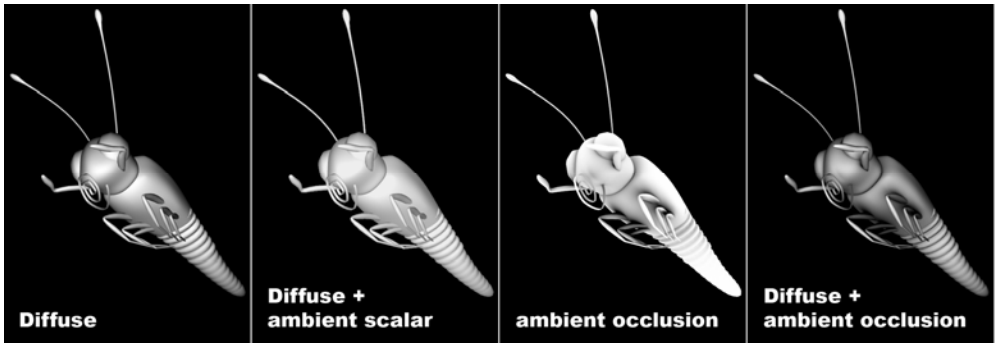


Figure 50 – Ambient occlusion usages (images from http://www.webopedia.com)

Ambient occlusion should not be seen as a substitute for normal shadow generation but as complement in order to make the scene look more realistic. The problem is that the self-shadows generated are not correct and in scenes with point light source they can be very far from the truth.

Advantages

- Very good looking shadowing effect that resembles that from real GI.
- Some steps can be pre-calculated.
- Can easily be used together with an environment map

Disadvantages

- Can sometimes be very far from how a real shadow would look in that lighting.
- Very new and untested technique for real time applications.
- Needs to be combined with other shadowing techniques to give better realism.

Fake Soft shadow with Smoothies

This algoritm by Eric Chan and Frédo Durand [Chan03] builds on the shadow map technique but with some very clever extensions. By adding extra primitives at the objects silhouettes called smoothies, the rendered fake shadows will look like real soft shadows. At the same time they will hide the aliasing artifacts that are so common in normal shadow map techniques. Even tough the shadows that are created are not geometrically correct they closely resemble shadows generated from other methods were real area light sources are used. The algorithm only computes the outer penumbra of the shadow. As a consequence, occluders will always project an umbra, even if the light source is very large with respect to the objects. This makes the scene appear darker than expected, an effect that is clearly noticeable except for very small light sources. The method is optimized for hardware rendering using shaders and is reasonable fast for a soft shadow algorithm. The authors have achieved 20 frames per second on scenes with more than 50,000 polygons.

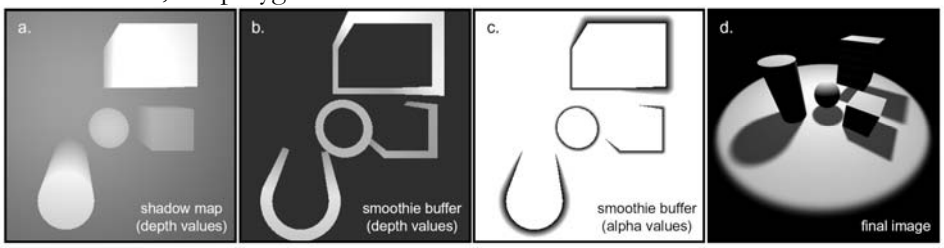


Figure 51 – a. normal shadow map, b. smoothie depth buffer, c. smoothie alpha buffer, d. final rendering (images from [Chan03])

Advantages

- Good looking soft shadows.
- Hides normal shadow map aliasing artifacts.
- Fast due to the shadow map approach.

Disadvantages

- The shadows will appear darker than they should with large light sources.
- Artifacts between adjacent edges except for small light sources.

Fast Soft Shadows

This technique by Michael Herf and Paul S. Heckbert [Herf97] is very straight forward and gives a convincing result for soft projective shadows. The light source is seen as an area light source from which a sequence of jittered samples is taken. For each sample a hard shadow is rendered into an accumulation buffer using projective shadow techniques. The results in the accumulation buffer are then averaged together into a texture that is put on the plane object, normally the floor or a wall. Many samples are needed in order to get a photo realistic result (256-512) but the rendering will increase in realism already after a few samples. The big drawback is that the method can not handle self-shadows which only make it interesting for some areas.

Figure 52 – Fast soft shadows in action (image from [Herf97])

Advantages

- Simple approach with good looking results

Disadvantages

- No self-shadowing

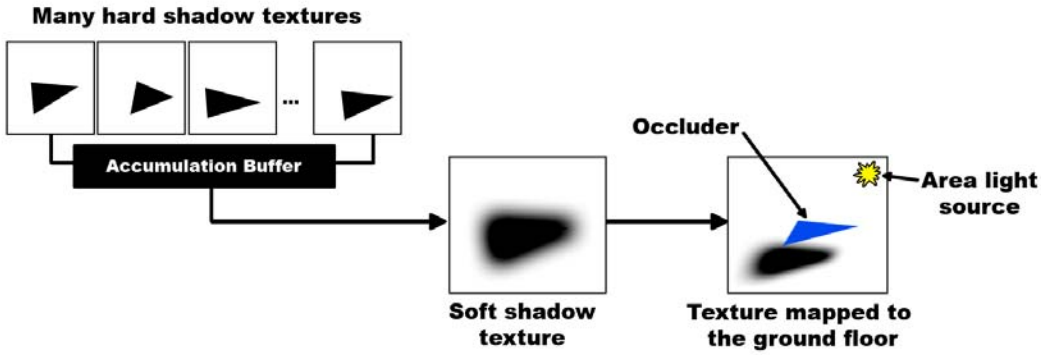


Figure 53 – Fast soft shadow operation

Radiosity

Figure 54 – Radiosity rendering (picture from www.iblcham.ch)

Radiosity is a Global Illumination method meaning that the light is modeled in a more physically correct way taking into account both direct and indirect lighting caused by diffuse reflections. In the method light is seen as energy that bounces around in the environment spreading the light. Every object in the surrounding will affect the lighting of all other objects thereby introducing effects such as color-bleeding and perfect soft shadows. Color-bleeding is the phenomena that make for example a diffuse red object cast a reddish hue on the floor even though the object is neither reflective nor a red light source in itself. While each object in a Direct Illumination scene must be lit by some light source for it to be visible, an object in a Radiosity scene may be lit simply by its surroundings. The Radiosity method was first introduced 1984 in a paper by Goral, Torrance & Greenberg [Goral84]. The inspiration came from the field of thermal radiation and the idea was to simulate energy (light) transference from diffuse surfaces. In order to simplify the algorithm the assumption that energy transferred between two surfaces is constant across the surfaces is made. Which means that to compute an accurate image, geometry in the scene description must be broken down into patches, which can then be recombined for the final image. The amount of light energy transferred between every pair of patches can be computed by using the known reflectivity of the reflecting patch, combined with the form factor of the two patches. This can be calculated based on their geometrical relationships and can be thought of as the fraction of the total possible emitting area of the first patch which is covered by the second patch.

To start with, only the area light sources will have any radiative energy. Next the interaction of energy from every patch to every other patch needs to be calculated which is an $O(n^2)$ problem. Then, a recursive subdivision technique is used to subdivide the meshes where ever there is an inconsistency across it. The lighting for the scene is then recalculated, storing the lighting values again. Except this time around the light that is being reflected off of each of the patches is also take into account, and all the patches are treated as if they were a light source themselves. The whole process then revives with more patches to calculate for because of the

subdivision. Until finally a user set thresh-hold is reached and the calculation is stopped. The result is a scene with thousands and thousands of polygons which all have their own light values, with Radiosity taken into account. From this radiosity processed scene the final scene is rendered using interpolation across the surfaces of all polygons.

Even though there have been many improvements to the algorithm the calculation of the radiosity solution is still extremely costly and therefore not to our knowledge possible to do in real time yet. However if the lighting conditions in a scene are constant then only the final rendering from the already radiosity processed scene needs to be rendered which can be done very fast. Constant lighting conditions mean that neither the lights nor the objects move only the camera.

Advantages

- Gives extremely good looking results.
- Reasonably fast once the Radiosity calculations have been done.

Disadvantages

- Very slow rendering.
- Not possible to do in real time.

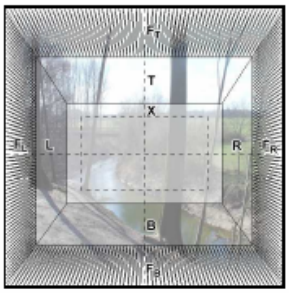
Figure 55 – Radiosity rendering of a very simple scene showing color bleeding (image from www.claus-figuren.de)

2.7. Related Work

This section discusses related work in the area of complete systems for augmented reality that tries to compute realistic lighting using some sort of IBL.

T. Ropinski et al. [Ropinski04] present a technique to render virtual objects into real scenes with a global illumination model called Virtual Reflections. A technique generating reflections on the virtual objects in real-time from the image to be augmented. They create a cube environment map from the acquired image and then use ordinary environment calculations for the reflections on the object. The constructed cube map can of course not be correctly created by using only one background image so this is the great hack of this technique. To create a correct cube map we would need six stand alone and correctly directed pictures. Since the used image in the method in reality only contains information about the back face of the cube map they are forced to construct the cube map by segmentation of the picture. The results are indeed fast but true realism is absent due to the incorrectly created cube map. The main advantage of the method is that it doesn't reuire any special equipment except the camera to capture the image, and this is almost always included in the Head Mounted Displays for AR applications. This method could be used in applications where true realism isn't that much of a deal and where instead speed is favored. Possibly there might be some use for this method in future AR gaming.

(a) virtual reflections



(b) segmentation of the scene image

(c) the generated cube map

Figure 56 – Segmentation and projection of image data for the generation of the cube map

Gibson [Gibson04] presents a complete framework for rendering correctly illuminated AR scenes. The framework shades the virtual objects consistently with the rest of the scene and also cast soft shadows between the worlds.

The presented system use a static camera set up which means no marker is needed to calculate extrinsic parameters. The scene is also reconstructed as a semi-automatic step where he uses primitive-based modeling to set up the most important parts of the scene. This is done to resolve occlusions between real and virtual objects but also for collision detection and the construction of an illumination environment used for the shading and shadowing of virtual and real objects. As a pre-process step the illumination is captured using a light probe and HDR imaging. Since this is done only once the framework is not prepared for changing light conditions. From this information the system builds an illuminated scene model by projecting captured radiance values to the environmental model. The shading of the virtual objects is done by separately render diffuse and specular reflectance. The diffuse shading is done by using an Irradiance Volume where the irradiance is stored in a volume based voxel representation. This means that in every grid in the scene there is an irradiance value, which costs a lot of computational power and storage.

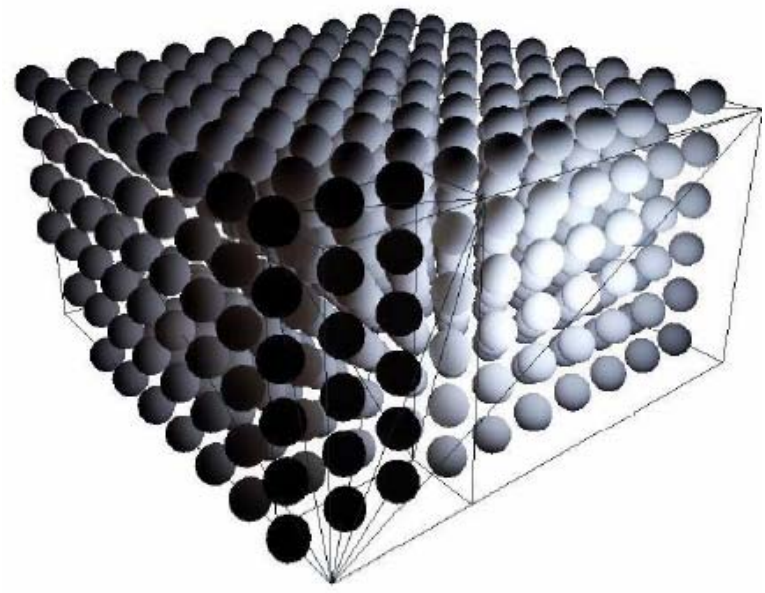


Figure 57 – Volumetric representation of irradiance at every point and direction in space

This is computed using Spherical Harmonics representation of the volume which leads to far more efficient computation. Any object can be shaded accordingly to its position in the scene and its diffuse reflectance properties. The specular reflections are generated using cubic environment maps. Since

using HDR imaging they must tone-map (HDR->LDR) all the result using the camera response function. The diffuse and specular components are tone-mapped individually and combined. The framework accounts for four types of shadow:

- Virtual-to-real (shadow cast by synthetic object)
- Virtual-to-virtual (self-shadowing on synthetic object)
- Real-to-virtual (shadow cast onto synthetic object)
- Real-to-real (existing shadow in the real scene)

By using special hierarchy the systems pre-computes and stores light transports for receivers in different hierarchies. This is later used to render a shadow map that encloses the synthetic objects with the given radiance transfer from sources and emitters. The self shadows are approximated using ambient occlusion. This is a very complex and in by all means an amazing framework. The result is impressive and works in real time. The lack of a dynamic camera set up reduces the functionality of the system and all the pre-calculation needed doesn't make the system that much of a real time system after all.

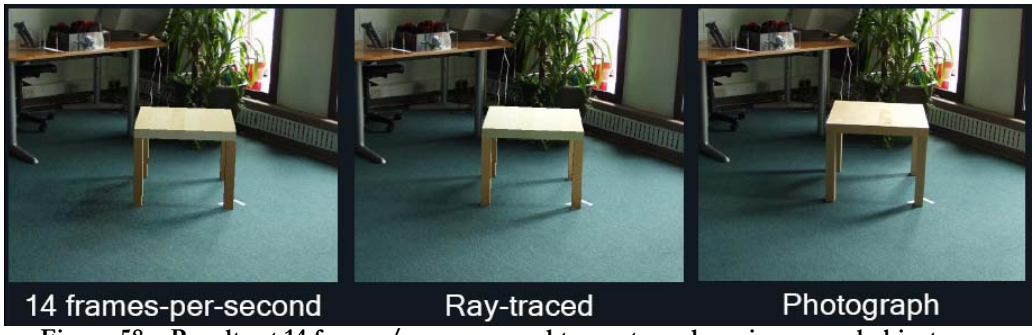


Figure 58 – Results at 14 frames/sec compared to ray-traced version an real object

Pomi et al. [Pomi04] presents a system for rendering synthetic objects into a real scene using a distributed ray tracing network. They capture the incident light with a HDR video light probe. The synthetic objects are then lit using a global ray tracing illumination method. The method also supports smooth shadows between the worlds. This is of course a very computationaly expensive method and that's the reason for the distributed network. The framework sends small parts of the scene to different clients for individual process. The result is very visually appealing but the cost for that is high. With 24 clients running dual Athlon MP 1800+ the car in Figure 59 can be rendered in 4.5 frames/ second.

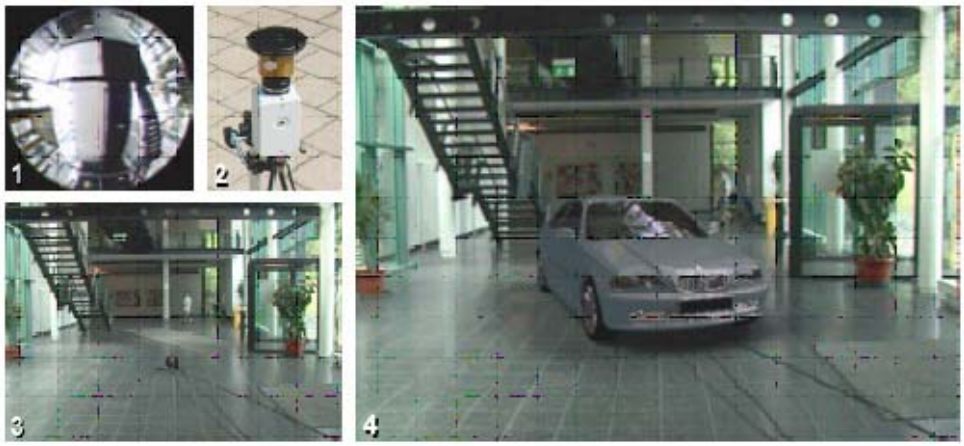


Figure 59 – Car model inserted into a live captured background. 1) A light probe HDR frame 2) HDR camera, 3) Background frame 4) Augmented view with shadows and light.

3. Implementation

3.1. IBL

3.1.1. Introduction

The first problem we had was how to actually capture our light information. The first idea we had was to do a pre-capture of the illumination just like Gibson [Gibson04]. That would have given us the possibility of using a high quality digital camera and take a series of photos to create a first-rate HDR picture. With this approached we knew we could achieve a great illumination map for our objects but the consequences was that the system would become rather static. A pre-calculated scene would not be able to adapt to new lighting conditions in the world. We also wanted to avoid reconstruction of scene geometry to make the application as dynamic and easy to use as possible. This meant that the inserted objects also had to be close to where the capture was done, since that's the only place where our capture is truly valid. Since we wanted to move our objects around in the room this would most certainly result in false information from a fixed illumination map. We also planed to make a system where we acquired several captures from different positions in our room and then depending on the position of our object we should interpolate a value from the existing maps. We realized that interpolation of light captures was not linear and would give us strange results, and also that we actually didn't have a clue where our object were in respect to our room. The only fact we had was the position of the marker in respect to our camera. We would have been forced to put markers all around our room for an object-to-world knowledge.

Fixed precalculated illumination maps	
Advantages	**Disadvantages**
High quality (Controlled capture)	Not dynamic to illumination alterations
Possible HDR image	Can't move objects in world if environmental reconstruction is low
Pre-calculated illumination maps (Fast in real time)	

Table 3 – Fixed precalculated illumination maps

The consequences made us think through our possibilities and we realized that with our demands on a dynamic system, in some way we had to capture our information on the fly. The solution for this was a compromise. Since we knew where the marker was in respect to the camera we could

attach a mirrored ball to our marker (figure 60). This would give us the possibility to find our sphere at any time and wherever the marker was in the world. The information obtained from the sphere would always be accurate for the current position of the object since our objects are always on our marker.

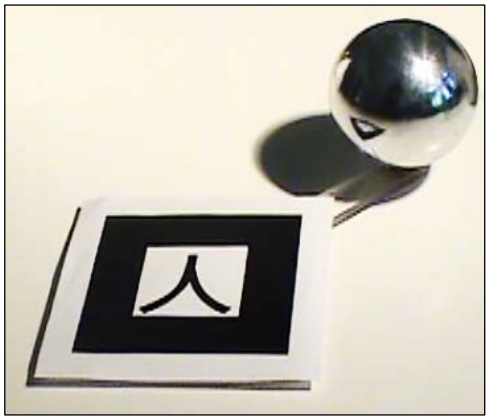


Figure 60 – The mirror ball attached to our marker

Table 4 – Illumination on the fly

Capture illumination information on the fly	
Advantages	**Disadvantages**
Dynamic to illumination alterations	Quality (Dependent on capture device and distance to camera)
Can move objects in world without any reconstruction of scene geometry	HDR images not possible (With web cam)
	Illumination maps calculated in real time (Slow)
	Requires special marker (Attached sphere)
	Marker reflected in sphere

Figure 61 – The quality of the sphere when positioned far away from the camera.

We understood that doing this would create a more dynamic system but also give us a great loss of quality and stability in the light capture. The resolution of the camera we had during our projects was only VGA. If we looked at the marker from a distant the size of our sphere in that image could be very small, for example 64*64 pixels (figure 61). A small sphere meant less data to do our analyses on and that would affect our result. We also got a reflection in our sphere from the attached marker that would create an inaccuracy in the resulting illumination map. Also, AR toolkit isn't totally stable and gave us certain fluctuations that meant it was hard to extract the image of the sphere accurately enough for some of our operations. These problems made us look into another solution as well. We experimented with a second camera that we also attached to our marker, facing the sphere. By analyzing the input from this second camera we could get a much higher resolution of our sphere and the stability problem was solved as well. But this led to new problems. When placing a camera, which in our case was rather big, so close to our sphere the camera was clearly visible in the reflection and the result were clearly affected. Since our camera was rather clumsy the marker now became awkward and hard to move since our second camera wasn't wireless. A better solution here would be to attach a dedicated wireless environment camera. This was equipment that we didn't have so we continued with our single camera setup and tried to deal with the existing problems.

3.1.2. Extracting Probe from Video Frame

To be able to extract the illumination information from our probe we had to be able to find it in every image frame. We first thought of making an image based solution where we would scan the image after spherical objects and decide which region to use. We realized that this would result in to much work for a simple problem. Actually this could be done fairly simple since we knew where our marker was in respect to our camera. This information is given to us by AR Toolkit. Since the probe was fixed in regard to the marker we could calculate its position in respect to the camera as well. When that information was known we could project its 3D positions to the image plane and decide where to find it in our 2D image as well.

To do this we first had to find the centre of the sphere in the image plane and then one of the edge pixels. This was all we needed to "crop" the sphere from the rest of the image.

We had to work in three coordinate spaces to get the desired result. We had the marker space where origo (0,0,0) is in the middle of the marker. We also introduced the sphere space where we defined origo in the centre of the sphere. To find the 2D coordinates we also had to use the camera projection matrix to project our 3D position to our 2D image.

To find the centre of our marker coordinate system in our image frame we could simply use the cameraParametersMatrix and markerTransformMatrix given by AR Toolkit.

In the marker coordinate system the centre is simply (0, 0, 0) and to obtain its position in our image (x, y) we calculate.

Image(x, y, z) = cameraIntrinsicParameters* cameraTransformationMatrix * (0, 0, 0)

Image (x,y,z) =

$$\begin{pmatrix} fs_x & 0 & u_0 & 0 \\ 0 & fsy & v0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} *(0,0,0)$$

Here our point in marker space is transformed to camera space and mapped to the image plane by our matrixes. From this we obtained a point in the image (x, y, z). Since our image is 2 dimensional we had to project the given point to 2D for a usable result. This was done by dividing x and y with z.

So our final positions in the image were:
x = x/z
y = y/z

Since the sphere was positioned somewhere close to our marked we created another coordinate system for our sphere. This system called sphereCoordinateSystem was defined with origo in the centre of the sphere. To be able to transform between the different systems we needed to construct a transformation matrix between the marker and sphere spaces. A sphereTransformationMatrix for this system was found by simply measuring in the real world where and how far from the marker the centre of the sphere was positioned. The matrix was represented with a translation matrix constructed with the measured data (S_x, S_y, S_z).

$$\text{sphereTransformationMatrix} = \begin{pmatrix} 1 & 0 & 0 & S_x \\ 0 & 1 & 0 & S_y \\ 0 & 0 & 1 & S_z \end{pmatrix}$$

The centre of the sphere in the image was simply found by finding the centre of the sphereCoordinateSystem(SC). The difference now was only that we initially had to translate the point from the SC to markerCoordinateSystem. This was calculated like this:

Centre(x, y, z) =
cameraIntrinsicParameters* cameraTransformationMatrix *
sphereTransfromationMatrix* (0, 0, 0)

Centre (x, y, z)

$$= \begin{pmatrix} fs_x & 0 & u_0 & 0 \\ 0 & fsy & v0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & S_x \\ 0 & 1 & 0 & S_y \\ 0 & 0 & 1 & S_z \end{pmatrix} * (0,0,0)$$

With this information we obtained the centre of the sphere in our image. We also needed to know how big the sphere was in the image to be able to analyze it. One way to decide this was to calculate the position of one point on the edge of the sphere. In the case with the centre point we didn't have to take into account from which angle we were looking at the sphere. Analyzing a point on the edge required us to first ensure that the point was really on the edge for the current view angle.

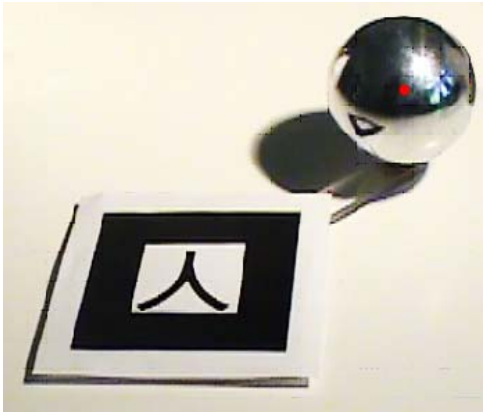


Figure 62 – Center of sphere found

To make sure of this we created an imaginary billboard plane which normal always pointed towards the eye. This billboard was initially positioned in SC with its normal pointing at negative z.

Since we knew the real radius of our sphere we could place a point in SC, radius distance from origo along our x axis. The coordinate became (radius, 0, 0). Before we could calculate where this point could be found in the image we hade to transform it with our billboard transformation. Doing this assured us that the point was positioned on the current edge and nowhere else. The billboardTransformationMatrix should rotate our plane to always point at the camera. This matrix could be obtained by inverting the matrix we had to transform from our marker to the camera space but ignoring the translation. In this way our plane will be pointing to our camera but still be in its original position.

Edge(x, y, z) =
cameraIntrinsicParameters* cameraTransformationMatrix *
sphereTransfromationMatrix* billboardTransformationMatrix*(radius,0,0)

Edge (x, y, z)=

$$\begin{pmatrix} fs_x & 0 & u_0 & 0 \\ 0 & fs_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} *$$

$$\begin{pmatrix} 1 & 0 & 0 & S_x \\ 0 & 1 & 0 & S_y \\ 0 & 0 & 1 & S_z \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \end{pmatrix}^{-1} * (\text{radius}, 0, 0)$$

We now had the position for our centre and one position for the edge. This information was all we needed to be able to extract the sphere from our image.

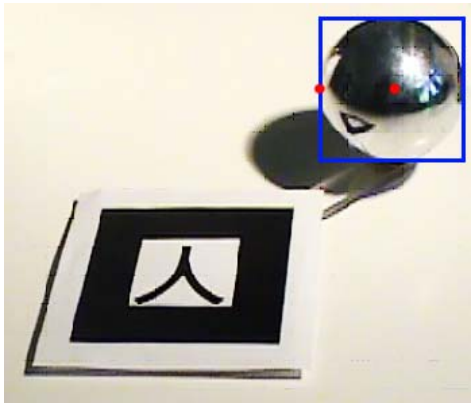


Figure 63 - Centre and edge of sphere found
.

3.1.3. Mapping to Cube Map

In an early stage of our project we decided to work with cubes instead of spheres in our environmental mapping.

The reason for this was:

- Hardware supported via CG
- Viewpoint independent
- Possible to move objects inside cube world without strange artifacts
- More frequently used and therefore more suitable for comparison and evaluation with other methods.

Initially we used the OpenGL extension for sphere mapping which was straight forward since our captured image of the mirror ball in reality was a perfect sphere map. The only thing we had to do was to rotate the texture in correspondent to camera position relative to the marker. Unfortunately sphere mapping wasn't directly supported by CG and to mix mapping in OpenGL and CG was hard and unstable. Since we decided to go for cube mapping we had to find a way to transform our captured sphere map to a cube map. In order to map our captured sphere image to a cube map we though of several methods.

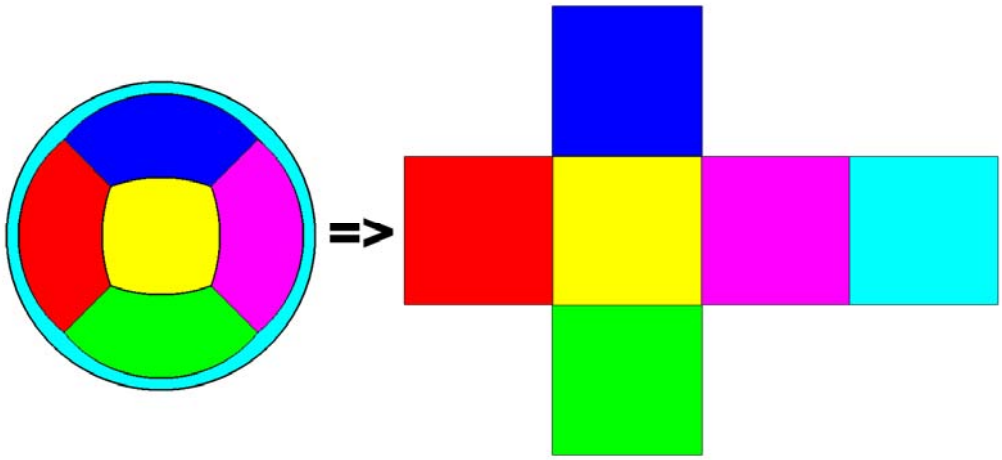


Figure 64 – Mapping sphere map to a cubical representation

First ideas:

- Image warping
- Look up table
- Rendering 6 views from the inside of a sphere mapped box, in 3D space on the fly.

Solution

After analyzing the suggested solutions we realized that all of them would give us a certain amount of error and probably even other problems like aliasing. But we recognized that we actually had all the information we needed to calculate an exact geometrical solution. So instead we solved the problem like this:

For every pixel position on the cube map (i,j) we wanted to know its origin in the sphere image (i´,j´). This could be found by calculating the normal vector n, positioned between the reflection vector r and the eye vector v. The eye vector was simply a fixed vector always directed straight through the front face of our cube and thereby along the z-axis (0, 0, 1).

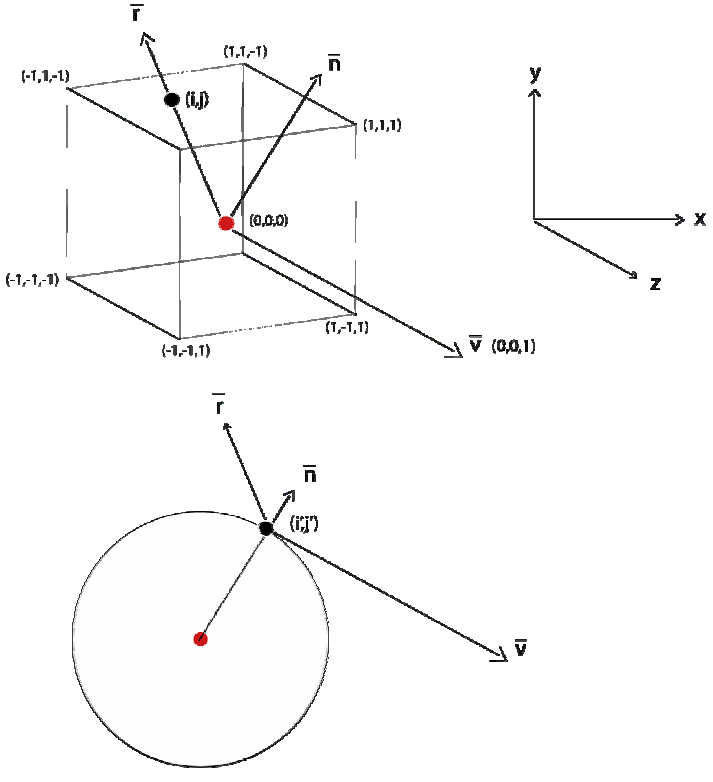


Figure 65 – The reflecting vector (r) is known and used to find the normal vector (n) in our transformation method

Figure 65 shows how these vectors can be though of in cube space and in sphere space.
The reflection vector was the vector defining our current pixel on the cube face (i,j). Since we needed 3 coordinates to define a vector in a 3D space we also had to consider a third coordinate. By taking into account that we knew on which of the faces the current pixel was located we could easily fixate this third component.

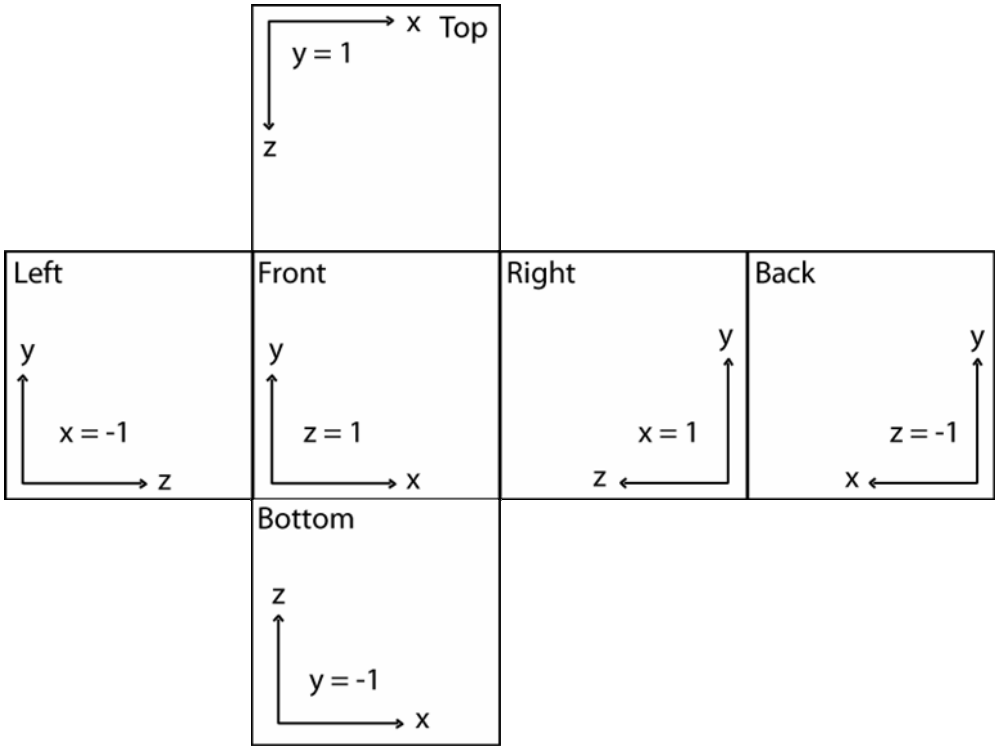


Figure 66 – The different aligned coordinate systems depending on cube face

On each face one vector value is constant through all pixels. For example on the front face the z value is always 1 and for the back face it is always the inverse -1. Depending on the alignment of the faces we fixated different axis and direction (1,-1). (figure 66)(table 5)

Face	Fixed value
Front	Z=1
Back	Z=-1
Left	X=-1
Right	X=1
Top	Y=1
Bottom	Y=-1

Table 5 – Fixed values on different cube faces

When calculating the vectors on the front face we always obtained results like (i,j,1).When vector **r** and **v** were known, vector **n** could easily be calculated**.** All we had to do was adding the vectors together (**r**+**v**). This gave us the normal vector (i´,j´, k´). K was eliminated to obtain the position in the 2D image. With the result we had obtained our (i´,j´) and could easily grab this information from our sphere image. Our method also required that we converted the input positions to the range [-1<=i,j<=1]. Since the size of the sphere image was different depending on where the sphere was captured by the camera.

The algorithm:

For every pixel in all six cube faces

- Resolve current cube face
- Resolve fixed axis and value for given face
- Transform the pixel (i,j) to legal limits [-1<=x,y,z<=1]
- Construct and normalize **r** (x,y,z)
- Calculate **n** (x,y,z) + (0,0,1) = (x,y,z+1)
- Normalize **n**
- Calculate (i`,j`) by transforming normalized **n** (x,y) back to
- sphereImage limits [0<=x,y<= sizeof(sphereImage)]
- Grab (i`,j`) value in sphere image and store in current position of the cube map

The resulting textures were bind and stored in a container with 6 different image buffers. This container is an extension to OpenGL called GL_TEXTURE_CUBE_MAP_ARB, where the images could be accessed individually with e.g GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT for the front face of the cube. This was a comfortable way to handle our textures and could easily be accessed from our CG shaders during rendering.

Costs

This calculation had to be done for every pixel in our cube map. So the numbers of iterations were: 6(cubefaces) * $sizeOfCube^2$
During the development we used web cameras with a resolution of 640*480 pixels. The average size of the sphere in the image altered depending on usage but was often not more then 64*64. We decided that the faces of our cube map could be represented with 64*64 pixels without loosing too much resolution in the transform.

3.1.4. Creating the Reflective Map

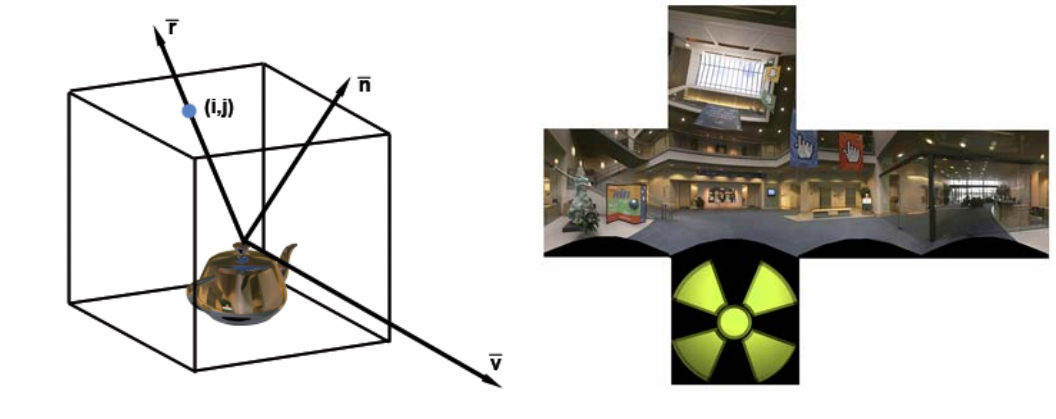


Figure 67 – An object placed in our cube map are mapped by calculating the reflecting vector (r)

To render a reflective object in our world we simply used the cube map exactly as we obtained it from our probe. To find the corresponding pixel position in our cube we calculated the reflective vector (**r)**. To find this vector we needed to know the normal (**n**) of the current vertex and the current viewing vector (**v**).

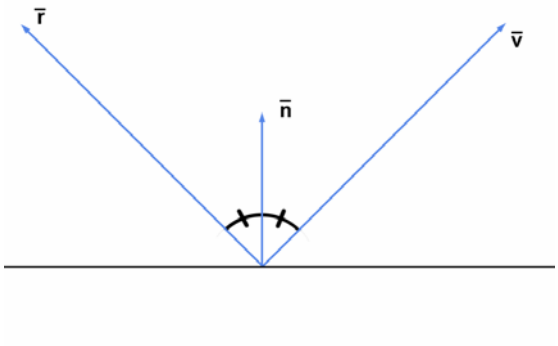


Figure 68 – The reflecting vector is calculated by the known view vector(v) and normal vector(n)

The angle between the reflective vector(**r**) and the normal (**n**) are the same as the angle between view vector(**v**) and (**n)** (simple reflection). The view vector was found by calculating the vector from the marker to the current camera position. The normal vector could be found by analyzing our imported object information. These calculations were all done in our vertex shader and were fast due to hardware supported methods. The hard part was to realize in what space we should operate in. Since all our vectors were defined in model space we had to transform all vectors to eye space. This could be done by multiplying our vectors with the modelView matrix

obtained by AR toolkit's transformation matrix. The reflection mapping ran smooth and the result was very impressing. With any object placed in our world we could alter the light in our room or sweep over our object with a flashlight. The result of the alteration was visible without delay on our synthetic object.

See figure 69 for an example of a reflective object. This worked very well as long as we kept our camera at a fixed position. When we started to move our camera in respect to the marker we got problems with a jittering texture on our objects.

Figure 69 – A teapot mapped using our reflective mapping method

This looked unnatural and the entire immersive feeling disappeared. The jittering was present because of alterations in our dynamically constructed maps. The probe that should represent our environment was extracted from our image with small amount of errors in every frame. The errors occur due to a certain amount of miscalculations done by AR toolkit but also because of rounding problems in our method to find the sphere. If we found the centre of our sphere a pixel away from the frame before we ended up with a whole new environment map. This was clearly visible and resulted in jittering. We tried to solve this by different types of filtering of the positional data. But to heavy filtering led to noticeable errors when movements were high and to light filtering still gave us jittering. This was the reason for us to explore the possibility of using a second fixed camera that was dedicated to capture only the sphere information. That solution terminated the jittering but as mentioned made our system to awkward. Instead we realized that we would have to restrict our system and make it less dynamic. We introduced a set of rules for our system that could control when to update our environment maps.

By looking at the position of our sphere in the world we could decide how much movement that had occurred since the previous frames. With this

knowledge we decided to only update our maps when we had a stable scene, meaning that the camera was fairly still. This decision resulted in that our system was only fully dynamic in a static situation and during movement we used the information obtained from the last known steady frame. This led to some differences in our system. We needed to store environmental data from our last accepted frame, and we also needed to store the transformation matrixes from that frame. Since our object had to be related to our old cube map transformation we needed to keep that fixed and only update the altering view vector for a correct result.

3.1.5. Creating the Diffuse Map

Illuminating our diffuse objects was way more complicated than creating the illusion of a perfect reflective material. In earlier work Gibson [Gibson04] calculated a irradiance volume and used spherical harmonics to reduce computational requirements when calculating the diffuse component. These volumes required advanced pre-computed calculations but gave a stunning result. We needed something simpler but still with a compelling result. In a perfect solution all vertexes on our object should be affected by all the lights in the environment that were in the visible hemisphere. The diffuseness of the vertexes should also be differently affected by the environment depending on the material properties of the object. As mentioned in the background chapter (2.4.2) the diffuse factor depends on the light source angle from the normal vector. The greater the angle the less influence the light has on the result. In our solution we

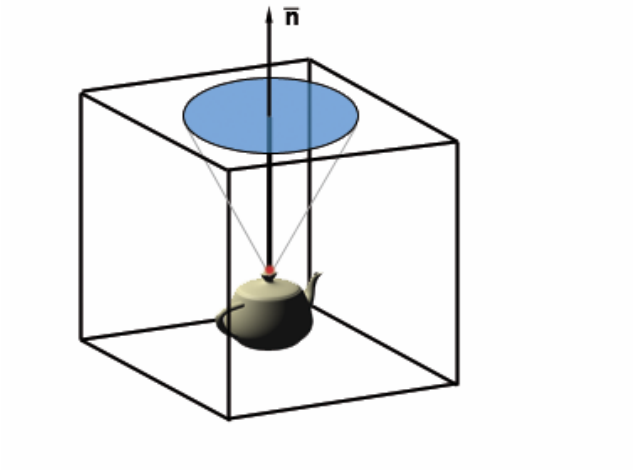


Figure 70 – A teapot lit by our diffuse light model where only the area around the normal is being considered

analyze only parts of our hemisphere over our vertex, the part closest to the vertex normal. This area could be seen as the area of most influence. Even if we ignore the remaining parts of the visible hemisphere the result should be appealing enough in combination with our textures and other lighting effects. We also restricted our calculations to be valid for all materials and they were not material unique in any way.

The main benefit of this method was that it was extremely fast. The trick was that we could store this area of interest in an environment map as well. The way to do that was to blur our given reflective map. This gives the effect that every pixel value in our map corresponds to a certain mean value for its nearby area. This was exactly what we needed. The diffuse light was not independent on viewing direction but only by the normal vector of the vertex. As in the case of reflective mapping we had to convert our normal to the eye space before we calculated our result.

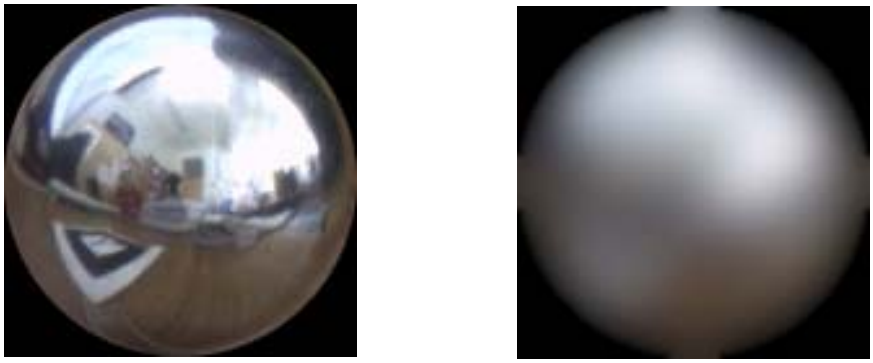


Figure 71 – An example of a blurred reflective map

As mentioned this was not in any way the true diffuse solution. It was a simple cheat but during our test phase none of our users seemed to complain about the visual realism. See figure 72 fore some of the results with different light setups.

Figure 72 – Different results of our diffuse light model with different light conditions

We got this method running well in real time but with the same jittering problems as we had with the reflections. We tested our system by alternating the light conditions in our room and by covering parts of the hemisphere. The result was very appealing and for almost no computational cost at all. The restrictions were of course present, and we could for example not at all make one material more or less *diffuse* than another, since they all used the same calculated map.

3.1.6. Estimating Light Positions

The next step in our pipeline was to analyze possible light sources for our shadow casting. This was a step that we couldn't really find a blameless solution for. We suffered much from the minimal resolution low dynamic range picture that we had of our sphere. In certain light conditions with a controlled light source it could be a fairly easy task to determine the source's position. In other more general situations it could be a completely impossible task. What we initially wanted from this step in order to be able to construct our shadows were three properties:

- position of the light
- intensity of light
- size of light source (Area)

In a controlled environment with a single point light source we could find its position by analyzing our sphere image and search for the brightest part. The only information we had about the intensity was the R+G+B value of that current pixel. Since the image was taken using only one exposure level we couldn't relate that value to anything else. Almost all light sources were represented with the value 255,255,255 which could mean anything from bright to very bright. This was the part of the project were the lack of a multi exposure image (HDR) was evident. When we tried to analyze the size of the light we tried to measure the size of the lit area around the mass centre of light. This was a hard task and there were no differences between the values around the light source, the values were all often the same 255,255,255. We came to the conclusion that we might have to surrender our first goals of this step. We tried to focus only on the position

Figure 73 – The need for a HDR image was obvious

estimation. Even this was a hard task due to the low dynamic range. As seen in figure 73 it was almost impossible to estimate from where the light was coming.

There were often big white connected areas present on our image, with no valuable information. This was an result of the current exposure.We thought of possible solutions to this. We tried to force our web camera to change exposure level at some chosen frames. That would have given us a better knowledge of the current light. Unfortunately we couldn't find any way to do this dynamically. We also thought of attaching a second sphere to our marker that we would paint black. That would actually have meant the same thing as altering the exposure level of the camera. Only the light sources would have been visibly in our black sphere. The big drawback with this method was that our marker would have become more awkward.

In our solution we made a low detail picture of our sphere to restrict the possible light positions and decrease jittering of positions. This information was then scanned for the brightest part (R+G+B) and that area was decided as a light source if it exceeded a certain threshold value. When this was done we had to calculate the corresponding light vector for that pixel. Since we were only interested in casting shadows on the ground floor we only searched the upper part of our sphere to speed up the process. To find our vector we needed to calculate the reverse mapping from our texture coordinates to our reflected vector. This mapping could be mathematically calculated the following way:

$$R_x = -\left(2\sqrt{\left(-4i^2+4i-1-4j^2+4j\right)}*\left(2j-1\right)\right)$$
$$R_y = 2\sqrt{\left(-4i^2+4i-1-4j^2+4j\right)}*\left(2i-1\right)$$
$$R_z = -8*i^2+8i-8j^2+8j-3$$

The calculation maps the current pixel in 2D to a vector in 3D pointing at our light source. The jittering problems with the environment maps were also present in this step. Our program could calculate the vector in every frame but we decided to do it as a calibration step by the user to avoid the flickering. The system only updated the light source position on demand from the user. This terminated the problem with the jittering and also gave us the possibility to calibrate our system at will, meaning that we could position the camera close to the sphere for a higher resolution.

3.2. Shadows

In the beginning of the project much time was invested in trying to find a Global illumination (GI) shadowing algorithm that directly could use the environment map that we captured with the system. After a lot of reading we came to the conclusion that this would not be possible for us to do in real time on the hardware we were using. We then spent much time trying to find working implementations of Ambient Occlusion (AO) for OpenGL to se it in action since the method was very interesting in theory. The technique had many of the requirements that we were looking for such as good visual result that should resembled GI but without the extensive cost as well as the ability to pre-process some of the calculations for the objects and the fact that the result could easily be combined with the diffuse lightning from an environment map. However the few demos we managed

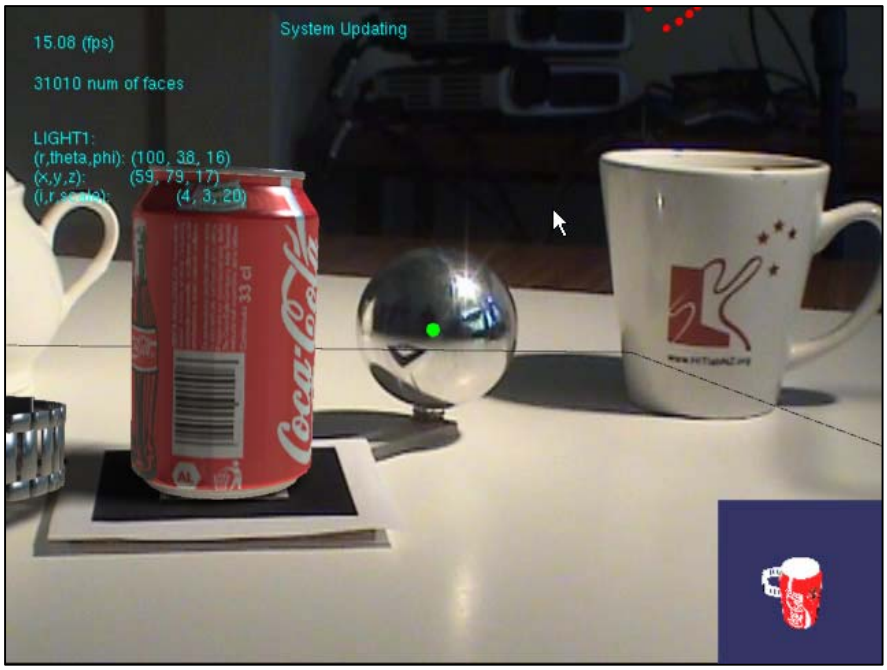


Figure 74 – Augmented reality scene with shadows

to compile were not satisfactory. The implementations we tried were very slow and the visual results not as convincing as we had hoped for. This combined with the fact that we realized that this technique would not take care of all our shadow problems, but had to be combined with other techniques for giving shadows that were responsive to their environment made us rethink our strategies. Finally we decided that the existing implementations of AO were not impressive enough for us to build on or to motivate us to write a new implementation ourselves. We then looked at a few different types of soft shadow map implementations such as

smoothies but decided together with our supervisor that maybe these state of the art methods would be too hard to write ourselves given our current experience in C++, OpenGl and shader languages combined with the limited documentation on how to actually implement these techniques. We instead decided to start with a simple, fast and straightforward method that we could refine over the time of the project. It was very important for us that the shadows were as realistic as possible since they would coexist with real shadows in the scenes. It soon also became very obvious that it was the ground shadows, or the contact shadows, of the objects that would be most visually important to get realism. Soft self-shadowing would be a nice asset but not that important for the realism when combined with material properties such as diffuse lightning, textures and specular highlighting. We therefore decided that we should concentrate on soft projective shadows and use normal hard shadow maps for the self-shadowing. A projective shadow technique was decided on for the floor that was similar to the fast soft shadows approach by Michael Herf and Paul S. Heckbert [Herf97] described in the shadow chapter (2.6.2). The big advantages being that the approach was easy and that the result nicely converged to a realistic rendering when the number of samples was increased. A very important step in the implementation of our projective shadow algorithm was the extension of a real time variable blur to the shadow texture. This made it possible to converge to a realistic looking soft shadow already after 25 samples thereby making a big increase in performance. It also meant new parameters to make the shadows look softer depending on the size of the area lights and the ambient lighting in the room. Towards the end of the implementation phase we also managed to extend our hard self-shadow algorithm to make soft self-shadows by adding a blur step in screen space to the implementation using the techniques we already had developed for the ground shadows.

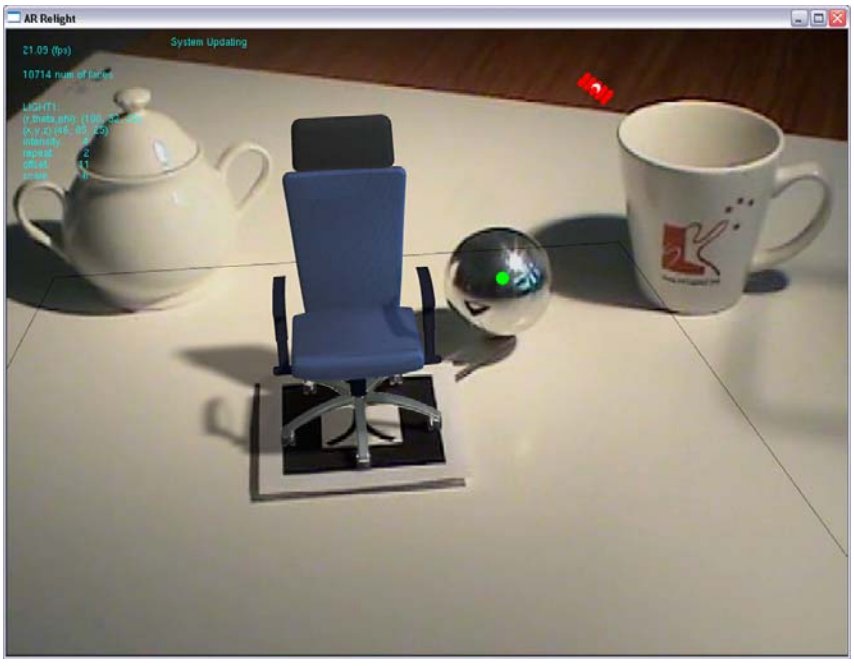


Figure 75 – Synthetic chair object showing cast and self shadows

A big problem with ARToolkit applications is that the point of reference is always the marker and there is no easy way to define a global one. If for example one would decide to move the marker in the real world the reference system would change. This combined with the desired handling of fast changes to the lighting conditions were the main reason why the shadows needed to be updated in real time. During the entire project however we had big problem with the instable data we got from ARToolkit which lead to flickering problems with the whole system. This was a big problem and limitation during the project that was finally solved by some simplifications and clever mathematics. More information can be found in the light probe section of this thesis. The solution that we decided on was to not do the sampling for every frame anymore but instead only as a calibration step that could be triggered by the user when needed. That being either when the lighting conditions had changed or when the marker had been moved. Of course this totally changed the prerequisite for the shadow calculations in a very positive way. Given that the focus of the project only was on rigid objects, i.e. objects without animation, all the shadow calculations didn't have to be in real time anymore since some steps could have been pre-computed into textures or other data structures. This opened up several completely new possibilities like real global illumination solutions using radiosity but unfortunately there were no time to make use of this possibility during the implementation of the project. However what could still be done was to increase the resolution on some of the already implemented techniques which made the result look better without any big sacrifices in performance.

3.2.1. Projective shadows mathematics

A projective shadow technique was used for generating the ground shadows. An introduction to the technique can be found in the shadow section of the background chapter (2.6.1). The objects geometry is projected on to a flat surface by a specific projection matrix. The projection can be described by a 4x4 transformation matrix in homogenous coordinates.

The mathematics for deriving the projection matrix is fairly straight forward.

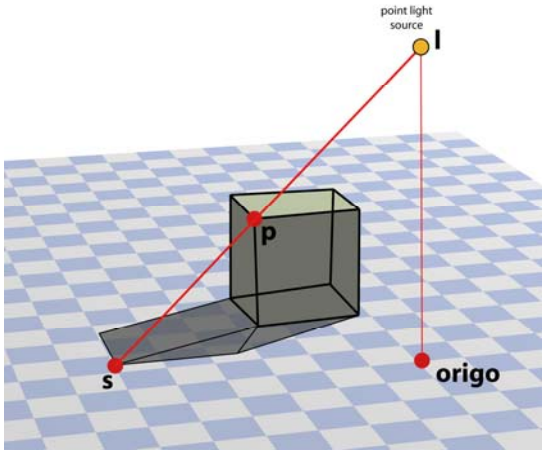


Figure 76 – Projection of point p to point s given the light source position l

Consider the situation in figure 76, where the light source is located at l = (l_x, l_y, l_z) and the receiving plane has the equation y = 0.

A point p = (p_x, p_y, p_z) on the occluder object, seen from light source l, will cast shadow on the ground plane at s = (s_x, 0, s_z) Point s can be expressed as

$$\mathbf{s} = \mathbf{l} + t(\mathbf{p} - \mathbf{l}) \qquad (1)$$

Solving for t with the requirement that $s_y = 0$ gives

$$t = \frac{l_y}{l_y - p_y}$$

Inserting into (1) we gain coordinates of s as a linear transformation:

$$\begin{pmatrix} s_x \\ 0 \\ s_z \\ 1 \end{pmatrix} = \begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix} \times \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

In the general case, it is not always true that the receiver polygon lie in the plane y = 0. For an arbitrary receiver plane P: nx + d = 0, where n is the plane's normal vector and x is a point on the plane, we can construct the transformation matrix M in the same way. If we combine equation (1) with the plane equation and set x = s, we obtain:

$$t = \frac{\mathbf{nl} + d}{\mathbf{n} \cdot (\mathbf{l} - \mathbf{p})}$$

Using this we can derive transformation matrix M, which satisfies s = Mp.

$$\mathbf{M} = \begin{bmatrix} \mathbf{nl} + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & \mathbf{nl} + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & \mathbf{nl} + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & \mathbf{nl} \end{bmatrix}$$

This shadow projection matrix is then used to calculate the projected shadow for each jittered light source.

3.2.2. Projective shadows implementation

Instead of point light sources, area light sources were used in our implementation and every area light consisted of a grid of jittered lights. Several different area lights could be handled by the application. The jittered grid was made by first constructing a perfect grid and then randomly position one point in every grid cell. By doing this the light positions and thereby the resulting shadows became more randomly distributed.

In order to better map to the shape and position of the light patches found on the captured environment sphere, the grid was constructed on a hemisphere surrounding the objects using spherical coordinates. The area lights could then easily be created by passing parameters for center point, area and grid size. All lighting calculations were done using a spherical coordinate system.

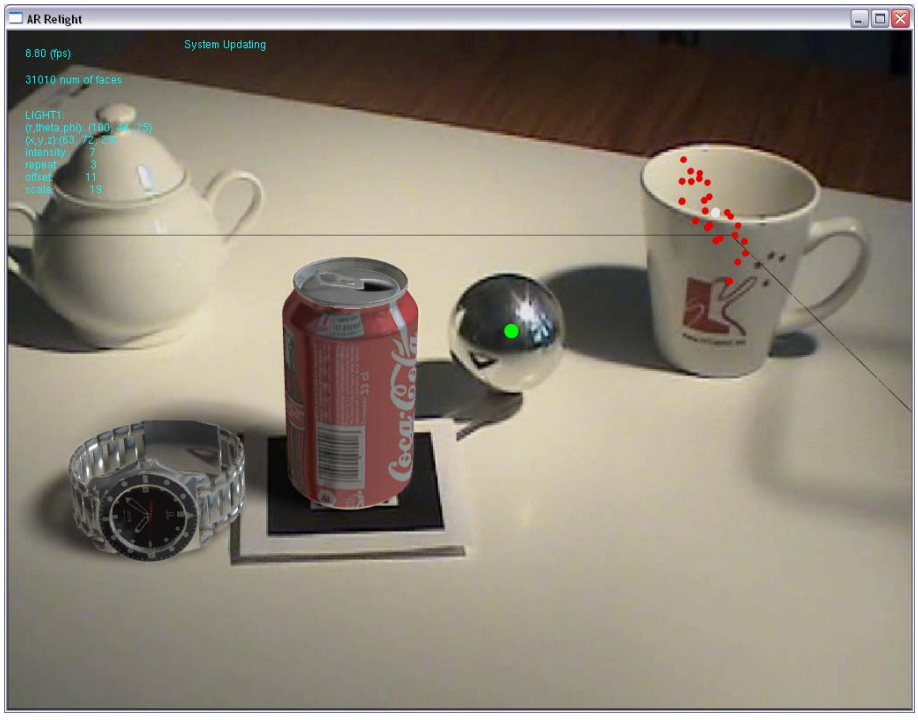


Figure 77 -The red dots represents the jittered light sources

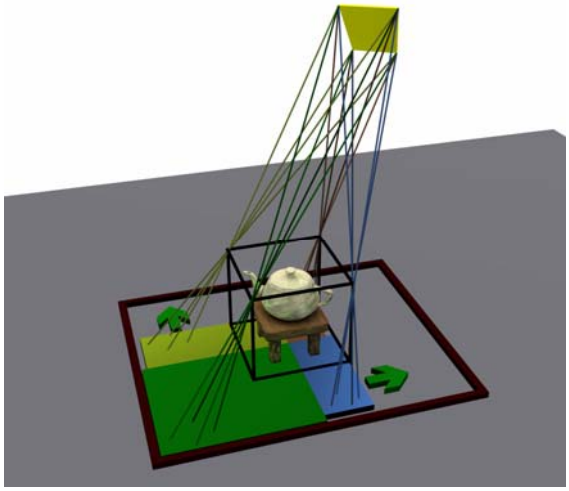


Figure 78 – Solution to the problem that an object could be projected outside the camera

The ground shadows were only calculated when needed and then rendered to a texture that was mapped to the floor. This was very important since it meant that we didn't have to update the ground shadows in every frame but only when the lighting conditions in the environment had changed. The rest of the time we only had to read back the shadow from a texture. One texture per area light was rendered. The production of the texture was done by rendering the object from a top down view thereby directly solving some of the problems associated with projective shadows such as anti-shadows and fake shadows since the location of the rendered objects compared to the ground floor was always known. This made it possible to write the implementation without using a stencil buffer thereby simplifying as well as speeding up the algorithm. However the problem that an object could be projected outside the receiving targets, in this case the camera view, was still very evident. The solution was to create a bounding box for every object and then project the corners from this bounding box for the extreme points on every main light on the floor, making sure that the projected geometry was in the camera view. If not, the camera was moved backwards to zoom out on the texture se figure 78.

The procedure for rendering the final floor textures was as follows with pseudo code:

1. The method drawOnMarker() was ran once per frame and it triggered all the rendering of the objects geometry as well as their shadows. redrawShadows() was where the actual calculations of the projective shadow were done and it was only run if it had not been initialized. renderShadowGroundPlane() only rendered the floor polygons with the texturing and was run in every frame.

```
void drawOnMarker() //Run for every frame
{
        …
        if (!shadowInitialized)
        {
                redrawShadows();
                shadowInitialized = true;
                …
        }
        renderShadowGroundPlane(…);
        …
}
```

2. The redrawShadows() method initiated the rendering of the shadows and the blurring. When that was done the shadows from the different area lights were composed together.

3. As described earlier, our method rendered the projective shadows to an accumulation buffer. This was done for every jittered light in every area light source.

```
void renderJitteredShadows()
{
        …
        for( each arealight )
        {
                setViewPortAndPerspective();


                for( each jitterLight )
                {
                        setupAccumulationBuffer( 1.0
/numOfJitteredLights);
        projectToGround(areaLight, jitterLight);
                }

                copyResultToTexture();
        }
        cleanup();

}
```

After the ground shadows were rendered to textures, a post-processing step was added were the textures were blurred before they were combined and applied to the ground floor. More information on how the blurring was done can be found in the section on blurring. The blurring made it possible to greatly reduce the number of jittered lights needed for each area light.

3.2.3. Self-shadowing

As mentioned in the introduction to this chapter the shadow map technique was used for adding the self-shadowing to the objects. How the shadow map method works can be found in the shadows section of the background chapter. It was hard to find example code and documentation for a usable implementation that worked well together with Cg but we finally succeeded to reuse much of the example code provided from a NVIDIA developer demo. Our implementation used vertex shaders as well as hardware depth buffers in 32-bit in order to increase performance and the visual result. One shadow for each main light was calculated. In order to make the shadows look soft and thereby improve the visual quality a similar blurring technique as the one used for the ground shadows was introduced. By only rendering the self-shadows to a texture in screen coordinates for each main light, these textures could then be blurred and superimposed on the original rendering, without self-shadows. Since the blurring was done in 2D screen coordinates the soft shadows were not physically accurate but they still looked convincing enough to trick the common eye into thinking they were correct. More importantly they better matched the soft ground shadows and thereby didn't make the self-shadows standout from the rest of the composition. By using the same C++ class for handling the blurring of both types of shadows it also made it easier to map the parameters for the shadows depending on the main light properties such as size and intensity. Since the shadow mapping technique is view dependent the generation of the textures and the blurring step had to be done for every frame compared to the ground shadows were this was not necessary.

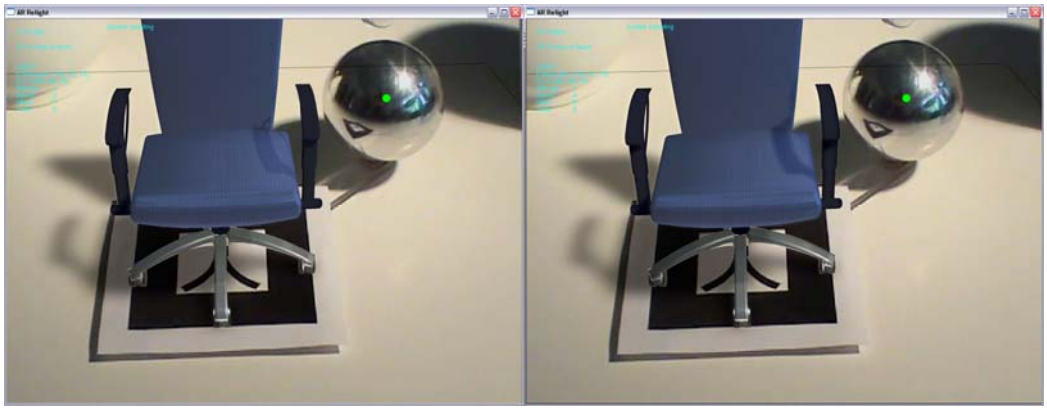


Figure 79 – Left picture displays a rendering with hard self-shadows and the right one a rendering with soft self-shadows

The procedure for rendering and composing the self-shadows using our shadow mapping technique was as follows with pseudo code:

1. The drawOnMarker() method was ran once per frame and it triggered all the rendering of the objects geometry as well as their shadows. Here the renderShadowGroundPlane() method was called and it rendered the ground shadow with a pre-rendered texture. The renderObjectWithShader() then rendered the complete objects with shading. And last the superImposeBlurredShadow() added the blurred self-shadow texture on top of the rendering in every frame.

```
void drawOnMarker() //Run for every frame
{
        …
        renderSceneFromLight()
        computeObjectSelfShadows();
        …
        renderShadowGroundPlane(…);
        renderObjectWithShader();
        superImposeBlurredShadow();

        …
}
```

2. The renderSceneFromLight () method was where the depth texture for the light was created. The view of the scene seen from the light was rendered to a depth texture in 32bit.

```
void renderSceneFromLight()
{
        …
        //Activate rendering to shadow map
        shadowMapBuffer->activate();

           //Set size of depth map
            glViewport(0,0,SMAP_SIZE, SMAP_SIZE);

           //Set camera to look at sceen from lightpos
           gluLookAt(getLightPos, boundingBox, …);

           renderObject();

        //Deactivate rendering to shadow map
        shadowMapBuffer->deactivate();
        …

}
```

3. The computeObjectSelfShadows() method was where the actual shadow map rendering logic was handled. The light depth texture was compared to the camera view in a Cg fragment shader to decide which fragments were in shadow and which were lit. Only the self-shadow pixels were rendered and then copied into a texture for later use.

4. In the last step the superImposeBlurredShadow() method was used to render the generated texture on top of the original rendering thereby superimposing the self-shadows.

```
void superImposeBlurredShadow()
{
        …

        //Set up an ortographic projection
        setOrthoProjection();

        //Enable alpha blending
        glEnable(GL_BLEND);

        //Bind the blurred shadow texture
        glBindTexture(combinedBlurTexture);

        //Draw the texture on a plane on top of everyting
        DrawScreenPlane();

        …
}
```

The result was a complete rendering with good looking self-shadows on the models.

3.2.4. Blurring

The blurring operation could be seen as a low pass filtering of the image and it could be done using several different techniques. We decided to do a real convolution since we knew how to speed up this process and since we knew it would give a correct result. The technique for doing this had to work in real time with good frame rate and with textures of good quality (1024x1024). In order to fulfill these requirements some smart implementation tricks were made and CG vertex and pixel shaders were created. Normally doing a 2D Gaussian blur is very costly since it involves a convolution that is dependent on the size of the blur kernel which makes it a $O(n^2)$ operation. If for example the Gaussian kernel is 10x10 a total of $10x10x1024x1024 \approx 1.1*10^8$ texture lookups have to be done per frame. An important speedup in our algorithm that made it O(2n) was that we used a separable kernel thereby greatly reducing the total cost. The biggest speedup however was due to the fact that the code was run on and optimized for the GPU using parallel calculations.

In order to do the blurring the texture with the hard shadows was put on a ground plane with a camera hovering above, having the texture exactly in view. The scene was then rendered once with the horizontal blur shader overwriting the texture and then once again with the vertical blur shader giving the final result.

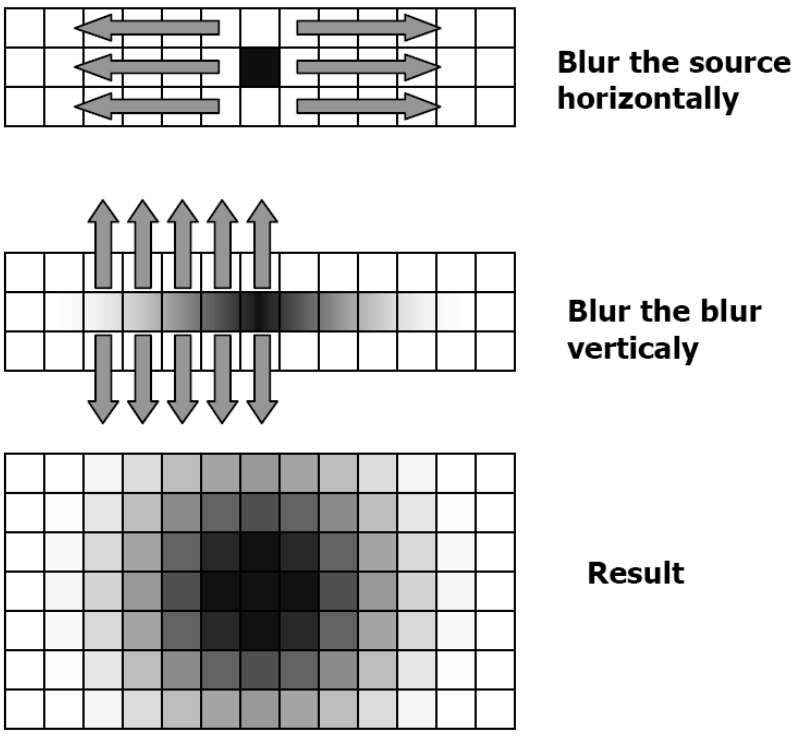


Figure 80 – The blurring operations

3.3. CG Shader - putting it all together

To render the objects with our own illumination shading approach we used a CG shader. In the shader the components for ambient light, diffuse light, specular light and reflectivity were added together as follows.

$$c = A_l \times A_m + K_d \times D_{light\ map} \times D_m + K_s \times S_l \times \max((N \bullet L), 0)^{n} + K_r \times R_{map}$$

A_l – The scene's ambient intensity
A_m – The material's ambient coefficient.

K_d – The coefficient for setting the amount of diffuse light.
$D_{light\ map}$ – The diffuse light intensity from the environment map
D_m – The material's diffuse coefficient.

K_s – The coefficient for setting the amount of specular light.
S_l – The light's specular intensity
N – The normal of the surface
L – The normalized light vector from the point being shaded to the light source.
n – The specular exponent

K_r – The coefficient for setting the amount of diffuse light.
R_{map} – The reflectivity intensity from the environment map

Figure 81 – The final shading result. A virtual chair with diffuse, specular, and reflective properties

The calculations were done on a per pixel basis with Phong interpolated normals meaning that the lighting equation was evaluated for every pixel in the resulting image. In order to correctly render per pixel lighting as well as per pixel diffuse and normal reflections the vertexes normals and positions had to be converted to eye space in the vertex program and feed to the pipeline for interpolation. The Blinn-Phong shading model was used for the specularity in order to get correct looking specular highlights. The diffuse material term D_m could either be taken directly from the object specification or from a texture. As mentioned before, no real self-shadows but only attached shadow calculations were done in the object shader, and they were done because they were a part of the Blinn-Phong model. Real self-shadows were instead later superimposed on top of the rendering.

See Appendix 1 for an overview of our framework.

4. Conclusion and Future Work

4.1. Lessons Learned

The frame rate of the webcams used together with AR Toolkit for OpenGL is a big bottleneck for the video based system. A starting frame rate of 60fps, instead of the more common 30fps used today, would give a great increase of performance for the entire system.

AR toolkit is good for prototyping and research but should maybe not be used for commercial applications, where stability is very important..

The need for real-time lighting updates is often not necessary. It can be enough if the lighting setup can be recaptured automatically by an instruction from the end user.

AR in complex lighting environments with video based tracking is a complicated area because of the bad handling of light in today's cameras. In these areas it is probably better to look at other tracking technique such as GPS and gyros.

C++ programming, debugging and linking is hard and can consume a lot of time in a project. Cg is a very powerful language to work with once you manage to set everything up.

4.2. Future Work

There are several areas for further work in this huge area. Some areas that we would like to further explore are:

- Enhanced cameras, for higher stability
- HDR light capture for better light information
- Dedicated light capturing camera
- Omni directional cameras.
- GI solution for calculating shadows and lighting – real-time update not necessary
- More advanced material properties such as refraction and bump mapping
- Modifications for outdoor usage
- Mobile devices

4.3. Conclusion

We have constructed a framework that can capture the lighting condition in a room and use this information to render synthetic objects in real time. The objects rendered have a realistic lighting that nicely blends with the lighting in the rest of the scene. This works well for indoor scenes with a direct response to changes in lighting condition. We have managed to use the captured lighting information to analyze were the most significant light sources in the room are located and then use this information to generate soft looking shadows. Our implementation handles both soft cast shadows and soft self shadows. The limitation is that this can only be done in very controlled lighting conditions indoors with a few lights. To make the system more general we need to concentrate our efforts on the light finding routines and look at different capturing approaches such as HDR and dedicated cameras. We feel that the addition of shadow information has helped to improve the immersion of the scene as well as the depth cues. However we have not made a full user study of this result but are instead basing this assumption on the feedback given from our test users and colleagues at the lab.

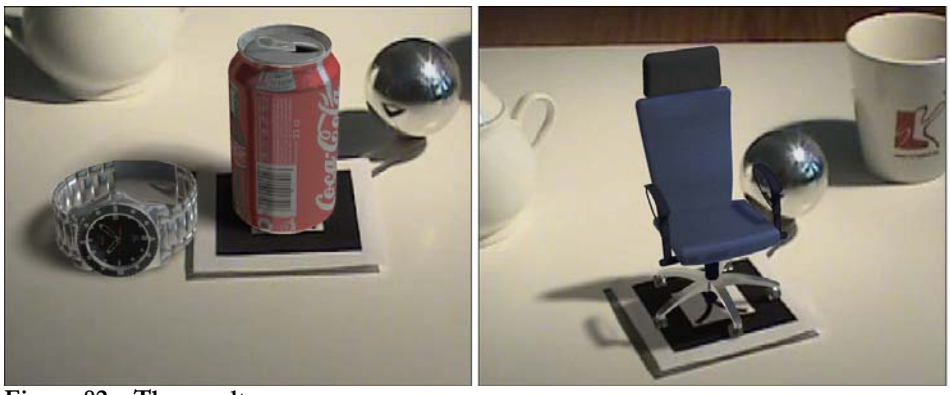


Figure 82 – The result

The handling of the system is very straightforward with most of the functions being automatic; this was also an initial requirement to keep the system as simple as possible for the end-user.

The system is only using standard computer hardware with a webcam to handle camera input. This was an initial requirement to keep the costs of the system down and thereby making it more accessible for end users. This has led to a few problems with quality mainly concerning the captured video image and we are planning to evaluate some mid-end and high-end hardware in the near future. A big problem in the field however is the lack of specific high quality AR hardware on the market, in reasonable price ranges. We are mainly thinking about cameras and HMD:s. This will hopefully change as the technology gets used in other areas and as more AR applications hit the market and drive the research forward.

We have learned a lot about C++, OpenGL, 3D graphics and shader programming that we think we will have a great benefit from in later projects. Computer graphics for real time is a huge, complex and very interesting area that moves more and more towards physical correct models and techniques that could only be done with offline rendering some years ago. We feel that we have gotten a good insight in this area and have great respect for the people working in it. We area also satisfied with the results that we have obtained during our project. The knowledge we have gain in general about AR has helped us a lot in understanding the areas most important strengths and weaknesses and we would like to continue doing projects and learn more about AR and real time graphics in the near future. We have also gained a lot of good contacts and friends, in the new and interesting area of AR, from our visit to HIT Lab NZ.

Bibliography

[Appel68] A. Appel. Some Techniques for Shading Machine Renderings of Solids, volume 32. AFIPS 1968 Spring Joint Computer Conf. 1968.

[Azuma97] R. Azuma, "A Survey of Augmented Reality", Presence: Teleoperators and Virtual Environments. Vol 6, no. 4, Aug. 1997, pp. 355-385.

[Azuma01] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, Blair. MacIntyre, Recent Advancees in Augmented Reality, IEE 2001

[Batagelo99] Harlen Costa Batagelo and Ilaim Costa Júnior. Realtime shadow generation using BSP trees and stencil buffers. In SIBGRAPI, volume 12, pages 93–102, October 1999.

[Billinghurst99] Mark Billinghurst and Hirokazu Kato. ARToolkit. http://www.hitl.washington.edu/artoolkit/, 1999.

[Blinn 76] J.F. Blinn, "Texture and Reflection in Computer Generated Images," Comm. ACM, vol. 19, no. 10, Oct. 1976, pp. 542-547.

[Blinn77] James F.Blinn. Models of light reflection for computer synthesized pictures. Computer Graphics, 11(2):192–198, July 1977.

[Blinn88] J. F. Blinn. Jim Blinn's corner: Me and my (fake) shadow. IEEE Computer Graphics and Applications, 8(1):82–86, Jan. 1988.

[Brabec03] Stefan Brabec and Hans-Peter Seidel. Shadow volumes on programmable graphics hardware. Computer Graphics Forum (Eurographics 2003), 25(3), September 2003.

[Chan03] Eric Chan and Fredo Durand. Rendering fake soft shadows with smoothies. In Rendering Techniques 2003 (14th Eurographics Symposium on Rendering). ACM Press, 2003.

[Crow77] Crow, F. C. 1977. Shadow algorithms for computer graphics. In Computer Graphics (Proceedings of SIGGRAPH 77), vol. 11, 242–248.

[Debevec96] DEBEVEC, P. E., TAYLOR, C. J., AND MALIK, J. Modeling and rendering architecture
fromphotographs: Ahybrid geometry- and image-based approach. In IGGRAPH'96 (August 1996), pp. 11–20.

[Debevec97] P. Debevec, "Recovering High Dynamic Range Radiance Mpas from Photographs",Computer Graphics (Proc. Siggraph 97), AMC Press, New York, 1997, pp. 369-378.

[Debevec98] P. Debevec. "Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-based Graphics with Global Illumination and High Dynamic Range Photography". Computer Graphics (Proceedings of SIGGRAPH 98), 32(4):189– 198, 1998.

[Everitt02] Everitt, C., and Kilgard, M. J. 2002. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. Published online at developer.nvidia.com.

[Fernando01] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg. Adaptive shadow maps. In SIGGRAPH 2001 Conference Proceedings, pages 387.390, 2001.

[Gibson04] S. Gibson, "Illumination Capture and Rendering for Augmented Reality", RFIA 2004

[Goral84] Cindy Goral, Ken Torrance, Modeling the interaction of light between diffuse surfaces. SIGGRAPH 1984 Conference Proceedings, pages 213-222, 1984

[Gouraud 71] H. Gouraud, "Computer Display of Curved Surfaces" Doctoral Thesis.University of Utah. 1971.

[Hanrahan90] Hanrahan and Lawson. A Language for Shading and Lighting Calculations, Computer Graphics 24(4):289-298, August 1990.

[Heckbert96] Paul Heckbert, Michael Herf. Fast Soft Shadows. SIGGRAPH '96 Visual Proceedings, page 145. Aug. 1996.

[Heidmann91] Heidmann, T. 1991. Real shadows real time. IRIS Universe

[Herf97] Paul Heckbert, Michael Herf. Simulating Soft Shadows with Graphics Hardware. CMU-CS-97-104, CS Dept, Carnegie Mellon U., Jan. 1997.

[Kilgard99] Kilgard, M. J. 1999. Improving shadows and reflections via the stencil buffer. Published online at developer.nvidia.com.

[Kilgard01] Kilgard, M. J. 2001. Shadow mapping with today's opengl hardware. published online at developer.nvidia.com.

[McCool99] McCool, M. D. 2000. Shadow volume reconstruction from depth maps. ACM Transactions on Graphics 19, 1 (January), 1–26. ISSN 0730-0301.

[Pharr04] Ambient Occlusion, Matt Pharr, Simon Green, GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison Wesley 2004.

[Phong73] Phong, Bui Tuong: Illumination of Computer-Generated Images, Department of Computer Science, University of Utah, UTEC-CSs-73-129, July 1973

[Pomi04] A. Pomi and P. Slusallek, "Interactive Mixed Reality Rendering in a Distributed Ray Tracing Framework", ISMAR 2004.

[Reeves87] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. In Computer Graphics (SIGGRAPH '87 Proceedings), pages 283–291, July 1987.

[Ropinski04] T. Ropinski, S. Wachenfeld and K. Hinrichs, Virtual Reflections for Augmented Reality Environments, ICAT 2004

[Stamminger02] Marc Stamminger and George Drettakis. Perspective shadow maps. ACM Transactions on Graphics (SIGGRAPH 2002), 21(3):557–562, 2002.

[Tadamura01] K. Tadamura, X. Qin, G. Jiao, and E. Nakamae. Rendering optimal solar shadows with plural sunlight depth buffers. The Visual Computer, 17(2):76.90, 2001.

[Vial03] Florent Vial "Natural Point Feature Tracking of a textured plane: A Real Time Augmented Reality Application", Human Interface Laboratory New Zealand

[Watt99] Allan. H. Watt, 3D Computer Graphics (3rd Edition),1999

[Williams78] Lance Williams. Casting curved shadows on curved surfaces. Computer Graphics (SIGGRAPH 1978), 12(3):270–274, 1978.

[Woo99] M. Woo, J. Neider, T. Davis, and D. Shreiner. OpenGL Programming Guide, Third Edition. Addison-Wesley, 1999.

Appendix 1 - Conceptual Class diagram